Collected Size Semantics for Strict Functional Programs over General Polymorphic Lists

Olha Shkaravska[1(✉)], Marko van Eekelen[2,3], and Alejandro Tamalet[4]

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

Abstract. Size analysis can be an important part of 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. In this way the analysis becomes applicable for *general*, polymorphic nested lists.

1 Introduction

Bounds on the resource consumption of programs can be used, and are often needed, to ensure correctness and security properties, in particular in devices with scarce resources as mobile phones and smart cards. Both the memory and the time consumption of a program often depend on the sizes of input and intermediate data. In many cases, analysis of size behavior is an important part of resource analysis, e.g. using lower and upper bounds of size dependencies to check and infer non-linear bounds on heap consumption [17,20]. Here, we consider size analysis of *strict* first-order functional programs over polymorphic non-cyclic *lists*. Function signatures are predefined in a program. A size dependency of a program is a *size function* that maps the size of inputs onto the sizes of the corresponding output. By size of a list we mean the number of the elements in it. For instance, the typical size dependency for a program `append`, that appends two lists of length n and m, is the function $append(n, m) = n + m$.

This work is part of the AHA project [20] which is sponsored by the Netherlands Organisation for Scientific Research (NWO) under grant nr. 612.063.511. and by the CHARTER project, EU Artrmis nr. 100039.

U. Dal Lago and R. Peña (Eds.): FOPARA 2013, LNCS 8552, pp. 143–159, 2014.
DOI: 10.1007/978-3-319-12466-7_9

In the article [16] size functions were defined by conditional multiple-choice rewriting rules. However, the type system from that article did not cover general lists. It only covered 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 restricts application of the approach, since the case of programs over lists of lists with variate lengths is the most frequent one.

This paper is devoted to collecting size dependencies using *multivalued* size functions. We use multivalued size functions to annotate types making 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).

In this paper, we remove the restriction of [16] and generalise the approach to obtain a general solution that generates conditional rewriting rules for size dependencies of first-order polymorphic programs over lists, for which the size(s) of an output depends only on the sizes of inputs.

We use an ML-like strict first-order language which is defined in Sect. 2. In Sect. 3 we define the type system which allows *size variables of higher-order kinds*, such that, e.g., a size variable M in the type $\mathsf{L}_n(\mathsf{L}_M(\alpha))$ represents the size $M(pos)$ of an internal list depending on its position pos in the outer list, where $0 \leq pos \leq n-1$. Moreover, we extend (checking and inferring of) multivalued size functions defined by higher-order rewriting rules. We define soundness and sketch its proof. Section 4 relates our work to other resource analysis work.

It is not surprising that the presented research is inspired by sized-types and dependent-ML frameworks [5,13,23]. There sized types were used to prove termination and to assure other correctness properties of a program. Lists annotated by n denoted lists of length *at most* n. This is a limitation for size analysis since it makes upper bounds for decreasing size dependencies rather imprecise. For instance, with such interpretation of sized lists we would have to replace the dependency $n-m$, where n and m are annotations of input lists, just with n. For a better size analysis we initially wanted to have double-index annotations of the form $\mathsf{L}_{n_1 \leq length \leq n_2}(\alpha)$, that allows to express lower and upper bounds on the length of the lists. Later we discovered that inferring such lower and upper bounds, especially for nested lists, can be done if we infer lower and upper bounds of multivalued functions that represent all possible lengths of lists. The ratio behind this approach is explained in more detail and by examples in the technical report [15].

However higher-order rewriting systems may be considered not only as a tool for deriving lower and upper bounds for list lengths. The may be considered as standalone recurrence relations to be solved by known methods. By solving of a recurrence relation one means finding a closed form for the function it defines. Solving non-linear recurrence relations is a difficult problem that does not enjoy any universal approach. Solutions are provided for some classes of them. Two of the coauthors of the presented paper recently published a result about nonlinear polynomial recurrences [14]. The discussion about solving *size-analysis-related*

higher-order recurrences, e.g. by possible reduction them to first-order ones, is behind the scope of the presented paper.

Yet another argument for using higher-order rewriting rules for size dependencies is the prospect of an extension of the presented analysis to higher-order programs in the spirit of [9].

