Tuple Interpretations for Higher-Order Complexity

Cynthia Kop
Department of Software Science, Radboud University Nijmegen, The Netherlands

Deivid Vale
Department of Software Science, Radboud University Nijmegen, The Netherlands

Abstract

We develop a class of algebraic interpretations for many-sorted and higher-order term rewriting systems that takes type information into account. Specifically, base-type terms are mapped to *tuples* of natural numbers and higher-order terms to functions between those tuples. Tuples may carry information relevant to the type; for instance, a term of type nat may be associated to a pair $\langle \mathsf{cost}, \mathsf{size} \rangle$ representing its evaluation cost and size. This class of interpretations results in a more fine-grained notion of complexity than runtime or derivational complexity, which makes it particularly useful to obtain complexity bounds for higher-order rewriting systems.

We show that rewriting systems compatible with tuple interpretations admit finite bounds on derivation height. Furthermore, we demonstrate how to mechanically construct tuple interpretations and how to orient β and η reductions within our technique. Finally, we relate our method to runtime complexity and prove that specific interpretation shapes imply certain runtime complexity bounds.

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

Keywords and phrases Complexity, higher-order term rewriting, many-sorted term rewriting, polynomial interpretations, weakly monotonic algebras

Digital Object Identifier 10.4230/LIPIcs.FSCD.2021.31

Related Version An extended appendix with full proofs and additional examples is available at [32].
Extended Version: https://arxiv.org/abs/2105.01112

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

Term rewriting systems (TRSs) are a conceptually simple but powerful computational model. It is simple because computation is modelled straightforwardly by step-by-step applications of transformation rules. It is powerful in the sense that any algorithm can be expressed in it (Turing Completeness). These characteristics make TRSs a formalism well-suited as an abstract analysis language, for instance to study properties of functional programs. We can then define specific analysis techniques for each property of interest.

One such property is *complexity*. The study of complexity has long been a topic of interest in term rewriting [11, 27, 25, 7, 24, 35], as it both holds relations to computational complexity [3, 11, 12] and resource analysis [6, 13] and is highly challenging. Most commonly studied are the notions of runtime and derivational complexity, which capture the number of steps that may be taken when starting with terms of a given size and shape. In essence, this is a form of resource analysis which abstracts away from the true machine cost of reduction in a rewriting engine but still has a close relation to it [8, 18, 1, 12].

These notions do not obviously extend to the *higher-order* setting, however. In higher-order term rewriting, a term may represent a function; yet, the size of a function does not tell us much about its behaviour. Rather, properties such as "the function is size-increasing" may be more relevant. Clearly a more sophisticated complexity notion is needed.

In this paper we will propose a new method to analyse many-sorted and higher-order term rewriting systems, which can be used as a foundation to obtain a variety of complexity results. This method is based on *interpretations* in a monotonic algebra as also used for termination analysis [39, 22], where a term of function type is mapped to a monotonic function. Unlike [39, 22], we map a term of base type not to an integer, but rather to a vector of integers describing different values of interest in the term. This will allow us to reason separately about – for instance – the length of a list and the size of its greatest element, and to describe the behaviour of a term of function type in a fine-grained way.

This method is also relevant for termination analysis, since we essentially generalise and extend *matrix interpretations* [35] to higher-order rewriting. In addition, the technique may add some power to the arsenal of a complexity or termination analysis tool for first-order term rewriting; in particular *many-sorted* term rewriting due to the way we use type information.

A note on terminology. We use the word "complexity" as it is commonly used in term rewriting: a worst-case measure of the number of steps in a reduction. In this paper we do not address the question of true resource use or connections to computational complexity. In particular, we do not address the true cost of beta-reduction. This is left to future work.

Outline of the paper. We will start by recalling the definition of and fixing notation for many-sorted and higher-order term rewriting (§2). Then, we will define tuple interpretations for many-sorted first-order rewriting to explore the idea (§3), discuss our primary objective of *higher-order* tuple interpretations (§4), and relate our method to runtime complexity (§5). Finally, we will discuss related work (§6) and end with conclusions and future work (§7).

2 Preliminaries

We assume the reader is familiar with first-order term rewriting and λ-calculus. In this section, we fix notation and discuss the higher-order rewriting format used in the paper.

2.1 First-Order Many-Sorted Rewriting

Many-sorted term rewriting [38] is in principle the same as first-order term rewriting. The only difference is that we impose a sort system and limit interest to well-sorted terms.

Formally, we assume given a non-empty set of *sorts* $\mathcal{S}$. A *many-sorted signature* consists of a set $\mathcal{F}$ of function symbols together with two functions that map each symbol to a finite sequence of *input sorts* and an *output sort*. Fixing a many-sorted signature, we will denote $\mathsf{f} :: [\iota_1 \times \cdots \times \iota_k] \Rightarrow \kappa$ if $\mathsf{f} \in \mathcal{F}$ and f has input sorts $\iota_1, \ldots, \iota_k$ and output sort κ. We also assume given a set $\mathcal{X} = \bigcup_{\iota \in \mathcal{S}} \mathcal{X}_\iota$ of variables disjoint from $\mathcal{F}$, such that all $\mathcal{X}_\iota$ are pairwise disjoint. The set $T_{fo}(\mathcal{F}, \mathcal{X})$ of *many-sorted terms* is inductively defined as the set of expressions s such that $s :: \kappa$ can be derived for some sort κ using the clauses:

$$x :: \kappa \text{ if } x \in \mathcal{X}_\kappa \qquad \mathsf{f}(s_1, \ldots, s_k) :: \kappa \text{ if } \mathsf{f} :: [\iota_1 \times \cdots \times \iota_k] \Rightarrow \kappa \text{ and each } s_i :: \iota_i$$

If $s :: \kappa$, we call κ the sort of s. Substitutions, rewrite rules and reduction are defined as usual in first-order term rewriting, except that substitutions are sort-preserving (each variable is mapped to a term of the same sort) and both sides of a rule have the same sort. We omit these definitions, since they are a special case of the higher-order definitions in Section 2.2.

▶ **Example 1.** We fix nat and list for the sorts of natural numbers and lists of natural numbers, respectively; and a signature with the symbols: $\mathsf{0} :: \mathsf{nat}$ (this is shorthand notation for $[] \Rightarrow \mathsf{nat}$), $\mathsf{s} :: [\mathsf{nat}] \Rightarrow \mathsf{nat}$, $\mathsf{nil} :: \mathsf{list}$, $\mathsf{cons} :: [\mathsf{nat} \times \mathsf{list}] \Rightarrow \mathsf{list}$, $\mathsf{rev} :: [\mathsf{list}] \Rightarrow \mathsf{list}$,

$\mathsf{sum} :: [\mathsf{list}] \Rightarrow \mathsf{nat}$, $\mathsf{append} :: [\mathsf{list} \times \mathsf{list}] \Rightarrow \mathsf{list}$, and $\oplus :: [\mathsf{nat} \times \mathsf{nat}] \Rightarrow \mathsf{nat}$. The rules below compute well-known functions over lists and numbers. We follow the convention of using infix notation for cons and $\oplus$, i.e., $\mathsf{cons}(x, xs)$ is written $x : xs$ and $\oplus(x, y)$ is written $x \oplus y$.

$$\begin{aligned} x \oplus \mathsf{0} &\to x & \mathsf{sum}(\mathsf{nil}) &\to \mathsf{0} \\ x \oplus \mathsf{s}(y) &\to \mathsf{s}(x \oplus y) & \mathsf{sum}(x : xs) &\to \mathsf{sum}(xs) \oplus x \\ \mathsf{append}(\mathsf{nil}, xs) &\to xs & \mathsf{rev}(\mathsf{nil}) &\to \mathsf{nil} \\ \mathsf{append}(x : xs, ys) &\to x : \mathsf{append}(xs, ys) & \mathsf{rev}(x : xs) &\to \mathsf{append}(\mathsf{rev}(xs), x : \mathsf{nil}) \end{aligned}$$

2.2 Higher-Order Rewriting

For higher-order rewriting, we will use *algebraic functional systems* (AFS), a slightly simplified form of a higher-order language introduced by Jouannaud and Okada [29]. This choice gives an easy presentation, as it combines algebraic definitions in a first-order style with a function mechanism using λ-abstractions and term applications.

Given a non-empty set of *sorts* $\mathcal{S}$, the set $\mathcal{ST}$ of *simple types* (or just *types*) is given by: (a) $\mathcal{S} \subseteq \mathcal{ST}$; (b) if $\sigma, \tau \in \mathcal{ST}$ then $\sigma \Rightarrow \tau \in \mathcal{ST}$. Types are denoted by σ, τ and sorts by ι, κ. A *higher-order signature* consists of a set $\mathcal{F}$ of function symbols together with two functions that map each symbol to a finite sequence of *input types* and an *output type*; fixing a signature, we denote this type information $\mathsf{f} :: [\sigma_1 \times \cdots \times \sigma_k] \Rightarrow \tau$. A function symbol is said to be higher-order if at least one of its input types or its output type is an arrow type.

We also assume given a set $\mathcal{X} = \bigcup_{\sigma \in \mathcal{ST}} \mathcal{X}_\sigma$ of variables disjoint from $\mathcal{F}$ (and pairwise disjoint) so that each $\mathcal{X}_\sigma$ is countably infinite. The set $T(\mathcal{F}, \mathcal{X})$ of terms is inductively defined as the set of expressions whose type can be derived using the following clauses:

$$\begin{array}{lll} x :: \sigma & \text{if } x \in \mathcal{X}_\sigma & (\lambda x.s) :: \sigma \Rightarrow \tau \quad \text{if } x \in \mathcal{X}_\sigma \text{ and } s :: \tau \\ (s\,t) :: \tau & \text{if } s :: \sigma \Rightarrow \tau \text{ and } t :: \sigma & \mathsf{f}(s_1, \ldots, s_k) :: \tau \quad \text{if } \mathsf{f} :: [\sigma_1 \times \cdots \times \sigma_k] \Rightarrow \tau \\ & & \qquad\qquad \text{and each } s_i :: \sigma_i \end{array}$$

If $s :: \sigma$, we say that σ is the type of s. It is easy to see that each term has a unique type.

As in the λ-calculus, a variable x is *bound* in a term if it occurs in the scope of an abstractor $\lambda x.$; it is *free* otherwise. A term is called *closed* if it has no free variables and *ground* if it also has no bound variables. Term equality is modulo α-conversion and bound variables are renamed if necessary. Application is left-associative and has precedence over abstractions; for example, $\lambda x.s\,t\,u$ reads $\lambda x.((s\,t)\,u)$. A substitution is a finite, type-preserving mapping $\gamma : \mathcal{X} \to T(\mathcal{F}, \mathcal{X})$, typically denoted $[x_1 := s_1, \ldots, x_n := t_n]$. Its *domain* $\{x_1, \ldots, x_n\}$ is denoted $\mathtt{dom}(\gamma)$. A substitution γ is applied to a term s, notation $s\gamma$, by renaming all bound variables in s to fresh variables and then replacing each $x \in \mathtt{dom}(\gamma)$ by $\gamma(x)$. Formally:

$$\begin{array}{lcll} x\gamma & = & \gamma(x) & \text{if } x \in \mathtt{dom}(\gamma) \\ x\gamma & = & x & \text{if } x \notin \mathtt{dom}(\gamma) \end{array} \qquad \begin{array}{rcll} (s\,t)\gamma & = & (s\gamma)\,(t\gamma) & \\ \mathsf{f}(s_1, \ldots, s_k)\gamma & = & \mathsf{f}(s_1\gamma, \ldots, s_k\gamma) & \\ (\lambda x.s)\gamma & = & \lambda y.(s([x := y]\gamma)) & \text{for } y \text{ fresh} \end{array}$$

Here, $[x := y]\gamma$ is the substitution that maps x to y and all variables in $\mathtt{dom}(\gamma)$ other than x to $\gamma(x)$. The result of $s\gamma$ is unique modulo α-renaming.

A *rewriting rule* is a pair of terms $\ell \to r$ of the same type such that all free variables of r also occur in ℓ. Given a set of rewriting rules $\mathcal{R}$, the rewrite relation induced by $\mathcal{R}$ on the set $T(\mathcal{F}, \mathcal{X})$ is the smallest monotonic relation that is stable under substitution and contains both all elements of $\mathcal{R}$ and β-reduction. That is, it is inductively generated by:

$$\begin{array}{rcll} (\lambda x.s)\ t & \to_{\mathcal{R}} & s[x := t] & \\ \ell\gamma & \to_{\mathcal{R}} & r\gamma & \text{if } \ell \to r \in \mathcal{R} \\ \mathsf{f}(\dots, s, \dots) & \to_{\mathcal{R}} & \mathsf{f}(\dots, t, \dots) & \text{if } s \to_{\mathcal{R}} t \end{array} \qquad \begin{array}{rcll} \lambda x.s & \to_{\mathcal{R}} & \lambda x.t & \text{if } s \to_{\mathcal{R}} t \\ s\ u & \to_{\mathcal{R}} & t\ u & \text{if } s \to_{\mathcal{R}} t \\ u\ s & \to_{\mathcal{R}} & u\ t & \text{if } s \to_{\mathcal{R}} t \end{array}$$

Note that we do not, by default, include the common η-reduction rule scheme ("$\lambda x.s\ x \to_{\mathcal{R}} s$ if x is not a free variable in s"). We avoid this because not all sources consider it, and it is easy to add by including, for all types σ, τ, a rule $\lambda x.F\ x \to F$ with $F \in \mathcal{X}_{\sigma \Rightarrow \tau}$ in $\mathcal{R}$.

An *algebraic functional system* (AFS) is the combination of a set of terms $T(\mathcal{F}, \mathcal{X})$ and a rewrite relation $\to_{\mathcal{R}}$ over $T(\mathcal{F}, \mathcal{X})$. An AFS is typically given by supplying $\mathcal{F}$ and $\mathcal{R}$.

A *many-sorted term rewriting system* (TRS), as discussed in Section 2.1, is a pair $(T_{fo}(\mathcal{F}, \mathcal{X}), \to_{\mathcal{R}})$ where $\mathcal{F}$ is a many-sorted signature and $\to_{\mathcal{R}}$ a rewrite relation over $T_{fo}(\mathcal{F}, \mathcal{X})$. That is, it is essentially an AFS where we only consider first-order terms.

▶ **Example 2.** Following common examples in higher-order rewriting, we will use (as a running example) the AFS $(\mathcal{F}, \mathcal{R})_{\texttt{fold}}$, with symbols $\mathsf{nil} :: \mathsf{list}$, $\mathsf{cons} :: [\mathsf{nat} \times \mathsf{list}] \Rightarrow \mathsf{list}$, $\mathsf{map} :: [(\mathsf{nat} \Rightarrow \mathsf{nat}) \times \mathsf{list}] \Rightarrow \mathsf{list}$, $\mathsf{foldl} :: [(\mathsf{nat} \Rightarrow \mathsf{nat} \Rightarrow \mathsf{nat}) \times \mathsf{nat} \times \mathsf{list}] \Rightarrow \mathsf{nat}$, and rules:

$$\begin{array}{rcl} \mathsf{foldl}(F, z, \mathsf{nil}) & \to & z \\ \mathsf{foldl}(F, z, x : xs) & \to & \mathsf{foldl}(F, (F\ z\ x), xs) \end{array} \qquad \begin{array}{rcl} \mathsf{map}(F, \mathsf{nil}) & \to & \mathsf{nil} \\ \mathsf{map}(F, x : xs) & \to & (F\,x) : \mathsf{map}(F, xs) \end{array}$$

2.3 Functions and orderings

An extended well-founded set is a tuple $(A, >, \geq)$ such that $>$ is a well-founded ordering on A; $\geq$ is a quasi-ordering on A; $x > y$ implies $x \geq y$; and $x > y \geq z$ implies $x > z$. Hence, it is permitted, but not required, that $\geq$ is the reflexive closure of $>$.

For sets A, B, the notation $A \Longrightarrow B$ denotes the set of functions from A to B. Function equality is extensional: for $f, g \in A \Longrightarrow B$ we say $f = g$ iff $f(x) = g(x)$ for all $x \in A$.

If $(A, >, \geq)$ and $(B, \succ, \succeq)$ are extended well-founded sets, we say that $f \in A \Longrightarrow B$ is *weakly monotonic* if $x \geq y$ implies $f(x) \succeq f(y)$. In addition, if $(A_1, >_1, \geq_1), \dots, (A_n, >_n, \geq_n)$ are all well-founded sets, we say that $f \in A_1 \times \cdots \times A_n \Longrightarrow B$ is weakly monotonic if we have $f(x_1, \dots, x_n) \succeq f(y_1, \dots, y_n)$ whenever $x_i \geq_i y_i$ for all $1 \leq i \leq n$. We say that f is *strict* in argument j if $x_j >_j y_j$ (and also $x_i \geq_i y_i$ for all i) implies $f(x_1, \dots, x_n) \succ f(y_1, \dots, y_n)$.

