Inferring static non-monotonically sized types through testing

Ron van Kesteren, Olha Shkaravska, Marko van Eekelen
{R.vanKesteren, O.Shkaravska, M.vanEekelen}@cs.ru.nl

Institute for Computing and Information Sciences
Radboud University Nijmegen

Abstract. We propose a size analysis algorithm that combines testing and type checking to automatically obtain static output-on-input size dependencies for first-order functions. Attention is restricted to functions for which the size of the result is strictly polynomial, not necessarily monotonic, in the sizes of the arguments.
To infer a size dependency, the algorithm generates hypotheses for increasing degrees of polynomials. For each degree, a polynomial is defined by a finite number of points. The function is evaluated with a large enough set of appropriate measurement data to get these points and determine the coefficients of the polynomial. The resulting hypothesis is then checked using an existing type checking procedure.
The algorithm is not tied to the current sized type checker. The sized type of a function will be inferred if it exists and if it is accepted by the sized type checker. For terminating functions, our sized type inference algorithm is complete with respect to type checking. Hence, using a more complete sized type checker yields a more complete sized type inference algorithm.

Keywords: Memory complexity analysis, type checking, testing, Lagrange interpolation

1 Introduction

Embedded systems or server applications often have limited resources available. Therefore, it can be important to know in advance how much time or memory a computation is going to take, for instance to determine how much memory should at least be put in a system to enable all desired operations. Economically, the developer does not want to include too much memory, but the costs of failure of the application will be much higher.

Such decisions can only reliably be based on formally verified upper bounds of the resource consumption. However, an advanced detailed analysis of these bounds requires knowledge of the sizes of the data structures used throughout the program [ESvK+07]. Trivially, the time it takes to iterate over a list depends on the size of that list. In this paper we focus on the task of automatically deriving the exact output-on-input size dependencies of functions.

Size dependencies can be represented in function *types*. We focus on shapely functions, where shapely means that the size relations are exactly polynomial (not necessarily monotonic). As an example, consider the function that computes the Cartesian product of two lists. It generates all pairs of elements, one taken from the first list, the other from the second.

```
pairs x []       = []
pairs x (y:ys)   = [x,y]:pairs x ys

cprod []     ys = []
cprod (x:xs) ys = pairs x ys ++ cprod xs ys
```

The size of a list is the number of nodes it consists of (its length). Given lists of size 3 and 2, the output is a list of size $3 * 2 = 6$ whose elements are pairs, i.e., lists of size 2.

```
cprod [1,2,3] [4,5] = [[1,4],[1,5],[2,4],[2,5],[3,4],[3,5]]
```

The *sized type* of the `cprod` function expresses the general relation between argument and result sizes. When the two input lists have size s_1 and s_2 respectively, the output is a list of lists, where the outer list has size $s_1 * s_2$ and the inner lists all have size 2.

$$\texttt{cprod} : \texttt{[Int]}^{s_1} \rightarrow \texttt{[Int]}^{s_2} \rightarrow \texttt{[[Int]}^{2}\texttt{]}^{s_1 * s_2}$$

In general, all lists at the input side, before the arrow, have an associated size variable. After the arrow, at the output side, all lists have an associated polynomial that determines the size of the output list. These polynomials are defined in terms of the input size variables. The current presentation is limited to a language over lists for reasons of simplicity; sized types are straightforwardly generalized to general data structures and other programming languages.

Recently, we have developed a sized type checking procedure to formally verify polynomially sized types (section 2) [SvKvE07]. Given a sized type, the procedure automatically checks if the function definition satisfies that type. Unfortunately, inferring such types is a lot more challenging than type checking and the type system approach does not straightforwardly extend (section 2.3). Therefore,

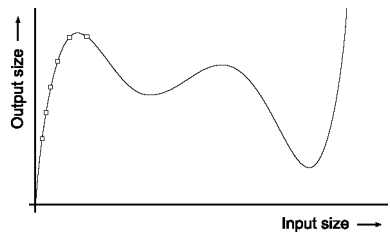

Fig. 1. A fifth degree polynomial is determined completely by any six of its points.

we have suggested an alternative method of inferring sized types [SvKvE07]. This paper develops this method into a practical type inference algorithm.

The method is based on the observation that it is relatively easy to generate hypotheses for a size dependency by testing. Because a polynomial of a given degree is determined by a finite number of values, its coefficients can be computed from the output sizes of run-time tests (figure 1). If the size expression is indeed a polynomial of that degree, it can be only *that* polynomial. This theory is used to create a practical algorithm that yields hypotheses for sized types (section 3).

Combining hypothesis generation and type checking yields an algorithm that can infer the sized type of a function (section 4). The algorithm generates hypotheses for an increasing degree. For each degree, hypotheses for all polynomial size expressions in the output type are determined. The resulting sized type is checked using the sized type checking procedure. Thus:

1. Infer the underlying type (without sizes) using standard type inference
2. Annotate the underlying type with size variables
3. Assume the degree of the polynomial
4. For every output size:
   - Determine which tests are needed
   - Do the required series of test runs
   - Compute the polynomial coefficients based on the test results
5. Annotate the type with the size expressions found
6. Check the annotated type
7. If checking fails, repeat from step 4 assuming a higher degree

In practice, an upper limit on the degree can be used as a stopping criterium. Note that the algorithm can also work with any other procedure that automatically checks polynomially sized types. Indeed, for terminating programs the algorithm is only guaranteed to find the sized type if one exists that is accepted by the type checker.

