Collected Size Semantics for Functional Programs over Polymorphic Nested Lists *

O. Shkaravska[1], M. van Eekelen[2,3], A. Tamalet[4]

[1] Max Planck Institute for Psycholinguistics
[2] Institute for Computing and Information Sciences (iCIS), Radboud University Nijmegen
[3] School for Computer Science, Open University of the Netherlands
[4] Globant, Rosario, Argentina
M.vanEekelen@cs.ru.nl

Abstract. Size analysis is an important prerequisite for heap consumption analysis. This paper is a part of ongoing work about typing support for checking output-on-input size dependencies for function definitions in a strict functional language. A significant restriction for our earlier results is that *inner* data structures (e.g. in a list of lists) all must have the same size. Here, we make a big step forwards by overcoming this limitation via the introduction of higher-order size annotations such that variate sizes of inner data structures can be expressed.

1 Introduction

Bounds on the resource consumption of programs can be used, and are often needed, to ensure correctness and security properties, in particular in devices with scarce resources as mobile phones and smart cards. Both the memory and the time consumption of a program often depend on the sizes of input and intermediate data. In many cases, analysis of size behavior is an important part of resource analysis, e.g. using lower and upper bounds of size dependencies to check and infer non-linear bounds on heap consumption [16, 13]. Here, we consider size analysis of *strict* functional programs over polymorphic *lists*. A size dependency of a program is a *size function* that maps the size of inputs onto the sizes of the corresponding output. For instance, the typical size dependency for a program `append`, that appends two lists of length n and m, is the function $\mathit{append}(n, m) = n + m$.

This paper is devoted to collecting size dependencies using *multivalued* size functions. Multivalued size functions can be defined by conditional multiple-choice rewriting rules [12]. These multivalued size functions are used to annotate types. They make it possible to express that there can be more than one possible

* This work was made possible by the AHA project [16] which was sponsored by the Netherlands Organisation for Scientific Research (NWO) under grant nr. 612.063.511 and by the CHARTER project, EU Artemis nr. 100039.

output size (like e.g. in the case of inserting an element to a list if it is not there already: the result will either have the same size or it will be one element larger).

However, the type system from [12] only covers programs over 'matrix-like' structures, e.g. $\mathsf{L}_n(\mathsf{L}_m(\alpha))$ leaving no way to express variate sizes of internal lists. This substantially restricted application of the approach, since the case of programs over lists of lists with variate lengths is the most frequent one.

In this paper, *we remove that restriction and generalise the approach to cover all polymorphic programs over lists for which the size(s) of an output depend only the size of first-order inputs*. We use an ML-like strict language which is defined in Section 2. In Section 3 we define the type system which allows *size variables of higher-order kinds*, such that, e.g., a size variable M in the type $\mathsf{L}_n(\mathsf{L}_M(\alpha))$ represents the size $M(pos)$ of an internal list depending on its position pos in the outer list, where $0 \leq pos \leq n-1$. Moreover, we extend (checking and inferring of) multivalued size functions allowing them to be defined with higher-order rewriting rules. We define soundness and sketch its proof. Section 4 gives a procedure for the generation of polynomial lower and upper bounds and a set of polynomials that covers the function defined by the higher-order rewriting rules. Section 5 relates our work to other resource analysis work.

2 Language

The type system is designed for a strict functional language over integers, booleans and (polymorphic) lists. Language expressions are defined by the grammar below 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 a 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.

$$
\begin{array}{lll}
\textit{Basic } b & ::= & c \mid \mathsf{unop\ x} \mid \mathsf{x\ binop\ y} \mid \mathsf{Nil} \mid \mathsf{Cons(z,\ l)} \mid \mathsf{f}(\mathsf{g}_1, \ldots, \mathsf{g}_l, \mathsf{z}_1, \ldots, \mathsf{z}_k) \\
\textit{Expr } e & ::= & b \mid \mathsf{if\ x\ then}\ e_1\ \mathsf{else}\ e_2 \mid \mathsf{let\ z} = b\ \mathsf{in}\ e_1 \\
& & \mid \mathsf{match\ l\ with} \mid \mathsf{Nil} \Rightarrow e_1 \\
& & \qquad\qquad \mid \mathsf{Cons}(\mathsf{z}_{\mathsf{hd}}, \mathsf{l}_{\mathsf{tl}}) \Rightarrow e_2 \\
& & \mid \mathsf{letfun\ f}(\mathsf{g}_1, \ldots, \mathsf{g}_l, \mathsf{z}_1, \ldots, \mathsf{z}_k) = e_1\ \mathsf{in}\ e_2
\end{array}
$$

The syntax distinguishes between zero-order let-binding of variables and higher-order letfun-binding of functions. 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 general language like ML may be compiled into.

3 Type System

We consider a type system constituted from zero-order and higher-order types and typing rules for each program construct. Size annotations represent lengths

of finite lists. Syntactically, size annotations are (higher-order) arithmetic expressions over constants, size variables and multivalued-function symbols. Let $\mathcal{R}$ be a numerical ring used to express and solve the size equations. Constants and size variables are *layered*:

- *The layer zero* is empty. It corresponds to the unsized types Int, Bool and α,where α is a type variable. Elements of these types have no size annotations.
- The *first layer* is the type $\mathcal{R}^{(1)} = \mathcal{R}$ of numerical zero-order constants (i.e. integers) and size variables, denoted by a and n, respectively (possibly decorated with subscripts). They represent lengths of outermost lists. Examples are $\mathsf{L}_5(\alpha)$ with $a = 5$, or $\mathsf{L}_n(\mathsf{L}_5(\alpha))$.
- The *second layer* consists of numerical first-order constants and variables of type $\mathcal{R}^{(2)} = \mathcal{R} \to \mathcal{R}$, denoted by B and M, respectively. They represent lengths of nested lists in a list. For instance, in the typing $\mathsf{l} : \mathsf{L}_n(\mathsf{L}_M(\alpha))$ the function $\lambda\, pos.M(pos)$ represents the length of the pos-th list in the master list l. Indexes start at 0, so $M(0)$ is the length head of the master list, and $M(n-1)$ is the length of its last element. Constants of the type $\mathcal{R} \to \mathcal{R}$ may be defined by an arithmetic expression or by a table. For instance, in $[\,[1,2],\,[3,4,5],\,[\,]\,]$ the length of the master list is $a = 3$ and B is given by the table $B(0) = 2$, $B(1) = 3$, $B(2) = 0$. For $pos \geq 2$, $B(pos)$ may be any arbitrary number.
- In general, the s-th *layer* consists of numerical $(s-1)$-th-order constants and variables of type $\mathcal{R}^{(s)} = \mathcal{R} \to \mathcal{R}^{(s-1)}$, denoted by a^s and n^s. They represent lengths of lists of "nestedness" s. For instance in $\mathsf{l} : \mathsf{L}_{n^1}(\ldots \mathsf{L}_{n^s}(\alpha)\ldots)$ the function $n^s(i_1)\ldots(i_{s-1})$ represents the length of the i_{s-1}-th list in the i_{s-2}-th list in ... in the i_1-th list of the master list l.

Let $\mathcal{R}^*$ denote the union $\bigcup_{s=1}^{\infty} \mathcal{R}^{(s)}$ and let n^* range over size variables of $\mathcal{R}^*$. Let $\overline{n}^*$ denote a vector of variables $(n_1^*, \ldots, n_k^*)$ for some $k \geq 0$.

Layering is extended to multivalued size functions, according to their return types (but not their parameter types):

- A function of the layer 1 is a function $f : (\mathcal{R}^*)^k \to 2^{\mathcal{R}}$ for some $k \geq 0$ that represents all possible sizes (depending on parameters from $(\mathcal{R}^*)^k$) of outer lists. For instance, if $f(n) = \{n,\, n+1\}$ in $\mathsf{l} : \mathsf{L}_{f(n)}(\alpha)$, then the length of l is either n or $n+1$. Another example has been given in the introduction: in the output type of the function $\mathsf{concat} : \mathsf{L}_n(\mathsf{L}_m(\alpha)) \to \mathsf{L}_{\mathit{concat}(n,\,M)}(\alpha)$, we have a function concat: $\mathcal{R}^{(1)} \times \mathcal{R}^{(2)} \to 2^{\mathcal{R}}$.
- A function of the layer s is a function of the type $(\mathcal{R}^*)^k \to (\mathcal{R} \to \ldots \to \mathcal{R} \to 2^{\mathcal{R}})$ that maps parameters from $(\mathcal{R}^*)^k$ to $s-1$-order multivalued functions of the type $\mathcal{R} \to \ldots \to \mathcal{R} \to 2^{\mathcal{R}}$. Its value $f(\overline{n}^*)(pos_1)\ldots(pos_{s-1})$ defines all possible sizes of the pos_{s-1} list in the pos_{s-2}-th list ... in the pos_1-the list of the master list.

