FOUNDATIONAL ASPECTS OF SIZE ANALYSIS: EXERCISES

OLHA SHKARAVSKA, MARKO VAN EEKELEN, AND ALEJANDRO TAMALET

Institute for Computing and Information Sciences, Radboud University Nijmegen
e-mail address: shkarav@cs.ru.nl

Institute for Computing and Information Sciences, Radboud University Nijmegen
e-mail address: marko@cs.ru.nl

Institute for Computing and Information Sciences, Radboud University Nijmegen
e-mail address: tamalet@cs.ru.nl

Introduction

From the point of view of a mathematician resource analysis of programs (function definitions) amounts to verifying satisfiability of arithmetic predicates for checking size dependencies and solving recurrences for inference. The exercises we are going to do should give you a flavor of these two aspects.

At the end we will see how our methodology is extended to *non-shapely* programs with polynomial lower and upper bounds.

This set of exercises is organized in three main sections. In Section 1 we will learn how to type check shapely function definitions, i.e. function definitions for which the size of the output can be described by a polynomial where the variables are the sizes of the inputs. We will work with simple programs whose only data types are integer and lists, taking the size of a list to be its length. In the Section 2 we will see how size functions of such programs may be inferred by test-based inference procedure. Finally, Section 3 is devoted to non-shapely programs with lower and upper polynomial size bounds. In its first part you will see how bounds are inferred and checked for function definitions over lists $\mathsf{L}_n(\alpha)$ or matrix-like structures, like $\mathsf{L}_n(\mathsf{L}_m(\alpha))$ etc. In its second part we will study how the method can be extended to general programs over lists, where the sizes of internal lists may be different. Given the limited amount of time, this section is consider optional.

Each section contains a detailed *how to* example and a some exercises. Exercises marks with (*) are more challenging.

This research is sponsored by the Netherlands Organisation for Scientific Research (NWO), project Amortised Heap Space Usage Analysis (AHA), grant number 612.063.511.

1. Type Checking For Shapely Function Definitions

As a proof of concept we have built a simple tool to type check and infer the size-aware type of programs written in a simple first-order functional language, whose only types are integers and lists.

Exercise 1. Your first task is to investigate the demonstrator (demo for short). Go to `http://www.aha.cs.ru.nl/`, where you can learn more about the AHA project, and follow the link *Demonstrator of polynomial size aware type checker and run-time test-based type inference* and then *type-checking and type inference environment.*

(1) Look through the list of examples on the right side.
(2) Investigate the options and the language.
(3) What is the option *Types are annotated with size parameters* for?

As you probably have noticed, in the current version of the tool the language (and the interface) is very simple. Some tips that may be useful for the next exercises:

- There are no booleans, so the condition of an if is an integer variable,
- if statements end with fi,
- Integer operations are written in prefix form, e.g., $+(2, 3)$ (ouch!)
- Functions are defined via letfun followed by in, and an expression to evaluate, which can be another function-definition-expression.
- Arguments of functions must be variable, so if we want to pass an expression as argument, we must first bind it to a variable via let.

1.1. A ***how to*** example: append.

Now we will learn how checking is done in more detail, "on a paper". We start with a simple function append that appends two lists.

$$\begin{aligned}
&\mathsf{append}(l_1, l_2) \colon \mathsf{L}_n(\alpha) \times \mathsf{L}_m(\alpha) \to \mathsf{L}_{n+m}(\alpha) = \\
&\quad \mathsf{match}\ l_1\ \mathsf{with} \mid \mathsf{nil} \Rightarrow l_2 \\
&\qquad\qquad\qquad\quad \mid \mathsf{cons}(hd, tl) \Rightarrow \mathsf{let}\ l = \mathsf{append}(tl, l_2)\ \mathsf{in}\ \mathsf{cons}(hd, l)
\end{aligned}$$

For the sake of convenience we denote the body of append via e_{append}.

1.1.1. *Construct type derivation tree in the backward style.*

We explain how to do that step-by-step, so the tree is in the verbose form:

(1) We start with the whole body of the function that defines our *main goal*: to prove $l_1 \colon \mathsf{L}_n(\alpha), l_2 \colon \mathsf{L}_m(\alpha) \vdash_\Sigma e_{\mathsf{append}} \colon \mathsf{L}_{n+m}(\alpha)$.

    A signature Σ contains the type we are checking: $\Sigma(\mathsf{append}) = \mathsf{L}_n(\alpha) \times \mathsf{L}_m(\alpha) \to \mathsf{L}_{n+m}(\alpha)$. We will see how this type will be used in the recursive call.

    The body of the function is a pattern-matching expression. Therefore, we have to prove two *subgoals* that correspond to the nil- and cons-branches respectively.

(2) Applying the Match-rule yields first the Nil-branch subgoal:

$$n = 0;\ l_1 \colon \mathsf{L}_n(\alpha), l_2 \colon \mathsf{L}_m(\alpha) \vdash_\Sigma l_2 \colon \mathsf{L}_{n+m}(\alpha)$$

(3) Continue with the nil-branch. The expression in the previous subgoal is a program variable l_2, so we apply the Var-rule, which is an axiom. We obtain the following subgoal:

$$n = 0 \vdash \mathsf{L}_{n+m}(\alpha) = \mathsf{L}_m(\alpha)$$

(4) Unfold the definition of a type equivalence above. We obtain a first-order entailment

$$n = 0 \vdash n + m = m$$

which after substitution $n = 0$ transforms into the trivially true entailment $\vdash m = m$.