The main contribution of this paper is developing the method suggested [SvKvE07] into a practical sized type inference algorithm. Specifically, this means dealing with cases where the function definition only *partially defines* the output size polynomial: when the output type is a nested list and the output value is the empty list, there is no information on the sizes of the inner lists.

2 Sized type checking

Essentially, our approach to sized type inference for shapely functions is based on reducing inference to sized type checking. This section briefly describes the existing strict size-aware type system for a functional language and accompanying type checking procedure [SvKvE07] that we use in the inference algorithm. This also motivates our approach to type inference.

2.1 Sized Types

The zero-order types we consider are integers, *strictly* sized lists of integers, *strictly* sized lists of *strictly* sized lists, etc. For lists of lists the element lists have to be of the same size and in fact it would be more precise to speak about matrix-like structures. For instance, the type $\texttt{[[Int]}^3\texttt{]}^2$ is given to a list which two elements are both lists of exactly three integers, such as `[[2,5,3], [7,1,6]]`.

$$\mathit{Types}\ \tau ::= \texttt{Int} \mid \alpha \mid \texttt{[}\tau\texttt{]}^p \qquad \alpha \in \mathit{TypeVar}$$

The p in this definition denotes a size expression. Size expressions are polynomials in size variables.

$$\mathit{SizeExpr}\ p ::= \mathbb{N} \mid s \mid p + p \mid p - p \mid p * p \qquad s \in \mathit{SizeVar}$$

For instance, type $\texttt{[}\alpha\texttt{]}^4$ represents a list containing four elements of some type α and $\texttt{[Int]}^{(s_1 - s_2)^2}$ represents a list of integers of size $(s_1 - s_2)^2$ where s_1 and s_2 are size variables. Size expressions are subject to the standard associativity, commutativity and distributivity laws for addition and multiplication. Types with negative sizes have no meaning.

Because the current system does not support Currying, first-order types are functions from tuples of zero-order types to zero-order types.

$$\mathit{FTypes}\ \tau^f ::= \tau_1\ \ldots\ \tau_n \rightarrow \tau_{n+1}$$

For example, the type of `cprod`, $\texttt{[Int]}^{s_1} \rightarrow \texttt{[Int]}^{s_2} \rightarrow \texttt{[[Int]}^2\texttt{]}^{s_1 * s_2}$ is a first-order type. In well-formed first-order types, the argument types are annotated only by size variables and the result type is annotated by size expressions in these variables. Type and size variables occurring in the result type should also occur in at least one of the argument types. Thus, the type of `cprod` is a well-formed type, whereas $\texttt{[}\alpha\texttt{]}^{s_1 + s_2} \rightarrow \texttt{[}\alpha\texttt{]}^{2 * s_1}$ is not.

2.2 Typing system

Previously, we have developed a sound size-aware type system and a type checking procedure for a first-order functional language with call-by-value semantics [SvKvE07]. The language supports lists and integers and standard constructs for pattern matching, if-then-else branching, and let-binding.

The typing rules follow the intuition on how sizes are used and changed during function evaluation. The construction of a list results in a list that is one element longer than the tail. The `then` and `else` parts of the if-statement are required to yield the same size. The same goes for the `nil` and `cons` branch of pattern matching, but that rule also takes into account that the matched list is known to be empty in the `nil` branch: when matching a list of size s, if the `cons` branch has size $s * 4$, the `nil` branch can have size 0 because, there, $s = 0$ and thus $0 = s * 4$.

In the formal rules, a context Γ is a mapping from zero-order program variables to zero-order types, a signature Σ is a mapping from function names to

first-order types, and D is a set of Diophantine equations that keeps track of which lists are empty. A typing judgment is a relation of the form $D;\ \Gamma\ \vdash_\Sigma e : \tau$ which means that if the free program variables of the expression e have the types defined by Γ, and the functions called have the types defined by Σ, and the size constraints D are satisfied, then e will be evaluated to a value of type τ, if it terminates. For example:

$$\frac{D \vdash p = p' + 1}{D;\ \Gamma,\ hd : \tau,\ tl : [\tau]^{p'}\ \vdash_\Sigma \mathsf{cons}(hd, tl) : [\tau]^{p}}\ \textsc{Cons}$$

$$\frac{\Gamma(x) = \texttt{Int} \qquad D;\ \Gamma\ \vdash_\Sigma e_t : \tau \qquad D;\ \Gamma\ \vdash_\Sigma e_f : \tau}{D;\ \Gamma\ \vdash_\Sigma \mathsf{if}\ x\ \mathsf{then}\ e_t\ \mathsf{else}\ e_f : \tau}\ \textsc{If}$$

