Collected Size Semantics for Functional Programs over Polymorphic Nested Lists *

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

Institute for Computing and Information Sciences
Radboud University Nijmegen

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

Bound on the resource consupmtion 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. 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 [13]. These multivalued size functions are used to annotate types. They make it possible to express that there can be more than one possible 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).

Consider e.g. the program $\mathsf{insert} : (\alpha \times \alpha \to \mathsf{Bool}) \times \alpha \times \mathsf{L}_n(\alpha) \to \mathsf{L}_{\mathit{insert}(n)}(\alpha)$ that inserts an element z of the type α in a list l, if this list does not contain an element z', such that the relation $\mathsf{g}(\mathsf{z}, \mathsf{z}')$ holds:

```
insert(g, z, l) = match l with | Nil ⇒ Cons(z, Nil)
                               | Cons(hd, tl) ⇒ if g(z, hd) then l
                                                else Cons(hd, insert(g, z, tl))
```

* This work is part of the AHA project [16] which is sponsored by the Netherlands Organisation for Scientific Research (NWO) under grant nr. 612.063.511.

Its size dependency $\mathit{insert}(n)$ represents the length of the output corresponding to an input of the length n. It is given by (where $\mid$ separates alternative rewriting rules):

$$\begin{array}{c} \vdash \mathit{insert}(0) \to 1 \\ n \geq 1 \vdash \mathit{insert}(n) \to n \mid \mathit{insert}(n-1) + 1 \end{array}$$

However, the type system from [13] 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*. Below, we first introduce the approach using a concrete example. 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 $M(\mathit{pos})$ of an internal list depending on its position pos in the outer list, where $0 \leq \mathit{pos} \leq n-1$. 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.

Informal Sketch of the Approach 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 expected annotated type for concat is $\mathsf{L}_n(\mathsf{L}_M(\alpha)) \to \mathsf{L}_{\mathit{concat}(n,M)}(\alpha)$ where n and M are size variables of types $\mathcal{N}$ (naturals) and $\mathcal{N} \to \mathcal{N}$, respectively and its size function $\mathit{concat}(n,\ M)$ is defined by the following rewriting rules

$$\vdash \mathit{concat}(0,\ M) \to 0 \qquad (1)$$

$$n \geq 1 \vdash \mathit{concat}(n,\ M) \to M(0) + \mathit{concat}(n-1,\ \lambda\, \mathit{pos}.\ M(\mathit{pos}+1)) \qquad (2)$$

where the first argument, n, of concat is the length of a "master" list of lists and the second argument, M, is a function that returns the size of an element at a given position pos in the master list. The head of a list is assumed to have position 0.

Consider the following expression: concat [$[1,2,3]$, $[4,5]$]. Here, $n=2$ and M is instantiated with a concrete function M_0 defined in a tabular way: $M_0(0)=3$, $M_0(1)=2$ and $M_0(\mathit{pos})$ for $\mathit{pos} \geq 2$ is arbitrary. We are now interested in calculating the result size defined by $\mathit{concat}(2,\ M_0)$:

$$\begin{array}{ll} \mathit{concat}(2,\ M_0) & \to M_0(0) + \mathit{concat}(1,\ \lambda\, \mathit{pos}.\ M_0(\mathit{pos}+1)) \\ & \to 3 + (\lambda\, \mathit{pos}.\ M_0(\mathit{pos}+1))(0) + \\ & \quad \mathit{concat}(1-1,\ \lambda\, \mathit{pos}'.\ \big((\lambda\ \mathit{pos}.\ M_0(\mathit{pos}+1))(\mathit{pos}'+1)\big)) \\ & = 3 + M_0(0+1) + \mathit{concat}(0,\ \lambda\, \mathit{pos}'.\ M_0((\mathit{pos}'+1)+1))) \\ & = 3 + M_0(1) + \mathit{concat}(0,\ \lambda\, \mathit{pos}'.\ M_0(\mathit{pos}'+2)) \\ & = 3 + M_0(1) + 0 = 3 + 2 + 0 = 5 \end{array}$$

However, a user often prefers to deal with *closed-form* size dependencies (i.e. without recursion) rather than with size functions given in the form of rewriting rules. We cannot always infer precise closed-forms but we will show that we can infer closed forms for *polynomial lower and upper bounds* of multivalued size functions. We focus on *piecewise polynomial* bounds, i.e., bounds that can be described by a finite number of polynomial families. 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 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.

In order to generate such bounds for general nested lists we need a nontrivial extension of the method described in [13]. In that work, size variables in rewriting rules are instantiated with finite numbers, whereas in this work we need to instantiate size variables of higher kinds with *finite multivalued maps*.

Here, we extend this methodology by instantiating size variables like M in $\mathit{concat}(n, M)$, of higher-order kinds, with finite multivalued maps. Consider how to infer size bounds for concat. Assume that the size of inner lists is at most n'. Then, the expected inferred result is $\{j\}_{0 \leq j \leq nn'}$ "covering" the range of $\mathit{concat}(n, M)$.

First, note that for fixed n and n' the map M will be finite and is "covered" by a finite multivalued map ϕ that sends any position to $\{0, \ldots, n'\}$. For instance, on $n = 2$ and $n' = 3$ the finite-multivalued-map variable ϕ is instantiated to ϕ_0, such that $\phi_0(0) = \phi(1) = \{0, 1, 2, 3\}$ and ϕ_0 is not defined on $\mathit{pos} \geq 2$. We will write finite multivalued maps as ordered sequences, so, e.g. ϕ_0 is $\langle \{0, 1, 2, 3\},\ \{0, 1, 2, 3\} \rangle$

From this point of view, *concat* in the "finite world" is presented by a function $\llcorner \mathit{concat} \lrcorner$ over finite sets and finite multivalued maps. The rewriting rules for $\llcorner \mathit{concat} \lrcorner$ are obtained by the obvious translation of the rewriting rules for concat.

$$\begin{array}{l} \vdash \llcorner \mathit{concat} \lrcorner(0,\ \phi) \to 0 \\ n \geq 1 \vdash \llcorner \mathit{concat} \lrcorner(n,\ \phi) \to \phi(0) +_{\{\}} \llcorner \mathit{concat} \lrcorner(n - 1,\ \phi_{+1}) \end{array}$$

where $\phi_+ := \lambda\, \mathit{pos}.\ \phi(\mathit{pos} + 1)$ is the 1-position-left shift for the finite sequence of sets presenting ϕ, and $+_{\{\}}$ is the elementwise addition of the elements of two sets. Note that for the sake of convenience n and 1 represent the singletons $\{n\}$ and $\{1\}$ respectively.

Now we want to infer a lower bound concat_l and an upper bound concat_u such that the family $\{\mathit{concat}_l + j\}_{0 \leq j \leq \mathit{concat}_u - \mathit{concat}_l}$ approximates *concat*. As in [13], the inferred family may not be an approximation of the actual output size, for instance, because the actual degree of bounds is higher than the one we have chosen. For that reason, there is a repeated procedure that starts with degree zero, infers, checks and finishes if the inferred family also checks correctly. If not it increments the degree and repeats the procedure. Now for the sake of brevity, assume that the first two steps (degrees zero and one) of this procedure have already been performed and that we are in the third step assuming degree two.

Assume that concat_l and concat_u are polynomials with degree $d = 2$. A bound on the size of an output for concat depends on two parameters, n and n'. So, an upper bound is a polynomial of degree two of two variables: $\mathit{concat}_u(n, n') = \gamma_{20}n^2 + \gamma_{11}nn' + \gamma_{02}(n')^2 + \gamma_{10}n + \gamma_{01}n' + \gamma_{00}$. Hence, to find concat_u one must know its value in 6 points. The same holds for concat_l. We evaluate the rewriting rules for $\llcorner \mathit{concat} \lrcorner$ in 6 points Let's start with $n = n' = 1$. Then ϕ is instantiated to $\langle\{0, 1\}\rangle$.

$$\begin{aligned}\llcorner \mathit{concat} \lrcorner(1,\,1) &\rightarrowtail \langle\{0,1\}\rangle(0) +_{\{\}} \llcorner \mathit{concat} \lrcorner(1-1,\,\langle\{0,1\}\rangle_{+1}) \\ &= \{0,1\} +_{\{\}} \llcorner \mathit{concat} \lrcorner(0,\,\langle\rangle) \rightarrow \{0,1\} +_{\{\}} \{0\} = \{0,1\}\end{aligned}$$

So, $\llcorner \mathit{concat} \lrcorner(1,\,1) = \{0, 1\}$. Similarly, $\llcorner \mathit{concat} \lrcorner(2,\,1) = \{0, 1, 2\}$, $\llcorner \mathit{concat} \lrcorner(3, 1) = \{0, 1, 2, 3\}$, $\llcorner \mathit{concat} \lrcorner(1,\,2) = \{0, 1, 2\}$, $\llcorner \mathit{concat} \lrcorner(2, 2) = \{0, 1, 2, 3, 4\}$ and finally $\llcorner \mathit{concat} \lrcorner(1,\,3) = \{0, 1, 2, 3\}$. Pick up the maximal values in these sets to define the right-hand side of the system of linear equations for the coefficients γ_{ij}:

$$\begin{aligned}&\gamma_{20} + \gamma_{02} + \gamma_{11} + \gamma_{10} + \gamma_{01} + \gamma_{00} \quad = 1 \\ &\ldots \\ &\gamma_{20} + 9\gamma_{02} + 3\gamma_{11} + \gamma_{10} + 3\gamma_{01} + \gamma_{00} = 3\end{aligned}$$

The solution is $(0, 0, 1, 0, 0, 0)$, so $\mathit{concat}_u(n, n') = nn'$. The similar system of concat_l has all zeros on its right-hand side. So, the coefficients for concat_l are all zeros. The inferred family is then $\{j'\}_{0 \le j' \le nn'}$.

