The following full text is a publisher's version.

For additional information about this publication click this link:
http://hdl.handle.net/2066/75339

Please be advised that this information was generated on 2017-09-01 and may be subject to change.
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 consumption of programs can be used, and are often needed, to ensure correctness and security properties, in particular in devices with scarce resources as mobile phones and smart cards. Both the memory and the time consumption of a program often depend on the sizes of input and intermediate data. 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 $\text{append}(n, m) = n + m$.

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 insert : $(\alpha \times \alpha \rightarrow \text{Bool}) \times \alpha \times \text{Ln}(\alpha) \rightarrow \text{Ln}(\text{insert}(\alpha)(\alpha))$ that inserts an element $z$ of the type $\alpha$ in a list $l$, if this list does not contain an element $z'$ such that the relation $g(z, z')$ holds:

\[
\text{insert}(g, z, l) = \text{match } l \text{ with } | \text{Nil } \Rightarrow \text{Cons}(z, \text{Nil}) \]
\[
| \text{Cons}(\text{hd}, \text{tl}) \Rightarrow \text{if } g(z, \text{hd}) \text{ then } l \text{ else Cons}(\text{hd}, \text{insert}(g, z, \text{tl}))
\]

* This work is part of the AHA project [16] which is sponsored by the Netherlands Organisation for Scientific Research (NWO) under grant nr. 612.003.511.
Its size dependency \( \text{insert}(n) \) represents the length of the output corresponding to an input of the length \( n \). It is given by (where \( | \) separates alternative rewriting rules):

\[
\begin{align*}
\vdash \text{insert}(0) & \rightarrow 1 \\
n \geq 1 \vdash \text{insert}(n) & \rightarrow n \mid \text{insert}(n - 1) + 1
\end{align*}
\]

However, the type system from [13] only covers programs over ‘matrix-like’ structures, e.g. \( L_n(L_m(\alpha)) \) 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 \( L_n(L_M(\alpha)) \) represents the size \( M(pos) \) of an internal list depending on its position \( pos \) in the outer list, where \( 0 \leq pos \leq n - 1 \). Moreover, we extend (checking and inferring of) multivalued size functions allowing them to be defined with higher-order rewriting rules. We define soundness and sketch its proof. Section 4 gives a procedure for the generation of polynomial lower and upper bounds and a set of polynomials that covers the function defined by the higher-order rewriting rules. Section 5 relates our work to other resource analysis work.

**Informal Sketch of the Approach** Consider the function \( \text{concat} \), which given a list of lists appends all the inner lists:

\[
\text{concat}(l) = \text{match } l \text{ with } | \text{Nil }^\rightarrow \text{Nil} \\
| \text{Cons}(\text{hd}, \text{tl}) \Rightarrow \text{append(\text{hd}, \text{concat(\text{tl})})}
\]

The expected annotated type for \( \text{concat} \) is \( L_n(L_M(\alpha)) \rightarrow L_{\text{concat}(n,M)}(\alpha) \) where \( n \) and \( M \) are size variables of types \( N \) (naturals) and \( N \rightarrow N \), respectively and its size function \( \text{concat}(n, M) \) is defined by the following rewriting rules

\[
\begin{align*}
\vdash \text{concat}(0, M) & \rightarrow 0 \\
n \geq 1 \vdash \text{concat}(n, M) & \rightarrow M(0) + \text{concat}(n - 1, \lambda pos. M(pos + 1))
\end{align*}
\]

where the first argument, \( n \), of \( \text{concat} \) 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: \( \text{concat } [ [1, 2, 3], [4, 5] ] \). 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 \( M_0(pos) \) for \( pos \geq 2 \) is arbitrary. We are now interested in calculating the result size defined by \( \text{concat}(2, M_0) \):

\[
\begin{align*}
\text{concat}(2, M_0) & \rightarrow M_0(0) + \text{concat}(1, \lambda pos. M_0(pos + 1)) \\
& \rightarrow 3 + (\lambda pos. M_0(pos + 1))(0) + \\
& \text{concat}(1 - 1, \lambda pos'. ((\lambda pos. M_0(pos + 1))(pos' + 1))) \\
& = 3 + M_0(0 + 1) + \text{concat}(0, \lambda pos'. M_0((pos' + 1) + 1)) \\
& = 3 + M_0(1) + \text{concat}(0, \lambda pos'. M_0(pos' + 2)) \\
& = 3 + M_0(1) + 0 = 3 + 2 + 0 = 5
\end{align*}
\]
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 \( \text{concat}(n, M) \), of higher-order kinds, with finite multivalued maps. Consider how to infer size bounds for \( \text{concat} \). Assume that the size of inner lists is at most \( n' \). Then, the expected inferred result is \( \{ j \} \) covering the range of \( \text{concat}(n, M) \).

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

From this point of view, \( \text{concat} \) in the “finite world” is presented by a function \( \text{concat_*} \) over finite sets and finite multivalued maps. The rewriting rules for \( \text{concat_*} \) are obtained by the obvious translation of the rewriting rules for \( \text{concat} \).

\[
\begin{align*}
\text{concat} \; (0, \phi) & \to 0 \\
n \geq 1 \quad \text{concat} \; (n, \phi) & \to \phi(0) + \text{concat} \; (n - 1, \phi) + \phi \text{concat} \; (n - 1, \phi + 1)
\end{align*}
\]

where \( \phi + := \lambda \text{pos}. \phi(\text{pos} + 1) \) is the 1-position-left shift for the finite sequence of sets presenting \( \phi \), 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 \( \text{concat}_l \) and an upper bound \( \text{concat}_u \) such that the family \( \{ \text{concat} + j \} \) approximates \( \text{concat} \). 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 $\text{concat}_t$ and $\text{concat}_w$ are polynomials with degree $d = 2$. A bound on the size of an output for $\text{concat}$ depends on two parameters, $n$ and $n'$. So, an upper bound is a polynomial of degree two of two variables: $\text{concat}_a(n, n') = \gamma_2n^2 + \gamma_1nn' + \gamma_0(n')^2 + \gamma_10n + \gamma_01n' + \gamma_00$. Hence, to find $\text{concat}_t$, one must know its value in 6 points. The same holds for $\text{concat}_w$. We evaluate the rewriting rules for $\text{concat}_t$ in 6 points. Let’s start with $n = n' = 1$. Then $\phi$ is instantiated to $\langle\{0, 1\}\rangle$.

\[
\mathcal{L}_{\text{concat}_t}(1, 1) \rightarrow \langle\{0, 1\}\rangle(0) + \langle\{0, 1\}\rangle(1, \langle\{0, 1\}\rangle + 1) = \langle\{0, 1\}\rangle + \langle\{0, 1\}\rangle + \langle\{0, 1\}\rangle + \langle\{0, 1\}\rangle + \langle\{0, 1\}\rangle + \langle\{0, 1\}\rangle.$
\]

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

\[
\begin{align*}
\gamma_20 + \gamma_10 + \gamma_00 &= 1 \\
\gamma_20 + 2\gamma_11 + \gamma_01 + \gamma_00 &= 3
\end{align*}
\]

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

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 $\exists$, and $+$ for $+$:

\[
\begin{align*}
n &= 0 \rightarrow \{j'\}_{0 < j' \leq nn'} \rightarrow \{0\} \\
n &\geq 1 \rightarrow \{j'\}_{0 < j' \leq nn'} \rightarrow \{j\} \rightarrow \{j'\}_{0 < j' \leq nn'}(n-1)n
\end{align*}
\]

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

\[
\forall n, \quad n = 0 \rightarrow \exists j', \ j' = 0 \wedge 0 < j' \leq nn' \quad \forall j, \ j' \geq 1 \wedge 0 < j' \leq nn' \rightarrow \exists j'', j'' = j + j' \wedge 0 < j'' \leq nn'
\]

So, the inferred bounds of $\text{concat}$ 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 $\text{False}$ and $\text{True}$, $x$ 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, $\text{unop}$ is a unary operation, either $-$ or $\neg$, $\text{binop}$ is one of the integer or boolean binary operations, and $f$ denotes a function name.

\[
\begin{align*}
\text{Basic} &::= c | \text{unop} x | x \text{ binop} y | \text{Nil} | \text{Cons}(z, l) | f(g_1, ... , g_i, z_1, ... , z_k) \\
\text{Expr} &::= b | \text{if} x \text{ then } e_1 \text{ else } e_2 \mid \text{let } z = b \text{ in } e_1 \mid \text{match } l \text{ with } \mid \text{Cons}(z_h, l_0) \Rightarrow e_2 \mid \text{letfun } f(g_1, ... , g_i, z_1, ... , z_k) = e_1 \text{ in } e_2
\end{align*}
\]
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 \( \alpha \) where \( \alpha \) 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 \( L_5(\alpha) \) with \( a = 5 \), or \( L_n(L_5(\alpha)) \).
- The second layer consists of numerical first-order constants and variables of type \( \mathcal{R}^{(2)} = \mathcal{R} \times \mathcal{R} \), denoted by \( B \) and \( M \), respectively. They represent lengths of nested lists in a list. For instance, in the typing \( l : L_n(L_M(\alpha)) \) the function \( \lambda \text{pos}.M(\text{pos}) \) represents the length of the \( \text{pos}-\text{th} \) 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 \( \mathcal{R} \rightarrow \mathcal{R} \) may be defined by an arithmetic expression or by a table. For instance, in \[ \begin{bmatrix} 1, 2 \end{bmatrix}, \begin{bmatrix} 3, 4, 5 \end{bmatrix}, \begin{bmatrix} \end{bmatrix} \] the length of the master list is \( a = 3 \) and \( B \) is given by the table \( B(0) = 2, B(1) = 3, B(2) = 0 \).
- In general, the \( s \)-th layer consists of numerical \((s-1)\)-th-order constants and variables of type \( \mathcal{R}^{(s)} = \mathcal{R} \rightarrow \mathcal{R}^{(s-1)} \), denoted by \( a^s \) and \( n^s \). They represent lengths of lists of “nestedness” \( s \). For instance in \( l : L_{n_1}(\ldots L_{n_2}(\alpha)\ldots) \) 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 \( n^s \) range over size variables of \( \mathcal{R}^* \).

