

Collected Size Semantics for Functional Programs over Lists ^{*}

O. Shkaravska, M. van Eekelen, A. Tamalet

Institute for Computing and Information Sciences
Radboud University Nijmegen

Abstract. This work introduces collected size semantics of strict functional programs over lists. The collected size semantics of a function definition is a multivalued size function that collects the dependencies between every possible output size and the corresponding input sizes. Such functions annotate standard types and are defined by conditional rewriting rules generated during type inference.

We focus on the connection between the rewriting rules and lower and upper bounds on the multivalued size functions, when the bounds are given by piecewise polynomials. We show how, given a set of conditional rewriting rules, one can infer bounds that define an indexed family of polynomials that approximates the multivalued size function.

Using collected size semantics we are able to infer non-monotonic and non-linear lower and upper polynomial bounds for many functional programs. As a feasibility study, we use the procedure to infer lower and upper polynomial size-bounds on typical functions of a list library.

1 Introduction

Estimating heap consumption is an active research area as it becomes more and more of an issue in many applications, e.g. distributed computing and programming for small devices like smart cards, mobile phones or embedded systems.

This work explores typing support for checking output-on-input size dependencies for function definitions (functions for short) in a strict functional language. Knowing lower and upper bounds of these dependencies one can apply *amortisation* [10] to check and infer tight non-linear bounds on heap consumption [15]. Size dependencies are presented via *multivalued size functions* defined by conditional multiple-choice rewriting rules generated during type inference. These functions are used to annotate types. Since one is mostly interested in lower and upper bounds for size functions, we establish connections between the rewriting rules and size bounds. We focus on piecewise polynomial bounds, i.e., bounds that can be described by a finite number of polynomials. Given a set of conditional multiple-choice rewriting rules, we show how to infer lower and upper bounds that define an indexed family of polynomials. Such a family fully

^{*} This work is part of the AHA project [15] which is sponsored by the Netherlands Organisation for Scientific Research (NWO) under grant nr. 612.063.511.

covers the size function induced by the rewriting rules in the sense that for each input, there is a polynomial in the family that describes the size of the output.

We work with strict functions over *matrix-like* lists of lists, i.e., every nested list must have the same length. (It is possible to omit this restriction by allowing higher-order size functions. This is a subject of our nearest-future work.) We allow higher-order functions only when the size of the output depends just on zero-order arguments.

This work continues a series of papers where we have studied output-on-input polynomial size dependencies, in which the polynomials are not necessary monotonic. In [13] we designed a type system where each type is annotated with a single polynomial size expression. It allows to type function definitions where the size of the output depends on the sizes of the inputs, but not on their values. For instance, `append`: $\mathbb{L}_n(\alpha) \times \mathbb{L}_m(\alpha) \rightarrow \mathbb{L}_{n+m}(\alpha)$, whereas `delete` (which deletes, from a list, the first occurrence of an element if it exists) does not have a type in that system since it may or may not delete an element. We also developed a test-based annotation inference procedure for that system in [16].

To get a global idea of the results of this paper, consider again the function `delete`. Its collected size semantics can be expressed by the multivalued function $f_{\text{delete}}(n) = \{n, \max_0(n-1)\}$, with $\max_0 = \max(0, n)$, where n denotes the length of the input list. Our type system allows to express and infer such multivalued size functions in the form of rewriting rules. For instance, $\vdash f_{\text{delete}}(0) \rightarrow 0$ and $n \geq 1 \vdash f_{\text{delete}}(n) \rightarrow n-1 \mid 1 + f_{\text{delete}}(n-1)$, where “ \vdash ” denotes a logical entailment and “ \mid ” denotes “multiple-choice rewriting”. However, a user often prefers to deal with size functions in *closed form*, i.e. without recursion, like $f(n) = \{n, \max_0(n-1)\}$, or with their lower and upper bounds. The problem of obtaining closed forms for rewriting rules does not have a general solution. We study how to approximate a closed-form solution with an indexed family of piecewise polynomials, if such an approximation exists. For f_{delete} we can infer the family $\{\max_0(n-i)\}_{0 \leq i \leq 1}$, which precisely describes it.

Let \bar{n} denote a vector of variables of the form (n_1, \dots, n_k) . The inference procedure is based on the well-known fact that a polynomial p of degree d is defined by a finite number of points of the form $\{(\bar{n}_i, p(\bar{n}_i))\}_{i=0}^d$, that determine a system of linear equations w.r.t. the polynomial coefficients. It takes the following parameters: the *degree* d of polynomial lower and upper bounds of the size function, an *initial point* \bar{n}^0 and a *step* to obtain the next point. Using these parameters, the procedure generates the points lying on the bounds. For instance, for `delete`, we choose degree 1, initial point $n^0 = 1$ and step 1. Then the procedure generates the test points \bar{n}^i . In the example it generates $n^0 = 1, n^1 = 2$. Next, the rewriting rules are used to calculate the sets $f(\bar{n}^i)$. For `delete` we obtain $f(n^0) = \{0, 1\}$ and $f(n^1) = \{1, 2\}$. The procedure picks up the minimal and maximal values from each of the set $f(\bar{n}^i)$ and computes the coefficients of the lower and upper polynomial bounds as the solutions of two corresponding linear systems. In the example, the lower bound $p_{\min}(n) = n-1$ is computed from the nodes $\{(1, 0), (2, 1)\}$, and the upper bound $p_{\max}(n) = n$ from $\{(1, 1), (2, 2)\}$. The obtained bounds p_{\min} and p_{\max} define an indexed fam-

ily of polynomials, which may be presented, for instance, as $\{p_{\min}(\bar{n}) + i\}_{i=0}^{\delta(\bar{n})}$, where $\delta(\bar{n}) = p_{\max}(\bar{n}) - p_{\min}(\bar{n})$. The procedure depends on user-defined parameters (an initial point, a step, a degree). The consequences of a bad choice of parameters is that the bound will not be tight or they may even be incorrect. Checking if a given indexed family of polynomials approximates a given function is shown to be similar to type checking types annotated with indexed families of polynomials [12].

The rest of this paper is organised as follows. In Section 2 we define the programming language and in Section 3 its size-aware type system. Section 3 also defines the semantics of program values w.r.t. zero-order types and the operational semantics of the language. We give an inference procedure for families of polynomials that approximate multivalued size functions and discuss examples in Section 4. In Section 5 we discuss the feasibility of applying the analysis to a typical list library. Related work is discussed in Section 6. Section 7 draws conclusions and gives directions to future work. The technical report [11] gives more examples of checking and inference in detail.

2 Language

The type system is designed for a strict functional language over integers and (polymorphic) lists. Algebraic data types could be added as we did in [14]. Language expressions are defined by the following grammar:

$$\begin{aligned}
 \text{Basic } b &::= c \mid \text{unop } x \mid x \text{ binop } y \mid \text{Nil} \mid \text{Cons}(z, l) \mid f(g_1, \dots, g_l, z_1, \dots, z_k) \\
 \text{Expr } e &::= b \mid \text{if } x \text{ then } e_1 \text{ else } e_2 \\
 &\quad \mid \text{let } z = b \text{ in } e_1 \\
 &\quad \mid \text{match } l \text{ with} \mid \text{Nil} \Rightarrow e_1 \\
 &\quad \quad \mid \text{Cons}(z_{\text{hd}}, l_t) \Rightarrow e_2 \\
 &\quad \mid \text{letfun } f(g_1, \dots, g_l, z_1, \dots, z_k) = e_1 \text{ in } e_2
 \end{aligned}$$

where c ranges over integer and boolean constants **False** and **True**, x and y denote program variables of integer and boolean types, l ranges over lists, z denotes a program variable of zero-order type, g ranges over higher-order program variables, **unop** is a unary operation, either $-$ or \neg , **binop** is one of the integer or boolean binary operations, and f denotes a function name. Variables may be decorated with sub- and superscripts.

The syntax distinguishes between zero-order let-binding of variables and higher-order letfun-binding of functions. In a function body, the only free program variables are its parameters. We prohibit head-nested let-expressions and restrict subexpressions in function calls to variables to make type checking straightforward. Program expressions of a general form may be equivalently transformed into expressions of this form. We consider this language as an intermediate language where a more user friendly language may be compiled into.

3 Type System

We consider a type system constituted from zero-order and higher-order types and typing rules corresponding to program constructs. Size annotations are mul-

tivalued numerical functions $f: \mathcal{R}^k \rightarrow 2^{\mathcal{R}}$ that represent lengths of finite lists and arithmetic operations over these lengths. \mathcal{R} can be any numerical ring; its choice influences decidability of type checking and the set of well-typed programs.

Zero-order types are assigned to program values, which are integers, booleans and finite lists. The list type is annotated by a multivalued size function:

$$\text{Types } \tau ::= \text{Int} \mid \text{Bool} \mid \alpha \mid \mathbf{L}_{f(\bar{n})}(\tau),$$

where α is a type variable and \bar{n} is a collection of size variables. The multivalued size functions f in our type system are defined by conditional rewriting rules. For example, consider a function `insert` that inserts an element z into a list l if and only if there is no element in l related to z by g .

$$\begin{aligned} \text{insert}(g, z, l) = & \\ & \text{match } l \text{ with } \mid \text{Nil} \Rightarrow \text{let } l' = \text{Nil} \text{ in } \text{Cons}(z, l') \\ & \mid \text{Cons}(\text{hd}, \text{tl}) \Rightarrow \text{if } g(z, \text{hd}) \text{ then } l \text{ else} \\ & \quad \text{let } l'' = \text{insert}(g, z, \text{tl}) \text{ in } \text{Cons}(\text{hd}, l'') \end{aligned}$$

The corresponding size rewriting system is

$$\begin{aligned} & \vdash f_{\text{insert}}(0) \rightarrow 1 \\ n \geq 1 & \vdash f_{\text{insert}}(n) \rightarrow n \mid 1 + f_{\text{insert}}(n-1) \end{aligned}$$

The type of `insert` is $(\alpha \times \alpha \rightarrow \text{Bool}) \times \alpha \times \mathbf{L}_n(\alpha) \rightarrow \mathbf{L}_{f_{\text{insert}}(n)}(\alpha)$. It is desirable to find closed forms for functions defined by such rewriting rules. In this work we are interested in the cases where closed-form solutions (or approximations of the solutions) are definable as indexed families of piecewise polynomials. For instance, a closed-form solution for f_{insert} is $\{n+i\}_{0 \leq i \leq 1}$.