Checking that the family obtained is indeed an approximation is done by checking the first-order predicate constructed in the following way. First, substitute in (1) and (2) the function applications for the corresponding approximations, the symbol $\rightarrow$ for $\supseteq$, and $+$ for $+_{\{\}}$:

$$\begin{aligned}&n = 0 \vdash \{j'\}_{0 \le j' \le nn'} \supseteq \{0\} \\ &n \ge 1 \vdash \{j'\}_{0 \le j' \le nn'} \supseteq \{j\}_{0 \le j \le n'} +_{\{\}} \{j''\}_{0 \le j'' \le (n-1)n'}\end{aligned}$$

Unfolding the definition of set inclusion gives the valid first-order predicates:

$$\begin{aligned}&\forall\, n. \quad n = 0 \qquad\qquad\qquad\qquad\qquad\qquad \vdash \exists\, j'.\ j' = 0 \wedge 0 \le j' \le nn' \\ &\forall\, j\, j''\, n \ge 1 \wedge 0 \le j \le n' \wedge 0 \le j'' \le (n-1)n' \vdash \exists\, j'.\ j' = j + j'' \ \wedge\ 0 \le j' \le nn'\end{aligned}$$

So, the inferred bounds of *concat* are accepted by the type checker, the loop is finished at $d = 2$ and the informal sketch of our approach is finished.

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.

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

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 $n+1$. This annotation is given in the output type of the function $\mathsf{insert} : (\alpha \times \alpha \to \mathsf{Bool}) \times \alpha \times \mathsf{L}_n(\alpha) \to \mathsf{L}_{\mathit{insert}(n)}(\alpha)$. The function insert, given a predicate $g : \alpha \times \alpha \to \mathsf{Bool}$, an element $\mathsf{z} : \alpha$ and a list $\mathsf{l} : \mathsf{L}_n(\alpha)$, inserts the element in the list if and only if there is no element in the list l related to z via g. 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: $\mathcal{R}^{(1)} \times \mathcal{R}^{(2)} \to 2^{\mathcal{R}}$. Here $\mathit{concat}(0, M) = 0$ and $\mathit{concat}(n, M) = M(0) + M(n-1,\, \lambda\, pos.\, M(pos+1))$ for $n \geq 1$.

– 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:

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

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 $0 \leq pos \leq n-1$.

$\mathit{tails}_2(n)(0) \quad = n$
$\mathit{tails}_2(n)(1) \quad = n-1, \ \text{if}\ n \geq 1$
$\mathit{tails}_2(n)(pos) = n - pos, \ \text{if}\ 0 \leq pos \leq n-1$
$\mathit{tails}_2(n)(pos) = \text{arbitrary if}\ pos \geq n$

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.

$$
\begin{array}{lllllllll}
p^1 & ::= a & \mid n,\ m & \mid f(p_1,\ldots,p_k) & \mid p^2(pos) & \mid p^2(pos-1) & \mid p^2(0) & \mid p^1_1\{+,-,*\}p^1_2 \\
p^2 & ::= B & \mid M & \mid g(p_1,\ldots,p_k) & \mid p^3(pos) & \mid p^3(pos-1) & \mid p^3(0) & \mid p^3_{+\mathbf{1}} \\
p^{s+1} & ::= a^s & \mid n^s & \mid f^s(p_1,\ldots,p_k) & \mid p^{s+1}(pos) & \mid p^{s+1}(pos-1) & \mid p^{s+1}(0) & \mid p^s_{+\mathbf{1}}
\end{array}
$$

where pos is a special variable of type $\mathcal{R}$ used to denote the position of an element in a list, and $p_{+\mathbf{1}}$ abbreviates $\lambda\ pos.\ p(pos)$. 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:

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

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 3.1. 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 $B(\mathit{pos}) = \mathit{pos}$ on $0 \leq \mathit{pos} \leq 1$. It is easy to see that $\mathit{concat}(2, B) = \{0\} + \{1\} = \{1\}$. Examples of their inhabitants are $[\mathsf{True},\ \mathsf{True}]$, $[\,[\,],\ [\mathsf{True}]\,]$ and $[\,\mathsf{True}\,]$, 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:

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

3.1 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:

$$
\begin{array}{lllll}
\mathit{Address} & \mathtt{adr} & ::= \ell \mid \mathtt{NULL} & \ell \in \mathit{Loc} \\
\mathit{Val} & v & ::= c \mid \mathtt{adr} & c \in \mathsf{Int} \cup \mathsf{Bool} \\
\mathit{Heap} & h & : \quad \mathit{Loc} \rightharpoonup \{\mathsf{hd}, \mathsf{tl}\} \rightharpoonup \mathit{Val} &
\end{array}
$$

We will write $h.\ell.\mathsf{hd}$ and $h.\ell.\mathsf{tl}$ for the results of applications $h\ \ell\ \mathsf{hd}$ and $h\ \ell\ \mathsf{tl}$, which denote the values stored in the heap h at the location ℓ at its fields hd and tl, respectively. Let $h.\ell.[\mathsf{hd} := v_h,\ \mathsf{tl} := v_t]$ 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* $\tau^\bullet$ is a set-theoretic interpretation given via the four-place relation $v \models^h_{\tau^\bullet} w$. Integer and boolean constants interpret themselves, and locations are interpreted as *non-cyclic lists*. Let $p^1(\overline{n}^*_0)$ denote the set of values of some expression p^1 applied to some values $\overline{n}^*_0$. Then

$$
\begin{array}{lll}
c & \models^h_{\mathsf{Int} \cup \mathsf{Bool}} & c \\
\mathtt{NULL} & \models^h_{\mathsf{L}_{p^1(\overline{n}^*_0)}(\tau^\bullet)} & [] \ \ \mathit{iff}\ \ 0 \in p^1(\overline{n}^*_0) \\
\ell & \models^h_{\mathsf{L}_{p^1(\overline{n}^*_0)}(\tau^\bullet)} & w_{\mathsf{hd}} :: w_{\mathsf{tl}} \ \mathit{iff}\ \ h.\ell.\mathsf{hd} \models^{h|_{dom(h)\setminus\{\ell\}}}_{\tau^\bullet(0)} w_{\mathsf{hd}}, \\
 & & \qquad\qquad\qquad\qquad\quad h.\ell.\mathsf{tl} \models^{h|_{dom(h)\setminus\{\ell\}}}_{\mathsf{L}_{p^1(\overline{n}^*_0)-1}(\tau^\bullet_{+\mathbf{1}})} w_{\mathsf{tl}}
\end{array}
$$

where $h|_{dom(h)\setminus\{\ell\}}$ denotes the heap equal to h everywhere except in ℓ, where it is undefined, $(p^s)_{+\mathbf{1}}$ and $\tau_{+\mathbf{1}}$ are abbreviations for $\lambda\, pos.\ p^s(pos+1)$ and $\lambda\, pos.\ \tau(pos+1)$, respectively and the application of a type to a first-layer size expression $\tau(p^1)$ is defined as follows:

$$
\begin{array}{l}
\tau^0(p^1) = \tau^0 \\
\tau^{1,\,s}(p^1) = \tau^{1,\,s} \\
(\mathsf{L}_{p^{s'}}(\tau^{s'+1\ s}))(p^1) := \mathsf{L}_{p^{s'}(p^1)}(\tau^{s'+1\ s}(p^1)),\ \text{for } s' \geq 2
\end{array}
$$

The $\mathit{length}_\_(-) : \mathit{Heap} \rightharpoonup \mathit{Address} \rightharpoonup \mathcal{N}$ of a *non-cyclic* chain of cons-cells in a heap is defined by induction in a usual way: $\mathit{length}_h(\mathtt{NULL}) = 0$ and $\mathit{length}_h(\ell) = 1 + \mathit{length}_{h|_{dom(h)\setminus\{\ell\}}}(h.\ell.\mathsf{tl})$. Note that the function $\mathit{length}_h(-)$ 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* $\mathtt{adr} \models^h_{\mathsf{L}_{p^1(\overline{n}^*_0)}(\tau^\bullet)} w$ *implies that* $\mathit{length}_h(\mathtt{adr}) \in p^1(\overline{n}^*_0)$.

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 [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: *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 rules are as follows:

$$\frac{c \in \mathsf{Int} \cup \mathsf{Bool}}{s;\ h;\ \mathcal{C} \vdash c \rightsquigarrow c;\ h}\ \textsc{OSCons} \qquad \frac{}{s;\ h;\ \mathcal{C} \vdash \mathsf{z} \rightsquigarrow s(\mathsf{z});\ h}\ \textsc{OSVar}$$

$$\frac{}{s;\ h;\ \mathcal{C} \vdash \mathsf{Nil} \rightsquigarrow \mathtt{NULL};\ h}\ \textsc{OSNil}$$

$$\frac{s(\mathsf{hd}) = v_{\mathsf{hd}} \qquad s(\mathsf{tl}) = v_{\mathsf{tl}} \qquad \ell \notin dom(h)}{s;\ h \vdash \mathsf{Cons}(\mathsf{hd},\ \mathsf{tl}) \rightsquigarrow \ell;\ h[\ell.\mathsf{hd} := v_{\mathsf{hd}},\ \ell.\mathsf{tl} := v_{\mathsf{tl}}]}\ \textsc{OSCons}$$

$$\frac{s(\mathsf{x}) = \mathsf{True} \qquad s;\ h;\ \mathcal{C} \vdash e_1 \rightsquigarrow v;\ h'}{s;\ h;\ \mathcal{C} \vdash \mathsf{if\ x\ then}\ e_1\ \mathsf{else}\ e_2 \rightsquigarrow v;\ h'}\ \textsc{OSIfTrue}$$

$$\frac{s(\mathsf{x}) = \mathsf{False} \qquad s;\ h;\ \mathcal{C} \vdash e_2 \rightsquigarrow v;\ h'}{s;\ h;\ \mathcal{C} \vdash \mathsf{if\ x\ then}\ e_1\ \mathsf{else}\ e_2 \rightsquigarrow v;\ h'}\ \textsc{OSIfFalse}$$

$$\frac{s;\ h;\ \mathcal{C} \vdash e_1 \rightsquigarrow v_1;\ h_1 \qquad s[\mathsf{z} := v_1];\ h_1;\ \mathcal{C} \vdash e_2 \rightsquigarrow v;\ h'}{s;\ h;\ \mathcal{C} \vdash \mathsf{let\ z} = e_1\ \mathsf{in}\ e_2 \rightsquigarrow v;\ h'}\ \textsc{OSLet}$$