Let \( \mathbb{n}^s \) denote a vector of variables \((n^s_1, \ldots, n^s_k)\) 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 \( f : (\mathcal{R}^*)^k \rightarrow 2^\mathcal{R} \) 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 \( l : L_{f(n)}(\alpha) \), then the length of \( l \) is either \( n \) or \( n + 1 \). This annotation is given in the output type of the function \( \text{insert} : (\alpha \times \alpha \rightarrow \text{Bool}) \times \alpha \times L_n(\alpha) \rightarrow L_{\text{insert}(n)}(\alpha) \). The function \( \text{insert} \), given a predicate \( g : \alpha \times \alpha \rightarrow \text{Bool} \), an element \( z : \alpha \) and a list \( l : L_n(\alpha) \), 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 \( \text{concat} : L_n(L_m(\alpha)) \rightarrow L_{\text{concat}(n, M)}(\alpha) \), we have a function \( \text{concat} : \mathcal{R}^{(1)} \times \mathcal{R}^{(2)} \rightarrow 2^R \). Here \( \text{concat}(0, M) = 0 \) and \( \text{concat}(n, M) = M(0) + M(n - 1, \lambda \text{pos} . M(\text{pos} + 1)) \) for \( n \geq 1 \).

- A function of the layer \( s \) is a function of the type \( (\mathcal{R}^s)^k \rightarrow (\mathcal{R} \rightarrow \ldots \rightarrow \mathcal{R} \rightarrow 2^R) \) that maps parameters from \( (\mathcal{R}^s)^k \) to \( s - 1 \)-order multivalued functions of the type \( \mathcal{R} \rightarrow \ldots \rightarrow \mathcal{R} \rightarrow 2^R \). Its value \( f(\mathcal{R}^s)(\text{pos}_1) \ldots (\text{pos}_{s-1}) \) defines all possible sizes of the \( \text{pos}_{s-1} \) list in the \( \text{pos}_{s-2} \)-th list ... in the \( \text{pos}_1 \)-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 \( \text{tails} : L_n(\alpha) \rightarrow L_{\text{tails}_1(n)}(L_{\text{tails}_2(n)}(\alpha)) \) that creates the list of all non-empty tails of the input list:

\[
\text{tails}(l) = \begin{cases} \text{Nil} \Rightarrow \text{Nil} \\ \text{Cons}(\text{hd}, \text{tl}) \Rightarrow \text{let} \ l' = \text{tails}(\text{tl}) \text{ in} \ \text{Cons}(l, l') \\
\end{cases}
\]

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

\[
\begin{align*}
\text{tails}_2(n)(0) &= n \\
\text{tails}_2(n)(1) &= n - 1, \text{ if } n \geq 1 \\
\text{tails}_2(n)(\text{pos}) &= n - \text{pos}, \text{ if } 0 \leq \text{pos} \leq n - 1 \\
\text{tails}_2(n)(\text{pos}) &= \text{arbitrary if } \text{pos} > n
\end{align*}
\]

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 \(+, -\), \(*\), \(\lambda\)-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 \( \mathcal{R} \rightarrow \ldots \rightarrow \mathcal{R} \rightarrow 2^R \).

When necessary, we denote a size expression of the layer \( s \) via \( p^s \).

\[
P^1 ::= a \mid n, m \mid f(p_1, \ldots, p_k) \mid p^s(\text{pos}) \mid p^s(\text{pos} - 1) \mid p^s(0) \mid p^s(\{+, -, \ast\}p)
\]

\[
P^2 ::= B \mid M \mid g(p_1, \ldots, p_k) \mid p^s(\text{pos}) \mid p^s(\text{pos} - 1) \mid p^s(0) \mid p^s(\text{pos}) \mid p^s(\{+, -, \ast\}p)
\]

\[
P^{s+1} ::= a^s \mid n^s \mid f^s(p_1, \ldots, p_k) \mid p^{s+1}(\text{pos}) \mid p^{s+1}(\text{pos} - 1) \mid p^{s+1}(0) \mid p^{s+1}_1
\]

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

\[
\tau^0 ::= \text{int} \mid \text{Bool} \mid \alpha
\]

\[
\tau^{s', s} ::= L_{\tau^{s'}}(L_{\tau^{s'+1}}(\ldots L_{\tau^s}(\tau^0) \ldots)) \text{ for } 1 \leq s' \leq s,
\]

\[
\tau^s ::= \tau^{s, s}
\]
where $\alpha$ is a type variable. It is easy to see that $\tau^s, s = L_p s^s + 1, s$. The types $\tau^0$ and $\tau^a$ are types of program expressions, but $\tau^s, s$ are only used in definitions and proofs but not in function types.

Let $\tau$ ranges over zero-order types. The sets $TV(\tau)$ and $SV(\tau)$ of type and size variables of a type $\tau$ are defined inductively in the obvious way. All empty lists of the same underlying type represent the same data structure. So, $SV(L_0(\tau)) = \emptyset$ for all $\tau$ and $L_0(L_m(\text{int}))$ represents the same structure as $L_0(L_0(\text{int}))$.

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

$$\text{GroundTypes} \quad \tau^* := \tau \text{ such that } SV(\tau) = \emptyset \land TV(\tau) = \emptyset$$

The semantics of ground types is defined in Section 3.1. Here we give some examples: $L_2(\text{bool})$, $L_2(L_B(\text{bool}))$, and $L_{\text{concat}(2, B)}(\text{bool})$, where $B(pos) = pos$ on $0 \leq pos \leq 1$. It is easy to see that $\text{concat}(2, B) = \{0\} + \{1\} = \{1\}$. Examples of their inhabitants are $\text{true}$, $\text{true}$, $\text{true}$, and $\text{true}$, respectively. Examples of non-ground types are $\alpha$, $L_n(\text{int})$, $L_n(L_M(\text{bool}))$ and $L_{\text{concat}(n, M)}(\text{bool})$ with unspecified $n$ and $M$.

Let $\tau^t$ denote a zero-order type where size expressions are all size variables or constants, like, e.g., $L_n(\alpha)$ and $L_n(L_M(\alpha))$. Function types are then defined inductively:

$$\text{FunctionTypes} \quad \tau^f := \tau^l_1 \times \ldots \times \tau^l_{k'} \times \tau^t_1 \times \ldots \times \tau^t_k \rightarrow \tau_0$$

where $k'$ may be zero (i.e. the list $\tau^l_1, \ldots, \tau^l_{k'}$ is empty) and $SV(\tau_0)$ contains only size variables of $\tau^l_1, \ldots, \tau^l_k$.

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

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

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

The semantics \( w \) of a program value \( v \) with respect to a specific heap \( h \) and a ground type \( \tau^* \) is a set-theoretic interpretation given via the four-place relation \( v \models_h^w \). Integer and boolean constants interpret themselves, and locations are interpreted as non-cyclic lists. Let \( p_1(\nu^*_0) \) denote the set of values of some expression \( p_1 \) applied to some values \( \nu^*_0 \). Then

\[
\begin{align*}
\text{null} & \models_h^{\text{null}}(\nu^*_0) (\tau^*) \iff 0 \in p_1(\nu^*_0) \\
\ell & \models_h^{\text{adr}}(\nu^*_0) (\tau^*) \text{ whd} := w_0 \iff h.\ell.hd \models_h^{h.\ell.hd \models_h^{\text{null}}(\nu^*_0)} w_0, \\
& \quad h.\ell.tl \models_h^{h.\ell.tl \models_h^{\text{null}}(\nu^*_0)} w_0
\end{align*}
\]

where \( h|_{\text{dom}(h) \backslash \{\ell\}} \) denotes the heap equal to \( h \) everywhere except in \( \ell \), where it is undefined, \( (\tau^*)_{+1} \) and \( \tau_{+1} \) are abbreviations for \( \lambda \text{pos}. \, p_1^* (\text{pos} + 1) \) and \( \lambda \text{pos}. \, \tau (\text{pos} + 1) \), respectively and the application of a type to a first-layer size expression \( \tau(p_1) \) is defined as follows:

\[
\begin{align*}
\tau^0(p_1) & = \tau^0 \\
\tau_{+1}^* (p_1) & = \tau_{+1}^* \\
\lambda p_1^*(\tau_{+1}^* s(p_1)) & := \lambda p_1^*(\tau_{+1}^* s(p_1)), \text{ for } s' \geq 2
\end{align*}
\]

The length_\( (-) \) : Heap \( \rightarrow \) Address \( \rightarrow \) \( \mathcal{N} \) of a non-cyclic chain of cons-cells in a heap is defined by induction in a usual way: length_\( h(\text{null}) = 0 \) and length_\( h(\ell) = 1 \) + length_\( h|_{\text{dom}(h) \backslash \{\ell\}} (h.\ell.tl) \). Note that the function length_\( h(\_ \_ \_ ) \) does not take sharing into account, in the sense that the actual total size of allocated shared lists is less than the sum of their lengths. Thus, the sum of the lengths of the lists provides an upper bound on the amount of memory actually allocated.

**Lemma 1 (Consistency of model relation).** The relation \( \text{adr} \models_h^{\text{adr}}(\nu^*_0) (\tau^*) \) implies that length_\( h(\text{adr}) \) \( \in p_1(\nu^*_0) \).

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, $\text{Store } s : \text{ProgramVars} \rightarrow \text{Val}$.