We say that $f \in A_1 \times \cdots \times A_n \Longrightarrow B$ is *strongly monotonic* if f is weakly monotonic and strict in all its arguments (and similar for $f \in A \Longrightarrow B$).

3 First-Order tuple interpretation

In this section, we will introduce the concept of tuple interpretations for many-sorted term rewriting. This is the core methodology which the higher-order theory is built on top of. This theory also has value by itself as a first-order termination and complexity technique.

It is common in the rewriting literature to use termination proofs to assess the *difficulty* of rewriting a term to normal form [7, 27]. The intuition comes from the idea that by ordering rewriting rules in descending order we gauge the order of magnitude of reduction. The same principle applies for syntactic [24, 25, 34] and semantic [27, 26, 35] termination proofs.

On the semantic side there is a natural strategy: given an extended well-founded set $\mathcal{A} = (A, >, \geq)$ find an interpretation from terms to elements of A so that $[\![s]\!] > [\![t]\!]$ whenever $s \to_{\mathcal{R}} t$. (This can typically be done by showing that $[\![\ell]\!] > [\![r]\!]$ for all rules $\ell \to r$). This

interpretation holds information about the complexity of $(\mathcal{F}, \mathcal{R})$ since the maximum length of a reduction starting in a term s is bounded by number of $>$ steps that may be done starting in $[\![s]\!]$. If $[\![s]\!]$ is a natural number, this gives a bound immediately.

In the setting of many-sorted term rewriting, we may formally define this as follows.

▶ **Definition 3.** *Let $\mathcal{S}$ be a set of sorts and $\mathcal{F}$ an $\mathcal{S}$-signature. A many-sorted monotonic algebra $\mathcal{A}$ consists of a family of extended well-founded sets $(A_\iota, >_\iota, \geq_\iota)_{\iota \in \mathcal{S}}$ together with an interpretation $\mathcal{J}$ which associates to each $\mathsf{f} :: [\iota_1 \times \cdots \times \iota_k] \Rightarrow \kappa$ in $\mathcal{F}$ a strongly monotonic function $\mathcal{J}_\mathsf{f} \in A_{\iota_1} \times \cdots \times A_{\iota_k} \Longrightarrow A_\kappa$. Let α be a function that maps variables of sort ι to elements of A_ι. We extend $\mathcal{J}$ to a function $[\![\cdot]\!]_\alpha$ that maps terms of sort ι to elements of A_ι, by letting $[\![x]\!]_\alpha = \alpha(x)$ if x is a variable of sort ι, and $[\![\mathsf{f}(s_1, \ldots, s_k)]\!]_\alpha = \mathcal{J}_\mathsf{f}([\![s_1]\!]_\alpha, \ldots, [\![s_k]\!]_\alpha)$. We say that a TRS $(\mathcal{F}, \mathcal{R})$ is* compatible *with $\mathcal{A}$ if $[\![\ell]\!]_\alpha > [\![r]\!]_\alpha$ for all α and all $\ell \to r \in \mathcal{R}$.*

We will generally omit the subscript α when it is clear from context, writing $[\![s]\!]$ instead of $[\![s]\!]_\alpha$. In examples, we may write something like $[\![s]\!] = x + y$ to mean $[\![s]\!]_\alpha = \alpha(x) + \alpha(y)$.

▶ **Theorem 4.** *If $(\mathcal{F}, \mathcal{R})$ is compatible with $\mathcal{A}$ then for all α: $[\![s]\!]_\alpha > [\![t]\!]_\alpha$ whenever $s \to_\mathcal{R} t$.*

Proof Sketch. By induction on the size of s using strong monotonicity of each $\mathcal{J}_\mathsf{f}$. ◀

A common notion in the literature on complexity of term rewriting is *derivation height*:

$$\mathtt{dh}_\mathcal{R}(t) := \max\{n \in \mathbb{N} \mid \exists s.\, t \to^n s\}.$$

Intuitively, $\mathtt{dh}_\mathcal{R}(t)$ describes the worst-case number of steps for all possible reductions starting in t. If $(\mathcal{F}, \mathcal{R})$ is terminating, then $\mathtt{dh}_\mathcal{R}(\cdot)$ is a total function. If $(A_\iota, >_\iota) = (\mathbb{N}, >)$ then we easily see that $\mathtt{dh}_\mathcal{R}(t) \leq [\![t]\!]$ for any term $t : \iota$. Hence, $[\![\cdot]\!]$ can be used to bound the derivation height function. However, this may give a severe overestimation, as demonstrated below.

▶ **Example 5.** Let $(\mathcal{F}, \mathcal{R})_{\mathsf{ab}}$ be the TRS with only a rule $\mathsf{a}(\mathsf{b}(x)) \to \mathsf{b}(\mathsf{a}(x))$ and signature $\mathsf{a}, \mathsf{b} : [\mathsf{string}] \Rightarrow \mathsf{string}$ and $\epsilon : \mathsf{string}$. We can prove termination by the following interpretation:

$$[\![\mathsf{a}(x)]\!] = 2 * x \qquad [\![\mathsf{b}(x)]\!] = x + 1 \qquad [\![\epsilon]\!] = 0$$

Indeed, we have $[\![\ell]\!] > [\![r]\!]$ for the only rule as $[\![\mathsf{a}(\mathsf{b}(x))]\!] = 2 * x + 2 > 2 * x + 1 = [\![\mathsf{b}(\mathsf{a}(x))]\!]$. Now consider a term $t = \mathsf{a}^n(\mathsf{b}^m(\epsilon))$. Then $\mathtt{dh}_\mathcal{R}(t) = n * m$ whereas $[\![t]\!] = 2^n m$; an exponential difference! Such an overestimation is problematic if we want to use $[\![\cdot]\!]$ to bound $\mathtt{dh}_\mathcal{R}(\cdot)$.

We could find a tight bound for the system of Example 5 by a reasoning like the following: for every term s, let $\#bs(s)$ be the number of b occurrences in s. For a term t, let $cost(t)$ denote $\sum\{\{\#bs(s) \mid \mathsf{a}(s) \text{ is a subterm of } t\}\}$. Then, the cost of a term decreases exactly by 1 in each step. As the normal form has cost 0, we find the tight bound $cost(\mathsf{a}^n(\mathsf{b}^m(\epsilon))) = n * m$.

This reasoning relies on tracking more than one value. We can formalise this reasoning using an algebra interpretation (and will do so in Example 8), by choosing the right $\mathcal{A}$:

▶ **Definition 6.** *A* tuple algebra *is an algebra $\mathcal{A} = (A, \mathcal{J})$ with $A = (A_\iota, >_\iota, \geq_\iota)_{\iota \in \mathcal{S}}$ such that each A_ι has the form $\mathbb{N}^{K[\iota]}$ (for an integer $K[\iota] \geq 1$) and we let $\langle n_1, \ldots, n_{K[\iota]} \rangle \geq_\iota \langle n'_1, \ldots, n'_{K[\iota]} \rangle$ if each $n_i \geq n'_i$, and $\langle n_1, \ldots, n_{K[\iota]} \rangle >_\iota \langle n'_1, \ldots, n'_{K[\iota]} \rangle$ if additionally $n_1 > n'_1$.*

Intuitively, the first component always indicates "cost": the number of steps needed to reduce a term to normal form. This is the component that needs to decrease in each rewrite step to have $[\![s]\!] > [\![t]\!]$ whenever $s \to_\mathcal{R} t$. The remaining components represent some value of interest for the sort. This could for example be the size of the term (or its normal form), the length of a list, or following Example 5, the number of occurrences of a specific symbol. For these components, we only require that they do not increase in a reduction step.

By the definition of $>_\iota$, and using Theorem 4, we can conclude:

▶ **Corollary 7.** *If a TRS $(\mathcal{F},\mathcal{R})$ is compatible with a tuple algebra then it is terminating and $\mathtt{dh}_{\mathcal{R}}(t) \leq [\![t]\!]_1$, for all terms t. (Here, $[\![t]\!]_1$ indicates the first component of the tuple $[\![t]\!]$.)*

Using this, we obtain a tight bound on the derivation height of $\mathsf{a}^n(\mathsf{b}^m(\epsilon))$ in Example 5:

▶ **Example 8.** The TRS $(\mathcal{F},\mathcal{R})_{\mathsf{ab}}$ is compatible with the tuple algebra with $A_{\mathsf{string}} = \mathbb{N}^2$ and

$$[\![\mathsf{a}(x)]\!] = \langle x_1 + x_2, x_2\rangle \qquad [\![\mathsf{b}(x)]\!] = \langle x_1, x_2 + 1\rangle \qquad [\![\epsilon]\!] = \langle 0, 0\rangle$$

Here, again, subscripts indicate tuple indexing; i.e., $\langle n, m\rangle_1 = n$ and $\langle n, m\rangle_2 = m$. Note that for every ground term s we have $[\![s]\!]_2 = \#bs(s)$. The first component exactly sums $\#bs(t)$ for every subterm t of s which has the form $\mathsf{a}(t')$. We have: $[\![\mathsf{a}(\mathsf{b}(x))]\!] = \langle x_1 + x_2 + 1, x_2 + 1\rangle >_{\mathsf{nat}} \langle x_1 + x_2, x_2 + 1\rangle = [\![\mathsf{b}(\mathsf{a}(x))]\!]$. The interpretation functions $\mathcal{J}_{\mathsf{a}}$ and $\mathcal{J}_{\mathsf{b}}$ are indeed monotonic. For example, for $\mathcal{J}_{\mathsf{a}}$: if $x >_{\mathsf{nat}} y$ then $x_1 + x_2 > y_1 + y_2$ (since $x_1 > y_1$ and $x_2 \geq y_2$) and $x_2 \geq y_2$; and if $x \geq_{\mathsf{nat}} y$ then $x_1 + x_2 \geq y_1 + y_2$ and $x_2 \geq y_2$. We have $[\![\mathsf{a}^n(\mathsf{b}^m(\epsilon))]\!] = (n{*}m, m)$.

To build strongly monotonic functions we can for instance use the following observation:

▶ **Lemma 9.** *A function $F : \mathbb{N}^{K[\iota_1]} \times \cdots \times \mathbb{N}^{K[\iota_k]} \Longrightarrow \mathbb{N}^{K[\kappa]}$ is strongly monotonic if we can write $F(x^1,\ldots,x^k) = \langle x_1^1 + \cdots + x_1^k + S_1(x^1,\ldots,x^k),\ S_2(x^1,\ldots,x^k),\ldots,\ S_{K[\kappa]}(x^1,\ldots,x^k)\rangle$, where each S_i is a weakly monotonic function in $\mathbb{N}^{K[\iota_1]} \times \cdots \times \mathbb{N}^{K[\iota_k]} \Longrightarrow \mathbb{N}$.*

Moreover, a function $S : \mathbb{N}^{K[\iota_1]} \times \cdots \times \mathbb{N}^{K[\iota_k]} \Longrightarrow \mathbb{N}$ is weakly monotonic if it is built from constants in $\mathbb{N}$, variable components x_j^n and weakly monotonic functions in $\mathbb{N}^n \Longrightarrow \mathbb{N}$.

For the "weakly monotonic functions in $\mathbb{N}^n \Longrightarrow \mathbb{N}$" we could for instance use $+$, $*$ or max.

To determine the length $K[\iota]$ of the tuple for a sort ι, we use a semantic approach, similar to one used in [19] in the context of functional languages: the elements of the tuple are values of interest for the sort. The two prominent examples in this paper are the sort nat of natural numbers – which is constructed from the symbols $\mathsf{0} :: \mathsf{nat}$ and $\mathsf{s} :: [\mathsf{nat}] \Rightarrow \mathsf{nat}$ – and the sort list of lists of natural numbers – which is constructed using $\mathsf{nil} :: \mathsf{list}$ and $\mathsf{cons} :: [\mathsf{nat} \times \mathsf{list}] \Rightarrow \mathsf{list}$. For natural numbers, we consider their size, so the number of ss. For lists, we consider both their length and an upper bound on the size of their elements. This gives $K[\mathsf{nat}] = 2$ (cost of reducing the term, size of its normal form) and $K[\mathsf{list}] = 3$ (cost of reducing, length of normal form, maximum element size). In the remainder of this paper, we will use x_{c} as syntactic sugar for x_1 (the cost component of x), x_{s} and x_{l} as x_2 and x_{m} as x_3.

▶ **Example 10.** Consider the TRS defined in Example 1. We start by giving an interpretation for the type constructors: the symbols $\mathsf{0}, \mathsf{nil}, \mathsf{s}$ and cons which are used to construct natural numbers and lists. To be in line with the semantics for the type interpretation, we let:

$$[\![\mathsf{0}]\!] = \langle 0, 0\rangle \qquad [\![\mathsf{s}(x)]\!] = \langle x_{\mathsf{c}}, x_{\mathsf{s}} + 1\rangle$$
$$[\![\mathsf{nil}]\!] = \langle 0, 0, 0\rangle \qquad [\![x : xs]\!] = \langle x_{\mathsf{c}} + xs_{\mathsf{c}}, xs_{\mathsf{l}} + 1, \max(x_{\mathsf{s}}, xs_{\mathsf{m}})\rangle$$

This expresses that $\mathsf{0}$ has no evaluation cost and size 0; analogously, nil has no evaluation cost and 0 as length and maximum element. The cost of evaluating a term $\mathsf{s}(t)$ depends entirely on the cost of the term's argument t; the size component counts the number of ss. The cost component for cons similarly sums the costs of its arguments, while the length is increased by 1, and the maximum element is the maximum between its head and tail.

For the remaining symbols we choose the following interpretations:

$$\begin{aligned}
[\![x \oplus y]\!] &= \langle x_{\mathsf{c}} + y_{\mathsf{c}} + y_{\mathsf{s}} + 1, x_{\mathsf{s}} + y_{\mathsf{s}}\rangle \\
[\![\mathsf{sum}(xs)]\!] &= \langle xs_{\mathsf{c}} + 2 * xs_{\mathsf{l}} + xs_{\mathsf{l}} * xs_{\mathsf{m}} + 1, xs_{\mathsf{l}} * xs_{\mathsf{m}}\rangle \\
[\![\mathsf{rev}(xs)]\!] &= \langle xs_{\mathsf{c}} + xs_{\mathsf{l}} + \tfrac{xs_{\mathsf{l}} * (xs_{\mathsf{l}}+1)}{2} + 1, xs_{\mathsf{l}}, xs_{\mathsf{m}}\rangle \\
[\![\mathsf{append}(xs, ys)]\!] &= \langle xs_{\mathsf{c}} + ys_{\mathsf{c}} + xs_{\mathsf{l}} + 1, xs_{\mathsf{l}} + ys_{\mathsf{l}}, \max(xs_{\mathsf{m}}, ys_{\mathsf{m}})\rangle
\end{aligned}$$

Checking compatibility is easily done for the interpretation above, and strong monotonicity follows by Lemma 9 (as $n \mapsto \frac{n*(n+1)}{2} \in \mathbb{N} \Longrightarrow \mathbb{N}$ is weakly monotonic). We see that the cost of evaluating append is linear in the first list length and independent of the size of the list elements, while evaluating sum gives a quadratic dependency on length and size combined.

Our tuple interpretations have some similarities with matrix interpretations [21], where also each term is associated to an n-tuple. In essence, matrix interpretations *are* tuple interpretations, for systems with only one sort. However, the shape of the interpretation functions $\mathcal{J}_{\mathsf{f}}$ in matrix interpretations is limited to functions following Lemma 9 where each S is a linear multivariate polynomial. Hence, our interpretations are a strict generalisation, which also admits interpretations such as those used for sum, rev and append in Example 10.

For the purpose of termination, tuple interpretations strictly extend the power of both polynomial interpretations and matrix interpretations already in the first-order case.