$$\frac{s(\mathsf{l}) = \mathtt{NULL} \qquad s;\ h;\ \mathcal{C} \vdash e_1 \rightsquigarrow v;\ h'}{\begin{array}{l}s;\ h;\ \mathcal{C} \vdash \mathsf{match\ l\ with} \mid \mathsf{Nil} \Rightarrow e_1 \\ \qquad\qquad\qquad\qquad\quad \mid \mathsf{Cons}(\mathsf{hd},\ \mathsf{tl}) \Rightarrow e_2\end{array} \rightsquigarrow v;\ h'}\ \textsc{OSMatch-Nil}$$

$$\frac{\begin{array}{c}h.s(\mathsf{l}).\mathsf{hd} = v_{\mathsf{hd}} \qquad h.s(\mathsf{l}).\mathsf{tl} = v_{\mathsf{tl}} \\ s[\mathsf{hd} := v_{\mathsf{hd}}, \mathsf{tl} := v_{\mathsf{tl}}];\ h;\ \mathcal{C} \vdash e_2 \rightsquigarrow v;\ h'\end{array}}{\begin{array}{l}s;\ h;\ \mathcal{C} \vdash \mathsf{match\ l\ with} \mid \mathsf{Nil} \Rightarrow e_1 \\ \qquad\qquad\qquad\qquad\quad \mid \mathsf{Cons}(\mathsf{hd},\ \mathsf{tl}) \Rightarrow e_2\end{array} \rightsquigarrow v;\ h'}\ \textsc{OSMatch-Cons}$$

$$\frac{s;\ h;\ \mathcal{C}[f := ((\mathsf{g}_1, \ldots, \mathsf{g}_{k'},\ \mathsf{z}_1, \ldots, \mathsf{z}_k) \times e_1)] \vdash e_2 \rightsquigarrow v;\ h'}{s;\ h;\ \mathcal{C} \vdash \mathsf{letfun\ f}(\mathsf{g}_1, \ldots, \mathsf{g}_{k'},\ \mathsf{z}_1, \ldots, \mathsf{z}_k) = e_1\ \mathsf{in}\ e_2 \rightsquigarrow v;\ h'}\ \textsc{OSLetFun}$$

$$\frac{\begin{array}{c}s(\mathsf{z}'_1) = v_1\ \ \ldots\ \ s(\mathsf{z}'_k) = v_k \\ \mathcal{C}(\mathsf{f}) = (\mathsf{g}_1, \ldots, \mathsf{g}_{k'},\ \mathsf{z}_1, \ldots, \mathsf{z}_k) \times e_f \\ [\mathsf{z}_1 := v_1, \ldots, \mathsf{z}_k := v_k];\ h;\ \mathcal{C} \vdash e_f[\mathsf{g}_1 := \mathsf{f}_1, \ldots, \mathsf{g}_{k'} := \mathsf{f}_{k'}] \rightsquigarrow v;\ h'\end{array}}{s;\ h;\ \mathcal{C} \vdash \mathsf{f}(\mathsf{f}_1, \ldots, \mathsf{f}_{k'},\ \mathsf{z}'_1, \ldots, \mathsf{z}'_k) \rightsquigarrow v;\ h'}\ \textsc{OSFunApp}$$

3.3 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.:

$$D \qquad \vdash p^1(\overline{n}^*) \to p'^1(\overline{n}^*)$$

$$D,\ m_1 \in p'^1(\overline{n}^*),\ 0 \leq pos \leq m_1 - 1 \qquad \vdash p^2(\overline{n}^*)(pos) \to p'^2(\overline{n}^*)(pos)$$
$$m_1,\ pos \text{ are fresh for } D$$

$$D, \left\{\begin{array}{l} m_1 \in p'^1(\overline{n}^*),\ 0 \leq pos_1 \leq m_1 - 1, \\ m_2 \in p'^2(\overline{n}^*)(pos_1),\ 0 \leq pos_2 \leq m_2 - 1 \end{array}\right\} \vdash \left\{\begin{array}{l} p^3(\overline{n}^*)(pos_1)(pos_2) \to \\ p'^3(\overline{n}^*)(pos_1)(pos_2) \end{array}\right.$$
$$m_1,\ pos_1,\ m_2,\ pos_2 \text{ are fresh for } D$$

$\ldots$

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

The typing judgement relation is defined by the following rules:

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

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

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

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:

$$\begin{array}{ll} n \geq 1 & \vdash \mathit{tails}_2(n)(0) \to n \\ n \geq 1,\ 1 \leq \mathit{pos} \leq n & \vdash \mathit{tails}_2(n)(\mathit{pos}) \to \mathit{tails}_2(n-1)(\mathit{pos}-1) \end{array}$$

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

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

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

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

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 1: inferring rewriting rules for concat In the introduction we have given the rewriting rules defining the type for $\mathsf{concat} : \mathsf{L}_n(\mathsf{L}_M(\alpha)) \to \mathsf{L}_{\mathit{concat}(n,M)}(\alpha)$, where

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

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: $\mathsf{l}\colon \mathsf{L}_n(\mathsf{L}_M(\alpha)) \vdash_\Sigma e_{\mathsf{concat}} \colon \mathsf{L}_{\mathit{concat}(n,M)}(\alpha)$, where e_{concat} denotes the body of concat.
2. Apply the Match-rule. In the nil-branch we obtain the subgoal $n = 0;\ \mathsf{l}\colon \mathsf{L}_n(\mathsf{L}_M(\alpha)) \vdash_\Sigma \mathsf{Nil}\colon \mathsf{L}_{\mathit{concat}(n,M)}(\alpha)$.
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 $(-)_{+\mathbf{1}}$: $n \geq 1 \vdash \mathit{concat}(n, M) \to M(0) + \mathit{concat}(n-1,\ \lambda\ \mathit{pos}.\, M(\mathit{pos}+1))$.

Example 2: inferring rewriting rules for tails Now we want to infer the rewriting rules for the size annotations in the type $\mathsf{tails} : \mathsf{L}_n(\alpha) \to \mathsf{L}_{\mathit{tails}_1(n)}(\mathsf{L}_{\mathit{tails}_2(n)}(\alpha))$. Recall, that the closed forms of the annotations are $\mathit{tails}_1(n) = n$ and $n \geq 1,\ 0 \leq \mathit{pos} \leq n-1 \vdash \mathit{tails}_2(n)(\mathit{pos}) = n - \mathit{pos}$, respectively. In this example we show how output lists of lists are treated.

1. The LetFun rule defines the main goal: $\mathsf{l}\colon \mathsf{L}_n(\alpha) \vdash_\Sigma e_{\mathsf{tails}}\colon \mathsf{L}_{\mathit{tails}_1(n)}(\mathsf{L}_{\mathit{tails}_2(n)}(\alpha))$, where e_{tails} denotes the body of tails.
2. Apply the Match-rule. In the nil-branch we obtain the subgoal $n = 0;\ \mathsf{l}\colon \mathsf{L}_n(\alpha) \vdash_\Sigma \mathsf{Nil}\colon \mathsf{L}_{\mathit{tails}_1(n)}(\mathsf{L}_{\mathit{tails}_2(n)}(\alpha))$.
3. Continue with the nil-branch. Apply the Nil rule and obtain $n = 0 \vdash \mathsf{L}_{\mathit{tails}_1(n)}(\mathsf{L}_{\mathit{tails}_2(n)}(\alpha)) \to \mathsf{L}_0(\tau^?)$.
4. Trivially, instantiate $\tau^? := \mathsf{L}_{\mathit{tails}_2(n)}(\alpha)$. Unfold the definition of the type rewriting: $n = 0 \vdash \mathit{tails}_1(n) \to 0$.
   Note, that the rewriting rules for $\mathit{tails}_2(n)$ in this branch are absent, since $n_1 \in \{n = 0\},\ 0 \leq \mathit{pos} \leq n_1 - 1$ is an empty set.
5. Now, consider the cons-branch. The subgoal there is $n \geq 1;\ \mathsf{l}\colon \mathsf{L}_n(\alpha),\ \mathsf{tl}\colon \mathsf{L}_{n-1}(\alpha_{+\mathbf{1}}) \vdash_\Sigma \mathsf{Cons}(\mathsf{l}, \mathsf{tails}(\mathsf{tl}))\colon \mathsf{L}_{\mathit{tails}_1(n)}(\mathsf{L}_{\mathit{tails}_2(n)}(\alpha))$. (Again, that in contexts we omit variables, on which the expression in the typing judgement under consideration does not depend.)

