Collected Size Semantics for Functional Programs *

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

Institute for Computing and Information Sciences
Radboud University Nijmegen

Abstract. This work introduces collected size semantics of strict functional programs over lists. It is presented via non-deterministic numerical functions annotating types. These functions are defined by conditional rewriting rules generated during type inference.
We focus on the connection between the size rewriting rules and lower and upper bounds on size dependencies, where the bounds are given by polynomials extended with the $\max_0$-operation. We show how, given a set of conditional rewriting rules, one can infer bounds that define an indexed family of $\max_0$-polynomials that approximates (from above) the non-deterministic size dependency.
Using collected size semantics we are able to infer non-monotonic and non-linear lower and upper bounds for many functional programs. As a feasibility study we consider the Haskell list library.

1 Introduction

Estimating heap consumption is an active research area as it becomes more and more of an issue in many applications, including programming for small devices, e.g. smart cards, mobile phones, embedded systems and distributed computing. This work explores typing support for checking output-on-input size dependencies for function definitions (functions for short) in a strict functional language. We allow higher-order functions, but we consider only those where the size of the output depends just on zero-order arguments. Size dependencies are presented via non-deterministic numerical functions defined by conditional non-deterministic rewriting rules. Numerical functions annotate types and the rules for them are generated during type inference. Since one is mostly interested in lower and upper bounds for size dependencies, we study connections between size rewriting rules and size bounds. We focus on bounds that are given by polynomials extended with the operation $\max_0(p) = \max(0, p)$ ($\max_0$-polynomials). Given a set of conditional non-deterministic size rewriting rules we show how to infer lower and upper bounds that define an indexed family of polynomials which fully covers the size dependency induced by the rewriting rules. Indices are delimited by quantifier-free first-order arithmetic predicates.

* This work is part of the AHA project [13] which is sponsored by the Netherlands Organisation for Scientific Research (NWO) under grant nr. 612.063.511.

1.1 Exploring size dependencies

We restrict our attention to a language with polymorphic lists as the only data type. For this language we develop a sound size-aware type system where types are annotated with non-deterministic numerical functions. Given a function definition, the type inference procedure will derive non-deterministic rewriting rules taking into account the various size dependencies of the function. The size rewriting rules define a *non-deterministic size function*. We study approximations from above for such non-deterministic size functions. An indexed family of deterministic functions $\{g(\overline{n},\overline{i})\}_{\overline{i}\in I}$ *approximates* a non-deterministic size function f from above if and only if for any input size $\overline{n}$ and for any output size $m \in f(\overline{n})$, there is an index $\overline{i} \in I$ such that $g(\overline{n},\overline{i}) = m$.

We show how one can *check* whether a given family approximates a given non-deterministic size function. We also show how one can *infer* such a family.

To get a global idea of the kind of results we expect from our size analysis, consider a function delete which deletes from a list the first occurrence of an element that satisfies a given predicate:

```
delete(g, z, l) =
  match l with | Nil ⇒ Nil
               | Cons(hd, tl) ⇒ if g(z, hd) then tl
                                else let z′ = delete(g, z, tl) in Cons(hd, z′)
```

The size of the result may vary between the size n of the list and $n-1$, but it must be always greater than 0. The corresponding non-deterministic size function is given by the set of rewriting rules

$$\begin{array}{r} \vdash f_{\mathsf{delete}}(0) \to 0 \\ n \geq 1 \vdash f_{\mathsf{delete}}(n) \to n-1 \,|\, 1 + f_{\mathsf{delete}}(n-1) \end{array}$$

The (tight) lower and upper bounds are $n-1$ and n. The corresponding approximating family of polynomials is $\{\max_0(n-i)\}_{0\leq i\leq 1}$ [1].

In our setting, many size-bounded functions are well-typed. Most functions in the Haskell list library can be assigned a sized type that precisely describes the possible output sizes in a strict context (see Section 5).

1.2 Our contribution and contents of the paper

We use non-deterministic size functions as annotations and consider families of $\max_0$-polynomials as approximations for size dependencies. We show that size functions defined by rewriting rules are very convenient to *infer* approximating families. Checking if a given family approximates a given size dependency is shown to be similar to type checking types annotated with indexed families [10]. Here, we give an inference procedure and discuss examples.

[1] In classical recursion theory $\max_0(n-m)$ is expressed as $n \dot{-} m$. Using $\dot{-}$ is equivalent to using max and min operations. Indeed, $\max(m,\, n) = m + (n \dot{-} m)$ and $\min(m,\, n) = m \dot{-} (m \dot{-} n)$.

This paper is organised as follows. In Section 2 we define the programming language and in Section 3 its size-aware type system. Section 3 also defines the semantics of program values w.r.t. zero-order types and the operational semantics of the language. In Section 5 we discuss the feasibility of applying the analysis to the Haskell list library. Approximation by families of $\max_0$-polynomials is discussed in Section 4 and related work in Section 6. Section 7 draws conclusions and gives directions to future work.

For the interested reader the technical report [9] gives more examples of checking and inference in detail.

2 Language

The type system is designed for a strict functional language over integers and (polymorphic) lists. Algebraic data types could be added as we did in [12]. Language expressions are defined by the following grammar:

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

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. Variables may be decorated with sub- and superscripts.

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

3 Type System

We consider a type system constituted from zero-order and higher-order types and typing rules corresponding to program constructs. Size annotations are multivariate (non-deterministic) numerical functions $f\colon \mathcal{R}^n \to 2^{\mathcal{R}}$ that represent lengths of finite lists and arithmetic operations over these lengths. $\mathcal{R}$ can be any

numerical ring; its choice influences decidability of type checking and the set of well-typed programs.

Zero-order types are assigned to program values, which are integers, booleans and finite lists. The list type is annotated by a size function that represents the possible lengths of the list:

$$\textit{Types}\ \tau ::= \mathsf{Int} \mid \mathsf{Bool} \mid \alpha \mid \mathsf{L}_{f(\overline{n})}(\tau),$$

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

```
insert(g, x, l) =
  match l with | Nil ⇒ let l′ = Nil in Cons(x, l′)
               | Cons(hd, tl) ⇒ if g(x, hd) then l else
                                  let l″ = insert(g, x, tl) in Cons(hd, l″)
```

The corresponding size rewriting system is

$$\begin{array}{l} \vdash f_{\mathsf{insert}}(0) \to 1 \\ n \geq 1 \vdash f_{\mathsf{insert}}(n) \to\ n \mid 1 + f_{\mathsf{insert}}(n-1) \end{array}$$

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

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

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

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

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

Let τ° denote a zero-order type where size expressions are all size variables or constants, like, e.g., $\mathsf{L}_n(\alpha)$. 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 $\tau_1^\circ, \ldots, \tau_k^\circ$. Consider, for instance, the function definition for $\mathsf{filter} : (\alpha \to \mathsf{Bool}) \times \mathsf{L}_n(\alpha) \to \mathsf{L}_{f_{\mathsf{filter}}(n)}(\alpha)$

```
filter(g, x) =
  match x with | Nil ⇒ Nil
               | Cons(hd, tl) ⇒ if g(hd) then let z = filter(g, tl) in Cons(hd, z)
                                else filter(g, tl)
```

The annotating function f_{filter} is defined by

$$\begin{array}{l} \vdash f_{\mathsf{filter}}(0) = 0 \\ n \geq 1 \vdash f_{\mathsf{filter}}(n) = \ 1 + f_{\mathsf{filter}}(n-1) \mid f_{\mathsf{filter}}(n-1) \end{array}$$

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

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

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

3.1 Semantics of zero-order types

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

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

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

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

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

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

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

$$\mathit{length} : \mathit{Heap} \rightharpoonup \mathit{Address} \rightharpoonup \mathcal{N}$$
$$\mathit{length}_h(\texttt{NULL}) = 0 \qquad \mathit{length}_h(\ell) = 1 + \mathit{length}_{h|_{dom(h)\setminus\{\ell\}}}(h.\ell.\mathsf{tl})$$

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

Lemma 1 (Consistency of model relation).
The relation $\texttt{adr} \models^{h}_{\mathsf{L}_{f(\overline{n}_0)(\tau^\bullet)}} w$ *implies that* $\mathit{length}_h(\texttt{adr}) \in f(\overline{n}_0)$.

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

3.2 Operational semantics of program expressions

The operational semantics is standard. It extends the semantics from [11] 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, $\mathit{Store}\ s : \mathit{ProgramVars} \rightharpoonup \mathit{Val}$.

Using heaps and a frame store and maintaining a mapping $\mathcal{C}$ of *closures*, from function names to the bodies of the function definitions, the operational semantics is defined by the following rules:

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

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

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

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

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

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

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

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

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

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

3.3 Typing rules

A typing judgement is a relation of the form $D, \Gamma \vdash_\Sigma e : \tau$, where D is a conjunction of statements of the form $0 \in f(\overline{n})$ and $m \geq 1 \in f(\overline{n})$. The signature Σ contains type assumptions for all the functions involved.

Given types $\tau = \mathsf{L}_{f_1(\overline{n})}(\ldots \mathsf{L}_{f_k(\overline{n})}(\alpha)\ldots)$ and $\tau' = \mathsf{L}_{f'_1(\overline{n})}(\ldots \mathsf{L}_{f'_k(\overline{n})}(\alpha)\ldots)$, let the entailment $D \vdash \tau \to \tau'$ abbreviate the collection of rewriting rules

$$\{D, n_1 \geq 1 \in f_1(\overline{n}), \ldots,\ n_{l-1} \geq 1 \in f_{l-1}(\overline{n}) \vdash f_l(\overline{n}) \to f'_l(\overline{n})\}_{1 \leq l \leq k}$$

where $n_1, \ldots, n_{l-1} \notin D$ denotes that there exists some positive values in $f_1(\overline{n})$, ... , $l-1(\overline{n})$, respectively.

The typing judgement relation is defined by the following rules.

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

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

$$\frac{D \vdash \tau' \to \mathsf{L}_{f(\overline{n})+1}(\tau_2) \qquad D \vdash \tau_2 \to \tau_1}{D,\ \Gamma,\ \mathsf{hd}\colon \tau_1,\ \mathsf{tl}\colon \mathsf{L}_{f(\overline{n})}(\tau_2) \vdash_\Sigma \mathsf{Cons(hd,\ tl)}\colon \tau'}\ \textsc{Cons}$$

$$\frac{\Gamma(\mathsf{x}) = \mathsf{Int} \qquad D,\ \Gamma \vdash_\Sigma e_t\colon \tau_1 \qquad D,\ \Gamma \vdash_\Sigma e_f\colon \tau_2}{D,\ \Gamma \vdash_\Sigma \mathsf{if\ x\ then}\ e_t\ \mathsf{else}\ e_f\colon \tau_1 \mid \tau_2}\ \textsc{If}$$

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

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

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

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

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

The function application rule computes a substitution σ from the formal size variables to the actual size expressions, and a set C of equations collecting restrictions on the actual input types. These restrictions are of the form $\tau \equiv \tau'$ abbreviating equality of the corresponding underlying types and annotations. The equation $\tau \equiv \tau'$ belongs to C if τ and τ' are substituted into to the same type, like, for instance, in the call to the function $\mathsf{scalarprod} : \mathsf{L}_m(\mathsf{Int}) \times \mathsf{L}_m(\mathsf{Int}) \to \mathsf{Int}$ with actual parameters $\mathsf{l}\colon \tau$ and $\mathsf{l}'\colon \tau'$. To see how the substitution σ is applied consider a formal size parameter m with $\sigma(m) = f'(\overline{n})$. Then

$$\sigma\Big(\mathsf{L}(\ldots \mathsf{L}_{f(m)}(\ldots \mathsf{L}(\alpha)\ldots)\ldots)\Big) = \mathsf{L}(\ldots \mathsf{L}_{f(f'(\overline{n}))}(\ldots \mathsf{L}(\alpha)\ldots)\ldots)$$