▶ **Example 11.** A TRS that implements division in [4] shows a limitation of polynomial interpretations: it contains a rule $\mathsf{quot}(\mathsf{s}(x), \mathsf{s}(y)) \to \mathsf{s}(\mathsf{quot}(\mathsf{minus}(x, y), \mathsf{s}(y)))$ which cannot be oriented by any polynomial interpretation, because $[\![\mathsf{minus}(x, \mathsf{s}(x))]\!] > [\![\mathsf{s}(x)]\!]$ for any strongly monotonic polynomial $\mathcal{J}_{\mathsf{minus}}$. Due to the duplication of y, this rule also cannot be handled by a matrix interpretation. However, we do have a compatible tuple interpretation:

$$\begin{array}{rclrcl} [\![\mathsf{0}]\!] & = & \langle 0, 0\rangle & [\![\mathsf{minus}(x,y)]\!] & = & \langle x_{\mathsf{c}} + y_{\mathsf{c}} + y_{\mathsf{s}} + 1, x_{\mathsf{s}}\rangle \\ [\![\mathsf{s}(x)]\!] & = & \langle x_{\mathsf{c}}, x_{\mathsf{s}} + 1\rangle & [\![\mathsf{quot}(x,y)]\!] & = & \langle x_{\mathsf{c}} + x_{\mathsf{s}} + y_{\mathsf{c}} + x_{\mathsf{s}} * y_{\mathsf{c}} + x_{\mathsf{s}} * y_{\mathsf{s}} + 1, x_{\mathsf{s}}\rangle \end{array}$$

In practice, in first-order termination or complexity analysis one would not exclusively use interpretations, but rather a combination of different techniques. In that context, tuple interpretations may be used as one part of a large toolbox. They are likely to offer a simple complexity proof in many cases, but they are unlikely to be an essential technique since so many other methods have already been developed. Indeed, all examples in this section can be handled with previously established theory. For instance, Example 5 can be handled with matrix interpretations, while sum and rev may be analysed using ideas from [24] and [35].

However, developing a new technique for first-order termination and traditional complexity analysis is not our goal. Our method *does* provide a more fine-grained notion of complexity, which may consider information such as the length of a list. Moreover, the first-order case is an important stepping stone towards higher-order analysis, where far fewer methods exist.

4 Higher-order tuple interpretations

In this section, we will extend the ideas from Section 3 to the higher-order setting, and hence define the core notion of this paper: higher-order tuple interpretations. To do this, we will build on the notion of *strongly monotonic algebras* originating in [39].

4.1 Strongly monotonic algebras

In first-order term rewriting, the complexity of a TRS is often measured as *runtime* or *derivational* complexity. Both measures consider initial terms s of a certain shape, and supply a bound on $\mathtt{dh}_{\mathcal{R}}(s)$ given the size of s. However, this is not a good approach for higher-order terms: the behaviour of a term of higher type generally cannot be captured in an integer.

▶ **Example 12.** Consider the AFS obtained by combining Examples 1 and 2. The evaluation cost of a term $\mathsf{foldl}(F, n, q)$ depends almost completely on the behaviour of the functional subterm F, and not only on its evaluation cost. To see this, consider two cases: $F_1 :=$

$\lambda x.\lambda y.y \oplus x$, and $F_2 := \lambda x.\lambda y.x \oplus x$. For natural numbers n, m, the evaluation cost of both $F_1(n, m)$ and $F_2(n, m)$ is the same: $n + 1$. However, the *size* of the result is different. Hence, the number of steps needed to compute $\mathsf{foldl}(F_1, n, q)$ for a number n and list q is quadratic in the size of n and q, while the number of steps needed for $\mathsf{foldl}(F_2, n, q)$ is exponential.

As Example 12 shows, higher-order rewriting is a natural place to separate cost and size. But more than that, we need to know what a function does with its arguments: whether it is size-increasing, how long it takes to evaluate them, and more.

This is naturally captured by the notion of (weakly or strongly) monotonic algebras for higher-order rewriting introduced by v.d. Pol [39]: here, a term of arrow type is interpreted as a function, which allows the interpretation to retain all relevant information.

Monotonic interpretations were originally defined for a different higher-order rewriting formalism, which does make some difference in the way abstraction and application is handled. *Weakly* monotonic algebras were transposed to AFSs in [22]; however, here we extend the more natural notion of *hereditarily* monotonic algebras which v.d. Pol only briefly considered.[1]

▶ **Definition 13.** *Let $\mathcal{S}$ be a set of sorts and $\mathcal{F}$ a higher-order signature. We assume given for every sort ι an extended well-founded set $(A_\iota, >_\iota, \geq_\iota)$. From this, we define the set of* strongly monotonic functionals*, as follows:*

- *For all sorts ι: $\mathcal{M}_\iota := A_\iota$ and $\sqsupset_\iota := >_\iota$ and $\sqsupseteq_\iota := \geq_\iota$.*
- *For an arrow type $\sigma \Rightarrow \tau$:*
  - *$\mathcal{M}_{\sigma\Rightarrow\tau} := \{F \in \mathcal{M}_\sigma \Longrightarrow \mathcal{M}_\tau \mid F$ is strongly monotoic$\}$*
  - *$F \sqsupset_{\sigma\Rightarrow\tau} G$ iff $\mathcal{M}_\sigma$ is non-empty and $\forall x \in \mathcal{M}_\sigma.F(x) \sqsupset_\tau G(x)$, and $F \sqsupseteq_{\sigma\Rightarrow\tau} G$ iff $\forall x \in \mathcal{M}_\sigma.F(x) \sqsupseteq_\tau G(x)$.*

That is, $\mathcal{M}_{\sigma\Rightarrow\tau}$ contains strongly monotonic functions from $\mathcal{M}_\sigma$ to $\mathcal{M}_\tau$ and both $\sqsupset_{\sigma\Rightarrow\tau}$ and $\sqsupseteq_{\sigma\Rightarrow\tau}$ do a point-wise comparison. By a straightforward induction on types we have:

▶ **Lemma 14.** *For all types σ, $(\mathcal{M}_\sigma, \sqsupset_\sigma, \sqsupseteq_\sigma)$ is an extended well-founded set; that is:*

- *$\sqsupset_\sigma$ is well-founded and $\sqsupseteq_\sigma$ is reflexive;*
- *both $\sqsupset_\sigma$ and $\sqsupseteq_\sigma$ are transitive;*
- *for all $x, y, z \in \mathcal{M}_\sigma$, $x \sqsupset_\sigma y$ implies $x \sqsupseteq_\sigma y$ and $x \sqsupset_\sigma y \sqsupseteq_\sigma z$ implies $x \sqsupset_\sigma z$.*

We will define *higher-order strongly monotonic algebras* as an extension of Definition 3, mapping a term of type σ to an element of $\mathcal{M}_\sigma$. Functional terms $\mathsf{f}(s_1, \ldots, s_k)$ and variables can be handled as before, but we now also have to deal with application and abstraction. Application is straightforward: since terms of higher type are mapped to functions, we can interpret application as function application, i.e., $[\![s \cdot t]\!]_\alpha := [\![s]\!]_\alpha([\![t]\!]_\alpha)$. However, abstraction is more difficult. The natural choice would be to view abstraction as defining a function; i.e., let $[\![\lambda x.s]\!]_\alpha$ be the function $d \mapsto [\![s]\!]_{\alpha[x:=d]}$. Unfortunately, this is not necessarily monotonic: $d \mapsto [\![s]\!]_{\alpha[x:=d]}$ is strongly monotonic only if x occurs freely in s. For example $\lambda x.\mathsf{0}$ would be mapped to a constant function, which is not in $\mathcal{M}_{\mathsf{nat}\Rightarrow\mathsf{nat}}$. Moreover, this definition would give $[\![(\lambda x.s) \cdot t]\!]_\alpha = [\![s[x := t]]\!]_\alpha$, so β-steps would not be counted toward the evaluation cost.

We handle both problems by using a choosable function $\mathit{MakeSM}_{\sigma,\tau}$, which takes a function that may be strongly monotonic or constant, and turns it strongly monotonic.

[1] In [39], v.d. Pol rejects hereditarily (or: strongly) monotonic algebras because they are not so well-suited for analysing the HRS format [36] where reasoning is modulo $\rightarrow_\beta$: it is impossible to both interpret all terms of function type to strongly monotonic functions and have $[\![(\lambda x.s)\ t]\!] = [\![s[x := t]]\!]$. In the AFS format, we do not have the latter requirement. In [22], where the authors considered the AFS format like we do here (but for interpretations to $\mathbb{N}$ rather than to tuples), weakly monotonic algebras were used because they are a more natural choice in the context of dependency pairs.

▶ **Definition 15.** *A (σ,τ)-monotonicity function $MakeSM_{\sigma,\tau}$ is a strongly monotonic function in $C_{\sigma,\tau} \Longrightarrow \mathcal{M}_{\sigma\Longrightarrow\tau}$, where the set $C_{\sigma,\tau}$ is defined as $\mathcal{M}_{\sigma\Rightarrow\tau} \cup \{F \in \mathcal{M}_\sigma \Longrightarrow \mathcal{M}_\tau \mid F(x) = F(y)$ for all $x, y \in \mathcal{M}_\sigma\}$. (Here, the set $C_{\sigma,\tau}$ is ordered by point-wise comparison.)*

With this definition, we are ready to define strongly monotonic algebras.

▶ **Definition 16.** *A strongly monotonic algebra $\mathcal{A}_\mathcal{M}$ consists of a family $(\mathcal{M}_\sigma, \sqsupset_\sigma, \sqsupseteq_\sigma)_{\sigma\in\mathcal{ST}}$, an interpretation function $\mathcal{J}$ which associates to each $\mathsf{f} :: [\sigma_1 \times \cdots \times \sigma_k] \Rightarrow \tau$ in $\mathcal{F}$ an element of $\mathcal{M}_{\sigma_1\Rightarrow\ldots\Rightarrow\sigma_k\Rightarrow\tau}$, and a (σ,τ)-monotonicity function $MakeSM_{\sigma,\tau}$, for each $\sigma,\tau \in \mathcal{ST}$.*

Let α be a function that maps variables of type σ to elements of $\mathcal{M}_\sigma$. We extend $\mathcal{J}$ to a function $[\![\cdot]\!]_\alpha$ that maps terms of type σ to elements of $\mathcal{M}_\sigma$, as follows:

$$[\![x]\!]_\alpha = \alpha(x) \text{ for variables } x \qquad [\![\mathsf{f}(s_1,\ldots,s_k)]\!]_\alpha = \mathcal{J}_\mathsf{f}([\![s_1]\!]_\alpha,\ldots,[\![s_k]\!]_\alpha)$$
$$[\![s\cdot t]\!]_\alpha = [\![s]\!]_\alpha([\![t]\!]_\alpha) \qquad [\![\lambda x.s]\!]_\alpha = MakeSM_{\sigma,\tau}(d \mapsto [\![s]\!]_{\alpha[x:=d]}) \text{ if } x :: \sigma \text{ and } s :: \tau$$

We can see by induction on s that for $s :: \sigma$ indeed $[\![s]\!]_\alpha \in \mathcal{M}_\sigma$. We say that an AFS $(\mathcal{F},\mathcal{R})$ is *compatible* with $\mathcal{A}_\mathcal{M}$ if for all valuations α both (1) $[\![\ell]\!]_\alpha \sqsupset [\![r]\!]_\alpha$, for all $\ell \to r \in \mathcal{R}$; and (2) $[\![(\lambda x.s)\,t]\!]_\alpha \sqsupset [\![s[x := t]]\!]_\alpha$, for any $s :: \sigma$, $t :: \tau$ and $x \in \mathcal{X}_\tau$.

As before, we will typically omit the α subscript and use notation like $[\![s]\!] = F(x+3)$ to denote $[\![s]\!]_\alpha = \alpha(F)(\alpha(x)+3)$. When types are not relevant, we will denote $\sqsupset$ instead of specifying $\sqsupset_\sigma$, and we may write $f \in \mathcal{M}$ to mean $f \in \mathcal{M}_\sigma$ for some $\sigma \in \mathcal{ST}$.

We extend Theorem 4 into the following compatibility result.

▶ **Theorem 17.** *If $(\mathcal{F},\mathcal{R})$ is compatible with $\mathcal{A}_\mathcal{M}$, then for all α, $[\![s]\!]_\alpha \sqsupset [\![t]\!]_\alpha$ when $s \to_\mathcal{R} t$.*

For Definition 13 and Theorem 17, we can choose the well-founded sets $(A_\iota, >_\iota, \geq_\iota)$ for each sort, and the functions $MakeSM_{\sigma,\tau}$ for each pair of types, as we desire. A *higher-order tuple algebra* is a strongly monotonic algebra where each $(A_\iota, >_\iota, \geq_\iota)$ follows Definition 6.

▶ **Example 18.** Let $A_\mathsf{nat} = \mathbb{N}^2$ and $A_\mathsf{list} = \mathbb{N}^3$ as before, and assume cons and nil are interpreted as in Example 10. Consider the rules for map in Example 2. We let:

$$[\![\mathsf{map}(F, xs)]\!] = \langle (xs_\mathsf{l}+1) * (F(\langle xs_\mathsf{c}, xs_\mathsf{m}\rangle)_\mathsf{c}+1),\ xs_\mathsf{l},\ F(xs_\mathsf{c}, xs_\mathsf{m})_\mathsf{s}\rangle$$

This expresses that map does not increase the list length (as the length component is just xs_l), the greatest element of the result is bounded by the value of F on the greatest element of xs, and the evaluation cost is mostly expressed by a number of F steps that is linear in the length of xs. We will see in Lemma 23 that $\mathcal{J}_\mathsf{map}$ is indeed strongly monotonic.

To prove compatibility of the AFS with $\mathcal{A}_\mathcal{M}$, we must first see that $[\![\ell]\!] \sqsupset [\![r]\!]$ for all rules $\ell \to_\mathcal{R} r$. For the first map rule this is easy: $[\![\mathsf{map}(F,\mathsf{nil})]\!] = \langle F(\langle 0,0\rangle)_\mathsf{c}+1, 0, F(\langle 0,0\rangle)_\mathsf{s}\rangle \sqsupset_\mathsf{list} \langle 0,0,0\rangle = [\![\mathsf{nil}]\!]$. For the second map rule, we must check that $\langle \text{cost-}\ell, \text{len-}\ell, \text{max-}\ell\rangle \sqsupset_\mathsf{list} \langle \text{cost-}r, \text{len-}r, \text{max-}r\rangle$; that is, cost-$\ell$ > cost-r and len-$\ell \geq$ len-r and max-$\ell \geq$ max-r, where:

$$\begin{array}{lcll}
\text{cost-}\ell & = & [\![\mathsf{map}(F, x:xs)]\!]_\mathsf{c} & = (xs_\mathsf{l}+2) * (F(\langle x_\mathsf{c} + xs_\mathsf{c}, \max(x_\mathsf{s}, xs_\mathsf{m})\rangle)_\mathsf{c}+1) \\
\text{cost-}r & = & [\![F(x) : \mathsf{map}(F, xs)]\!]_\mathsf{c} & = F(\langle x_\mathsf{c}, x_\mathsf{s}\rangle)_\mathsf{c} + (xs_\mathsf{l}+1) * (F(\langle xs_\mathsf{c}, xs_\mathsf{m}\rangle)_\mathsf{c}+1) \\
\text{len-}\ell & = & [\![\mathsf{map}(F, x:xs)]\!]_\mathsf{l} & = xs_\mathsf{l}+1 \quad = \quad [\![F(x) : \mathsf{map}(F, xs)]\!]_\mathsf{l} \quad = \quad \text{len-}r \\
\text{max-}\ell & = & [\![\mathsf{map}(F, x:xs)]\!]_\mathsf{m} & = F(\langle x_\mathsf{c} + xs_\mathsf{c}, \max(x_\mathsf{s}, xs_\mathsf{m})\rangle)_\mathsf{s} \\
\text{max-}r & = & [\![F(x) : \mathsf{map}(F, xs)]\!]_\mathsf{m} & = \max(F(\langle x_\mathsf{c}, x_\mathsf{s}\rangle)_\mathsf{s}, F(\langle xs_\mathsf{c}, xs_\mathsf{m}\rangle)_\mathsf{s})
\end{array}$$

To see why cost-ℓ > cost-r, we observe that for all x, xs: $\langle x_\mathsf{c} + xs_\mathsf{c}, \max(x_\mathsf{s} + xs_\mathsf{m})\rangle \sqsupseteq_\mathsf{nat}$ both $\langle x_\mathsf{c}, x_\mathsf{s}\rangle$ and $\langle xs_\mathsf{c}, xs_\mathsf{m}\rangle$. Since $F \in \mathcal{M}_{\mathsf{nat}\Rightarrow\mathsf{nat}}$ therefore $F(\langle x_\mathsf{c} + xs_\mathsf{c}, \max(x_\mathsf{s} + xs_\mathsf{m})\rangle) \sqsupseteq_\mathsf{nat}$ both $F(\langle x_\mathsf{c}, x_\mathsf{s}\rangle)$ and $F(\langle xs_\mathsf{c}, xs_\mathsf{m}\rangle)$. We find max-$\ell \geq$ max-r by a similar reasoning.