If a function is single-valued, we will omit the set brackets on its output. As an example, consider the function definition for $\mathsf{tails} : \mathsf{L}_n(\alpha) \to \mathsf{L}_{\mathit{tails}_1(n)}(\mathsf{L}_{\mathit{tails}_2(n)}(\alpha))$ that creates the list of all non-empty tails of the input list:

$$\begin{array}{l}\mathsf{tails(l)} = \mathsf{match\ l\ with} \mid \mathsf{Nil} \Rightarrow \mathsf{Nil} \\ \qquad\qquad\qquad\qquad \mid \mathsf{Cons(hd,\ tl)} \Rightarrow \mathsf{let\ l'} = \mathsf{tails(tl)\ in\ Cons(l,\ l')}\end{array}$$

For instance, on [1, 2, 3] it outputs [[1, 2, 3], [2, 3], [3]]. It is easy to see that $\mathit{tails}_1 : \mathcal{R} \to 2^{\mathcal{R}}$ is the identity $\mathit{tails}_1(n) = n$ and $\mathit{tails}_2 : \mathcal{R} \to (\mathcal{R} \to 2^{\mathcal{R}})$ for $n \geq 1$ is defined by $\mathit{tails}_2(n)(pos) = n - pos$, if $0 \leq pos \leq n-1$.

A *size expression* p is constructed from size constants, variables, multivalued-function symbols and operations of all layers. We will denote functions of the first and second layers via f and g, respectively. Admissible operations are arithmetic operations $+$, $-$, $*$, λ-abstraction and application. Layering is defined for size expressions as it has been defined for multivalued size functions. *A size expression is of layer s if it returns a value of order $s-1$ of type $\mathcal{R} \to \ldots \to \mathcal{R} \to 2^{\mathcal{R}}$.* When necessary, we denote a size expression of the layer s via p^s. For instance:

$$\begin{array}{l} p^1 ::= a \mid n,\ pos \mid f(p_1, \ldots, p_k) \mid p^1\,\{+, -, *\}\,p^1 \mid p^2(p^1) \\ p^2 ::= B \mid M \mid g(p_1, \ldots, p_k) \mid \lambda\, pos.\, p^1 \mid p^3(p^1) \end{array}$$

where pos is a special variable of type $\mathcal{R}$ used to denote the position of an element in a list. We also assume that constants (e.g. a) and size variables (e.g. n) represent singleton sets.

Zero-order annotated types are defined as follows:

$$\begin{array}{ll} \tau^0 & ::= \mathsf{Int} \mid \mathsf{Bool} \mid \alpha \\ \tau^{s',\,s} & ::= \mathsf{L}_{p^{s'}}(\mathsf{L}_{p^{s'+1}}(\ldots \mathsf{L}_{p^s}(\tau^0)\ldots)) \ \text{ for } \ 1 \leq s' \leq s, \\ \tau^s & ::= \tau^{1,\,s} \end{array}$$

where α is a type variable. It is easy to see that $\tau^{s',\,s} = \mathsf{L}_{p^{s'}}(\tau^{s'+1,\,s})$. The types τ^0 and τ^s are types of program expressions, but $\tau^{s',\,s}$ are only used in definitions and proofs but not in function types.

Let τ ranges over zero-order types. The sets $TV(\tau)$ and $SV(\tau)$ of type and size variables of a type τ are defined inductively in the obvious way. All empty lists of the same underlying type represent the same data structure. So, $SV(\mathsf{L}_0(\tau)) = \emptyset$ forall τ and $\mathsf{L}_0(\mathsf{L}_m(\mathsf{Int}))$ represents the same structure as $\mathsf{L}_0(\mathsf{L}_0(\mathsf{Int}))$.

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

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

The semantics of ground types is defined in Section 3. Here we give some examples: $\mathsf{L}_2(\mathsf{Bool})$, $\mathsf{L}_2(\mathsf{L}_B(\mathsf{Bool}))$, and $\mathsf{L}_{\mathit{concat}(2,\,B)}(\mathsf{Bool})$, where $B(pos) = pos$ on $0 \leq pos \leq 1$. It is easy to see that $\mathit{concat}(2, B) = \{0\} + \{1\} = \{1\}$. Examples of their inhabitants are [True, True], [[], [True]] and [True], respectively. Examples of non-ground types are α, $\mathsf{L}_n(\mathsf{Int})$, $\mathsf{L}_n(\mathsf{L}_M(\mathsf{Bool}))$ and $\mathsf{L}_{\mathit{concat}(n,M)}(\mathsf{Bool})$ with unspecified n and M.

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

$$\mathit{FunctionTypes} \quad \tau^f \ ::= \ \tau_1^f \times \ldots \times \tau_{k'}^f \times \tau_1^\circ \times \ldots \times \tau_k^\circ \to \tau_0$$

where k' may be zero (i.e. the list $\tau_1^f, \ldots, \tau_{k'}^f$ is empty) and $SV(\tau_0)$ contains only size variables of $\tau_1^\circ, \ldots, \tau_k^\circ$.

Multivalued size functions f in the output types of function signatures in general are defined by conditional rewriting rules, as we have seen in the introduction. It is desirable to find closed forms for functions defined by such rewriting rules.

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: $SV(\Gamma) = \bigcup_{\mathsf{x} \in dom(\Gamma)} SV(\Gamma(\mathsf{x}))$.

Heap Semantics

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. The full heap semantics are given in [11] in which also the following lemma is proven by induction on the relation $\models$.

Lemma 1 (Consistency of model relation). *The relation* $\mathtt{adr} \models^{h}_{\mathsf{L}_{p^1(\overline{n}_0^*)}(\tau\bullet)} w$ *implies that* $length_h(\mathtt{adr}) \in p^1(\overline{n}_0^*)$.

Operational semantics of program expressions

The operational semantics is standard. It extends the semantics from [14] 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, $Store\ s\colon ProgramVars \rightharpoonup Val$.

Using a heap, a frame store and mapping $\mathcal{C}$ (*closures*) from function names to function bodies, the operational semantics of program expressions is defined inductively in a standard way. The full operational semantics is given in the technical report [11].

3.1 Typing rules

A typing judgement is a relation of the form $D, \Gamma \vdash_{\Sigma} e : \tau$, i.e. given a set of constraints D, a zero-order context Γ and a higher-order signature Σ, an expression e has a type τ. The set D of disequations and memberships is relevant

only when a rule for pattern-matching and constructors are applied. When the nil-branch is entered on a list $\mathsf{L}_{p^1(\overline{n}^*)}(\alpha)$, then D is extended with $0 \in p^1(\overline{n}^*)$. When the cons-branch is entered, then D is extended with $m \geq 1,\ m \in p(\overline{n}^*)$, where m is a fresh size variable in D. When a constructor is applied, D is extended with position-delimiting disequations.