$$\frac{\begin{array}{c} p = 0,\ D;\ \Gamma,\ x : [\tau']^{p}\ \vdash_\Sigma e_{\mathsf{nil}} : \tau \\ hd, tl \notin dom(\Gamma) \qquad D;\ \Gamma, hd : \tau',\ x : [\tau']^{p},\ tl : [\tau']^{p-1}\ \vdash_\Sigma e_{\mathsf{cons}} : \tau \end{array}}{\begin{array}{l} D; \Gamma, x : [\tau']^{p}\ \vdash_\Sigma \mathsf{match}\ x\ \mathsf{with} \mid \mathsf{nil} \Rightarrow e_{\mathsf{nil}} \qquad\qquad : \tau \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \mid \mathsf{cons}(hd, tl) \Rightarrow e_{\mathsf{cons}} \end{array}}\ \textsc{Match}$$

Sized type checking eventually amounts to checking entailments of the form $D \vdash p = p'$, which means that $p = p'$ is derivable from D in the axiomatics of the ring of integers. Because p and p' are known polynomials of universally quantified size variables, comparing them is straightforward. For instance, for the `cprod` function we obtain $s_1 = 0 \vdash s_1 * s_2 = 0$ (in the `nil` branch) and $\vdash s_1 * s_2 = s_2 + (s_1 - 1) * s_2$ (in the `cons` branch). A syntactical condition that prohibits let-bindings before pattern matching was shown to be necessary and sufficient to make type checking decidable for this system [SvKvE07].

2.3 Motivation

Type inference in this type system is not straightforward. Applying the typing rules to types with unknown size expressions leads to sets of non-linear equations [SvKvE07] for which we know that there is no algorithm that solves them all. Of course, it is possible to write an algorithm that solves a subset of these cases, but then it is hard to determine to which subset of function definitions this corresponds and, consequently, if type inference is complete. It is also hard, and not desirable, to restrict the type system so that we can be sure that only solvable equations are generated (as Mycroft [Myc84] did for the Milner calculus [Mil78]). Both approaches most likely add unwanted restrictions, whereas we want our type inference algorithm to be as complete as possible.

The testing approach presented in this paper does not use the type system directly. Hypotheses for types are constructed based only on the observed behavior of the function. This avoids solving non-linear systems of equations. To validate the hypotheses we use the existing, decidable, type checking algorithm. However, in practice any type checker can be used. The algorithm ensures that, for terminating programs, type inference is complete with respect to the type checker that is used.

3 Generating size hypotheses

This section develops a procedure that uses run-time tests to automatically obtain a hypothesis for an output size polynomial, given its maximum degree. This hypothesis is correct if the output size is in fact a polynomial of the same or lower degree. In section 4, this is combined with the type checker from section 2 to obtain a sized type inference algorithm.

The essence of the problem is giving the conditions under which a set of data points has a unique polynomial interpolation and constructing an algorithm to find points satisfying these conditions. This is complicated by the fact that for nested lists the size function is only partially defined by the function definition (section 3.3).

3.1 Interpolating a polynomial

Looking at the sizes of the arguments and results of some tests of the `cprod` function gives the impression that the size of the outer list in the output is always the product of the sizes of the arguments. More specifically, if $p_1(s_1, s_2)$ is the size of the outer list given arguments of size s_1 and s_2, tests yielding $p_1(1,3) = 3$, $p_1(4,6) = 24$, and $p_1(3,5) = 15$ may be interpolated to $p_1(s_1, s_2) = s_1 * s_2$. Such a hypothesis can also be derived automatically by fitting a polynomial to the size data. We are looking for the polynomial that best approaches the data, i.e., the Lagrange interpolation. The Lagrange interpolation is unique under some conditions on the data, which are explored in polynomial interpolation theory [CL87,Lor92]. If the true size expression is polynomial and the degree of the unique Lagrange interpolation is high enough, the interpolating polynomial coincides with the true size expression.

We seek a condition under which the interpolation is unique. In the well-known univariate case this is simple. A polynomial $p(x)$ of degree m with coefficients $a_1, \ldots, a_{m+1}$ can be written as follows:

$$a_1 \; + \; a_2\, x \; + \; \ldots \; + \; a_{m+1}\, x^m \; = \; p(x)$$

The values of the polynomial function in any $m+1$ points determine a system of linear equations w.r.t. the polynomial coefficients. More specifically, given the set $\big(x_i, p(x_i)\big)$ of pairs of numbers, where $1 \le i \le m+1$, and coefficients $a_1, \ \ldots \ , a_{m+1}$, the set of equations can be represented in the following matrix form, where only the a_i are unknown:

$$\begin{pmatrix} 1 & x_1 & \cdots & x_1^{m-1} & x_1^m \\ 1 & x_2 & \cdots & x_2^{m-1} & x_2^m \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 1 & x_m & \cdots & x_m^{m-1} & x_m^m \\ 1 & x_{m+1} & \cdots & x_{m+1}^{m-1} & x_{m+1}^m \end{pmatrix} \begin{pmatrix} a_1 \\ a_2 \\ \vdots \\ a_m \\ a_{m+1} \end{pmatrix} = \begin{pmatrix} p(x_1) \\ p(x_2) \\ \vdots \\ p(x_m) \\ p(x_{m+1}) \end{pmatrix}$$

The determinant of the left matrix, contains the measurement points, is called the Vandermonde determinant. For pairwise different points $x_1, \ldots, \ x_{m+1}$ it is

non-zero. This means that, as long as the output size is measured for $m+1$ different input sizes, there exists a unique solution for the system of equations and, thus, a unique interpolating polynomial.