(5) Now follow the cons-branch defined by the MATCH-rule applied to e_{append}.

$$l_1 : \mathsf{L}_n(\alpha), l_2 : \mathsf{L}_m(\alpha) \ \vdash_\Sigma \big(\mathsf{let}\ l = \mathsf{append}(tl, l_2)\ \mathsf{in}\ \mathsf{cons}(hd, l)\big) : \mathsf{L}_{n+m}(\alpha)$$

This is a let-construct, so we apply the LET-rule to obtain two subgoals, corresponding to the let-binding and the let-body respectively.

(6) In the let-binding we have a subgoal

$$tl : \mathsf{L}_{n-1}(\alpha), l_2 : \mathsf{L}_m(\alpha) \ \vdash_\Sigma \mathsf{append}(tl, l_2) : \tau^?$$

where $\tau^?$ is an unknown type, which we will reconstruct on the next step using the appropriate axiom. In the context we omit the program variables hd and l_1, on which the function call does not depend.

(7) The expression in the binding clause is a function call, so we apply FUNAPP, with the known type of append, $\mathsf{L}_n(\alpha) \times \mathsf{L}_m(\alpha) \rightarrow \mathsf{L}_{n+m}(\alpha)$. We substitute the sizes n and m of the formal parameters with the sizes of the actual parameters, $n - 1$ and m respectively. We obtain

$$\vdash \tau^? = \mathsf{L}_{(n-1)+m}(\alpha)$$

Now the type $\tau^?$ is reconstructed and may be used in the let-body.

(8) In the let-body we have a subgoal

$$hd : \alpha, l : \mathsf{L}_{(n-1)+m}(\alpha) \ \vdash_\Sigma \mathsf{cons}(hd, l) : \mathsf{L}_{n+m}(\alpha)$$

Again, in the context we omit the program variables tl, l_1 and l_2 on which the expression in the subgoal does not depend.

(9) Applying the CONS-rule yields $\vdash n + m = (n - 1) + m + 1$ which is trivially true.

In general, type checking shapely function definitions amounts to checking first-order entailments of the form

$$\ldots, g(n_1, \ldots, n_k) = 0, \ldots \vdash p_1(n_1, \ldots, n_k) = p_2(n_1, \ldots, n_k)$$

where g, p_1, p_2 are known polynomials.

Here consider programs where pattern matching is allowed only on program parameters or their tails. Therefore, the polynomial g on the left-hand side of the entailment is of the very simple form $n_i - c_i = 0$, where $1 \le i \le k$ and c_i is an integer constant. Replacing size variables n_i with the corresponding constants on the r.h.s yields an equality of two polynomials. Trivially, it holds if and only if the corresponding polynomial coefficients are equal.

This restriction make type checking decidable. The procedure still works fine for many programs that do not satisfy it, but in theory it can fail to acknowledge legitimate types. Can you imagine why? (*hint*: on the l.h.s. of entailments we may get arbitrary polynomials).

1.2. $\mathsf{conspack}$. Let a function $\mathsf{conspack}$, which inserts an integer into a list, be defined by the following body

$$
\begin{array}{l}
\mathsf{conspack}(x, l) = \\
\mathsf{match}\ l\ \mathsf{with} \mid \mathsf{nil} \Rightarrow \mathsf{cons}(x, l) \\
\qquad\qquad\quad\ \mid \mathsf{cons}(hd, tl) \Rightarrow \mathsf{let}\ y = x - hd \\
\qquad\qquad\qquad\qquad\qquad\qquad \mathsf{in\ if}\ y \quad \mathsf{then\ let}\ l' = \mathsf{conspack}(x, tl) \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathsf{in\ cons}(hd, l') \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathsf{else\ cons}(x, l)
\end{array}
$$

The function $\mathsf{conspack}$ inserts an integer x before the first occurrence $x' \in l$, such that $x = x'$. For instance

- on $2, [\,]$ it returns $[2]$,
- on $2, [1, 3]$ it returns $[1, 3, 2]$,
- on $2, [1, 2, 3]$ it returns $[1, 2, 2, 3]$,
- on $2, [1, 3, 2]$ it returns $[1, 3, 2, 2]$.

Exercise 2.

(1) Write a code for $\mathsf{conspack}$ in our language. What type does it have? Check it using the demo.

(2) Prove on a paper that the function definition for $\mathsf{conspack}$ has the type $\mathtt{Int} \times \mathsf{L}_n(\mathtt{Int}) \to \mathsf{L}_{n+1}(\mathtt{Int})$.

1.3. sqdiff. Let the function sqdiff be defined by the following body:

$$
\begin{array}{l}
\mathsf{sqdiff}(l_1, l_2) = \\
\mathsf{match}\ l_1\ \mathsf{with} \mid \mathsf{nil} \Rightarrow \mathsf{cprod}(l_2, l_2) \\
\qquad\qquad\quad\ \mid \mathsf{cons}(hd_1, tl_1) \Rightarrow \mathsf{match}\ l_2\ \mathsf{with} \mid \mathsf{nil} \Rightarrow \mathsf{cprod}(l_1, l_1) \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad\ \mid \mathsf{cons}(hd_2, tl_2) \Rightarrow \mathsf{sqdiff}(tl_1, tl_2)
\end{array}
$$

You can find the corresponding code in the list of our demo examples.

Exercise 3. Prove on a paper that $\mathsf{sqdiff}\colon \mathsf{L}_n(\alpha) \times \mathsf{L}_m(\alpha) \to \mathsf{L}_{(n-m)^2}(\mathsf{L}_2(\alpha))$. The typing $\mathsf{cprod}\colon \mathsf{L}_n(\alpha) \times \mathsf{L}_m(\alpha) \to \mathsf{L}_{nm}(\mathsf{L}_2(\alpha))$ can be assumed.