4.2 Interpreting abstractions

Example 18 is not complete: we have not yet defined the functions $\mathit{MakeSM}_{\sigma,\tau}$, and we have not shown that $[\![(\lambda x.s)\ t]\!] \sqsupset [\![s[x := t]]\!]$ always holds. To achieve this, we will define some standard functions to build elements of $\mathcal{M}$. This allows us to easily construct strongly monotonic functionals, both to build $\mathit{MakeSM}_{\sigma,\tau}$ and to create interpretation functions $\mathcal{J}_{\mathsf{f}}$.

▶ **Definition 19.** *For every type σ, we define:* $\mathtt{0}_\sigma \in \mathcal{M}_\sigma$; $\mathtt{costof}_\sigma \in \mathcal{M}_\sigma \Longrightarrow \mathbb{N}$; *and* $\mathtt{addc}_\sigma \in \mathbb{N} \times \mathcal{M}_\sigma \Longrightarrow \mathcal{M}_\sigma$ *by mutual recursion on σ as follows.*

$$\begin{array}{ll}
\mathtt{0}_\iota = \langle 0, \ldots, 0\rangle & \mathtt{0}_{\sigma\Rightarrow\tau} = d \mapsto \mathtt{addc}_\tau(\mathtt{costof}_\sigma(d), \mathtt{0}_\tau) \\
\mathtt{costof}_\iota(\langle n_1, \ldots, n_{K[\iota]}\rangle) = n_1 & \mathtt{costof}_{\sigma\Rightarrow\tau}(F) = \mathtt{costof}_\tau(F(\mathtt{0}_\sigma)) \\
\mathtt{addc}_\iota(c, \langle n_1, \ldots, n_{K[\iota]}\rangle) = \langle c + n_1, n_2, \ldots, n_{K[\iota]}\rangle & \mathtt{addc}_{\sigma\Rightarrow\tau}(c, F) = d \mapsto \mathtt{addc}_\tau(c, F(d))
\end{array}$$

Here, $\mathtt{0}_\sigma$ defines the *minimal element* of $\mathcal{M}_\sigma$. The function $\mathtt{costof}_\sigma$ maps every F to the cost component of $F(\mathtt{0}_{\sigma_1}, \ldots, \mathtt{0}_{\sigma_m})$; hence, if $F \sqsupset_\sigma G$ we have $\mathtt{costof}_\sigma(F) > \mathtt{costof}_\sigma(G)$. The function $\mathtt{addc}_\sigma$ pointwise increases an element of $\mathcal{M}_\sigma$ by adding to the cost component: if $F(x_1, \ldots, x_m) = \langle n_1, \ldots, n_k\rangle$, then $\mathtt{addc}(c, F)(x_1, \ldots, x_m) = \langle c + n_1, n_2, \ldots, n_k\rangle$.

It is easy to see that $\mathtt{0}_\sigma$ and $\mathtt{addc}_\sigma(n, X)$ are in $\mathcal{M}$ for all σ (by induction on σ), and that $\mathtt{costof}_\sigma$ and $\mathtt{addc}_\sigma$ are strict in all their arguments. Various properties of these functions are detailed in the appendix (Lemmas B.4–B.8). We will particularly use that always $F(\mathtt{addc}(n, x)) \sqsupseteq \mathtt{addc}(n, F(x))$ (Lemma B.7) and $\mathtt{costof}(F(x)) \geq \mathtt{costof}(x)$ (Lemma B.8).

We can use these functions to for instance create candidates for $\mathit{MakeSM}_{\sigma,\tau}$. While many suitable definitions are possible, we will particularly consider the following:

▶ **Definition 20.** *For types σ, τ, and F a weakly monotonic function in $\mathcal{M}_\sigma \Longrightarrow \mathcal{M}_\tau$, let:*

$$\Phi_{\sigma,\tau}(F) = \begin{cases} d \mapsto \mathtt{addc}_{\sigma\Rightarrow\tau}(1, F(d)) & \text{if } F \text{ is in } \mathcal{M}_{\sigma\Rightarrow\tau} \\ d \mapsto \mathtt{addc}_{\sigma\Rightarrow\tau}(\mathtt{costof}_\sigma(d) + 1, F(d)) & \text{otherwise} \end{cases}$$

Then $\Phi_{\sigma,\tau}$ is a (σ,τ)-monotonicity function. To see this, the most challenging part is proving that $\Phi_{\sigma,\tau}(F) \sqsupset \Phi_{\sigma,\tau}(G)$ if $F \sqsupset G$ and $F \in \mathcal{M}_{\sigma\Rightarrow\tau}$ while G is a constant function. We can prove this using the result that $x \sqsupset y$ implies $\mathtt{addc}(1, x) \sqsupseteq y$ for all x, y. We have:

▶ **Lemma 21.** *If $\mathit{MakeSM}_{\sigma,\tau} = \Phi_{\sigma,\tau}$ then $[\![(\lambda x.s)\, t]\!] \sqsupset_\tau [\![s[x := t]]\!]$, for $s :: \tau$, $t :: \sigma$, $x \in \mathcal{X}_\sigma$.*

Proof Sketch. We expand $\mathit{MakeSM}_{\sigma,\tau}$ to achieve $[\![(\lambda x.s)\, t]\!]_\alpha = \mathtt{addc}_\tau(\mathtt{costof}_\sigma([\![t]\!]_\alpha) + 1, [\![s]\!]_{\alpha[x:=[\![t]\!]]})$ or $[\![(\lambda x.s)\, t]\!]_\alpha = \mathtt{addc}_\tau(1, [\![s]\!]_{\alpha[x:=[\![t]\!]]})$. By induction on τ we prove that $\mathtt{addc}_\tau(n, F) \sqsupset_\tau F$ for all $n \geq 1$. So either way, $[\![(\lambda x.s)\, t]\!]_\alpha \sqsupset_\tau [\![s]\!]_{\alpha[x:=[\![t]\!]]}$. Finally, we prove a substitution lemma, $[\![s]\!]_{\alpha[x:=[\![t]\!]_\alpha]} = [\![s[x := t]]\!]_\alpha$, by induction on s. ◀

In examples in the remainder of this paper, we will assume that $\mathit{MakeSM}_{\sigma,\tau} = \Phi_{\sigma,\tau}$. With these choices we do not only orient the β-rule (and thus satisfy item (2) of the compatibility conditions), but also the η-reduction rules mentioned in Section 2.2.

▶ **Lemma 22.** *If $\mathit{MakeSM}_{\sigma,\tau} = \Phi_{\sigma,\tau}$ then for any $F \in \mathcal{X}_{\sigma\Rightarrow\tau}$ we have: $[\![\lambda x.F\, x]\!] \sqsupset_{\sigma\Rightarrow\tau} [\![F]\!]$.*

Proof Sketch. Since $F \neq x$, we have $[\![F]\!]_{\alpha[x:=d]} = \alpha(F)$ for all α and d. Consequently, $[\![\lambda x.F\, x]\!] \sqsupseteq_{\sigma\Rightarrow\tau} d \mapsto \mathtt{addc}_\tau(1, F(d))$ either way. We are done as: $\mathtt{addc}_\tau(1, F(d)) \sqsupset_\tau F(d)$. ◀

4.3 Creating strongly monotonic interpretation functions

We can use Theorem 17 to obtain bounds on the derivation heights of given terms. However, to achieve this, we must find an interpretation function $\mathcal{J}$, and prove that each $\mathcal{J}_{\mathsf{f}}$ is in $\mathcal{M}$. We will now explore ways to construct such strongly monotonic functions. It turns out to be useful to also consider *weakly* monotonic functions. In the following, we will write "f is $\mathrm{wm}(A_1, \dots, A_k; B)$" to mean that f is a weakly monotonic function in $A_1 \times \cdots \times A_k \Longrightarrow B$.

▶ **Lemma 23.** *Let $x^1, \dots, x^k$ be variables ranging over $\mathcal{M}_{\sigma_1}, \dots, \mathcal{M}_{\sigma_k}$ respectively; we shortly denote this sequence $\vec{x}$. We let $\overrightarrow{\mathcal{M}_\sigma}$ denote the sequence $\mathcal{M}_{\sigma_1}, \dots, \mathcal{M}_{\sigma_k}$. Then:*

1. *if $F(\vec{x}) = x^i$ then F is $wm(\overrightarrow{\mathcal{M}_\sigma}; \mathcal{M}_{\sigma_i})$, and F is strict in argument i;*
2. *if $F(\vec{x}) = x^i(F_1(\vec{x}), \dots, F_n(\vec{x}))$, $\sigma_i = \tau_1 \Rightarrow \dots \Rightarrow \tau_n \Rightarrow \rho$, and each F_j is $wm(\overrightarrow{\mathcal{M}_\sigma}; \mathcal{M}_{\tau_j})$ then F is $wm(\overrightarrow{\mathcal{M}_\sigma}; \mathcal{M}_\rho)$ and for all $p \in \{1, \dots, k\}$: F is strict in argument p if $p = i$ or some F_j is strict in argument p;*
3. *if $F(\vec{x}) = \langle G_1(\vec{x}), \dots, G_{K[\iota]}(\vec{x})\rangle$ and each G_j is $wm(\overrightarrow{\mathcal{M}_\sigma}; \mathbb{N})$ then F is $wm(\overrightarrow{\mathcal{M}_\sigma}; \mathcal{M}_\iota)$, and for all $p \in \{1, \dots, k\}$: F is strict in argument p if G_1 is.*

The last result uses functions mapping to $\mathbb{N}$; these can be constructed using the observations:

4. *if $G(\vec{x}) = n$ for some $n \in \mathbb{N}$ then G is $wm(\overrightarrow{\mathcal{M}_\sigma}; \mathbb{N})$;*
5. *if $G(\vec{x}) = x^i_j$ and $\sigma_i = \iota \in \mathcal{S}$ and $1 \leq j \leq K[\iota]$, then G is $wm(\overrightarrow{\mathcal{M}_\sigma}; \mathbb{N})$, and G is strict in argument i if $j = 1$;*
6. *if $G(\vec{x}) = f(G_1(\vec{x}), \dots, G_n(\vec{x}))$ and all G_j are $wm(\overrightarrow{\mathcal{M}_\sigma}; \mathbb{N})$ and f is $wm(\mathbb{N}, \dots, \mathbb{N}; \mathbb{N})$, then G is $wm(\overrightarrow{\mathcal{M}_\sigma}; \mathbb{N})$, and for all $p \in \{1, \dots, k\}$: G is strict in argument p if, for some $j \in \{1, \dots, n\}$: G_j is strict in argument p and f is strict in argument j;*
7. *if $G(\vec{x}) = F(\vec{x})_j$ and F is $wm(\overrightarrow{\mathcal{M}_\sigma}; \mathcal{M}_\iota)$ and $1 \leq j \leq K[\iota]$ then G is $wm(\overrightarrow{\mathcal{M}_\sigma}; \mathbb{N})$ and if $j = 1$ then for all $p \in \{1, \dots, k\}$: G is strict in argument p if F is.*

Proof Sketch. We easily see that in each case, F or G is in the given function space. To show weak monotonicity, assume given both $\vec{x}$ and $\vec{y}$ such that each $x^i \sqsupseteq y^i$; we then check for all cases that $F(\vec{x}) \sqsupseteq F(\vec{y})$, or $G(\vec{x}) \geq G(\vec{y})$. For the strictness conditions, we assume that $x^p \sqsupset y^p$ and similarly check all cases. ◀

The reader may recognise items (4–6): these largely correspond to the sufficient conditions for a weakly monotonic function S in Lemma 9. For the function f in item (6), we could for instance choose $+$, $*$ or max, where $+$ is strict in all arguments. However, we can get beyond Lemma 9 by using the other items; for example, applying variables to each other.

Now, if a function f is $\mathrm{wm}(\overrightarrow{\mathcal{M}_\sigma}; \mathcal{M}_\tau)$ and f is strict in all its arguments, then we easily see that the function $d_1 \mapsto \cdots \mapsto d_k \mapsto f(d_1, \dots, d_k)$ is an element of $\mathcal{M}_{\sigma_1 \Rightarrow \dots \Rightarrow \sigma_k \Rightarrow \tau}$. To illustrate how this can be used in practice, we show monotonicity of $\mathcal{J}_{\mathsf{map}}$ of Example 18:

▶ **Example 24.** Suppose $\mathcal{J}_{\mathsf{map}}(F, q) = (\ F(\langle q_{\mathsf{c}}, q_{\mathsf{m}}\rangle)_{\mathsf{c}} + q_{\mathsf{l}} * F(\langle q_{\mathsf{c}}, q_{\mathsf{m}}\rangle)_{\mathsf{c}} + q_{\mathsf{l}} + 1\ ,\ q_{\mathsf{l}}\ ,\ F(\langle q_{\mathsf{c}}, q_{\mathsf{m}}\rangle)_{\mathsf{l}}\ )$. By (5), the functions $(F, q) \mapsto q_i$ are $\mathrm{wm}(\mathcal{M}_{\mathsf{nat}\Rightarrow\mathsf{nat}}, \mathcal{M}_{\mathsf{list}}; \mathbb{N})$ for $i \in \{\mathsf{c}, \mathsf{l}, \mathsf{m}\}$ and moreover, $(F, q) \mapsto q_{\mathsf{c}}$ is strict in argument 2. Hence, by (3), $(F, q) \mapsto \langle q_{\mathsf{c}}, q_{\mathsf{m}}\rangle$ is $\mathrm{wm}(\mathcal{M}_{\mathsf{nat}\Rightarrow\mathsf{nat}}, \mathcal{M}_{\mathsf{list}}; \mathcal{M}_{\mathsf{nat}})$ and strict in argument 2. Therefore, by (2), $(F, q) \mapsto F(\langle q_{\mathsf{c}}, q_{\mathsf{m}}\rangle)$ is $\mathrm{wm}(\mathcal{M}_{\mathsf{nat}\Rightarrow\mathsf{nat}}, \mathcal{M}_{\mathsf{list}}; \mathcal{M}_{\mathsf{nat}})$ and strict in both arguments. Hence, by (7), $(F, q) \mapsto F(\langle q_{\mathsf{c}}, q_{\mathsf{m}}\rangle)_{\mathsf{c}}$ and $(F, q) \mapsto F(\langle q_{\mathsf{c}}, q_{\mathsf{m}}\rangle)_{\mathsf{l}}$ are $\mathrm{wm}(\mathcal{M}_{\mathsf{nat}\Rightarrow\mathsf{nat}}, \mathcal{M}_{\mathsf{list}}; \mathbb{N})$ and the former is strict in both arguments.

Continuing like this, it is not hard to see how we can iteratively prove that $(F, q) \mapsto (\ F(\langle q_{\mathsf{c}}, q_{\mathsf{m}}\rangle)_{\mathsf{c}} + q_{\mathsf{l}} * F(\langle q_{\mathsf{c}}, q_{\mathsf{m}}\rangle)_{\mathsf{c}} + q_{\mathsf{l}} + 1\ ,\ q_{\mathsf{l}}\ ,\ F(\langle q_{\mathsf{c}}, q_{\mathsf{m}}\rangle)_{\mathsf{l}}\ )$ is $\mathrm{wm}(\mathcal{M}_{\mathsf{nat}\Rightarrow\mathsf{nat}}, \mathcal{M}_{\mathsf{list}}; \mathcal{M}_{\mathsf{list}})$ and strict in both arguments, which immediately gives $\mathcal{J}_{\mathsf{map}} \in \mathcal{M}_{(\mathsf{nat}\Rightarrow\mathsf{nat})\Rightarrow\mathsf{list}\Rightarrow\mathsf{list}}$.

In practice, it is usually not needed to write such an elaborate proof: Lemma 23 essentially tells us that if a function is built exclusively using variables and variable applications, projections $F(\vec{x})_j$, constants, and weakly monotonic operators over the natural numbers, then that function is weakly monotonic; we only need to check that the cost component indeed increases if one of the variables x^i is increased.

Unfortunately, while Lemma 23 is useful for rules like the ones for map, it is not enough to handle functions like foldl, where the same function is repeatedly applied on a term. As foldl-like functions occur more often in higher-order rewriting, we should also address this.

To handle iteration, we define: for a function $Q \in A \Longrightarrow A$ and natural number n, let $Q^n(a)$ indicate repeated function application; that is, $Q^0(a) = a$ and $Q^{n+1}(a) = Q^n(Q(a))$.