The conditions under which there exists a unique polynomial that interpolates *multivariate* data are not so trivial. A polynomial of degree m and dimension n (the number of variables) has $N_m^n = \binom{m+n}{n}$ coefficients. The condition under which a set of data uniquely determines a polynomial interpolation is stated as a condition on a set of *nodes* $W = \{\bar{w}_i : i = 1, \ldots, N_m^n\}$, the input sizes for which a measurement is done, such that for every set of associated measurement data $\{f_i : i = 1, \ldots, N_m^n\}$, there is a unique polynomial $p(\bar{w}) = \Sigma_{0 \le |j| \le m} a_j \bar{w}^j$ with total degree m which interpolates the given data at the nodes [CL87]. That is, $p(\bar{w}_i) = f_i$, where $1 \le i \le N_m^n$. Here $\bar{w}^j = w_1^{j_1} \ldots w_n^{j_n}$, $|j| = j_1 + \ldots + j_n$ is the usual multivariate notation. In the next subsections, node configurations that satisfy this condition are defined, starting with bivariate polynomials and ending with the general case.

3.2 Measuring bivariate polynomials

For a two-dimensional polynomial of degree m, the condition on the nodes that guarantees a unique polynomial interpolation is as follows. In the input space, there are $m+1$ lines, each containing $m+1, \ldots, 1$ of the nodes, respectively, and the nodes do not lie on the intersections of the lines. Such a configuration is depicted for parallel lines in figure 2a. This corresponds to the **NCA** configuration studied, for instance, by Chui [CL87].

Definition 1 (Two-dimensional node configuration). *There exist lines in the input space, $\gamma_1, \ldots, \gamma_{m+1}$, such that $m+1$ nodes of W lie on γ_{m+1}, m nodes of W lie on $\gamma_m \setminus \gamma_{m+1}$, ..., and 1 node of W lies on $\gamma_1 \setminus (\gamma_2 \cup \ldots \cup \gamma_{m+1})$.*

Assuming the function terminates on all inputs, such points can be found algorithmically, at least for outermost lists, using a triangle of points on parallel lines (figure 2b).

An example of the two dimensional case is the `cprod` function from the introduction. Standard type inference and annotating gives the following type:

$$\texttt{cprod} : [\alpha]^{s_1}\ [\alpha]^{s_2} \to [[\alpha]^{p_2(s_1,s_2)}]^{p_1(s_1,s_2)}$$

We derive that $p_1(s_1, s_2) = s_1 * s_2$ assuming p_1 is a quadratic polynomial:

$$p_1(s_1, s_2) = a_{0,0} + a_{0,1}s_1 + a_{1,0}s_2 + a_{1,1}s_1s_2 + a_{0,2}s_1^2 + a_{2,0}s_2^2$$

Running the function at the six nodes from figure 2b gives the following results:

s_1	s_2	`x`	`y`	`cprod x y`	$p_1(s_1, s_2)$	$p_2(s_1, s_2)$
0	0	`[]`	`[]`	`[]`	0	–
1	0	`[0]`	`[]`	`[]`	0	–
0	1	`[]`	`[0]`	`[]`	0	–
1	1	`[0]`	`[1]`	`[[0, 1]]`	1	2
2	1	`[0,1]`	`[2]`	`[[0,2],[1,2]]`	2	2
1	2	`[0]`	`[1,2]`	`[[0,1],[0,2]]`	2	2

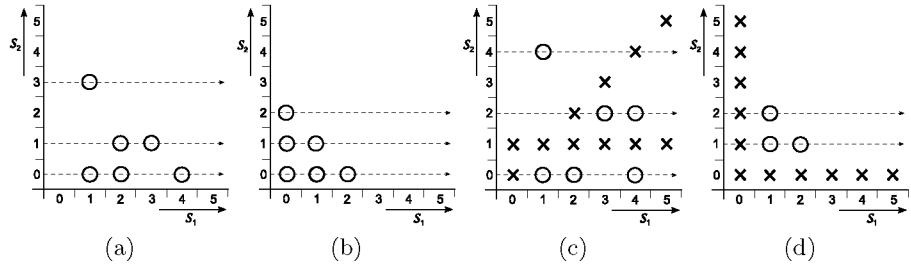

Fig. 2. (a) A node configuration that has a unique two-dimensional polynomial interpolation (b) A more systematic node configuration that has a unique two-dimensional polynomial interpolation (c) Undefined points complicate finding a node configuration (d) Undefined measurements for the pairs in the output of `cprod`