1.4. $\mathsf{scalar\_prod}$. Let the function $\mathsf{scalar\_prod}$ that returns a scalar product of two vectors (placed in a 1-element list) be defined by

$$
\begin{array}{l}
\mathsf{scalar\_prod}(l_1, l_2) = \\
\mathsf{match}\ l_1\ \mathsf{with} \mid \mathsf{nil} \Rightarrow \mathsf{match}\ l_2\ \mathsf{with} \mid \mathsf{nil} \Rightarrow \mathsf{cons}(0, \mathsf{nil}) \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \mid \mathsf{cons}(hd_2, tl_2) \Rightarrow \mathsf{scalar\_prod}(l_1, l_2)\ \mathtt{(*\ nontermination\ *)} \\
\qquad\qquad\quad\ \mid \mathsf{cons}(hd_1, tl_1) \Rightarrow \mathsf{match}\ l_2\ \mathsf{with} \mid \mathsf{nil} \Rightarrow \mathsf{scalar\_prod}(l_1, l_2)\ \mathtt{(*\ nontermination\ *)} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \mid \mathsf{cons}(hd_2, tl_2) \Rightarrow \mathsf{let}\ l = \mathsf{scalar\_prod}(tl_1, tl_2) \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad\ \mathsf{in\ let}\ y = hd_1 * hd_2 \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathsf{in\ replace}(y, l)
\end{array}
$$

where $\mathsf{replace} : \mathtt{Int} \times \mathsf{L}_n(\mathtt{Int}) \to \mathsf{L}_n(\mathtt{Int})$ is defined by

$$\begin{aligned}&\mathsf{replace}(x,l) = \\ &\mathsf{match}\ l\ \mathsf{with} \mid \mathsf{nil} \Rightarrow \mathsf{nil} \\ &\qquad\qquad\quad \mid \mathsf{cons}(hd, tl) \Rightarrow \mathsf{cons}(x+hd, tl)\end{aligned}$$

This function, given an integer x and a list l, replaces the head hd of the list with the sum $x + hd$. For instance, on 2, $[1,4]$ it returns $[3,4]$.

Note that $\mathsf{cons}(0, \mathsf{nil})$ is the sugared presentation for $\mathsf{let}\ x = 0\ \mathsf{in}\ \mathsf{let}\ l = \mathsf{nil}\ \mathsf{in}\ \mathsf{cons}(x,l)$. In the demo you can use sugared presentations of composed function calls, which in our basic language is done via nested let-expressions. Nested let-expressions are more convenient to perform checking "on-paper".

Further, it is easy to see that scalar_prod is not defined on two lists of different length.

Assume that the type of the auxiliary function replace is given and checked.

Exercise 4.

(1) Prove using demo and on a paper that $\mathsf{scalar\_prod} : \mathsf{L}_n(\mathtt{Int}) \times \mathsf{L}_n(\mathtt{Int}) \to \mathsf{L}_1(\mathtt{Int})$,
(2) Prove using demo and on a paper that $\mathsf{scalar\_prod} : \mathsf{L}_n(\mathtt{Int}) \times \mathsf{L}_m(\mathtt{Int}) \to \mathsf{L}_1(\mathtt{Int})$ is type checked as well.

2. Type Inference For Shapely Function Definitions

A test-based inference method is described in detail in [5], Section 5. We start with a *how to* example and then you will be inferring size dependencies.

The type inference algorithm is semi-decidable, that is if the size dependency of a function definition is not precisely a polynomial, then the procedure does not terminate. Moreover, the procedure does not terminate if the chosen test-points correspond to data where the function does not terminate.

2.1. A *how to* example: append.

We show how one infers the typing $\mathsf{append} : \mathsf{L}_n(\alpha) \times \mathsf{L}_m(\alpha) \to \mathsf{L}_{n+m}(\alpha)$. More precisely, we are going to *infer size annotations* of this typing, assuming that the underlying typing $\mathsf{append} : \mathsf{L}(\alpha) \times \mathsf{L}(\alpha) \to \mathsf{L}(\alpha)$ is already inferred.

First, go to the demo and untick `Types are annotated with size parameters`. Now you are in the inference mode. Find the example *u*append3, which stays for *un-annotated* append3. Infer its annotation. Together with the annotation for append3 the annotation for append will be inferred.

Now you will see how inference is done in more detail, "on a paper".

(1) The input of the inference procedure is $\mathsf{append} : \mathsf{L}_n(\alpha) \times \mathsf{L}_m(\alpha) \to \mathsf{L}_{p^?(n,m)}(\alpha)$, that is we supply the procedure with the underlying type and the list of the size variables, n, m assigned to the input types of append. The task is to *reconstruct* $p^?(n,m)$.
(2) Here we make an assumption about the degree d of the polynomial $p^?$: let $d = 1$. Note that in the demo size variables are assigned to input types automatically in the most obvious way: each input list type has its own size variable. However, it can be improved, so that different options are tried or a user assigns size variables to input types manually.

    A linear polynomial ($d = 1$) of two variables has *three* coefficients: $p^?(n,m) = a_{10}n + a_{01}m + a_{00}$. Therefore, to compute these coefficients, we must have the values $p(n_1,m_1), p(n_2,m_2), p(n_3,m_3)$ of the polynomial in some three 2-dimensional nodes, $(n_1,m_1), (n_2,m_2), (n_3,m_3)$ such that the system

$$\left.\begin{array}{llll} a_{10}n_1 & + a_{01}m_1 & + a_{00} & = p(n_1, m_1) \\ a_{10}n_2 & + a_{01}m_2 & + a_{00} & = p(n_2, m_2) \\ a_{10}n_3 & + a_{01}m_3 & + a_{00} & = p(n_3, m_3) \end{array}\right\} (1)$$

where a_{10}, a_{01}, a_{00} are variables, has a unique solution.