6. In the type of tl unfold the definition of $(-)_{+\mathtt{1}}$:
   $n \geq 1;\ \mathsf{l}\colon \mathsf{L}_n(\alpha),\ \mathsf{tl}\colon \mathsf{L}_{n-1}(\alpha) \vdash_\Sigma \mathsf{Cons}(\mathsf{l}, \mathsf{tails}(\mathsf{tl}))\colon \mathsf{L}_{\mathit{tails}_1(n)}(\mathsf{L}_{\mathit{tails}_2(n)}(\alpha))$.
7. The expression in the judgement above is a sugared let-construct. So, we apply the LET-rule. In the binding we have the subgoal: $n \geq 1;\ \mathsf{tl}\colon \mathsf{L}_{n-1}(\alpha) \vdash_\Sigma \mathsf{tails}(\mathsf{tl})\colon \tau^?$.
8. Using FUNAPP-rule we instantiate the type $\tau^? := \mathsf{L}_{\mathit{tails}_1(n-1)}(\mathsf{L}_{\mathit{tails}_2(n-1)}(\alpha))$.
9. Therefore, the subgoal for the let-body is
   $n \geq 1;\ \mathsf{l}\colon \mathsf{L}_n(\alpha),\ \mathsf{z}\colon \mathsf{L}_{\mathit{tails}_1(n-1)}(\mathsf{L}_{\mathit{tails}_2(n-1)}(\alpha)) \vdash_\Sigma \mathsf{Cons}(\mathsf{hd}, \mathsf{z})\colon \mathsf{L}_{\mathit{tails}_1(n)}(\mathsf{L}_{\mathit{tails}_2(n)}(\alpha))$.
10. Apply the CONS-rule. We obtain the predicates
$$\begin{array}{ll} n \geq 1 & \vdash \mathsf{L}_{\mathit{tails}_1(n)}(\mathsf{L}_{\mathit{tails}_2(n)}(\alpha)) \to \mathsf{L}_{\mathit{tails}_1(n-1)+1}(\tau^2) \\ n \geq 1 & \vdash \tau^2(0) \to \mathsf{L}_n(\alpha) \\ n \geq 1,\ 1 \leq \mathit{pos} \leq n & \vdash \tau^2(\mathit{pos}) \to (\mathsf{L}_{\mathit{tails}_2(n-1)}(\alpha))(\mathit{pos}-1) \end{array}$$
11. Trivially, instantiate $\tau^2 := \mathsf{L}_{\mathit{tails}_2(n)}(\alpha)$. We obtain
$$\begin{array}{ll} n \geq 1 & \vdash \mathsf{L}_{\mathit{tails}_1(n)}(\mathsf{L}_{\mathit{tails}_2(n)}(\alpha)) \to \mathsf{L}_{\mathit{tails}_1(n-1)+1}(\mathsf{L}_{\mathit{tails}_2(n)}(\alpha)) \\ n \geq 1 & \vdash (\mathsf{L}_{\mathit{tails}_2(n)}(\alpha))(0) \to \mathsf{L}_n(\alpha) \\ n \geq 1,\ 1 \leq \mathit{pos} \leq n & \vdash \mathsf{L}_{\mathit{tails}_2(n)}(\alpha)(\mathit{pos}) \to (\mathsf{L}_{\mathit{tails}_2(n-1)}(\alpha))(\mathit{pos}-1) \end{array}$$
12. Unfold the definition of type-typewriting. For tails_1 we obtain $n \geq 1 \vdash \mathit{tails}_1(n) = \mathit{tails}_1(n-1)+1$, and for tails_2, unfolding the definition of application of a type to a first-layer expression, we obtain
$$\begin{array}{ll} n \geq 1 & \vdash \mathit{tails}_2(n)(0) \to n \\ n \geq 1,\ 1 \leq \mathit{pos} \leq n & \vdash \mathit{tails}_2(n)(\mathit{pos}) \to \mathit{tails}_2(n-1)(\mathit{pos}-1) \end{array}$$

It is easy to see that $\mathit{tails}_1(n) = n$ is a closed form for the obtained rewriting system for f: $\mathit{tails}_1(0) = 0$ and $\mathit{tails}_1(n) \to \mathit{tails}_1(n-1)+1$ with $n \geq 1$. Further, $\mathit{tails}_2(n)(\mathit{pos}) = n-i$ for $0 \leq \mathit{pos} \leq n-1$ solves the rewriting system for g. Indeed, by induction on $n \geq 2$, $\mathit{tails}_2(n)(\mathit{pos}) = \mathit{tails}_2(n-1)(\mathit{pos}-1) = (n-1)-(\mathit{pos}-1) = n-i$ for $i \geq 1$, with the base $\mathit{tails}_2(1)(0) = 1$, and having $\mathit{tails}_2(n)(\mathit{pos}) = n$ for $\mathit{pos} = 0$.

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 $\Gamma^\bullet$:

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

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:

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

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: $(\mathsf{L}_{p(\overline{n}^*)}(\tau))_{\eta\epsilon} = \mathsf{L}_{p(\overline{n}^*_\epsilon)}(\tau_{\eta\epsilon})$.

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

Proof. Induction on $\models$. The case where v is an integer or a boolean is straightforward since τ' and τ will be Int or Bool, respectively.

Let $\tau = \mathsf{L}_{p^1(\overline{n}^*)}(\tau'')$ and $\tau' = \mathsf{L}_{p^1(\overline{n}^*)}(\tau''')$ for some τ'' and τ''', and let $\epsilon(\overline{n}^*) = \overline{n}^*_0$.

Assume $v = \texttt{NULL}$. Then $0 \in p'_1(\overline{n}_0)$ and $w = []$. Since $p_1(\overline{n}^*_0) \to p'_1(\overline{n}^*_0)$, that is $p'_1(\overline{n}^*_0) \subseteq p_1(\overline{n}^*_0)$, we have $0 \in p_1(\overline{n}^*_0)$ and $v \models^h_{\tau_{\eta\epsilon}} []$.

Now assume that $v = \ell$ and $w = w_{hd} :: w_{tl}$, where $h.\ell.hd \models^{h|_{dom(h)\setminus\{\ell\}}}_{\tau'''_{\eta\epsilon}} w_{hd}$ and $h.\ell.tl \models^{h|_{dom(h)\setminus\{\ell\}}}_{\mathsf{L}_{f'(\overline{n}_0)-1}(\tau'''_{\eta\epsilon})} w_{tl}$. Since there is $n \in p_1(\overline{n}^*_0)$ with $n \geq 1$ (because we are in the non-empty case), we have $D \vdash \tau'' \to \tau'''$, from which follows that $D \vdash \tau''_{+1} \to \tau'''_{+1}$. Then, by induction, $h.\ell.hd \models^{h|_{dom(h)\setminus\{\ell\}}}_{\tau''_{\eta\epsilon}(0)} w_{hd}$. Since $p_1(\overline{n}^*) \to p'_1(\overline{n}^*)$, we have that $p_1(\overline{n}^*) - 1 \to p'_1(\overline{n}^*) - 1$, and by induction $h.\ell.tl \models^{h|_{dom(h)\setminus\{\ell\}}}_{\mathsf{L}_{p'_1(\overline{n}^*_0)-1}((\tau''_{+1})_{\eta\epsilon})} w_{tl}$. □

This lemma may seem counterintuitive on a first sight because it looks like a type preservation lemma where the type τ and τ' are swapped. However, a rewriting rule is different from an evaluation step. The idea behind this lemma is that on a rewriting rule there are several choices on the left hand side (τ) and one in particular is chosen to obtain the right hand side (τ'). So, if a value has type τ', it also has type τ.

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. This is possible because we assume that the evaluation of e termintates (with a value v). We have to show that $Valid_{\mathsf{val}}(v, \tau_{\eta\epsilon}, h')$, i.e., that there is a w such that $v \models^{h'}_{\tau_{\eta\epsilon}} w$. This is proved for each of the operational-semantic rules.

OSNull: In this case $e = \mathsf{Nil}$, $v = \texttt{NULL}$ and $h' = h$. From the NIL typing rule we have that $D \vdash \tau \to \mathsf{L}_0(\tau')$. According to the definition of rewriting rule, $\tau = \mathsf{L}_{p(\overline{n}^*)}(\tau'')$ for some p, $\overline{n}^*$ and τ'', where $p(\overline{n}^*) \to 0$. But then $p(\overline{n}^*_\epsilon) \to 0$ and hence $0 \in p(\overline{n}^*_\epsilon)$. But then from the definition of model relation we get that $\texttt{NULL} \models^{h'}_{\mathsf{L}_{p(\overline{n}^*_\epsilon)}(\tau''_{\eta\epsilon})} \texttt{[]}$ and thus $\texttt{NULL} \models^{h'}_{\tau_{\eta\epsilon}} \texttt{[]}$.

OSVar: In this case $e = \mathsf{z}$, $v = s(\mathsf{z})$ and $h' = h$. Since $dom(s) = dom(\Gamma)$, there is a τ' such that $\Gamma(\mathsf{z}) = \tau'$, and because $Valid_{\mathsf{store}}(dom(s), \Gamma_{\eta\epsilon}, s, h)$, there is a w such that $s(\mathsf{z}) \models^{h}_{\tau'_{\eta\epsilon}} w$. Now from the VAR typing rule, $D \vdash \tau \to \tau'$. Since D_ϵ holds, we can now apply the Lemma 2 to obtain $v \models^{h'}_{\tau_{\eta\epsilon}} w$.

OSCons: In this case $e = \mathsf{Cons}(\mathsf{hd}, \mathsf{tl})$, $v = \ell$ for some location $\ell \notin dom(h)$ and $h' = h.\ell.[hd := s(\mathsf{hd}), tl := s(\mathsf{tl})]$.
From the CONS typing rule we have that $\mathsf{hd} : \tau_1$ and $\mathsf{tl} : \mathsf{L}_{p^1(\overline{n}^*)}(\tau_2)$, and the judgements $D \vdash \tau' \to \mathsf{L}_{p^1(\overline{n}^*)+1}(\tau'_2)$, $D \vdash \tau'_2(0) \to \tau_1$ and $n \in p^1(\overline{n}^*), 1 \le pos \le n, D \vdash \tau'_2(pos) \to \tau_2(pos-1)$. Since $Valid_{\mathsf{store}}(dom(s), \Gamma_{\eta\epsilon}, s, h)$, there exist w_{hd} and w_{tl} such that $s(\mathsf{hd}) \models^{h}_{\tau_{1\eta\epsilon}} w_{hd}$ and $s(\mathsf{tl}) \models^{h}_{\mathsf{L}_{p^1(\overline{n}^*_\epsilon)}(\tau_{2\eta\epsilon})} w_{tl}$. Therefore, $h'.\ell.hd \models^{h}_{\tau_{1\eta\epsilon}} w_{hd}$ and $h'.\ell.tl \models^{h}_{\mathsf{L}_{p^1(\overline{n}^*_\epsilon)}(\tau_{2\eta\epsilon})} w_{tl}$.
It is easy to see that $h = h'|_{dom(h')\setminus\{\ell\}}$, thus, $h'.\ell.hd \models^{h'|_{dom(h')\setminus\{\ell\}}}_{\tau_{1\eta\epsilon}} s(\mathsf{hd})$ and $h'.\ell.tl \models^{h'|_{dom(h')\setminus\{\ell\}}}_{\mathsf{L}_{p^1(\overline{n}^*_\epsilon)}(\tau_{2\eta\epsilon})} s(\mathsf{tl})$. From the judgement $D \vdash \tau'_2(0) \to \tau_1$ and Lemma 2, $h'.\ell.hd \models^{h'|_{dom(h')\setminus\{\ell\}}}_{\tau'_{2\eta\epsilon}(0)} w_{hd}$. Now we want to show that $h'.\ell.tl \models^{h'|_{dom(h')\setminus\{\ell\}}}_{\mathsf{L}_{p^1(\overline{n}^*_\epsilon)+1}(\tau'_{2\eta\epsilon})} w_{tl}$, and then by the definition of model relation we can obtain the desired result: $\ell \models^{h'}_{\tau_{\eta\epsilon}} w_{hd} :: w_{tl}$.
The judgement $n \in p^1(\overline{n}^*), 1 \le pos \le n, D \vdash \tau'_2(pos) \to \tau_2(pos-1)$. is equivalent to $n \in p^1(\overline{n}^*), 1 \le pos \le n, D \vdash \tau'_2(pos+1) \to \tau_2(pos)$. Recall that τ_{+1} is defined as $\lambda\, pos.\ \tau(pos+1)$, thus we have the judgement $n \in p^1(\overline{n}^*), 1 \le pos \le n, D \vdash (\tau'_2)_{+1}(pos) \to \tau_2(pos)$. Now by definition of rewriting rules, $n \in p^1(\overline{n}^*), 1 \le pos \le n, D \vdash \mathsf{L}_{p^1(\overline{n}^*)}((\tau'_2)_{+1}) \to \mathsf{L}_{p^1(\overline{n}^*)}(\tau_2)$. Instantiating Lemma 2 with this judgement and $h'.\ell.tl \models^{h'|_{dom(h')\setminus\{\ell\}}}_{\mathsf{L}_{p^1(\overline{n}^*_\epsilon)}(\tau_{2\eta\epsilon})} w_{tl}$, we get $h'.\ell.tl \models^{h'|_{dom(h')\setminus\{\ell\}}}_{\mathsf{L}_{p^1(\overline{n}^*_\epsilon)+1}(\tau'_{2\eta\epsilon})} w_{tl}$.