This defines the following linear system of equations for the coefficients of p_1:

$$\begin{aligned} a_{0,0} &= 0 \\ a_{0,0} + a_{0,1} + a_{0,2} &= 0 \\ a_{0,0} + a_{1,0} + a_{2,0} &= 0 \\ a_{0,0} + a_{0,1} + a_{1,0} + a_{0,2} + a_{1,1} + a_{2,0} &= 1 \\ a_{0,0} + 2a_{0,1} + a_{1,0} + 4a_{0,2} + 2a_{1,1} + a_{2,0} &= 2 \\ a_{0,0} + a_{0,1} + 2a_{1,0} + a_{0,2} + 2a_{1,1} + 4a_{2,0} &= 2 \end{aligned}$$

The unique solution is $a_{1,1} = 1$ with the rest of the coefficients zero. Thus, we obtain the correct $p_1(s_1, s_2)$ equal to $s_1 * s_2$.

This procedure is relatively straightforward. However, there is a problem in repeating it for p_2. There are cases in which nodes have no corresponding output size (the dashes in the table). `cprod` only *partially* defines p_2, because the size of the inner lists can only be determined when there is at least one such a list. Thus, the outer list may not be empty. As can be seen in figure 2d, for `cprod` this is always the case when one of the two input lists is empty. In the next section, we show that, despite this, it is still possible to always find enough measurements and give an upper bound on the number of nodes that have to be searched.

3.3 Handling partial definedness

From the example in the previous section, it is clear that care should be taken when searching for hypotheses for output types with nested lists. In general, for $[\ldots[\alpha]^{p_k}\ldots]^{p_1}$ we will not find a value for p_j at a node if one of the outer polynomials, p_1 to p_{j-1}, is zero at that node. Thus, the nodes where p_1 to p_{j-1} are zero should be excluded from the testing process. Here, we show that, despite this, it is always possible to find enough nodes so that it becomes possible to construct an algorithm to find them.

First note that we do not consider nested lists with the size of the outer list a constant zero, like $[[\tau]^q]^0$, because it is not a principal type. Also, remember that we are searching parallel lines $p(x, i)$ for the node configuration. Then, for

any non-zero polynomial there is a finite number of lines $y = i$, which we will call *root lines*, where $p(x,\, i) = 0$ (see lemma 1). There are infinitely many other lines.

Lemma 1. *A polynomial $p(x,\, y)$ of degree m that is not constant 0 has at most m root lines $y = i$, such that $p(x,\, i) = 0$.*

Proof. Suppose there are more than m root lines. Then, it is easy to pick $1, \ldots, m+1$ nodes on $m+1$ root lines. With these nodes, at which $p(x, y) = 0$, the system of linear equations for the coefficients of p will have the zero-solution, that is, all the coefficients of p will be zeros. This contradicts the assumption that p is not constant 0.

Because of this property, diagonal search can always find as many nodes $(x,\, y)$ as desired, such that $p(x, y) \neq 0$ (see figure 2c, where roots are marked with crosses). In fact, without requiring diagonal search, we can give a limit on the number of parallel lines $y = i$ and nodes on them that have to be searched at most. Essentially, we just try to find the triangle shape (as in figure 2b) while skipping all crosses. First, we show that for a nested list type $\texttt{[[}\alpha\texttt{]}^{q}\texttt{]}^{p}$ with bivariate polynomial sizes q and p, only the nodes in $[0, \ldots, m_1 + m_2] \times [0, \ldots, m_1 + m_2]$ have to be searched to determine q, where m_1 and m_2 are the degrees of p and q respectively.

Say one needs to find coefficients of an output type $\texttt{[[}\alpha\texttt{]}^{q}\texttt{]}^{p}$, and let $n = 2$ be the amount of variables, m_1 be the degree of $p(x, y)$ and m_2 be the degree of $q(x, y)$. One looks for test points for q that determine a unique polynomial interpolation at places where $p(x, y) \neq 0$. We restrict ourselves to lines γ parallel to the x-axis and we look for $(m_2 + 1)(m_2 + 2)/2$ data points satisfying the condition from definition 1.

Lemma 2. *When looking for test points for a polynomial $q(x, y)$ that determine a unique polynomial interpolation at places where another polynomial $p(x, y) \neq 0$, it is sufficient to search the lines $y = 0, \ldots, y = m_1 + m_2$ in the square $[0, \ldots, m_1 + m_2] \times [0, \ldots, m_1 + m_2]$.*

Proof. For the configuration it is sufficient to have $m_2 + 1$ lines with at least $m_2 + 1$ points where $p(x, y) \neq 0$. Due to lemma 1 there are at most m_1 lines $y = i$ such that $p(x,\, i) = 0$, so at least $m_2 + 1$ are not root lines for p. The polynomial $p(x,\, j)$, with $y = j$ not a root line, has at most degree m_1, thus $y = j$ contains at most m_1 nodes $(x,\, j)$, such that $p(x,\, j) = 0$. Otherwise, it would have been constant zero, and thus a root line. Hence, this leaves at least $m_2 + 1$ points on these lines for which p is not zero.