Using a heap, a frame store and mapping $C$ (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:

\[
\begin{align*}
\text{OSVAR} & : s; h; \mathcal{C} \vdash z : \mathcal{V} & \quad \text{OSVAR} \\
\text{OSCONS} & : s; h; \mathcal{C} \vdash e \rightarrow c; h' & \quad \text{OSCONS} \\
\text{OSNIL} & : s; h; \mathcal{C} \vdash \text{Nil} : \text{NULL}; h & \quad \text{OSNIL} \\
\text{OSIFTRUE} & : s; h; \mathcal{C} \vdash \text{if } x \text{ then } e_1 \text{ else } e_2 : v; h' & \quad \text{OSIFTRUE} \\
\text{OSIFFALSE} & : s; h; \mathcal{C} \vdash \text{if } x \text{ then } e_1 \text{ else } e_2 : v; h' & \quad \text{OSIFFALSE} \\
\text{OSLET} & : s; h; \mathcal{C} \vdash e_1 : v; h' & \quad \text{OSLET} \\
\text{OSMATCH-NIL} & : s; h; \mathcal{C} \vdash \text{match } l \text{ with } | \text{Nil} \Rightarrow e_1 ; h' | \text{Cons}(hd, tl) \Rightarrow e_2 ; h' & \quad \text{OSMATCH-NIL} \\
\text{OSMATCH-CONS} & : s; h; \mathcal{C} \vdash \text{match } l \text{ with } | \text{Nil} \Rightarrow e_1 ; h' | \text{Cons}(hd, tl) \Rightarrow e_2 ; h' & \quad \text{OSMATCH-CONS} \\
\text{OSLETFUN} & : s; h; \mathcal{C} \vdash \text{letfun } \mathcal{f}(g_1, \ldots, g_k, z_1, \ldots, z_k) = e_1 \text{ in } e_2 : v; h' & \quad \text{OSLETFUN} \\
\text{OSFUNAPP} & : s; h; \mathcal{C} \vdash f(e_1, \ldots, e_k, z_1, \ldots, z_k) : v; h' & \quad \text{OSFUNAPP}
\end{align*}
\]
3.3 Typing rules

A typing judgement is a relation of the form $D, \Gamma \vdash x : \tau$, i.e. given a set of constraints $D$, a zero-order context $\Gamma$ and a higher-order signature $\Sigma$, an expression $e$ has a type $\tau$. 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 $L_p(n)(a)$, then $D$ is extended with $0 \notin p(n)$. When the cons-branch is entered, then $D$ is extended with $m \geq 1$, $m \notin p(n)$, where $m$ is a fresh size variable in $D$. When a constructor is applied, $D$ is extended with position-delimiting disequations.

Given types $\tau = L_{p_1(n)}(\ldots L_{p_{\tau}(n)}(\alpha) \ldots)$ and $\tau' = L_{p_1(n)}(\ldots L_{p_{\tau}(n)}(\alpha) \ldots)$, let the entailment $D \vdash \tau \rightarrow \tau'$ abbreviate the collection of rules that (conditionally) rewrite $p^1(n) \rightarrow p^1(n)$ etc.:

\[
D, m_1 \in p^1(n') \, 0 \leq pos \leq m_1 - 1 \quad \vdash p^1(n') \rightarrow p^2(n')(pos)
\]

\[
D, \{ \begin{array}{l}
m_1 \in p^1(n') \, 0 \leq pos_1 \leq m_1 - 1, \\
m_2 \in p^1(n')(pos_1) \, 0 \leq pos_2 \leq m_2 - 1 \\
m_1, pos_1, m_2, pos_2 \text{ are fresh for } D
\end{array} \quad \vdash \{ \begin{array}{l}
p^1(n')(pos_1)(pos_2) \rightarrow p^2(n')(pos_1)(pos_2) \\
p^1(n')(pos_1) \rightarrow p^2(n')(pos_1)
\end{array}
\]

... 

\[
D, \{ \begin{array}{l}
m_1 \in p^1(n') \, 0 \leq pos_1 \leq m_1 - 1, \\
m_s \in p^{s-1}(n')(pos_1) \ldots (pos_{s-1}), \\
0 \leq pos_s \leq m_s - 1 \\
m_1, pos_1, \ldots, m_s, pos_s \text{ are fresh for } D
\end{array} \quad \vdash \{ \begin{array}{l}
p^1(n')(pos_1) \ldots (pos_s) \rightarrow p^s(n')(pos_1) \ldots (pos_s) \\
p^1(n')(pos_1) \rightarrow p^{s-1}(n')(pos_1) \\
\end{array}
\]

The typing judgement relation is defined by the following rules:

\[
\frac{D, \Gamma \vdash \tau : \text{Int}}{\text{IC}}
\]

\[
\frac{D, \Gamma \vdash \tau : \text{Bool}}{\text{BC}}
\]

\[
\frac{D, \Gamma \vdash \tau' \rightarrow \tau}{\text{VAR}}
\]

\[
\frac{D, \Gamma \vdash \text{Nil} : \tau'}{\text{NIL}}
\]

\[
\frac{D \vdash \tau' \rightarrow \tau}{\text{CONS}}
\]

\[
\frac{D, \Gamma \vdash \text{Nil} : \tau_1 \vdash \tau_2}{\text{CONS}}
\]

\[
\frac{D, \Gamma \vdash \text{Cons}(	ext{hd}, \text{tl}) : \tau'}{\text{CONS}}
\]

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

Moreover, backward application of the $\text{Cons}$-rule to $n \geq 1, \text{tl} : L_{\text{tl}(\text{tl})}(\text{tl}(\text{tl}))$, $\text{tl}(\text{tl}) : L_{\text{tl}(\text{tl})}(\text{tl}(\text{tl}))$, allows to infer the
rewriting rules for the sizes of the inner lists of the output for \texttt{tails}:

\[
\begin{align*}
  n \geq 1 \quad & \Rightarrow \text{tails}_2(n)(0) \rightarrow n \\
  n \geq 1, \ 1 \leq \text{pos} \leq n \quad & \Rightarrow \text{tails}_2(n)(\text{pos}) \rightarrow \text{tails}_2(n-1)(\text{pos}-1)
\end{align*}
\]

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

\[
\Gamma(x) = \text{Bool} \\
D, \Gamma \vdash \Sigma' \ e_1 : \tau_1 \\
D, \Gamma \vdash \Sigma' \ e_2 : \tau_2 \\
\text{IF}
\]

\[
\begin{align*}
  z \notin \text{dom}(\Gamma) \\
  D, \Gamma \vdash \Sigma' \ e_1 : \tau_z \\
  D, \Gamma, z \vdash \Sigma' \ e_2 : \tau \\
\end{align*}
\]

\[
\text{LET}
\]

\[
D, 0 \in p^1(\overline{n'}) \Rightarrow \Gamma, \ e_1 : \text{Nil} : \tau' \Rightarrow \text{hd, tl} \not\in \text{dom}(\Gamma)
\]

\[
D, m \geq 1 \in p^1(\overline{n'}) \Rightarrow \Gamma, \text{hd} : \tau(0) \Rightarrow \Gamma, \text{tl} : \text{Nil} : \tau' \Rightarrow \text{ef, cons} : \tau'
\]

\[
\text{MATCH}
\]

where $n' \not\in SV(D)$. 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.

\[
\begin{align*}
\Sigma(f) &= \tau^1_1 \times \ldots \times \tau^1_{p^1} \times \tau^2_1 \times \ldots \times \tau^2_{p^2} \rightarrow \tau_0 \\
\Sigma(g_i) &= \tau^1_{i_1} \times \ldots \times \tau^1_{i_{p^1}} \\
\Sigma(f, g_1, \ldots, g_k, z_1, \ldots, z_k) &= \text{let fun} f(g_1, \ldots, g_k, z_1, \ldots, z_k) = e_1 \text{ in } e_2 : \tau^2
\end{align*}
\]

\[
\text{LETFUN}
\]

where $\sigma$ 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 $\tau^1_1 = L(\ldots L_n(\tau^0) \ldots)$ and $\tau^2 = L(\ldots L_p(\overline{n'}) \ldots)$, then $\sigma(n^\alpha) := p^1(n^\alpha)$. Further, if $\tau^1_1 = L(\ldots L_n(\tau^0) \ldots)$, then $C$ contains $p^1(\overline{n'}) = a^\alpha$. Eventually $\sigma(\tau_0) = L(\ldots L_f(\ldots, a^\alpha, \ldots)(\ldots \ldots)\ldots)$ is defined as $L(\ldots L_f(\ldots, a^\alpha, \ldots)(\ldots \ldots)\ldots)$.

As an example of a case when $C$ is needed, consider a call of a function \texttt{scalarprod} : $L_m(\text{Int}) \times L_n(\text{Int}) \rightarrow \text{Int}$ on actual size arguments $l_1 : L_{n+1}(\text{Int})$ and $l_2 : L_{m-1}(\text{Int})$. Then $C$ contains $n+1 = m-1$. It will hold if $D$ contains $n = m-2$.

\[\text{Example 1: inferring rewriting rules for } \texttt{concat} \]

In the introduction we have given the rewriting rules defining the type for \texttt{concat} : $L_n(L_M(\alpha)) \rightarrow L_{\text{concat}(n, M)}(\alpha)$, where