Given types $\tau = \mathsf{L}_{p^1(\overline{n}^*)}(\ldots \mathsf{L}_{p^s(\overline{n}^*)}(\alpha)\ldots)$ and $\tau' = \mathsf{L}_{p'^1(\overline{n}^*)}(\ldots \mathsf{L}_{p'^s(\overline{n}^*)}(\alpha)\ldots)$, let the entailment $D \vdash \tau \to \tau'$ abbreviate the collection of rules that (conditionally) rewrite $p^1(\overline{n}^*) \to p'^1(\overline{n}^*)$ etc.:

$$
\begin{array}{l}
D \qquad\qquad \vdash p^1(\overline{n}^*) \to p'^1(\overline{n}^*) \\
D,\ m_1 \in p'^1(\overline{n}^*),\ 0 \leq pos \leq m_1 - 1 \vdash p^2(\overline{n}^*)(pos) \to p'^2(\overline{n}^*)(pos) \\
\quad m_1,\ pos \ \text{ are fresh for } \ D \\
\ldots \\
D, \left\{\begin{array}{l} m_1 \in p'^1(\overline{n}^*),\ 0 \leq pos_1 \leq m_1 - 1, \ldots, \\ m_s \in p'^{s-1}(\overline{n}^*)(pos_1)\ldots(pos_{s-1}), \\ 0 \leq pos_s \leq m_s - 1 \end{array}\right\} \vdash \left\{\begin{array}{l} p^s(\overline{n}^*)(pos_1)\ldots(pos_s) \to \\ p'^s(\overline{n}^*)(pos_1)\ldots(pos_s) \end{array}\right. \\
\quad m_1,\ pos_1,\ \ldots,\ m_s,\ pos_s \ \text{ are fresh for } \ D
\end{array}
$$

The typing judgement relation is defined by the following rules:

$$\frac{}{D,\ \Gamma \vdash_\Sigma \imath\colon \mathsf{Int}}\ \textsc{IConst} \qquad \frac{}{D,\ \Gamma \vdash_\Sigma \mathsf{b}\colon \mathsf{Bool}}\ \textsc{BConst}$$

$$\frac{D \vdash \tau' \to \tau}{D,\ \Gamma,\ \mathsf{z}\colon \tau \vdash_\Sigma \mathsf{z}\colon \tau'}\ \textsc{Var} \qquad \frac{D \vdash \tau' \to \mathsf{L}_0(\tau)}{D,\ \Gamma \vdash_\Sigma \mathsf{Nil}\colon \tau'}\ \textsc{Nil}$$

$$\frac{\begin{array}{ll} D & \vdash \tau' \to \mathsf{L}_{p^1(\overline{n}^*)+1}(\tau_2') \\ D & \vdash \tau_2'(0) \to \tau_1 \\ 1 \leq m \in p^1(\overline{n}^*),\ 1 \leq pos \leq m; & D \vdash \tau_2'(pos) \to \tau_2(pos-1) \end{array}}{D,\ \Gamma,\ \mathsf{hd}\colon \tau_1,\ \mathsf{tl}\colon \mathsf{L}_{p^1(\overline{n}^*)}(\tau_2) \vdash_\Sigma \mathsf{Cons}(\mathsf{hd}, \mathsf{tl})\colon \tau'}\ \textsc{Cons}$$

where n is fresh in $D, \Gamma, \tau_1, \tau_2$. Note, that the obvious naive version of this rule, with the judgement $D,\ \Gamma,\ \mathsf{hd} : \tau,\ \mathsf{tl} : \mathsf{L}_{p^1(\overline{n}^*)}(\tau) \vdash_\Sigma \mathsf{Cons}(\mathsf{hd}, \mathsf{tl}) : \tau'$ in the conclusion and the side condition $D \vdash \tau' \to \mathsf{L}_{p^1(\overline{n}^*)+1}(\tau)$, is less general. It does not allow the length of hd, if it is a list, to differ from the length of the internal lists of tl. For instance, the naive version is not applicable to the constructor over $\mathsf{hd}\colon \mathsf{L}_5(\alpha)$ and $\mathsf{tl}\colon \mathsf{L}_n(\mathsf{L}_6(\alpha))$, whereas the presented rule accepts the type $\mathsf{L}_{n+1}(\mathsf{L}_{\lambda\, pos.g(pos)}(\alpha))$, where $g(0) = 5$ and $g(pos) = 6$ for $1 \leq pos \leq n$.

Moreover, backward application of the Cons-rule to $n \geq 1;\ \mathsf{l}\colon \mathsf{L}_n(\alpha),\ \mathsf{l}'\colon \mathsf{L}_{\mathit{tails}_1(n-1)}(\mathsf{L}_{\mathit{tails}_2(n-1)}(\alpha)) \vdash_\Sigma \mathsf{Cons}(\mathsf{l}, \mathsf{l}')\colon \mathsf{L}_{\mathit{tails}_1(n)}(\mathsf{L}_{\mathit{tails}_2(n)}(\alpha))$ allows to infer the rewriting rules for the sizes of the inner lists of the output for tails: $n \geq 1 \vdash \mathit{tails}_2(n)(0) \to n$ and $n \geq 1,\ 1 \leq pos \leq n-1 \vdash \mathit{tails}_2(n)(pos) \to \mathit{tails}_2(n-1)(pos-1)$.

The If-rule "collects" the size dependencies of both branches:

$$\frac{\begin{array}{c} D \vdash \tau \to \tau_1 \mid \tau_2 \\ \Gamma(\mathsf{x}) = \mathsf{Bool} \qquad D,\ \Gamma \vdash_\Sigma e_t\colon \tau_1 \qquad D,\ \Gamma \vdash_\Sigma e_f\colon \tau_2 \end{array}}{D,\ \Gamma \vdash_\Sigma \mathsf{if}\ \mathsf{x}\ \mathsf{then}\ e_t\ \mathsf{else}\ e_f\colon \tau}\ \textsc{If}$$

$$\frac{\mathsf{z} \notin dom(\Gamma) \qquad D,\ \Gamma \vdash_\Sigma e_1\colon \tau_z \qquad D,\ \Gamma,\ \mathsf{z}\colon \tau_z \vdash_\Sigma e_2\colon \tau}{D,\ \Gamma \vdash_\Sigma \mathsf{let}\ \mathsf{z} = e_1\ \mathsf{in}\ e_2\colon \tau}\ \textsc{Let}$$

$$\frac{\begin{array}{c} D,\ 0 \in p^1(\overline{n}^*),\ \Gamma,\ \mathsf{I}\colon \mathsf{L}_{p^1(\overline{n}^*)}(\tau) \vdash_\Sigma e_{\mathsf{Nil}} : \tau' \qquad \mathsf{hd}, \mathsf{tl} \notin dom(\Gamma) \\ D,\ m \geq 1 \in p^1(\overline{n}^*),\ \Gamma, \mathsf{hd}\colon \tau(0),\ \mathsf{I}\colon \mathsf{L}_{p^1(\overline{n}^*)}(\tau),\ \mathsf{tl}\colon \mathsf{L}_{p^1(\overline{n}^*)-1}(\tau_{+1}) \vdash_\Sigma e_{\mathsf{Cons}} : \tau' \end{array}}{D;\ \mathsf{I}\colon \mathsf{L}_{p^1(\overline{n}^*)}(\tau) \vdash_\Sigma \begin{array}{l} \mathsf{match\ I\ with} \mid \mathsf{Nil} \Rightarrow e_{\mathsf{Nil}} \\ \qquad\qquad\quad \mid \mathsf{Cons}(\mathsf{hd}, \mathsf{tl}) \Rightarrow e_{\mathsf{Cons}} \end{array} : \tau'}\ \textsc{Match}$$

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

$$\frac{\begin{array}{c} \Sigma(f) = \tau_1^f \times \ldots \times \tau_{k'}^f \times \tau_1^\circ \times \cdots \times \tau_k^\circ \to \tau_0 \\ \Sigma(\mathsf{g}_1) = \tau_1^f, \ldots, \Sigma(\mathsf{g}_{k'}) = \tau_{k'}^f \\ \mathsf{z}_1 : \tau_1^\circ, \ldots, \mathsf{z}_k : \tau_k^\circ \vdash_\Sigma e_1 : \tau_0 \qquad \Gamma \vdash_\Sigma e_2 : \tau' \end{array}}{\Gamma \vdash_\Sigma \mathsf{letfun}\ f(\mathsf{g}_1, \ldots, \mathsf{g}_{k'},\ \mathsf{z}_1, \ldots, \mathsf{z}_k) = e_1\ \mathsf{in}\ e_2 : \tau'}\ \textsc{LetFun}$$

$$\frac{\begin{array}{c} \Sigma(f) = \tau_1^f \times \ldots \times \tau_{k'}^f \times \tau_1^\circ \times \ldots \times \tau_k^\circ \to \tau_0 \\ \text{the type of } \mathsf{g}_i \text{ is an instance of the type } \tau_i^f; \\ D \vdash \tau \to \sigma(\tau_0) \qquad D \vdash C \end{array}}{D, \Gamma, \mathsf{z}_1 : \tau_1, \ldots, \mathsf{z}_k : \tau_k \vdash_\Sigma \mathsf{f}(\mathsf{g}_1, \ldots, \mathsf{g}_{k'},\ \mathsf{z}_1, \ldots, \mathsf{z}_k) : \tau}\ \textsc{FunApp}$$