▶ **Lemma 25.** *Suppose F is $wm(\overrightarrow{\mathcal{M}_\sigma}, \mathcal{M}_{\tau\Rightarrow\tau})$ and G is $wm(\overrightarrow{\mathcal{M}_\sigma}; \mathbb{N})$. Suppose that for all $u^1 \in \mathcal{M}_{\sigma_1}, \ldots, u^k \in \mathcal{M}_{\sigma_k}$ and $v \in \mathcal{M}_\tau$ we have: $F(u^1, \ldots, u^k, v) \sqsupseteq_\tau v$. Then the function $(x^1, \ldots, x^k) \mapsto F(x^1, \ldots, x^k)^{G(x^1,\ldots,x^k)}$ is $wm(\overrightarrow{\mathcal{M}_\sigma}, \mathcal{M}_{\tau\Rightarrow\tau})$.*

With this in hand, we can orient the foldl rules of Example 2.

▶ **Example 26.** For $F \in \mathcal{M}_{\mathsf{nat}\Rightarrow\mathsf{nat}\Rightarrow\mathsf{nat}}$ and $x, y \in \mathcal{M}_{\mathsf{nat}}$, let *Helper* be defined by:

$$\mathit{Helper}(F, y, x) = \langle F(x, y)_{\mathsf{c}},\ \max(x_{\mathsf{s}}, F(x, y)_{\mathsf{s}})\rangle.$$

Then *Helper* is $\mathrm{wm}(\mathcal{M}_{\mathsf{nat}\Rightarrow\mathsf{nat}\Rightarrow\mathsf{nat}}, \mathcal{M}_{\mathsf{nat}}, \mathcal{M}_{\mathsf{nat}}; \mathcal{M}_{\mathsf{nat}})$ and strict in its third argument by Lemma 23(1,2,3,6,7), Hence, *Helper* is $\mathrm{wm}(\mathcal{M}_{\mathsf{nat}\Rightarrow\mathsf{nat}\Rightarrow\mathsf{nat}}, \mathcal{M}_{\mathsf{nat}}; \mathcal{M}_{\mathsf{nat}\Rightarrow\mathsf{nat}})$. Since, in general, $\mathtt{costof}_{\mathsf{nat}}(F(x, y)) \geq \mathtt{costof}_{\mathsf{nat}}(x)$, we have $\mathit{Helper}(F, y, x) \sqsupseteq_{\mathsf{nat}} x$. Using Lemma 25, we therefore see that the function $(F, z, xs) \mapsto \mathit{Helper}(F, \langle xs_{\mathsf{c}}, xs_{\mathsf{m}}\rangle)^{xs_{\mathsf{l}}}(z)$ is weakly monotonic, and strict in its second argument. This ensures that the following function is in $\mathcal{M}$.

$$[\![\mathsf{foldl}(F, z, xs)]\!] = \mathit{Helper}(F, \langle xs_{\mathsf{c}}, xs_{\mathsf{m}}\rangle)^{xs_{\mathsf{l}}}(\langle 1 + xs_{\mathsf{c}} + xs_{\mathsf{l}} + F(0_{\mathsf{nat}}, 0_{\mathsf{nat}})_{\mathsf{c}} + z_{\mathsf{c}}, z_{\mathsf{s}}\rangle)$$

This interpretation function is compatible with the rules for foldl in Example 2. First, we have $[\![\mathsf{foldl}(F, z, \mathsf{nil})]\!] = \langle\ 1 + F(0_{\mathsf{nat}}, 0_{\mathsf{nat}})_{\mathsf{c}} + z_{\mathsf{c}},\ z_{\mathsf{s}}\ \rangle \sqsupseteq_{\mathsf{nat}} \langle z_{\mathsf{c}}, z_{\mathsf{s}}\rangle = z$, which orients the first rule. For the second, we will use the general property that (**) $F(\mathtt{addc}(n, x), y) \sqsupseteq \mathtt{addc}(n, F(x, y))$ (Lemma B.6). We denote $A := \langle x_{\mathsf{c}} + xs_{\mathsf{c}}, \max(x_{\mathsf{s}}, xs_{\mathsf{m}})\rangle$ and $B := 1 + xs_{\mathsf{c}} + xs_{\mathsf{l}} + F(0_{\mathsf{nat}}, 0_{\mathsf{nat}})_{\mathsf{c}} + z_{\mathsf{c}}$. Then we have $[\![\mathsf{foldl}(F, z, x : xs)]\!] = \mathit{Helper}(F, A)^{xs_{\mathsf{l}}+1}(\langle B + x_{\mathsf{c}} + 1, z_{\mathsf{s}}\rangle)$, which:

$$\begin{array}{ll}
\sqsupseteq_{\mathsf{nat}} & \mathit{Helper}(F, A)^{xs_{\mathsf{l}}}(\mathit{Helper}(F, A, \langle B, z_{\mathsf{s}}\rangle)) \text{ because } \langle B + x_{\mathsf{c}} + 1, z_{\mathsf{s}}\rangle \sqsupseteq_{\mathsf{nat}} \langle B, z_{\mathsf{s}}\rangle \\
\sqsupseteq_{\mathsf{nat}} & \mathit{Helper}(F, A)^{xs_{\mathsf{l}}}(F(\langle B, z_{\mathsf{s}}\rangle, A)) \text{ because } \mathit{Helper}(F, n, m) \sqsupseteq_{\mathsf{nat}} F(m, n) \\
\sqsupseteq_{\mathsf{nat}} & \mathit{Helper}(F, \langle xs_{\mathsf{c}}, xs_{\mathsf{m}}\rangle)^{xs_{\mathsf{l}}}(F(\langle B, z_{\mathsf{s}}\rangle, x)) \text{ because } A \sqsupseteq_{\mathsf{nat}} \langle xs_{\mathsf{c}}, xs_{\mathsf{m}}\rangle \text{ and } A \sqsupseteq_{\mathsf{nat}} x \\
\sqsupseteq_{\mathsf{nat}} & \mathit{Helper}(F, \langle xs_{\mathsf{c}}, xs_{\mathsf{m}}\rangle)^{xs_{\mathsf{l}}}(\mathtt{addc}_{\mathsf{nat}}(1 + xs_{\mathsf{c}} + xs_{\mathsf{l}} + F(0_{\mathsf{nat}}, 0_{\mathsf{nat}})_{\mathsf{c}}, F(z, x))) \text{ by } (**) \\
= & [\![\mathsf{foldl}(F, (F\ z\ x), xs)]\!].
\end{array}$$

The interpretation in Example 26 may *seem* too convoluted for practical use: it does not obviously tell us something like "F is applied a linear number of times on terms whose size is bounded by n". However, its value becomes clear when we plug in specific bounds for F.

▶ **Example 27.** The function sum, defined in Example 1, could alternatively be defined in terms of foldl: let $\mathsf{sum}(xs) \to \mathsf{foldl}(\lambda xy.(x \oplus y), 0, xs)$. To find an interpretation for this function, we use the interpretation functions for 0, s, nil, cons and $\oplus$ from Example 10. Then $[\![\lambda xy.(x \oplus y)]\!] = d, e \mapsto (d_{\mathsf{c}} + e_{\mathsf{c}} + e_{\mathsf{s}} + 3, d_{\mathsf{s}} + e_{\mathsf{s}})$. We easily see that $\mathit{Helper}([\![\lambda xy.(x \oplus y)]\!], \langle xs_{\mathsf{c}}, xs_{\mathsf{m}}\rangle, z) = \langle z_{\mathsf{c}} + xs_{\mathsf{c}} + xs_{\mathsf{m}} + 3, z_{\mathsf{s}} + xs_{\mathsf{m}}\rangle$. Importantly, the iteration variable z is used in a very innocent way: although its size is increased, this increase is by the same number (xs_{m}) in every iteration step. Moreover, the length of z does not affect the evaluation

cost. Hence, we can choose $[\![\mathsf{sum}(xs)]\!] = \langle 5 + xs_{\mathsf{c}} + xs_{\mathsf{l}} + xs_{\mathsf{l}} * (xs_{\mathsf{c}} + xs_{\mathsf{m}} + 3), xs_{\mathsf{l}} * xs_{\mathsf{m}} \rangle$. This is close to the interpretation from Example 10 but differs both in a small overhead for the β-reductions, and because our interpretation of foldl slightly overestimates the true cost.

This approach can be used to obtain bounds for any function that may be defined in terms of foldl, which includes many first-order functions. For example, with a small change to the signature of foldl, we could let $\mathsf{rev}(xs) = \mathsf{foldl}(\lambda xy.(y : x), \mathsf{nil}, xs)$; however, this would necessitate corresponding changes in the interpretation of foldl.

5 Finding complexity bounds

A key notion in complexity analysis of first-order rewriting is *runtime complexity*. In this section, we will define a conservative notion of runtime complexity for higher-order term rewriting, and show how our interpretations can be used to find runtime complexity bounds.

In first-order (and many-sorted) term rewriting, a *defined symbol* is any function symbol f such that there is a rule $\mathsf{f}(\ell_1, \ldots, \ell_k) \to r$ in the system; all other symbols are called *constructors*. A *ground constructor term* is a ground term without defined symbols. A *basic term* has the form $\mathsf{f}(s_1, \ldots, s_k)$ with f a defined symbol and $s_1, \ldots, s_k$ all ground constructor terms. The *runtime complexity* of a TRS is then a function φ in $(\mathbb{N} \setminus \{0\}) \Longrightarrow \mathbb{N}$ that maps each n to a number $\varphi(n)$ so that for every basic term s of size at most n: $\mathtt{dh}_{\mathcal{R}}(s) \leq \varphi(n)$.

The comparable notion of *derivational complexity* considers the derivation height for arbitrary ground terms of size n, but we will not use that here, since it can often give very high bounds that are not necessarily representative for realistic use of the system. In practice, a computation with a TRS would typically start with a main function, which takes *data* (e.g., natural numbers, lists) as input. This is exactly a basic term. Hence, the notion of runtime complexity roughly captures the worst-case number of steps for a realistic computation.

It is not obvious how this notion translates to the higher-order setting. It may be tempting to literally apply the definition to an AFS, but a "ground constructor term" (or perhaps "closed constructor term") is not a natural concept in higher-order rewriting; it does not intuitively capture data. Moreover, we would like to create a *robust* notion which can be extended to simple functional programming languages, so which is not subject to minor language difference like whether partial application of function symbols is allowed.

Instead, there are two obvious ways to capture the idea of input in higher-order rewriting:

- *closed irreducible terms*; this includes all ground constructor terms, but also for instance $\lambda x.\mathsf{0} \oplus x$ (but not $\lambda x.x \oplus \mathsf{0}$, since this can be rewritten following the rules in Example 1);
- *data*: this includes only ground constructor terms with no higher-order subterms.

As we observed in Example 12, the size of a higher-order term does not capture its behaviour. Hence, a notion of runtime complexity using closed irreducible terms is not obviously meaningful – and might be closer to *derivational* complexity due to defined symbols inside abstractions. Therefore, we here take the conservative choice and consider *data*.

▶ **Definition 28.** *In an AFS $(\mathcal{F}, \mathcal{R})$, a* data constructor *is a function symbol $\mathsf{c} :: [\iota_1 \times \cdots \times \iota_k] \Rightarrow \iota_0$ with each $\iota_i \in \mathcal{S}$, such that there is no rule of the form $\mathsf{c}(\ell_1, \ldots, \ell_k) \to r$. A* data term *is a term $\mathsf{c}(d_1, \ldots, d_k)$ such that c is a constructor and all d_i are also data terms.*

In practice, a sort is defined by its data constructors. For example, nat is defined by 0 and s, and list by nil and cons. In typical examples of first- and higher-order term rewriting systems, rules are defined to exhaustively pattern match on all constructors for a sort.

With this definition, we can conservatively extend the original notion of runtime complexity to be applicable to both many-sorted and higher-order term rewriting.

▶ **Definition 29.** *A* basic term *is a term of the form* $\mathsf{f}(d_1,\dots,d_k)$ *with all* d_i *data terms and* f *not a data constructor. We let* $|d|$ *denote the total number of symbols in a basic term* d*.*

The runtime complexity *of an AFS is a function* $\varphi \in (\mathbb{N}\setminus\{0\}) \Longrightarrow \mathbb{N}$ *so that for all* n *and basic terms* d*, with* $|d| \leq n$*:* $\mathtt{dh}_{\mathcal{R}}(d) \leq \varphi(n)$*.*

Note that if $\mathsf{f}(d_1,\dots,d_k)$ is a basic term, then $\mathsf{f} :: [\iota_1 \times \cdots \times \iota_k] \Rightarrow \tau$ with all ι_i sorts. Hence, higher-order runtime complexity considers the same (first-order) notion of basic terms as the first-order case; terms such as $\mathsf{map}(F,s)$ or even $\mathsf{map}(\lambda x.\mathsf{s}(x),\mathsf{nil})$ are not basic. One might reasonably question whether such a first-order notion is useful when studying the complexity of *higher-order* term rewriting. However, we argue that it is: runtime complexity aims to address the length of computations that begin at a typical starting point. When performing a *full program* analysis of an AFS, the computation will still typically start in a basic term, for instance; the entry-point symbol main applied to some user input $d_1,\dots,d_k$.

▶ **Example 30.** We consider an AFS from the Termination Problem Database, v11.0 [16].

$$\begin{array}{rcl@{\qquad}rcl} x \oplus \mathsf{0} & \to_{\mathcal{R}} & x & \mathsf{rec}(\mathsf{0},y,F) & \to_{\mathcal{R}} & y \\ x \oplus \mathsf{s}(y) & \to_{\mathcal{R}} & \mathsf{s}(x \oplus y) & \mathsf{rec}(\mathsf{s}(x),y,F) & \to_{\mathcal{R}} & F \cdot x \cdot \mathsf{rec}(x,y,F) \\ & & & x \otimes y & \to_{\mathcal{R}} & \mathsf{rec}(y,\mathsf{0},\lambda n.\lambda m.x \oplus m) \end{array}$$

Here, $\mathsf{rec} :: [\mathsf{nat} \times \mathsf{nat} \times (\mathsf{nat} \Rightarrow \mathsf{nat} \Rightarrow \mathsf{nat})] \Rightarrow \mathsf{nat}$. The only basic terms have the form $\mathsf{s}^n(\mathsf{0}) \oplus \mathsf{s}^m(\mathsf{0})$ or $\mathsf{s}^n(\mathsf{0}) \otimes \mathsf{s}^m(\mathsf{0})$. Using our method, we obtain cubic runtime complexity; to be precise: $\mathcal{O}(m^2 * n)$. The interpretation functions are found in Appendix A.

To derive runtime complexity for both first- and higher-order rewriting, our approach is to consider bounds for the functions $\mathcal{J}_{\mathsf{f}}$; we only need to consider the first-order symbols f.

▶ **Definition 31.** *Let* $P \in \mathcal{M}_{\iota_1 \Rightarrow \dots \Rightarrow \iota_m \Rightarrow \kappa}$ *be of the form* $P(x^1,\dots,x^m) = \langle P_1(x^1,\dots,x^m), \dots, P_{K[\kappa]}(x^1,\dots,x^m)\rangle$*. Then* P *is* linearly bounded *if each component function* P_l *of* P *is upper-bounded by a positive linear polynomial, i.e., there is a constant* $a \in \mathbb{N}$ *such that* $P_l(x^1,\dots,x^m) \leq a*(1+\sum_{i=1}^{m}\sum_{j=1}^{K[\iota_i]} x_j^i)$*. We say that* P *is* additive *if there exists a constant* $a \in \mathbb{N}$ *such that* $\sum_{l=1}^{K[\kappa]} P_l(x^1,\dots,x^m) \leq a + \sum_{i=1}^{m}\sum_{j=1}^{K[\iota_i]} x_j^i$*.*

By this definition, P_l is not required to be a linear function, only to be bounded by one. This means that for instance $\min(x_j^i, 2 * x_b^a)$ can be used, but $x_j^i * x_b^a$ cannot. It is easily checked that all the data constructors in this paper have an additive interpretation. For example, for $\mathcal{J}_{\mathsf{cons}}$: $(x_{\mathsf{c}} + xs_{\mathsf{c}}) + (x_{\mathsf{l}} + 1) + \max(x_{\mathsf{s}}, xs_{\mathsf{m}}) \leq 1 + x_{\mathsf{c}} + x_{\mathsf{s}} + xs_{\mathsf{c}} + xs_{\mathsf{l}} + xs_{\mathsf{s}}$.