\[
\begin{align*}
  \vdash \text{concat}(0, M) & \rightarrow 0 \\
  n \geq 1 \quad & \vdash \text{concat}(n, M) \rightarrow M(0) + \text{concat}(n-1, \lambda \text{pos. } M(\text{pos} + 1))
\end{align*}
\]
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 \texttt{LetFun} rule defines the main goal: \( \Gamma : \text{Ln} M (\alpha) \vdash \epsilon \text{conc} : \text{L}_{\text{concat}(n,M)}(\alpha) \), where \( \epsilon \text{conc} \) denotes the body of \( \text{concat} \).
2. Apply the MATCH-rule. In the nil-branch we obtain the subgoal \( n = 0; \Gamma : \text{Ln} M (\alpha) \vdash \epsilon \text{Nil} : \text{L}_{\text{concat}(n,M)}(\alpha) \).
3. Continue with the nil-branch. Apply the \texttt{Nil} rule and obtain \( n = 0 \vdash \text{L}_{\text{concat}(n,M)}(\alpha) \rightarrow L_0(\tau^?) \).
4. Instantiate \( \tau^? = \alpha \). Unfold the definition of type rewriting: \( n = 0 \vdash \text{concat}(n,M) \rightarrow 0 \).
5. Now, consider the cons-branch. The subgoal there is \( n \geq 1; \text{hd} : \text{L}_M(\alpha)(0), \text{tl} : \text{L}_{n-1}(\text{L}_M(\alpha) + \Sigma \text{append}(\text{hd}, \text{concat}(\text{tl}))) : \text{L}_{\text{concat}(n,M)}(\alpha) \).
(Not 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 \((-\_+\_):\)
\( n \geq 1; \text{hd} : \text{L}_M(\alpha)(0), \text{tl} : \text{L}_{n-1}(\text{L}_M(\alpha) + \Sigma \text{append}(\text{hd}, \text{concat}(\text{tl}))) : \text{L}_{\text{concat}(n,M)}(\alpha) \).
7. The expression in the judgement above is a sugared let-construct. So, we apply the \texttt{LET}-rule. In the binding we get the goal: \( n \geq 1; \text{tl} : \text{L}_{n-1}(\text{L}_M(\alpha)) + \Sigma \text{concat}(\text{tl}) : \tau^? \).
8. Using \texttt{FUNApp}-rule we instantiate the type \( t^? := \text{L}_{\text{concat}(n-1,M+1)}(\alpha) \).
9. Therefore, the subgoal for the let-body is \( n \geq 1; \text{hd} : \text{L}_M(\alpha)(0), \text{tl} : \text{L}_{n-1}(\text{L}_M(\alpha) + \Sigma \text{append}(\text{hd}, \text{concat}(\text{tl}))) : \text{L}_{\text{concat}(n,M)}(\alpha) \).
10. Unfold the definition of type rewriting and the definition of the operation \((-\_+\_):\)
\( n \geq 1; \text{hd} : \text{L}_M(\alpha)(0), \text{tl} : \text{L}_{n-1}(\text{L}_M(\alpha) + \Sigma \text{append}(\text{hd}, \text{concat}(\text{tl}))) : \text{L}_{\text{concat}(n,M)}(\alpha) \).

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

1. The \texttt{LetFun} rule defines the main goal: \( \Gamma : \text{Ln}(\alpha) \vdash \epsilon \text{tails} : \text{L}_{\text{tails}(n)}(\text{L}_{\text{tails}(n)}(\alpha)) \),
where \( \epsilon \text{tails} \) denotes the body of \( \text{tails} \).
2. Apply the MATCH-rule. In the nil-branch we obtain the subgoal \( n = 0; \Gamma : \text{Ln}(\alpha) \vdash \epsilon \text{Nil} : \text{L}_{\text{tails}(n)}(\text{L}_{\text{tails}(n)}(\alpha)) \).
3. Continue with the nil-branch. Apply the \texttt{Nil} rule and obtain \( n = 0 \vdash \text{L}_{\text{tails}(n)}(\text{L}_{\text{tails}(n)}(\alpha)) \rightarrow L_0(\tau^?) \).
4. Trivially, instantiate \( \tau^? := \text{L}_{\text{tails}(n)}(\alpha) \). Unfold the definition of the type rewriting:
\( n = 0 \vdash \text{tails}(n) \rightarrow 0 \).
Note, that the rewriting rules for \( \text{tails}_2(n) \) in this branch are absent, since \( n_1 \in \{n = 0\}, 0 \leq \text{pos} \leq n_1 - 1 \) is an empty set.
5. Now, consider the cons-branch. The subgoal there is \( n \geq 1; \text{hd} : \text{L}_n(\alpha), \text{tl} : \text{L}_{n-1}(\text{L}_M(\alpha)) + \Sigma \text{concat}(\text{tl}) : \text{L}_{\text{tails}(n)}(\text{L}_{\text{tails}(n)}(\alpha)) \).
(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 \( (\_)^+) \):
\[
\begin{align*}
\text{Let } n &\geq 1; \ l : L_{n-1}(\alpha), \ tl : L_n(a) \implies \text{Cons}(l, \text{tails}(tl)) : L_{n+1}(\alpha), \\
\text{Using } \text{FUNApp-rule} \text{ we instantiate the type } \tau^2 := L_{n+1}(\alpha) &\iff \text{for } \alpha, \ n \geq 1.
\end{align*}
\]

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: \( n \geq 1; \ tl : L_{n-1}(\alpha) \implies \text{tails}(tl) : \tau^2 \).

8. Using FUNAPP-rule we instantiate the type \( \tau^2 := L_{n+1}(\alpha) \).

9. Therefore, the subgoal for the let-body is
\[
\begin{align*}
n &\geq 1; \ l : L_{n-1}(\alpha), \ z : L_{n+1}(\alpha) \implies \text{Cons}(\text{hd}, \text{hd}) : L_{n+1}(\alpha).
\end{align*}
\]

10. Apply the CONS-rule. We obtain the predicates
\[
\begin{align*}
n &\geq 1 \implies L_{n+1}(\alpha) \iff L_n(\alpha) \\
n &\geq 1 \implies \tau^2(0) \iff L_n(\alpha) \\
n &\geq 1, \ 1 \leq \text{pos} \leq n \implies \tau^2(\text{pos}) \iff (L_{n+1}(\alpha))(\text{pos} - 1)
\end{align*}
\]

11. Trivially, instantiate \( \tau^2 := L_{n+1}(\alpha) \). We obtain
\[
\begin{align*}
n &\geq 1 \implies L_{n+1}(\alpha) \iff L_n(\alpha) \\
n &\geq 1, \ 1 \leq \text{pos} \leq n \implies (L_{n+1}(\alpha))(\text{pos} - 1)
\end{align*}
\]

12. Unfold the definition of type-typewriting. For \( \text{tail}_1 \) we obtain \( n \geq 1 \implies \text{tail}_1(n) = \text{tail}_1(n-1) + 1 \), and for \( \text{tail}_2 \), unfolding the definition of application of a type to a first-layer expression, we obtain
\[
\begin{align*}
n &\geq 1 \implies \text{tail}_2(n)(0) \iff n \\
n &\geq 1, \ 1 \leq \text{pos} \leq n \implies \text{tail}_2(n)(\text{pos}) \iff (\text{tail}_2(n-1))(\text{pos} - 1)
\end{align*}
\]

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

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 \( \Gamma^* \):
\[
\begin{align*}
\text{Valid}_v(w, \tau, h) &\iff w \Vdash_h \tau \\
\text{Valid}_v(s, \Gamma^*, s, h) &\iff \forall x : e : \text{Vars}. \text{Valid}_v(s(x), \Gamma^*(x), h)
\end{align*}
\]

Let a valuation \( \epsilon^v \) map size variables to constants of the layer \( s \), and let an instantiation \( \eta^v \) map type variables to ground types:
\[
\begin{align*}
\text{Valuation } &\epsilon^v : \text{SizeVariables}^s \to (\mathcal{R} \to \cdots \to \mathcal{R} \to \mathcal{R}^\mathcal{R}) \\
\text{Instantiation } &\eta^v : \text{TypeVariables}^s \to \tau^{*^s}
\end{align*}
\]

Let \( \epsilon \) and \( \eta \) 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 \( \eta \) and \( \epsilon \) as subscripts. For example, \( \eta(\epsilon(\tau)) \) becomes \( \tau_{\eta\epsilon} \) and \( \epsilon(D) \) becomes \( D_\epsilon \). Note that \( D \) contains no type variables and hence \( D_\eta = D \). Valuations and instantiations distribute over size functions in the following way: \( (L_{\eta(\epsilon(\tau))})(\eta\epsilon) = L_{\eta(\epsilon(\tau))}(\tau_{\eta\epsilon}) \).
Lemma 2 (Rewriting preserves model relation (i.e. implies set-theoretic inclusion of types)). Let $D(\overline{n}) \vdash \tau \rightarrow \tau'$. Let a valuation $\epsilon$ and a type instantiation $\eta$ be such that $v \Vdash_{\tau,\eta} w$ and $D_\epsilon$ hold. Then $v \Vdash_{\tau',\eta} w$ holds as well.

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

Let $\tau = Lp^1(\overline{n}^*)$ and $\tau' = Lp^1(\overline{n}^*)(\tau''')$ for some $\tau''$ and $\tau'''$, and let $c(\overline{n}^*) = \overline{n}_0^*$. Assume $v = \text{NULL}$. Then $0 \in p_1^1(\overline{n}_0^*)$ and $w = \emptyset$. Since $p_1^1(\overline{n}_0^*) \rightarrow p_1^1(\overline{n}_0^*)$, that is $p_1^1(\overline{n}_0^*) \subseteq p_1(\overline{n}_0^*)$, we have $0 \in p_1(\overline{n}_0^*)$ and $v \Vdash_{\tau,\eta} \emptyset$.

Now assume that $v = \ell$ and $w = w_{hd} :: w_{ld}$, where $h.\ell.\ell.1 \Vdash_{\tau'''}[\ell]\{\ell\} w_{ld}$ and $h.\ell.\ell.1 \Vdash_{\tau'''}[\ell]\{\ell\} w_{ld}$. Since there is $n \in p_1(\overline{n}_0^*)$ with $n \geq 1$ (because we are in the non-empty case), we have $D \vdash \tau'' \rightarrow \tau'''$, from which follows that $D \vdash \tau'' \rightarrow \tau''' \rightarrow \tau_{n+1}$. Then, by induction, $h.\ell.\ell.1 \Vdash_{\tau_{n+1}}[\ell]\{\ell\} w_{ld}$. Since $p_1(\overline{n}^*) \rightarrow p_1^1(\overline{n}^*)$, we have that $p_1(\overline{n}^*) - 1 \rightarrow p_1^1(\overline{n}^*) - 1$, and by induction $h.\ell.\ell.1 \Vdash_{\tau_{n+1}}[\ell]\{\ell\} w_{ld}$. □

This lemma may seem counterintuitive on a first sight because it looks like a type preservation lemma where the type $\tau$ and $\tau'$ 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 ($\tau$) and one in particular is chosen to obtain the right hand side ($\tau'$). So, if a value has type $\tau'$, it also has type $\tau$.