2 Language

The type system is designed for a strict first-order functional language over integers, booleans and (polymorphic) lists. Language expressions are defined by the grammar below where c ranges over integer and boolean constants False and True, x and y denote program variables of integer and boolean types, l ranges over lists, z denotes a program variable of a zero-order type, g ranges over higher-order program variables, unop is a unary operation, either $-$ or $\neg$, binop is one of the integer or boolean binary operations, and f denotes a function name.

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

The syntax distinguishes between let-binding of variables and letfun-binding of functions (function definitions). 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 $\mathsf{L}_n(\mathsf{L}_5(\alpha))$ where n is a first-layer expression.

- The *second layer* consists of 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 of the 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 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 $\lambda\, i_1 \ldots i_{s-1}.\, n^s(i_1)\ldots(i_{s-1})$ 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):

- Let $(\mathcal{R}^*)^k$ denote a k-fold product of $\mathcal{R}$. A function of the layer 1 is a function $f : (\mathcal{R}^*)^k \to 2^{\mathcal{R}}$ for some $k \geq 0$ that represents all possible sizes (depending on parameters from $(\mathcal{R}^*)^k$) of outer lists. For instance, if $f(n) = \{n,\, n+1\}$ in $\mathsf{l} : \mathsf{L}_{f(n)}(\alpha)$, then the length of l is either n or $n+1$.
  Another example is the output type of the function over a list of lists concatenating the elements of an input list: $\mathsf{concat} : \mathsf{L}_n(\mathsf{L}_M(\alpha)) \to \mathsf{L}_{\mathit{concat}(n,\,M)}(\alpha)$, where concat: $\mathcal{R}^{(1)} \times \mathcal{R}^{(2)} \to 2^{\mathcal{R}}$. In this case it is easy to spot the closed form $\mathit{concat}(n, M) = \sum_{i=0}^{n-1} M(i)$.
- 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 curly brackets in its value. As yet another example consider a function definition

$$\mathsf{tails} : \mathsf{L}_n(\alpha) \to \mathsf{L}_{\mathit{tails}_1(n)}(\mathsf{L}_{\mathit{tails}_2(n)}(\alpha))$$

that creates the list of all non-empty tails of the input list:

$$\begin{aligned} \mathsf{tails(l)} = \mathsf{match\ l\ with} \;&|\; \mathsf{Nil} \Rightarrow \mathsf{Nil} \\ &|\; \mathsf{Cons(hd,\ tl)} \Rightarrow \mathsf{let\ l'} = \mathsf{tails(tl)\ in\ Cons(l,\ l')} \end{aligned}$$

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

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

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

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:

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

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, whereas $\tau^{s',\,s}$ 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*:

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

The semantics of ground types is defined in Sect. 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 $B(pos) = pos$ on $0 \le pos \le 1$. 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:

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

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 are defined by conditional rewriting rules. It is desirable to find closed forms for functions defined by such rewriting rules. This is a topic of ongoing work.

A context Γ is a mapping from zero-order program variables to zero-order types. A signature Σ is a mapping from function names to function types. The definition of $SV(-)$ is straightforwardly extended to contexts: $SV(\Gamma) = \bigcup_{\mathsf{x}\in dom(\Gamma)} SV(\Gamma(\mathsf{x}))$.

3.1 Heap Semantics

In our semantic model, the purpose of the heap is to store lists.[1] 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:

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

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

$$c \models^h_{\mathsf{Int}\cup\mathsf{Bool}} c, \qquad \mathtt{NULL} \models^h_{\mathsf{L}_{p^1(\overline{n}_0^*)}(\tau^\bullet)} [] \ \text{ iff } \ 0 \in p^1(\overline{n}_0^*),$$

$$\ell \models^h_{\mathsf{L}_{p^1(\overline{n}_0^*)}(\tau^\bullet)} w_{\mathsf{hd}} :: w_{\mathsf{tl}} \text{ iff } \ h.\ell.\mathsf{hd} \models^{h|_{dom(h)\setminus\{\ell\}}}_{\tau^\bullet(0)} w_{\mathsf{hd}} \text{ and } h.\ell.\mathsf{tl} \models^{h|_{dom(h)\setminus\{\ell\}}}_{\mathsf{L}_{p^1(\overline{n}_0^*)-1}(\tau^\bullet_{+\mathtt{1}})} w_{\mathsf{tl}}$$

where $h|_{dom(h)\setminus\{\ell\}}$ denotes the heap equal to h everywhere except in ℓ, where it is undefined. Subtracting ℓ in the heaps of the recursive definition assures absence of cyclic lists since we exclude the reference to the same ℓ from the

