Polynomial Size Analysis of First-Order Functions

Olha Shkaravska, Ron van Kesteren, and Marko van Eekelen

Institute for Computing and Information Sciences
Radboud University Nijmegen
{O.Shkaravska,R.vanKesteren,M.vanEekelen}@cs.ru.nl

Abstract. We present a size-aware type system for first-order shapely function definitions. Here, a function definition is called *shapely* when the size of the result is determined exactly by a polynomial in the sizes of the arguments. Examples of shapely function definitions may be matrix multiplication and the Cartesian product of two lists.

The type checking problem for the type system is shown to be undecidable in general. We define a natural syntactic restriction such that the type checking becomes decidable, even though size polynomials are not necessarily linear or monotonic. Furthermore, a method that infers polynomial size dependencies for a non-trivial class of function definitions is suggested.[1]

Keywords: Shapely Functions, Size Analysis, Type Checking, Diophantine equations.

1 Introduction

We explore typing support for checking size dependencies for *shapely* first-order function definitions (functions for short). The shapeliness of these functions lies in the fact that the size of the result is a polynomial in terms of the arguments' sizes.

Without loss of generality, we restrict our attention to a language with polymorphic lists as the only data-type. For such a language, this paper develops a size-aware type system for which we define a fully automatic type checking procedure.

A typical example of a shapely function in this language is the Cartesian product, which is given below. It uses the auxiliary function pairs that creates pairs of a single value and the elements of a list. To get a Cartesian product, cprod does this for all elements from the first list separately and appends the resulting intermediate lists. Furthermore, the function definition of append is assumed:

$$\begin{aligned}\mathsf{cprod}(x, y) = \mathsf{match}\ x\ \mathsf{with}\ &\mid \mathsf{nil} \Rightarrow \mathsf{nil}\\ &\mid \mathsf{cons}(hd, tl) \Rightarrow \mathsf{append}(\mathsf{pairs}(hd, y),\ \mathsf{cprod}(tl, y))\end{aligned}$$

[1] This research is sponsored by the Netherlands Organisation for Scientific Research (NWO), project Amortized Heap Space Usage Analysis (AHA), grantnr. 612.063.511.

S. Ronchi Della Rocca (Ed.): TLCA 2007, LNCS 4583, pp. 351–365, 2007.

where

$$\begin{aligned}\mathsf{pairs}(x,y) = \mathsf{match}\ y\ \mathsf{with}\ &\mid \mathsf{nil} \Rightarrow \mathsf{nil}\\ &\mid \mathsf{cons}(hd,tl) \Rightarrow \mathsf{cons}(\mathsf{cons}(x,hd,\mathsf{nil}),\mathsf{pairs}(x,tl))\end{aligned}$$

Given two lists, for instance [1, 2, 3] and [4, 5], it returns the list with all pairs created by taking one element from the first list and one element from the second list: [[1, 4], [1, 5], [2, 4], [2, 5], [3, 4], [3, 5]]. Hence, given two lists of length n and m, it always returns a list of length nm containing pairs. This can be expressed in a type by $\mathsf{L}_n(\alpha) \times \mathsf{L}_m(\alpha) \rightarrow \mathsf{L}_{n*m}(\mathsf{L}_2(\alpha))$.

Shapeliness is restrictive, but it is an important foundational step. It makes type checking decidable in the non-linear case and it allows to infer types "out-of-the-box", since experimental points are positioned exactly on the graph of the polynomial. Exact sizes will be used in future work to derive lower/upper bounds on the output sizes because many non-shapely functions may be transformed into shapely in such a way that the new functions output-size polynomial will be an lower/upper bound for output sizes of the original function. We need such bounds for our AHA project.

1.1 Related Work

Information about input-output size dependencies is applied in time and space analysis and optimization, because run time and heap-space consumption obviously depend on the sizes of the data structures involved in the computations. Knowledge of the exact size of data structures can be used to improve heap space analysis for expressions with destructive pattern matching. Amortized heap space analysis has been developed for linear bounds by Hofmann and Jost [5]. Precise knowledge of sizes is required to extend this approach to non-linear bounds. Another application of exact size information is *load distribution* for parallel computation. For instance, size information helps to distribute a storage effectively and to safely store vector fragments [3].