As a slightly more involving example, consider a function rel that produces all pairs of elements from two argument lists that are related to each other according to a given predicate. For instance $\mathsf{rel}(>,\ [2,\ 3,\ 5],\ [2,\ 3]) = [[3,\ 2],\ [5,\ 2],\ [5,\ 3]]$. This function calls an auxiliary function rel_pairs, that given a single element x and a list, produces the list of all pairs $(x,\ y)$ of the related elements, where y runs over the list. The definitions for rel and rel_pairs are

```
rel(g, l1, l2) = match l1 with | Nil ⇒ Nil
                               | Cons(hd, tl) ⇒ append(rel_pairs(g, hd, l2), rel(g, tl, l2))

and rel_pairs(g, x, l) = match l with | Nil ⇒ Nil
                                      | Cons(hd, tl) ⇒ if g(x, hd) then
                                                         Cons(Cons(x, Cons(hd, Nil)),
                                                              rel_pairs(x, tl))
                                                       else rel_pairs(x, tl)
```

The types are $(\alpha \to \alpha \to \mathsf{Bool}) \times \mathsf{L}_n(\alpha) \times \mathsf{L}_m(\alpha) \to \mathsf{L}_{f_{\mathsf{rel}_1}}(\mathsf{L}_{f_{\mathsf{rel}_2}}(\alpha))$ and $(\alpha \to \alpha \to \mathsf{Bool}) \times \alpha \times \mathsf{L}_m(\alpha) \to \mathsf{L}_{f_{\mathsf{rel\_pairs}_1}}(\mathsf{L}_{f_{\mathsf{rel\_pairs}_2}}(\alpha))$, respectively. The size dependencies for rel are given by the rewriting rules

$$\vdash f_{\mathsf{rel}_1}(0,m) \to 0 \qquad \vdash f_{\mathsf{rel}_2}(0,m) \to \mathsf{undefined}$$
$$n \geq 1 \vdash f_{\mathsf{rel}_1}(n,m) \to f_{\mathsf{rel\_pairs}_1}(m) + f_{\mathsf{rel}_1}(n-1,m) \quad n \geq 1 \vdash f_{\mathsf{rel}_2}(n,m) \to f_{\mathsf{rel\_pairs}_2}(m)$$

where
$$\vdash f_{\mathsf{rel\_pairs}_1}(0) \to 0$$
$$m \geq 1 \vdash f_{\mathsf{rel\_pairs}_1}(m) \to 1 + f_{\mathsf{rel\_pairs}_1}(m-1) \,|\, f_{\mathsf{rel\_pairs}_1}(m-1)$$

and
$$\vdash f_{\mathsf{rel\_pairs}_2}(0) \to \mathsf{undefined}$$
$$m \geq 1 \vdash f_{\mathsf{rel\_pairs}_2}(m) \to 2$$

3.4 Semantics of typing judgements (soundness)

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

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

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

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

Valuations and instantiations distribute over annotations in the following way: $\eta\epsilon((\mathsf{L}_{f(\overline{n})}(\tau))) = \mathsf{L}_{f(\epsilon(\overline{n}))}(\eta(\epsilon(\tau)))$. Now, stating the soundness theorem is straightforward:

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*

- $\mathit{dom}(s) = \mathit{dom}(\Gamma)$
- $s;\ h;\ \mathcal{C} \vdash e \rightsquigarrow v;\ h'$,
- $D,\ \Gamma \vdash_\Sigma e : \tau$ *is a node in the derivation tree for some function body,*
- $D(\epsilon(\overline{n}))$ *holds, where $\overline{n}$ is the set of size variables from $\mathit{dom}(\Gamma)$,*

the following implication holds:

$$\forall_{\eta,\epsilon} [\ \mathit{Valid}_{\mathsf{store}}(\mathit{dom}(s), \eta(\epsilon(\Gamma)), s, h) \implies \mathit{Valid}_{\mathsf{val}}(v, \eta(\epsilon(\tau)), h')\ ]$$

Proof. The proof is done by induction on the size of the derivation tree for the operational-semantic judgement. For the sake of convenience we will denote $D(\epsilon(\overline{n}))$ via D_ϵ, $\eta(\epsilon(\tau))$ via $\tau_{\eta\epsilon}$ and $\eta(\epsilon(\Gamma))$ via $\Gamma_{\eta\epsilon}$. Given $s;\ h;\ \mathcal{C} \vdash e \rightsquigarrow v;\ h'$

fix some Γ, Σ, and τ, such that D, $\Gamma \vdash_\Sigma e\colon \tau$. One can easily check by induction that $TV(\tau) \subseteq TV(\Gamma)$. Fix a valuation $\epsilon\colon SV(\Gamma) \cup SV(D) \to \mathcal{R}$, and a type instantiation $\eta\colon TV(\Gamma) \to \tau^\bullet$ such that the assumptions of the lemma hold. We must show that $\mathit{Valid}_{\mathsf{val}}(v, \tau_{\eta\epsilon}, h')$ holds. The full proof is given in the technical report [9]. Below, we consider only the most interesting case: pattern matching.

OSNull: In this case $v = \texttt{NULL}$, and according to the definition of the model relation we have $\texttt{NULL} \models^h_{\mathsf{L}_0(\tau'_{\eta\epsilon})} \texttt{[]}$ for τ' from the typing rule. Now we use the fact that $D \vdash \tau \to \mathsf{L}_0(\tau')$ and D_ϵ holds to obtain $\texttt{NULL} \models^h_{\tau_{\eta,\epsilon}} \texttt{[]}$.

OSVar: In this case $v = s(\mathsf{x})$. From $\mathit{Valid}_{\mathsf{store}}(\mathit{dom}(s), (\Gamma' \cup (x : \tau')_{\eta\epsilon}, h, s)$ for the corresponding τ' it follows that $s(\mathsf{x}) \models^h_{\tau'} w$ for some w. Now, $D \vdash \tau \to \tau'$ and D_ϵ imply $v \models^h_{\tau_{\eta,\epsilon}} w$.

OSCons: In this case $e = \mathsf{Cons}(\mathsf{hd}, \mathsf{tl})$ and Γ is "Γ', $\mathsf{hd} : \tau_1, \mathsf{tl} : \mathsf{L}_{f(\overline{n})}(\tau_2)$" for some Γ', hd, tl, f and τ'. Since $\mathit{Valid}_{\mathsf{store}}(\mathit{dom}(s), \Gamma_{\eta\epsilon}, s, h)$ there exist w_{hd} and w_{tl} such that $s(\mathsf{hd}) \models^h_{\tau_{1\,\eta\epsilon}} w_{\mathsf{hd}}$ and $s(\mathsf{tl}) \models^h_{\mathsf{L}_{f(\epsilon(\overline{n}))}(\tau_{2\,\eta\epsilon})} w_{\mathsf{tl}}$. From the operational semantics judgement we have that $v = \ell$ for some location $\ell \notin \mathit{dom}(h)$, and $h' = h[\ell.\mathsf{hd} := s(\mathsf{hd}),\ \ell.\mathsf{tl} := s(\mathsf{tl})]$. Therefore, $h'.\ell.\mathsf{hd} \models^h_{\tau_{1\,\eta\epsilon}} w_{\mathsf{hd}}$ and $h'.\ell.\mathsf{tl} \models^h_{\mathsf{L}_{f(\epsilon(\overline{n}))}(\tau_{2\,\eta\epsilon})} w_{\mathsf{tl}}$ also hold. It is easy to see that $h = h'|_{\mathit{dom}(h')\setminus\{\ell\}}$. Thus,

$$h'.\ell.\mathsf{hd} \models^{h'|_{\mathit{dom}(h')\setminus\{\ell\}}}_{\tau_{1\,\eta\epsilon}} w_{\mathsf{hd}}$$
$$h'.\ell.\mathsf{tl} \models^{h'|_{\mathit{dom}(h')\setminus\{\ell\}}}_{\mathsf{L}_{f(\epsilon(\overline{n}))}(\tau_{2\,\eta\epsilon})} w_{\mathsf{tl}}$$

Applying $D \vdash \tau_2 \to \tau_1$ and D_ϵ gives $\ell \models^{h'}_{\mathsf{L}_{f(\epsilon(\overline{n}))+1}(\tau_{2\,\eta\epsilon})} w_{\mathsf{hd}} :: w_{\mathsf{tl}}$. Using the fact that $D \vdash \tau \to \mathsf{L}_{f(\overline{n})+1}(\tau'_2)$ we obtain $v \models^h_{\tau_{\eta\epsilon}} w_{\mathsf{hd}} :: w_{\mathsf{tl}}$.

OSIfTrue: In this case $e = \mathsf{if}\ x\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2$ for some e_1, e_2, and x. We apply the induction hypothesis to the derivation of $s;\ h;\ \mathcal{C} \vdash e_1 \rightsquigarrow v;\ h'$, with the same η, ϵ to obtain

$\mathit{Valid}_{\mathsf{store}}(\mathit{dom}(s), \Gamma_{\eta\epsilon}, s, x) \implies \mathit{Valid}_{\mathsf{val}}(v, \tau_{1\ \eta\epsilon}, h')$

Due to the condition of the lemma, we have $\mathit{Valid}_{\mathsf{val}}(v, \tau_{1\,\eta\epsilon}, h')$, and thus $\mathit{Valid}_{\mathsf{val}}(v, \tau_{\eta\epsilon}, h')$.

OSIfFalse: Exactly as the true-case, but with e_2 instead of e_1.

OSLetFun: The result follows from the induction hypothesis for

$$s;\ h;\ \mathcal{C}[\mathsf{f} := (\mathsf{x} \times e_1)] \vdash e_2 \rightsquigarrow v;\ h',$$

with $\Gamma \vdash_\Sigma e_2 : \tau$ and the same η, ϵ.

OSLet: In this case e is $\mathsf{let}\ \mathsf{z} = e_1\ \mathsf{in}\ e_2$ for some z, e_1, and e_2 and we have $s;\ h;\ \mathcal{C} \vdash e_1 \rightsquigarrow v_1;\ h_1$ and $s[\mathsf{z} := v_1];\ h_1;\ \mathcal{C} \vdash e_2 \rightsquigarrow v;\ h'$ for some v_1 and h_1. Applying the induction hypothesis to the let-binding branch in let-rule's antecedent gives $\mathit{Valid}_{\mathsf{store}}(\mathit{dom}(s), D, \Gamma_{\eta\epsilon}, s, h) \implies \mathit{Valid}_{\mathsf{val}}(v_1, \tau'_{\eta\epsilon}, h_1)$. Now apply the induction hypothesis to the other branch to get

$\mathit{Valid}_{\mathsf{store}}(\mathit{dom}(s[\mathsf{z} := v_1]), \Gamma_{\eta\epsilon} \cup \{\mathsf{z} : \tau'_{\eta\epsilon}\}, s[\mathsf{z} := v_1], h_1) \implies \mathit{Valid}_{\mathsf{val}}(v, \tau_{\eta\epsilon}, h')$

Fix some $\mathsf{z}' \in \mathit{dom}(s[\mathsf{z} := v_1])$. If $\mathsf{z}' = \mathsf{z}$, then $\mathit{Valid}_{\mathsf{val}}(v_1, \tau'_{\eta\epsilon}, h_1)$ implies $\mathit{Valid}_{\mathsf{val}}(s[\mathsf{z} := v_1](\mathsf{z}), \tau'_{\eta\epsilon}, h_1)$. If $\mathsf{z}' \neq \mathsf{z}$, then $s[\mathsf{z} := v_1](\mathsf{z}') = s(\mathsf{z}')$. Sharing

of data structures in the heap is benign (no destructive pattern matching and assignments), hence $h|_{\mathcal{R}(h,\ s(\mathsf{z}'))} = h_1|_{\mathcal{R}(h,\ s(\mathsf{z}'))}$. Thus, we have that $s(\mathsf{z}') \models^{h}_{\Gamma_{\eta\epsilon}(\mathsf{z}')} w'_z$ implies $s(\mathsf{z}') \models^{h_1}_{\Gamma_{\eta\epsilon}(\mathsf{z}')} w'_z$ implies $s[\mathsf{z} := v_1](\mathsf{z}') \models^{h_1}_{\Gamma_{\eta\epsilon}(\mathsf{z})} w_{\mathsf{z}'}$. So, $\mathit{Valid}_{\mathsf{val}}(s[\mathsf{z} := v_1](\mathsf{z}'), \Gamma_{\eta\epsilon}(\mathsf{z}'), h_1)$. Hence,

$\mathit{Valid}_{\mathsf{store}}(\mathit{dom}(s[\mathsf{z} := v_1]), \Gamma_{\eta\epsilon} \cup \{\mathsf{z}\colon \tau'_{\eta\epsilon}\}, s[\mathsf{z} := v_1], h_1)$

and we may apply the induction assumption.