OSIfTrue: In this case $e = \mathsf{if}\ \mathsf{x}\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2$, with $s(\mathsf{x}) = \mathsf{True}$ and $s;\ h;\ \mathcal{C} \vdash e_1 \rightsquigarrow v;\ h'$. From the IF typing rule we get that $D, \Gamma \vdash_\Sigma e_1 : \tau_1$, $D, \Gamma \vdash_\Sigma e_2 : \tau_2$ and $D \vdash \tau \to \tau_1 \mid \tau_2$. Since e_1 is evaluated in the same context as e ($s;\ h;\ \mathcal{C}$), we can use the induction hypothesis to get $Valid_{\mathsf{val}}(v, \tau_{1\eta\epsilon}, h')$. Since by definition of $D \vdash \tau \to \tau_1 \mid \tau_2$, $D \vdash \tau \to \tau_1$, using Lemma 2 we obtain $Valid_{\mathsf{val}}(v, \tau_{\eta\epsilon}, h')$.

OSIfFalse: Similar to the true case.

OSLet: In this case e is $\mathsf{let}\ \mathsf{z} = e_1\ \mathsf{in}\ e_2$, where $s;\ h;\ \mathcal{C} \vdash e_1 \rightsquigarrow v_1;\ h_1$ and $s[\mathsf{z} := v_1];\ h_1;\ \mathcal{C} \vdash e_2 \rightsquigarrow v;\ h'$. From the LET typing rule we have that $\mathsf{z} \notin dom(\Gamma)$, $D, \Gamma \vdash_\Sigma e_1 : \tau'$ and $D, \Gamma, \mathsf{z} : \tau' \vdash_\Sigma e_2 : \tau$. Applying the induc-

tion hypothesis to the antecedents of the operational semantics, we get that $\mathit{Valid}_{\mathsf{val}}(v_1, \tau'_{\eta\epsilon}, h_1)$ and that if $\mathit{Valid}_{\mathsf{store}}(\mathit{dom}(s[\mathsf{z} := v_1]), \Gamma_{\eta\epsilon} \cup \{\mathsf{z}\colon \tau'_{\eta\epsilon}\}, s[\mathsf{z} := v_1], h_1)$ then $\mathit{Valid}_{\mathsf{val}}(v, \tau_{\eta\epsilon}, h')$.
Fix some $\mathsf{z}' \in \mathit{dom}(s[\mathsf{z} := v_1])$. If $\mathsf{z}' = \mathsf{z}$, then $\mathit{Valid}_{\mathsf{val}}(v_1, \tau'_{\eta\epsilon}, h_1)$ implies $\mathit{Valid}_{\mathsf{val}}(s[\mathsf{z} := v_1](\mathsf{z}), \tau'_{\eta\epsilon}, h_1)$. If $\mathsf{z}' \neq \mathsf{z}$, then $s[\mathsf{z} := v_1](\mathsf{z}') = s(\mathsf{z}')$. Sharing of data structures in the heap is benign (no destructive pattern matching and assignments), hence $h|_{\mathcal{R}(h,\ s(\mathsf{z}'))} = h_1|_{\mathcal{R}(h,\ s(\mathsf{z}'))}$. Thus, we have that $s(\mathsf{z}') \models^{h}_{\Gamma_{\eta\epsilon}(\mathsf{z}')} w'_z$ implies $s(\mathsf{z}') \models^{h_1}_{\Gamma_{\eta\epsilon}(\mathsf{z}')} w'_z$ and then $s[\mathsf{z} := v_1](\mathsf{z}') \models^{h_1}_{\Gamma_{\eta\epsilon}(\mathsf{z}')} w_{\mathsf{z}'}$. So, $\mathit{Valid}_{\mathsf{val}}(s[\mathsf{z} := v_1](\mathsf{z}'), \Gamma_{\eta\epsilon}(\mathsf{z}'), h_1)$. Hence, $\mathit{Valid}_{\mathsf{store}}(\mathit{dom}(s[\mathsf{z} := v_1]), \Gamma_{\eta\epsilon} \cup \{\mathsf{z}\colon \tau'_{\eta\epsilon}\}, s[\mathsf{z} := v_1], h_1)$ and we can now apply the induction hypothesis.

OSMatch-Nil: In this case $e = \mathsf{match\ l\ with} \mid \mathsf{Nil} \Rightarrow e_1 \mid \mathsf{Cons(hd, tl)} \Rightarrow e_2$ where $s(\mathsf{l}) = \mathtt{NULL}$ and $s;\ h;\ \mathcal{C} \vdash e_1 \rightsquigarrow v;\ h'$. From the MATCH typing rule we have that $\mathsf{l}\colon \mathsf{L}_{p^1(\overline{n}^*)}(\tau')$ and $D, 0 \in p^1(\overline{n}^*),\ \Gamma', \mathsf{l}\colon \mathsf{L}_{p^1(\overline{n}^*)}(\tau') \vdash_{\Sigma} e_1 \colon \tau$.
From $\mathit{Valid}_{\mathsf{store}}(\mathit{dom}(s), \Gamma_{\eta\epsilon}, s, h)$ we get $\mathit{Valid}_{\mathsf{val}}(s(\mathsf{l}), \mathsf{L}_{p^1(\overline{n}^*_\epsilon)}(\tau'_{\eta\epsilon}), h)$ and since $s(\mathsf{l}) = \mathtt{NULL}$, from the definition of model relation we get that $0 \in p^1(\overline{n}^*)$.
Therefore, the typing judgement about e_1 reduces to $D,\ \Gamma', \mathsf{l}\colon \mathsf{L}_{p^1(\overline{n}^*)}(\tau') \vdash_{\Sigma} e_1 \colon \tau$, where $\Gamma = \Gamma', \mathsf{l}\colon \mathsf{L}_{p^1(\overline{n}^*)}(\tau'_{\eta\epsilon})$. We can now apply the induction hypothesis to obtain $\mathit{Valid}_{\mathsf{val}}(v, \tau_{\eta\epsilon}, h')$.

OSMatch-Cons: In this case $e = \mathsf{match\ l\ with} \mid \mathsf{Nil} \Rightarrow e_1 \mid \mathsf{Cons(hd, tl)} \Rightarrow e_2$. The typing context has the form $\Gamma = \Gamma' \cup \{\mathsf{l} : \mathsf{L}_{p^1(\overline{n}^*)}(\tau')\}$. From the operational semantics we know that $h.s(\mathsf{l}).\mathit{hd} = v_{hd}$ and $h.s(\mathsf{l}).\mathit{tl} = v_{tl}$, that is, $s(\mathsf{l}) \neq \mathtt{NULL}$. Due to the validity of $s(\mathsf{l})$ and Lemma 1, there exists $n_0 \geq 1 \in p^1(\overline{n}^*_\epsilon)$. From the validity $s(\mathsf{l}) \models^{h}_{\mathsf{L}_{p^1(\epsilon(\overline{n}^*))}(\tau'_{\eta\epsilon})} w_{hd} :: w_{tl}$, the validities of v_{hd} and v_{tl} follow: $v_{hd} \models^{h}_{\tau'_{\eta\epsilon}(0)} w_{hd}$ and $v_{tl} \models^{h}_{\mathsf{L}_{p^1_{\overline{n}^*_\epsilon - 1}((\tau'_{\eta\epsilon})+1)}} w_{tl}$.
From the MATCH typing rule we have that $D, n_0 \geq 1 \in p^1(\overline{n}^*);\ \Gamma'' \vdash_{\Sigma} e_2 \colon \tau_{\eta\epsilon}$, where $\Gamma'' = \Gamma \cup \{\mathsf{hd}\colon \tau'(0), \mathsf{tl}\colon \mathsf{L}_{p^1(\overline{n}^*)-1}(\tau'_{+\mathbf{1}})\}$.
From $\mathit{Valid}_{\mathsf{store}}(\mathit{dom}(s), \Gamma_{\eta\epsilon}, s, h)$ and the results above, we obtain that $\mathit{Valid}_{\mathsf{store}}(\mathit{dom}(s'), \Gamma''_{\eta\epsilon}, s', h)$, where $s' = s[\mathsf{hd} := v_{hd}][\mathsf{tl} := v_{tl}]$. With $\epsilon' = \epsilon[n_0 := \mathit{length}_h(s(\mathsf{l}))]$, the induction hypothesis yields $\mathit{Valid}_{\mathsf{val}}(v, \tau_{\eta\epsilon'}, h')$.
Now, since $n_0 \notin \mathit{SV}(\tau)$ (and thus, $\tau_{\eta\epsilon} = \tau_{\eta\epsilon'}$), we have $\mathit{Valid}_{\mathsf{val}}(v, \tau_{\eta\epsilon}, h')$.