▶ **Lemma 32.** *Let* $(\mathcal{F},\mathcal{R})$ *be an AFS or TRS that is compatible with a strongly monotonic algebra with interpretation function* $\mathcal{J}$*. Then:*

1. *if* $\mathcal{J}_{\mathsf{c}}$ *is additive for all data constructors* c*, then there exists a constant* $b > 0$ *in* $\mathbb{N}$ *so that for all data terms* s*: if* $|s| \leq n$ *then* $[\![s]\!]_l \leq b * n$*, for each component* $[\![s]\!]_l$ *of* $[\![s]\!]$*;*
2. *if* $\mathcal{J}_{\mathsf{c}}$ *is linearly bounded for all data constructors* c*, then there exists a constant* $b > 0$ *in* $\mathbb{N}$ *so that for all data terms* s*: if* $|s| \leq n$ *then* $[\![s]\!]_l \leq 2^{b*n}$*, for each component* $[\![s]\!]_l$ *of* $[\![s]\!]$*.*

By using Lemma 32, we quickly obtain some ways to bound runtime complexity:

▶ **Corollary 33.** *Let* $(\mathcal{F},\mathcal{R})$ *be an AFS or TRS that is compatible with a strongly monotonic algebra with interpretation function* $\mathcal{J}$*, and let* $\mathcal{F}_C$ *denote its set of data constructors, and* $\mathcal{F}_B$ *the set of all other symbols* f *with a signature* $\mathsf{f} :: [\iota_1 \times \cdots \times \iota_m] \Rightarrow \tau$*. Then:*

- *if* $\mathcal{J}_{\mathsf{f}}$ *is additive for all* $\mathsf{f} \in \mathcal{F}_C \cup \mathcal{F}_B$*, then* $(\mathcal{F},\mathcal{R})$ *has linear runtime complexity;*
- *if* $\mathcal{J}_{\mathsf{c}}$ *is additive for all* $\mathsf{c} \in \mathcal{F}_C$ *and for all* $\mathsf{f} \in \mathcal{F}_B$*,* $\mathcal{J}_{\mathsf{f}}(\vec{x}) = (P_1(\vec{x}),\dots,P_k(\vec{x}))$ *where* P_1 *is bounded by a polynomial, then* $(\mathcal{F},\mathcal{R})$ *has polynomial runtime complexity;*
- *if* $\mathcal{J}_{\mathsf{f}}$ *is linearly bounded for all* $\mathsf{f} \in \mathcal{F}_C \cup \mathcal{F}_B$*, then* $(\mathcal{F},\mathcal{R})$ *has exponential runtime complexity.*

We could easily use these results as part of an automatic complexity tool – and indeed, combine them with other methods for complexity analysis. However, this is not truly our goal: runtime complexity is only a part of the picture, especially in higher-order term rewriting where we may want to analyse modules that get much more hairy input. Our technique aims to give more fine-grained information, where we consider the impact of input with certain properties – like the length of a list or the depth of a tree. For this, the person interested in the analysis should be the one to decide on the interpretations of the constructors.

With this information given, though, it should be possible to automatically find interpretations for the other functions. The search for the best strategy requires dedicated research, which we leave to future work; however, we expect Lemmas 23 and 25 to play a large role. We also note that while the cost component may depend on the other components, the other components (which represent a kind of size property) typically do not depend on the cost.

6 On Related Work

Rewriting. There are several first-order complexity techniques based on interpretations. For example, in [11], the consequences of using additive, linear, and polynomial interpretations to the natural numbers are investigated; and in [26], context-dependent interpretations are introduced, which map terms to real numbers to obtain tighter bounds. Most closely related to our approach are *matrix interpretations* [21, 35], and a technique by the first author for complexity analysis of conditional term rewriting [31]. In both cases, terms are mapped to tuples as they are in our approach, although neither considers sort information, and matrix interpretations use linear interpretation functions. Our technique is a generalisation of both.

Higher-order Rewriting. In *higher-order* term rewriting (but a formalism without λ-abstraction), Baillot and Dal Lago [10] develop a version of higher-order polynomial interpretations which, like the present work, is based on v.d. Pol's higher-order interpretations [39]. In similar ways to our Section 5, the authors enforce polynomial bounds on derivational complexity by imposing restrictions on the shape of interpretations. Their method differs from ours in various ways, most importantly by mapping terms to $\mathbb{N}$ rather than tuples. In addition, the interpretations are limited to higher-order polynomials. This yields an ordering with the subterm property (i.e., $\mathsf{f}(\dots, s, \dots) \sqsupset s$), which means that TRSs like Example 11 cannot be handled. Moreover, it is not possible to find a general interpretation for functions like foldl or rec; the method can only handle instances of foldl with a linear function.

Beyond this, it unfortunately seems that relatively little work has thus far been done on complexity analysis of higher-order term rewriting. However, complexity of *functional programs* is an active field of research with a close relation to higher-order term rewriting.

Functional Programming. There are various techniques to statically analyse resource use of functional programs. These may be fully automated [5, 9, 42], semi-automated designed to reason about programmer specified-bounds [45, 15, 23], or even manual techniques, integrated with type system or program logic semantics [14, 17]. We discuss the most pertinent ones.

An approach using rewriting for full-program analysis is to translate functional programs to TRSs [6], which can be analysed using first-order complexity techniques. This takes advantage of the large body of work on first-order complexity, but loses information; the transformation often yields a system that is harder to analyse than the original.

The research methodology in most studies in functional programming differs significantly from rewriting techniques. Nevertheless, there are some studies with clear connections to our approach; in particular our separation of cost and size (and other structural properties). Most

relevant, in [19] the authors use a similar approach by giving semantics to a complexity-aware intermediate language allowing arbitrary user-defined notions for size – such as list length or maximum element size; recurrence relations are then extracted to represent the complexity.

Additionally, most modern complexity analysis is done via enhancements at the type system level [2, 5, 28, 40, 23, 20]. For example, types may be annotated with a counter, the heap size or a data type's size measure. Notably, a line of work on Resource-Aware ML [28, 37, 30] studies resource use of OCaml programs with methods based on Tarjan's amortized analysis [43]. Types are annotated with *potentials* (a cost measure), and type inference generates a set of linear constraints which is sent over to an external solver. For Haskell, Liquid Haskell [41, 44] provides a language to annotate types, which can be used to prove properties of the program; this was recently extended to include complexity [23]. Unlike RAML, this approach is not fully automatic: type annotations are checked, not derived.

These works in functional programming have a different purpose from ours: they study the resource use in a specific language, typically with a fixed evaluation strategy. Our method, in contrast, allows for arbitrary evaluation, which could be specified to various strategies in future work. Moreover, most of these works limit interest to full-program analysis. We do this for runtime complexity, but our method offers more, by providing general interpretations for individual functions like map or foldl. Similarly, most of these works impose additive type annotations for the constructors; we do not restrict the constructor interpretations outside Lemma 32. On the other hand, many do consider (shallow) polymorphism, which we do not.

While in functional programming one considers resource usage [40, 28], rewriting is concerned with the number of steps, which can be translated to a form of resource measure if the true cost of each step is kept low. This is achieved by imposing restrictions on reduction strategy and term representation [1, 18]. Our approach carries the blessing of being general and machine independent and the curse of not necessarily being a reasonable cost model.

7 Conclusion and Future Work

In this paper, we have introduced tuple interpretations for many-sorted and higher-order term rewriting. This includes providing a new definition of strongly monotonic algebras, a compatibility theorem, a function *MakeSM* that orients β- and η-reductions, and several lemmas to prove monotonicity of interpretation functions. We also show that for certain restrictions on interpretation functions, we find linear, polynomial or exponential bounds on runtime complexity (for a simple but natural definition of higher-order runtime complexity).

Our type-based, semantical approach allows us to relate various "size" notions (e.g., list length, tree depth, term size. etc.) to reduction cost, and thus offers a more fine-grained analysis than traditional notions like runtime complexity. Most importantly, we can express the complexity of a higher-order function in terms of the behaviour of its (function) arguments. In the future, we hope that this could be used towards a truly higher-order complexity notion.

Some further examples and weaknesses. Aside from the three higher-order examples in this paper, we have successfully applied our method to a variety of higher-order benchmarks in the Termination Problem Database [16], all with additive interpretations for the constructors. Two additional examples (filter and deriv) are included in Appendix A.

A clear weakness we discovered was that our method can only handle "plain function-passing" systems [33]. That is, we typically do not succeed on systems where a variable of function type occurs inside a subterm of base type, and occurs outside this subterm in the right-hand side. Examples of such systems are ordrec, which has a rule $\mathsf{ordrec}(\mathsf{lim}(F), x, G, H) \to_{\mathcal{R}} H \cdot F \cdot (\lambda n.\mathsf{ordrec}(F \cdot n, x, G, H))$ with $\mathsf{lim} :: [\mathsf{nat} \Rightarrow \mathsf{ord}] \Rightarrow \mathsf{ord}$, and apply, which has a rule $\mathsf{lapply}(x, \mathsf{fcons}(F, xs)) \to_{\mathcal{R}} F \cdot \mathsf{lapply}(x, xs)$ with $\mathsf{fcons} :: [(\mathsf{a} \Rightarrow \mathsf{a}) \times \mathsf{listf}] \Rightarrow \mathsf{listf}$.

Future work. We intend to consider the effect of different evaluation strategies, such as innermost evaluation, weak-innermost evaluation (where rewriting below an abstraction is not allowed, as is commonly the case in functional programming) or outermost evaluation. This extension is likely to be an important step towards another goal: to more closely relate our complexity notion to a reasonable measure of resource consumption in a rewriting engine.

In addition, we plan to extend first-order complexity techniques like dependency tuples [24], which may allow us to overcome the weakness described above. Another goal is to enrich our type system to support a notion of polymorphism and add polymorphic interpretations into the play. We also aim to develop a tool to automatically find suitable tuple interpretations.

References

1 B. Accatoli and U. Dal Lago. (leftmost-outermost) beta reduction is invariant, indeed. *LMCS*, 2016. doi:10.2168/LMCS-12(1:4)2016.

2 S. Alves, D. Kesner, and D. Ventura. A quantitative understanding of pattern matching. In *Proc. TYPES*, LIPIcs, 2020. doi:10.4230/LIPIcs.TYPES.2019.3.

3 T. Arai and G. Moser. Proofs of termination of rewrite systems for polytime functions. In *Proc. FSTTCS*, 2005. doi:10.1007/11590156_4.

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

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

6 M. Avanzini, U. Dal Lago, and G. Moser. Analysing the complexity of functional programs: Higher-order meets first-order. In *Proc. ICFP*, 2015. doi:10.1145/2784731.2784753.

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

8 M. Avanzini and G. Moser. Closing the gap between runtime complexity and polytime computability. In *Proc. RTA*, 2010. doi:10.4230/LIPIcs.RTA.2010.33.

9 Ralph B. Automated higher-order complexity analysis. *TCS*, 2004. doi:10.1016/j.tcs.2003.10.022.

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

11 G. Bonfante, A. Cichon, J. Marion, and H. Touzet. Complexity classes and rewrite systems with polynomial interpretation. In *Proc. CSL*, 1998. doi:10.1007/10703163_25.

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

13 M. Brockschmidt, F. Emmes, S. Falke, C. Fuhs, and J. Giesl. Alternating runtime and size complexity analysis of integer programs. In *Proc. TACAS*, 2014. doi:10.1007/978-3-642-54862-8_10.

14 Q. Carbonneaux, J. Hoffmann, T. Ramananandro, and Z. Shao. End-to-end verification of stack-space bounds for C programs. *SIGPLAN Not.*, 2014. doi:10.1145/2666356.2594301.

15 E. Çiçek, D. Garg, and U. Acar. Refinement types for incremental computational complexity. In *Proc. ESOP*, 2015. doi:10.1007/978-3-662-46669-8_17.

16 Community. Termination problem database, version 11.0. Directory Higher_Order_Rewriting_Union_Beta/Mixed_HO_10/, 2019. URL: http://termination-portal.org/wiki/TPDB.

17 U. Dal Lago and M. Gaboardi. Linear dependent types and relative completeness. In *Proc. LICS*, 2011. doi:10.1109/LICS.2011.22.

18 U. Dal Lago and S. Martini. Derivational complexity is an invariant cost model. In *Proc. FOPARA*, 2010. doi:10.1007/978-3-642-15331-0_7.

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

20 A. Das, S. Balzer, J. Hoffman, F. Pfenning, and I. Santurkar. Resource-aware session types for digital contracts, 2019. arXiv:1902.06056.

21 J. Endrullis, J. Waldmann, and H. Zantema. Matrix interpretations for proving termination of term rewriting. *JAR*, 2008. doi:10.1007/11814771_47.

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

23 M. A. T. Handley, N. Vazou, and G. Hutton. Liquidate your assets: Reasoning about resource usage in liquid haskell. *ACM POPL*, 2019. doi:10.1145/3371092.

24 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.

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

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

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

28 J. Hoffmann, K. Aehlig, and M. Hofmann. Resource aware ml. In *Proc. CAV*, 2012. doi:10.1007/978-3-642-31424-7_64.

29 J. Jouannaud and M. Okada. A computation model for executable higher-order algebraic specification languages. In *Proc. LICS*, 1991. doi:10.1109/LICS.1991.151659.

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

31 C. Kop, A. Middeldorp, and T. Sternagel. Complexity of conditional term rewriting. *LMCS*, 2017. doi:10.23638/LMCS-13(1:6)2017.

32 C. Kop and D. Vale. Tuple interpretations for higher-order complexity (extended), 2021. arXiv:2105.01112.

33 K. Kusakari and M. Sakai. Enhancing dependency pair method using strong computability in simply-typed term rewriting. *AAECC*, 2007. doi:10.1007/s00200-007-0046-9.

34 G. Moser. Derivational complexity of knuth-bendix orders revisited. In *Proc. LPAR*, 2006. doi:10.1007/11916277_6.

35 G. Moser, A. Schnabl, and J. Waldmann. Complexity analysis of term rewriting based on matrix and context dependent interpretations. In *Proc. FSTTCS*, 2008. doi:10.4230/LIPIcs.FSTTCS.2008.1762.

36 T. Nipkow. Higher-order critical pairs. In *Proc. LICS*, 1991. doi:10.1109/LICS.1991.151658.

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

38 E. Ohlebusch. *Advanced Topics in Term Rewriting*. Springer, 2002. doi:10.1007/978-1-4757-3661-8.

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

40 V. Rajani, M. Gaboardi, D. Garg, and J. Hoffmann. A unifying type-theory for higher-order (amortized) cost analysis. *ACM POPL*, 2021. doi:10.1145/3434308.

41 P. M. Rondon, M. Kawaguci, and R. Jhala. Liquid types. *SIGPLAN Not.*, 2008. doi:10.1145/1379022.1375602.

42 M. Sinn, F. Zuleger, and H. Veith. A simple and scalable static analysis for bound analysis and amortized complexity analysis. In *Proc. CAV*, 2014. doi:10.1007/978-3-319-08867-9_50.

43 R. E. Tarjan. Amortized computational complexity. *ADM*, 1985. doi:10.1137/0606031.

44 N. Vazou, P. M. Rondon, and R. Jhala. Abstract refinement types. In *Proc. ESOP*, 2013. doi:10.1007/978-3-642-37036-6_13.

45 P. Wang, D. Wang, and A. Chlipala. Timl: A functional language for practical complexity analysis with invariants. *ACM POPL*, 2017. doi:10.1145/3133903.