(3) The system above has a unique solution if the nodes $(n_1, m_1), (n_2, m_2), (n_3, m_3)$ satisfy NCA-configuration [5] for three 2-dimensional nodes, that is two of them lie on a line on the Cartesian plane and the third one lies on another line and does not lie on the intersection of these two lines. We take $(1, 1)$ and $(2, 1)$ lying on $y = 1$ and $(1, 2)$ lying on $y = 2$.

(4) Now we generate three pairs of input lists for append with lengths $(1, 1)$, $(2, 1)$ and $(1, 2)$ respectively. Since append is shapely it does not matter what we put as elements in these lists. For instance, it may be arbitrary integer numbers. So, we generate three input pairs:
- [1], [2] with the lengths $(1, 1)$ resp.,
- [1, 2], [3] with the lengths $(2, 1)$ resp.,
- [1], [2, 3] with the lengths $(1, 2)$ resp.

(5) Now we run append on these data:

input lengths	input list 1	input list 2	output of append	length of the output
$(1, 1)$	[1]	[2]	[1, 2]	2
$(2, 1)$	[1, 2]	[3]	[1, 2, 3]	3
$(1, 2)$	[1]	[2, 3]	[1, 2, 3]	3

(6) Using the table above, we generate a system of linear equations w.r.t. the coefficients a_{10}, a_{01}, a_{00}, corresponding to scheme (1):

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

(7) Solve the system above. The solution is $a_{10} = a_{01} = 1$ and $a_{00} = 0$. Therefore $p^?(n, m) = n + m$.

(8) Check the typing append : $\mathsf{L}_n(\alpha) \times \mathsf{L}_m(\alpha) \to \mathsf{L}_{n+m}(\alpha)$. As we have seen in Section 1, this typing is correct.

Exercise 5. Infer the annotation for append on a paper, assuming $d = 2$.

Exercise 6.

(1) Code conspack in demo and infer the annotation for it in demo.

(2) Infer the annotation for conspack on a paper, assuming the degree of the size dependency $d = 1$. If you obtain the typing conspack : $\texttt{Int} \times \mathsf{L}_n(\texttt{Int}) \to \mathsf{L}_{n+1}(\texttt{Int})$, then you do need to check it, since it has been checked. If you obtain another typing, check it.

Exercise 7.

(1) Experiment in the demo with usqdiff.

(2) Infer the annotation for sqdiff assuming the degree of the size dependency $d = 2$. If you obtain the typing sqdiff : $\mathsf{L}_n(\alpha) \times \mathsf{L}_m(\alpha) \to \mathsf{L}_{(n-m)^2}(\alpha)$, then you do need to check it, since it has been checked. If you obtain another typing, check it.

(3) Try to infer on a paper the the annotation for sqdiff assuming the degree of the size dependency $d = 1$. Type check it.

Exercise 8.

(1) Infer on a paper the typing for $\mathsf{scalar\_prod}$, based on one size variable, that is the input for the inference procedure is $\mathsf{scalar\_prod} : \mathsf{L}_n(\mathtt{Int}) \times \mathsf{L}_n(\mathtt{Int}) \to \mathsf{L}_{p^?(n)}(\mathtt{Int})$. You may start with any degree from: $d = 1, 2, \ldots$.

(2) (*) Code $\mathsf{scalar\_prod}$ in demo. Infer its type. Why have you inferred the type, $\mathsf{scalar\_prod} : \mathsf{L}_n(\mathtt{Int}) \times \mathsf{L}_m(\mathtt{Int}) \to \mathsf{L}_1(\mathtt{Int})$ but not the better one above? How should you implement the inference, so that only the type $\mathsf{scalar\_prod} : \mathsf{L}_n(\mathtt{Int}) \times \mathsf{L}_n(\mathtt{Int}) \to \mathsf{L}_1(\mathtt{Int})$ is inferred?

3. Beyond Shapely Programs

In the previous sections we have considered shapely function definitions that is function definitions for which the size(s) of an output is exactly a polynomial function of the sizes of the corresponding inputs. Our method was initially designed only for such programs. It was clear that this initial setting was very restrictive, since programs with *lower and upper polynomial bounds* on size dependencies were out of consideration.

In our further papers [2] and [3] (or [4] for more detail) we adopt the inference method for function definitions with polynomial lower and upper bounds. In this section we give two examples that informally explain how this extension works. Those who would like to know it in more detail are referred to the mentioned papers. However, we believe, that the examples and common sense are enough to do simple exercises on inference and checking of lower and upper bounds.

3.1. How to infer and check lower and upper polynomial bounds for insert.

It is a simple routine to extend our initial language with the boolean type $\mathtt{Bool}$. Consider a polymorphic program insert, that given an element $z : \alpha$, a predicate $g : \alpha \times \alpha \to \mathtt{Bool}$, and a list $l : \mathsf{L}_n(\alpha)$, inserts the element into the list if and only if there is no $z' \in l$, such that $g(z, z')$ holds:

$$
\begin{array}{l}
\mathsf{insert}(g, z, l) = \\
\mathsf{match}\ l\ \mathsf{with}\ \mid \mathsf{nil} \Rightarrow \mathsf{cons}(z, \mathsf{nil}) \\
\qquad\qquad\qquad \mid \mathsf{cons}(hd, tl) \Rightarrow \mathsf{if}\ g(z, hd)\ \mathsf{then}\ l\ \mathsf{else\ let}\ l' = \mathsf{insert}(z, tl) \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathsf{in\ cons}(hd, l')
\end{array}
$$

For instance, with g being equality of two integers, on $2,\ [1, 2, 3]$ it returns $[1, 2, 3]$ and on $2,\ [3, 4, 5]$ it returns $[3, 4, 5, 2]$.