[1] We introduce heap semantics to instrument the proof of the soundness of the typing system. As an anonymous reviewer pointed out, for the presented first-order setting without sharing it would be sufficient to use set-theoretical denotations in the operational-semantics rules to prove the soundness. However we plan to continue research in this direction for more general settings, where heap semantics may be an adequate model.

substructures. Next, $(p^s)_{+\mathbf{1}}$ 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:

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

3.2 Operational Semantics of Program Expressions

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

Using a heap, a frame store and mapping $\mathcal{C}$ (*closures*) from function names to function bodies, the operational semantics of program expressions is defined inductively in a standard way. Here we give some of the rules as examples. The full operational semantics is given in the technical report [15].

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

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

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

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

Note that in the function-application rule above, the function variables from the function definition do not become variables in $\mathcal{C}$, since $\mathcal{C}$ does not contain higher-order (function) variables because we deal with first-order programs. $\mathcal{C}$ contains only higher-order constants (functions). In the presented work function arguments in LetFun rule play a role of placeholders and is a syntactic sugar. To make it a classic first-order language one has to replace a "higher-order" function

definition of f with the collection of function definitions without higher-order placeholders, just enumerating all function instantiations that are possible (at the current call of f). To make it fully higher order, extension of the language in the spirit of [9] should be studied.

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.:

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

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

A notation $p \to p'$ means that $p = p'$ in the axiomatics of $\mathcal{R}$. An arrow is introduced because the type system is used to infer rewriting rules defining size functions. The inference process amounts to backward application of the typing rules, so that at the end of the process a size function appears on the left-hand side of the rewriting rules. Examples are given further.

The typing judgment relation is defined by the following rules:

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

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

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

where m is fresh in $D, \Gamma, \tau_1, \tau_2$.

Backward application of the Cons-rule to

$$n \geq 1;\ \mathsf{l}\colon \mathsf{L}_n(\alpha),\ \mathsf{l}'\colon \mathsf{L}_{\mathit{tails}_1(n-1)}(\mathsf{L}_{\mathit{tails}_2(n-1)}(\alpha)) \vdash_\Sigma \mathsf{Cons}(\mathsf{l}, \mathsf{l}')\colon \mathsf{L}_{\mathit{tails}_1(n)}(\mathsf{L}_{\mathit{tails}_2(n)}(\alpha))$$

allows to infer the rewriting rules for the sizes of the inner lists of the output for tails: $n \geq 1 \vdash \mathit{tails}_2(n)(0) \to n$ *and* $n \geq 1,\ 1 \leq \mathit{pos} \leq n-1 \vdash \mathit{tails}_2(n)(\mathit{pos}) \to \mathit{tails}_2(n-1)(\mathit{pos}-1)$.

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

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

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

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

where $m \notin SV(D)$. 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.

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

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

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 $\tau_\imath^\circ = \mathsf{L}(\ldots \mathsf{L}_{n^s}(\tau^{\circ\prime}) \ldots)$ and $\tau_\imath = \mathsf{L}(\ldots \mathsf{L}_{p_i^s(\overline{n}^*)}(\tau') \ldots)$, then $\sigma(n^s) := p_i^s(\overline{n}^*)$. If $\tau_\imath^\circ = \tau_{\imath'}^\circ$, then the corresponding size expressions are equal, that is C contains $p_i^s = p_{i'}^s$. Further, if $\tau_\imath^\circ = \mathsf{L}(\ldots \mathsf{L}_{a^s}(\tau^{\circ\prime}) \ldots)$, 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 $\mathsf{l}_1\colon \mathsf{L}_{n+1}(\mathsf{Int})$ and $\mathsf{l}_2\colon \mathsf{L}_{m-1}(\mathsf{Int})$. Then C contains $n+1 = m-1$. It will hold if D contains $n = m-2$.