OSMatch-Nil: In this case $e = \mathsf{match}\ \mathsf{x}\ \mathsf{with} \mid \mathsf{Nil} \Rightarrow e_1 \mid \mathsf{Cons}(\mathsf{hd}, \mathsf{tl}) \Rightarrow e_2$ for some x, hd, tl, e_1, and e_2. The typing context has the form $\Gamma = \Gamma' \cup \{\mathsf{x}\colon \mathsf{L}_f(\tau)\}$ for some Γ', τ' and f. The operational-semantics derivation gives $s(\mathsf{x}) = \mathtt{NULL}$, hence, validity for $s(\mathsf{x})$ gives $0 \in f(\epsilon(\overline{n}))$. Therefore, $s(\mathsf{x}) \models^{h}_{\mathsf{L}_{f_\epsilon}(\tau')} [\,]$. This means that $\mathit{Valid}_{\mathsf{store}}(\mathit{dom}(s), \Gamma'_{\eta\epsilon'},\ \mathsf{x}\colon \mathsf{L}_{f_\epsilon}(\tau'), s, h)$. We can apply the the induction hypothesis with $D_\epsilon,\ 0 \in f(\epsilon(\overline{n}));\ \Gamma,\ \mathsf{x}\colon \mathsf{L}_{f_\epsilon}(\tau') \vdash_\Sigma e\colon \tau$ to obtain $\mathit{Valid}_{\mathsf{val}}(v, \tau_{\eta\epsilon}, h')$.

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

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

$$\mathit{Valid}_{\mathsf{store}}(\mathit{dom}(s'), \Gamma_{\eta\epsilon},\ \mathsf{x}\colon \mathsf{L}_{f(\epsilon(\overline{n}))}(\tau'_{\eta\epsilon}),\ \mathsf{hd}\colon \tau'_{\eta\epsilon},\ \mathsf{tl}\colon \mathsf{L}_{f(\epsilon(\overline{n}))-1}(\tau')_{\eta\epsilon}, s', h)$$

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

$$\Gamma',\ \mathsf{x}\colon \mathsf{L}_{f(\epsilon(\overline{n}))}(\tau'_{\eta\epsilon}),\ \mathsf{hd}\colon \tau'_{\eta\epsilon},\ \mathsf{tl}\colon \mathsf{L}_{f(\epsilon(\overline{n}))-1}(\tau'_{\eta\epsilon}) \vdash_\Sigma e_2\colon \tau_{\eta\epsilon}$$

With $D_\epsilon, n_0 \geq 1 \in f(\overline{n})$ and $\epsilon' = \epsilon[n_0 := \mathit{length}_h(s(\mathsf{x}))]$ the induction hypothesis yields

$$\mathit{Valid}_{\mathsf{store}}(\mathit{dom}(s'), \left\{\begin{array}{l} \Gamma'_{\eta\epsilon} \cup \\ \{\mathsf{x}\colon \mathsf{L}_{f(\epsilon'(\overline{n}))}(\tau'_{\eta\epsilon'})\} \cup \\ \{\mathsf{hd}\colon \tau'_{\eta\epsilon'}\} \cup \\ \{\mathsf{tl}\colon \mathsf{L}_{f(\epsilon'(\overline{n}))-1}(\tau')\}_{\eta\epsilon'}\} \end{array}\right\}, s\begin{bmatrix}\mathsf{hd} := v_{\mathsf{hd}}, \\ \mathsf{tl} := v_{\mathsf{tl}}\end{bmatrix}, h) \implies \mathit{Valid}_{\mathsf{val}}(v, \tau_{\eta\epsilon'}, h')\,.$$

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

OSFun: We want to apply the induction assumption to

$$[y_1 := v_1, \ldots, y_k := v_k];\ h;\ \mathcal{C} \vdash e_f \rightsquigarrow v;\ h'.$$

Since the original typing judgement is a node in a derivation tree, where all called in e functions are defined via letfun, there must be a node in

the derivation tree with True, $y_1 : \tau^\circ, \ldots, y_k : \tau_k^\circ \vdash_\Sigma e_f : \tau_0$. Trivially, the domains of the frame store $[y_1 := v_1, \ldots, y_k := v_k]$ and the context $y_1 : \tau^\circ, \ldots, y_k : \tau_k^\circ$ coincide.
We take η' and ϵ', such that

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

True ("no conditions") holds trivially on ϵ'. From the induction assumption we have $\mathit{Valid}_{\mathsf{store}}((y_1, \ldots\ y_k), (y_1 : \tau^\circ_{1\ \eta'\epsilon'}, \ldots, y_k : \tau^\circ_{k,\ \eta'\epsilon'}), [y_1 := v_1, \ldots, y_n := v_n], h)$
$\Longrightarrow\ \mathit{Valid}_{\mathsf{val}}(v, \tau_{0\eta'\epsilon'}, h')$
From $\mathit{Valid}_{\mathsf{store}}(\mathit{dom}(s), \Gamma_{\eta\epsilon}, s, h)$ we have validity of the values of the actual parameters: $v_i \models^h_{\Gamma_{\eta\epsilon}(\mathsf{x}_i)} w_i$ for some w_i, where $1 \leq i \leq k$. Since $\Gamma_{\eta\epsilon}(\mathsf{x}_i) = \tau^\circ_{i\ \eta'\epsilon'}$, 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
$\eta\epsilon(\sigma(\tau_0)) = \eta\epsilon(\tau_0[\ldots\alpha := \tau_\alpha \ldots][\ldots n_{ij} := f_{ij} \ldots]) =$
$\tau_0[\ldots\alpha := \eta(\tau_\alpha)\ldots][\ldots n_{ij} := \epsilon(f_{ij})\ldots] = \tau_{0\ \eta'\epsilon'}$
Therefore, we obtain $\mathit{Valid}_{\mathsf{val}}(v, \eta(\epsilon(\sigma(\tau_0))), h')$ and using the rule $D \vdash \tau \rightarrow \sigma(\tau_0)$ we obtain $\mathit{Valid}_{\mathsf{val}}(v, \tau_{\eta\epsilon}, h')$. We apply the lemma **??** to obtain $v \models^h_{\tau_{\eta\epsilon}} w$.

□

ToDo: Lemma: rewriting preserves model relation

4 Approximation of non-deterministic size functions by families of $\max_0$-polynomials

4.1 Checking if a family approximates a given size function

A simple way to ensure decidability of type checking is to require that all statements in D have the form $n - c = 0$ or $n - c \geq 1$, where c is a constant. This requirement is fulfilled by program expression satisfying the the syntactical condition *pattern matching is done only on function parameters or variables bound to by other pattern matchings.* This condition was first formulated by us in [11].

This condition is not very restrictive [14], since primitive recursion over lists does not need pattern matching on function calls:

$$\begin{aligned}\mathsf{f}(\mathsf{l}, \bar{\mathsf{x}}) = \mathsf{match\ l\ with}\ &\mid \mathsf{Nil} \Rightarrow e_1\\ &\mid \mathsf{Cons(hd,\ tl)} \Rightarrow e_2(\mathsf{f(tl)}, \mathsf{l}, \bar{\mathsf{x}})\end{aligned}$$

The condition is sufficient but not necessary for decidability. In general, type checking depends on solving polynomial equations of the form $p(\overline{n}) = 0$. In practice, programs have quite simple size dependencies that give rise to simple and solvable equations. We believe that, as part of future work, the reasoning given below may be extended to function definitions with pattern matchings on function calls with simple (linear or quadratic) size dependencies. Then, possible non-determinism of the size dependencies should be carefully taken into account.

We want to show that, given a function definition and its *non-deterministic size function and some indexed family of* $\max_0$*-polynomials, there is a set of first-order arithmetic predicates such that their satisfiability implies that the family approximates the size function.* Such predicates arise as side conditions during type checking for types annotated with indexed families directly. We have studied their decidability in [10].

We start with the necessary formalities. Let f_l, where $1 \leq l \leq s$, be the size dependencies of all functions called in a given function body. Let the families

$$\{g_l(\overline{n},\, \overline{j}_l)\}_{\overline{j}_l.\ Q_l(\overline{n},\overline{j}_l)}$$

be their approximations. Let $p(\overline{n})$ be a $\max_0$-polynomial expression possibly containing $f_1, \ldots, f_l$ as sub-expressions. Such $p(\overline{n})$ appears on the r.h.s. of the arrows in rewriting rules. We inductively define the *substitution* $[-]$ *of the approximations into* p. The substitution $[p] = \{g(\overline{n},\overline{j})\}_{\overline{j}}$ is defined as follows:

- $[n] = n$.
- $[p_1(\overline{n}) + p_2(\overline{n})] = \{g'_1(\overline{n},\overline{j}'_1) + g'_2(\overline{n},\overline{j}'_2)\}_{\overline{j}'_1,\ \overline{j}'_2.\ Q'_1(\overline{n},\overline{j}'_1)\wedge Q'_2(\overline{n},\overline{j}'_2)}$, where $[p_\imath] = \{g'_\imath(\overline{n},\, \overline{j}'_\imath)\}_{\overline{j}'_\imath.\, Q'_\imath(\overline{n},\overline{j}'_\imath)}$, with $\imath = 1, 2$.
- The definition for $p_1 - p_2$ and $p_1 * p_2$ is similarly to the previous one.
- $[\max_0(p'(\overline{n}))] = \{\max_0(g'(\overline{n},\overline{j}'))\}_{\overline{j}'.\ Q'_1(\overline{n},\overline{j}')}$, where $[p'] = \{g'(\overline{n},\, \overline{j}')\}_{\overline{j}'.\, Q'(\overline{n},\overline{j}')}$.
- $[f_l(p_1(\overline{n}), \ldots, p_t(\overline{n}))] = \{g_l(g'_1(\overline{n},\overline{j}'_1), \ldots, g'_t(\overline{n},\overline{j}'_t), \overline{j}_l)\}$ where $\overline{j}_l$ satisfies $Q_l(g'_1(\overline{n},\overline{j}'_1), \ldots, g'_t(\overline{n},\overline{j}'_t), \overline{j}_l)$ and $\bigwedge_{\imath=1,\ldots,t} Q'_\imath(\overline{n},\, \overline{j}'_\imath)$ holds, and, moreover, $[p_\imath] = \{g'_\imath(\overline{n},\, \overline{j}'_\imath)\}_{\overline{j}'_\imath.\ Q'_\imath(\overline{n},\overline{j}'_\imath)}$, with $1 \leq \imath \leq t$.