The sets $TV(\tau)$ and $SV(\tau)$ of type and size variables of a type τ are defined inductively in the obvious way. Note that $SV(\mathbf{L}_0(\tau)) = \emptyset$, since all empty lists of the same underlying type represent the same data structure. For instance, $\mathbf{L}_0(\mathbf{L}_m(\text{Int}))$ represent the same structure as $\mathbf{L}_0(\mathbf{L}_0(\text{Int}))$.

Zero-order types without type variables and size variables are *ground types*:

$$\text{GroundTypes } \tau^\bullet ::= \tau \text{ such that } SV(\tau) = \emptyset \wedge TV(\tau) = \emptyset$$

The semantics of ground types is defined in Section 3.1. Here we give some examples: `Int`, $\mathbf{L}_5(\text{Bool})$, $\mathbf{L}_{f_{\text{insert}}(2)}(\text{Bool})$ with $\mathcal{R} = \text{Int}$ are ground types, whereas α , $\mathbf{L}_{n+5}(\text{Int})$ and $\mathbf{L}_{f_{\text{insert}}(n)}(\text{Bool})$ with non-specified n are not. Examples of inhabitants of ground types are `[True, True]` and `[False, True, True]` for $\mathbf{L}_{f_{\text{insert}}(2)}(\text{Bool})$.

Let τ° denote a zero-order type where size expressions are all size variables or constants, like, e.g., $\mathbf{L}_n(\alpha)$. Function types are then defined inductively:

$$\text{FunctionTypes } \tau^f ::= \tau_1^f \times \dots \times \tau_{k'}^f \times \tau_1^\circ \times \dots \times \tau_k^\circ \rightarrow \tau_0$$

where k' may be zero (i.e. the list $\tau_1^f, \dots, \tau_{k'}^f$ is empty) and $SV(\tau_0)$ contains only size variables of $\tau_1^\circ, \dots, \tau_k^\circ$. Consider, for instance, the function definition for `filter`: $(\alpha \rightarrow \text{Bool}) \times \mathbf{L}_n(\alpha) \rightarrow \mathbf{L}_{f_{\text{filter}}(n)}(\alpha)$

$$\begin{aligned} \text{filter}(g, l) = & \\ & \text{match } l \text{ with } \mid \text{Nil} \Rightarrow \text{Nil} \\ & \mid \text{Cons}(\text{hd}, \text{tl}) \Rightarrow \text{if } g(\text{hd}) \text{ then let } l' = \text{filter}(g, \text{tl}) \text{ in } \text{Cons}(\text{hd}, l') \\ & \quad \text{else } \text{filter}(g, \text{tl}) \end{aligned}$$

The size function f_{filter} is defined by

$$\begin{aligned} &\vdash f_{\text{filter}}(0) = 0 \\ n \geq 1 &\vdash f_{\text{filter}}(n) = 1 + f_{\text{filter}}(n-1) \mid f_{\text{filter}}(n-1) \end{aligned}$$

The closed-form solution for f_{filter} is $\{i\}_{0 \leq i \leq n}$.

A context Γ is a mapping from zero-order variables to zero-order types. A signature Σ is a mapping from function names to function types. The definition of $SV(-)$ is straightforwardly extended to contexts:

$$SV(\Gamma) = \bigcup_{z \in \text{dom}(\Gamma)} SV(\Gamma(z))$$

3.1 Semantics of zero-order types

In our semantic model, the purpose of the heap is to store lists. Therefore, a heap is a finite collection of locations ℓ that can store list elements. A location is the address of a cons-cell consisting of a head field hd , which stores a list element, and a tail field tl , which contains the location of the next cons-cell of the list, or the NULL address. Formally, a program value is either an integer or boolean constant, a location or the null-address and a heap is a finite partial mapping from locations and fields into program values:

$$\begin{aligned} \text{Address} \quad \text{adr} &::= \ell \mid \text{NULL} & \ell \in \text{Loc} \\ \text{Val} \quad v &::= c \mid \text{adr} & c \in \text{Int} \cup \text{Bool} \\ \text{Heap} \quad h &: \text{Loc} \rightarrow \{hd, tl\} \rightarrow \text{Val} \end{aligned}$$

We will write $h.\ell.hd$ and $h.\ell.tl$ for the results of applications $h \ell hd$ and $h \ell tl$, which denote the values stored in the heap h at the location ℓ at its fields hd and tl , respectively. Let $h.\ell.[hd := v_h, tl := v_t]$ denote the heap equal to h everywhere but in ℓ , which at the hd -field of ℓ gets the value v_h and at the tl -field of ℓ gets the value v_t .

The semantics w of a program value v with respect to a specific heap h and a ground type τ^\bullet is a set-theoretic interpretation given via the four-place relation $v \models_{\tau^\bullet}^h w$. Integer and boolean constants interpret themselves, and locations are interpreted as non-cyclic lists:

$$\begin{aligned} c &\models_{\text{Int} \cup \text{Bool}}^h c \\ \text{NULL} &\models_{L_{f(\bar{n}_0)}(\tau^\bullet)}^h \square \quad \text{iff } 0 \in f(\bar{n}_0) \\ \ell &\models_{L_{f(\bar{n}_0)}(\tau^\bullet)}^h w_{hd} :: w_{tl} \quad \text{iff } \ell \in \text{dom}(h), \\ & \quad h.\ell.hd \models_{\tau^\bullet}^{h|_{\text{dom}(h) \setminus \{\ell\}}} w_{hd}, \\ & \quad h.\ell.tl \models_{L_{f(\bar{n}_0)-1}(\tau^\bullet)}^{h|_{\text{dom}(h) \setminus \{\ell\}}} w_{tl} \end{aligned}$$

where $h|_{\text{dom}(h) \setminus \{\ell\}}$ denotes the heap equal to h everywhere except in ℓ , where it is undefined.

It is easy to establish a natural connection between the size functions in a ground list type and the length of a chain of cons-cells that “implements” its inhabitant in a heap. The length is defined by the function:

$$\begin{aligned} \text{length} &: \text{Heap} \rightarrow \text{Address} \rightarrow \mathcal{N} \\ \text{length}_h(\text{NULL}) &= 0 \quad \text{length}_h(\ell) = 1 + \text{length}_{h|_{\text{dom}(h) \setminus \{\ell\}}}(h.\ell.tl) \end{aligned}$$

Note that the function $\text{length}_h(-)$ does not take sharing into account, in the sense that the actual total size of allocated shared lists is less than the sum of their lengths. Thus, the sum of the lengths of the lists provides an upper bound on the amount of memory actually allocated.

Lemma 1 (Consistency of model relation).

The relation $\text{adr} \models_{\perp_{f(\bar{n}_0)}^h}(\tau \bullet)$ implies that $\text{length}_h(\text{adr}) \in f(\bar{n}_0)$.

The proof is done by induction on the relation \models .

3.2 Operational semantics of program expressions

The operational semantics is standard. It extends the semantics from [13] with higher-order functions.

We introduce a *frame store* as a mapping from program variables to program values. This mapping is maintained when a function body is evaluated. Before evaluation of the function body starts, the store contains only the actual parameters of the function. During evaluation, the store is extended with the variables introduced by pattern matching or `let`-constructs. These variables are eventually bound to the actual parameters. Thus there is no access beyond the current frame. Formally, a frame store s is a finite partial map from variables to values, *Store* $s: \text{ProgramVars} \rightarrow \text{Val}$.

Using heaps and a frame store and maintaining a mapping \mathcal{C} of *closures*, from function names to the bodies of the function definitions, the operational semantics of program expressions is defined inductively in the usual way. Here we give some of the rules as examples. The full operational semantics may be found in the technical report [11].

$$\begin{aligned} &\frac{c \in \text{Int} \cup \text{Bool}}{s; h; \mathcal{C} \vdash c \rightsquigarrow c; h} \text{OSCONST} \quad \frac{}{s; h; \mathcal{C} \vdash z \rightsquigarrow s(z); h} \text{OSVAR} \\ &\frac{\begin{array}{c} h.s(l).hd = v_{hd} \quad h.s(l).tl = v_{tl} \\ s[hd := v_{hd}, tl := v_{tl}]; h; \mathcal{C} \vdash e_2 \rightsquigarrow v; h' \end{array}}{s; h; \mathcal{C} \vdash \text{match } l \text{ with } \begin{array}{l} | \text{Nil} \Rightarrow e_1 \\ | \text{Cons}(hd, tl) \Rightarrow e_2 \end{array} \rightsquigarrow v; h'} \text{OSMATCH-CONS} \end{aligned}$$

3.3 Typing rules

A typing judgement is a relation of the form $D, \Gamma \vdash_{\Sigma} e: \tau$. Informally, it means that with the set of constraints D in the zero-order variable context Γ the expression e has type τ where the signature Σ contains type assumptions for all called functions. The set D of disequations and inclusions is relevant only when a rule for pattern-matching is applied. When the `nil`-branch is entered on a list $\perp_{f(\bar{n})}(\alpha)$, then D is extended with $0 \in f(\bar{n})$. When the `cons`-branch is entered,

then D is extended with $n' \geq 1$, $n' \in f(\bar{n})$, where n' is a fresh size variable in D .

Given types $\tau = \mathbf{L}_{f_1(\bar{n})}(\dots \mathbf{L}_{f_k(\bar{n})}(\alpha) \dots)$ and $\tau' = \mathbf{L}_{f'_1(\bar{n})}(\dots \mathbf{L}_{f'_k(\bar{n})}(\alpha) \dots)$, let the entailment $D \vdash \tau \rightarrow \tau'$ abbreviate the collection of rules that (conditionally) rewrite $f_i(\bar{n})$ to $f'_i(\bar{n})$:

$$\begin{array}{ll} & D \vdash f_1(\bar{n}) \rightarrow f'_1(\bar{n}) \\ \text{if there exists a positive value in } f'_1(\bar{n}) \text{ then} & D \vdash f_2(\bar{n}) \rightarrow f'_2(\bar{n}) \\ \text{if there exist positive values in } f'_1(\bar{n}), f'_2(\bar{n}) \text{ then} & D \vdash f_3(\bar{n}) \rightarrow f'_3(\bar{n}) \\ \dots & \\ \text{if there exist positive values in } f'_1(\bar{n}), \dots, f'_{k-1}(\bar{n}) \text{ then} & D \vdash f_k(\bar{n}) \rightarrow f'_k(\bar{n}) \end{array}$$

For instance, the entailment $n \geq 2 \vdash \mathbf{L}_{f_1(n)}(\mathbf{L}_{f_2(n)}(\alpha)) \rightarrow \mathbf{L}_{n-1}(\mathbf{L}_{n^2}(\alpha))$ abbreviates the rules $n \geq 2 \vdash f_1(n) \rightarrow n-1$ and $n \geq 2 \vdash f_2(n) \rightarrow n^2$. However, the entailment $n = 1 \vdash \mathbf{L}_{f_1(n)}(\mathbf{L}_{f_2(n)}(\alpha)) \rightarrow \mathbf{L}_{n-1}(\mathbf{L}_{n^2}(\alpha))$ abbreviates the single rule $n = 1 \vdash f_1(n) = n-1$. The rule $n = 1 \vdash f_2(n) \rightarrow n^2$ is not present because $f_1(1) = 0$ and thus the outer list must be empty.