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 $C$, expression $e$, value $v$, context $\Gamma$, quantifier-free formula $D$, signature $\Sigma$, type $\tau$, size valuation $\epsilon$, and type instantiation $\eta$ such that

- $\text{dom}(s) = \text{dom}(\Gamma)$, $\epsilon : SV(\Gamma) \cup SV(D) \rightarrow \mathcal{R}$ and $\eta$: $TV(\Gamma) \rightarrow \tau^*$.
- $D_\epsilon$ holds,
- $s$, $h$: $C \vdash e \rightsquigarrow v$; $h'$ and $D$, $\Gamma \vdash_\Sigma e : \tau$,
- $\text{Valid}_{\text{store}}(\text{dom}(s), \Gamma_{\eta_c}, s, h),$

then $v$ is valid according to its return type $\tau$ in $h'$, i.e., $\text{Valid}_{\text{val}}(v, \tau_{\eta_c}, h')$.

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 $e$ terminates (with a value $v$). We have to show that $\text{Valid}_{\text{val}}(v, \tau_{\eta_c}, h')$, i.e., that there is a $w$ such that $v \Vdash_{\tau_{\eta_c}} w$. This is proved for each of the operational-semantic rules.
OSNull: In this case e = Nil, v = NULL and h' = h. From the NULL typing rule we have that D ⊢ τ → L₀(τ'). According to the definition of rewriting rule, τ = L_p(τ^n) for some p, m^n and τ', where p(m^n) → 0. But then p(m^n) → 0 and hence 0 ∈ p(m^n). But then from the definition of model relation we get that NULL |^ b' τ |^ b m^n | and thus NULL |^ b' τ |^ b m^n |.

OSVar: In this case e = z, v = s(z) and h' = h. Since dom(s) = dom(Γ), there is a τ' such that Γ(z) = τ', and because Validstore(dom(s), Γ_ne, s, h), there is a w such that s(z) |^ b τ' w. Now from the VAR typing rule, D ⊢ τ → τ'.

OSCons: In this case e = Cons(hd, tl), v = ℓ for some location ℓ ∈ dom(h) and h' = h. From the CONS typing rule we have that hd: τ₁ and tl: τ₂, and the judgements D ⊢ τ' → L_p(τ^n) (τ''), D ⊢ τ''(0) → τ₁ and n ∈ p(τ^n), 1 ≤ pos ≤ n, D ⊢ τ''(pos) → τ₂(pos - 1). Since Validstore(dom(s), Γ_ne, s, h), there exist w hd and w tl such that s(hd) |^ b τ₁ w hd and s(tl) |^ b τ₂ w tl.

OSIfTrue: In this case e = if x then e₁ else e₂, with s(x) = True and s; h; C ⊢ e₁: τ₁, e₂: τ₂. Since e₁ is evaluated in the same context as e (s; h; C), we can use the induction hypothesis to get Validva|(v, T₁, h').

OSIfFalse: Similar to the true case.

OSLet: In this case e is let z = e₁ in e₂, where s; h; C ⊢ e₁: τ₁, e₂: τ₂ and D ⊢ τ → τ₁. From the LET typing rule we have that z ∈ dom(Γ) and s; h; C ⊢ e₁: τ₁ and s[z := v₁]; h₁; C ⊢ e₂: τ₂. From the LET typing rule we have that z ∈ dom(Γ) and s; h; C ⊢ e₁: τ₁ and s[z := v₁]; h₁; C ⊢ e₂: τ₂. Applying the induc-
tion hypothesis to the antecedents of the operational semantics, we get that
\[ \text{Valid}_{\text{val}}(v, \tau_{\text{ne}}, h_1) \] and that if \( \text{Valid}_{\text{store}}(s[z := v_1], \Gamma_\tau \cup \{ z : \tau_{\text{ne}} \}, s[z := v_1], h_1) \) then \( \text{Valid}_{\text{val}}(v, \tau_{\text{ne}}, h') \).

Fix some \( z' \in \text{dom}(s[z := v_1]) \). If \( z' = z \), then \( \text{Valid}_{\text{val}}(v_1, \tau_{\text{ne}}, h_1) \) implies 
\[ \text{Valid}_{\text{store}}(s[z := v_1](z), \Gamma_\tau \cup \{ z : \tau_{\text{ne}} \}, s[z := v_1], h_1) \] and if \( z' \neq z \), then \( s[z := v_1](z') = s(z') \). Sharing of data structures in the heap is benign (no destructive pattern matching and assignments), hence \( h|R(h, s(z')) = h_1|R(h, s(z')) \). Thus, we have that
\[ s(z') \models h_1(R(h, s(z')) \models s(z') \] and then \( s[z := v_1](z') = s(z') \).

So, \( \text{Valid}_{\text{val}}(s[z := v_1](z'), \Gamma_\tau(z'), h_1) \). Hence, \( \text{Valid}_{\text{store}}(s[z := v_1], \Gamma_\tau \cup \{ z : \tau_{\text{ne}} \}, s[z := v_1], h_1) \) and we can now apply the induction hypothesis.

**OSMatch-Nil:** In this case \( e = \text{match } l \text{ with } | \text{Nil} \Rightarrow e_1 | \text{Cons}(hd, tl) \Rightarrow e_2 \) where \( s(l) = \text{NULL} \) and \( s; h; C \vdash e_1 \leadsto v; h' \). From the MATCH typing rule we have that \( \Gamma'' = \Gamma'' \cup \{ l : L_p(f(n))(\tau') \} \) and \( D, 0 \in p^1(\tau') \). From \( \text{Valid}_{\text{store}}(s(l), \Gamma_\tau, s, h) \) we get \( \text{Valid}_{\text{val}}(s(l), L_p(f(n))(\tau'), h) \) and since \( s(l) = \text{NULL} \), from the definition of model relation we get that \( 0 \in p^1(\tau') \).

Therefore, the typing judgement about \( e_1 \) reduces to \( \Gamma'' = \Gamma'' \cup \{ l : L_p(f(n))(\tau') \} \vdash e_1 : \tau \), where \( e_1 = \text{match } l \text{ with } | \text{Nil} \Rightarrow e_1 | \text{Cons}(hd, tl) \Rightarrow e_2 \). The typing context has the form \( \Gamma'' = \Gamma'' \cup \{ l : L_p(f(n))(\tau') \} \). From the operational semantics we know that \( h.s(l).hd = v_{hd} \) and \( h.s(l).tl = v_{tl} \), that is, \( s(l) \neq \text{NULL} \). Due to the validity of \( s(l) \) and Lemma 1, there exists \( n_0 \geq 1 \in p^1(\tau') \). From the validity \( s(l) \models h_{td} = v_{td} \) and \( v_{td} = L_{p_{n_0}^1}(\tau_{\text{ne}}) \), the validities of \( v_{hd} \) and \( v_{tl} \) follow: \( v_{hd} = L_{p_{n_0}^1}(0) \) and \( v_{td} = L_{p_{n_0}^1}(1) \). From the MATCH typing rule we have that \( D, n_0 \geq 1 \in p^1(\tau ''); \Gamma'' = \Gamma'' \vdash e_2 : t \).

**OSMatch-Cons:** In this case \( e = \text{match } l \text{ with } | \text{Nil} \Rightarrow e_1 | \text{Cons}(hd, tl) \Rightarrow e_2 \) where \( s; h; C \vdash e_1 \leadsto v; h' \). From the LETFun typing rule we have that \( \Gamma'' = \Gamma'' \cup \{ l : L_p(f(n))(\tau') \} \) and the results above, we obtain that \( \text{Valid}_{\text{store}}(s(l'), \Gamma_\tau, s, h) \) for \( s(l') = s[hd := v_{hd}, tl := v_{tl}] \). With \( e' = e[e_1 := \text{match } l \text{ with } | \text{Nil} \Rightarrow e_1 | \text{Cons}(hd, tl) \Rightarrow e_2] \), the induction hypothesis yields \( \text{Valid}_{\text{val}}(v, \tau_{\text{ne}}, h') \). Now, since \( n_0 \notin SV(\tau) \) (and thus, \( \tau_{\text{ne}} = \tau_{\text{ne}}' \)), we have \( \text{Valid}_{\text{val}}(v, \tau_{\text{ne}}, h') \).

**OSLetFun:** Here \( e = \text{letfun } f(f_1, \ldots, f_k, z_1, \ldots, z_k) = e_1 \text{ in } e_2 \), where \( s; h; C \vdash e_1 \leadsto v; h' \). From the LETFun typing rule we have that \( \Gamma'' = \Gamma'' \vdash e_2 : \tau \). Applying the induction hypothesis to these judgements with the same \( n \) and \( \eta \), we obtain \( \text{Valid}_{\text{val}}(v, \tau_{\text{ne}}, h') \) as desired.

**OSFunApp:** In this case \( e = f(f_1, \ldots, f_k, z_1', \ldots, z_k') \), where \( C(f) = (g_1, \ldots, g_k, z_1', \ldots, z_k') \times e_1 \) and \( (z_1 := v_1, \ldots, z_k := v_k) \); \( C \vdash e_1[g_1 := f_1, \ldots, g_k := f_k] \leadsto v; h' \). We want to apply the induction hypothesis 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 \( \text{True} \). Take \( \eta' \) and \( \epsilon' \) such that
\(- n'(a) = n(Ta), \text{ where } Ta \text{ is such that } a \text{ is replaced by } Ta \text{ in the instantiation of } \sigma \text{ of the signature in this application of the FUNApp-rule.} \)

\(- e'(njj) = e(fjj), \text{ where } njj \text{ is replaced by } jj \text{ in the instantiation of } \sigma \text{ of the signature in this application of the FUNApp-rule.} \)

True holds trivially on \(e'\). From the inductive hypothesis we have that if \(\text{Valid}_{\text{store}}((y_1, \ldots, y_k), (y_1 : \tau_{1}, \ldots, y_k : \tau_{k}), [y_1 := v_1, \ldots, y_n := v_n], h)\) then \(\text{Valid}_{\text{eval}}(v, \tau_{0}, h')\).