This straightforwardly generalizes to all nested types with polynomials in two variables, say $\texttt{[}\ldots\texttt{[}\alpha\texttt{]}^{p_k}\ldots\texttt{]}^{p_1}$. If we want to derive the coefficients of p_i, searching the square of input values $[0, \ldots, \Sigma_{i=1}^{k} m_i] \times [0, \ldots, \Sigma_{i=1}^{k} m_i]$ suffices, where m_i is the degree of p_i. Each p_i has at most m_i root lines, so there are at most $\Sigma_{j=1}^{i-1} m_j$ root lines. Also, each of the p_i can have at most m_i zeros on a non

root line. Hence, when the length is $\Sigma_{j=1}^{k} m_j + 1$ there are always $m_i + 1$ values known.

For `cprod` there are two size expressions to derive, p_1 for the outer list and p_2 for the inner lists. Deriving that $p_1(s_1, s_2) = s_1 * s_2$ is no problem. Because p_1 has roots for $s_1 = 0$ and for $s_2 = 0$, these nodes should be skipped when measuring p_2 (see figure 2d).

3.4 Generalizing to n-dimensional polynomials

The generalization of the condition on nodes for a unique polynomial interpolation to polynomials in n variables, is a straightforward inductive generalization of the two-dimensional case. In a hyperspace there have to be hyperplanes, on each of which nodes lie that satisfy the condition in the $n-1$ dimensional case. A hyperplane K_j^n may be viewed as a set in which test points for a polynomial of $n-1$ variable of the degree j lie. There must be $N_j^{n-1} = N_j^n - N_{j-1}^n$ such points. The condition on the nodes is defined by:

Definition 2 (n-dimensional node configuration). *The **NCA** configuration for n variables (n-dimensional space) is defined inductively on n [CL87]. Let $\{x_1, \ldots, x_{N_m^n}\}$ be a set of distinct points in $\mathcal{R}^n$ such that there exist $m+1$ hyperplanes K_j^n, $0 \leq j \leq m$ with*

$$x_{N_{m-1}^n+1}, \ldots, x_{N_m^n} \in K_m^n$$
$$x_{N_{j-1}^n+1}, \ldots, x_{N_j^n} \in K_j^n \setminus \{K_{j+1}^n \cup \ldots \cup K_m^n\}, \text{for } 0 \leq j \leq m-1$$

*and each of set of points $x_{N_{j-1}^n+1}, \ldots, x_{N_j^n}$, $0 \leq j \leq n$, considered as points in $\mathcal{R}^{n-1}$ satisfies **NCA** in $\mathcal{R}^{n-1}$.*

Thus, similarly to lines in a square in the two dimensional case, parallel hyperplanes in a hyperspace have to be searched. Using a reasoning similar to the two-dimensional case one can show that it is always sufficient to search a hypercube with sides $[0, \ldots, \Sigma_{i=1}^{k} m_i]$. The proof is also straightforwardly generalized.

4 Automatically inferring sized types

The type checking procedure from section 2 and the size hypothesis generation from section 3 are combined into a type inference algorithm by generating and checking hypothesis for an increasing degree. The algorithm is semi-decidable: it only terminates when the function is well-typable in the type system of the type checker used.

4.1 The algorithm

For any shapely program, the underlying type (the type without size annotations) can be derived by a standard type inference algorithm [Mil78]. After

Function: TryIncreasingDegrees
Input: the function definition
Output: the sized type of that function

```
TryIncreasingDegrees(m, f) =
  let type  = InferUnderlyingType(f)
      atype = AnnotateWithSizeVariables(type)
      vs    = GetOutputSizeVariables(atype)
      stype = GetSizedType(m, f, atype, vs, [ ])
  in if (CheckSizedType(stype, f)) then stype
     else TryIncreasingDegrees(m+1, f)
```

Function: GetSizedType
Input: a degree, the function definition with its annotated type, the variables to derive and the polynomials already derived
Output: the sized type of that function if the degree is high enough

```
GetSizedType(m, f, atype, [ ], ps) =
  AnnotateWithSizeExpressions(atype, ps)
GetSizedType(m, f, atype, v:vs, ps) =
  let nodes   = GetNodeConf(m, atype, ps)
      results = RunTests(f, nodes)
      p       = DerivePolynomial(m, v, atype, results)
  in GetSizedType(m, f, atype, vs, p:ps)
```

Fig. 3. The weak type inference algorithm in pseudo-code

straightforwardly annotating input sizes with size variables and output sizes with size expression variables, we have for example

$$\texttt{cprod} : [\alpha]^{s_1} \rightarrow [\alpha]^{s_2} \rightarrow [[\alpha]^{p_2(s_1,s_2)}]^{p_1(s_1,s_2)}$$

To derive the size expressions on the right hand side we use the following procedure. First, the maximum degree of the occurring size expressions is assumed, starting with zero. Then, a hypothesis is generated for each size expression. This is done from the outside in, because of the problems with partially definedness noted in section 3.3. After hypotheses have been obtained for all size expressions they are added to the type and this hypothesis type is checked using the type checking algorithm. If it is accepted, the type is returned. If not, the procedure is repeated for a higher degree.