The typing judgement relation is defined by the following rules:

$$\begin{array}{c} \frac{}{D, \Gamma \vdash_{\Sigma} c: \mathbf{Int}} \text{ICONST} \quad \frac{}{D, \Gamma \vdash_{\Sigma} b: \mathbf{Bool}} \text{BCONST} \\ \\ \frac{D \vdash \tau' \rightarrow \tau}{D, \Gamma, \mathbf{z}: \tau \vdash_{\Sigma} \mathbf{z}: \tau'} \text{VAR} \quad \frac{D \vdash \tau' \rightarrow \mathbf{L}_0(\tau)}{D, \Gamma \vdash_{\Sigma} \mathbf{Nil}: \tau'} \text{NIL} \\ \\ \frac{D \vdash \tau' \rightarrow \mathbf{L}_{f(\bar{n})+1}(\tau_2) \quad D \vdash \tau_2 \rightarrow \tau_1}{D, \Gamma, \mathbf{hd}: \tau_1, \mathbf{tl}: \mathbf{L}_{f(\bar{n})}(\tau_2) \vdash_{\Sigma} \mathbf{Cons}(\mathbf{hd}, \mathbf{tl}): \tau'} \text{CONS} \\ \\ \frac{\Gamma(\mathbf{x}) = \mathbf{Bool} \quad \frac{D \vdash \tau \rightarrow \tau_1 \mid \tau_2}{D, \Gamma \vdash_{\Sigma} e_t: \tau_1} \quad \frac{D, \Gamma \vdash_{\Sigma} e_f: \tau_2}{D, \Gamma \vdash_{\Sigma} \mathbf{if } \mathbf{x} \text{ then } e_t \text{ else } e_f: \tau} \text{IF} \\ \\ \frac{\mathbf{z} \notin \text{dom}(\Gamma) \quad \frac{D, \Gamma \vdash_{\Sigma} e_1: \tau_z \quad D, \Gamma, \mathbf{z}: \tau_z \vdash_{\Sigma} e_2: \tau}{D, \Gamma \vdash_{\Sigma} \mathbf{let } \mathbf{z} = e_1 \text{ in } e_2: \tau} \text{LET} \\ \\ \frac{D, 0 \in f(\bar{n}), \Gamma, \mathbf{l}: \mathbf{L}_{f(\bar{n})}(\tau) \vdash_{\Sigma} e_{\text{Nil}}: \tau' \quad \mathbf{hd}, \mathbf{tl} \notin \text{dom}(\Gamma) \quad D, n' \geq 1 \in f(\bar{n}), \Gamma, \mathbf{hd}: \tau, \mathbf{l}: \mathbf{L}_{f(\bar{n})}(\tau), \mathbf{tl}: \mathbf{L}_{f(\bar{n})-1}(\tau) \vdash_{\Sigma} e_{\text{Cons}}: \tau'}{D; \mathbf{l}: \mathbf{L}_{f(\bar{n})}(\tau) \vdash_{\Sigma} \mathbf{match } \mathbf{l} \text{ with } \left| \begin{array}{l} \mathbf{Nil} \Rightarrow e_{\text{Nil}} \\ \mathbf{Cons}(\mathbf{hd}, \mathbf{tl}) \Rightarrow e_{\text{Cons}} \end{array} \right. : \tau'} \text{MATCH} \end{array}$$

where $n' \notin SV(D)$. Note that if in the MATCH-rule f is single-valued, then the statements in the nil and cons branches are $f(\bar{n}) = 0$ and $f(\bar{n}) \geq 1$, respectively.

$$\frac{\Sigma(f) = \tau_1^f \times \dots \times \tau_{k'}^f \times \tau_1^{\circ} \times \dots \times \tau_k^{\circ} \rightarrow \tau_0 \quad \Sigma(\mathbf{g}_1) = \tau_1^f, \dots, \Sigma(\mathbf{g}_{k'}) = \tau_{k'}^f \quad \mathbf{z}_1: \tau_1^{\circ}, \dots, \mathbf{z}_k: \tau_k^{\circ} \vdash_{\Sigma} e_1: \tau_0 \quad D; \Gamma \vdash_{\Sigma} e_2: \tau'}{D; \Gamma \vdash_{\Sigma} \mathbf{letfun } f(\mathbf{g}_1, \dots, \mathbf{g}_{k'}, \mathbf{z}_1, \dots, \mathbf{z}_k) = e_1 \text{ in } e_2: \tau'} \text{LETFUN}$$

$$\frac{\begin{array}{l} \Sigma(f) = \tau_1^f \times \dots \times \tau_{k'}^f \times \tau_1^\circ \times \dots \times \tau_k^\circ \rightarrow \tau_0 \\ \Sigma(\mathbf{g}_i) \text{ is an instance of the type } \tau_i^f; \\ D \vdash \tau \rightarrow \sigma(\tau_0) \quad D \vdash C(\tau_1, \dots, \tau_k) \end{array}}{D, \Gamma, \mathbf{z}_1 : \tau_1, \dots, \mathbf{z}_k : \tau_k \vdash_{\Sigma} f(\mathbf{g}_1, \dots, \mathbf{g}_{k'}, \mathbf{z}_1, \dots, \mathbf{z}_k) : \tau} \text{FUNAPP}$$

The function application rule computes a substitution σ from the formal size and type variables to the actual size expressions and types, and a set C of equations collecting restrictions on the actual input types. These restrictions are of the form $\tau \equiv \tau'$ abbreviating equality of the corresponding underlying types and size functions. The equation $\tau \equiv \tau'$ belongs to C if τ and τ' are actual types corresponding to the same formal type. As an example of such an equivalence consider a call to a function `scalarprod` : $\mathbf{L}_m(\mathbf{Int}) \times \mathbf{L}_m(\mathbf{Int}) \rightarrow \mathbf{Int}$. Due to the occurrence of m in both arguments the actual parameters $l : \tau$ and $l' : \tau'$ corresponding to the same formal type $\mathbf{L}_m(\mathbf{Int})$ must have equal sizes. To see how the substitution σ is applied, consider a formal size parameter m with $\sigma(m) = f'(\bar{n})$. Then

$$\sigma\left(\mathbf{L}(\dots \mathbf{L}_{f(m)}(\dots \mathbf{L}(\alpha) \dots) \dots)\right) = \mathbf{L}(\dots \mathbf{L}_{f'(f'(\bar{n}))}(\dots \mathbf{L}(\alpha) \dots) \dots).$$

Now we illustrate with an example how the typing rules are used to construct rewriting rules for multivalued size functions. Consider a function `rel` that produces all pairs of elements from two argument lists that are related to each other according to a given predicate. For instance `rel(>, [2, 3, 5], [2, 3]) = [[3, 2], [5, 2], [5, 3]]`. This function calls an auxiliary function `rel_pairs`, that given a single element z and a list, produces the list of all pairs (z, z') of the related elements, where z' runs over the list. The definitions for `rel` and `rel_pairs` are

$$\text{rel}(g, l_1, l_2) = \text{match } l_1 \text{ with } \begin{array}{l} | \text{Nil} \Rightarrow \text{Nil} \\ | \text{Cons}(\text{hd}, \text{tl}) \Rightarrow \text{append}(\text{rel_pairs}(g, \text{hd}, l_2), \text{rel}(g, \text{tl}, l_2)) \end{array}$$

$$\text{and } \text{rel_pairs}(g, z, l) = \text{match } l \text{ with } \begin{array}{l} | \text{Nil} \Rightarrow \text{Nil} \\ | \text{Cons}(\text{hd}, \text{tl}) \Rightarrow \text{if } g(z, \text{hd}) \text{ then} \\ \quad \text{Cons}(\text{Cons}(z, \text{Cons}(\text{hd}, \text{Nil})), \text{rel_pairs}(g, z, \text{tl})) \\ \quad \text{else } \text{rel_pairs}(g, z, \text{tl}) \end{array}$$

The types are $(\alpha \rightarrow \alpha \rightarrow \text{Bool}) \times \mathbf{L}_n(\alpha) \times \mathbf{L}_m(\alpha) \rightarrow \mathbf{L}_{f_{\text{rel}_1}}(\mathbf{L}_{f_{\text{rel}_2}}(\alpha))$ and $(\alpha \rightarrow \alpha \rightarrow \text{Bool}) \times \alpha \times \mathbf{L}_m(\alpha) \rightarrow \mathbf{L}_{f_{\text{rel_pairs}_1}}(\mathbf{L}_{f_{\text{rel_pairs}_2}}(\alpha))$, respectively. We want to construct rewriting rules for $f_{\text{rel_pairs}_1}$ and $f_{\text{rel_pairs}_2}$. We apply typing rules in the backward style to the body of `rel_pairs`. For the sake of convenience, below in the typing judgements, we list only the relevant variables of the context.

1. We want to infer $f_{\text{rel_pairs}_1}$ and $f_{\text{rel_pairs}_2}$ such that

$$\mathbf{z} : \alpha, l : \mathbf{L}_n(\alpha) \vdash_{\Sigma} e_{\text{rel_pairs}} : \mathbf{L}_{f_{\text{rel_pairs}_1}}(\mathbf{L}_{f_{\text{rel_pairs}_2}}(\alpha))$$
2. We start applying the match-rule since $e_{\text{rel_pairs}}$ is given by a pattern-matching. We obtain

$$\text{Nil-branch: } n = 0 \vdash_{\Sigma} \text{Nil} : \mathbf{L}_{f_{\text{rel_pairs}_1}}(\mathbf{L}_{f_{\text{rel_pairs}_2}}(\alpha))$$

$$\text{Cons-branch: } n \geq 1; \mathbf{z} : \alpha, l : \mathbf{L}_n(\alpha) \vdash_{\Sigma} e' : \mathbf{L}_{f_{\text{rel_pairs}_1}}(\mathbf{L}_{f_{\text{rel_pairs}_2}}(\alpha))$$

where e' is the if-expression in the cons-branch.