By induction on the structure of p one can prove the following lemma.

Lemma 2 (Substitution of approximations is an approximation). *Given* p*, a family* $[p] = \{g(\overline{n},\overline{j})\}_{\overline{j}}$*, constructed as above, approximates* p*.*

Proof. Consider the most involving case $p = f_l(p_1(\overline{n}), \ldots, p_t(\overline{n}))$. Fix some $\overline{n}$ and $m \in p(\overline{n})$. There are $m_1 \in p_1(\overline{n}), \ldots, m_t \in p_t(\overline{n})$ such that $m \in f_l(m_1, \ldots m_t)$. By the induction assumptions there are $\overline{j}'_1, \ldots, \overline{j}_t$, such that $m_1 = g'_1(\overline{n},\overline{j}_1)$, ..., $m_t = g'(\overline{n},\overline{j}_t)$ and $Q'_1(\overline{n},\overline{j}_1), \ldots Q'_t(\overline{n},\overline{j}_t)$ hold. Since $\{g_l(\overline{n},\overline{j}_l)\}_{\overline{j}_l.Q(\overline{n},\overline{j}_l)}$ approximates f_l, there are $\overline{j}_l$ such that $m = g(m_1, \ldots, m_t, \overline{j}_l)$ and $Q_l(m_1 \ldots., m_t, \overline{j}_l)$ holds. Thus, $\{g_l(g'_1(\overline{n},\overline{j}'_1), \ldots, g'_t(\overline{n},\overline{j}'_t), \overline{j}_l)\}$ approximates p.

We extend this definition for types of the form $\tau = \mathsf{L}_{p_1}(\ldots \mathsf{L}_{p_k}(\alpha)\ldots)$. The type $[\tau] = \mathsf{L}_{[p_1]}(\ldots \mathsf{L}_{[p_k]}(\alpha)\ldots)$ denotes the approximation of τ by the families of polynomials as defined above.

To define first-order predicates sufficient for checking the correctness of approximations, recall the definition of $\preccurlyeq$ on types annotated with indexed families directly. Let $T = \mathsf{L}_{\{g^1(\overline{n},\overline{i}^1)\}_{\overline{i}^1.\ Q^1(\overline{n},\overline{i}^1)}}(\ldots \mathsf{L}_{\{g^k(\overline{n},\overline{i}^k)\}_{\overline{i}^k.\ Q^k(\overline{n},\overline{i}^k)}}(\alpha)\ldots)$. Let $T' = \mathsf{L}_{\{g'^1(\overline{n},\overline{j}^1)\}_{\overline{j}^1.\ Q'^1(\overline{n},\overline{j}^1)}}(\ldots \mathsf{L}_{\{g'^k(\overline{n},\overline{j}^k)\}_{\overline{j}^k.\ Q'^k(\overline{n},\overline{j}^k)}}(\alpha)\ldots)$. Then $D \vdash T' \preccurlyeq T$ holds if and only if

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

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

$$D \vdash \mathsf{L}_{\{g'^2(\overline{n},\bar{j}^2)\}_{\bar{j}^2.\ Q'^2(\overline{n},\bar{j}^2)}}(\ldots \mathsf{L}_{\{g'^k(\overline{n},\bar{j}^k)\}_{\bar{j}^k.\ Q'^k(\overline{n},\bar{j}^k)}}(\alpha)\ldots) \preccurlyeq \mathsf{L}_{\{g^2(\overline{n},\bar{i}^2)\}_{\bar{i}^2.\ Q^2(\overline{n},\bar{i}^2)}}(\ldots \mathsf{L}_{\{g^k(\overline{n},\bar{i}^k)\}_{\bar{i}^k.\ Q^k(\overline{n},\bar{i}^k)}}(\alpha)\ldots)$$

Lemma 3 (Checking). *Let $f = f_{l_0} \in \{f_1(\overline{n}), \ldots, f_l(\overline{n})\}$ and all f_l with $l \neq l_0$ be approximated by $\{g(\overline{n}, \bar{j}_l)\}_{\bar{j}_l.\ Q(\overline{n},\bar{j}_l)}$. Let f be defined by rewriting rules given by $D \vdash \tau \to \tau'$. A family $\{g(\overline{n}, \bar{i})\}_{\bar{i}.\, Q(\overline{n},\bar{i})}$ approximates f if $D \vdash [\tau'] \preccurlyeq [\tau]$ for all $D \vdash \tau \to \tau'$ that define f.*

Proof. By induction on the derivation chain for an arbitrary fixed $m \in f(\overline{n})$. Consider an induction step k. Let some $m \in f(\overline{n})$ is obtained by the rule $D \vdash f(n) \to p(\overline{n})$, where p may contain f_l-s as subexpressions, including f. Thus, $m \in p(\overline{n})$ as well and $D(\overline{n})$ holds. Let $[p] = \{g'(\overline{n}, \bar{j})\}_{\bar{j}.\ Q'(\overline{n},\bar{j})}$ Then due to $D \vdash [\tau'] \preccurlyeq [\tau]$ we have $\forall\ \overline{n}\ \bar{j}.\ \exists \bar{i}.\ D(\overline{n}) \wedge Q'(\overline{n}, \bar{j}) \Longrightarrow g'(\overline{n}, \bar{j}) = g(\overline{n}, \bar{i}) \wedge Q(\overline{n}, \bar{i})$. Due to the lemma **??** (and the induction assumption: $g(\overline{n}, \bar{i})$ approximates the "sub-function" f, that is all the values of f obtained until the step $k-1$) the substitution $[p]$ approximates p, that is for $m \in p(\overline{n})$ there exists $\bar{j}_0$, such that $m = g'(\overline{n}, \bar{j}_0)$ and $Q'(\overline{n}, \bar{j}_0)$ holds. Applying the entailment above, we obtain that there exists $\bar{i}_0$ such that $m = g(\overline{n}, \bar{i}_0)$ and $Q(\overline{n}, \bar{i}_0)$ holds.

Example insert. Checking whether $\{n + i\}_{0 \leq i \leq 1}$ reduces to checking the entailments

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

Which is solved by instantiating $?i$ to 0, 0 and j, respectively.

Example rel. Checking whether $\{i\}_{0 \leq i \leq nm}$ reduces to checking the entailments

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

See Section 3 for the definitions of the functions insert and rel.

4.2 Inference of an approximating family for a given size function

To give an idea of the procedure to infer approximating families of $\max_0$-polynomials, we start with a simple example. We show how to infer polynomial lower and upper bounds on size dependencies for insert and how to construct an approximating family from it. Recall the size rewriting system for insert:

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

The function insert has a polynomial lower and upper size bounds $p_{\min}(n) = n$ and $p_{\max}(n) = n + 1$, respectively. It is easy to see that they are accurate, i.e. they are the *greatest lower bound.* and the *lowest upper bound*, respectively. First note that for insert, given any n, there is an evaluation path for $f_{\mathsf{insert}}(n)$ that evaluates to $p_{\min}(n)$, and there is a path that evaluates to $p_{\max}(n)$. Now, assume that $p_{\min}$ and $p_{\max}$ are linear, that is, that they are of the form $a_{\min}n+b_{\min}$ and $a_{\max}n+b_{\max}$, respectively. We want to find the coefficients $a_{\min},\ b_{\min}, a_{\max},\ b_{\max}$ (as we did in [11] for strict polynomial size dependencies, where the lower and upper bounds were equal). To reconstruct $p_{\min}$, one needs to know two points on its graph, and the same holds for $p_{\max}$. Take $n = 1$ and $n = 2$. Evaluating the t.r.s. gives $p(1) = \{1,\ 2\}$ and $p(2) = \{2,\ 3\}$. Pick up the minimal values from $p(1)$ and $p(2)$ and notice that the graph of $p_{\min}$ contains the points (1, 1) and (2, 2). Similarly, pick up the maximal values of $p(1)$ and $p(2)$ and notice that $p_{\max}$ contains (1, 2) and (2, 3). We obtain two systems of equations, for $a_{\min},\ b_{\min}$ and $a_{\max},\ b_{\max}$, respectively:

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

Solving these linear systems we get $a_{\min} = 1$, $b_{\min} = 0$ and $a_{\max} = 1$, $b_{\max} = 1$. Thus, we reconstruct the expressions for $p_{\min}(n) = n$ and $p_{\max}(n) = n + 1$.

The output size dependency for insert is $p_{min}(n) + i$, where $0 \leq i \leq \delta(n)$ with $\delta(n) = p_{\max}(n) - p_{\min}(n) = 1$. The rest of the job is to check whether this reconstruction approximates the solution of the rewriting rules.

Here the dependency is one-variable and the systems of linear equations w.r.t. the polynomial coefficients are trivially consistent if one chooses different testing size values, $n = 1$ and $n = 2$. This is because the matrix of such a system has a 1-variable non-zero *Vandermonde* determinant.

In the multi-variate case, say s variables, the consistency of the systems for $p_{\min}$ and $p_{\max}$ (for which the corresponding multivariate Vandermonde determinant is non-zero) depends on a more involving condition. If the testing values, i.e. points in an s-dimensional space, form an **NCA** configuration [14], then the systems for $p_{\min}$ and $p_{\max}$ have unique solutions, that is the polynomials are defined uniquely.

Here, we will treat the case for $s = 2$. Let d be the degree of a polynomial and N_d^2 denote the amount of its coefficients. N_d^2 points that form a set W on a plane lie in a *2-dimensional* ***NCA*** *configuration* if there exist lines $\gamma_1, \ldots, \gamma_{d+1}$ in the space $\mathcal{R}^2$, such that $d + 1$ nodes of W lie on γ_{d+1} and d nodes of W lie on $\gamma_d \setminus \gamma_{d+1}$, ..., and finally 1 node of W lies on $\gamma_1 \setminus (\gamma_2 \cup \ldots \cup \gamma_{d+1})$.

The simplest examples of points on the plane satisfying **NCA** are "triangles", like $(1, 1)$, $(2, 1)$, $(3, 1)$, $(1, 2)$, $(2, 2)$ $(1, 3)$.

Now we give a general procedure for inferring lower and upper polynomial bounds from a given system of size rewriting rules. Without loss of generality we assume that the lower and upper polynomial bounds have the same degree. If necessary, this can be achieved by assuming the coefficients at the higher degrees of the lower bound to be zeros.

INPUT:	The degree d of a hypothetical upper and lower polynomial bounds, the system G of size rewriting rules, s size variables, $\overline{n} = (n_1, \ldots, n_s)$
OUTPUT:	A lower $p_{\min}$ and an upper $p_{\max}$ bound or the proposal to increase the degree and repeat the procedure.
PROCEDURE:	1. Pick up N_d^s points in the s-dimensional space that form an **NCA** configuration; let they constitute the set W.
	2. For any $w_i = (n_1, \ldots, n_s) \in W$ compute the set $f_i = f(n_1, \ldots, n_s)$.
	3. For any f_i pick up its minimal $f_i^{\min}$ and maximal $f_i^{\max}$ values.
	4.1. Compute $p_{\min}$ that interpolates the points $(w_i; f_i^{\min})$ by solving the system of linear equations w.r.t. its coefficients.
	4.2. Compute $p_{\max}$ that interpolates the points $(w_i; f_i^{\max})$ by solving the system of linear equations w.r.t. its coefficients.
	5. Check whether the family $\{p_{\min}(\overline{n}) + i\}_{0 \le i \le (p_{\max}(\overline{n}) - p_{\min}(\overline{n}))}$ approximates the nondeterministic function defined by G.
	5.1 If "yes": stop and output $p_{\min}$ and $p_{\max}$.
	5.2 If "not": pick up another test set W or increase the degree d.