Example 1: Inferring Rewriting Rules for concat. Consider the function concat, which given a list of lists appends all the inner lists:

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

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

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

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
   $n = 0$;
   $tv|\mathsf{L}_n(\mathsf{L}_M(\alpha)) \vdash_\Sigma \mathsf{Nil} \colon \mathsf{L}_{\mathit{concat}(n,M)}(\alpha)$.
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
   $n \geq 1;\ \mathsf{hd} : \mathsf{L}_M(\alpha)(0),\ \mathsf{tl} : \mathsf{L}_{n-1}(\mathsf{L}_M(\alpha)_{+\mathbf{1}}) \vdash_\Sigma \mathsf{append}(\mathsf{hd}, \mathsf{concat}(\mathsf{tl})) : \mathsf{L}_{\mathit{concat}(n,M)}(\alpha)$.
6. Unfold the definition of $(-)_{+\mathbf{1}}$:
   $n \geq 1;\ \mathsf{hd} : \mathsf{L}_{M(0)}(\alpha),\ \mathsf{tl} : \mathsf{L}_{n-1}(\mathsf{L}_{M_{+\mathbf{1}}}(\alpha)) \vdash_\Sigma \mathsf{append}(\mathsf{hd}, \mathsf{concat}(\mathsf{tl})) : \mathsf{L}_{\mathit{concat}(n,M)}(\alpha)$.
7. The expression in the judgment 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 $(-)_{+\mathbf{1}}$: $n \geq 1 \vdash \mathit{concat}(n, M) \to M(0) + \mathit{concat}(n-1,\ \lambda\ pos.\, M(pos+1))$.

Example 2: Inferring Rewriting Rules for tails. Now, we will infer the rewriting rules for the size annotations in the type

$$\mathsf{tails} : \mathsf{L}_n(\alpha) \to \mathsf{L}_{\mathit{tails}_1(n)}(\mathsf{L}_{\mathit{tails}_2(n)}(\alpha))$$

of the function tails as defined on page 3.

1. The LetFun rule defines the main goal: $\mathsf{l} : \mathsf{L}_n(\alpha) \vdash_\Sigma e_{\mathsf{tails}} : \mathsf{L}_{\mathit{tails}_1(n)}(\mathsf{L}_{\mathit{tails}_2(n)}(\alpha))$, where e_{tails} denotes the body of tails.
2. Apply the Match-rule. In the nil-branch we obtain the subgoal
   $n = 0$;
   $tv|\mathsf{L}_n(\alpha) \vdash_\Sigma \mathsf{Nil} : \mathsf{L}_{\mathit{tails}_1(n)}(\mathsf{L}_{\mathit{tails}_2(n)}(\alpha))$.
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 $n_1 \in \{n = 0\},\ 0 \le \mathit{pos} \le n_1 - 1$ is an empty set.
5. Now, consider the cons-branch. The subgoal there is
   $n \ge 1;\ \mathsf{l} : \mathsf{L}_n(\alpha),\ \mathsf{tl} : \mathsf{L}_{n-1}(\alpha_{+\mathbf{1}}) \vdash_\Sigma \mathsf{Cons}(\mathsf{l}, \mathsf{tails}(\mathsf{tl})) : \mathsf{L}_{\mathit{tails}_1(n)}(\mathsf{L}_{\mathit{tails}_2(n)}(\alpha))$.
6. In the type of tl unfold the definition of $(-)_{+\mathbf{1}}$:
   $n \ge 1;\ \mathsf{l} : \mathsf{L}_n(\alpha),\ \mathsf{tl} : \mathsf{L}_{n-1}(\alpha) \vdash_\Sigma \mathsf{Cons}(\mathsf{l}, \mathsf{tails}(\mathsf{tl})) : \mathsf{L}_{\mathit{tails}_1(n)}(\mathsf{L}_{\mathit{tails}_2(n)}(\alpha))$.
7. The expression in the judgment above is a sugared let-construct. So, we apply the Let-rule. In the binding we have the subgoal: $n \ge 1;\ \mathsf{tl} : \mathsf{L}_{n-1}(\alpha) \vdash_\Sigma \mathsf{tails}(\mathsf{tl}) : \tau^?$.
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
   $n \ge 1;\ \mathsf{l} : \mathsf{L}_n(\alpha),\ \mathsf{l}' : \mathsf{L}_{\mathit{tails}_1(n-1)}(\mathsf{L}_{\mathit{tails}_2(n-1)}(\alpha)) \vdash_\Sigma \mathsf{Cons}(\mathsf{hd}, \mathsf{l}') : \mathsf{L}_{\mathit{tails}_1(n)}(\mathsf{L}_{\mathit{tails}_2(n)}(\alpha))$.
10. Applying the Cons-rule we obtain