3. Since the expression in the nil-branch is just Nil, we apply the nil-rule and obtain $n = 0 \vdash \mathbf{L}_{f_{\text{rel_pairs}_1}}(\mathbf{L}_{f_{\text{rel_pairs}_2}}(\alpha)) \rightarrow \mathbf{L}_0(\tau_0)$ that according to the definition of $D \vdash \tau \rightarrow \tau'$ reduces to $n = 0 \vdash f_{\text{rel_pairs}_1}(n) \rightarrow 0$.
4. Apply the if-rule to the expression e' in the cons-branch to obtain that $\mathbf{L}_{f_{\text{rel_pairs}_1}}(\mathbf{L}_{f_{\text{rel_pairs}_2}}(\alpha)) \rightarrow \tau_1 \mid \tau_2$, where

$$n \geq 1; \mathbf{z}: \alpha, \text{hd}: \alpha, \text{tl}: \mathbf{L}_{n-1}(\alpha) \vdash_{\Sigma} \text{Cons}(\text{Cons}(\mathbf{z}, \text{Cons}(\text{hd}, \text{Nil})), \text{rel_pairs}(\mathbf{g}, \mathbf{z}, \text{tl})): \tau_1$$

$$n \geq 1; \mathbf{z}: \alpha, \text{tl}: \mathbf{L}_{n-1}(\alpha) \vdash_{\Sigma} \text{rel_pairs}(\mathbf{g}, \mathbf{z}, \text{tl}): \tau_2$$

Note, that the expression in the true-branch abbreviates the chain of let-bindings:

let $\mathbf{z}_1 = \text{Nil}$ in let $\mathbf{z}_2 = \text{Cons}(\text{hd}, \mathbf{z}_1)$ in let $\mathbf{z}_3 = \text{Cons}(\mathbf{z}, \mathbf{z}_2)$ in
 let $\mathbf{z}_4 = \text{rel_pairs}(\mathbf{g}, \mathbf{z}, \text{tl})$ in $\text{Cons}(\mathbf{z}_3, \mathbf{z}_4)$

Let $e_{\text{body}_1}, \dots, e_{\text{body}_4}$ denote the let-bodies corresponding to the let-bindings of $\mathbf{z}_1, \dots, \mathbf{z}_4$, respectively.

5. Applying the let-rule to \mathbf{z}_1 -binding gives

$$\text{let}_1: n \geq 1 \vdash_{\Sigma} \text{Nil}: ?\tau^1$$

$$\text{body}_1: n \geq 1; \mathbf{z}_1: ?\tau^1, \dots \vdash_{\Sigma} e_{\text{body}_1}: \tau_1$$
6. Applying the nil-rule to the let-branch instantiates $?\tau^1$ with $\mathbf{L}_0(?\tau^{10})$, so we obtain $n \geq 1; \mathbf{z}_1: \mathbf{L}_0(?\tau^{10}), \dots \vdash_{\Sigma} e_{\text{body}_1}: \tau_1$.
7. Applying the let-rule to \mathbf{z}_2 -binding gives

$$\text{let}_2: n \geq 1; \text{hd}: \alpha, \mathbf{z}_1: \mathbf{L}_0(?\tau^{10}) \vdash_{\Sigma} \text{Cons}(\text{hd}, \mathbf{z}_1): ?\tau^2$$

$$\text{body}_2: n \geq 1; \mathbf{z}_2: ?\tau^2, \dots \vdash_{\Sigma} e_{\text{body}_2}: \tau_1$$
8. Applying the cons-rule to the let-branch instantiates $?\tau^{10}$ with α and $?\tau^2$ with $\mathbf{L}_1(\alpha)$, so we obtain $\text{body}_2: n \geq 1; \mathbf{z}: \alpha, \mathbf{z}_2: \mathbf{L}_1(\alpha), \dots \vdash_{\Sigma} e_{\text{body}_2}: \tau_1$.
9. Similarly, applying the let- and cons-rules for \mathbf{z}_3 -binding gives

$$\text{body}_3: n \geq 1; \mathbf{z}: \alpha, \text{tl}: \mathbf{L}_{n-1}(\alpha), \mathbf{z}_3: \mathbf{L}_2(\alpha) \vdash_{\Sigma} e_{\text{body}_3}: \tau_1$$
10. Applying the let- and funapp-rules for \mathbf{z}_4 -binding gives

$$\text{body}_4: n \geq 1; \mathbf{z}_3: \mathbf{L}_2(\alpha), \mathbf{z}_4: \mathbf{L}_{f_{\text{rel_pairs}_1}(n-1)}(\mathbf{L}_{f_{\text{rel_pairs}_2}(n-1)}(\alpha)) \vdash_{\Sigma} \text{Cons}(\mathbf{z}_3, \mathbf{z}_4): \tau_1$$
11. Applying the cons-rule gives $n \geq 1 \vdash \tau_1 \rightarrow \mathbf{L}_{f_{\text{rel_pairs}_1}(n-1)+1}(\mathbf{L}_{f_{\text{rel_pairs}_2}(n-1)}(\alpha))$ and $n \geq 1 \vdash f_{\text{rel_pairs}_2}(n-1) \rightarrow 2$.
12. Applying the function application rule in the false-branch gives $n \geq 1 \vdash \tau_2 \rightarrow \mathbf{L}_{f_{\text{rel_pairs}_1}(n-1)}(\mathbf{L}_{f_{\text{rel_pairs}_2}(n-1)}(\alpha))$.
13. Recalling the multiple-choice-rewriting side condition from the application of the if-rule we obtain

$$n \geq 1 \vdash \mathbf{L}_{f_{\text{rel_pairs}_1}(n)}(\mathbf{L}_{f_{\text{rel_pairs}_2}(n)}(\alpha)) \rightarrow$$

$$\mathbf{L}_{f_{\text{rel_pairs}_1}(n-1)+1}(\mathbf{L}_{f_{\text{rel_pairs}_2}(n-1)}(\alpha)) \mid \mathbf{L}_{f_{\text{rel_pairs}_1}(n-1)}(\mathbf{L}_{f_{\text{rel_pairs}_2}(n-1)}(\alpha))$$

that means

$$n \geq 1 \quad \vdash f_{\text{rel_pairs}_1}(n) \rightarrow f_{\text{rel_pairs}_1}(n-1) + 1 \mid$$

$$f_{\text{rel_pairs}_1}(n-1)$$

$$n' \in f_{\text{rel_pairs}_1}(n-1) + 1, n' \geq 1, n \geq 1 \vdash f_{\text{rel_pairs}_2}(n) \rightarrow f_{\text{rel_pairs}_2}(n-1)$$

$$n' \in f_{\text{rel_pairs}_1}(n-1), n' \geq 1, n \geq 1 \quad \vdash f_{\text{rel_pairs}_2}(n) \rightarrow f_{\text{rel_pairs}_2}(n-1)$$

Recall that $\vdash f_{\text{rel_pairs}_1}(0) \rightarrow 0$ and $n \geq 1 \vdash f_{\text{rel_pairs}_2}(n-1) \rightarrow 2$ due to the nil-rule in the nil-branch and the last cons-rule respectively. So, combining this altogether gives

$$\begin{aligned}
& \vdash f_{\text{rel_pairs}_1}(0) \rightarrow 0 \\
n \geq 1 & \vdash f_{\text{rel_pairs}_1}(n) \rightarrow f_{\text{rel_pairs}_1}(n-1) + 1 \mid f_{\text{rel_pairs}_1}(n-1) \\
n \geq 0 & \vdash f_{\text{rel_pairs}_2}(n) \rightarrow 2
\end{aligned}$$

Similarly we obtain rewriting rules for the multivalued size functions for rel.

$$\begin{aligned}
& \vdash f_{\text{rel}_1}(0, m) \rightarrow 0 \\
n \geq 1 & \vdash f_{\text{rel}_1}(n, m) \rightarrow f_{\text{rel_pairs}_1}(m) + f_{\text{rel}_1}(n-1, m) \\
n \geq 1 & \vdash f_{\text{rel}_2}(n, m) \rightarrow f_{\text{rel_pairs}_2}(m)
\end{aligned}$$

3.4 Semantics of typing judgements (soundness)

The set-theoretic semantics of typing judgements is formalised later in this section as the soundness theorem, which is defined by means of the following two predicates. One indicates if a program value is *valid* with respect to a certain heap and a ground type. The other does the same for sets of values and types, taken from a frame store and a ground context Γ^\bullet :

$$\begin{aligned}
\text{Valid}_{\text{val}}(v, \tau^\bullet, h) &= \exists_w [v \models_{\tau^\bullet}^h w] \\
\text{Valid}_{\text{store}}(\text{vars}, \Gamma^\bullet, s, h) &= \forall_{z \in \text{vars}} [\text{Valid}_{\text{val}}(s(z), \Gamma^\bullet(z), h)]
\end{aligned}$$

Let a valuation ϵ map size variables to concrete sizes (numbers from the ring \mathcal{R}) and an instantiation η map type variables to ground types:

$$\begin{aligned}
\text{Valuation } \epsilon &: \text{SizeVariables} \rightarrow \mathcal{R} \\
\text{Instantiation } \eta &: \text{TypeVariables} \rightarrow \tau^\bullet
\end{aligned}$$

Valuations and instantiations distribute over types and size functions in the following way: $\eta \epsilon (\llbracket f(\bar{n}) \rrbracket(\tau)) = \llbracket f(\epsilon(\bar{n})) \rrbracket(\eta(\epsilon(\tau)))$.

For the sake of convenience we abbreviate $D(\epsilon(\bar{n}))$ to D_ϵ , $\eta(\epsilon(\tau))$ to $\tau_{\eta\epsilon}$ and $\eta(\epsilon(\Gamma))$ to $\Gamma_{\eta\epsilon}$.