(1) At the beginning we note that in many cases (including shapely programs) while studying size dependencies it is convenient to reduce an original program under consideration to its size abstraction, that is to collection of (recursive) rewriting rules for its size function.

Consider, how it is done for insert. Let $f_{\mathsf{insert}}(n)$ be the size of an output if the size of the input is n. Look at the code. The nil-branch gives that $f_{\mathsf{insert}}(0) \to 1$. In the cons-branch there are two possibilities for computing $f_{\mathsf{insert}}(n)$:

- the program returns the input list l, and then $f_{\mathsf{insert}}(n) \to n$,

- the program recursively calls insert on the tail of l, with a size function $f_{\mathsf{insert}}(n-1)$ and then returns a cons-cell on this recursive call, that is it returns a list of length $1 + f_{\mathsf{insert}}(n-1)$.

To sum up, we obtain the following rewriting system for the size dependency $f_{\mathsf{insert}}(n)$;

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

where $\mid$ separates two possible branches for computing $f_{\mathsf{insert}}(n)$.

In resource analysis one often uses such abstractions of program to *resource* functions or *resource recurrences*. For instance, we advice an interested reader to look at the work of German Puebla's group [1].

(2) Now we formulate our intention more concretely: using the fitting-polynomial methodology from the previous section, we can compute the lower $f_{\mathsf{insert}\ \min}(n) = n$ and the upper $f_{\mathsf{insert}\ \max}(n) = n+1$ bounds for $f_{\mathsf{insert}}(n)$. Formally, the size function f_{insert} is a *multivalued* size function. It has the type $\mathcal{R} \to 2^{\mathcal{R}}$, where $\mathcal{R}$ is a chosen numerical ring, like integers, rationals or reals. (Later, using the lower and the upper bounds, we will obtain $f_{\mathsf{insert}}(n) \subseteq \{f_{\mathsf{insert}\,\min}(n)+i\}_{0 \leq i \leq f_{\mathsf{insert}\,\max}(n) - f_{\mathsf{insert}\,\min}(n)} = \{n+i\}_{0 \leq i \leq 1}$.)

(3) Again, assume that the bounds depend on the size variable n, and, for the sake of convenience, let the degree $d = 1$ for both. A polynomial of one variable of the degree 1 (linear) is given by two coefficients. Thus $f_{\mathsf{insert}\ \min}(n) = a_{\min\ 1} n + a_{\min\ 0}$ and $f_{\mathsf{insert}\ \max}(n) = a_{\max\ 1} n + a_{\max\ 0}$ are defined by two nodes (1-dimensional points).

(4) Let these nodes are $n = 1, 2$.

(5) Differently to the initial version of our method we do not need to generate test data, if we have a rewriting system for a size function. We compute the values of a size function directly on concrete sizes. Thus, in our example we compute $f_{\mathsf{insert}}(1)$ and $f_{\mathsf{insert}}(2)$:

n	$f_{\mathsf{insert}}(n)$
1	$f_{\mathsf{insert}}(1) \to 1 \mid 1 + f_{\mathsf{insert}}(0) = \{1, 1 + f_{\mathsf{insert}}(0)\} = \{1, 1+1\} = \{1, 2\}$
2	$f_{\mathsf{insert}}(2) \to 2 \mid 1 + f_{\mathsf{insert}}(1) = \{2, 1 + f_{\mathsf{insert}}(1)\} = \{2, 1+1, 1+2\} = \{2, 3\}$

(6) Now, we see that

$$\begin{array}{ll} f_{\mathsf{insert}\,\min}(1) = 1 & f_{\mathsf{insert}\,\min}(2) = 2 \\ f_{\mathsf{insert}\,\max}(1) = 2 & f_{\mathsf{insert}\,\max}(2) = 3 \end{array}$$

So we have two systems of linear equations, for the lower and upper bounds respectively:

$$\left.\begin{array}{ll} a_{\min\ 1} + a_{\min\ 0} & = f_{\mathsf{insert}\,\min}(1) = 1 \\ 2a_{\min\ 1} + a_{\min\ 0} & = f_{\mathsf{insert}\,\min}(2) = 2 \end{array}\right\} \qquad \left.\begin{array}{ll} a_{\max\ 1} + a_{\max\ 0} & = f_{\mathsf{insert}\,\max}(1) = 2 \\ 2a_{\max\ 1} + a_{\max\ 0} & = f_{\mathsf{insert}\,\max}(2) = 3 \end{array}\right\}$$

(7) Solving these systems gives $a_{\min\ 1} = 1$, $a_{\min\ 0} = 0$ and $a_{\max\ 1} = 1$, $a_{\max\ 0} = 1$. That is, $f_{\mathsf{insert}\,\min}(n) = n$ and $f_{\mathsf{insert}\,\max}(n) = n + 1$. Thus, for $f_{\mathsf{insert}}(n)$ we have

$$f_{\mathsf{insert}}(n) \subseteq \{f_{\mathsf{insert}\,\min}(n) + i\}_{0 \leq i \leq f_{\mathsf{insert}\,\max}(n) - f_{\mathsf{insert}\,\min}(n)} = \{n+i\}_{0 \leq i \leq 1}$$