Figure 3 shows the algorithm in pseudo-code. Note that if the assumed degree is lower than the true degree, the derived polynomials may be wrong. In that case, also the places where the size function is undefined cannot be determined correctly. It might happen that the node configuration includes points where the size expression is undefined so the test results do not provide enough information to uniquely infer the polynomial. In that case, by convention, the zero polynomial is returned.

If a type is rejected, this can mean two things. First, the assumed degree was too low and one of the size expressions has a higher degree. That is why the

function	m	nr. of tests	type suggested	type checker
`cprod`	0	1 (1)	$[\alpha]^{s_1} \to [\alpha]^{s_2} \to [[\alpha]^2]^0$	reject
(n = 2, k = 2)	1	8 (9)	$[\alpha]^{s_1} \to [\alpha]^{s_2} \to [[\alpha]^2]^{s_1+s_2-1}$	reject
	2	14 (25)	$[\alpha]^{s_1} \to [\alpha]^{s_2} \to [[\alpha]^2]^{s_1*s_2}$	accept
`append`	0	1 (1)	$[\alpha]^{s_1} \to [\alpha]^{s_2} \to [\alpha]^0$	reject
(n = 2, k = 1)	1	3 (4)	$[\alpha]^{s_1} \to [\alpha]^{s_2} \to [\alpha]^{s_1+s_2}$	accept
`competition`	0	1 (1)	$[\alpha]^{s_1} \to [[\alpha]^0]^0$	reject
(n = 1, k = 2)	1	3 (3)	$[\alpha]^{s_1} \to [[\alpha]^0]^{s_1}$	reject
	2	5 (5)	$[\alpha]^{s_1} \to [[\alpha]^2]^{s_1^2-s_1}$	accept
`sqdiff`	0	1 (1)	$[\alpha]^{s_1} \to [\alpha]^{s_2} \to [[\alpha]^0]^0$	reject
(n = 2, k = 2)	0	3 (9)	$[\alpha]^{s_1} \to [\alpha]^{s_2} \to [[\alpha]^0]^0$	reject
	0	8 (25)	$[\alpha]^{s_1} \to [\alpha]^{s_2} \to [[\alpha]^2]^{(s_1-s_2)^2}$	accept

Table 1. Type construction for four functions (n is the number of input variables, k the number of output polynomials). For each iteration of the algorithm, the degree (m) and the number of tests required (and the theoretical maximum $(1 + km)^n$) to get a hypothesis is given assuming the space was searched using diagonal search.

procedure continues for a higher degree. Another possibility is that one if the size expressions is not a polynomial (the function definition is not shapely) or that the type cannot be checked due to incompleteness. In that case the algorithm will not terminate. Fortunately, in practice a suitable stopping criterium may be known. If the function is well-typable, the procedure will eventually find the correct sized type and terminate.

4.2 Examples

The algorithm is illustrated by four functions: `cprod` (Cartesian product), `append` (standard list concatenation), `competition` (generates a competition in which every team plays a home and away match against every other team), and `sqdiff` (illustration of non-monotonicity).

```
competition  xs      = randomize_order (competition' xs [])
competition' []  ys  = []
competition' x:xs ys = pairs x xs ++ competition' xs x:ys

sqdiff []   ys       = cprod ys ys
sqdiff x    []       = cprod xs xs
sqdiff x:xs y:ys     = sqdiff xs ys
```

For each function, table 1 gives the hypotheses generated for each iteration of the algorithm until the correct type has been found. As can be seen, in practice the number of tests is much lower than the theoretical maximum.

5 Discussion and Future Work

The algorithm currently has three apparent limitations. First, the algorithm has two possible sources of non-termination. Second, it only works for exact sizes and not for upper bounds. Third, it is developed for a first-order functional language with lists as the only supported data structures. Here, these issues are discussed and improvements are suggested.

5.1 Sources of Nontermination

Because the algorithm uses run-time tests, it does not terminate when one of these tests does not terminate. In practice, however, this is not an important problem, because the analysis will typically be run on a stable product where non-termination should be rare. Just in case, a termination analysis can be done first or the algorithm may be adapted to start looking for replacement tests if evaluation of a test takes too long and non-termination is suspected. In general, this problem is very related to test-case construction, which is an active field of research.

The second source of nontermination is the iteration over increasing degrees of polynomials. If none of the generated types is accepted by the checker, either because the function definition is not shapely or due to incompleteness, the algorithm in principle does not stop. In practice, often an upper bound can be put on the degree because only size expressions of low degree are desired.

5.2 Shapely programs

The current hypothesis generation algorithm relies on the limitation to shapely programs; output sizes need to be exactly polynomial in the input size. In practice many programs are not shapely, but still have a polynomial upper bound. For instance, inserting an element in a set only increases the set by one if the element was not in it yet. Its upper bound would be:

$$\texttt{insert} : [\alpha]^s \to \alpha \to [\alpha]^{s+1}$$

To extend our approach to such upper bounds, we have begun studying program transformations that transform an unshapely function into a shapely function with the strict size dependency corresponding to an upper bound of the size dependency of the original function. For instance, the `insert` function would be transformed into a shapely function that always inserts the element. We believe that in many practical cases the testing approach combined with program transformations will succeed in providing good upper bounds.