Lemma 2 (Rewriting preserves model relation (i.e. implies set-theoretic inclusion of types)). *Let $D(\bar{n}) \vdash \tau \rightarrow \tau'$. Let a valuation ϵ and a type instantiation η be such that $\ell \models_{\tau_{\eta\epsilon}}^h w$ and D_ϵ hold. Then $\ell \models_{\tau_{\eta\epsilon}}^h w$ holds as well.*

Proof. Induction on \models . Let $\tau = \llbracket f(\bar{n}) \rrbracket(\tau'')$ and $\tau' = \llbracket f(\bar{n}) \rrbracket(\tau''')$ for some τ'' , τ''' . Let $\epsilon(\bar{n}) = \bar{n}_0$.

Suppose, $v = \text{NULL}$. Then $0 \in f'(\bar{n}_0)$ and $w = []$. Since $f(\bar{n}_0) \rightarrow f'(\bar{n}_0)$, that is $f'(\bar{n}_0) \subseteq f(\bar{n}_0)$, we have $0 \in f(\bar{n}_0)$ and $v \models_{\tau_{\eta\epsilon}}^h []$.

Let now $v = \ell$ and $w = w_{hd} :: w_{tl}$ where $h.\ell.hd \models_{\tau_{\eta\epsilon}'''}^{h \mid_{\text{dom}(h) \setminus \{\ell\}}} w_{hd}$ and $h.\ell.tl \models_{\llbracket f'(\bar{n}_0) \rrbracket^{-1}(\tau_{\eta\epsilon}''')}^{h \mid_{\text{dom}(h) \setminus \{\ell\}}} w_{tl}$. Since there is $n \in f(\bar{n}_0)$, $n \geq 1$ we have $D \vdash \tau'' \rightarrow \tau'''$ and by induction $h.\ell.hd \models_{\tau_{\eta\epsilon}''}^{h \mid_{\text{dom}(h) \setminus \{\ell\}}} w_{hd}$. Since $f(\bar{n}) \rightarrow f'(\bar{n})$ we have $f(\bar{n}) - 1 \rightarrow f'(\bar{n}) - 1$ and by induction $h.\ell.tl \models_{\llbracket f(\bar{n}_0) \rrbracket^{-1}(\tau_{\eta\epsilon}'')}^{h \mid_{\text{dom}(h) \setminus \{\ell\}}} w_{tl}$. \square

Now, stating the soundness theorem is straightforward. Informally, it states that assuming that the context zero-order variables are *valid*, i.e. indeed point to lists of the sizes mentioned in the input types, then the result in the heap will be *valid*, i.e. of the size indicated in the output type.

Theorem 1 (Soundness). For any store s , heaps h and h' , closure \mathcal{C} , expression e , value v , context Γ , quantifier-free formula D , signature Σ , type τ , size valuation ϵ , and type instantiation η such that

- the expression e terminates with the value v , i.e. in terms of operational semantics the relation $s; h; \mathcal{C} \vdash e \rightsquigarrow v; h'$ holds,
- $D, \Gamma \vdash_{\Sigma} e: \tau$ is a node in the derivation tree for some function body,
- $\text{dom}(s) = \text{dom}(\Gamma)$,
- $D(\epsilon(\bar{n}))$ holds, where \bar{n} is the set of size variables from $\text{dom}(\Gamma \cup D)$,
- $\text{Valid}_{\text{store}}(\text{dom}(s), \eta(\epsilon(\Gamma)), s, h)$ holds,

then the return value v is valid according to its return type τ , i.e.

$$\text{Valid}_{\text{val}}(v, \eta(\epsilon(\tau)), h')$$

holds.

Proof. The proof is done by induction on the size of the derivation tree for the operational-semantic judgement. One can easily check by induction that $TV(\tau) \subseteq TV(\Gamma)$. Fix a valuation $\epsilon: SV(\Gamma) \cup SV(D) \rightarrow \mathcal{R}$, and a type instantiation $\eta: TV(\Gamma) \rightarrow \tau^{\bullet}$ such that the assumptions of the lemma hold. We must show that $\text{Valid}_{\text{val}}(v, \tau_{\eta\epsilon}, h')$ holds. The full proof is given in the technical report [11]. Below, we consider only the most interesting case: the cons-branch of matching.

OSMatch-Cons: In this case $e = \text{match } l \text{ with } | \text{Nil} \Rightarrow e_1 | \text{Cons}(\text{hd}, \text{tl}) \Rightarrow e_2$ for some $l, \text{hd}, \text{tl}, e_1$ and e_2 . The typing context has the form $\Gamma = \Gamma' \cup \{l: \mathbb{L}_{f(\bar{n})}(\tau')\}$ for some Γ', τ' and f . From the operational semantics we know that $h.s(l).hd = v_{hd}$ and $h.s(l).tl = v_{tl}$ for some v_{hd} and v_{tl} , that is $s(l) \neq \text{NULL}$. Due to the validity of $s(l)$ and Lemma 1, there exists $n_0 \geq 1 \in f(\epsilon(\bar{n}))$. From the validity $s(l) \models_{\mathbb{L}_{f(\epsilon(\bar{n}))}(\tau'_{\eta\epsilon})}^h w_{hd} : w_{tl}$ the validities of v_{hd} and v_{tl} follow: $v_{hd} \models_{\tau'_{\eta\epsilon}}^h w_{hd}, v_{tl} \models_{\mathbb{L}_{f(\epsilon(\bar{n}))^{-1}}(\tau'_{\eta\epsilon})}^h w_{tl}$.

From $\text{Valid}_{\text{store}}(\text{dom}(s), \Gamma_{\eta\epsilon}, s, h)$ and the results above, we obtain

$$\text{Valid}_{\text{store}}(\text{dom}(s'), \Gamma_{\eta\epsilon}, l: \mathbb{L}_{f(\epsilon(\bar{n}))}(\tau'_{\eta\epsilon}), \text{hd}: \tau'_{\eta\epsilon}, \text{tl}: \mathbb{L}_{f(\epsilon(\bar{n}))^{-1}}(\tau'_{\eta\epsilon}), s', h)$$

where $s' = s[\text{hd} := v_{hd}][\text{tl} := v_{tl}]$. From the typing rule for e we obtain that

$$D, n_0 \geq 1 \in f(\bar{n}); \Gamma', l: \mathbb{L}_{f(\epsilon(\bar{n}))}(\tau'_{\eta\epsilon}), \text{hd}: \tau'_{\eta\epsilon}, \text{tl}: \mathbb{L}_{f(\epsilon(\bar{n}))^{-1}}(\tau'_{\eta\epsilon}) \vdash_{\Sigma} e_2: \tau_{\eta\epsilon}$$

With $\epsilon' = \epsilon[n_0 := \text{length}_h(s(l))]$ the induction hypothesis yields

$$\text{Valid}_{\text{store}}(\text{dom}(s'), \left\{ \begin{array}{l} \Gamma'_{\eta\epsilon} \cup \\ \{l: \mathbb{L}_{f(\epsilon'(\bar{n}))}(\tau'_{\eta\epsilon'}), \\ \text{hd}: \tau'_{\eta\epsilon'}, \\ \text{tl}: \mathbb{L}_{f(\epsilon'(\bar{n}))^{-1}}(\tau'_{\eta\epsilon'})\} \end{array} \right\}, s \left[\begin{array}{l} \text{hd} := v_{hd}, \\ \text{tl} := v_{tl} \end{array} \right], h) \implies \\ \text{Valid}_{\text{val}}(v, \tau_{\eta\epsilon'}, h').$$

Now from the induction hypothesis and the fact that $n_0 \notin SV(\tau)$ (and thus, $\tau_{\eta\epsilon} = \tau_{\eta\epsilon'}$), we have $\text{Valid}_{\text{val}}(v, \tau_{\eta\epsilon}, h')$. □

4 Approximation of multivalued size functions

In practice, size functions in closed forms, like $f(n) = \{n, n + 1\}$ for `insert`, are preferable to ones in the form of rewriting rules. However, inference of closed forms is a hard problem. Instead, we propose to infer their approximations given by indexed families of piecewise polynomials.

Definition. A family $\{g(\bar{n}, \bar{i})\}_{Q(\bar{n}, \bar{i})}$ of piecewise polynomials, where $Q(\bar{n}, \bar{i})$ is a quantifier-free first-order arithmetic predicate, approximates a multivalued function f if and only if for all \bar{n} in the domain of f , $f(\bar{n}) \subseteq \{g(\bar{n}, \bar{i})\}_{Q(\bar{n}, \bar{i})}$. In other words, for all $m \in f(\bar{n})$, there exists \bar{i} such that $m = g(\bar{n}, \bar{i})$ and the predicate $Q(\bar{n}, \bar{i})$ holds.

Given a multivalued size function in the form of rewriting rules, the inference procedure first generates a candidate approximating family and then checks if it indeed approximates the function.

4.1 Inferring a candidate approximating family of polynomials

To give an idea behind the interactive procedure that infers approximating families of piecewise polynomials, we start with a simple example. We show how to infer candidate polynomial lower and upper bounds for the size function of `insert` and how to construct an approximating family from it. Recall the size rewriting system for `insert`:

$$\begin{aligned} &\vdash f_{\text{insert}}(0) \rightarrow 1 \\ n \geq 1 &\vdash f_{\text{insert}}(n) \rightarrow n \mid 1 + f_{\text{insert}}(n - 1) \end{aligned}$$

Assume that p_{\min} and p_{\max} are linear, that is, that they are of the form $a_{\min}n + b_{\min}$ and $a_{\max}n + b_{\max}$, respectively. We want to find the coefficients a_{\min} , b_{\min} , a_{\max} , b_{\max} (as we did in [13] for strict polynomial (single-valued) size functions, where the lower and upper bounds were equal). To reconstruct p_{\min} , one needs to know two points on its graph, and the same holds for p_{\max} . Take $n = 1$ and $n = 2$. Evaluating the rewriting rules gives $f_{\text{insert}}(1) = \{1, 2\}$ and $f_{\text{insert}}(2) = \{2, 3\}$. Pick up the minimal values from $f_{\text{insert}}(1)$ and $f_{\text{insert}}(2)$ and assume that they are the output of p_{\min} for those inputs, i.e., that the graph of p_{\min} contains the points $(1, 1)$ and $(2, 2)$. Similarly, pick up the maximal values from $f_{\text{insert}}(1)$ and $f_{\text{insert}}(2)$ and assume that p_{\max} contains $(1, 2)$ and $(2, 3)$. We obtain two systems of equations, for a_{\min} , b_{\min} and a_{\max} , b_{\max} , respectively:

$$\begin{cases} a_{\min} + b_{\min} = 1 \\ 2a_{\min} + b_{\min} = 2 \end{cases} \quad \begin{cases} a_{\max} + b_{\max} = 2 \\ 2a_{\max} + b_{\max} = 3 \end{cases}$$

Solving these linear systems we get $a_{\min} = 1$, $b_{\min} = 0$ and $a_{\max} = 1$, $b_{\max} = 1$. Thus, we reconstruct the expressions for $p_{\min}(n) = n$ and $p_{\max}(n) = n + 1$, and the approximating family $p_{\min}(n) + i$, where $0 \leq i \leq \delta(n)$ with $\delta(n) = p_{\max}(n) - p_{\min}(n) = 1$. The rest of the job is to check whether this reconstruction approximates the solution of the rewriting rules. We discuss it in Section 4.2.