From \(\text{Valid}_{\text{store}}(\text{dom}(s), \Gamma_{\text{nc}}, s, h)\) we get the validity of the values of the actual parameters: \(v_i \vdash_{\tau_{\text{nc}}} w_i\) for some \(w_i\), with \(1 \leq i \leq k\). Since \(\Gamma_{\text{nc}}(l_i) = \tau_{i}, \epsilon'\), the left-hand side of the implication holds, and one obtains \(\text{Valid}_{\text{eval}}(v, \tau_{0}, h')\). It is easy to see that

\[\sigma(\tau_0) = \eta(\tau_0)[\ldots, \alpha := \tau_0][\ldots] = \tau_0[\ldots, \alpha := \eta(\tau_0)[\ldots] = \epsilon(\tau_0)) = \tau_{0, \epsilon'}\]

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

\[\square\]

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 \(n^*_i \in \overline{n}\) of a layer \(s \geq 2\), let its range be given in the form \(T(n^*_i) = \{p_i(n, n', i)\}_{i \geq 0}\), which is a short cut for \(\{p_i(n_0, n'_0, i) \mid i \geq 0\}\). Here \(Q\) is a first-order arithmetic predicate and \(\overline{n}\) are fresh w.r.t. \(n\). We introduce fresh size variables like \(n^*\) and assumptions as the one above if we know nothing about \(n^*\), where \(s \geq 2\). In general such default assumptions are of the form \(\text{range}(n^*) \subseteq \{i\}_{i \geq 0} \leq n\).

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

Sometimes it is convenient to consider more specific estimates, where positions are mentioned explicitly. For instance, \(\text{tail}_2(n)(\overline{\text{pos}}) = n - \overline{\text{pos}}\) for \(n \geq 1\) and \(0 \leq \overline{\text{pos}} \leq n - 1\). 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 \(\sum_{\text{pos} = 0}^{n-1}\text{tail}_2(n - \overline{\text{pos}}) = \sum_{\text{pos} = 0}^{n-1}(n - \overline{\text{pos}}) = \frac{n(n + 1)}{2}\). This is definitely more precise than the position-free estimate \(\text{tail}_2(n) \subseteq \{i\}_{i \leq \overline{\text{pos}}} \leq n\). In general, position-aware estimates for bound of internal lists have the form \(\text{range}(n^*) \subseteq T(n^*) \land D(\overline{n}, \overline{n}', \overline{\text{pos}})\) implies \(f(n^*) \subseteq \{p(\overline{n}, \overline{n}', \overline{\text{pos}})\}_{Q(\overline{n}, \overline{n}', \overline{\text{pos}})}\).

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

Our main assumption is that for any fixed \( \pi_0, \pi_0 \) the sets \( T(\pi^n) \) are finite. For instance, for \( n' = 3 \) the range of \( M \) is included into the set \( \{0, 1, 2, 3\} \). Moreover, for fixed \( n \) and \( n' \) the function \( M \) is reduced to the finite multivalued map \( \phi \) such that \( \phi(\text{pos}) \) is the set of all possible lengths of the “inner” lists. E.g. with \( n = 2 \) and \( n' = 3 \) we have \( \phi \) instantiated as \( \phi(0) = \phi(1) = \{0, 1, 2, 3\} \).

With fixed \( \pi \) and \( \pi' \) the function \( f \) is translated to an auxiliary function \( \mathcal{L}_{f,j} \) over finite sets and maps. For instance, \( \text{concat}(\pi, M) \) becomes \( \mathcal{L}_{\text{concat}}(\mathcal{L}(\phi)) \). Now we show how to translate \( f \) to the function \( \mathcal{L}_{f,j} \), 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

\[
(\text{FiniteSet}, \text{FiniteMMap})^* \rightarrow \text{FiniteSet}
\]

where \((\text{FiniteSet}, \text{FiniteMMap})^*\) is a finite Cartesian product of finite sets and finite multivalued maps. Binary arithmetic operations are lifted to sets: if \( \oplus \) is one of the arithmetic operations \(+, -, \times\), then \( \mu_1 \oplus \{\mu_2 \} := \{x \oplus y \} \subseteq \mu_1 \land \mu_2 \).

A finite multivalued map is a mapping from positions to finite sets: \( \text{FiniteMMap} : \text{Positions}_d \rightarrow \ldots \rightarrow \text{Positions}_d \rightarrow \text{FiniteSet} \) where \( \text{Positions}_d = \{0, \ldots, d-1\} \). An example of finite multivalued maps is \( (\{1, 2, 3\}, \{1\}) \), which sends 0 to \( \{1, 2, 3\} \) and 1 to \( \{1\} \). We denote a multivalued map via \( \phi \) and \( \mu \) denotes either a finite map or set. There is an empty map denoted via \( \emptyset \). The only operation over multivalued maps, which is relevant to our task, is left shift \( [-] \downarrow k \) 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 \textsc{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 \( \mathcal{L} \rightarrow \mathcal{L} \) that maps size expressions onto expressions over finite sets and finite multivalued maps is inducively defined on the structure of size expressions. We define the translation \( \mathcal{L} \rightarrow \mathcal{L} : \text{SizeExpressions} \rightarrow \text{FiniteMMapSetExpressions} \) as follows:

- first-layer constants represent themselves: \( \mathcal{L}a : = a \);
- higher-layer constants, \( s \geq 2 \) are translated into their restrictions: \( \mathcal{L} a^s : = a^s \); 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 \( \{0, \ldots, \max (p^1(\pi^n)) - 1\} \), where the expression \( p^1 \) is given by the type \( L(p^1(\pi^n))(L(a^2(\ldots))) \).
for instance, \( a^2 \cdot a = a^2 \), where \( a^2(0) = 0 \) and \( a^2(1) = \{0, 1\} \) and \( a^2 \) is taken from the type \( \text{L_n+2(L_n \ldots)} \) with \( n := 0 \);

- positions \( \text{pos} \) and first-layer variables \( n \) are translated to themselves: \( \text{\_pos}_j := \text{pos} \) and \( \text{\_n}_j := n \); 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 \( \text{\_n^s}_j := n^s \);

- translation \( \text{\_p1 \_p2}_j := \text{\_p1}_j \_p2 \) is defined on first-layer size expressions;

- translation \( \text{\_p+i}_j := \text{\_p}_j \), where \( p' \) is a first-layer expression with no free occurrences of \( \text{\_pos} \);

- \( \text{\_g}(p_1, \ldots, p_k)_j := \text{\_g}_j(p_1_j, \ldots, p_k_j) \).

Given a rewriting rule \( f(n^1, \ldots, n^k)(\text{pos}_1\ldots\text{pos}_{k-1}) \rightarrow p \) for a numerical multivalued function \( f \), we construct the corresponding rewriting rule for \( \text{\_f}_j \) as \( \text{\_f}_j(n^1, \ldots, n^k)(\text{\_pos}_1\ldots\text{pos}_{k-1}) \rightarrow \text{\_p}_j \). For instance, the rewriting rule \( n > 1, 0 \leq \text{pos} \leq n - 1 \rightarrow \text{tails}_2(n)(\text{pos}) \rightarrow \text{tails}_2(n - 1)(\text{pos} - 1) \) is translated to \( n > 1, 0 \leq \text{pos} \leq n - 1 \rightarrow \text{\_tails}_2_j(n)(\text{pos}) \rightarrow \text{\_tails}_2_j(n - 1)(\text{pos} - 1) \).

Generating a family
Consider a brunch of \( f \), defined by the rule \( D_1(\overline{n}^*, \overline{m}) \land D_2(\overline{m}, \text{\_pos}) \rightarrow f(\overline{n}^*)(\text{\_pos}) \rightarrow p \). We will construct an estimate for the range of \( f \) in the form \( \{f_l(n, n')(\text{\_pos}) + t \mid 0 \leq t \leq f_u(n, n')(\text{\_pos}) - f_l(n, n')(\text{\_pos})\} \). 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 \((V + d)\) points \((n_0, n'_0, \text{\_pos}_0)\), for which there exists \( m \) such that \( D_1(n^*, m) \land D_2(m, \text{\_pos}) \) holds, that uniquely define a polynomial of degree \( d \) with \( V = |n| + |n'| + |\text{\_pos}_0| \) variables. We have discussed how to choose such points in [17]. For instance, assuming \( d = 2 \) for \( \text{concat} \), assuming \( d = 1 \) for \( \text{tails}_2 \), we take the set of test points \((n_0, n'_0, \text{\_pos}_0) \) as \((1, 1, 0), (2, 1, 0), (3, 0, 0)\).

2. For each \((n_0, n'_0, \text{\_pos}_0)\) from the set of test points do:
   (a) for any \( n^i \in \overline{n} \) assign \( \phi_i := \lambda \text{pos}'. \{p_i(n_0, n'_0, \text{\_pos}_0)\} \), which is a constant multivalued map; e.g. \( \phi := \{0, 1\} \) for \( \text{in} \) in \( \text{concat}(n, M) \) with \( n = 1, n' = 1 \); for instance, for \( \text{concat} \), with \( n = 2\) and \( n' = 3 \) we have \( \phi(2, 3) = \{2\} \) and \( \phi(2, 3) = \{0, 1, 2, 3\} \);
   (b) compute \( \text{\_f}_j(n, \phi)(\text{\_pos}) \) using the rewriting rules; e.g.

   yet another example is \( \text{\_tails}_2_j(2)(1) \rightarrow \text{\_tails}_2_j(2 - 1)(1 - 1) \rightarrow \text{\_tails}_2_j(1)(0) \rightarrow 1 \).
(c) assign \( f_{\text{min}}(n_0, n', p_0) \) := \( \min (f(n_0, n', p_0)) \) and \( f_{\text{max}}(n_0, n', p_0) \) := \( \max (f(n_0, n', p_0)) \); e.g. \( \text{concat}_{\text{min}}(1, 3) := 0 \) and \( \text{concat}_{\text{max}}(1, 3) := 3 \); also \( \text{tails}_{\text{min}}(2, 1) = \text{tails}_{\text{max}}(2, 1) = 1 \).