where σ is an instantiation of the formal size variables with the actual size expressions, and C consists of equations between size expressions that are constructed in the following way. If $\tau_i^\circ = \mathsf{L}(\ldots \mathsf{L}_{n^s}(\tau^{\circ\prime}) \ldots)$ and $\tau_i = \mathsf{L}(\ldots \mathsf{L}_{p_i^s(\overline{n}^*)}(\tau') \ldots)$, then $\sigma(n^s) := p_i^s(\overline{n}^*)$. If $\tau_i^\circ = \tau_{i'}^\circ$, then the corresponding size expressions are equal, that is C contains $p_i^s = p_{i'}^s$. Further, if $\tau_i^\circ = \mathsf{L}(\ldots \mathsf{L}_{a^s}(\tau^{\circ\prime}) \ldots)$, then C contains $p_i^s(\overline{n}^*) = a^s$. Eventually $\sigma(\tau_0)$ for $\tau^0 = \mathsf{L}(\ldots \mathsf{L}_{f(\ldots,n^s,\ldots)}(\ldots \mathsf{L}(\alpha) \ldots) \ldots)$ is defined as $\mathsf{L}(\ldots \mathsf{L}_{f(\ldots,p^s(\overline{n}^*),\ldots)}(\ldots \mathsf{L}(\alpha) \ldots) \ldots)$.

As an example of a case when C is needed, consider a call of a function $\mathsf{scalarprod} : \mathsf{L}_m(\mathsf{Int}) \times \mathsf{L}_m(\mathsf{Int}) \to \mathsf{Int}$ on actual size arguments $\mathsf{l}_1 : \mathsf{L}_{n+1}(\mathsf{Int})$ and $\mathsf{l}_2 : \mathsf{L}_{m-1}(\mathsf{Int})$. Then C contains $n+1 = m-1$. It will hold if D contains $n = m-2$.

Example: inferring rewriting rules for concat Consider the function concat, which given a list of lists appends all the inner lists:

$$\begin{array}{l} \mathsf{concat(l)} = \mathsf{match\ l\ with} \mid \mathsf{Nil} \Rightarrow \mathsf{Nil} \\ \qquad\qquad\qquad\qquad\quad \mid \mathsf{Cons(hd,\ tl)} \Rightarrow \mathsf{append(hd,\ concat(tl))} \end{array}$$

The rewriting rules defining the type for $\mathsf{concat} : \mathsf{L}_n(\mathsf{L}_M(\alpha)) \to \mathsf{L}_{\mathit{concat}(n,M)}(\alpha)$ are

$$\begin{array}{c} \vdash \mathit{concat}(0, M) \to 0 \\ n \geq 1 \vdash \mathit{concat}(n, M) \to M(0) + \mathit{concat}(n-1, \lambda\, pos.\, M(pos+1)) \end{array}$$

Now we show how the typing rules are used to infer this rewriting system. We apply the rules as in a subgoal-directed backward-style proof.

1. The LETFUN rule defines the main goal: $\mathsf{I}\colon \mathsf{L}_n(\mathsf{L}_M(\alpha)) \vdash_\Sigma e_{\mathsf{concat}} : \mathsf{L}_{\mathit{concat}(n,M)}(\alpha)$, where e_{concat} denotes the body of concat.
2. Apply the MATCH-rule. In the nil-branch we obtain the subgoal $n = 0;\ \mathsf{I}\colon \mathsf{L}_n(\mathsf{L}_M(\alpha)) \vdash_\Sigma \mathsf{Nil}\colon \mathsf{L}_{\mathit{concat}(n,M)}(\alpha)$.

3. Continue with the nil-branch. Apply the NIL rule and obtain $n = 0 \vdash \mathsf{L}_{\mathit{concat}(n,M)}(\alpha) \to \mathsf{L}_0(\tau^?)$.
4. Instantiate $\tau^? = \alpha$. Unfold the definition of type rewriting: $n = 0 \vdash \mathit{concat}(n, M) \to 0$.
5. Now, consider the cons-branch. The subgoal there is $n \geq 1;\ \mathsf{hd}\colon \mathsf{L}_M(\alpha)(0),\ \mathsf{tl}\colon \mathsf{L}_{n-1}(\mathsf{L}_M(\alpha)_{+\mathbf{1}}) \vdash_\Sigma \mathsf{append}(\mathsf{hd}, \mathsf{concat}(\mathsf{tl}))\colon \mathsf{L}_{\mathit{concat}(n,M)}(\alpha)$. (Note that in contexts we omit variables on which the expression does not depend.)
6. Unfold the definition of application of a type to a first-level expression and the definition for $(-)_{+\mathbf{1}}$: $n \geq 1;\ \mathsf{hd}\colon \mathsf{L}_{M(0)}(\alpha),\ \mathsf{tl}\colon \mathsf{L}_{n-1}(\mathsf{L}_{M_{+\mathbf{1}}}(\alpha)) \vdash_\Sigma \mathsf{append}(\mathsf{hd}, \mathsf{concat}(\mathsf{tl}))\colon \mathsf{L}_{\mathit{concat}(n,M)}(\alpha)$.
7. The expression in the judgement above is a sugared let-construct. So, we apply the LET-rule. In the binding we get the goal: $n \geq 1;\ \mathsf{tl}\colon \mathsf{L}_{n-1}(\mathsf{L}_{M_{+\mathbf{1}}}(\alpha)) \vdash_\Sigma \mathsf{concat}(\mathsf{tl})\colon \tau^?$.
8. Using FUNAPP-rule we instantiate the type $\tau^? := \mathsf{L}_{\mathit{concat}(n-1,M_{+\mathbf{1}})}(\alpha)$.
9. Therefore, the subgoal for the let-body is $n \geq 1;\ \mathsf{hd}\colon \mathsf{L}_{M(0)}(\alpha),\ \mathsf{l}'\colon \mathsf{L}_{\mathit{concat}(n-1,M_{+\mathbf{1}})}(\alpha) \vdash_\Sigma \mathsf{append}(\mathsf{hd}, \mathsf{l}')\colon \mathsf{L}_{\mathit{concat}(n,M)}(\alpha)$.
10. Apply the FUNNAPP-rule. In this rule use the type $\mathsf{append} : \mathsf{L}_{n_1}(\alpha') \times \mathsf{L}_{n_2}(\alpha') \to \mathsf{L}_{n_1+n_2}(\alpha')$ and $\sigma(n_1) := M(0)$, $\sigma(n_2) := \mathit{concat}(n-1, M_{+\mathbf{1}})$. We obtain the predicate $n \geq 1 \vdash \mathsf{L}_{\mathit{concat}(n,M)}(\alpha) \to \mathsf{L}_{M(0)+\mathit{concat}(n-1,M_{+\mathbf{1}})}(\alpha)$.
11. Unfold the definition of type rewriting and the definition of the operation$(-)_{+\mathbf{1}}$: $n \geq 1 \vdash \mathit{concat}(n, M) \to M(0) + \mathit{concat}(n-1,\ \lambda\ \mathit{pos}.\, M(\mathit{pos}+1))$.

3.2 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 $\Gamma^\bullet$:

$$\begin{aligned} \mathit{Valid}_{\mathsf{val}}(v, \tau^\bullet, h) &= \exists_w.\ v \models^h_{\tau^\bullet} w \\ \mathit{Valid}_{\mathsf{store}}(\mathit{vars}, \Gamma^\bullet, s, h) &= \forall_{x \in \mathit{vars}}.\ \mathit{Valid}_{\mathsf{val}}(s(x), \Gamma^\bullet(x), h) \end{aligned}$$

Let a valuation ϵ^s map size variables to constants of the layer s, and let an instantiation η map type variables to ground types:

$$\begin{aligned} &\mathit{Valuation} && \epsilon^s : \mathit{SizeVariables}^s \to (\mathcal{R} \to \ldots \to \mathcal{R} \to 2^{\mathcal{R}}) \\ &\mathit{Instantiation} && \eta^s : \mathit{TypeVariables}^s \to \tau^{\bullet s} \end{aligned}$$

Let ϵ and η be the direct sums of some $\epsilon^1, \ldots, \epsilon^k$ and $\eta^1, \ldots, \eta^k$ respectively. We will usually write the application of η and ϵ as subscripts. For example, $\eta(\epsilon(\tau))$ becomes $\tau_{\eta\epsilon}$ and $\epsilon(D)$ becomes D_ϵ. Note that D contains no type variables and hence $D_\eta = D$. Valuations and instantiations distribute over size functions in the following way: $(\mathsf{L}_{p(\overline{n}^*)}(\tau))_{\eta\epsilon} = \mathsf{L}_{p(\overline{n}^*_\epsilon)}(\tau_{\eta\epsilon})$.