The analysis of (exact) input-output size dependencies of functions itself has been explored in a series of work. Some interesting work on shape analysis has been done by Jay and Sekanina [7]. In this work, a shapely program expression is translated into a corresponding abstract program expression over sizes. Thus, the dependency of the result size on the argument sizes has the form of a program expression. However, deriving an arithmetic function from it is beyond the scope of their work.

Functional dependencies of sizes in a *recurrent form* may be derived via program analysis and transformation, as in the work of Herrmann and Lengauer [6], or through a type inference procedure, as presented by Vasconcelos and Hammond [12]. Both results can be applied to non-shapely functions, higher-order functions and non-linear size expressions. However, solving the recurrence equations to obtain a closed-form solution is left as an open problem for external solvers. In the second paper monotonic bounds are studied.

To our knowledge, the only work yielding closed-form solutions for size dependencies is limited to monotonic dependencies. For instance, in the well-known

work of Pareto [8], where *non-strict* sized types are used to prove termination, monotonic linear upper bounds are inferred. There linearity is a sufficient condition for the type checking procedure to be decidable.In the series of works on polynomial quasi-interpretations [1] one studies polynomial upper bounds. The checking and inference rely on real arithmetic. Our approach differs twofold. Firstly, quasi-interpretations give monotonic bounds. With non-monotonic size dependencies polynomial quasi-interpretations may lead to significant overestimations. Secondly, to get exact bounds we use integer arithmetic instead of the real one.

The approaches summarized in the previous paragraphs either leave the (possibly undecidable) solving of recurrences as a problem external to their approach, or are limited to monotonic dependencies.

1.2 Contents of the Paper

In this work, we go beyond monotonicity and linearity and consider a type checking procedure for a first-order functional programming language (section 2) with polynomial size dependencies (section 3). We show that type checking is reduced to the entailment checking over Diophantine equations. Type checking is shown to be undecidable in general, but decidable under a natural syntactic condition ("no-let-before-match", section 4). We suggest a method for type inference in section 5. It terminates on a nontrivial class of shapely functions. It non-terminates when either the function under consideration non-terminates, or it is not shapely, or its correct size dependency is rejected by the type-checker due type-checker's incompleteness. (Note that there is no complete shapeliness-checker.)

In section 6 we define a heap-aware semantics of types and expressions and sketch the proof of the soundness statement with respect to this semantics. Finally, in section 7 we overview the results and discuss further work.

2 Language

The typing system is designed for a first-order functional language over integers and (polymorphic) lists.

The syntax of language expressions is defined by the following grammar, where c ranges over integer constants, x and y denote zero-order program variables, and f denotes a function name (the example in the introduction used a sugared version of this syntax):

$$
\begin{array}{rcl}
\mathit{Basic}\ b & ::= & c \mid \mathsf{nil} \mid \mathsf{cons}(x, y) \mid f(x_1, \ldots, x_n) \\
\mathit{Expr}\ \ e & ::= & \mathsf{letfun}\ f(x_1, \ldots, x_n) = e_1\ \mathsf{in}\ e_2 \\
& & \mid b \mid \mathsf{let}\ x = b\ \mathsf{in}\ e \mid \mathsf{if}\ x\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 \\
& & \mid \mathsf{match}\ x\ \mathsf{with} \mid \mathsf{nil} \Rightarrow e_1 \\
& & \qquad\qquad \mid \mathsf{cons}(x_{\mathtt{hd}}, x_{\mathtt{tl}}) \Rightarrow e_2
\end{array}
$$

The syntax distinguishes between zero-order let-binding of variables and first-order letfun-binding of functions. In a function body, the only free program

variables that may occur are its parameters: $FV(e_1) \subseteq \{x_1, \ldots, x_n\}$. The operational semantics is standard, therefore the definition is postponed until it is used to prove soundness (section 6.1).

We prohibit head-nested let-expressions and restrict sub-expressions in function calls to variables to make type-checking straightforward. Program expressions of a general form may be equivalently transformed to expressions of this form. It is useful to think of the presented language as an intermediate language.

3 Type System

Sized types are derived using a type and effect system in which types are annotated with size expressions. Size expressions are polynomials representing lengths of finite lists and arithmetic operations over these lengths:

$$\textit{SizeExpr } p ::= \mathbb{N} \mid n \mid p + p \mid p - p \mid p * p,$$

where n, possibly decorated with sub- and superscripts, denotes a size variable, which stands for any concrete size (natural number). For any natural number k, n^k denotes the k-fold product $n * \ldots * n$.