$$\begin{array}{ll} n \ge 1 & \vdash \mathsf{L}_{\mathit{tails}_1(n)}(\mathsf{L}_{\mathit{tails}_2(n)}(\alpha)) \to \mathsf{L}_{\mathit{tails}_1(n-1)+1}(\mathsf{L}_{\mathit{tails}_2(n)}(\alpha)) \\ n \ge 1 & \vdash (\mathsf{L}_{\mathit{tails}_2(n)}(\alpha))(0) \to \mathsf{L}_n(\alpha) \\ n \ge 1,\ 1 \le \mathit{pos} \le n & \vdash \mathsf{L}_{\mathit{tails}_2(n)}(\alpha)(\mathit{pos}) \to (\mathsf{L}_{\mathit{tails}_2(n-1)}(\alpha))(\mathit{pos} - 1) \end{array}$$

11. Unfold the definition of type-typewriting. For tails_1 we obtain $n \ge 1 \vdash \mathit{tails}_1(n) = \mathit{tails}_1(n-1) + 1$, and for tails_2 applied to 0 and to $\mathit{pos} \ge 1$ we obtain

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

    respectively.

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 $n \ge 1$. Further,

$\mathit{tails}_2(n)(pos) = n - pos$ for $0 \leq pos \leq n-1$ solves the rewriting system for g. Indeed, by induction on $n \geq 2$, $\mathit{tails}_2(n)(pos) = \mathit{tails}_2(n-1)(pos-1) = (n-1)-(pos-1) = n - pos$ for $pos \geq 1$, with the base $\mathit{tails}_2(1)(0) = 1$, and having $\mathit{tails}_2(n)(pos) = n$ for $pos = 0$.

3.4 Semantics of Typing Judgments (Soundness)

The set-theoretic semantics of typing judgments 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 application of η and ϵ to types τ and size relations D as subscripts. For example, $\eta(\epsilon(\tau))$ becomes $\tau_{\eta\epsilon}$ and $\epsilon(D)$ becomes D_ϵ. Note that D contains no type variables and hence $D_\eta = D$. Valuations and instantiations distribute over size functions in the following way: $(\mathsf{L}_{p(\overline{n}^*)}(\tau))_{\eta\epsilon} = \mathsf{L}_{p(\overline{n}^*_\epsilon)}(\tau_{\eta\epsilon})$.

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

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

- $dom(s) = dom(\Gamma)$, $\epsilon\colon SV(\Gamma) \cup SV(D) \to \mathcal{R}$ *and* $\eta\colon TV(\Gamma) \to \tau^\bullet$,
- *the expression e when evaluated in h evaluates to v, and the new heap is h', i.e., $s; h;\ \mathcal{C} \vdash e \rightsquigarrow v; h'$,*
- D_ϵ *holds,*
- $D, \Gamma \vdash_\Sigma e\colon \tau'$,
- $\mathit{Valid}_{\mathsf{store}}(dom(s), \Gamma_{\eta\epsilon}, s, h)$,

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

Proof. The proof is done by induction on the size of the derivation tree for the operational-semantic judgment. This is possible because we assume that the evaluation of e terminates (with a value v). We have to prove $\mathit{Valid}_{\mathsf{val}}(v, \tau'_{\eta\epsilon}, h')$, i.e., that there is a w such that $v \models^{h'}_{\tau'_{\eta\epsilon}} w$. In the proof we omit some technical lemmata and easier cases. They can be found in the technical report.

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, we have $D \vdash \tau' \to \tau$. Since D_ϵ holds, we 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 $\mathsf{hd}\colon \tau_1$ and $\mathsf{tl}\colon \mathsf{L}_{p^1(\overline{n}^*)}(\tau_2)$, and for some τ'_2 the judgments $D \vdash \tau' \to \mathsf{L}_{p^1(\overline{n}^*)+1}(\tau'_2)$, $D \vdash \tau'_2(0) \to \tau_1$ and $n \in p^1(\overline{n}^*), 1 \le pos \le n, D \vdash \tau'_2(pos) \to \tau_2(pos-1)$ hold. 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

$$h'.\ell.tl \models^{h}_{\mathsf{L}_{p^1(\overline{n}^*_\epsilon)}(\tau_{2\eta\epsilon})} w_{tl}.$$