Informally, the soundness theorem states that, assuming that the zero-order context variables are *valid*, i.e., that they indeed point to lists of the sizes mentioned in the input types, then the result in the heap will be *valid*, i.e., it will have 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*

- $dom(s) = dom(\Gamma)$, $\epsilon\colon SV(\Gamma) \cup SV(D) \to \mathcal{R}$ *and* $\eta\colon TV(\Gamma) \to \tau^\bullet$,
- D_ϵ *holds,*
- $s;\, h;\, \mathcal{C} \vdash e \rightsquigarrow v;\, h'$ *and* $D,\, \Gamma \vdash_\Sigma e\colon \tau$,
- $Valid_{\mathsf{store}}(dom(s), \Gamma_{\eta\epsilon}, s, h)$,

then v is valid according to its return type τ in h', i.e., $Valid_{\mathsf{val}}(v, \tau_{\eta\epsilon}, h')$.

Proof. The proof is done by induction on the size of the derivation tree for the operational-semantic judgement. It can be found on the technical report [11].

4 Inferring Families of Polynomials

Consider a multivalued size function f over variables $\overline{n}^*$ given by (recursive) rewriting rules. Our aim is to obtain a closed form (i.e. a recursion-free from) of f. It is clear that this is not always possible. In this section, we show how to obtain an approximation of the closed form of f by constructing a family (i.e. a set) that includes the range of f.

Let $\overline{n} \subseteq \overline{n}^*$ be the list of all first-layer variables of $\overline{n}^*$. For any variable $n_l^s \in \overline{n}^*$ of a layer $s \geq 2$, let its range be given in the form $T(n_l^s) = \{p_l(\overline{n}, \overline{n}', \bar{i})\}_{Q_l(\overline{n},\overline{n}',\bar{i})}$, which is a short cut for $\{p_l(\overline{n}_0, \overline{n}'_0, \bar{i}) \mid \exists\, \bar{i}.\ Q_l(\overline{n}_0, \overline{n}'_0, \bar{i})\}$. Here Q is a first-order arithmetic predicate and $\overline{n}'$ are fresh w.r.t. $\overline{n}$. We introduce fresh size variables like n' and assumptions as the one above if we know nothing about n^s, where $s \geq 2$. In general such default assumptions are of the form $range(n^s) \subseteq \{i\}_{n'_1 \leq i \leq n_2}$.

We will show how, given a conditional rewriting rule with the l.h.s $D_1(\overline{n}^*, \overline{m}) \wedge D_2(\overline{m}, \overline{pos})$, to obtain $\{p(\overline{n}, \overline{n}', \bar{i})\}_{Q(\overline{n},\overline{n}',\bar{i})}$ such that if for all higher-layer variables $range(n^s) \subseteq T(n^s)$ and $D_1(\overline{n}^*, \overline{m})$ holds then $f(\overline{n}^*) \subseteq \{p(\overline{n}, \overline{n}', \bar{i})\}_{Q(\overline{n},\overline{n}',\bar{i})}$.

4.1 Generating a candidate family to cover the range of a size function

Our main assumption is that for any fixed $\overline{n}_0, \overline{n}'_0$ the sets $T(n_l^s)$ are finite. For instance, for $n' = 3$ the *range* of M is included into the set $\big(T(M) = \{i\}_{0 \leq i \leq n'}\big)_{n':=3} = \{0, 1, 2, 3\}$. Moreover, for fixed n and n' the function M is reduced to the finite multivalued map ϕ such that $\phi(pos)$ is the set of all possible lengths of the "inner" lists. E.g. with $n = 2$ and $n' = 3$ we have ϕ instantiated as $\phi(0) = \phi(1) = \{0, 1, 2, 3\}$.

With fixed $\overline{n}$ and $\overline{n}'$ the function f is translated to an auxiliary function $\llcorner f \lrcorner$ over finite sets and maps. For instance, $\mathit{concat}(n, M)$ becomes $\llcorner \mathit{concat} \lrcorner(n, \phi) \to \phi(0) + \llcorner \mathit{concat} \lrcorner(n-1, \phi_{+1})$. Now we show how to translate f to the function $\llcorner f \lrcorner$, which will be used later to obtain a family of polynomials that possibly covers the range of f.

Rewriting rules for an auxiliary function over finite sets We are going to introduce auxiliary functions of type $(\mathit{FiniteSet}, \mathit{FiniteMMap})^* \to \mathit{FiniteSet}$, where $(\mathit{FiniteSet}, \mathit{FiniteMMap})^*$ is a finite Cartesian product of finite sets and finite multivalued maps. Binary arithmetic operations are lifted to sets: if $\circledast$ is one of the arithmetic operations $+, -, *$, then $\mu_1 \circledast_{\{\}} \mu_2 := \{x \circledast y\}_{x \in \mu_1 \wedge y \in \mu_2}$.

A *finite multivalued map* is a mapping from positions to finite sets:

$$\mathit{FiniteMMap} : \mathit{Positions}_{d_1} \to \ldots \to \mathit{Positions}_{d_k} \to \mathit{FiniteSet}$$

where $\mathit{Positions}_{d_l} = \{0, \ldots, d_l - 1\}$. An example of finite multivalued maps is $\langle\{1,2,3\},\{1\}\rangle$, which sends 0 to $\{1,2,3\}$ and 1 to $\{1\}$. We denote a multivalued map via ϕ and μ denotes either a finite map or set. There is an empty map denoted via $\langle\,\rangle$. The only operation over multivalued maps, which is relevant to our task, is left shift $[-]_{+k}$ sending $\langle\mu_0, \ldots, \mu_{d-1}\rangle$ to $\langle\mu_k, \ldots, \mu_{d-1}\rangle$.

The straightforward translation $\llcorner - \lrcorner$ that maps size expressions onto expressions over finite sets and finite multivalued maps is inductively defined on the structure of size expressions. Constants, first-layer and position variables represents themselves. A variable n_l^s of the layer $s \geq 2$ is translated into a finite-set variable ϕ_l. The construction $\lambda\, pos.\,(p)(pos{+}p')$ is translated into $\llcorner p \lrcorner_{[+\llcorner p' \lrcorner]}$. The function application $g(p_1, \ldots, p_k)$ is translated into $\llcorner g \lrcorner(\llcorner p_1 \lrcorner, \ldots, \llcorner p_k \lrcorner)$. The detailed definition of $\llcorner - \lrcorner$ is given in [11].

Given a rewriting rule $f(n_1^{s_1}, \ldots n_k^{s_k})(pos_1 \ldots pos_{s-1}) \to p$ for a numerical multivalued function f, we construct the corresponding rewriting rule for $\llcorner f \lrcorner$ as $\llcorner f \lrcorner(\llcorner n_1^{s_1} \lrcorner, \ldots \llcorner n_k^{s_k} \lrcorner)(pos_1 \ldots pos_{s-1}) \to \llcorner p \lrcorner$.

Generating a family Consider a brunch of f, defined by the rule $D_1(\overline{n}^*, \overline{m}) \wedge D_2(\overline{m}, \overline{pos}) \vdash f(\overline{n}^*)(\overline{pos}) \to p$. We will construct an estimate for the range of f in the form $\{f_l(\overline{n}, \overline{n}')(\overline{pos}) + i\}_{0 \leq i \leq f_u(\overline{n},\overline{n}')(\overline{pos}) - f_l(\overline{n},\overline{n}')(\overline{pos})}$, where $f_l(\overline{n}, \overline{n}')(\overline{pos}) \leq f(\overline{n}^*)(\overline{pos}) \leq f_u(\overline{n}, \overline{n}')(\overline{pos})$. We show now how to compute candidates for bounds f_l and f_u if they are polynomial. First, we need to assume their degree(s) d.