It is easy to see that we have inferred accurate bound for `insert`, i.e. the *greatest lower* and the *lowest upper* bounds for the multivalued size function. Moreover, given any $n \geq 1$, there is an evaluation path for $f_{\text{insert}}(n)$ that evaluates to $p_{\min}(n)$, and there is a path that evaluates to $p_{\max}(n)$. It explains the choice of the *step=1*: it is enough to take two consecutive natural numbers to generate the systems of equations for the coefficients of the linear lower and upper bounds.

The bounds for `insert` are one-variable and the systems of linear equations w.r.t. the polynomial coefficients are trivially consistent if one chooses different testing size values, in the example $n = 1$ and $n = 2$. The reason for this is that the matrix of such a system has a 1-variable non-zero *Vandermonde* determinant. In the multivariate case, say s variables, the consistency of the systems for p_{\min} and p_{\max} (for which the corresponding multivariate Vandermonde determinant is non-zero) depends on a more involving condition. If the testing values, i.e. the points in an s -dimensional space, lie in a so called *Node Configuration A (NCA configuration [7])*, the systems for p_{\min} and p_{\max} have unique solutions, and thus the polynomials are uniquely defined.

We describe an **NCA** configuration for the case $s = 2$ in detail. Let d be the degree of a polynomial and N_d^2 denote the amount of its coefficients. A set W of N_d^2 points on a plane lie in a *2-dimensional NCA configuration* if there exist lines $\gamma_1, \dots, \gamma_{d+1}$ in the space \mathcal{R}^2 , such that $d+1$ points of W lie on γ_{d+1} , d points of W lie on $\gamma_d \setminus \gamma_{d+1}, \dots$, and finally 1 point of W lies on $\gamma_1 \setminus (\gamma_2 \cup \dots \cup \gamma_{d+1})$. The simplest example of an **NCA** configuration on a plane is a “triangle” of points, where $d + 1$ different points lie on the line $y = 1$, d points lie on the line $y = 2, \dots$, and 1 point lies on the line $y = d + 1$. For instance, with $d = 2$ a two variable polynomial has $N_2^2 = \binom{2+2}{2} = 6$ coefficients, hence we pick up 6 points: $(1, 1), (2, 1), (3, 1), (1, 2), (2, 2)$ and $(1, 3)$.

For dimensions $s > 2$ this configuration is formulated inductively, using the notion of a hyperplane [7]. Since the definition itself is technically involved, we just give an example of an **NCA** for 3 variables ($s = 3$) and degree $d = 2$. To define a polynomial of three variables of degree 2 one needs to know $N_2^3 = \binom{2+3}{3} = 10$ coefficients, hence we need to place 10 points:

1. on the plane $x = 0$ take the “triangle” of $N_2^2 = 6$ points that lies in the 2-dimensional **NCA**, say $(0, 0, 0), (0, 0, 1), (0, 0, 2), (0, 1, 0), (0, 1, 1), (0, 2, 0)$,
2. on the plane $x = 1$ take the “triangle” of $N_1^2 = 3$ points that lies in the 2-dimensional **NCA**, say $(1, 0, 0), (1, 0, 1), (1, 1, 0)$,
3. on the plane $x = 2$ take the point $(2, 0, 0)$.

Now we give a general procedure for inferring lower and upper polynomial bounds from a given system of size rewriting rules.

- INPUT: The degrees d_{\min}, d_{\max} of hypothetical upper and lower bounds, s size variables, $\bar{n} = (n_1, \dots, n_s)$, initial test points $w_{\min}^0 = \bar{n}_{\min}^0, w_{\max}^0 = \bar{n}_{\max}^0$, steps $\epsilon_{\min}, \epsilon_{\max}$ and the system G of size rewriting rules.
- OUTPUT: A lower p_{\min} and an upper p_{\max} bound or the proposal to repeat the procedure for higher degrees and/or other $w_{\min}^0, w_{\max}^0, \epsilon_{\min}, \epsilon_{\max}$.
- PROCEDURE: 1. According to the initial points and steps, pick up N_d^s points $w = (n_1, \dots, n_s)$ in the s -dimensional space that lie in **NCA** configuration; let them constitute the sets W_{\min} . Similarly, generate W_{\max} .
2. For any $w^i \in W_{\min}$ compute the set $f_{i, \min} = f(w^i)$. Similarly, compute $f_{j, \max}$ for any $w^j \in W_{\max}$.
 3. For any f_i (f_j) pick up its minimal f_i^{\min} (maximal f_j^{\max}) values.
 - 4.1. Interpolate p_{\min} using the points (w_i, f_i^{\min}) by solving the system of linear equations w.r.t. its coefficients.
 - 4.2. Interpolate p_{\max} using the points (w_j, f_j^{\max}) by solving the system of linear equations w.r.t. its coefficients.
 5. Check whether the family $\{p_{\min}(\bar{n}) + i\}_{0 \leq i \leq (p_{\max}(\bar{n}) - p_{\min}(\bar{n}))}$ approximates the multivalued function defined by G .
 - 5.1. If “yes”: stop and output p_{\min} and p_{\max} .
 - 5.2. If “not”: pick up other parameters $d, w_{\min}^0, w_{\max}^0, \epsilon_{\min}, \epsilon_{\max}$

The choice of the parameters $w_{\min}^0, w_{\max}^0, \epsilon_{\min}, \epsilon_{\max}$ is crucial. Based on them, the procedure generates the points $(w, f(w))$. A bad choice of parameters has one of two consequences: either no bounds will be detected even if they exist, or loose bounds will be inferred. The first happens when W_{\min} (resp., W_{\max}) are constructed in such a way that there is no bound p_{\min} (resp., p_{\max}) such that its graph contains all points from W_{\min} (resp., W_{\max}). Consider, for instance, a function **divtwo**: $L_n(\alpha) \rightarrow L_{f(n)}(\alpha)$ that takes a list of length n and returns a list of length $n/2$ if n is even, and $(n-1)/2$, if n is odd. The rewriting rules for the size function are $f(0) \rightarrow 0, f(1) \rightarrow 0, n \geq 2 \vdash f(n) \rightarrow f(n-2) + 1$. Take $d = 1, n_{\min, \max}^0 = 0, \epsilon_{\min, \max} = 1$. Then $f(0) = f(1) = 0$. There is no linear upper bound that contains both, $(0, 0)$ and $(1, 0)$, points since output type $L_0(\alpha)$ is rejected by the checker. Still, linear bounds can be obtained if suitable parameters are provided. Take e.g. $n_{\min}^0 = 3, n_{\max}^0 = 2, \epsilon_{\min, \max} = 2$. Then $f(3) = 1, f(5) = 2$ and $p_{\min}(n) = (n-1)/2$, similarly $f(2) = 1, f(4) = 2$ and $p_{\max}(n) = n/2$.

Inferring rough lower (upper) bound happens when the graph of some lower (upper) bound does contain all points W_{\min} (resp., W_{\max}), but the bound itself is rough. For instance, this happens when $n^0 = 0$ for **insert**. Then $f(0) = 1, f(1) = \{1, 2\}$, so the inferred $p_{\min}(n) = 1$.

The examples above show that users should choose the parameters based on common sense and their intuitive knowledge about the functions under considerations. We recommend not to include the base-of-recursion sizes into sets of test points since these cases are usually “non-typical”.

Adaptations for inferring families of piecewise polynomials are possible. The user hints the inference system on which areas P_i she assumes different pieces of polynomial bounds. Different parameters must be provided for each piece.

As a more elaborated example, consider the inference procedure for the function rel (defined in Section 3.3). The inferred size rewriting system is:

$$\begin{aligned} & \vdash f_{\text{rel}_1}(0, m) \rightarrow 0 \\ n \geq 1 \vdash f_{\text{rel}_1}(n, m) & \rightarrow f_{\text{rel_pairs}_1}(m) + f_{\text{rel}_1}(n-1, m) \quad n \geq 1 \vdash f_{\text{rel}_2}(n, m) \rightarrow f_{\text{rel_pairs}_2}(m) \end{aligned}$$

We show how to infer the family $\{i\}_{0 \leq i \leq nm}$. A quadratic polynomial $q(n, m) = a_{20}n^2 + a_{02}m^2 + a_{11}nm + a_{10}n + a_{01}m + a_{00}$ of two variables has 6 coefficients, so to define the polynomial one needs to know 6 points (n_i, m_i, q_i) on the graph of q . The coefficients are computed as the solution of the system of linear equations $q_i = a_{20}n_i^2 + a_{02}m_i^2 + a_{11}n_im_i + a_{10}n_i + a_{01}m_i + a_{00}$, where $1 \leq i \leq 6$. For instance, one can take the points (n, m) from $\{(1, 1), (2, 1), (3, 1), (1, 2), (2, 2), (1, 3)\}$. Then, the linear system w.r.t. the coefficients of q has the form

$$\begin{aligned} a_{20} + a_{02} + a_{11} + a_{10} + a_{01} + a_{00} &= q(1, 1) \\ 4a_{20} + a_{02} + 2a_{11} + 2a_{10} + a_{01} + a_{00} &= q(2, 1) \\ 9a_{20} + 3a_{02} + 3a_{11} + 3a_{10} + a_{01} + a_{00} &= q(3, 1) \\ a_{20} + 4a_{02} + 2a_{11} + a_{10} + 2a_{01} + a_{00} &= q(1, 2) \\ 4a_{20} + 4a_{02} + 4a_{11} + 2a_{10} + 2a_{01} + a_{00} &= q(2, 2) \\ a_{20} + 9a_{02} + 3a_{11} + a_{10} + 3a_{01} + a_{00} &= q(1, 3) \end{aligned}$$