By the definition of h' we have $h = h'|_{dom(h')\setminus\{\ell\}}$, 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 judgment $D \vdash \tau'_2(0) \to \tau_1$ we obtain $h'.\ell.hd \models^{h'|_{dom(h')\setminus\{\ell\}}}_{\tau'_{2\eta\epsilon}(0)} w_{hd}$. Now we want to show that

$$h'.\ell.tl \models^{h'|_{dom(h')\setminus\{\ell\}}}_{\mathsf{L}_{p^1(\overline{n}^*_\epsilon)}(\tau'_{2\eta\epsilon_{+1}})} w_{tl},$$

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 judgment $n \in p^1(\overline{n}^*), 1 \le pos \le n, D \vdash \tau'_2(pos) \to \tau_2(pos-1)$ is equivalent to $n \in p^1(\overline{n}^*), 0 \le pos \le n-1, D \vdash \tau'_2(pos+1) \to \tau_2(pos)$. Recall that τ_{+1} is defined as $\lambda\, pos.\ \tau(pos+1)$, thus we have the judgment $n \in p^1(\overline{n}^*), 0 \le pos \le n-1, D \vdash (\tau'_2)_{+1}(pos) \to \tau_2(pos)$. Now by definition of rewriting rules for types we have $D \vdash \mathsf{L}_{p^1(\overline{n}^*)}((\tau'_2)_{+1}) \to \mathsf{L}_{p^1(\overline{n}^*)}(\tau_2)$. Using $h'.\ell.tl \models^{h'|_{dom(h')\setminus\{\ell\}}}_{\mathsf{L}_{p^1(\overline{n}^*_\epsilon)}(\tau_{2\eta\epsilon})} w_{tl}$, we get $h'.\ell.tl \models^{h'|_{dom(h')\setminus\{\ell\}}}_{\mathsf{L}_{p^1(\overline{n}^*_\epsilon)}(\tau'_{2\eta\epsilon_{+1}})} w_{tl}$.

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, $\Gamma, \mathsf{z}\colon \tau \vdash_\Sigma e_2 : \tau'$. Applying the induction hypothesis to the antecedents of the operational semantics, we get that $Valid_{\mathsf{val}}(v_1, \tau_{\eta\epsilon}, h_1)$ and that if $Valid_{\mathsf{store}}(dom(s[\mathsf{z} := v_1]), \Gamma_{\eta\epsilon} \cup \{\mathsf{z}\colon \tau_{\eta\epsilon}\}, s[\mathsf{z} := v_1], h_1)$ then $Valid_{\mathsf{val}}(v, \tau'_{\eta\epsilon}, h')$.

Fix some $\mathsf{z}' \in dom(s[\mathsf{z} := v_1])$. If $\mathsf{z}' = \mathsf{z}$, then $Valid_{\mathsf{val}}(v_1, \tau_{\eta\epsilon}, h_1)$ implies $Valid_{\mathsf{val}}(s[\mathsf{z} := v_1](\mathsf{z}), \tau_{\eta\epsilon}, h_1)$. 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 $s(\mathsf{z}') \models^{h}_{\Gamma_{\eta\epsilon}(\mathsf{z}')} w'_z$ 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, $Valid_{\mathsf{val}}(s[\mathsf{z} := v_1](\mathsf{z}'), \Gamma_{\eta\epsilon}(\mathsf{z}'), h_1)$. Hence, $Valid_{\mathsf{store}}(dom(s[\mathsf{z} := v_1]), \Gamma_{\eta\epsilon} \cup \{\mathsf{z}\colon \tau_{\eta\epsilon}\}, s[\mathsf{z} := v_1], h_1)$ and we can now apply the induction hypothesis.

OSMatch-Cons: In this case $e = \mathsf{match\ l\ with\ |\ Nil} \Rightarrow e_1 \mid \mathsf{Cons(hd, tl)} \Rightarrow e_2$. for some l, hd, tl, e_1 and e_2. The typing context has the form $\Gamma = \Gamma' \cup \{\mathsf{l} : \mathsf{L}_{p^1(\overline{n}^*)}(\tau)\}$ for some Γ', τ and f. From the operational semantics we know that $h.s(\mathsf{l}).hd = v_{hd}$ and $h.s(\mathsf{l}).tl = v_{tl}$, for some v_{hd} and v_{tl}, that is, $s(\mathsf{l}) \neq \mathtt{NULL}$. Due to the validity of $s(\mathsf{l})$ there exists $1 \leq n_0 \in p^1(\overline{n}^*_\epsilon)$. 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 $D, n_0 \geq 1 \in p^1(\overline{n}^*)$; $\Gamma'' \vdash_\Sigma e_2 : \tau'_{\eta\epsilon}$, where $\Gamma'' = \Gamma \cup \{\mathsf{hd} : \tau(0), \mathsf{tl} : \mathsf{L}_{p^1(\overline{n}^*)-1}(\tau_{+\mathbf{1}})\}$.