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

$$\textit{Types } \tau ::= \texttt{Int} \mid \alpha \mid \mathsf{L}_p(\tau),$$

where α is a type variable. Note that this structure entails that if the elements of a list are lists themselves, all these element-lists have to be of the same size. Thus, instead of lists it would be more precise to talk about matrix-like structures. For instance, the type $\mathsf{L}_6(\mathsf{L}_2(\texttt{Int}))$ is given to a list which elements are all lists of exactly two integers, such as $[[1,4],[1,5],[2,4],[2,5],[3,4],[3,5]]$.

It is easy to see that sets $\mathsf{L}_0(\mathsf{L}_m(\texttt{Int}))$ are equal and contain the single element $[\,]$ for all m. The similar holds for $\mathsf{L}_0(\mathsf{L}_m(\alpha))$. This induces the natural equivalence relation on types. For instance $\mathsf{L}_q(\mathsf{L}_0(\mathsf{L}_p(\alpha))) \equiv \mathsf{L}_q(\mathsf{L}_0(\mathsf{L}_{p'}(\alpha)))$. The canonical representative of this class is $\mathsf{L}_q(\mathsf{L}_0(\mathsf{L}_0(\alpha)))$. Everywhere below τ (decorated with sub- or superscripts) denote in fact the canonical representative of $\tau_{\equiv}$. The equivalence expresses the fact that sizes of lists that do not exist, because a containing list is empty, are not important.

The sets $FV(\tau)$ and $FVS(\tau)$ of the free type and size variables of a type τ are defined inductively in the obvious way. Note, that $FVS(\mathsf{L}_0(\mathsf{L}_m(\alpha))) = \emptyset$, since the type is equivalent to $\mathsf{L}_0(\mathsf{L}_0(\alpha))$.

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

$$\textit{GTypes } \tau^\bullet ::= \tau \text{ such that } FVS(\tau) = \emptyset \wedge FV(\tau) = \emptyset\,,$$

First-order types are assigned to shapely functions over values of a zero-order type. Let τ° denote a zero order type of which the annotations are all size variables. First-order types are then defined by:

$$\begin{aligned} \mathit{FTypes}\ \tau^f ::= {} & \tau_1^\circ \times \ldots \times \tau_n^\circ \to \tau_{n+1} \\ & \text{such that } \mathit{FVS}(*\tau_{n+1}) \subseteq \mathit{FVS}(*\tau_1^\circ) \cup \cdots \cup \mathit{FVS}(*\tau_n^\circ) \\ & \text{for all instantiations } * \text{ of type variables with size expressions.} \end{aligned}$$

Recalling the Cartesian product from the introduction, one expects append to be of type $\mathsf{L}_n(\alpha) \times \mathsf{L}_m(\alpha) \to \mathsf{L}_{n+m}(\alpha)$, pairs of type $\alpha \times \mathsf{L}_m(\alpha) \to \mathsf{L}_m(\mathsf{L}_2(\alpha))$, and cprod of type $\mathsf{L}_n(\alpha) \times \mathsf{L}_m(\alpha) \to \mathsf{L}_{n*m}(\mathsf{L}_2(\alpha))$.

A context Γ is a mapping from zero-order variables to zero-order types. A signature Σ is a mapping from function names to first-order types. The definition of $\mathit{FVS}(-)$ is straightforwardly extended to contexts.

3.1 Typing Rules

A typing judgment is a relation of the form $D;\ \Gamma\ \vdash_\Sigma e : \tau$, where D is a set of Diophantine equations which is used to keep track of the size information. In the current language, the only place where size information is available is in the nil-branch of the match-rule. The signature Σ contains the type assumptions for the functions that are going to be checked.