OSLetFun: Here $e = \mathsf{letfun\ f}(\mathsf{f}_1, \ldots, \mathsf{f}_{k'}, \mathsf{z}'_1, \ldots, \mathsf{z}'_k) = e_1\ \mathsf{in}\ e_2$, where $s;\ h;\ \mathcal{C}[\mathsf{f} := ((\mathsf{g}_1, \ldots, \mathsf{g}_{k'}, \mathsf{z}'_1, \ldots, \mathsf{z}'_k) \times e_1)] \vdash e_2 \rightsquigarrow v;\ h'$. From the LETFUN typing rule we have that $\Gamma \vdash_{\Sigma} e_2 \colon \tau$. Applying the induction hypothesis to these judgements with the same η and ϵ, we obtain $\mathit{Valid}_{\mathsf{val}}(v, \tau_{\eta\epsilon}, h')$ as desired.

OSFunApp: In this case $e = \mathsf{f}(\mathsf{f}_1, \ldots, \mathsf{f}_{k'}, \mathsf{z}'_1, \ldots, \mathsf{z}'_k)$, where $\mathcal{C}(\mathsf{f}) = (\mathsf{g}_1, \ldots, \mathsf{g}_{k'}, \mathsf{z}'_1, \ldots, \mathsf{z}'_k) \times e_1)$ and $[\mathsf{z}_1 := v_1, \ldots, \mathsf{z}_k := v_k];\ h;\ \mathcal{C} \vdash e_f[\mathsf{g}_1 := \mathsf{f}_1, \ldots, \mathsf{g}_{k'} := \mathsf{f}_{k'}] \rightsquigarrow v;\ h'$. We want to apply the induction hypotesis to this judgement.
Since all functions called in e are defined via letfun, there must be a node in the derivation tree of the original typing judgement of the form $\mathsf{True},\ y_1 : \tau^\circ, \ldots, y_k : \tau^\circ_k \vdash_{\Sigma} e_f : \tau_0$. Trivially, the domains of the frame store $[y_1 := v_1, \ldots, y_k := v_k]$ and the context $y_1 : \tau^\circ, \ldots, y_k : \tau^\circ_k$ coincide.
Take η' and ϵ' such that

– $\eta'(\alpha) = \eta(\tau_\alpha)$, where τ_α is such that α is replaced by τ_α in the instantiation σ of the signature in *this* application of the FunApp-rule.
– $\epsilon'(n_{ij}) = \epsilon(f_{ij})$, where n_{ij} is replaced by f_{ij} in the instantiation σ of the signature in *this* application of the FunApp-rule.

True holds trivially on ϵ'. From the induction hypothesis we have that if $\mathit{Valid}_{\mathsf{store}}((y_1, \ldots\ y_k), (y_1 : \tau^\circ_{1\,\eta'\epsilon'}, \ldots, y_k : \tau^\circ_{k\,\eta'\epsilon'}), [y_1 := v_1, \ldots, y_n := v_n], h)$ then $\mathit{Valid}_{\mathsf{val}}(v, \tau_{0\eta'\epsilon'}, h')$.
From $\mathit{Valid}_{\mathsf{store}}(\mathit{dom}(s), \Gamma_{\eta\epsilon}, s, h)$ we get the validity of the values of the actual parameters: $v_i \models^h_{\Gamma_{\eta\epsilon}(\mathsf{l}_i)} w_i$ for some w_i, with $1 \leq i \leq k$. Since $\Gamma_{\eta\epsilon}(\mathsf{l}_i) = \tau^\circ_{i\,\eta'\epsilon'}$, the left-hand side of the implication holds, and one obtains $\mathit{Valid}_{\mathsf{val}}(v, \tau_{0\ \eta'\epsilon'}, h')$. It is easy to see that

$$\sigma(\tau_0)) = \eta\epsilon(\tau_0[\ldots\alpha := \tau_\alpha\ldots][\ldots n_{ij} := f_{ij}\ldots]) =$$
$$\tau_0[\ldots\alpha := \eta(\tau_\alpha)\ldots][\ldots n_{ij} := \epsilon(f_{ij})\ldots] = \tau_{0\ \eta'\epsilon'}$$

Therefore, we obtain $\mathit{Valid}_{\mathsf{val}}(v, \sigma(\tau_{0\,\eta\epsilon}), h')$ and using the rule $D \vdash \tau \rightarrow \sigma(\tau_0)$ we obtain $\mathit{Valid}_{\mathsf{val}}(v, \tau_{\eta\epsilon}, h')$ by Lemma 2.

□

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 $n^s_l \in \overline{n}^*$ of a layer $s \geq 2$, let its range be given in the form $T(n^s_l) = \{p_l(\overline{n}, \overline{n}', \overline{i})\}_{Q_l(\overline{n},\overline{n}',\overline{i})}$, which is a short cut for $\{p_l(\overline{n}_0, \overline{n}'_0, \overline{i}) \mid \exists\ \overline{i}.\ Q_l(\overline{n}_0, \overline{n}'_0, \overline{i})\}$. 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 $\mathit{range}(n^s) \subseteq \{i\}_{n'_1 \leq i \leq n_2}$.

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 $\{p(\overline{n}, \overline{n}', \overline{i})\}_{Q(\overline{n},\overline{n}',\overline{i})}$ such that if for all higher-layer variables $\mathit{range}(n^s) \subseteq T(n^s)$ and $D_1(\overline{n}^*, \overline{m})$ holds then $f(\overline{n}^*) \subseteq \{p(\overline{n}, \overline{n}', \overline{i})\}_{Q(\overline{n},\overline{n}',\overline{i})}$.

Sometimes it is convenient to consider more specific estimates, where positions are mentioned explicitly. For instance, $\mathit{tails}_2(n)(\mathit{pos}) = n - \mathit{pos}$ for $n \geq 1$ and $0 \leq \mathit{pos} \leq n - 1$. Such position-aware estimates may be used to obtain tight position-free bounds on the overall size of the output structure. This is done by summations over positions. In the example above we have that the overall length of the internal lists is $\Sigma_{pos=0}^{n-1} \mathit{tails}_2(n - \mathit{pos}) = \Sigma_{pos=0}^{n-1}(n - \mathit{pos}) = \Sigma_{l=1}^{n} l = \dfrac{n(1+n)}{2}$. This is definitely more precise than the position-free estimate $\mathit{tails}_2(n) \subseteq \{i\}_{0 \leq i \leq n}$. In general, position-aware estimates for bound of internal lists have the form "$\mathit{range}(n^s) \subseteq T(n^s) \wedge D(\overline{n}, \overline{n}', \overline{pos})$ implies $f(\overline{n}^*)(\overline{pos}) \subseteq \{p(\overline{n}, \overline{n}', \overline{pos}, \overline{i})\}_{Q(\overline{n},\overline{n}',\overline{pos},\overline{i}) \wedge D(\overline{n},\overline{n}',\overline{pos})}$".

compositions!

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 $T(n^s_l)$ are finite. For instance, for $n' = 3$ the *range* of M is included into the set $\big(T(M) = \{i\}_{0\le i\le n'}\big)_{n':=3} = \{0,1,2,3\}$. 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 $\langle\mu_k, \ldots, \mu_{d-1}\rangle$.

Symmetrically, to mirror constructor application we could have used concatenation operation on finite multivalued maps. However, we do not use concatenation here. The reason is that this operation is defined explicitly via the rewriting rules in the antecedent of the Cons-rule, so it is not a part of the syntax of size annotations. Therefore, here it will be not a part of the syntax of expressions over finite sets and multivalued maps, but will be defined within rewriting systems for functions over finite sets and multivalued maps, when necessary.

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. We define the translation $\llcorner - \lrcorner : \mathit{SizeExpressions} \to \mathit{FiniteMMapSetExpressions}$ as follows:

- first-layer constants represents themselves: $\llcorner a \lrcorner := a$;
- higher-layer constants, $s \ge 2$ are translated into their restrictions: $\llcorner a^s \lrcorner := a'^s$; since we fix the sizes of lists, then e.g. for $s = 2$ the map a'^2 represents the restriction of the map a^2 to the set $\{0, \ldots, \max\big(p^1(\overline{n}^*)\big) - 1\}$, where the expression p^1 is given by the type $\mathsf{L}_{p^1(\overline{n}*)}(\mathsf{L}_{a^2}(\ldots))$;

for instance, $\llcorner a^2 \lrcorner = a^2$, where $a'^2(0) = 0$ and $a'^2(1) = \{0, 1\}$ and a^2 is taken from the type $\mathsf{L}_{n+2}(\mathsf{L}_{a^2}(\ldots))$ with $n := 0$;
- positions pos and first-layer variables n are translated to themselves: $\llcorner pos \lrcorner := pos$ and $\llcorner n \lrcorner := n$; they represent the corresponding singleton sets;
- for a higher-layer variable n^s from the set of parameters $\overline{n}^*$; where $s \geq 2$, we introduce a fresh variable ϕ: $\llcorner n^s \lrcorner := \phi$;
- translation $\llcorner p_1 \circledast p_2 \lrcorner := \llcorner p_1 \lrcorner \circledast_{\{\}} \llcorner p_2 \lrcorner$ is defined on first-layer size expressions;
- $\llcorner p_{+\mathbf{1}} \lrcorner := \llcorner p \lrcorner_{+\mathbf{1}}$, where p' is a first-layer expression with no free occurrences of pos;
- $\llcorner (p)(pos) \lrcorner := \llcorner p \lrcorner (pos)$,
- $\llcorner g(p_1, \ldots, p_k) \lrcorner := \llcorner g \lrcorner (\llcorner p_1 \lrcorner, \ldots, \llcorner p_k \lrcorner)$.