We have given the core procedure. It admits many adaptations that are mainly related to choosing test size values so that $p_{\min}$ and $p_{\max}$ will eventually land on them. Adaptations for inferring the bounds for $\max_0$ are also possible, however interaction with a user is necessary. Expressions with $\max_0$ are piecewise polynomials, hence we expect that the user may have to hint the inference system on which areas P_i one assumes different "pieces" of polynomial bounds. The set $W_i \subseteq P_i$ of test size values must be chosen separately for each P_i.

Consider as a second example, the inference procedure for the function rel. The inferred size rewriting system of interest is:

$$\begin{array}{cc} \vdash f_{\mathsf{rel}_1}(0,m) \to 0 & \vdash f_{\mathsf{rel}_2}(0,m) \to ? \\ n \ge 1 \vdash f_{\mathsf{rel}_1}(n,m) \to f_{\mathsf{rel\_pairs}_1}(m) + f_{\mathsf{rel}_1}(n-1,m) & n \ge 1 \vdash f_{\mathsf{rel}_2}(n,m) \to f_{\mathsf{rel\_pairs}_2}(m) \end{array}$$

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

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

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

$$
\begin{array}{ll}
f_{\mathsf{rel}}(1,1) = j + 0 & = \{0,\, 1\} \\
f_{\mathsf{rel}}(2,1) = j + p(1,\, 1) & = \{0,\, 1,\, 2\} \\
f_{\mathsf{rel}}(3,1) = j + p(2,\, 1) & = \{0,\, 1,\, 2,\, 3\} \\
f_{\mathsf{rel}}(1,2) = j + 0 & = \{0,\, 1,\, 2\} \\
f_{\mathsf{rel}}(2,2) = j + p(1,\, 2) & = \{0,\, 1,\, 2,\, 3,\, 4\} \\
f_{\mathsf{rel}}(1,3) = j + 0 & = \{0,\, 1,\, 2,\, 3\}
\end{array}
$$

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

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

The solution is $(0, 0, 1, 0, 0, 0)$, so $p_{\max}(n, m) = nm$. The similar system of $p_{\min}$ has all zeros on its r.h.s., so the coefficients for $p_{\min}$ are all zeros and the inferred family is indeed $\{i\}_{0 \leq i \leq nm}$. This family approximates the non-deterministic size dependency f_1.

For the coefficients of $p_{\min}$ one has the system

$$
\begin{array}{ll}
a_{20} + a_{02} + a_{11} + a_{10} + a_{01} + a_{00} & = 0 \\
4a_{20} + a_{02} + 2a_{11} + 2a_{10} + a_{01} + a_{00} & = 0 \\
9a_{20} + 3a_{02} + 3a_{11} + 3a_{10} + a_{01} + a_{00} & = 0 \\
a_{20} + 4a_{02} + 2a_{11} + a_{10} + 2a_{01} + a_{00} & = 0 \\
4a_{20} + 4a_{02} + 4a_{11} + 2a_{10} + 2a_{01} + a_{00} & = 0 \\
a_{20} + 9a_{02} + 3a_{11} + a_{10} + 3a_{01} + a_{00} & = 0
\end{array}
$$

which trivially gives that all the coefficients for $p_{\min}$ are zero. The solution is $(0, 0, 1, 0, 0, 0)$, so $p_{\max}(n, m) = nm$ and the inferred family is indeed $\{i\}_{0 \leq i \leq nm}$. This family approximates the non-deterministic size dependency f_1.

5 Feasibility of Analysing the Haskell library

As a small feasibility study used our analysis on the Haskell list Library, in particular we took Hugs' list library version of September 2006. Since we have an intermediate language we first needed to make some assumptions.

First of all, we assume strict semantics which means that we get a good approximation for the use of the function in a strict input and output context. Secondly, it must be possible to translate the function into our language. Our type system requires that inner lists all have the same length, which is not the case for e.g. transpose. This restriction may be removed in a future version of our work. Thirdly, we ignore classes of types like Eq and Ord and we write the functions uncurrified. Finally, we write *interface types* where the family is given as an annotation that can be inferred directly from the set of term rewriting rules as shown above.

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

Many functions (like append, tail, init, map, reverse, concat, union and sort) are shapely, i.e. have a deterministic exact polynomial size dependency. These functions are typable in the type systems developed in [11, 12]. Of these functions we only give the type of append: $\mathsf{L}_n(\alpha) \times \mathsf{L}_m(\alpha) \to \mathsf{L}_{n+m}(\alpha)$.

Some shapely functions need $\max_0$ and/or higher order functions. They can be dealt with using the framework of this paper as follows:

$$\begin{array}{ll}
\mathsf{intersperse} & : \alpha \times \mathsf{L}_n(\alpha) \to \mathsf{L}_{\max_0(2*n-1)}(\alpha) \\
\mathsf{scanl} & : (\alpha \times \beta \to \alpha) \times \alpha \times \mathsf{L}_n(\beta) \to \mathsf{L}_{n+1}(\alpha) \\
\mathsf{scanl1} & : (\alpha \times \alpha \to \alpha) \times \mathsf{L}_n(\alpha) \to \mathsf{L}_n(\alpha)
\end{array}$$

For function definitions with a non-deterministic size dependency a family of polynomials is needed to express the different possible sizes of the output. In fact, many functions fall in this category. Their types are listed below.

$$\begin{array}{ll}
\mathsf{takeWhile} & : (\alpha \to \mathsf{Bool}) \to \mathsf{L}_n(\alpha) \to \mathsf{L}_{\{i\}_{0\le i\le n}}(\alpha) \\
\mathsf{dropWhile} & : (\alpha \to \mathsf{Bool}) \to \mathsf{L}_n(\alpha) \to \mathsf{L}_{\{i\}_{0\le i\le n}}(\alpha) \\
\mathsf{inits},\ \mathsf{tails} & : \mathsf{L}_n(\alpha) \to \mathsf{L}_{n+1}(\mathsf{L}_{\{i\}_{0\le i\le n}}(\alpha)) \\
\mathsf{filter} & : (\alpha \to \mathsf{Bool}) \to \mathsf{L}_n(\alpha) \to \mathsf{L}_{\{i\}_{0\le i\le n}}(\alpha) \\
\mathsf{elemIndices} & : \alpha \times \mathsf{L}_n(\alpha) \to \mathsf{L}_{\{i\}_{0\le i\le n}}(\mathsf{Int}) \\
\mathsf{findIndices} & : (\alpha \to \mathsf{Bool}) \times \mathsf{L}_n(\alpha) \to \mathsf{L}_{\{i\}_{0\le i\le n}}(\mathsf{Int}) \\
\mathsf{nub} & : \mathsf{L}_n(\alpha) \to \mathsf{L}_{\{i\}_{0\le i\le n}}(\alpha) \\
\mathsf{nubBy} & : (\alpha \times \alpha \to \mathsf{Bool}) \times \mathsf{L}_n(\alpha) \to \mathsf{L}_{\{i\}_{0\le i\le n}}(\alpha) \\
\mathsf{delete} & : \alpha \times \mathsf{L}_n(\alpha) \to \mathsf{L}_{\{\max_0(n-i)\}_{0\le i\le 1}}(\alpha) \\
\mathsf{deleteBy} & : (\alpha \times \alpha \to \mathsf{Bool}) \times \mathsf{L}_n(\alpha) \to \mathsf{L}_{\{\max_0(n-i)\}_{0\le i\le 1}}(\alpha) \\
\mathsf{rdelete} & : \mathsf{L}_n(\alpha) \times \mathsf{L}_m(\alpha) \to \mathsf{L}_{\{\max_0(n-i)\}_{0\le i\le m}}(\alpha) \\
\mathsf{deleteFirstsBy} & : (\alpha \times \alpha \to \mathsf{Bool}) \times \mathsf{L}_n(\alpha) \times \mathsf{L}_m(\alpha) \to \mathsf{L}_{\{\max_0(n-i)\}_{0\le i\le m}}(\alpha) \\
\mathsf{intersect} & : \mathsf{L}_n(\alpha) \times \mathsf{L}_m(\alpha) \to \mathsf{L}_{\{i\}_{0\le i\le \max_0(n,\max_0(n-m))}}(\alpha) \\
\mathsf{intersectBy} & : (\alpha \times \alpha \to \mathsf{Bool}) \times \mathsf{L}_n(\alpha) \times \mathsf{L}_m(\alpha) \to \mathsf{L}_{\{i\}_{0\le i\le \max_0(n,\max_0(n-m))}}(\alpha)
\end{array}$$

Adding algebraic data types to our language, many other functions could be analysed. There are, of course, functions whose sized types cannot be expressed in our type system or our procedure cannot deal with them. Firstly, when *the size of the result cannot be determined statically* because it depends on the value of the arguments (like unfoldr). Secondly, when *the size of the output depends on a higher-order parameter* our framework cannot deal with it (this is the case for concatMap). Finally, if *the size of the output is infinite*, we cannot derive that (e.g. iterate, repeat and cycle).

6 Related Work

This research continues our work on polynomial size dependencies [11, 14, 12] for shapely functions which have a deterministic, exact input-output polynomial size dependency. Our non-monotonic framework resembles [2] in which the authors describe *monotonic* resource consumption for Java bytecode by means

of Cost Equation Systems (CESs), which are similar to, but more general than recurrence equations. CESs express the cost of a program in terms of the size of its input data. In a further step, a closed-form solution or upper bound can sometimes be found by using existing Computer Algebra Systems, such as *Maple* and *Mathematica*. This work is continued by the authors in [1], where mechanisms for solving and upper bounding CESs are studied. However, they do not consider non-monotonic dependencies.

Our approach is related to size analysis with polynomial quasi-interpretations [6, 3]. There, a program is interpreted as a *monotonic* polynomial extended with the max operation. For instance, $\mathsf{Cons}(\mathsf{hd}, \mathsf{tl})$ is interpreted as $T + 1$, where T is a numerical variable abstracting tl. Using such interpretations one obtains upper monotonic-polynomial bounds for size dependencies. The main difference with our approach is that we are interested in non-monotonic lower and upper bounds. To our knowledge, non-monotonic quasi-interpretations have not been studied for size analysis, but only for proving termination [8]. In this work one considers some unspecified algorithmically decidable classes of non-negative and negative polynomials and introduces abstract variables for the rest.

The EmBounded project aims to identify and certify resource-bounded code in *Hume*, a domain-specific high-level programming language for real-time embedded systems. In his thesis, Pedro Vasconcelos [15] uses abstract interpretation to automatically infer linear approximations of the sizes of recursive data types and the stack and heap of recursive functions written in a subset of *Hume*.

Several papers have studied programming languages with *implicit computational complexity* properties [7, 5]. This line of research is motivated both by the perspective of automated complexity analysis and by fundamental goals, in particular to give natural characterisations of complexity classes, like PTIME or PSPACE. Resource analysis may be performed within a *Proof Carrying Code* framework. In [4] the authors introduce resource policies for mobile code to be run on smart devices. Policies are integrated into a proof-carrying code architecture. Two forms of policies are used: *guaranteed policies* which come with proofs and *target policies* which describe limits of the device.

7 Conclusions and Future Work

This paper presents a size-aware type system that describes non-deterministic size-dependencies between the inputs and the output of a function. It allows to express a family of output sizes via indexed *non-monotonic* polynomials augmented with the $\max_0$ operation. This feature greatly increases the applicability of our earlier size analysis, which was limited to exact sizes. The extra expressibility comes at a cost: we have crossed the border of decidability. However, this does not make the analysis infeasible in practice. It turns out that a great deal of the Hugs-Haskell list library can be annotated correctly.