1. Choose $\binom{V+d}{d}$ points $(\overline{n}_0, \overline{n}'_0, \overline{pos}_0)$, that uniquely define a polynomial of degree d with $V = |\overline{n}| + |\overline{n}'| + |\overline{pos}_0|$ variables. We have discussed how to choose such points in [17]. For instance, assuming $d = 1$ for $\mathit{tails}_{2\ l}(n)(pos)$ and $\mathit{tails}_{2\ u}(n)(pos)$ we take the set of test points (n_0, pos_0) as $\{(2,0),(2,1),(3,0)\}$.
2. For each $(\overline{n}_0, \overline{n}'_0, \overline{pos}_0)$ from the set of test points do:
   (a) for any $n_l^s \in \overline{n}^*$ assign $\phi_l := \lambda\ pos'.\{p_l(\overline{n}_0, \overline{n}'_0, \overline{i})\}_{Q_l(\overline{n}_0,\overline{n}'_0,\overline{i})}$, which is a constant multivalued map; e.g. $\phi := \langle\{0,1\}\rangle$ for M in $\mathit{concat}(n, M)$ with $n = 1,\ n' = 1$.
   (b) compute $\llcorner f \lrcorner(\overline{n}, \overline{\phi})(pos)$ using the rewriting rules; e.g. $\llcorner \mathit{tails} \lrcorner_2(2)(1) \to \llcorner \mathit{tails} \lrcorner_2(2-1)(1-1) = \llcorner \mathit{tails} \lrcorner_2(1)(0) \to 1$.
   (c) assign $f_{\min}(\overline{n}_0, \overline{n}'_0, \overline{pos}_0) := \min\left(\llcorner f \lrcorner(\overline{n}_0, \overline{n}'_0, \overline{pos}_0)\right)$ and $f_{\max}(\overline{n}_0, \overline{n}'_0, \overline{pos}_0) := \max\left(f'(\overline{n}_0, \overline{n}'_0, \overline{pos}_0)\right)$; e.g. $\mathit{tails}_{2,\min}(2,1) = \mathit{tails}_{2,\max}(2,1) = 1$.
   (d) add to the lists of equations w.r.t. the coefficients of f_l and f_u the equations with $f_{\min}(\overline{n}_0, \overline{n}'_0, \overline{pos}_0)$ and $f_{\max}(\overline{n}_0, \overline{n}'_0, \overline{pos}_0)$ on the r.h.s., respectively; e.g., $\mathit{tails}_{2\ u}(2,0)$ defines $2a_{u,10} + a_{u,01} + a_{u,00} = \mathit{tails}_{2\ \max}(2,1) = 1$.

3. Solve the linear systems for the coefficients f_l and f_u. For instance, for *tails*$_2$ we obtain *tails*$_{2\,l}(n, pos) =$ *tails*$_{2\,u}(n, pos) = n - pos$. Thus, we have obtained polynomial lower and upper bounds for the size function f.
4. On the previous step we have obtained the bounds for the size function f, from which construct a family of polynomials in the form given in the begin of this subsection.
   If the size function is of the first layer, we output the family as it is. For instance, for *concat* we return $\{i\}_{0\leq i\leq nn'}$.
   If f is of the layer $s \geq 2$, then the bounds depend on positions $\overline{pos}$. In this case, replace $\overline{pos}$ with new indices $\overline{j}$ to obtain $\{f_l(\overline{n}, \overline{n}', \overline{j}) + i\}_{Q'(\overline{n},\overline{n}',\overline{m},\overline{j})}$ where Q' abbreviates $0 \leq i \leq f_u(\overline{n}, \overline{n}', \overline{j}) - f_l(\overline{n}, \overline{n}', \overline{j}) \wedge D_2(\overline{m}, \overline{j})$. Note that $D_2(\overline{m}, \overline{j})$ consists of disequations of the form $0 \leq j \leq m-1$ or $1 \leq j \leq m$. Replace m that belongs to the set $p(\overline{n}^*)(pos_1)\dots(pos_{s-1})$ with the already derived upper bound for this set. For instance, for $\llcorner$*tails*$\lrcorner_2(n)$ we obtain $\{n-j\}_{1\leq j\leq n-1}$ on $n \geq 1$. The family is completed to $\{n-j\}_{0\leq j\leq n-1}$ by $\llcorner$*tails*$\lrcorner_2(n)(0) = n$.
5. The return family needs to be checked. The checking is done by reducing rewriting rules to set inclusions and, eventually, to first-order predicates. The reduction has been sketched in the introduction. For more detail, see [11].
   If a type-checker accepts the family then the job is done. Otherwise we need to analyse the failure. Rejection may happen if either the program's size bounds are not polynomial, or we have chosen wrong parameter d and/or the set of test points. We may repeat the procedure for a larger d and/or other test points (see [17] for a discussion on how to choose test points for such procedures).

4.2 Checking if a given family covers the range of a function

To give a sufficient condition for a given family of polynomials to cover the range of the function f we first need to fill-in the specification table T for functions g that occur in the rewriting rules for f and their variables (formal parameters).

Informally, the problem of checking if a family of polynomials $T(f(\overline{n}^*))$ "covers" a given multivalued function f amounts to checking if for any computation path for $f(\overline{n}^*)(\overline{pos})$ the result will be in $(T(f(\overline{n}^*))$. In other words, for any rewriting rule $D \vdash f(\overline{n}^*)(\overline{pos}) \to p$ the following inclusion holds: $D \vdash T(f(\overline{n}^*)) \supseteq$ *range*(p), given that the range each higher-layer size variable $n^s \in \overline{n}^*$ is $T(n^s)$.

Let $\overline{n}^*_g$ be the list of the formal size parameters of g and $\overline{n}_g \subseteq \overline{n}^*_g$ are first-layer variables. The table is constructed as follows.

- if $n^s_g \in \overline{n}^*_g$, where $s \geq 2$, then $T(n^s_g)$ is given in the form $\{p(\overline{n}_g, \overline{n}'_g, \overline{i})\}_{Q(\overline{n}_g,\overline{n}'_g,\overline{i})}$, where $\overline{n}'_g$ are fresh first-layer size variables, and a polynomial $p(\overline{n}_g, \overline{n}'_g, \overline{i})$ and a predicate $Q(\overline{n}_g, \overline{n}'_g, \overline{i})$ are
  - either given by a user,
  - or are set by default to $\{i\}_{0\leq i\leq n'_g}$ or $\{i\}_{n_{g'_1}\leq i\leq n_{g'_2}}$;

– $T\big(g(\overline{n}^*_g)\big)$ has the form $\{p(\overline{n}_g, \overline{n}'_g, \overline{i})\}_{Q(\overline{n}_g, \overline{n}'_g, \overline{i})}$. Note, that we treat higher-layer constants as functions, that is their specifications must be present in the table as well, in the form $T(a) = \{p(\overline{i})\}_{Q(\overline{i})}$. In principle, the range of a may be generated automatically and then there is no need to add it to the table T. To avoid technical overhead we do not consider this otimisation in the presented work and leave it for the future.

For instance, the table T, which is used to check the family $\{i\}_{0 \le i \le nn'}$ for *concat*, contains $T(+\!\!+, n_1, n_2) = n_1 + n_2$, $T(M) = \{i\}_{0 \le i \le n'}$, $T(\textit{concat}, n, n') = \{i\}_{0 \le i \le nn'}$.

Let $\overline{n}^*$ be the set of the free size variables of f. Let $\mathit{rhs}(f)$ denote the conditions from the rewriting rules defining f. The set $\mathit{rhs}(f)$ consists of memberships like $m \in p(\overline{n}^*)$, position restrictions like $0 \le \mathit{pos} \le m-1$ (from the definition of type rewriting) or $1 \le \mathit{pos} \le m$ (a side condition of the cons-rule) and disequations $m \ge 1$ (a side condition of the constructor-rule and of cons-branch in the match-rule).

Definition 1. *The specification $T(f(\overline{n}^*))$ is valid if and only if given that the specifications of all functions $g \neq f$ used in its definition are valid, if $\overline{n}^*, \overline{\mathit{pos}}$ are s.t. $f(\overline{n}^*)(\overline{\mathit{pos}})$ terminates, then $\bigwedge_{n^s \in \overline{n}^*,\ s \ge 2} n^s(\overline{\mathit{pos}}) \subseteq T(n^s)$ implies $f(\overline{n}^*)(\overline{\mathit{pos}}) \subseteq T\big(f(\overline{n}^*)\big)$.*

Let $p_{\overline{n}^*, \overline{\mathit{pos}}}$ denote a size expression with free size variables $\overline{n}^*$ and free position variables $\overline{\mathit{pos}}$. The result of its application to some values $\overline{x}^*, \overline{x}_{\overline{\mathit{pos}}}$ is denoted via $p_{\overline{n}^*, \overline{\mathit{pos}}}(\overline{x}^*, \overline{x}_{\overline{\mathit{pos}}})$.