To reconstruct p_{\min} and p_{\max} , consider all possible evaluation paths for f_{rel} at these points, using the fact that for any fixed n, m there is only finite number of indices j satisfying $0 \leq j \leq m$.

$$\begin{aligned} f_{\text{rel}}(1, 1) = j + 0 &= \{0, 1\} \\ f_{\text{rel}}(2, 1) = j + f_{\text{rel}}(1, 1) &= \{0, 1, 2\} \\ f_{\text{rel}}(3, 1) = j + f_{\text{rel}}(2, 1) &= \{0, 1, 2, 3\} \\ f_{\text{rel}}(1, 2) = j + 0 &= \{0, 1, 2\} \\ f_{\text{rel}}(2, 2) = j + f_{\text{rel}}(1, 2) &= \{0, 1, 2, 3, 4\} \\ f_{\text{rel}}(1, 3) = j + 0 &= \{0, 1, 2, 3\} \end{aligned}$$

Thus, for the coefficients of p_{\max} one has the system

$$\begin{aligned} a_{20} + a_{02} + a_{11} + a_{10} + a_{01} + a_{00} &= 1 \\ 4a_{20} + a_{02} + 2a_{11} + 2a_{10} + a_{01} + a_{00} &= 2 \\ 9a_{20} + 3a_{02} + 3a_{11} + 3a_{10} + a_{01} + a_{00} &= 3 \\ a_{20} + 4a_{02} + 2a_{11} + a_{10} + 2a_{01} + a_{00} &= 2 \\ 4a_{20} + 4a_{02} + 4a_{11} + 2a_{10} + 2a_{01} + a_{00} &= 4 \\ a_{20} + 9a_{02} + 3a_{11} + a_{10} + 3a_{01} + a_{00} &= 3 \end{aligned}$$

The solution is $(0, 0, 1, 0, 0, 0)$, so $p_{\max}(n, m) = nm$. The system for p_{\min} has all zeros on its right hand side, thus $p_{\min} = 0$. The inferred family is indeed $\{i\}_{0 \leq i \leq nm}$, which approximates the multivalued size function f_{rel_1} .

4.2 Checking if a family approximates a size function

Checking an inferred family is similar to type checking types annotated with families of piecewise polynomials directly [12]. In that type system, for instance, the output type of `insert` is $\mathbb{L}_{n+i}^{0 \leq i \leq 1}(\alpha)$.

We show that, *given a multivalued size function and some indexed family of piecewise polynomials, there is a set of first-order arithmetic entailments such that their satisfiability implies that the family approximates the size function.* Such predicates are obtained by substituting indexed families of polynomials, which are to be checked as approximations, for the corresponding multivalued-function symbols in the rewriting rules. For instance, verifying whether the family $\{n+i\}_{0 \leq i \leq 1}$ approximates $f_{\text{insert}}(n)$ reduces to checking the entailments

$$\begin{aligned} n = 0 & \quad \vdash 1 = n + ?i \wedge 0 \leq ?i \leq 1 \\ n \geq 1 & \quad \vdash n = n + ?i \wedge 0 \leq ?i \leq 1 \\ n \geq 1, 0 \leq j \leq 1 & \vdash 1 + (n-1) + j = n + ?i \wedge 0 \leq ?i \leq 1 \end{aligned}$$

Checking succeeds by instantiating $?i$ to 1, 0 and j , respectively. Substitution of an indexed family of polynomials $\{g(\bar{n}, \bar{i})\}_{Q(\bar{n}, \bar{i})}$ for a multivalued-function symbol f is defined in the usual way. Let an arithmetic expression $\varepsilon(\bar{n}, \bar{i})$ contain size variables \bar{n} , indices \bar{i} (such that $Q(\bar{n}, \bar{i})$), symbols $+$, $-$, $*$ and symbols of multivalued functions. Examples of $\varepsilon(\bar{n}, \bar{i})$ are $1 + f_{\text{insert}}(n-1)$, $f_{\text{insert}}(n-2) + f_{\text{insert}}(n-1)$ and $f_{\text{insert}}(m-1, n) + i$, with $0 \leq i \leq 1$. Substituting the family $\{n+i\}_{0 \leq i \leq 1}$, for f_{insert} in the first expression results in $1 + n - 1 + i = n + i$. In the second expression it gives $n - 2 + i_1 + n - 1 + i_2 = 2n - 3 + i_1 + i_2$, where $0 \leq i_1, i_2 \leq 1$. The substitution $\{n+j\}_{0 \leq j \leq m}$, for $f_{\text{insert}}(m-1, n)$ in the third expression results in $n + j + i$ with $0 \leq j \leq m-1$ and $0 \leq i \leq 1$.

We generalise substitution to types $\tau = \mathbb{L}_{\varepsilon_1}^{Q_1(\bar{n}, \bar{i}_1)}(\dots \mathbb{L}_{\varepsilon_k}^{Q_k(\bar{n}, \bar{i}_k)}(\alpha) \dots)$, which annotated by indexed families of expressions, in the natural way:

$$[\tau] = \mathbb{L}_{[\varepsilon_1]}^{Q'_1(\bar{n}, \bar{i}_1, \bar{j}_1)}(\dots \mathbb{L}_{[\varepsilon_k]}^{Q'_k(\bar{n}, \bar{i}_k, \bar{j}_k)}(\alpha) \dots)$$

To construct predicates to check candidate approximations, one also needs the notion of subtyping for types annotated by indexed families of piece-wise polynomials directly [12]. Examples of subtypings in those type system are $\vdash \mathbb{L}_{n+i}^{0 \leq i \leq 1}(\alpha) \preceq \mathbb{L}_{n+i}^{0 \leq i \leq 2}(\alpha)$ and $n = 0 \vdash \mathbb{L}_n(\mathbb{L}_i^{0 \leq i \leq 2}(\alpha)) \preceq \mathbb{L}_n(\mathbb{L}_2(\alpha))$. Let

$$\begin{aligned} T &= \mathbb{L}_{g^1(\bar{n}, \bar{i}^1)}^{Q^1(\bar{n}, \bar{i}^1)}(\dots \mathbb{L}_{g^k(\bar{n}, \bar{i}^k)}^{Q^k(\bar{n}, \bar{i}^k)}(\alpha) \dots) \\ T' &= \mathbb{L}_{g'^1(\bar{n}, \bar{j}^1)}^{Q'^1(\bar{n}, \bar{j}^1)}(\dots \mathbb{L}_{g'^k(\bar{n}, \bar{j}^k)}^{Q'^k(\bar{n}, \bar{j}^k)}(\alpha) \dots) \end{aligned}$$

Then $D \vdash T' \preceq T$ holds if and only if

$$\forall \bar{n} \bar{j}^1. D(\bar{n}) \wedge Q'^1(\bar{n}, \bar{j}^1) \implies \exists \bar{i}^1. g'^1(\bar{n}, \bar{j}^1) = g^1(\bar{n}, \bar{i}^1) \wedge Q^1(\bar{n}, \bar{i}^1)$$

and if, moreover, there exists \bar{j}^1 such that $D(\bar{n}) \wedge Q'^1(\bar{n}, \bar{j}^1)$ and $g'^1(\bar{n}, \bar{j}^1) \geq 1$ then

$$D \vdash \mathbb{L}_{g'^2(\bar{n}, \bar{j}^2)}^{Q'^2(\bar{n}, \bar{j}^2)}(\dots \mathbb{L}_{g'^k(\bar{n}, \bar{j}^k)}^{Q'^k(\bar{n}, \bar{j}^k)}(\alpha) \dots) \preceq \mathbb{L}_{g^2(\bar{n}, \bar{i}^2)}^{Q^2(\bar{n}, \bar{i}^2)}(\dots \mathbb{L}_{g^k(\bar{n}, \bar{i}^k)}^{Q^k(\bar{n}, \bar{i}^k)}(\alpha) \dots)$$

Let $\tau = L_{f_1}(\dots L_{f_s}(\alpha))$ and $D \vdash \tau \rightarrow \tau'$. To check if a family $\{g_i(\bar{n}, \bar{i}_i)\}_{Q(\bar{n}, \bar{i}_i)}$ approximates f_i , for all $1 \leq i \leq s$, one uses the following lemma.

Lemma 3 (Checking). *Let $[\phi_l](\bar{n}, \bar{j}_l)$, with $1 \leq l \leq t$, approximate multivalued-function symbols $\{\phi_1(\bar{n}), \dots, \phi_t(\bar{n})\}$ that occur τ' but not in τ . Let $[\tau]$ and $[\tau']$ be obtained by substituting g_i and $[\phi_l]$ for the corresponding function symbols f_i and ϕ_l . Then $D \vdash [\tau'] \preceq [\tau]$ implies that $\{g(\bar{n}, \bar{i}_i)\}_{Q(\bar{n}, \bar{i}_i)}$ approximates f_i , where $1 \leq i \leq s$.*

Proof. By induction on the length of the rewriting chain for an arbitrary i , fixing some $m \in f_i(\bar{n})$.

As an example, checking whether $\{i\}_{0 \leq i \leq nm}$ approximates $f_{1, \text{rel}}$ reduces to checking the entailments

$$\begin{array}{l} n = 0 \\ n \geq 1, 0 \leq j' \leq m, 0 \leq j \leq (n-1)m \end{array} \vdash \begin{array}{l} 0 = ?i \wedge 0 \leq ?i \leq nm \\ j' + j = ?i \wedge 0 \leq ?i \leq nm \end{array}$$

The decidability problem of checking whether an indexed family of piecewise polynomials approximates a given multivalued size function is treated similarly to the decidability of type checking for the system annotated with such families directly [12]. In particular, checking is decidable when function definitions satisfy the syntactical condition from [13] and output approximations are finite families of polynomials. Also, checking is decidable for indexed families of piecewise linear polynomials with indices delimited by linear predicates.

5 Feasibility of analysing of a typical list library

As a small feasibility study, we applied our analysis to a typical list library, The functions were adapted from Hugs' list library, version September 2006. Since we have an intermediate language we first needed to make some assumptions.