A Extended examples

Extrec. The system in Example 30 has the following interpretation:

$$\begin{array}{rcl}
[\![0]\!] & = & \langle 0,0\rangle \\
[\![\mathsf{s}(x)]\!] & = & \langle x_\mathsf{c}, x_\mathsf{s}+1\rangle \\
[\![x\oplus y]\!] & = & \langle x_\mathsf{c}+y_\mathsf{c}+y_\mathsf{s}+1, x_\mathsf{s}+y_\mathsf{s}\rangle \\
[\![x\otimes y]\!] & = & \langle 1+y_\mathsf{s}*(x_\mathsf{c}+y_\mathsf{c}+x_\mathsf{s}*(y_\mathsf{s}+1)/2+3), x_\mathsf{s}*y_\mathsf{s}\rangle \\
[\![\mathsf{rec}(x,y,F)]\!] & = & \mathit{Helper}(x,F)^{x_\mathsf{s}}(\ \langle 1+x_\mathsf{c}+y_\mathsf{c}+x_\mathsf{s}+F(\mathsf{0}_\mathsf{nat},\mathsf{0}_\mathsf{nat})_\mathsf{c}, y_\mathsf{s}\rangle\ ) \\
\mathit{Helper}(x,F) & = & z\mapsto \langle F(x,z)_\mathsf{c}, \max(z_\mathsf{s}, F(x,z)_\mathsf{s})\rangle
\end{array}$$

Then we always have (*A) $\mathit{Helper}(x,F)(z) \sqsupseteq_\mathsf{nat} z$ because $F(x,z)_\mathsf{c} \geq z_\mathsf{c}$ which we will see in Lemma B.8, and clearly $\max(z_\mathsf{s}, F(x,z)_\mathsf{s}) \geq z_\mathsf{s}$. Hence, the monotonicity requirements are satisfied. We also clearly have (*B) $\mathit{Helper}(x,F)(z) \sqsupseteq_\mathsf{nat} F(x,z)$, since clearly $\max(z_\mathsf{s}, F(x,z)_\mathsf{s}) \geq F(x,z)_\mathsf{s}$. For most rules, it is easy to see that $[\![\ell]\!] \sqsupset [\![r]\!]$. We only show:

- $[\![\mathsf{rec}(\mathsf{s}(x),y,F)]\!] \sqsupseteq_\mathsf{nat} [\![F\cdot x\cdot \mathsf{rec}(x,y,F)]\!]$:
  $[\![\mathsf{rec}(\mathsf{s}(x),y,F)]\!] = \mathit{Helper}(\langle x_\mathsf{c}, x_\mathsf{s}+1\rangle, F)^{x_\mathsf{s}+1}(\ \langle 1+x_\mathsf{c}+y_\mathsf{c}+(x_\mathsf{s}+1)+F(\mathsf{0}_\mathsf{nat},\mathsf{0}_\mathsf{nat})_\mathsf{c}, y_\mathsf{s}\rangle\ ) = \mathit{Helper}(\langle x_\mathsf{c}, x_\mathsf{s}+1\rangle, F)(\mathit{Helper}(\langle x_\mathsf{c}, x_\mathsf{s}+1\rangle, F)^{x_\mathsf{s}}(\ \langle 2+x_\mathsf{c}+y_\mathsf{c}+x_\mathsf{s}+F(\mathsf{0}_\mathsf{nat},\mathsf{0}_\mathsf{nat})_\mathsf{c}, y_\mathsf{s}\rangle\ )) \sqsupseteq_\mathsf{nat} F(\langle x_\mathsf{c}, x_\mathsf{s}+1\rangle, \mathit{Helper}(\langle x_\mathsf{c}, x_\mathsf{s}+1\rangle, F)^{x_\mathsf{s}}(\ \langle 2+x_\mathsf{c}+y_\mathsf{c}+x_\mathsf{s}+F(\mathsf{0}_\mathsf{nat},\mathsf{0}_\mathsf{nat})_\mathsf{c}, y_\mathsf{s}\rangle\ ))$ by (*B), $\sqsupseteq_\mathsf{nat} F(\langle x_\mathsf{c}, x_\mathsf{s}\rangle, \mathit{Helper}(\langle x_\mathsf{c}, x_\mathsf{s}\rangle, F)^{x_\mathsf{s}}(\ \langle 1+x_\mathsf{c}+y_\mathsf{c}+x_\mathsf{s}+F(\mathsf{0}_\mathsf{nat},\mathsf{0}_\mathsf{nat})_\mathsf{c}, y_\mathsf{s}\rangle\ ))$ by monotonicity, $= F(x, \mathit{Helper}(x,F)^{x_\mathsf{s}}(\ \langle 1+x_\mathsf{c}+y_\mathsf{c}+x_\mathsf{s}+F(\mathsf{0}_\mathsf{nat},\mathsf{0}_\mathsf{nat})_\mathsf{c}, y_\mathsf{s}\rangle\ )) = [\![F\cdot x\cdot \mathsf{rec}(x,y,F)]\!]$.
- $[\![x\otimes y]\!] \sqsupseteq_\mathsf{nat} [\![\mathsf{rec}(y,\mathsf{0},\lambda n.\lambda m.x\oplus m)]\!]$:
  - $[\![\lambda n.\lambda m.x\oplus m]\!] = n\mapsto m\mapsto \langle x_\mathsf{c}+n_\mathsf{c}+m_\mathsf{c}+m_\mathsf{s}+3, x_\mathsf{s}+m_\mathsf{s}\rangle$
  - $\mathit{Helper}(y, [\![\lambda n.\lambda m.x\oplus m]\!]) = m\mapsto \langle x_\mathsf{c}+y_\mathsf{c}+m_\mathsf{c}+m_\mathsf{s}+3, x_\mathsf{s}+m_\mathsf{s}\rangle$
  - For given i, $\mathit{Helper}(y, [\![\lambda n.\lambda m.x\oplus m]\!])^i(m)_\mathsf{s} = (\sum_{j=0}^{i} x_\mathsf{s}) + m_\mathsf{s} = x_\mathsf{s}*i+m_\mathsf{s}$
  - $\mathit{Helper}(y, [\![\lambda n.\lambda m.x\oplus m]\!])^{y_\mathsf{s}} = m\mapsto \langle \sum_{i=1}^{y_\mathsf{s}}(x_\mathsf{c}+y_\mathsf{c}+(x_\mathsf{s}*i+m_\mathsf{s})+3)+m_\mathsf{c}, y_\mathsf{s}*x_\mathsf{s}+m_\mathsf{s}\rangle = \langle y_\mathsf{s}*(x_\mathsf{c}+y_\mathsf{c}+m_\mathsf{s}+3)+x_\mathsf{s}*\sum_{i=1}^{y_\mathsf{s}}(i)+m_\mathsf{c}, y_\mathsf{s}*x_\mathsf{s}+m_\mathsf{s}\rangle = \langle y_\mathsf{s}*(x_\mathsf{c}+y_\mathsf{c}+m_\mathsf{s}+3)+x_\mathsf{s}*(y_\mathsf{s}*(y_\mathsf{s}+1)/2)+m_\mathsf{c}, y_\mathsf{s}*x_\mathsf{s}+m_\mathsf{s}\rangle = \langle y_\mathsf{s}*(x_\mathsf{c}+y_\mathsf{c}+m_\mathsf{s}+x_\mathsf{s}*(y_\mathsf{s}+1)/2+3)+m_\mathsf{c}, y_\mathsf{s}*x_\mathsf{s}+m_\mathsf{s}\rangle$

  Hence, $[\![x\otimes y]\!] = \langle 1+y_\mathsf{s}*(x_\mathsf{c}+y_\mathsf{c}+x_\mathsf{s}*(y_\mathsf{s}+1)/2+3), x_\mathsf{s}*y_\mathsf{s}\rangle \sqsupseteq_\mathsf{nat} \langle y_\mathsf{s}*(x_\mathsf{c}+y_\mathsf{c}+x_\mathsf{s}*(y_\mathsf{s}+1)/2+3)+0, x_\mathsf{s}*y_\mathsf{s}+0\rangle = [\![\mathsf{rec}(y,\mathsf{0},\lambda n.\lambda m.x\oplus m)]\!]$

Filter. We show an example from the Termination Problem Database, v11.0.

$$\begin{array}{rcl@{\qquad}rcl}
\mathsf{rand}(x) & \to_\mathcal{R} & x & \mathsf{filter}(F,\mathsf{nil}) & \to_\mathcal{R} & \mathsf{nil} \\
\mathsf{rand}(\mathsf{s}(x)) & \to_\mathcal{R} & \mathsf{rand}(x) & \mathsf{filter}(F,x:xs) & \to_\mathcal{R} & \mathsf{consif}(F\cdot x, x, \mathsf{filter}(F,xs)) \\
\mathsf{bool}(\mathsf{0}) & \to_\mathcal{R} & \mathsf{false} & \mathsf{consif}(\mathsf{true},x,xs) & \to_\mathcal{R} & x:xs \\
\mathsf{bool}(\mathsf{s}(\mathsf{0})) & \to_\mathcal{R} & \mathsf{true} & \mathsf{consif}(\mathsf{false},x,xs) & \to_\mathcal{R} & xs
\end{array}$$

We will use the notation q instead of xs to avoid clutter in the proof. We let $\mathcal{M}_\mathsf{nat} = \mathbb{N}^2$ and $\mathcal{M}_\mathsf{list} = \mathbb{N}^3$ as before, and additionally let $\mathcal{M}_\mathsf{boolean} = \mathbb{N}$ (so no size components). We let:

$$\begin{array}{rcl@{\qquad}rcl@{\qquad}rcl}
[\![\mathsf{true}]\!] & = & \langle 0\rangle & [\![\mathsf{s}(x)]\!] & = & \langle x_\mathsf{c}, x_\mathsf{s}+1\rangle & [\![\mathsf{bool}(x)]\!] & = & \langle x_\mathsf{c}+1\rangle \\
[\![\mathsf{false}]\!] & = & \langle 0\rangle & [\![\mathsf{nil}]\!] & = & \langle 0,0,0\rangle & [\![\mathsf{rand}(x)]\!] & = & \langle 1+x_\mathsf{c}+x_\mathsf{s}, x_\mathsf{s}\rangle \\
[\![\mathsf{0}]\!] & = & \langle 0,0\rangle & [\![x:q]\!] & = & \langle x_\mathsf{c}+q_\mathsf{c}, q_\mathsf{l}+1, \max(x_\mathsf{s}, q_\mathsf{m})\rangle & & & \\
[\![\mathsf{consif}(z,x,q)]\!] & = & \langle z_\mathsf{c}+x_\mathsf{c}+q_\mathsf{c}+1, q_\mathsf{l}+1, \max(x_\mathsf{s}, q_\mathsf{m})\rangle & & & & & & \\
[\![\mathsf{filter}(F,q)]\!] & = & \langle 1+(q_\mathsf{l}+1)*(2+q_\mathsf{c}+F(\langle q_\mathsf{c}, q_\mathsf{m}\rangle)_\mathsf{c}), q_\mathsf{l}, q_\mathsf{m}\rangle & & & & & &
\end{array}$$

It is easy to see that monotonicity requirements are satisfied. As for orienting the rules, we show only the second filter rule.

- $[\![\mathsf{filter}(F, x : q)]\!] \sqsupset_{\mathsf{list}} [\![\mathsf{consif}(F \cdot x, x, \mathsf{filter}(F, q))]\!]$
  $[\![\mathsf{filter}(F, x : q)]\!] = \langle 1+(q_{\mathsf{l}}+2)*(2+x_{\mathsf{c}}+q_{\mathsf{c}}+F(\langle x_{\mathsf{c}}+q_{\mathsf{c}}, \max(x_{\mathsf{s}}, q_{\mathsf{m}})\rangle)_{\mathsf{c}}), q_{\mathsf{l}}+1, \max(x_{\mathsf{s}}, q_{\mathsf{m}})\rangle =$
  $\langle 3+x_{\mathsf{c}}+q_{\mathsf{c}}+F(\langle x_{\mathsf{c}}+q_{\mathsf{c}}, \max(x_{\mathsf{s}}, q_{\mathsf{m}})\rangle)_{\mathsf{c}}+(q_{\mathsf{l}}+1)*(2+x_{\mathsf{c}}+q_{\mathsf{c}}+F(\langle x_{\mathsf{c}}+q_{\mathsf{c}}, \max(x_{\mathsf{s}}, q_{\mathsf{m}})\rangle)_{\mathsf{c}}), q_{\mathsf{l}}+$
  $1, \max(x_{\mathsf{s}}, q_{\mathsf{m}})\rangle \sqsupset_{\mathsf{list}} \langle 2+x_{\mathsf{c}}+F(\langle x_{\mathsf{c}}, x_{\mathsf{s}}\rangle)_{\mathsf{c}}+(q_{\mathsf{l}}+1)*(2+q_{\mathsf{c}}+F(\langle q_{\mathsf{c}}, q_{\mathsf{m}}\rangle)_{\mathsf{c}}), q_{\mathsf{l}}+1, \max(x_{\mathsf{s}}, q_{\mathsf{m}})\rangle$
  $= \langle F(x)_{\mathsf{c}}+x_{\mathsf{c}}+(1+(q_{\mathsf{l}}+1)*(2+q_{\mathsf{c}}+F(\langle q_{\mathsf{c}}, q_{\mathsf{m}}\rangle)_{\mathsf{c}}))+1, q_{\mathsf{l}}+1, \max(x_{\mathsf{s}}, q_{\mathsf{m}})\rangle = \langle F(x)_{\mathsf{c}}+x_{\mathsf{c}}+$
  $[\![\mathsf{filter}(F, q)]\!]_{\mathsf{c}}+1, [\![\mathsf{filter}(F, q)]\!]_{\mathsf{l}}+1, \max(x_{\mathsf{s}}, [\![\mathsf{filter}(F, q)]\!]_{\mathsf{m}})\rangle = [\![\mathsf{consif}(F \cdot x, x, \mathsf{filter}(F, q))]\!]$.

Deriv. Our final example also comes from the termination problem database.

$$\begin{array}{rcl}
\mathsf{der}(\lambda x.y) & \to_{\mathcal{R}} & \lambda z.0 \qquad \mathsf{der}(\lambda x.\mathsf{sin}(x)) \to_{\mathcal{R}} \lambda z.\mathsf{cos}(z) \\
\mathsf{der}(\lambda x.x) & \to_{\mathcal{R}} & \lambda z.1 \qquad \mathsf{der}(\lambda x.\mathsf{cos}(x)) \to_{\mathcal{R}} \lambda z.\mathsf{min}(\mathsf{cos}(z)) \\
\mathsf{der}(\lambda x.\mathsf{plus}(F \cdot x, G \cdot x)) & \to_{\mathcal{R}} & \lambda z.\mathsf{plus}(\mathsf{der}(F) \cdot z, \mathsf{der}(G) \cdot z) \\
\mathsf{der}(\lambda x.\mathsf{times}(F \cdot x, G \cdot x)) & \to_{\mathcal{R}} & \lambda z.\mathsf{plus}(\mathsf{times}(\mathsf{der}(F) \cdot z, G \cdot z), \mathsf{times}(F \cdot z, \mathsf{der}(G) \cdot z)) \\
\mathsf{der}(\lambda x.\mathsf{ln}(F \cdot x)) & \to_{\mathcal{R}} & \lambda z.\mathsf{div}(\mathsf{der}(F) \cdot z, F \cdot z)
\end{array}$$

With $\mathsf{der} :: [\mathsf{real} \Rightarrow \mathsf{real}] \Rightarrow \mathsf{real} \Rightarrow \mathsf{real}$. We let $\mathcal{M}_{\mathsf{real}} = \mathbb{N}^3$ where the first component indicates cost, and the second and third component roughly indicate the number of plus/times/ln occurrences and the number of times/ln occurrences respectively. We will denote x_{s} for x_2, and $x_\star$ for x_3. We use the following interpretation:

$$\begin{array}{rcl@{\qquad}rcl}
[\![0]\!] & = & \langle 0,0,0\rangle & [\![\mathsf{plus}(x,y)]\!] & = & \langle x_{\mathsf{c}}+y_{\mathsf{c}}, x_{\mathsf{s}}+y_{\mathsf{s}}+1, x_\star+y_\star\rangle \\
[\![1]\!] & = & \langle 0,0,0\rangle & [\![\mathsf{times}(x,y)]\!] & = & \langle x_{\mathsf{c}}+y_{\mathsf{c}}, x_{\mathsf{s}}+y_{\mathsf{s}}+1, x_\star+y_\star+1\rangle \\
[\![\mathsf{cos}(x)]\!] & = & x & [\![\mathsf{ln}(x)]\!] & = & \langle x_{\mathsf{c}}, x_{\mathsf{s}}+1, x_\star+1\rangle \\
[\![\mathsf{sin}(x)]\!] & = & x & [\![\mathsf{der}(F)]\!] & = & z \mapsto \langle \\
[\![\mathsf{min}(x)]\!] & = & \langle x_{\mathsf{c}}, 0, 0\rangle & & & \quad 1+F(z)_{\mathsf{c}}+2*F(z)_{\mathsf{s}}+F(z)_\star*F(z)_{\mathsf{c}}, \\
[\![\mathsf{div}(x,y)]\!] & = & \langle x_{\mathsf{c}}+y_{\mathsf{c}}, 0, 0\rangle & & & \quad F(z)_{\mathsf{s}}*(F(z)_\star+1), \\
& & & & & \quad F(z)_\star*(F(z)_\star+1)\ \rangle
\end{array}$$

It is easy to see that monotonicity requirements are satisfied. In addition, all the rules are oriented by this interpretation. We only show the one for times.