(8) Now we have to check, if indeed, $\mathsf{insert} : (\alpha\times\alpha \to \texttt{Bool})\times\alpha\times\mathsf{L}_n(\alpha) \to \mathsf{L}_{\{n+i\}_{0\le i\le 1}}(\alpha)$. The checking extends the checking procedure for shapely functions. Here, the output annotation should contain *all* the values of the size function in any branch of the computations (but not necessary be equal, as for shapely programs):
- the nil-branch $n = 0 \vdash f_{\mathsf{insert}}(n) \to 1$ gives rise to the following inclusion: $n = 0 \vdash \{n+i\}_{0\le i\le 1} \supseteq \{1\}$,
- for the true-branch in the cons-branch we have $n \ge 1 \vdash \{n+i\}_{0\le i\le 1} \supseteq \{n\}$,
- for the false-branch in the cons-branch we have $n \ge 1 \vdash \{n+i\}_{0\le i\le 1} \supseteq \{1\} + \{(n-1)+i'\}_{0\le i'\le 1}$

where $+$ is lifted to sets and defined as pairwise addition of the sets' elements.

(9) By unfolding the definition of a set inclusion, the inclusions above are trivially turned into the first-order predicates:
- $n = 0 \Rightarrow \exists\ i.\ 0 \le i \le 1 \wedge n + i = 1$,
- $n \ge 1 \Rightarrow \exists\ i.\ 0 \le i \le 1 \wedge n + i = n$,
- $\forall\ n\ i'.\ 0 \le i' \le 1 \wedge n \ge 1 \Rightarrow \exists\ i.\ 0 \le i \le 1 \wedge n + i = 1 + (n-1) + i'$

In this example it is easy to check that i may be instantiated as $i = 1$, $i = 0$ and $i = i'$ for each of the branches respectively. In general one have to instantiate the existential quantifiers in the first-order arithmetics. This is, in general, undecidable in integers (but still, decidable for linear size functions). It is decidable in reals, however real arithmetics has some disadvantages which we do not discuss here.

Exercise 9. Infer (and check), on a paper, polynomial lower and upper bounds for the function $\mathsf{filter} : (\alpha \to \texttt{Bool}) \times \mathsf{L}_n(\alpha) \to \mathsf{L}_{f_{\mathsf{filter}(n)}}(\alpha)$:

$$\begin{array}{l} \mathsf{filter}(g, l) = \\ \mathsf{match}\ l\ \mathsf{with}\ \mid \mathsf{nil} \Rightarrow \mathsf{nil} \\ \qquad\qquad\quad\ \mid \mathsf{cons}(hd, tl) \Rightarrow \mathsf{let}\ l' = \mathsf{filter}(g, tl) \\ \qquad\qquad\qquad\qquad\qquad\qquad \mathsf{in\ if}\ g(hd)\ \mathsf{then}\ \mathsf{cons}(hd, l')\ \mathsf{else}\ l' \end{array}$$

Given a unary predicate g and a list l, it returns the list of elements of l, that satisfy g. For instance if g is `even` then on $[1, 2, 3]$ it returns $[2]$.

3.2. **How to deal with programs over nested lists.** In general, in structures of the type $\mathsf{L}(\mathsf{L}(\alpha))$ the internal lists may be of different length, like, for instance, in $[\,[1, 2],\ [3, 4, 5],\ [\,]\,]$. Formally this fact may be expressed by introduction of length functions $\lambda\ k.M(k)$ that express lengths of internal lists, where $M(0)$ is the length of the head list, $M(1)$ is the length of the element following the head, etc. For instance, in the example above $M(0) = 2$, $M(1) = 3$, $M(2) = 0$ and $M(k)$ is arbitrary for $k \ge 3$.

It looks rather natural and simple, however it leads to higher-order size functions, since size variables may represent functions. Consider, for instance, the program $\mathsf{conc} : \mathsf{L}_n(\mathsf{L}_M(\alpha)) \to \mathsf{L}_{f_{\mathsf{conc}}(n,M)}(\alpha)$, that given a list of lists returns the concatenation of its elements:

$$\begin{array}{l} \mathsf{conc}(l) = \\ \mathsf{match}\ l\ \mathsf{with}\ \mid \mathsf{nil} \Rightarrow \mathsf{nil} \\ \qquad\qquad\quad\ \mid \mathsf{cons}(hd, tl) \Rightarrow \mathsf{let}\ l' = \mathsf{conc}(tl) \\ \qquad\qquad\qquad\qquad\qquad\qquad \mathsf{in\ append}(hd, l') \end{array}$$

For instance, on our list $[\,[1, 2],\ [3, 4, 5],\ [\,]\,]$ it returns $[\,1, 2, 3, 4, 5\,]$. We start with generating the collection of the rewriting rules computing the size function $f_{\mathsf{conc}}(n, M)$.

(1) Sure, in this case the rewriting rules are higher-order. First, we note that conc is called recursively on the tail of the list argument. So, we need to express the length function of the tail, M', via the length function M of the whole list. It is not difficult to see that M' is just the *left-shift* of M: $M'(k) = M(k+1)$. Indeed, the 0-th element of the tail is the first element of the list, the 1-st element of the tail is the second element of the list, etc. For instance, for our list $[\,[1,2],\ [3,4,5],\ [\,]\,]$ we have $M'(0) = 3$, $M'(1) = 0$ and $M(k)$ is arbitrary for $k \geq 2$.
We will denote the left shift of M via M_{+1}, so $M' = M_{+1}$.