Firstly, we assume strict semantics, which means that we cannot deal with infinite lists. Hence functions like `repeat` were omitted. Secondly, it must be possible to translate the function into our language. Our type system requires that inner lists all have the same length, which is not the case for a general version of e.g. `concat`. This restriction may be removed in a future version of our work. Thirdly, we ignore classes of types like `Eq` and `Ord` and we write the functions uncurried. Finally, we write *interface types* where the family is given as an annotation that can be inferred directly from the set of term rewriting rules as shown above. Annotations in interface types are approximations studied in Section 4.

Many functions (like `head`, `null`, `length`, `elem`, `notElem`, `and`, `or`, `any`, `all`, `sum`, `product`, `maximum`, `minimum`, `isPrefix`, `isSuffix`, `isInfix`, and `atIndex`) do not return lists and thus they are analysable but not interesting from the size dependency point of view. E.g., the type for `length` is $L_n(\alpha) \rightarrow \text{Int}$.

The family of fold functions are parametric on the type of the result and hence not suitable for our analysis. Even an specialised version that returns lists

would still be out of the scope of our analysis because the length of the output would depend on the length of the list returned by the high-order parameter.

Other functions (like `append`, `tail`, `init`, `map`, `reverse` and `sort`, and restricted versions of `concat` and `union`) are shapely, i.e. they have an exact polynomial size function. These functions are typable in the type systems developed in [13, 14]. Of these functions we only give the type of `append`: $L_n(\alpha) \times L_m(\alpha) \rightarrow L_{n+m}(\alpha)$.

Some functions have a precise size dependency but they need \max_0 or they have first-order arguments, and thus they cannot be handled by our previous systems. Now we can type them as follows:

$$\begin{aligned} \text{intersperse} &: \alpha \times L_n(\alpha) \rightarrow L_{\max_0(2*n-1)}(\alpha) \\ \text{scanl} &: (\alpha \times \beta \rightarrow \alpha) \times \alpha \times L_n(\beta) \rightarrow L_{n+1}(\alpha) \\ \text{scanl1} &: (\alpha \times \alpha \rightarrow \alpha) \times L_n(\alpha) \rightarrow L_n(\alpha) \end{aligned}$$

For functions with a multivalued size function, an indexed family of polynomials is needed to express the possible output sizes. Several functions have a list of size n among their arguments and perform some filtering of the elements, returning a list of length at most n , i.e., $L_i^{0 \leq i \leq n}(\alpha)$. The functions `takeWhile`, `dropWhile`, `filter`, `findIndices`, `elemIndices`, `nub` and `nubBy` fall in this category.

Probably more interesting are the types of the functions that may delete some elements. `delete` and `deleteBy` take a list of size n and return a list with maybe one element less: $L_{\max_0(n-1)}^{0 \leq i \leq 1}(\alpha)$; `deleteFirstBy` and `(\)` take a list of size n and a list of size m and delete at most m elements from the first list: $L_{\max_0(n-i)}^{0 \leq i \leq m}(\alpha)$; finally, given lists of size n and m , `intersect` and `intersectBy` return a list of length at most $\min(n, m)$: $L_i^{0 \leq i \leq \max_0(n, \max_0(n-m))}(\alpha)$.

Adding algebraic data types to our language, many other functions could be analysed. There are, of course, functions whose sized types cannot be expressed in our type system or our procedure cannot deal with them. Our type system cannot express types where the size of the output depends on a higher-order parameter (this is the case for `concatMap`). Furthermore, we cannot express types where the size of the result depends on the *value* of the arguments, i.e., the size cannot be determined statically (like `unfoldr`).

6 Related Work

This research extends our work [13, 16, 14] about shapely function definitions that have a single-valued, exact input-output polynomial size functions. Our non-monotonic framework resembles [2] in which the authors describe *monotonic* resource consumption for Java bytecode by means of Cost Equation Systems (CESs), which are similar to, but more general than recurrence equations. CESs express the cost of a program in terms of the size of its input data. In a further step, a closed-form solution or upper bound can sometimes be found by using existing Computer Algebra Systems, such as *Maple* and *Mathematica*. This work is continued by the authors in [1], where mechanisms for solving and upper bounding CESs are studied. However, they do not consider non-monotonic size functions.

Our approach is related to size analysis with polynomial quasi-interpretations [6, 3]. There, a program is interpreted as a *monotonic* polynomial extended with the max operation. For instance, $\text{Cons}(\text{hd}, \text{tl})$ is interpreted as $T + 1$, where T is a numerical variable abstracting tl . Using such interpretations one obtains upper monotonic-polynomial bounds for size functions. The main difference with our approach is that we are interested in non-monotonic lower and upper bounds. In particular, we may infer the size function $(n - m)^2$ for $\text{sqdiff} : \mathbb{L}_n(\alpha) \times \mathbb{L}_m(\alpha) \rightarrow \mathbb{L}_{(n-m)^2}(\alpha)$ (in this simple example the tight lower and upper bounds coincide), see e.g. [16]. To our knowledge, non-monotonic quasi-interpretations have not been studied for size analysis, but only for proving termination [9]. In this work one considers some unspecified algorithmically decidable classes of non-negative and negative polynomials and introduces abstract variables for the rest.

The EmBounded project aims to identify and certify resource-bounded code in *Hume*, a domain-specific high-level programming language for real-time embedded systems. In his thesis, Pedro Vasconcelos [17] uses abstract interpretation to automatically infer linear approximations of the sizes of recursive data types and the stack and heap of recursive functions written in a subset of *Hume*.

Several papers have studied programming languages with *implicit computational complexity* properties [8, 5]. This line of research is motivated both by the perspective of automated complexity analysis and by fundamental goals, in particular to give natural characterisations of complexity classes, like PTIME or PSPACE. Resource analysis may be performed within a *Proof Carrying Code* framework. In [4] the authors introduce resource policies for mobile code to be run on smart devices. Policies are integrated into a proof-carrying code architecture. Two forms of policies are used: *guaranteed policies* which come with proofs and *target policies* which describe limits of the device.

7 Conclusions and Future Work

This paper presents a size-aware type system that describes multivalued size functions expressing the dependency between the sizes of inputs and the output size of a function definition. It allows to approximate multivalued output size functions via indexed *non-monotonic* polynomials augmented with the \max_0 operation. This feature greatly increases the applicability of our earlier size analysis, which was limited to exact sizes. The extra expressibility comes at a cost: we have crossed the border of decidability. However, this does not make the analysis infeasible in practice.

Our next step will be to extend our prototype implementation, available via www.aha.cs.ru.nl, to cope with different output sizes and apply it in some case studies. After that, as part of the AHA project, we will transfer our size analysis results to the world of imperative programs.

References

1. E. Albert, P. Arenas, S. Genaim, and G. Puebla. Automatic Inference of Upper Bounds for Recurrence Relations in Cost Analysis. In *Static Analysis, 15-th International Symposium*, volume 5079 of *LNCS*, pages 221–237, 2008.
2. E. Albert, P. Arenas, S. Genaim, G. Puebla, and D. Zanardini. Cost Analysis of Java Bytecode. In *16th European Symposium on Programming, ESOP'07*, volume 4421 of *LNCS*, pages 157–172. Springer, 2007.
3. R. M. Amadio. Synthesis of max-plus quasi-interpretations. *Fundamenta Informaticae*, 65(1-2):29–60, 2004.
4. D. Aspinall and K. MacKenzie. Mobile Resource Guarantees and Policies. In G. Barthe, B. Grégoire, M. Huisman, and J.-L. Lanet, editors, *CASSIS 2005*, volume 3956 of *LNCS*, pages 16–36. Springer, 2006.
5. V. Atassi, P. Baillot, and K. Terui. Verification of Ptime Reducibility for System F Terms: Type Inference in Dual Light Affine Logic. *Logical Methods in Computer Science*, 3(4), 2007.
6. G. Bonfante, J.-Y. Marion, and J.-Y. Moyen. Quasi-interpretations, a way to control resources. *Theoretical Computer Science*, 2005.
7. C. K. Chui and M.-J. Lai. Vandermonde determinants and lagrange interpolation in R^s . *Nonlinear and convex analysis*, pages 23–35, 1987.
8. M. Gaboardi, J.-Y. Marion, and S. Ronchi Della Rocca. A logical account of PSPACE. In *35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages POPL 2008, San Francisco, January 10-12, 2008, Proceedings*, pages 121–131, 2008.
9. N. Hirokawa and A. Middeldorp. Polynomial interpretations with negative coefficients. In *Artificial Intelligence and Symbolic Computation*, volume 3249 of *LNCS*, 2004.
10. C. Okasaki. *Purely Functional Data Structures*. Cambridge University Press, 1998.
11. O. Shkaravska, M. van Eekelen, and A. Tamalet. Collected size semantics for functional programs. Technical Report ICIS-R08021, Radboud University Nijmegen, November 2008.
12. O. Shkaravska, M. van Eekelen, and A. Tamalet. Polynomial size complexity analysis with families of piecewise polynomials. Technical Report ICIS-R08020, Radboud University Nijmegen, November 2008.
13. O. Shkaravska, R. van Kesteren, and M. van Eekelen. Polynomial Size Analysis for First-Order Functions. In S. R. D. Rocca, editor, *Typed Lambda Calculi and Applications (TLCA'2007), Paris, France*, volume 4583 of *LNCS*, pages 351–366. Springer, 2007.
14. A. Tamalet, O. Shkaravska, and M. van Eekelen. Size Analysis of Algebraic Data Types. In M. Morazán, editor, *Selected Papers of the 9th International Symposium on Trends in Functional Programming (TFP'08)*. Intellect Publishers. 2008, to appear.
15. M. van Eekelen, O. Shkaravska, R. van Kesteren, B. Jacobs, E. Poll, and S. Smetters. AHA: Amortized Heap Space Usage Analysis. In M. Morazán, editor, *Selected Papers of the 8th International Symposium on Trends in Functional Programming (TFP'07), New York, USA*, pages 36–53. Intellect Publishers, UK, 2007.
16. R. van Kesteren, O. Shkaravska, and M. van Eekelen. Inferring static non-monotonically sized types through testing. In *Proceedings of 16th International Workshop on Functional and (Constraint) Logic Programming (WFLP'07), Paris, France*, volume 216C of *ENTCS*, pages 45–63, 2007.

17. P. B. Vasconcelos. *Space Cost Analysis Using Sized Types*. PhD thesis, School of Computer Science, University of St. Andrews, August 2008.