- $[\![\mathsf{der}(\lambda x.\mathsf{times}(F \cdot x, G \cdot x))]\!] \sqsupset_{\mathsf{real}} [\![\lambda z.\mathsf{plus}(\mathsf{times}(\mathsf{der}(F) \cdot z, G \cdot z), \mathsf{times}(F \cdot z, \mathsf{der}(G) \cdot z))]\!]$
  - $[\![\lambda x.\mathsf{times}(F \cdot x, G \cdot x))]\!] = x \mapsto \langle 1+F(x)_{\mathsf{c}}+G(x)_{\mathsf{c}}, F(x)_{\mathsf{s}}+G(x)_{\mathsf{s}}+1, F(x)_\star+G(x)_\star+1\rangle$
  - $[\![\mathsf{times}(\mathsf{der}(F) \cdot z, G \cdot z)]\!] = \langle 1+F(z)_{\mathsf{c}}+2*F(z)_{\mathsf{s}}+F(z)_\star*F(z)_{\mathsf{c}}+G(z)_{\mathsf{c}}, F(z)_{\mathsf{s}}*(F(z)_\star+1)+G(z)_{\mathsf{s}}+1, F(x)_\star*(F(z)_\star+1)+G(z)_\star+1\rangle$
  - $[\![\mathsf{times}(F \cdot z, \mathsf{der}(G) \cdot z))]\!] = \langle 1+G(z)_{\mathsf{c}}+2*G(z)_{\mathsf{s}}+G(z)_\star*G(z)_{\mathsf{c}}+F(z)_{\mathsf{c}}, G(z)_{\mathsf{s}}*(G(z)_\star+1)+F(z)_{\mathsf{s}}+1, G(z)_\star*(G(z)_\star+1)+F(z)_\star+1\rangle$

  $[\![\mathsf{der}(\lambda x.\mathsf{times}(F \cdot x, G \cdot x))]\!] = z \mapsto \langle 1+\mathrm{cost}, \mathrm{size}, \mathrm{star}\rangle$, where:
  - $\mathrm{cost} = (1+F(z)_{\mathsf{c}}+G(z)_{\mathsf{c}})+2*(F(z)_{\mathsf{s}}+G(z)_{\mathsf{s}}+1)+(F(z)_\star+G(z)_\star+1)*(1+F(z)_{\mathsf{c}}+G(z)_{\mathsf{c}})$;
  - $\mathrm{size} = (F(z)_{\mathsf{s}}+G(z)_{\mathsf{s}}+1)*(F(z)_\star+G(z)_\star+2)$;
  - $\mathrm{star} = (F(z)_\star+G(z)_\star+1)*(F(z)_\star+G(z)_\star+2)$.

  We have $\mathrm{size} = F(z)_{\mathsf{s}}+G(z)_{\mathsf{s}}+1+(F(z)_{\mathsf{s}}+G(z)_{\mathsf{s}}+1)*(F(z)_\star+G(z)_\star+1) \geq F(z)_{\mathsf{s}}+G(z)_{\mathsf{s}}+1+F(z)_{\mathsf{s}}*(F(z)_\star+1)+G(z)_{\mathsf{s}}*(G(z)_\star+1)+1*1 = (F(z)_{\mathsf{s}}*(F(z)_\star+1)+G(z)_{\mathsf{s}}+1)+(G(z)_{\mathsf{s}}*(G(z)_\star+1)+F(z)_{\mathsf{s}}+1) = [\![\mathsf{plus}(\mathsf{times}(\mathsf{der}(F) \cdot z, G \cdot z), \mathsf{times}(F \cdot z, \mathsf{der}(G) \cdot z))]\!]_{\mathsf{s}}$
  The proof that $\mathrm{star} \geq [\![\mathsf{plus}(\mathsf{times}(\mathsf{der}(F) \cdot z, G \cdot z), \mathsf{times}(F \cdot z, \mathsf{der}(G) \cdot z))]\!]_\star$ is the same, just with $\cdot_{\mathsf{s}}$ replaced by $\cdot_\star$.
  Finally, $\mathrm{cost} > F(z)_{\mathsf{c}}+G(z)_{\mathsf{c}}+2*F(z)_{\mathsf{s}}+2*G(z)_{\mathsf{s}}+2+1+F(z)_{\mathsf{c}}+G(z)_{\mathsf{c}}+(F(z)_\star+G(z)_\star)*(F(z)_{\mathsf{c}}+G(z)_{\mathsf{c}}) = 1+1+F(z)_{\mathsf{c}}+2*F(z)_{\mathsf{s}}+F(z)_\star*F(z)_{\mathsf{c}}+G(z)_{\mathsf{c}}+1+G(z)_{\mathsf{c}}+2*G(z)_{\mathsf{s}}+G(z)_\star*G(z)_{\mathsf{c}}+F(z)_{\mathsf{c}} = 1+[\![\mathsf{plus}(\mathsf{times}(\mathsf{der}(F) \cdot z, G \cdot z), \mathsf{times}(F \cdot z, \mathsf{der}(G) \cdot z))]\!]_{\mathsf{c}}$

B Proof sketches and unstated lemmas

We here present proof sketches for lemmas in the text where they were omitted, as well as unstated lemmas that for instance support the correctness of our definition. Complete proofs can be found in the extended appendix [32].

Proof Sketch of Lemma 14. Each individual statement follows by induction on σ. ◀

In the text, we quietly asserted that Definition 16 is well-defined. This follows from:

▶ **Lemma B.1.** *For all terms $s :: \sigma$ and suitable α as described in Definition 16 we have: $[\![s]\!]_\alpha \in \mathcal{M}_s$, and for all variables x occurring in the domain of α: $d \mapsto [\![s]\!]_{[x:=d]}$ is either a strongly monotonic function, or a constant function.*

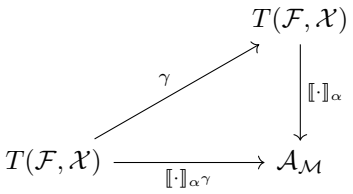
Proof Sketch. By induction on the form of s. The second part of the induction hypothesis is used to prove that $[\![\lambda x.s]\!] \in \mathcal{M}$, as *MakeSM* must be applied on either a strongly monotonic or a constant function. ◀

To prove Theorem 17 we need an AFS version of the so called *Substitution Lemma*. We begin by giving a systematic way of extending a substitution (seen as a morphism between terms) to a valuation, seen as morphism from terms to elements of $\mathcal{A}_\mathcal{M}$.

▶ **Definition B.2.** *Given a substitution $\gamma = [x_1 := s_1, \ldots, x_n := s_n]$ and a valuation α, we define α^γ as the valuation such that $\alpha^\gamma(x) = \alpha(x)$, if $x \notin \mathtt{dom}(\gamma)$; and $\alpha^\gamma(x) = [\![x\gamma]\!]_\alpha$, otherwise.*

▶ **Lemma B.3** (Substitution Lemma)**.** *For any substitution γ and valuation α, $[\![s\gamma]\!]_\alpha = [\![s]\!]_{\alpha^\gamma}$. Additionally, if $[\![s]\!] \sqsupset_\sigma [\![t]\!]$ ($[\![s]\!] \sqsupseteq_\sigma [\![t]\!]$), then $[\![s\gamma]\!] \sqsupset_\sigma [\![t\gamma]\!]$ ($[\![s\gamma]\!] \sqsupseteq_\sigma [\![t\gamma]\!]$).*

Proof.
By inspection of Definition B.2 it can be easily shown by induction on s that the diagram to the right commutes. As a consequence, if $[\![s]\!]_\alpha \sqsupset_\sigma [\![t]\!]$ for any valuation α, then $[\![s]\!]_{\alpha^\gamma} \sqsupset_\sigma [\![t]\!]_{\alpha^\gamma}$ in particular. So $[\![s\gamma]\!]_\alpha \sqsupset_\sigma [\![t]\!]_\alpha$.
The case for $\sqsupseteq_\sigma$ is analogous.



◀

Proof Sketch of Theorem 17. This follows easily by induction on the definition of $s \to_\mathcal{R} t$, using the substitution lemma. ◀

We posit some results regarding the functions $\mathtt{0}_\sigma$, $\mathtt{addc}_\sigma$ and $\mathtt{costof}_\sigma$.

▶ **Lemma B.4.** *For all types σ: (1) $\mathtt{0}_\sigma \in \mathcal{M}_\sigma$; (2) for all $n \in \mathbb{N}$ and $x \in \mathcal{M}_\sigma$: $\mathtt{addc}_\sigma(n, x) \in \mathcal{M}_\sigma$; (3) $\mathtt{costof}_\sigma$ is weakly monotonic and strict in its first argument; (4) $\mathtt{addc}_\sigma$ is weakly monotonic and strict in both its arguments.*

Proof Sketch. All claims follow easily by a mutual induction on σ. ◀

▶ **Lemma B.5.** *For all types σ, for all $x \in \mathcal{M}_\sigma$: (1) $\mathtt{addc}_\sigma(0, x) = x$; (2) for all $n, m \in \mathbb{N}$: $\mathtt{addc}_\sigma(n, \mathtt{addc}_\sigma(m, x)) = \mathtt{addc}_\sigma(n+m, x)$; (3) if $n > 0$ then $\mathtt{addc}_\sigma(n, x) \sqsupset_\sigma x$; (4) if $y \in \mathcal{M}_\sigma$ is such that $x \sqsupset_\sigma y$ then $x \sqsupseteq_\sigma \mathtt{addc}_(1, y)$; (5) for all $n \in \mathbb{N}$: $\mathtt{costof}_\sigma(\mathtt{addc}_\sigma(n, x)) = n + \mathtt{costof}_\sigma(x)$.*

Proof Sketch. All claims follow easily by induction on σ. ◀

▶ **Lemma B.6.** *For all* σ, τ, $F \in \mathcal{M}_{\sigma\Rightarrow\tau}$, $x \in \mathcal{M}_\sigma$, $n \in \mathbb{N}$: $F(\mathtt{addc}_\sigma(n, x)) \sqsupseteq_\sigma \mathtt{addc}_\tau(n, F(x))$.

Proof Sketch. By induction on n, using the various claims in Lemma B.5. ◀

▶ **Lemma B.7.** *For all types* σ *and all* $x \in \mathcal{M}_\sigma$: $x \sqsupseteq_\sigma \mathtt{addc}_\sigma(\mathtt{costof}_\sigma(x), \mathtt{0}_\sigma)$.

Proof Sketch. By induction on σ, using Lemmas B.4–B.6. ◀

▶ **Lemma B.8.** *For* $F \in \mathcal{M}_{\sigma\Rightarrow\tau}$ *and* $x \in \mathcal{M}_\sigma$ *we have:* $\mathtt{costof}_\tau(F(x)) \geq \mathtt{costof}_\sigma(x)$.

Proof. Let $n := \mathtt{costof}_\sigma(x)$. By Lemma B.7, $x \sqsupseteq_\sigma \mathtt{addc}_\sigma(\mathtt{costof}_\sigma(x), \mathtt{0}_\sigma) = \mathtt{addc}_\sigma(n, \mathtt{0}_\sigma)$. Hence, by monotonicity of F, $F(x) \sqsupseteq_\tau F(\mathtt{addc}_\sigma(n, \mathtt{0}_\sigma))$. By Lemma B.6, this implies that $F(x) \sqsupseteq_\tau \mathtt{addc}_\tau(n, F(\mathtt{0}_\sigma))$. Since $\mathtt{costof}_\tau$ is strict in its first argument by Lemma B.4(3), we thus have $\mathtt{costof}_\tau(F(x)) \geq \mathtt{costof}_\sigma(\mathtt{addc}_\tau(n, F(\mathtt{0}_\sigma)))$, which $\geq n$ by Lemma B.5(5). ◀

We can now prove that Definition 20 indeed defines a (σ, τ)-monotonicity function.

▶ **Lemma B.9.** *Let* σ, τ *be simple types. Then* $\Phi_{\sigma,\tau}$ *is a* (σ, τ)*-monotonicity function.*

Proof Sketch. By case analysis and Lemma B.4 we see that $\Phi_{\sigma,\tau}$ maps $C_{\sigma,\tau}$ to $\mathcal{M}_{\sigma,\tau}$. To see that $\Phi_{\sigma,\tau}$ is strongly monotonic we also use a case analysis. If F and G are both constant functions or both strongly monotonic, the result follows easily; F is constant and G not cannot occur because eventually $\mathtt{costof}(G)(x) > \mathtt{costof}(F)(x)$; and if F is strongly monotonic and G is constant then $F(x) \sqsupset \mathtt{addc}(\mathtt{costof}(x), G(x))$ because $G(x) = G(\mathtt{0})$ and $F(x) \sqsupseteq F(\mathtt{addc}(\mathtt{costof}(x), \mathtt{0})) \sqsupseteq \mathtt{addc}(\mathtt{costof}(x), F(\mathtt{0}))$ by Lemmas B.4–B.8. ◀

Proof of Lemma 21. We have either $[\![(\lambda x.s)\cdot t]\!]_\alpha = \mathtt{addc}_\tau(\mathtt{costof}_\sigma([\![t]\!]_\alpha)+1, [\![s]\!]_{\alpha[x:=[\![t]\!]]})$ or $[\![(\lambda x.s)\cdot t]\!]_\alpha = \mathtt{addc}_\tau(1, [\![s]\!]_{\alpha[x:=[\![t]\!]]})$. By Lemma B.5(3) we have $[\![(\lambda x.s)\cdot t]\!]_\alpha \sqsupset_\tau [\![s]\!]_{\alpha[x:=[\![t]\!]]}$ in both cases. By Lemma B.3, $[\![s]\!]_{\alpha[x:=[\![t]\!]]} = [\![s[x := b]]\!]_\alpha$. This completes the proof. ◀

Proof of Lemma 22. Since $F \neq x$, we have $d \mapsto [\![F \cdot x]\!]_{\alpha[x:=d]} = d \mapsto \alpha(F)(d)$, which by extensionality is $\alpha(F)$. Since $\alpha(F)$ is monotonic we have $[\![\lambda x.F\, x]\!]_\alpha = \Phi_{\sigma,\tau}(d \mapsto [\![F \cdot x]\!]_{\alpha x:=d}) = \Phi_{\sigma,\tau}(\alpha(F)) = \mathtt{addc}_{\sigma,\tau}(1, \alpha(F))$. By Lemma B.5(3) this $\sqsupset_{\sigma\Rightarrow\tau} \alpha(F) = [\![F]\!]$. ◀

Proof Sketch of Lemma 25. Let $Q(x_1, \ldots, x_k) := y \mapsto F(x_1, \ldots, x_k)^{G(x_1,\ldots,x_k)}(y)$. To see that Q maps to $\mathcal{M}_{\tau\Rightarrow\tau}$, so that Q is strongly monotonic. We show that $F(\vec{u})^n(x) \sqsupset F(\vec{u})^n(y)$ whenever $x \sqsupset y$ by a straighforward induction on n, and similar for $x \sqsupseteq y$. To see that Q is weakly monotonic in its first k arguments, we show by induction on n that for all n, m with $n \geq m$ we have $F(u_1, \ldots, u_k)^n \sqsupseteq_{\tau\Rightarrow\tau} F(u'_1, \ldots, u'_k)^m$ if each $u_i \sqsupseteq u'_i$. The result then follows because $G(u_1, \ldots, u_k) \geq G(u'_1, \ldots, u'_k)$ by weak monotonicity of G. ◀

Proof Sketch of Lemma 32. For claim (1), let b be the largest of the constants used for each constructor; i.e., we have $\sum_{l=1}^{K[\kappa]} P_l(x^1, \ldots, x^m) \leq b + \sum_{i=1}^m \sum_{j=1}^{K[\iota_i]} x^i_j$ whenever $\mathcal{J}_{\mathsf{c}}(\vec{x}) = \langle P_1(\vec{x}), \ldots, P_{K_\kappa}(\vec{x})\rangle$. We prove by induction on the size of a data term $s :: \kappa$ that $\sum_{l=1}^{K[\kappa]} [\![s]\!]_l \leq a * |s|$. Then certainly $[\![s]\!]_l \leq b * |s|$ holds for any component $[\![s]\!]_l$.

For claim (2), let a be the largest of the constants used for each constructor c and component P_l, and let k be the largest value $K[\iota]$ for any sort in the program; let $b := \max(2, a * k)$. We prove by induction on the size of a data term s that $[\![s]\!]_l \leq 2^{b*|s|}$, In the proof, we use that $n + m \leq n * m$ whenever $n, m \geq 2$ and $2 * n \leq 2^n$ if $n \geq 2$, and hence: $2 * a * k * \sum_{i=1}^m 2^{a*k*|s_i|} \leq (2 * a * k) * \prod_{i=1}^m 2^{a*k*|s_i|} \leq 2^b * 2^{b*\sum_{i=1}^m |s_i|}$. ◀