(2) Similarly to the example insert, we parse the body of conc and obtain the following rewriting system for its size function:

$$\begin{array}{ll} n = 0 & \vdash f_{\mathsf{conc}}(n, M) \to 0 \\ n \geq 1 & \vdash f_{\mathsf{conc}}(n, M) \to M(0) + f_{\mathsf{conc}}(n-1, M_{+1}) \end{array}$$

(3) As in the case of insert, the rewriting system is not our end result in size analysis, but is just a tool to compute *closed*, i.e. recursion free, forms of lower and upper bounds on the size function of conc. Again, we are interested in usual polynomial bounds, not higher-order ones.

(4) What to do with the higher-order parameter M? We introduce a fresh usual size variable for it, m, meaning that for all $k \geq 0$ we have $0 \leq M(k) \leq m$. So, we want to obtain a typing of the following form

$$\mathsf{conc} : \mathsf{L}_n(\mathsf{L}_{\{i\}_{0\leq i\leq m}}(\alpha)) \to \mathsf{L}_{\{f_{\mathsf{conc}\ \min}(n,m)+i\}_{0\leq i\leq f_{\mathsf{conc}\ \max}(n,m)-f_{\mathsf{conc}\ \min}(n,m)}}(\alpha)$$

(5) Again, we assume the degree of lower and upper bounds. For the sake of simplicity, let it be $d = 2$. A polynomial of degree two of two variables is defined by six coefficients, so we need to know the values of $f_{\mathsf{conc}\ \min}$ and $f_{\mathsf{conc}\max}$ in six 2-dimensional points, satisfying NCA-configuration. Then we will have to solve the linear systems for the coefficients of $f_{\mathsf{conc}\ \min}$ and $f_{\mathsf{conc}\ \max}$. For $f_{\mathsf{conc}\ \max}$ (which is nontrivial in this case), the system is:

$$\begin{array}{ll}
a_{\max\ 20}n_1^2 + a_{\max\ 11}n_1m_1 + a_{\max\ 02}m_1^2 + a_{\max\ 10}n_1 + a_{\max\ 01}m_1 + a_{\max\ 00} & = f_{\mathsf{conc}\ \max}(n_1, m_1) \\
a_{\max\ 20}n_2^2 + a_{\max\ 11}n_2m_2 + a_{\max\ 02}m_2^2 + a_{\max\ 10}n_2 + a_{\max\ 01}m_2 + a_{\max\ 00} & = f_{\mathsf{conc}\ \max}(n_2, m_2) \\
a_{\max\ 20}n_3^2 + a_{\max\ 11}n_3m_3 + a_{\max\ 02}m_3^2 + a_{\max\ 10}n_3 + a_{\max\ 01}m_3 + a_{\max\ 00} & = f_{\mathsf{conc}\ \max}(n_3, m_3) \\
a_{\max\ 20}n_4^2 + a_{\max\ 11}n_4m_4 + a_{\max\ 02}m_4^2 + a_{\max\ 10}n_4 + a_{\max\ 01}m_4 + a_{\max\ 00} & = f_{\mathsf{conc}\ \max}(n_4, m_4) \\
a_{\max\ 20}n_5^2 + a_{\max\ 11}n_5m_5 + a_{\max\ 02}m_5^2 + a_{\max\ 10}n_5 + a_{\max\ 01}m_5 + a_{\max\ 00} & = f_{\mathsf{conc}\ \max}(n_5, m_5) \\
a_{\max\ 20}n_6^2 + a_{\max\ 11}n_6m_6 + a_{\max\ 02}m_6^2 + a_{\max\ 10}n_6 + a_{\max\ 01}m_6 + a_{\max\ 00} & = f_{\mathsf{conc}\ \max}(n_6, m_6)
\end{array}$$

The system for $f_{\mathsf{conc}\ \min}$ is similar.

(6) We choose the following nodes:

$$\begin{array}{lll} (n_1, m_1) = (1, 1) & (n_2, m_2) = (2, 1) & (n_3, m_3) = (3, 1) \\ (n_4, m_4) = (1, 2) & (n_5, m_5) = (2, 2) & \\ (n_6, m_6) = (1, 3) & & \end{array}$$

Now we need to compute $f_{\mathsf{conc}\max}$ (resp. $f_{\mathsf{conc}\ \min}$) in these nodes.

(7) We transform the rewriting system for the higher-order function f_{conc} into a rewriting system for the function f'_{conc} over numerical sets (and numbers):

$$\begin{array}{ll} n = 0 & \vdash f'_{\mathsf{conc}}(n, \{i\}_{0\le i\le m}) \to \{0\} \\ n \ge 1 & \vdash f'_{\mathsf{conc}}(n, \{i\}_{0\le i\le m}) \to \{i\}_{0\le i\le m} + f'_{\mathsf{conc}}(n-1, \{i\}_{0\le i\le m}) \end{array}$$

where $+$ is a pairwise addition of sets' elements. Note, that the second argument in the recursive call is the same as the second argument of the function f'_{conc}: this is because the elements of the tail have the same length bounds as the elements of the list. It easy to see that $f'_{\mathsf{conc}}(n, m) \supseteq f_{\mathsf{conc}}(n, M)$ if $M(k) \le m$ for all k.