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 $n_0 \notin SV(\tau')$ (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 $\Gamma \vdash_\Sigma e_2 : \tau'$. Applying the induction hypothesis to these judgments with the same η and ϵ, we obtain $\mathit{Valid}_{\mathsf{val}}(v, \tau'_{\eta\epsilon}, h')$ as desired.

OSFunApp: In this case $e = \mathsf{f}(\mathsf{f}_1, \ldots, \mathsf{f}_{k'}, \mathsf{z}'_1, \ldots, \mathsf{z}'_k)$, with $\mathcal{C}(\mathsf{f}) = (\mathsf{g}_1, \ldots, \mathsf{g}_{k'}, \mathsf{z}'_1, \ldots, \mathsf{z}'_k) \times e_1)$ 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 hypothesis to this judgment.

Since all functions called in e are defined via letfun, there must be a node in the derivation tree of the original typing judgment of the form $\mathsf{True},\ y_1 : \tau^\circ, \ldots, y_k : \tau^\circ_k \vdash_\Sigma e_f : \tau_0$. Let σ be the substitution over type variables and size variables that ensures this instance of the FUNAPP-rule. In particular, $D \vdash \tau \to \sigma(\tau_0)$. Take η' and ϵ' such that $\eta'(\alpha) = \eta\big(\sigma(\alpha)\big)$, and $\epsilon'(n^*_{ij}) = \epsilon\big(\sigma(n^*_{ij})\big)$. 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

$$\begin{aligned}
&\tau'_{\eta\epsilon} = \big(\sigma(\tau_0)\big)_{\eta\epsilon} = \\
&\tau_0[\ldots\alpha := \eta\big(\sigma(\alpha)\big)\ldots][\ldots n^*_{ij} := \epsilon\big(\sigma(n^*_{ij})\big)\ldots] = \\
&\tau_{0\,\eta'\epsilon'}
\end{aligned}$$

Therefore from obtain $\mathit{Valid}_{\mathsf{val}}(v, \sigma(\tau_{0\,\eta'\epsilon'}), h')$ we obtain $\mathit{Valid}_{\mathsf{val}}(v, \tau'_{\eta\epsilon}, h')$.

□

4 Related Work

This research extends our work [17–19,21] about shapely function definitions that have a single-valued, exact input-output polynomial size functions. Our non-monotonic framework resembles [1] in which the authors describe *monotonic* resource consumption for Java bytecode by means of Cost Equation Systems (CESs), which are similar to, but more general than recurrence equations. CESs express the cost of a program in terms of the size of its input data. In a further step, a closed-form solution or upper bound can be found by using existing Computer Algebra Systems, or a by a specially designed by the authors recurrence solver. However, they consider non-monotonic size functions only if they are linear, e.g. $s(x, y) = x - y$.

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

Hofmann and Jost 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 [6], and recently as max-plus expressions on the depth of data structures [7]. Again, the main difference with our work is that we not require linear size functions. Recently, this analysis has been extended to include multivariate non-linear resource polynomials [10]. The difference with our work is that we also allow general polynomials including non-monotonic polynomials like $x^2 - y^2$ where [10] allows only non-negative linear combinations of base polynomials.

In his thesis, Pedro Vasconcelos [22] 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 [3,8]. 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.

5 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 generalising size analysis on algebraic data types and implementation work incorporating the

results of this paper in our polynomial size analysis prototype (see `resourceanalysis.cs.ru.nl/#prototypes`).