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 $\llcorner f \lrcorner(\llcorner n_1^{s_1} \lrcorner, \ldots \llcorner n_k^{s_k} \lrcorner)(pos_1 \ldots pos_{s-1}) \to \llcorner p \lrcorner$. For instance, the rewriting rule $n \geq 1, 0 \leq pos \leq n-1 \vdash \mathit{tails}_2(n)(pos) \to \mathit{tails}_2(n-1)(pos-1)$ is translated to $n \geq 1, 0 \leq pos \leq n-1 \vdash \llcorner \mathit{tails} \lrcorner_2(n)(pos) \to \llcorner \mathit{tails} \lrcorner_2(n-1)(pos-1)$.

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 $(\overline{n}_0, \overline{n}'_0, \overline{pos}_0)$, for which there exists $\overline{m}$ such that $D_1(\overline{n}^*, \overline{m}) \wedge D_1(\overline{m}, \overline{pos})$ holds, 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 $d = 2$ for $\mathit{concat}_{l,\,u}(n, n')$ we take the finite set of test points (n_0, n'_0) $\{(1,1)$ as $(2,1)$, $(3,1)$, $(1,2)$, $(2,2)$, $(1,3)\}$. 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 $\phi_l := \lambda\ pos'.\{p_l(\overline{n}_0, \overline{n}'_0, \bar{i})\}_{Q_l(\overline{n}_0,\overline{n}'_0,\bar{i})}$, which is a constant multivalued map; e.g. $\phi := \langle\{0,1\}\rangle$ for M in $\mathit{concat}(n, M)$ with $n = 1,\ n' = 1.$; for instance, for $\llcorner \mathit{concat} \lrcorner$, with $n = 2$ and $n' = 3$ we have $n_{(2,3)} = \{n\} = \{2\}$ and $\phi_{(2,3)} = \langle\{0,1,2,3\},\ \{0,1,2,3\}\rangle$;
   (b) compute $\llcorner f \lrcorner(\overline{n}, \overline{\phi})(pos)$ using the rewriting rules; e.g.

$$\begin{aligned}
\llcorner \mathit{concat} \lrcorner(n_{(2,3)}, \phi_{(2,3)}) \to\ & \phi_{(2,3)}(0) +_{\{\}} \llcorner \mathit{concat} \lrcorner(n_{(2,3)} - 1, \phi_{(2,3)\ +1}) = \\
& \{0,1,2,3\} +_{\{\}} \llcorner \mathit{concat} \lrcorner(2-1, \langle\{0,1,2,3\},\ \{0,1,2,3\}\rangle_{+1}) = \\
& \{0,1,2,3\} +_{\{\}} \llcorner \mathit{concat} \lrcorner(1, \langle\{0,1,2,3\}\rangle) \to \\
& \{0,1,2,3\} +_{\{\}} \{0,1,2,3\} +_{\{\}} \llcorner \mathit{concat} \lrcorner(1-1, \langle\{0,1,2,3\}\rangle_{+1}) = \\
& \{0,1,\ldots,6\} +_{\{\}} \llcorner \mathit{concat} \lrcorner(0, \langle\rangle) \to \\
& \{0,1,\ldots,6\} +_{\{\}} \{0\} = \{0,1,\ldots,6\}
\end{aligned}$$

   yet another example is $\llcorner \mathit{tails} \lrcorner_2(2)(1) \to \llcorner \mathit{tails} \lrcorner_2(2-1)(1-1) = \llcorner \mathit{tails} \lrcorner_2(1)(0) \to 1$.

(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. $\mathit{concat}_{\min}(1,3) := 0$ and $\mathit{concat}_{\max}(1,3) := 3$; also $\mathit{tails}_{2,\min}(2,1) = \mathit{tails}_{2,\max}(2,1) = 1$.

(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, solving the system for $\mathit{concat}_l(n,n')$ and $\mathit{concat}_u(n,n')$ gives $\mathit{concat}_l(n,n') = 0$ and $\mathit{concat}_u(n,n') = nn'$; for tails_2 we obtain $\mathit{tails}_{2\,l}(n,pos) = \mathit{tails}_{2\,u}(n,pos) = n - pos$. 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 $\{i\}_{0 \le i \le nn'}$.
   If f is of the layer $s \ge 2$, 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 $0 \le i \le f_u(\overline{n}, \overline{n}', \overline{j}) - f_l(\overline{n}, \overline{n}', \overline{j}) \wedge D_2(\overline{m}, \overline{j})$. Note that $D_2(\overline{m}, \overline{j})$ consists of disequations of the form $0 \le j \le m-1$ or $1 \le j \le m$. Replace m that belongs to the set $p(\overline{n}^*)(pos_1)\ldots(pos_{s-1})$ with the already derived upper bound for this set. For instance, for $\llcorner \mathit{tails} \lrcorner_2(n)$ we obtain $\{n-j\}_{1 \le j \le n-1}$ on $n \ge 1$. The family is completed to $\{n-j\}_{0 \le j \le n-1}$ by $\llcorner \mathit{tails} \lrcorner_2(n)(0) = n$.
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 4.2.
   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: $D \vdash T(f(\overline{n}^*)) \supseteq \mathit{range}(p)$, 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 $n_g^s \in \overline{n}_g^*$, where $s \geq 2$, then $T(n_g^s)$ is given in the form $\{p(\overline{n}_g, \overline{n}'_g, \bar{i})\}_{Q(\overline{n}_g, \overline{n}'_g, \bar{i})}$, where $\overline{n}'_g$ are fresh first-layer size variables, and a polynomial $p(\overline{n}_g, \overline{n}'_g, \bar{i})$ and a predicate $Q(\overline{n}_g, \overline{n}'_g, \bar{i})$ 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 $\{p(\overline{n}_g, \overline{n}'_g, \bar{i})\}_{Q(\overline{n}_g, \overline{n}'_g, \bar{i})}$. Note, that we treat higher-layer constants as functions, that is their specifications must be present in the table as well, in the form $T(a) = \{p(\bar{i})\}_{Q(\bar{i})}$. 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 $\{i\}_{0 \leq i \leq nn'}$ for *concat*, contains $T(+\!\!+, n_1, n_2) = n_1 + n_2$, $T(M) = \{i\}_{0 \leq i \leq n'}$, $T(\textit{concat}, n, n') = \{i\}_{0 \leq i \leq nn'}$.

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 $0 \leq \mathit{pos} \leq m-1$ (from the definition of type rewriting) or $1 \leq \mathit{pos} \leq m$ (a side condition of the cons-rule) and disequations $m \geq 1$ (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 $\bigwedge_{n^s \in \overline{n}^*,\ s \geq 2} n^s(\overline{\mathit{pos}}) \subseteq T(n^s)$ 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 $s \geq 2$;
– 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 $s \geq 2$, the range map is defined by the spec. table, $(\!|n^s|\!) := T(n^s)$;
– if $\circledast$ is one of the arithmetic operations $+, -, *$, then
$(\!|p_1 \circledast p_2|\!) := (\!|p_1|\!) \circledast_{\{\}} (\!|p_2|\!)$;
– $(\!|p(0)|\!) := (\!|p|\!)$;

– $(\!|p(pos)|\!) := (\!|p|\!)$;
– $(\!|p(pos-1)|\!) := (\!|p|\!)$;
– $(\!|p_{+\mathbf{1}}|\!) := (\!|p|\!)$;
– 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 $1 \le l \le k$;
2. after that we (inductively) compute the range sets $(\!|p'_l|\!)$ of the higher-layer actual parameters p'_l, where $1 \le l \le k'$;
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:

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

For the sake of convenience we denote the last set via $(\!|p'_l|\!)_\sigma$.
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 $n \geq 1 \vdash \mathit{concat}(n, M) \rightarrow M(0) + \mathit{concat}(n-1, M_{+\mathbf{1}})$.

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

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 $\mathit{concat} \circ \mathit{tails}(n) \subseteq \{i\}_{0 \leq i \leq n^2}$. We have $\mathit{concat} \circ \mathit{tails}(n) \rightarrow \mathit{concat}(n, \mathit{tails}_2(n))$. We already know that $T(\mathit{concat}(n, M)) = \{i\}_{0 \leq i \leq nn'}$ for $T(M) = \{i\}_{0 \leq i \leq n'}$. Now we need to match $T(M)$ with the annotation of the actual parameter $\mathsf{L}_{\mathit{tails}_2(n)}(\alpha)$. We know that $T(\mathit{tails}_2(n)) = \{i\}_{0 \leq i \leq n}$, so we assume $\sigma(n') = \{n\}$. Indeed, $(\!|\mathit{tails}_2(n)|\!) = \{i\}_{0 \leq i \leq n} \subseteq \sigma(T(M)) = \{i\}_{0 \leq i \leq \sigma(n')}$, thus σ is a valid substitution.

Lemma 3 (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 $\bigwedge_{n^s \in \overline{n}^*,\ s \geq 2} n^s(\overline{pos}) \subseteq T(n^s)$ 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 $p_{\overline{n}^*, \overline{pos}}$.