Next, we define a *range map* $(\!|-|\!) : \mathit{SizeExpression} \to \mathit{IndexedPolynomial} \times \mathit{1stOrderPredicate}$, where the first-order predicate in the image delimits the indices of the polynomial. Let $(\!|p|\!)_1$ and $(\!|p|\!)_2$ stay for the first projection (the polynomial) and the second projection (the predicate that bounds the indices) of $(\!|p|\!)$, resp. A correct range map $(\!|p|\!)$ is defined by induction over the structure of its argument p, which is an expression with free size variables $\overline{n}^*$:

– for a first-layer constant a the range map is defined obviously as $(\!|a|\!) := \{a\}$;
– $(\!|a^s|\!) := T(a^s)$, where $s \ge 2$;
– for a first-layer variable n from the set of parameters $\overline{n}^*$ the range map is defined as $(\!|n|\!) := \{n\}$,
– for a higher-layer variable n^s from the set of parameters $\overline{n}^*$, where $s \ge 2$, the range map is defined by the spec. table, $(\!|n^s|\!) := T(n^s)$;
– if $\circledast$ is one of the arithmetic operations $+, -, *$, then $(\!|p_1 \circledast p_2|\!) := (\!|p_1|\!) \circledast_{\{\}} (\!|p_2|\!)$;
– $(\!|p(0)|\!) := (\!|p|\!)$;
– $(\!|p(\mathit{pos})|\!) := (\!|p|\!)$;
– $(\!|p(\mathit{pos}-1)|\!) := (\!|p|\!)$;
– $(\!|p_{+\mathbf{1}}|\!) := (\!|p|\!)$;
– in a function call $g(p^1_1, \ldots, p^1_k, p'_1, \ldots p'_{k'})$ we match the actual parameters with the fromal parameters $\overline{n}_g, \overline{n}'_g$ of the specification

$$T\big(g(\overline{n}^*_g)\big) = \{p(n_{g_1}, \ldots, n_{g_k}, \overline{n}'_g, \overline{j})\}_{Q(\overline{n}_g, \overline{n}'_g, \overline{j})}$$

First, note that since the function call terminates, then there must be a rewriting rule $D_g \vdash g(\overline{n}^*_g)(\overline{pos}) \to p_g$ applicable for this call. From what follows that if we replace in D_g the formal parameters $\overline{n}^*_g$ with the corresponding actual size expressions, then the result of the replacement D'_g should be valid on the actual size expressions.

Now continue as follows:

1. we first (inductively) compute the range sets $(\!|p^1_l|\!)$ of the first-layer actual parameters p^1_l, where $1 \leq l \leq k$;
2. after that we (inductively) compute the range sets $(\!|p'_l|\!)$ of the higher-layer actual parameters p'_l, where $1 \leq l \leq k'$;
3. after that the most difficult part of the matching "formal vs. actual parameters" is to be done: finding a substitution σ : *FreshSizeVar* $\to$ *IndexedPolynomial*$\times$ *1stOrderPredicate*, such that for all formal n^s_{gl}, with $T(n^s_{g\ l}) = \{p''_l(\overline{n}_g, \overline{n}'_g, \overline{j}')\}_{Q''_l(\overline{n}_g, \overline{n}'_g, \overline{j}')}$, the following inclusion must be provable from D'_g:

$$(\!|p'_l|\!) \subseteq \{p''_l((\!|p_1|\!)_1, \ldots, (\!|p_k|\!)_k,\ \sigma_1(\overline{n}'_g), \overline{j}')\}_{\substack{Q''_l((\!|p_1|\!)_1, \ldots, (\!|p_k|\!)_k,\ \sigma_1(\overline{n}'_g), \overline{j}')\wedge \\ \bigwedge_{l=1}^{k} (\!|p^1_l|\!)_2 \wedge \\ \bigwedge_{l=1}^{k'} \sigma_2(n'_{g\ l})}}$$

Finding a substitution σ is the most difficult part of the procedure. It is a source of undecidability of inference in general, since it amounts to the instantiation of existential quantifiers in Peano arithmetic. However, in some cases (e.g. for linear predicates) finding a substitution may be done automatically.

4. eventually

$$\begin{array}{l}(\!|g(p^1_1, \ldots, p^1_k, p'_1, \ldots p'_{k'})|\!) := \\ \{p((\!|p^1_1|\!)_1, \ldots, (\!|p^1_k|\!)_1,\ \sigma_1(\overline{n}'_g),\ \overline{j})\}_{\substack{Q((\!|p^1_1|\!)_1, \ldots, (\!|p^1_k|\!)_1,\ \sigma_1(\overline{n}'_g),\ \overline{j})\wedge \\ \bigwedge_{l=1}^{k} (\!|p^1_l|\!)_2 \wedge \bigwedge_{l=1}^{k'} \sigma_2(n'_{g\ l})}}\end{array}$$

Sometimes, for the sake of convenience, the polynomial p and the delimiting predicate Q form the specification $T(\mathit{program}(\overline{n}^*)) = \{p(\overline{n}, \overline{n}', \overline{i})\}_{Q(\overline{n}, \overline{n}', ibar)}$ are denoted via $(\!|\mathit{program}|\!)_1$ and $(\!|\mathit{program}|\!)_2$ respectively.

As an instance, consider the r.h.s. of the rewriting rule $n \geq 1 \vdash \mathit{concat}(n, M) \to M(0) + \mathit{concat}(n-1, M_{+\mathbf{1}})$.

$$\begin{array}{l}(\!|M(0) + \mathit{concat}(n-1, M_{+\mathbf{1}})|\!) = \\ (\!|M(0)|\!) +_{\{\}} (\!|\mathit{concat}(n-1, M_{+\mathbf{1}})|\!) = \\ (\!|M|\!) +_{\{\}} \{(\!|\mathit{concat}|\!)_1((\!|n-1|\!)_1, \sigma_1(n'), i)\}_{(\!|\mathit{concat}|\!)_2((\!|n-1|\!)_1, \sigma_1(n'), i) \wedge \sigma_1(n')} = \\ \{i\}_{0 \leq i \leq n'} +_{\{\}} \{i\}_{0 \leq (n-1)n'}\end{array}$$

where $\sigma(n') = \{n'\}$. Note that the scope of an index limited to the set it is "attached" to.

Another example shows that substitutions for fresh size variables $\overline{n}'_g$ are not always identities as in the example above. Consider the composition $\mathsf{concat}(\mathsf{tails}(\mathsf{l}))$ with l be of the type $\mathsf{L}_n(\alpha)$. We want to check the rough but still sound estimate $\mathit{concat} \circ \mathit{tails}(n) \subseteq \{i\}_{0\le i\le n^2}$. We have $\mathit{concat} \circ \mathit{tails}(n) \to \mathit{concat}(n, \mathit{tails}_2(n))$. We already know that $T(\mathit{concat}(n, M)) = \{i\}_{0\le i\le nn'}$ for $T(M) = \{i\}_{0\le i\le n'}$. Now we need to match $T(M)$ with the annotation of the actual parameter $\mathsf{L}_{\mathit{tails}_2(n)}(\alpha)$. We know that $T(\mathit{tails}_2(n)) = \{i\}_{0\le i\le n}$, so we assume $\sigma(n') = \{n\}$. Indeed, $(\!|\mathit{tails}_2(n)|\!) = \{i\}_{0\le i\le n} \subseteq \sigma(T(M)) = \{i\}_{0\le i\le \sigma(n')}$, thus σ is a valid substitution.

Lemma 2 (Consistency of range map: basic). *Given an expression $p_{\overline{n}^*,\overline{pos}}$, if the specifications $T(g(\overline{n}^*))$ of all the functions g that occur in it are valid, then for all $\overline{n}^*, \overline{n}', \overline{pos}$, such that $\bigwedge_{n^s\in\overline{n}^*,\ s\ge 2} n^s(\overline{pos}) \subseteq T(n^s)$ and $p_{\overline{n},\overline{pos}}(\overline{n}^*, \overline{pos})$ terminates, the inclusion $p_{\overline{n}^*,\overline{pos}}(\overline{n}^*, \overline{pos}) \subseteq (\!|p_{\overline{n}^*,\overline{pos}}|\!)$ holds.*

Proof. Fix $\overline{n}^*, \overline{pos}$, such that $p_{\overline{n}^*,\overline{pos}}$ terminates on them. The proof is done by induction on the structure of $p_{\overline{n}^*,\overline{pos}}$. It is given in [11].