(d) add to the lists of equations w.r.t. the coefficients of \( f_i \) and \( f_u \) the equations with \( f_{\text{min}}(n_0, n', p_0) \) and \( f_{\text{max}}(n_0, n', p_0) \) on the r.h.s., respectively; e.g., \( \text{tails}_u(2, 0) \) defines \( 2a_{u, 10} + a_{u, 01} + a_{u, 00} = \text{tails}_{\text{max}}(2, 1) = 1 \).

3. Solve the linear systems for the coefficients \( f_i \) and \( f_u \). For instance, solving the system for \( \text{concat}(n, n') \) and \( \text{concat}_u(n, n') \) gives \( \text{concat}_{i}(n, n') = 0 \) and \( \text{concat}_{u}(n, n') = n'n' \); for \( \text{tails}_2 \) we obtain \( \text{tails}_{2, i}(n, p) = \text{tails}_{2, u}(n, p) = n - p \). 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 \( \text{concat} \) we return \( \{i\}_{0 \leq i \leq n'} \).

If \( f \) is of the layer \( s \geq 2 \), then the bounds depend on positions \( p_0 \). In this case, replace \( p_0 \) with new indices \( j \) to obtain \( \{f(n, n', j) + i\}_{Q'(n, n', j)} \) where \( Q' \) abbreviates \( 0 \leq i \leq f(n, n', j) - f(n, n', j) \land D_2(n, j) \). Note that \( D_2(n, j) \) consists of disequations of the form \( 0 \leq j \leq m - 1 \) or \( 1 \leq j \leq m \). Replace \( m \) that belongs to the set \( p(n')(p_{n'} \ldots (p_{n-1}) \) with the already derived upper bound for this set. For instance, for \( \text{tails}_{x2}^{-}(n) \) we obtain \( (n - j)_{1 \leq j \leq n-1} \) on \( n \geq 1 \). The family is completed to \( (n - j)_{0 \leq j \leq n-1} \) by \( \text{tails}_{x2}^{-}(n)(0) = n \).

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 \( T(f(n')) \) “covers” a given multivalued function \( f \) amounts to checking if for any computation path for \( f(n')(p_0) \) the result will be in \( T(f(n')) \). In other words, for any rewriting rule \( D \vdash f(n')(p_0) \rightarrow p \) the following inclusion holds: \( D \vdash T(f(n')) \supseteq \text{range}(p) \), given that the range each higher-layer size variable \( n' \in n' \) is \( T(n') \).

Let \( n' \) be the list of the formal size parameters of \( g \) and \( n' \subseteq n' \) are first-layer variables. The table is constructed as follows.

\[ n(0) = n \]

20
if \( n^* \in \overline{n^*} \), where \( s \geq 2 \), then \( T(n^*) \) is given in the form \( \{p(\overline{n^*}, \overline{n^*}, i)\}_Q(\overline{n^*}, \overline{n^*}, i) \)

where \( \overline{n^*} \) are fresh first-layer size variables, and a polynomial \( p(\overline{n^*}, \overline{n^*}, i) \) and a predicate \( Q(\overline{n^*}, \overline{n^*}, i) \) are

- either given by a user,
- or are set by default to \( \{i\}_{0 \leq i \leq n^*_1} \) or \( \{i\}_{n^*_1 \leq i \leq n^*_2} \);

\( T(g(\overline{n^*})) \) has the form \( \{p(\overline{n^*}, \overline{n^*}, i)\}_Q(\overline{n^*}, \overline{n^*}, i) \). Note, that we treat higher-layer constants as functions, that is their specifications must be present in the table as well, in the form \( T(a) = \{p(\overline{a})\}_Q(\overline{a}) \). 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 optimisation in the presented work and leave it for the future.

For instance, the table \( T \), which is used to check the family \( \{i\}_{0 \leq i \leq n'} \) for \( \text{concat} \), contains \( T(++, n_1, n_2) = n_1 + n_2 \), \( T(M) = \{i\}_{0 \leq i \leq n'} \), \( T(\text{concat}, n, n') = \{i\}_{0 \leq i \leq n'} \).

Let \( n^* \) be the set of the free size variables of \( f \). Let \( \text{rhs}(f) \) denote the conditions from the rewriting rules defining \( f \). The set \( \text{rhs}(f) \) consists of memberships like \( m \in p(\overline{n^*}) \), position restrictions like \( 0 \leq \text{pos} \leq m - 1 \) (from the definition of type rewriting) or \( 1 \leq \text{pos} \leq m \) (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 \( g = f \) used in its definition are valid, \( \forall n^* \), \( \forall \text{pos} \) are s.t. \( f(\overline{n^*}, \overline{\text{pos}}) \) terminates, then \( \bigwedge \forall n^* \in \overline{n^*}, s \geq 2 \ n^* \in \overline{n^*} \subseteq T(n^*) \) implies \( f(\overline{n^*}, \overline{\text{pos}}) \subseteq T(f(\overline{n^*})) \).

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

Next, we define a *range map* \( [-] : SizeExpression \rightarrow IndexedPolynomial \times \text{1stOrderPredicate} \), where the first-order predicate in the image delimits the indices of the polynomial. Let \( \langle p_1 \rangle \) and \( \langle p_2 \rangle \) stay for the first projection (the polynomial) and the second projection (the predicate that bounds the indices) of \( \langle p \rangle \), resp. A correct range map \( \langle p \rangle \) 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 \( \langle a \rangle := \{a\} \);
- \( \langle a^s \rangle := T(a^s) \), where \( s \geq 2 \);
- for a first-layer variable \( n \) from the set of parameters \( \overline{n^*} \) the range map is defined as \( \langle n \rangle := \{n\} \);
- 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, \( \langle n^s \rangle := T(n^s) \);
- if \( \oplus \) is one of the arithmetic operations \( +, -, \star \), then
  \( \langle p_1 \oplus p_2 \rangle := \langle p_1 \rangle \oplus (\langle p_2 \rangle) \);
  \( \langle p(0) \rangle := \langle p \rangle \);
in a function call $g(p_1, \ldots, p_k, p'_1, \ldots, p'_{\ell'})$ we match the actual parameters with the formal parameters $\vec{n}, \vec{n}_g$ of the specification

$$T(g(\vec{n}_g)) = \{p(n_{g1}, \ldots, n_{gk}, \vec{n}_g, j)\}Q(n_{g}, n_{g'}, j)$$

First, note that since the function call terminates, then there must be a rewriting rule $D_g \vdash g(\vec{n}_g)(\text{pos}) \rightarrow p_g$ applicable for this call. From what follows that if we replace in $D_g$ the formal parameters $\vec{n}_g$ 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 $\langle p_1 \rangle_i$ of the first-layer actual parameters $p_1$, where $1 \leq l \leq k$;
2. after that we (inductively) compute the range sets $\langle p_2 \rangle_i$ of the higher-layer actual parameters $p_i$, where $1 \leq l \leq k'$;
3. after that the most difficult part of the matching “formal vs. actual parameters” is to be done: finding a substitution $\sigma : \text{FreshSizeVar} \rightarrow \text{IndexedPolynomial} \times 1\text{stOrderPredicate}$, such that for all formal $n_{gi}$, with $T(n_{gi}) = \{p''_i((\vec{n}_g, \vec{n}_g, j'))\}Q''(n_{g}, n_{g'}, j')$, the following inclusion must be provable from $D'_g$:

$$\langle p_1 \rangle_i \subseteq \{p''_i((p_1)_i, \ldots, (p_k)_i, \sigma_1(\vec{n}_g), j')\}Q''((p_1)_i, \ldots, (p_k)_i, \sigma_1(\vec{n}_g), j')\wedge$$

$$\bigwedge_{i=1}^k ((p_i)_i)_{i=1}^\forall \wedge \bigwedge_{i=1}^k \sigma_2(n_{g'}_{i})$$

For the sake of convenience we denote the last set via $\langle p'_1 \rangle_i$.

Finding a substitution $\sigma$ 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

$$\langle g(p_1,\ldots,p_k,p'_1,\ldots,p'_{\ell'}) \rangle :=$$

$$\{p((p_1)_i,\ldots,(p_k)_i, \sigma_1(\vec{n}_g), j)\}Q((p_1)_i,\ldots,(p_k)_i, \sigma_1(\vec{n}_g), j)\wedge$$

$$\bigwedge_{i=1}^k ((p_i)_i)_{i=1}^\forall \wedge \bigwedge_{i=1}^k \sigma_2(n_{g'}_{i})$$

Sometimes, for the sake of convenience, the polynomial $p$ and the delimiting predicate $Q$ form the specification $T(\text{program}(\vec{n})) = \{p(\vec{n}, \vec{n'}, \vec{v})\}Q(\vec{n}, \vec{n'}, \vec{v})$ are denoted via $\langle \text{program} \rangle_1$ and $\langle \text{program} \rangle_2$ respectively.

22
As an instance, consider the r.h.s. of the rewriting rule \( n > 1 \rightarrow \text{concat}(n, M) \rightarrow M(0) + \text{concat}(n-1, M+1) \).

\[
\begin{align*}
(M(0) + \text{concat}(n-1, M+1)) &= \\
(M(0)) + 1 \{\text{concat}(n-1, M+1)\} &= \\
(M + 1) \{\{\text{concat}(n-1, M+1)\} \sigma_1(n')\} &= \\
\{i\}_{0 \leq i < n'} + 1 \{i\}_{0 \leq i < (n-1)n'}
\end{align*}
\]

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 \( \tilde{m} \) are not always identities as in the example above. Consider the composition \( \text{concat}(\text{tails}(1)) \) with \( l \) be of the type \( L_n(\alpha) \). We want to check the rough but still sound estimate \( \text{concat} \circ \text{tails}(n) = \text{concat}(n, \text{tails}_2(n)) \). We already know that \( T(\text{concat}(n, M)) = \{i\}_{0 \leq i < n'} \) for \( T(M) = \{i\}_{0 \leq i < n'} \). Now we need to match \( T(M) \) with the annotation of the actual parameter \( \text{tails}_2(n)(\alpha) \). We know that \( T(\text{tails}_2(n)) = \{i\}_{0 \leq i < n} \), so we assume \( \sigma(n') = \{n\} \). Indeed, \( \{\text{tails}_2(n)\} = \{i\}_{0 \leq i \leq n} \subseteq \sigma(T(M)) = \{i\}_{0 \leq i \leq n'} \), thus \( \sigma \) is a valid substitution.