- The statement of the lemma for the base cases (constants and variables) follows directly from the definition of $(\!|-|\!)$ and the validity of the specifications for the higher-layer variables and constants.
- Let $p_{\overline{n}^*, \overline{pos}} \equiv p_{1\ \overline{n}^*, \overline{pos}} \circledast p_{1\ \overline{n}^*, \overline{pos}}$. By induction assumption, $p_{l\ \overline{n}^*, \overline{pos}}(\overline{n}^*, \overline{pos}) \subseteq (\!|p_{l\ \overline{n}^*, \overline{pos}}|\!)$, where $l = 1, 2$. From the definition

$$p_{\overline{n}^*, \overline{pos}}(\overline{n}^*, \overline{pos}) := p_{1\ \overline{n}^*, \overline{pos}}(\overline{n}^*, \overline{pos}) \circledast_{\{\}} p_{2\ \overline{n}^*, \overline{pos}}(\overline{n}^*, \overline{pos})$$

  it follows that $p_{\overline{n}^*, \overline{pos}}(\overline{n}^*, \overline{pos}) \subseteq (\!|p_{1\ \overline{n}^*, \overline{pos}}|\!) \circledast_{\{\}} (\!|p_{2\ \overline{n}^*, \overline{pos}}|\!) := (\!|p_{\overline{n}^*, \overline{pos}}|\!)$.
- Let $p_{\overline{n}^*, \overline{pos}} \equiv p'_{\overline{n}^*, pos_0,\ \overline{pos}}(0)$. By induction assumption, $p'_{\overline{n}^*, pos_0, \overline{pos}}(\overline{n}^*,\ 0,\ \overline{pos}) \subseteq (\!|p'_{\overline{n}^*, pos_0, \overline{pos}}|\!)$. Therefore,

$$
\begin{array}{rl}
p_{\overline{n}^*, \overline{pos}}(\overline{n}^*, \overline{pos}) = p'_{\overline{n}^*, pos_0, \overline{pos}}(\overline{n}^*,\ 0,\ \overline{pos}) \subseteq & \\
(\!|p'_{\overline{n}^*, pos_0, \overline{pos}}|\!) =^{\text{def. of } (\!|-|\!)} & \\
(\!|p'_{\overline{n}^*, pos_0, \overline{pos}}(0)|\!) =^{\text{sructure of } p} & \\
(\!|p_{\overline{n}^*, \overline{pos}}|\!) &
\end{array}
$$

- The other cases, where p is an application of another size expression to a position, are treated similarly.
- Let $p_{\overline{n}^*, \overline{pos}} \equiv [p'_{\overline{n}^*, pos, \overline{pos}'}]_{+\mathbf{1}}$ for some p', where $\overline{pos} = (pos, \overline{pos}')$. Therefore, $p_{\overline{n}^*, \overline{pos}}(\overline{n}^*, \overline{pos}) = p'_{\overline{n}^*, pos, \overline{pos}'}(\overline{n}^*, pos + 1, \overline{pos}')$ According to the induction assumption, $p'_{\overline{n}^*, pos, \overline{pos}'}(\overline{n}^*, pos + 1, \overline{pos}') \subseteq (\!|p'_{\overline{n}^*, pos, \overline{pos}'}|\!)$. According to the definition of $(\!|-|\!)$, the last set is equal to $(\!|[p'_{\overline{n}^*, pos, \overline{pos}'}]_{+\mathbf{1}}|\!)$, which is exactly $(\!|p'_{\overline{n}^*, pos, \overline{pos}'}|\!)$.

– Consider the function call

$$[g(p_1^1,\ldots,p_k^1,\ p'_1,\ldots,p'_{k'})](\overline{n},\overline{pos}) :=$$
$$g\big(p_1^1(\overline{n},\overline{pos}),\ldots,p_k^1(\overline{n},\overline{pos}),\ p'_1(\overline{n},\overline{pos}),\ldots,p'_{k'}(\overline{n},\overline{pos})\big)$$

According to the actual-parameter listing, a formal parameter n_l^s of g is instantiated with the actual parameter expressed by $p'_{l\,\overline{n^*},\overline{pos}}$. The similar holds for the first-layer formal and corresponding actual parameters. Now we want to apply the validity of $T(g(\overline{n}_g^*))$. Instantiate $\overline{n}'_{g\ l}$ with $\sigma(\overline{n}'_{g\ l})$ from the definition of $(\!|-|\!)$ for the function call under consideration. Further, according to the induction assumption for the actual parameters $p'_{l\,\overline{n^*},\overline{pos}}(\overline{n^*},\overline{pos})$ and the definition of σ we obtain

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

This is exactly means, that the actual parameters satisfy the specifications for the corresponding higher-layer variables of g. Therefore, we are allowed to apply the validity of $T(g)$ and obtain:

$$\begin{array}{l} [g(p_1^1,\ldots,p_k^1,\ p'_1,\ldots,p'_{k'})](\overline{n},\overline{pos}) \subseteq \\ \{p((\!|p_1^1|\!)_1,\ldots,(\!|p_k^1|\!)_1,\sigma(\overline{n}'_g)_1,\overline{j})\}_{Q(p_1^1,\ldots,p_k^1,\sigma(\overline{n}'_g)_1,\overline{j})\wedge\sigma(\overline{n}'_g)_2\wedge\bigwedge_{l=1}^{k}p_l^1|\!)_1)} \end{array}$$

where the last set is exactly $(\!|g(p_1^1,\ldots,p_k^1,\ p'_1,\ldots,p'_{k'})|\!)$ according to the definition of $(\!|-|\!)$.

Given a collection of a right-hand side conditions D or its instances by actual parameters, let $(\!|D|\!)$ denote the result of substituting of size expressions p, which occurs in D, for the corresponding sests $(\!|p|\!)$.

Lemma 4 (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 $\bigwedge_{n^s\in\overline{n^*},\ s\geq 2} n^s(\overline{pos}) \subseteq T(n^s)$ 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 by induction on the deepness of the recursion in the calls of f occuring in $p_{\overline{n^*},\overline{pos}}(\overline{n^*},\overline{pos})$.

– If the depness $d = 0$, then f does not occur in p. Hence, we apply Lemma 3 directly.
– Let the deepness $d \geq 1$. Run the inductive proof on the structure of p.
  • If p is NOT a call of f, then the proof schema is the same as for the corresponding clause of Lemma 3.

- Consider a function call

$$[f(p_1^1,\ldots,p_k^1,\ p'_1,\ldots,p'_{k'})](\overline{n},\overline{pos}) :=$$
$$f(p_1^1(\overline{n},\overline{pos}),\ldots,p_k^1(\overline{n},\overline{pos}),\ p'_1(\overline{n},\overline{pos}),\ldots,p'_{k'}(\overline{n},\overline{pos}))$$

Since this call terminates, there must be a rule $D \vdash T(f(\overline{n}_f^*)) \to p_f$ applicable for the actual parameters of the call. According to the actual-parameter listing, a formal parameter n_l^s is instantiated with the actual parameter expressed by $p'_{l\overline{n}^*,\overline{pos}}$. The similar holds for the first-layer formal and corresponding actual parameters and the corresponding instance of D should hold, allowing us to use the rewriting rule. Applying the rewriting rule we obtain

$$[f(p_1,\ldots,p'_{k'})](\overline{n}^*,\overline{pos}) :=$$
$$f(p_1(\overline{n}^*,\overline{pos}),\ldots,p'_{k'}(\overline{n}^*,\overline{pos})) =$$
$$p_f(p_1(\overline{n}^*,\overline{pos}),\ldots,p'_{k'}(\overline{n}^*,\overline{pos}))$$

We may apply induction-on-the-deepness assumption, since the deepness of the recursive calls of f in p_f is one less then in p. Therefore, $p_f(p_1(\overline{n}^*,\overline{pos}),\ldots,p'_{k'}(\overline{n}^*,\overline{pos})) \subseteq (\!|p_f(p_1,\ldots,p'_{k'})|\!)$.
Now, as we have pointed out above, D implies $(\!|D|\!)$. Therefore we may apply the inclusion $(\!|D|\!) \vdash T(f(\overline{n}^*)) \supseteq p_f$, more precisely, its instantiation with the first-layer actual parameters and $\sigma(\overline{n}'_f)$ for the fresh size variables, taken from the definition of $(\!|-|\!)$ for unction calls. Thus, we obtain that

$$p_f(p_1(\overline{n}^*,\overline{pos}),\ldots,p'_{k'}(\overline{n}^*,\overline{pos})) \subseteq$$
$$(\!|p_f(p_1,\ldots,p'_{k'})|\!) \subseteq$$
$$(\!|p_f(p_1,\ldots,p'_{k'})|\!)_\sigma \subseteq T(f(\overline{n}_f^*))_\sigma =^{\text{definition}} (\!|p_{\overline{n}^*,\ \overline{pos}}|\!)$$

Theorem 2 (Checking). *If all called in the definition of f functions $g \neq f$ have valid specifications $T\big(g(\overline{n}_g^*)\big)$, and for each rule $D \vdash f(\overline{n}*)(pos_1)\ldots(pos_{s-1}) \to p$ the inclusion $(\!|D|\!) \vdash T(f(\overline{n}*)) \supseteq (\!|p|\!)$ 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, $D \vdash f(\overline{n}^*)(\overline{pos}) \to p$. 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 4 we obtain $f(\overline{n}*)(\overline{pos}) \subseteq (\!|p|\!)$. 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] 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 *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. To our knowledge, non-monotonic quasi-interpretations have not been studied for size analysis, but only for proving termination [10]. In this work one considers some unspecified algorithmically decidable classes of non-negative and negative polynomials and introduces abstract variables for the rest.

Hoffman and Jost have presented a heap space analysis [11] 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 [12], although without an inference proceudure. Brian Campbell extended this approach to infer bounds on *stack* space usage in terms of the total size of the input [7], and recently as max-plus expressions on the depth of data structures [8]. Again, the main difference with our work is that we not require linear size functions.

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 [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 [9, 5]. 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. In [4] the authors introduce resource policies for mobile code to be run on smart devices and certify resource bounds in a Proof Carrying Code system.

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, 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 16^{th} *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*, 2009, to appear.
7. B. Campbell. *Space Cost Analysis Using Sized Types*. PhD thesis, School of Informatics, University of Edinburgh, 2008.
8. 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.
9. 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.
10. N. Hirokawa and A. Middeldorp. Polynomial interpretations with negative coefficients. In *Artificial Intelligence and Symbolic Comp.*, volume 3249 of *LNCS*, 2004.
11. M. Hofmann and S. Jost. Static prediction of heap space usage for first-order functional programs. *SIGPLAN Not.*, 38(1):185–197, 2003.
12. 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.
13. O. Shkaravska, M. van Eekelen, and A. Tamalet. Collected Size Semantics for Functional Programs. In S.-B. Scholz, editor, *Implementation and Application of Functional Languages:* 20^{th} *International Workshop, IFL 2008, Hertfordshire, UK, 2008. Revised Papers*, LNCS. Springer-Verlag, 2008. to appear.
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.