5.3 Wider applicability

In this paper, the work has been presented for a simple functional language over lists. We plan to extend and implement the algorithm for an existing language with more general data structures. Good candidates are XML transformation languages [Wad00,Fri06] because such transformations are very likely to

be shapely. For these applications, the general type inference algorithm will stay the same. The only requirement is that a type checker exists, or is developed, that supports the language.

6 Conclusion

We have developed an algorithm that infers static non-monotonically sized types through interpolating data from run-time tests. Because the dynamically generated types are only accepted after checking them by a formal type checking algorithm, the types are static: the size expressions hold for every possible future run of the program.

The key idea in this approach is the use of a dynamic testing procedure to generate hypotheses for the sized types. This replaces an otherwise infeasible to define formal type inference procedure and essentially reduces type inference to type checking. As a consequence, type inference is complete with respect to type checking.

6.1 Related work

Some interesting initial work on inferring size relations within the output of XML transformations has been done by Su and Wassermann [SW04]. Although this work does not yield output-on-input dependencies, it is able to infer size relations within the output type, for instance if two branches have the same number of elements.

Herrmann and Lengauer have presented a size analysis for functional programs over nested lists [HL01]. However, they do not solve recurrence equations in their size expressions, as this is not important for their goal of program parallelization.

Other work on size analysis has been restricted to monotonic dependencies. Research by Pareto has yielded an algorithm to automatically check linear sized types where size expression are upper bounds [Par98]. Construction of non-linear upper bounds using a traditional type system approach has been presented by Hammond and Vasconcellos [VK04], but this work leaves recurrence equations unsolved and is limited to monotonic dependencies. The work on quasi-interpretations by Amadio [Ama03] also requires monotonic dependencies.

References

[Ama03] R. Amadio. Max-plus quasi-interpretations. In M. Hofmann, editor, *Typed Lambda Calculi and Applications, 6th International Conference, TLCA 2003*, volume 2701 of *Lecture Notes in Computer Science*, pages 31–45, Valencia, Spain, June 10-12 2003. Springer.

[CL87] C. Chui and H.C. Lai. Vandermonde determinant and Lagrange interpolation in R^s. In *Nonlinear and convex analysis*, pages 23–35, 1987.

[ESvK+07] M. van Eekelen, O. Shkaravska, R. van Kesteren, B. Jacobs, E. Poll, and S. Smetsers. Aha: Amortized heap space usage analysis. In M. Morazn and H. Nillsson, editors, *Proceedings of the Eighth Symposium on Trends in Functional Programming, TFP 2007*, pages XVI–1–16, Seton Hall University, New York City, USA, 2-4 april 2007. Submitted for selected papers.

[Fri06] A. Frisch. OCaml + XDuce. In *ICFP '06: Proceedings of the eleventh ACM SIGPLAN international conference on Functional programming*, pages 192–200, New York, NY, USA, 2006. ACM Press.

[HL01] C. A. Herrmann and C. Lengauer. A transformational approach which combines size inference and program optimization. In Walid Taha, editor, *Semantics, Applications, and Implementation of Program Generation (SAIG'01)*, Lecture Notes in Computer Science 2196, pages 199–218. Springer-Verlag, 2001.

[Lor92] R. A. Lorenz. *Multivariate Birkhoff Interpolation, Lecture Notes in Math.*, volume 1516. Springer-Verlag, New York, 1992.

[Mil78] R. Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17(3):348–375, December 1978.

[Myc84] A. Mycroft. Polymorphic type schemes and recursive definitions. In *Proceedings of the 6th International Symopsium on Programming*, LNCS 167, pages 217–228, 1984.

[Par98] L. Pareto. *Sized Types*. Chalmers University of Technology, Göteborg, 1998. Dissertation for the Licentiate Degree in Computing Science.

[SvKvE07] O. Shkaravska, R. van Kesteren, and M. van Eekelen. Polynomial size analysis of first-order functions. In *Proceedings of International Conference on Typed Lambda Calculi and Applications (TLCA) 2007*, LNCS, Paris, June 26–28 2007. Springer. To appear.

[SW04] Z. Su and G. Wassermann. Type-based inference of size relationships for xml transformations. Technical Report U CD//CSE-2004-8, UC Davis, April 2004 2004.

[VK04] P. B. Vasconcelos and Hammond K. Inferring cost equations for recursive, polymorphic and higher-order functional programs. In P. Trinder, G. Michaelson, and R. Peña, editors, *Implementation of Functional Languages: 15th International Workshop, IFL 2003, Edinburgh, UK, September 8–11, 2003. Revised Papers*, volume 3145 of *Lecture Notes in Computer Science*, pages 86–101. Springer-Verlag, Berlin, 2004.

[Wad00] Philip Wadler. A formal semantics of patterns in XSLT and XPath. *Markup Lang.*, 2(2):183–202, 2000.