Lemma 3 (Consistency of range map). *Given an expression $p_{\overline{n}^*,\overline{pos}}$, let the specifications $T(g(\overline{n}^*_g))$ of all the functions g that occur in it be valid, except may be the specification $T(f(\overline{n}^*_f))$ for f, for which we do not know if it is valid or not. Let for each rewriting rule $D \vdash f(\overline{n}^*)(\overline{pos}) \to p_f$ the inclusion $(\!|D|\!) \vdash T(f(\overline{n}^*)) \supseteq (\!|p_f|\!)$ holds. Then for all $\overline{n}^*, \overline{n}', \overline{pos}$, such that $\bigwedge_{n^s\in\overline{n}^*,\ s\ge 2} n^s(\overline{pos}) \subseteq T(n^s)$ and $p_{\overline{n},\overline{pos}}(\overline{n}^*, \overline{pos})$ terminates, the inclusion $p_{\overline{n}^*,\overline{pos}}(\overline{n}^*, \overline{pos}) \subseteq (\!|p_{\overline{n}^*,\overline{pos}}|\!)$ holds.*

Proof. It is done exactly as the proof of Lemma 2, except that we will consider calls of the function f as a special case. The full proof is given in [11].

Theorem 2 (Checking). *If all called in the definition of f functions $g \ne f$ have valid specifications $T\big(g(\overline{n}^*_g)\big)$, and for each rule $D \vdash f^s(\overline{n}*)(pos_1)\dots(pos_{s-1}) \to p$ the inclusion $D \vdash T(f(\overline{n}*)) \supseteq (\!|p|\!)_D$ holds then the specification $T\big(f(\overline{n}^*)\big)$ is also valid.*

Proof. Fix some $\overline{n}^*, \overline{pos}$ such that the function f is defined on them. It means that there must be a rewriting rule applicable to these parameters, say, $D \vdash f(\overline{n}^*)(\overline{pos}) \to p$ and some $\overline{m}$, such that $D(\overline{n}, \overline{m}, \overline{pos})$ holds. Since this rule is used as the first rule to compute $f(\overline{n}*)(\overline{pos})$ we obtain that $f(\overline{n}*)(\overline{pos}) = p$. Form Lemma 3 we obtain $f(\overline{n}*)(\overline{pos}) \subseteq (\!|p|\!)_D$. From the condition of the lemma we have $f(\overline{n}*)(\overline{pos}) \subseteq T(f(\overline{n}*))$.

5 Related Work

This research extends our work [14, 17, 15, 13] about shapely function definitions that have a single-valued, exact input-output polynomial size functions. Our non-monotonic framework resembles [1] 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 *Mathematica*. However, they do not consider non-monotonic size functions.

Our approach is related to size analysis with polynomial quasi-interpretations [4, 2]. There, a program is interpreted as a *monotonic* polynomial extended with the max operation.

Hofmann and Jost have presented a heap space analysis [9] to infer linear space bound of functional programs with explicit memory deallocation. It uses type annotations and an amortisation analysis that assign a *potential*, i.e. hypothetical free space, to data structures. The type system ensures that the potential to the input is an upper bound on the total memory required to satisfy all allocations. They have extended their analysis to object-oriented programs [10], although without an inference procedure. Brian Campbell extended this approach to infer bounds on *stack* space usage in terms of the total size of the input [5], and recently as max-plus expressions on the depth of data structures [6]. Again, the main difference with our work is that we not require linear size functions. Recently, this analysis has been extended to include multivariate non-linear resource polynomials [8]. The difference with our work is that we also allow general polynomials including non-monotonic polynomials like $x^2 - y^2$ where [8] allows only nonnegative linear combinations of base polynomials.

In his thesis, Pedro Vasconcelos [18] 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 [7, 3]. This line of research is motivated both by the perspective of automated complexity analysis and providing natural characterisations of complexity classes like PTIME or PSPACE. Resource analysis may also be performed within a *Proof Carrying Code* framework.

6 Conclusions and Future Work

We have presented a system that combines lower/upper bounds and higher-order size annotations to express, type check and infer reasonable approximations for polynomial size dependencies for strict functional programs using general lists.

Future work will include research on adding algebraic data types, making a prototype possibly using dependent types, applying the prototype for larger programs and transferring the results to an imperative object-oriented language.

References

1. E. Albert, P. Arenas, S. Genaim, G. Puebla, and D. Zanardini. Cost Analysis of Java Bytecode. In 16^{th} *European Symposium on Programming, ESOP'07*, volume 4421 of *LNCS*, pages 157–172. Springer, 2007.

2. R. M. Amadio. Synthesis of max-plus quasi-interpretations. *Fundamenta Informaticae*, 65(1-2):29–60, 2004.
3. 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.
4. G. Bonfante, J. Y. Marion, and J. Y. Moyen. Quasi-interpretations a way to control resources. *Theor. Comput. Sci.*, 412(25):2776–2796, June 2011.
5. B. Campbell. *Space Cost Analysis Using Sized Types*. PhD thesis, School of Informatics, University of Edinburgh, 2008.
6. B. Campbell. Amortised memory analysis using the depth of data structures. In G. Castagna, editor, *ESOP 2009*, volume 5502 of *LNCS*, pages 190–204. Springer-Verlag, 2009.
7. M. Gaboardi, J.-Y. Marion, and S. Ronchi Della Rocca. A logical account of PSPACE. In 35^{th} *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages POPL 2008, San Francisco, January 10-12, 2008, Proceedings*, pages 121–131, 2008.
8. J. Hoffmann, K. Aehlig, and M. Hofmann. Multivariate amortized resource analysis. *ACM Trans. Program. Lang. Syst.*, 34(3):14:1–14:62, Nov. 2012.
9. M. Hofmann and S. Jost. Static prediction of heap space usage for first-order functional programs. *SIGPLAN Not.*, 38(1):185–197, 2003.
10. M. Hofmann and S. Jost. Type-based amortised heap-space analysis. In P. Sestoft, editor, *ESOP 2006*, volume 3924 of *LNCS*, pages 22–37, 2006.
11. O. Shkaravska, M. van Eekelen, and A. Tamalet. Collected size semantics for functional programs over polymorphic nested lists. Technical Report ICIS-R09003, Radboud University Nijmegen, July 2009.
12. O. Shkaravska, M. van Eekelen, and A. Tamalet. Collected Size Semantics for Functional Programs over Lists. In S.-B. Scholz, editor, *Selected Paers of the* 20^{th} *Symposium on Implementation and Application of Functional Languages, IFL 2008. International Workshop, Hertfordshire, UK*, volume 5836 of *LNCS*, pages 118–137. Springer-Verlag, 2011.
13. O. Shkaravska, M. C. J. D. van Eekelen, and R. van Kesteren. Polynomial size analysis of first-order shapely functions. *Logical Methods in Computer Science*, 5(2), 2009.
14. 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.
15. A. Tamalet, O. Shkaravska, and M. van Eekelen. Size Analysis of Algebraic Data Types. In P. Achten, P. Koopman, and M. Morazán, editors, *Trends in Functional Programming Volume 9 (TFP'08)*. Intellect Publishers, 2009.
16. M. van Eekelen, O. Shkaravska, R. van Kesteren, B. Jacobs, E. Poll, and S. Smetsers. AHA: Amortized Heap Space Usage Analysis. In M. Morazán, editor, *Selected Papers of the* 8^{th} *International Symposium on Trends in Functional Programming (TFP'07), New York, USA*, pages 36–53. Intellect Publishers, UK, 2007.
17. R. van Kesteren, O. Shkaravska, and M. van Eekelen. Inferring static non-monotonically sized types through testing. In *Proceedings of* 16^{th} *International Workshop on Functional and (Constraint) Logic Programming (WFLP'07), Paris, France*, volume 216C of *ENTCS*, pages 45–63, 2007.
18. P. B. Vasconcelos. *Space Cost Analysis Using Sized Types*. PhD thesis, School of Computer Science, University of St. Andrews, August 2008.