**Lemma 3 (Consistency of range map: basic).** Given an expression \( p_{\tilde{m}, \tilde{\tilde{m}}} \), if the specifications \( T(g(\tilde{m})) \) of all the functions \( g \) that occur in it are valid, then for all \( \tilde{m}, \tilde{\tilde{m}}, \tilde{\text{pos}} \), such that \( \bigwedge_{m \epsilon \tilde{m}, n \geq 2} n^{\tilde{m}}(\tilde{\text{pos}}) \subseteq T(n^{\tilde{m}}) \) and \( p_{\tilde{m}, \tilde{\text{pos}}}(\tilde{m}, \tilde{\text{pos}}) \) terminates, the inclusion \( p_{\tilde{m}, \tilde{\text{pos}}}(\tilde{m}, \tilde{\text{pos}}) \subseteq (p_{\tilde{m}, \tilde{\text{pos}}}) \) holds.

**Proof.** Fix \( \tilde{m}, \tilde{\text{pos}} \), such that \( p_{\tilde{m}, \tilde{\text{pos}}} \) terminates on them. The proof is done by induction on the structure of \( p_{\tilde{m}, \tilde{\text{pos}}} \).

- 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 \( p_{\tilde{m}, \tilde{\text{pos}}} = p_1 \tilde{m}, \tilde{\text{pos}} \oplus p_1 \tilde{m}, \tilde{\text{pos}} \). By induction assumption, \( p_1 \tilde{m}, \tilde{\text{pos}}(\tilde{m}, \tilde{\text{pos}}) \subseteq (p_1 \tilde{m}, \tilde{\text{pos}}) \), where \( l = 1, 2 \). From the definition

\[
p_{\tilde{m}, \tilde{\text{pos}}}(\tilde{m}, \tilde{\text{pos}}) := p_1 \tilde{m}, \tilde{\text{pos}}(\tilde{m}, \tilde{\text{pos}}) \oplus 1 p_2 \tilde{m}, \tilde{\text{pos}}(\tilde{m}, \tilde{\text{pos}})
\]

it follows that \( p_{\tilde{m}, \tilde{\text{pos}}}(\tilde{m}, \tilde{\text{pos}}) \subseteq (p_1 \tilde{m}, \tilde{\text{pos}}) \oplus 1 (p_2 \tilde{m}, \tilde{\text{pos}}) := (p_{\tilde{m}, \tilde{\text{pos}}}) \).
- Let \( p_{\tilde{m}, \tilde{\text{pos}}} = p_{\tilde{m}, \tilde{\text{pos}}}, \text{pos}(0) \). By induction assumption, \( p_{\tilde{m}, \tilde{\text{pos}}}, \text{pos}(\tilde{m}, \text{pos}) \subseteq (p_{\tilde{m}, \tilde{\text{pos}}}, \text{pos}) \). Therefore,

\[
p_{\tilde{m}, \tilde{\text{pos}}}(\tilde{m}, \text{pos}) = p_{\tilde{m}, \tilde{\text{pos}}}, \text{pos}(\tilde{m}, \text{pos}) \subseteq (p_{\tilde{m}, \tilde{\text{pos}}}, \text{pos}) = \text{def. of } \{-\}
\]

- The other cases, where \( p \) is an application of another size expression to a position, are treated similarly.
- Let \( p_{\tilde{m}, \tilde{\text{pos}}} = p_{\tilde{m}', \tilde{\text{pos}}, \tilde{\tilde{m}}+1} \) for some \( p' \), where \( \tilde{\text{pos}} = (\text{pos}, \tilde{\text{pos}}') \). Therefore, \( p_{\tilde{m}, \tilde{\text{pos}}}(\tilde{m}, \text{pos}) = p_{\tilde{m}', \tilde{\text{pos}}, \tilde{\text{pos}}'(\tilde{m}, \text{pos} + 1, \tilde{\text{pos}}')} \). According to the induction assumption, \( p_{\tilde{m}, \tilde{\text{pos}}, \tilde{\text{pos}}'}(\tilde{m}, \text{pos} + 1, \tilde{\text{pos}}') \subseteq (p_{\tilde{m}', \tilde{\text{pos}}, \tilde{\text{pos}}'}) \). According to the definition of \( \{-\} \), the last set is equal to \( (p_{\tilde{m}, \tilde{\text{pos}}, \tilde{\text{pos}}'} + 1) \), which is exactly \( (p_{\tilde{m}, \tilde{\text{pos}}, \tilde{\text{pos}}'}) \).
Consider the function call

\[
g(p_1, \ldots, p_k, p_1', \ldots, p_l') :=
g(p_1'(\bar{n}, \bar{pos}), \ldots, p_k'(\bar{n}, \bar{pos}), p_1'(\bar{n}, \bar{pos}), \ldots, p_l'(\bar{n}, \bar{pos}))
\]

According to the actual-parameter listing, a formal parameter \(n_p\) of \(g\) is instantiated with the actual parameter expressed by \(p_1'(\bar{n}, \bar{pos})\). The similar holds for the first-layer formal and corresponding actual parameters. Now we want to apply the validity of \(T(g(\bar{n}_p))\). Instantiate \(\bar{n}_p\) with \(\sigma(\bar{n}_p')\) from the definition of \(\langle \_ \rangle\) for the function call under consideration. Further, according to the induction assumption for the actual parameters \(p_1'(\bar{n}_p, \bar{pos})\) and the definition of \(\sigma\) we obtain

\[
p_1'(\bar{n}_p, \bar{pos}) \subseteq \langle p_1'(\bar{n}_p, \bar{pos}) \rangle \subseteq 
\{p_1'(\bar{n}_p, \bar{pos}) \mid \sigma(\bar{n}_p, J')\}
\]

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:

\[
[g(p_1, \ldots, p_k, p_1', \ldots, p_l')](\bar{n}, \bar{pos}) \subseteq 
\{p_1'(\bar{n}_p, \bar{pos}) \mid \sigma(\bar{n}_p, J')\} \cap 
\{p_1'(\bar{n}_p, \bar{pos}) \mid \sigma(\bar{n}_p, J')\} \cap 
\{p_1'(\bar{n}_p, \bar{pos}) \mid \sigma(\bar{n}_p, J')\}
\]

where the last set is exactly \(\langle g(p_1, \ldots, p_k, p_1', \ldots, p_l') \rangle\) according to the definition of \(\langle \_ \rangle\).

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

**Lemma 4 (Consistency of range map).** Given an expression \(p_{n^*, \bar{pos}}\), let the specifications \(T(g(\bar{n}_p))\) of all the functions \(g\) that occur in it be valid, except may be the specification \(T(f(\bar{n}_p))\) for \(f\), for which we do not know if it is valid or not. Let for each rewriting rule \(D \rightarrow f(\bar{n}) \rightarrow p_f\) the inclusion \(\langle D \rangle \supseteq \langle p_f \rangle\) holds. Then for all \(\bar{n}, \bar{n}_p, \bar{pos}\), such that \(\bigwedge_{n^*, n^*} T(n^*) \supseteq T(n^*)\) and \(p_{n^*, \bar{pos}}(\bar{n}, \bar{pos})\) terminates, the inclusion \(p_{n^*, \bar{pos}}(\bar{n}, \bar{pos}) \subseteq \langle p_{n^*, \bar{pos}} \rangle\) holds.

**Proof.** It is done by induction on the deepness of the recursion in the calls of \(f\) occurring in \(p_{n^*, \bar{pos}}(\bar{n}, \bar{pos})\).

- If the deepness \(d = 0\), 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
\[ f(p_1, \ldots, p_k, p'_1, \ldots, p'_{k'})[(\overline{n}, \overline{pos})] := f(p_1(\overline{n}, \overline{pos}), \ldots, p_k(\overline{n}, \overline{pos}), p'_1, \ldots, p'_{k'}(\overline{n}, \overline{pos})) \]

Since this call terminates, there must be a rule \( D \vdash T(f(\overline{n})) \rightarrow p_f \) applicable for the actual parameters of the call. According to the actual-parameter listing, a formal parameter \( n'_j \) is instantiated with the actual parameter expressed by \( p_j(n, \overline{pos}) \). 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
\[ f(p_1(\overline{n}, \overline{pos}), \ldots, p_k(\overline{n}, \overline{pos}), p'_1, \ldots, p'_{k'}(\overline{n}, \overline{pos})) = p_f(p_1(\overline{n}, \overline{pos}), \ldots, p_k(\overline{n}, \overline{pos})) \]

We may apply induction-on-the-deepness assumption, since the deepness of the recursive calls of \( f \) in \( p_f \) is one less than 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 \( \langle D \rangle \). Therefore we may apply the inclusion \( \langle D \rangle \vdash T(f(\overline{n})) \supseteq p_f \), more precisely, its instantiation with the first-layer actual parameters and \( \sigma(\overline{n}) \) for the fresh size variables, taken from the definition of \( \langle - \rangle \) for function calls. Thus, we obtain that
\[ p_f(p_1(\overline{n}, \overline{pos}), \ldots, p_k(\overline{n}, \overline{pos})) \subseteq \{ p_f(p_1, \ldots, p_k) \} \subseteq T(f(\overline{n})) \]

**Theorem 2 (Checking).** If all called in the definition of \( f \) functions \( g \neq f \) have valid specifications \( T(g(\overline{n})) \), and for each rule \( D \vdash f(\overline{n})(\overline{pos}_1) \ldots (\overline{pos}_{s-1}) \rightarrow p \) the inclusion \( \langle D \rangle \vdash T(f(\overline{n})) \supseteq \{ p \} \) holds then the specification \( T(f(\overline{n})) \) 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, \( D \vdash f(\overline{n})(\overline{pos}) \rightarrow p \). 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 as 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 procedure. 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 do 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