(8) Now we can compute all possible values of f'_{conc} in the given nodes using the new rewriting system:

$$\begin{array}{lll} f'_{\mathsf{conc}}(1, \{0,1\}) & \to \{0,1\} + f'_{\mathsf{conc}}(0, \{0,1\}) \to \{0,1\} + \{0\} & = \{0,1\} \\ f'_{\mathsf{conc}}(2, \{0,1\}) & \to \{0,1\} + f'_{\mathsf{conc}}(1, \{0,1\}) \to^* \{0,1\} + \{0,1\} & = \{0,1,2\} \\ f'_{\mathsf{conc}}(3, \{0,1\}) & \to \{0,1\} + f'_{\mathsf{conc}}(2, \{0,1\}) \to^* \{0,1\} + \{0,1,2\} & = \{0,1,2,3\} \\ f'_{\mathsf{conc}}(1, \{0,1,2\}) & \to \{0,1,2\} + f'_{\mathsf{conc}}(0, \{0,1,2\}) \to \{0,1,2\} + \{0\} & = \{0,1,2\} \\ f'_{\mathsf{conc}}(2, \{0,1,2\}) & \to \{0,1,2\} + f'_{\mathsf{conc}}(1, \{0,1,2\}) \to^* \{0,1,2\} + \{0,1,2\} & = \{0,1,2,3,4\} \\ f'_{\mathsf{conc}}(1, \{0,1,2,3\}) & \to \{0,1,2,3\} + f'_{\mathsf{conc}}(0, \{0,1,2,3\}) \to \{0,1,2,3\} + \{0\} & = \{0,1,2,3\} \end{array}$$

(9) Now, pick up the maximal elements of each set. They constitute the r.h.s of the linear system for the coefficients of $f_{\mathsf{conc}\,\max}$:

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

(10) Solving this system gives that $a_{\max\ 11} = 1$ and the rest of the coefficients are zero. Thus, $f_{\mathsf{conc}\ \max}(n, m) = nm$.

(11) Similarly, $f_{\mathsf{conc}\ \min}(n, m) = 0$.

(12) Further, the length of the output on an input of the type $\mathsf{L}_n(\mathsf{L}_M(\alpha))$ should be in the set $\{f_{\mathsf{conc}\ \min}(n,m) + i\}_{0 \le i \le f_{\mathsf{conc}\ \max}(n,m) - f_{\mathsf{conc}\ \min}(n,m)} = \{i\}_{0\le i\le nm}$.

(13) To check if the computed bounds $f_{\mathsf{conc}\ \max}(n, m)$ and $f_{\mathsf{conc}\ \min}(n, m) = 0$ are indeed correct, we need to check the following typing:

$$\mathsf{conc} : \mathsf{L}_n(\mathsf{L}_{\{i\}_{0\le i\le m}}(\alpha)) \to \mathsf{L}_{\{i\}_{0\le i\le nm}}(\alpha)$$

(14) Following the computation scheme for f'_{conc}, defined by its rewriting rules, we conclude that the following inclusions must hold

$$\begin{array}{l} n = 0 \vdash \{i\}_{0\le i\le nm} \supseteq \{0\} \\ n \ge 1 \vdash \{i\}_{0\le i\le nm} \supseteq \{i\}_{0\le i\le m} + \{i\}_{0\le i\le (n-1)m} \end{array}$$

(15) Unfolding the definition of set inclusions we obtain the following first-order entailments:

$$\begin{array}{l} \forall\ n\, m \ge 0.\, n = 0 \Rightarrow \exists\ i.0 \le i \le nm \wedge i = 0 \\ \forall\ n\, m\, i'\, i'' \ge 0. n \ge 1 \wedge i' \le m \wedge i'' \le (n-1)m \Rightarrow \exists i.\ 0 \le i \le nm \wedge i = i' + i'' \end{array}$$

These entailments hold (and easily provable even in integer arithmetics, since they are linear).

(16) Conclusion: we have derived and proven the correctness of the lower $f_{\mathsf{conc}\ \min}(n, m) = 0$ and the upper $f_{\mathsf{conc}\ \max}(n, m) = nm$ polynomial size bounds for conc, given the internal lists of an input do not contain more than m elements.

Exercise 10. (*) Infer and check, on a paper, lower and upper bounds for the size functions $f_{\mathsf{tails}\,1}(n)$ and $f_{\mathsf{tails}\,2}(n)$ of the program $\mathsf{tails} : \mathsf{L}_n(\alpha) \to \mathsf{L}_{f_{\mathsf{tails}\,1}(n)}(\mathsf{L}_{f_{\mathsf{tails}\,2}(n)}(\alpha))$:

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

This program, given a list, returns the list of its nonempty tails. For instance, on $[1, 2, 3]$ it returns $[\,[1, 2, 3],\ [2, 3],\ [3]\,]$

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] O. Shkaravska, M. van Eekelen, and A. Tamalet. Collected Size Semantics for Functional Programs. In S.-B. Scholz, editor, *Implementation and Application of Functional Languages: 20^{th} International Workshop, IFL 2008, Hertfordshire, UK, 2008. Revised Papers*, LNCS. Springer-Verlag, 2008. to appear.

[3] O. Shkaravska, M. van Eekelen, and A. Tamalet. Collected size semantics for functional programs over polymorphic nested lists. In Z. Horvath and V. Szok, editors, *Trends in Functional Programming 2009*. Etovos Lorand University of Budapest, 2009.

[4] O. Shkaravska, M. van Eekelen, and A. Tamalet. Collected size semantics for functional programs over polymorphic nested lists. Technical Report ICIS-R09003, Radboud University Nijmegen, July 2009.

[5] O. Shkaravska, M. C. J. D. van Eekelen, and R. van Kesteren. Polynomial size analysis of first-order shapely functions. *Logical Methods in Computer Science*, 5, issue 2, paper 10:1–35, 2009. Special Issue with Selected Papers from TLCA 2007.