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

References

1. E. Albert, P. Arenas, S. Genaim, and G. Puebla. Automatic Inference of Upper Bounds for Recurrence Relations in Cost Analysis. In *Static Analysis, 15-th International Symposium*, volume 5079 of *LNCS*, pages 221–237, 2008.
2. E. Albert, P. Arenas, S. Genaim, G. Puebla, and D. Zanardini. Cost Analysis of Java Bytecode. In 16^{th} *European Symposium on Programming, ESOP'07*, volume 4421 of *LNCS*, pages 157–172. Springer, 2007.
3. R. M. Amadio. Synthesis of max-plus quasi-interpretations. *Fundamenta Informaticae*, 65(1-2):29–60, 2004.
4. D. Aspinall and K. MacKenzie. Mobile Resource Guarantees and Policies. In G. Barthe, B. Grégoire, M. Huisman, and J.-L. Lanet, editors, *CASSIS 2005*, volume 3956 of *LNCS*, pages 16–36. Springer, 2006.
5. V. Atassi, P. Baillot, and K. Terui. Verification of Ptime Reducibility for System F Terms: Type Inference in Dual Light Affine Logic. *Logical Methods in Computer Science*, 3(4), 2007.
6. G. Bonfante, J.-Y. Marion, and J.-Y. Moyen. Quasi-interpretations, a way to control resources. *Theoretical Computer Science*, 2005.
7. M. Gaboardi, J.-Y. Marion, and S. Ronchi Della Rocca. A logical account of PSPACE. In 35^{th} *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages POPL 2008, San Francisco, January 10-12, 2008, Proceedings*, pages 121–131, 2008.
8. N. Hirokawa and A. Middeldorp. Polynomial interpretations with negative coefficients. In *Artificial Intelligence and Symbolic Computation*, volume 3249 of *LNCS*, 2004.
9. O. Shkaravska, M. van Eekelen, and A. Tamalet. Collected size semantics for functional programs. Technical Report ICIS-R08021, Radboud University Nijmegen, November 2008.
10. O. Shkaravska, M. van Eekelen, and A. Tamalet. Size analysis with indexed families of $\max_0$-polynomials. Technical Report ICIS-R08020, Radboud University Nijmegen, November 2008.
11. O. Shkaravska, R. van Kesteren, and M. van Eekelen. Polynomial Size Analysis for First-Order Functions. In S. R. D. Rocca, editor, *Typed Lambda Calculi and Applications (TLCA'2007), Paris, France*, volume 4583 of *LNCS*, pages 351–366. Springer, 2007.
12. A. Tamalet, O. Shkaravska, and M. van Eekelen. Size Analysis of Algebraic Data Types. In M. Morazán, editor, *Selected Papers of the* 9^{th} *International Symposium on Trends in Functional Programming (TFP'08)*. Intellect Publishers. 2008, to appear.
13. M. van Eekelen, O. Shkaravska, R. van Kesteren, B. Jacobs, E. Poll, and S. Smetsers. AHA: Amortized Heap Space Usage Analysis. In M. Morazán, editor, *Selected Papers of the* 8^{th} *International Symposium on Trends in Functional Programming (TFP'07), New York, USA*, pages 36–53. Intellect Publishers, UK, 2007.
14. R. van Kesteren, O. Shkaravska, and M. van Eekelen. Inferring static non-monotonically sized types through testing. In *Proceedings of* 16^{th} *International Workshop on Functional and (Constraint) Logic Programming (WFLP'07), Paris, France*, volume 216C of *ENTCS*, pages 45–63, 2007.
15. P. B. Vasconcelos. *Space Cost Analysis Using Sized Types*. PhD thesis, School of Computer Science, University of St. Andrews, August 2008.

A Examples

A.1 insert

```
insert(g, x, y) =
match y with | Nil ⇒ let z = Nil in Cons(x, z)
             | Cons(hd, tl) ⇒ if g(z, hd) then y else let z′ = insert(g, z, tl) in Cons(hd, z)
```

The inferred t.r.s.: $n = 0 \vdash f(n) \to 1$
$n \geq 1 \vdash f(n) \to n \,|\, f(n-1) + 1$

Checking if the family $\{n + i\}_{0 \leq i \leq 1}$ approximates the solution of the t.r.s.

$$\begin{array}{ll} n = 0 & \vdash 1 = n{+}?i \\ n \geq 1,\ 0 \leq j' \leq 1 & \vdash n = n{+}?i \\ n \geq 1,\ 0 \leq j \leq 1 & \vdash (n-1) + j = n{+}?i \end{array}$$

Inferring the family $\{n + i\}_{0 \leq i \leq 1}$. The function insert has a polynomial lower $p_{\min}(n)$ and an upper $p_{\max}(n)$ size bounds. It is easy to see that the *strict* ones (i.e. the g.l.b. and the l.u.b.) are equal to n and $n + 1$ respectively. We show briefly how to infer them. First note, that for insert, given any n, there is an evaluation path for $p(n)$ that evaluates to $p_{\min}(n)$, and there is a path that evaluates to $p_{\max}(n)$. Now, assume that $p_{\min}$ and $p_{\max}$ are linear, that is are of the form $a_{\min}n + b_{\min}$ and $a_{\max}n + b_{\max}$ respectively. We want to find the unknown coefficients $a_{\min},\ b_{\min}, a_{\max},\ b_{\max}$ (as we have done it in [11] for strict polynomial size dependencies, where the lower and upper bounds have been equal). To reconstruct $p_{\min}$, one needs to know two points on its graph and the same for $p_{\max}$. Take $n = 1$ and $n = 2$. Evaluating t.r.s. gives $p(1) = \{1,\ 2\}$ and $p(2) = \{2,\ 3\}$. We pick up the minimal values from $p(1)$ and $p(2)$ to see that the graph of $p_{\min}$ contains points (1, 1) and (2, 2). Similarly, we pick up the maximal values of $p(1)$ and $p(2)$ to see that $p_{\max}$ contains (1, 2) and (2, 3). We obtain two systems of equations, for $a_{\min},\ b_{\min}$ and $a_{\max},\ b_{\max}$, respectively:

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

Solving these linear systems we obtain that $a_{\min} = 1$, $b_{\min} = 0$ and $a_{\max} = 1$, $b_{\max} = 1$. Thus, we reconstruct the closed formulae for $p_{\min}(n) = n$ and $p_{\max}(n) = n + 1$.

The output size dependency for insert is $p_{min}(n) + i$, where $0 \leq i \leq \delta(n)$ with $\delta(n) = p_{\max}(n) - p_{\min}(n) = 1$, that is we have reconstructed the family $\{n + i\}_{0 \leq i \leq 1}$.

A.2 rinsert

```
rinsert(g, z, y) =
match x with | Nil ⇒ y
             | Cons(hd, tl) ⇒ let z = rinsert(g, tl, y) in insert(g, hd, z)
```

The inferred t.r.s.:
$$n = 0 \vdash f(n,m) \to m$$
$$n \geq 1, 0 \leq i \leq 1 \vdash f(n,m) \to f_{\mathsf{insert}}(f(n-1,m))$$
$$\text{approximated } n \geq 1, 0 \leq i \leq 1 \vdash f(n,m) \to f(n-1,m) + i$$

Checking if the family $\{m+i\}_{0\leq i\leq n}$ approximates the solution of the t.r.s.

$$n = 0 \qquad \vdash m = m{+}?i$$
$$n \geq 1,\ 0 \leq j' \leq 1,\ 0 \leq j \leq n-1 \vdash m + j + j' = m{+}?i$$

Inferring the family $\{m+i\}_{0\leq i\leq n}$. Assume that $p_{\min}$ and $p_{\max}$ are linear polynomials of two variables. To reconstruct $p_{\min}$ and $p_{\max}$ consider all possible evaluation paths for p at the nodes $(1,1)$, $(2,1)$, $(1,2)$, recalling that $0 \leq j' \leq 1$ and $p(0,m) = m$:

$$p(1,1) = p(0,\ 1) + j' = \{1,\ 2\}$$
$$p(2,1) = p(1,\ 1) + j' = \{1,\ 2,\ 3\}$$
$$p(1,2) = p(0,\ 2) + j' = \{2,\ 3\}$$

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

$$\begin{aligned} a_{10} + a_{01} + a_{00} &= 1 \\ 2a_{10} + a_{01} + a_{00} &= 1 \\ a_{10} + 2a_{01} + a_{00} &= 2 \end{aligned}$$

Solving this system gives that $a_{10} = a_{00} = 0$, $a_{01} = 1$, that is $p_{\min}(n,m) = m$.

For the coefficients of $p_{\max}$ one has the system

$$\begin{aligned} a_{10} + a_{01} + a_{00} &= 2 \\ 2a_{10} + a_{01} + a_{00} &= 3 \\ a_{10} + 2a_{01} + a_{00} &= 3 \end{aligned}$$

Solving this system gives that $a_{00} = 0$, $a_{10} = a_{01} = 1$, that is $p_{\max}(n,m) = n{+}m$. We have reconstructed the family $\{m+i\}_{0\leq i\leq n}$.

A.3 delete

```
delete(g, z, l) =
match l with | Nil ⇒ Nil
             | Cons(hd, tl) ⇒ if g(z, hd) then tl else let z′ = delete(g, z, tl) in Cons(hd, z′)
```

The inferred t.r.s.:
$$n = 0 \vdash f(n) \to 0$$
$$n \geq 1 \vdash f(n) \to n-1 \,|\, f(n-1) + 1$$

Checking if the family $\{\max_0(n-i)\}_{0\le i\le 1}$ approximates the solution of the t.r.s.

$$\begin{array}{ll} n=0 & \vdash 0=\max_0(n-?i)\\ n\ge 1,\ 0\le j\le 1 & \vdash n-1=\max_0(n-?i)\\ n\ge 1,\ 0\le j\le 1 & \vdash \max_0((n-1)-j)=\max_0(n-?i)\end{array}$$

Inferring the family $\{\max_0(n-i)\}_{0\le i\le 1}$. Assume that $p_{\min}$ and $p_{\max}$ are linear polynomials of one variable. To reconstruct $p_{\min}$ and $p_{\max}$ consider all possible evaluation paths for p at the nodes (1), (2), recalling that $p(0)=0$:

$$\begin{aligned} p(1)&=\Big(0 \mid p(0)+1\Big)=\{0,\,1\}\\ p(2)&=\Big(1 \mid p(1)+1\Big)=\{1,\,2\}\end{aligned}$$

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

$$\begin{aligned} a_1+a_0\ &=0\\ 2a_1+a_0&=1\end{aligned}$$

Solving this system gives that $a_1=1,\ a_0=-1$, that is $p_{\min}(n)=n-1$.

For the coefficients of $p_{\max}$ one has the system

$$\begin{aligned} a_1+a_0\ &=1\\ 2a_1+a_0&=2\end{aligned}$$

Solving this system gives that $a_0=1$, $a_1=0$, that is $p_{\max}(n)=n$ on $n\ge 1$. So, $n\ge 1$: $\delta(n)=p_{\max}(n)-p_{\min}(n)=1$, $p_{\min}(n)+i=n-1+i$.

We have reconstructedthe family $\{\max_0(n-i)\}_{0\le i\le 1}$.