References

1. Albert, E., Arenas, P., Genaim, S., Puebla, G., Zanardini, D.: Cost analysis of java bytecode. In: De Nicola, R. (ed.) ESOP 2007. LNCS, vol. 4421, pp. 157–172. Springer, Heidelberg (2007)
2. Amadio, R.M.: Synthesis of max-plus quasi-interpretations. Fundam. Informaticae **65**(1–2), 29–60 (2004)
3. Atassi, V., Baillot, P., Terui, K.: Verification of ptime reducibility for system F terms: type inference in dual light affine logic. Logical Meth. Comput. Sci. **3**(4), 1–32 (2007)
4. Bonfante, G., Marion, J.Y., Moyen, J.Y.: Quasi-interpretations a way to control resources. Theor. Comput. Sci. **412**(25), 2776–2796 (2011)
5. Reistadand, B., Gifford, D.K.: Static dependent costs for estimating execution time. In: Proceedings of the Conference on Lisp and Functional Programming FP'94, pp. 65–78. ACM Press, January 1994
6. Campbell, B.: Space cost analysis using sized types. Ph.D. thesis, School of Informatics, University of Edinburgh (2008)
7. Campbell, B.: Amortised memory analysis using the depth of data structures. In: Castagna, G. (ed.) ESOP 2009. LNCS, vol. 5502, pp. 190–204. Springer, Heidelberg (2009)
8. Gaboardi, M., Marion, J.-Y., Ronchi Della Rocca, S.: A logical account of PSPACE. In: Proceedings of the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages POPL 2008, San Francisco, January 10–12, 2008, pp. 121–131 (2008)
9. Góbi, A., Shkaravska, O., van Eekelen, M.: Higher-order size checking without subtyping. In: Loidl, H.-W., Peña, R. (eds.) TFP 2012. LNCS, vol. 7829, pp. 53–68. Springer, Heidelberg (2013)
10. Hoffmann, J., Aehlig, K., Hofmann, M.: Multivariate amortized resource analysis. ACM Trans. Program. Lang. Syst. **34**(3), 14:1–14:62 (2012)
11. Hofmann, M., Jost, S.: Static prediction of heap space usage for first-order functional programs. SIGPLAN Not. **38**(1), 185–197 (2003)
12. Hofmann, M., Jost, S.: Type-based amortised heap-space analysis. In: Sestoft, P. (ed.) ESOP 2006. LNCS, vol. 3924, pp. 22–37. Springer, Heidelberg (2006)
13. Hughes, L.P.J., Sabry, A.: Proving the correctness of reactive systems using sized types. In: Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '96, pp. 410–423. ACM (1996)
14. Shkaravska, O., van Eekelen, M.: Univariate polynomial solutions of algebraic difference equations. J. Symbolic Comput. **60**, 15–28 (2014)
15. Shkaravska, O., van Eekelen, M., Tamalet, A.: Collected size semantics for functional programs over polymorphic nested lists. Technical report ICIS-R09003, Radboud University Nijmegen, July 2009
16. Shkaravska, O., van Eekelen, M., Tamalet, A.: Collected size semantics for functional programs over lists. In: Scholz, S.-B., Chitil, O. (eds.) IFL 2008. LNCS, vol. 5836, pp. 118–137. Springer, Heidelberg (2011)
17. Shkaravska, O., van Eekelen, M.C.J.D., van Kesteren, R.: Polynomial size analysis of first-order shapely functions. Log. Meth. Comput. Sci. **5**(2), 1–35 (2009)

18. Shkaravska, O., van Kesteren, R., van Eekelen, M.: Polynomial size analysis of first-order functions. In: Della Rocca, S.R. (ed.) TLCA 2007. LNCS, vol. 4583, pp. 351–365. Springer, Heidelberg (2007)
19. Tamalet, A., Shkaravska, O., van Eekelen, M.: Size analysis of algebraic data types. In: Achten, P., Koopman, P., Morazán, M. (eds.) Trends in Functional Programming, TFP'08, vol. 9. Intellect Publishers (2009)
20. van Eekelen, M., Shkaravska, O., van Kesteren, R., Jacobs, B., Poll, E., Smetsers, S.: AHA: amortized heap space usage analysis. In: Morazán, M. (ed.) Selected Papers of the 8th International Symposium on Trends in Functional Programming (TFP'07), pp. 36–53. Intellect Publishers, New York (2007)
21. van Kesteren, R., Shkaravska, O., van Eekelen, M.: Inferring static non-monotonically sized types through testing. In: Proceedings of 16th International Workshop on Functional and (Constraint) Logic Programming (WFLP'07). ENTCS, Paris, France, vol. 216C, pp. 45–63 (2007)
22. Vasconcelos, P.B.: Space cost analysis using sized types. Ph.D. thesis, School of Computer Science, University of St. Andrews, August 2008
23. Xi, H., Pfenning, F.: Dependent types in practical programming. In: Proceedings of the 26th ACM SIGPLAN Symposium on Principles of Programming Languages, San Antonio, pp. 214–227, January 1999