A.4 rdelete

```
rdelete(g, l1, l2) =
match l1 with | Nil ⇒ l2
              | Cons(hd, tl) ⇒ let l' = rdelete(g, tl, l2) in delete(g, hd, l')
```

The inferred t.r.s.:

$$\begin{array}{ll} & n=0\vdash f(n,m)\to 0\\ & n\ge 1\vdash f(n,m)\to f_{\mathsf{delete}}(f(n-1,m))\\ \text{approximated} & n\ge 1\vdash f(n,m)=\max_0(f(n-1,m)-j)\end{array}$$

Checking if the family $\{\max_0(m-i)\}_{0\le i\le n}$ approximates the solution of the t.r.s.

$$\begin{array}{ll} n=0 & \vdash 0=\max_0(n-?i)\\ n\ge 1,\ 0\le j'\le 1,\ 0\le j\le n-1 & \vdash \max_0(\max_0(m-1)-j')=\max_0(m-?i)\end{array}$$

Inferring the family $\{\max_0(m-i)\}_{0\le i\le n}$. Assume that $p_{\min}$ and $p_{\max}$ are linear polynomials of two variables. To reconstruct $p_{\min}$ and $p_{\max}$ consider all possible evaluation paths for p at the nodes $(1,1)$, $(2,1)$, $(1,2)$, recalling that $0 \le j' \le 1$ and $p(0,m) = m$:

$$\begin{array}{l} p(1,1) = p(0,1) - j' = \{1,\, 0\} \\ p(2,1) = p(1,1) - j' = \{1,\, 0,\, -1\} \\ p(1,2) = p(0,2) - j' = \{2,\, 1\} \end{array}$$

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

$$\begin{array}{l} a_{10} + a_{01} + a_{00} \;\; = 0 \\ 2a_{10} + a_{01} + a_{00} = -1 \\ a_{10} + 2a_{01} + a_{00} = 1 \end{array}$$

Solving this system gives that $a_{10} = -1$, $a_{00} = 0$, $a_{01} = 1$, that is $p_{\min}(n,m) = m - n$.

For the coefficients of $p_{\max}$ one has the system

$$\begin{array}{l} a_{10} + a_{01} + a_{00} \;\; = 1 \\ 2a_{10} + a_{01} + a_{00} = 1 \\ a_{10} + 2a_{01} + a_{00} = 2 \end{array}$$

Solving this system gives that $a_{01} = a_{00} = 0$, $a_{10} = 1$, that is $p_{\max}(n,m) = m$.

So, $\delta(n,m) = p_{\max}(n,m) - p_{\min}(n,m) = n$, $p_{\min} + i = (m-n) + i$. We have reconstructed the family $\{\max_0(m-i)\}_{0\le i\le n}$.

A.5 deleteall

```
deleteall(g, x, y) =
match y with | Nil ⇒ Nil
             | Cons(hd, tl) ⇒ if g(x, hd) then
                                deleteall(g, x, tl)
                              else let z = deleteall(g, x, tl) in Cons(hd, z)
```

The inferred t.r.s.: $n = 0 \vdash f(n) \to 0$
$n \ge 1 \vdash f(n) \to f(n-1) \,|\, f(n-1) + 1$

Checking if the family $\{i\}_{0\le i\le n}$ approximates the solution of the t.r.s.

$$\begin{array}{ll} n = 0 & \vdash 0 = ?i \\ n \ge 1,\ 0 \le j \le n-1 & \vdash j - 1 = ?i \\ n \ge 1,\ 0 \le j \le 1 & \vdash j - 1 + 1 = ?i \end{array}$$

Inferring the family $\{i\}_{0\le i\le n}$. Assume that $p_{\min}$ and $p_{\max}$ are linear polynomials of one variable. To reconstruct $p_{\min}$ and $p_{\max}$ consider all possible evaluation paths for p at the nodes (1), (2), recalling that $p(0) = 0$:

$$p(1) = \Big(p(0) \mid p(0) + 1\Big) = \{0, 1\}$$
$$p(2) = \Big(p(1) \mid p(1) + 1\Big) = \{0, 1, 2\}$$

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

$$\begin{aligned} a_1 + a_0 &= 0 \\ 2a_1 + a_0 &= 0 \end{aligned}$$

Solving this system gives that $a_1 = a_0 = 0$, that is $p_{\min}(n) = 0$.

For the coefficients of $p_{\max}$ one has the system

$$\begin{aligned} a_1 + a_0 &= 1 \\ 2a_1 + a_0 &= 2 \end{aligned}$$

Solving this system gives that $a_0 = 0$, $a_1 = 1$, that is $p_{\max}(n) = n$.

So, $\delta(n) = p_{\max}(n) - p_{\min}(n) = n$, $p_{\min} + i = i$ and we reconstruct the family $\{i\}_{0\le i\le n}$.

A.6 filter

```
filter(g, x) =
match x with | Nil ⇒ Nil
             | Cons(hd, tl) ⇒ if g(hd) then let z = filter(g, tl) in Cons(hd, z) else filter(g, tl)
```

The inferred t.r.s.: $n = 0 \vdash f(n) \to 0$
$n \ge 1 \vdash f(n) \to f(n-1) + 1 \,|\, f(n-1)$

Checking if the family $\{i\}_{0\le i\le n}$ approximates the solution of the t.r.s.

$$\begin{array}{ll} n = 0 & \vdash 1 = n + ?i \\ n \ge 1,\ 0 \le j \le j - 1 & \vdash j + 1 = ?i \\ n \ge 1 & \vdash j = ?i \end{array}$$

Inferring the family $\{i\}_{0\le i\le n}$. Assume that $p_{\min}$ and $p_{\max}$ are linear polynomials of one variable. To reconstruct $p_{\min}$ and $p_{\max}$ consider all possible evaluation paths for p at the nodes (1), (2), recalling that $p(0) = 0$:

$$p(1) = \Big(p(0) + 1 \mid p(0)\Big) = \{0, 1\}$$
$$p(2) = \Big(p(1) + 1 \mid p(1)\Big) = \{0, 1, 2\}$$

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

$$\begin{aligned} a_1 + a_0 &= 0 \\ 2a_1 + a_0 &= 0 \end{aligned}$$

Solving this system gives that $a_1 = a_0 = 0$, that is $p_{\min}(n) = 0$.

For the coefficients of $p_{\max}$ one has the system

$$\begin{aligned} a_1 + a_0 &= 1 \\ 2a_1 + a_0 &= 2 \end{aligned}$$

Solving this system gives that $a_0 = 0$, $a_1 = 1$, that is $p_{\max}(n) = n$.

So, $\delta(n) = p_{\max}(n) - p_{\min}(n) = n$, $p_{\min} + i = i$. We have reconstructed the family $\{i\}_{0 \le i \le n}$.

A.7 divtwo

```
divtwo(x) =
match x with | Nil ⇒ Nil
             | Cons(hd, tl) ⇒ match tl with | Nil ⇒ Nil
                                            | Cons(hd′, tl′) ⇒
                              let y = divtwo(tl′) in Cons(hd, y)
```

The inferred t.r.s.: $\begin{aligned} n = 0 \quad &\vdash f(n) = 0 \\ n \ge 1,\ n - 1 = 0 &\vdash f(n) \to 0 \\ n \ge 1, n - 1 \ge 1 &\vdash f(n) \to f(n-2) + 1 \end{aligned}$

Checking if the family $\{\frac{\max_0(n-i)}{2}\}_{0 \le i \le 1}$ approximates the solution of the t.r.s.

$$\begin{aligned} n = 0 \quad &\vdash f(n) = 0 \\ n = 1 \quad &\vdash f(n) = 0 \\ n \ge 1,\ 0 \le j \le 1 &\vdash \frac{\max_0(n-j)}{2} + 1 = \frac{\max_0(n-?i)}{2} \end{aligned}$$

Inferring the family $\{n+i\}_{0 \le i \le 1}$. This is an example of the size dependency where there are n, such that there is no evaluation path for $p(n)$ that evaluates to $p_{\min}$ and similarly for $p_{\max}$. (In this case, $p(n)$ evaluates to $p_{\min} = \frac{n-1}{2}$ only on odd n and $p(n)$ evaluates to $p_{\min} = \frac{n}{2}$ only on even n.)

Now we relax the condition "for each n there is an evaluation path for $p(n)$ that evaluates to $p_{\min}(n)$ and an evaluation path that evaluates to $p_{\max}(n)$". We consider the following relaxation: "There is K such that for each n there are $0 \le k_1, k_2 \le K$, such that there is an evaluation path for $p(n + k_1)$ that

evaluates to $p_{\min}(n+k_1)$ and an evaluation path $p(n+k_2)$ that evaluates to $p_{\max}(n)$". So, to infer the type for such size dependencies, one needs to assume not only the degree of $p_{\min}$ and $p_{\max}$, but the number K as well.

Assume that $p_{\min}$ and $p_{\max}$ are linear polynomials of one variable. Assume that $K=1$. To reconstruct $p_{\min}$ and $p_{\max}$ consider all possible evaluation paths for p at the nodes $(2+i)$, $(2+(K+1)+j)$, with $0 \le i,j \le K$, recalling that $p(0)=p(1)=0$:

$$\begin{array}{l} p(2) = 1 + p(0) = \{1\} \\ p(3) = 1 + p(1) = \{1\} \\ p(4) = 1 + p(2) = \{2\} \\ p(5) = 1 + p(3) = \{2\} \end{array}$$

According to the condition, the graph of $p_{\min}$:

- given $n=2$ contains $p(2)$ or $p(3)$,
- given $n=4$ contains $p(4)$ or $p(5)$.

Thus, for the coefficients of $p_{\min}$ solves one of the following 4 systems

$$\begin{array}{ll} \text{via points} \quad p(2),\, p(4): & \text{via points} \quad p(2),\, p(5): \\ 2a_1 + a_0 = 1 & 2a_1 + a_0 = 1 \\ 4a_1 + a_0 = 2 & 5a_1 + a_0 = 2 \end{array}$$

$$\begin{array}{ll} \text{via points} \quad p(3),\, p(4): & \text{via points} \quad p(3),\, p(5): \\ 3a_1 + a_0 = 1 & 3a_1 + a_0 = 1 \\ 4a_1 + a_0 = 2 & 5a_1 + a_0 = 2 \end{array}$$

Solving these systems gives that

- for $p(2)$, $p(4)$ one has $a_1 = \dfrac{1}{2},\ a_0 = 0$, that is $p_{\min}(n) = \dfrac{n}{2}$
- for $p(2)$, $p(5)$ one has $a_1 = \dfrac{1}{3},\ a_0 = \dfrac{1}{3}$, that is $p_{\min}(n) = \dfrac{n}{3} + \dfrac{1}{3}$
- for $p(3)$, $p(4)$ one has $a_1 = 1,\ a_0 = -2$, that is $p_{\min}(n) = \dfrac{n}{2}$
- for $p(3)$, $p(4)$ one has $a_1 = \dfrac{1}{2},\ a_0 = -\dfrac{1}{2}$, that is $p_{\min}(n) = \dfrac{n}{2} - \dfrac{1}{2}$

The same is for $p_{\max}$. So, there are $4^2 = 16$ possible pairs $(p_{min},\, p_{\max})$ to check.Each pair must be presented in the form of collections $\dfrac{p'_{\min}(n)+i}{d}$ where $0 \le i \le \delta(n)$ is integer, and $p'_{\min}(n),\ \delta(n)$ are integer polynomials. For instance, for the (correct) pair $(\dfrac{n}{2} - \dfrac{1}{2}, \dfrac{n}{2})$ one first notices that the pair defines the collection of $\dfrac{n}{2} - \dfrac{1}{2} + j$, with $j = 0,\ \dfrac{1}{2}$. It is easy to see that this is the same as $\dfrac{n-1+2j}{2}$, with $j = 0,\ \dfrac{1}{2}$. By putting $j = 2i-1$ one presents the same collection in the form $\dfrac{n-i}{2}$ with $0 \le j \le 1$.

Case Study: Haskell's List Library

As a case study we applied our type inference procedure to Haskell's List Library, in particular to the implementation Hugs version September 2006.

Assumptions

Since for reasons of simplicity we have work with a basic language, we need to do some assumptions.

- It must be possible to translate the function into our language. Since our type system is more restrictive, sometimes we cannot define an equivalent function for all cases. Take for instance transpose: $\mathsf{L}_n(\mathsf{L}_m(\alpha)) \to \mathsf{L}_m(\mathsf{L}_n(\alpha))$; Firstly, in our language it must be undefined for the empty list because in that case m is undefined. Secondly, the type system requires that the inner lists have all the same length, which is not the case in the Haskell version.
- We ignore classes of types like Eq and Ord.
- We write the functions uncurrified.
- We have changed some function names. We the function (!!) is called atIndex and (\\) is called rdelete.

Functions that do not return lists

The following functions do not return lists and thus are not interesting from the size dependency point of view:

head	$: \mathsf{L}_n(\alpha) \to \alpha$
null	$: \mathsf{L}_n(\alpha) \to \mathsf{Bool}$
length	$: \mathsf{L}_n(\alpha) \to \mathsf{Int}$
and, or	$: \mathsf{L}_n(\mathsf{Bool}) \to \mathsf{Bool}$
any, all	$: (\alpha \to \mathsf{Bool}) \times \mathsf{L}_n(\alpha) \to \mathsf{Bool}$
sum, product	$: \mathsf{L}_{Int}(\alpha) \to \mathsf{Int}$
maximum, minimum	$: \mathsf{L}_{Int}(\alpha) \to \mathsf{Int}$
isPrefix, isSuffix, isInfix	$: \mathsf{L}_n(\alpha) \times \mathsf{L}_m(\alpha) \to \mathsf{Bool}$
elem, notElem	$: \alpha \times \mathsf{L}_n(\alpha) \to \mathsf{Bool}$
atIndex	$: \mathsf{L}_n(\alpha) \times \mathsf{Int} \to \alpha$

Functions with exact size relation

For the following functions, the size of the output is can be expressed with a unique polynomial with respect to the size of the inputs. These functions are typable in the type systems developed in [11, 12] (except the one using max_0).

$$
\begin{array}{ll}
\mathsf{append} & : \mathsf{L}_n(\alpha) \times \mathsf{L}_m(\alpha) \to \mathsf{L}_{n+m}(\alpha) \\
\mathsf{tail} & : \mathsf{L}_n(\alpha) \to \mathsf{L}_{n-1}(\alpha) \\
\mathsf{init} & : \mathsf{L}_n(\alpha) \to \mathsf{L}_{n-1}(\alpha) \\
\mathsf{map} & : (\alpha \to \beta) \times \mathsf{L}_n(\alpha) \to \mathsf{L}_n(\beta) \\
\mathsf{reverse} & : \mathsf{L}_n(\alpha) \to \mathsf{L}_n(\alpha) \\
\mathsf{intersperse} & : \alpha \times \mathsf{L}_n(\alpha) \to \mathsf{L}_{2*n \dot{-} 1}(\alpha) \\
\mathsf{transpose} & : \mathsf{L}_n(\mathsf{L}_m(\alpha)) \to \mathsf{L}_m(\mathsf{L}_n(\alpha)) \\
\mathsf{concat} & : \mathsf{L}_n(\mathsf{L}_m(\alpha)) \to \mathsf{L}_{n*m}(\alpha) \\
\mathsf{scanl} & : (\alpha \times \beta \to \alpha) \times \alpha \times \mathsf{L}_n(\beta) \to \mathsf{L}_{n+1}(\alpha) \\
\mathsf{scanl1} & : (\alpha \times \alpha \to \alpha) \times \mathsf{L}_n(\alpha) \to \mathsf{L}_n(\alpha) \\
\mathsf{union} & : \mathsf{L}_n(\alpha) \times \mathsf{L}_m(\alpha) \to \mathsf{L}_{n+m}(\alpha) \\
\mathsf{unionBy} & : (\alpha \times \alpha \to \mathsf{Bool}) \times \mathsf{L}_n(\alpha) \times \mathsf{L}_m(\alpha) \to \mathsf{L}_{n+m}(\alpha) \\
\mathsf{sort} & : \mathsf{L}_n(\alpha) \to \mathsf{L}_n(\alpha) \\
\mathsf{insert} & : \mathsf{L}_n(\alpha) \to \mathsf{L}_{n+1}(\alpha) \\
\mathsf{intersection} & : \mathsf{L}_n(\alpha) \times \mathsf{L}_m(\alpha) \to \mathsf{L}_{n+(m \dot{-} n)}(\alpha) \\
\mathsf{intersectBy} & : (\alpha \times \alpha \to \mathsf{Bool}) \times \mathsf{L}_n(\alpha) \times \mathsf{L}_m(\alpha) \to \mathsf{L}_{n+(m \dot{-} n)}(\alpha)
\end{array}
$$

Functions with indexed family size relation

For these functions definitions the size dependency is not exact and thus they need a family of polynomials to express the different possible sizes of the output.

$$
\begin{array}{ll}
\mathsf{takeWhile} & : (\alpha \to \mathsf{Bool}) \to \mathsf{L}_n(\alpha) \to \mathsf{L}_{\{i\}_{0 \le i \le n}}(\alpha) \\
\mathsf{dropWhile} & : (\alpha \to \mathsf{Bool}) \to \mathsf{L}_n(\alpha) \to \mathsf{L}_{\{i\}_{0 \le i \le n}}(\alpha) \\
\mathsf{inits},\ \mathsf{tails} & : \mathsf{L}_n(\alpha) \to \mathsf{L}_{n+1}(\mathsf{L}_{\{i\}_{0 \le i \le n}}(\alpha)) \\
\mathsf{filter} & : (\alpha \to \mathsf{Bool}) \to \mathsf{L}_n(\alpha) \to \mathsf{L}_{\{i\}_{0 \le i \le n}}(\alpha) \\
\mathsf{elemIndices} & : \alpha \times \mathsf{L}_n(\alpha) \to \mathsf{L}_{\{i\}_{0 \le i \le n}}(\mathsf{Int}) \\
\mathsf{findIndices} & : (\alpha \to \mathsf{Bool}) \times \mathsf{L}_n(\alpha) \to \mathsf{L}_{\{i\}_{0 \le i \le n}}(\mathsf{Int}) \\
\mathsf{nub} & : \mathsf{L}_n(\alpha) \to \mathsf{L}_{\{i\}_{0 \le i \le n}}(\alpha) \\
\mathsf{nubBy} & : (\alpha \times \alpha \to \mathsf{Bool}) \times \mathsf{L}_n(\alpha) \to \mathsf{L}_{\{i\}_{0 \le i \le n}}(\alpha) \\
\mathsf{delete} & : \alpha \times \mathsf{L}_n(\alpha) \to \mathsf{L}_{\{\max_0(n-i)\}_{0 \le i \le 1}}(\alpha) \\
\mathsf{deleteBy} & : (\alpha \times \alpha \to \mathsf{Bool}) \times \mathsf{L}_n(\alpha) \to \mathsf{L}_{\{\max_0(n-i)\}_{0 \le i \le 1}}(\alpha) \\
\mathsf{rdelete} & : \mathsf{L}_n(\alpha) \times \mathsf{L}_m(\alpha) \to \mathsf{L}_{\{\max_0(n-i)\}_{0 \le i \le m}}(\alpha) \\
\mathsf{deleteFirstsBy} & : (\alpha \times \alpha \to \mathsf{Bool}) \times \mathsf{L}_n(\alpha) \times \mathsf{L}_m(\alpha) \to \mathsf{L}_{\{\max_0(n-i)\}_{0 \le i \le m}}(\alpha)
\end{array}
$$

Functions that use other datatypes

The language presented in this article has only the basic types Bool, Int and polymorphic lists. However, it is possible to add pairs and algrebraic data types following ideas from [12]. In particular we need sized naturals, Nat, and Maybe. With these additions we would be able to deal with many more functions.

$$
\begin{array}{ll}
\mathsf{mapAccumL},\ \mathsf{mapAccumR} & : (\alpha \to \beta \to (\alpha, \gamma)) \times \alpha \times \mathsf{L}_n(\beta) \to (\alpha, \mathsf{L}_n(\gamma)) \\
\mathsf{span},\ \mathsf{break},\ \mathsf{partition} & : (\alpha \to \mathsf{Bool}) \to \mathsf{L}_n(\alpha) \to (\mathsf{L}_{\{i\}_{0 \le i \le n}}(\alpha),\ \mathsf{L}_{\{i\}_{0 \le i \le n}}(\alpha))
\end{array}
$$

The type of a span, does not capture the fact that the sum of the length of the resulting lists is equal to the length of the input list. This could be achieved if the index is part of the pair data structure and is shared by its elements. Then we could express

$$\mathsf{span}, \mathsf{break}, \mathsf{partition}\colon (\alpha \to \mathsf{Bool}) \to \mathsf{L}_n(\alpha) \to (\mathsf{L}_{\{i\}}(\alpha),\ \mathsf{L}_{\{n-i\}}(\alpha))_{\{0 \le i \le n\}}$$

The following types do not use this extension since it is not covered by our inference procedure. We write $\mathsf{Nat}(n)$ to denote a natural number to which we assign the symbolic value n. This value can be used in size annotation of the output.

$$\begin{array}{ll}
\mathsf{replicate} & : \mathsf{Nat}(n) \times \alpha \to \mathsf{L}_n(\alpha) \\
\mathsf{take} & : \mathsf{Nat}(n) \times \mathsf{L}_m(\alpha) \to \mathsf{L}_{\{i\}_{0 \le i \le n}}(\alpha) \\
\mathsf{drop} & : \mathsf{Nat}(n) \times \mathsf{L}_m(\alpha) \to \mathsf{L}_{max_0(m-n)}(\alpha) \\
\mathsf{splitAt} & : \mathsf{Nat}(n) \times \mathsf{L}_m(\alpha) \to (\mathsf{L}_{\{i\}_{0 \le i \le n}}(\alpha), \mathsf{L}_{\{max_0(m-i)\}_{0 \le i \le n}}(\alpha)) \\
\mathsf{zip} & : \mathsf{L}_n(\alpha) \times \mathsf{L}_m(\beta) \to \mathsf{L}_{max_0(a-max_0(b-a))}((\alpha,\ \beta)) \\
\mathsf{unzip} & : \mathsf{L}_n((\alpha,\ \beta)) \to (L(a,n), L(b,n))
\end{array}$$

We could also specify a more restricted type for zip where we require both lists to have the same length:

$$\mathsf{zip}\colon \mathsf{L}_n(\alpha) \to \mathsf{L}_n(\beta) \to \mathsf{L}_n((\alpha,\ \beta))$$

Functions not covered by our type system

There are functions whose sized types cannot be expressed in our type system. We can divide three in four categories:

- *The size of the result cannot be determined statically.* These functions create lists in a way such that it is not possible to determine it lenght statically, because it depends on the value of the arguments. In the list library we find unfoldr, whose Haskell type is $(\beta \to \mathsf{Maybe}(\alpha,\ \beta)) \times \beta \to \mathsf{L}(\alpha)$.
- *The size of the output depends on a higher-order parameter.* The only example in the list library is concatMap, whose type could be expressed as $(\alpha \to \mathsf{L}_m(\beta)) \times \mathsf{L}_n(\alpha) \to \mathsf{L}_{n*m}(\beta)$. However, we do not allow such size dependencies in our type system to keep tractability of our type inference procedure.
- *The size of the output is infinite* We work only with finite lists, hence we cannot deal with functions that return infinite lists. If ∞ represents and infinite length, we could say that

$$\begin{array}{ll}
\mathsf{iterate} & : (\alpha \to \alpha) \times \alpha \to \mathsf{L}_\infty(\alpha) \\
\mathsf{repeat} & : \alpha \to \mathsf{L}_\infty(\alpha) \\
\mathsf{cycle} & : \mathsf{L}_n(\alpha) \to \mathsf{L}_\infty(\alpha)
\end{array}$$