In the typing rules, $D \vdash p = p'$ means that $p = p'$ is derivable from D from equational reasoning in the ring of integers. $D \vdash \tau = \tau'$ is a shorthand that means that τ and τ' have the same underlying type and equality of their size annotations is derivable. The typing judgment relation is defined by the following rules:

$$\frac{}{D;\ \Gamma\ \vdash_\Sigma c : \mathtt{Int}}\ \textsc{IConst} \qquad \frac{D \vdash \tau = \tau'}{D;\ \Gamma,\ x:\tau\ \vdash_\Sigma x : \tau'}\ \textsc{Var}$$

$$\frac{\mathit{FVS}(\mathsf{L}_p(\tau)) \subseteq \mathit{FVS}(\Gamma) \qquad D \vdash p = 0}{D;\ \Gamma\ \vdash_\Sigma \mathsf{nil} : \mathsf{L}_p(\tau)}\ \textsc{Nil}$$

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

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

$$\frac{x \notin \mathit{dom}(\Gamma) \qquad D;\ \Gamma\ \vdash_\Sigma e_1 : \tau_x \qquad D;\ \Gamma,\ x:\tau_x\ \vdash_\Sigma e_2 : \tau}{D;\ \Gamma\ \vdash_\Sigma \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 : \tau}\ \textsc{Let}$$

$$\frac{\begin{array}{c} p = 0,\ D;\ \Gamma,\ x:\mathsf{L}_p(\tau')\ \vdash_\Sigma e_{\mathsf{nil}} : \tau \\ hd, tl \notin \mathit{dom}(\Gamma) \qquad D;\ \Gamma, hd:\tau',\ x:\mathsf{L}_p(\tau'),\ tl:\mathsf{L}_{p-1}(\tau')\ \vdash_\Sigma e_{\mathsf{cons}} : \tau \end{array}}{\begin{array}{rl} D;\ \Gamma,\ x:\mathsf{L}_p(\tau')\ \vdash_\Sigma \mathsf{match}\ x\ \mathsf{with} & \mid \mathsf{nil} \Rightarrow e_{\mathsf{nil}} \\ & \mid \mathsf{cons}(hd, tl) \Rightarrow e_{\mathsf{cons}} \end{array} : \tau}\ \textsc{Match}$$

The rule LetFun demands that all defined functions, including recursive ones, must be in the domain of the signature, and the corresponding first-order type must pass type-checking. We do not prohibit calls of not-defined functions in the code. In practice, a type checker may allow calls of undefined within the given code functions. This happens when a specification comes from a trusty source.

Such relaxed treatment of LetFun does make sense for functions written in another language. However, for the soundness proof one needs all called functions to be defined within an expression under consideration.

$$\frac{\Sigma(f) = \tau_1^\circ \times \cdots \times \tau_n^\circ \to \tau_{n+1} \qquad \mathsf{True};\ x_1{:}\tau_1^\circ, \ldots, x_n{:}\tau_n^\circ \ \vdash_\Sigma e_1 : \tau_{n+1} \qquad D;\ \Gamma \ \vdash_\Sigma e_2 : \tau'}{D;\ \Gamma \ \vdash_\Sigma \mathsf{letfun}\ f(x_1, \ldots, x_n) = e_1\ \mathsf{in}\ e_2 : \tau'}\ \textsc{LetFun}$$

In the FunApp-rule, Θ computes the substitution $*$ from the first argument (whose size expressions, by definition of first order types, are always variables) to the second argument, and the set C of equations over size expressions from ${\tau_1}' \times \cdots \times \tau_k'$. The set C contains $p = p'$ if and only the expressions are substituted to the same size variable, like, for instance, to m in $\mathsf{L}_m(\mathtt{Int}) \times \mathsf{L}_m(\mathtt{Int}) \to \mathtt{Int}$.

$$\frac{\langle *, C\rangle = \Theta(\tau_1^\circ \times \cdots \times \tau_n^\circ, {\tau_1}' \times \cdots \times {\tau_n}') \qquad \Sigma(f) = \tau_1^\circ \times \ldots \times \tau_n^\circ \to \tau_{n+1} \qquad D \vdash *(\tau_{n+1}) = \tau'_{n+1} \qquad D \vdash C}{D;\ \Gamma, x_1{:}{\tau_1}', \ldots, x_n{:}{\tau_n}' \ \vdash_\Sigma f(x_1, \ldots, x_k){:}{\tau_{n+1}}'}\ \textsc{FunApp}$$

The type system needs no conditions on non-negativity of size expressions. Size expressions in types of meaningful data structures are always non-negative. The soundness of the type system (section 6.2) ensures that this property is preserved throughout (the evaluation of) a well-typed expression.

3.2 Example of Type Checking

Because for every syntactic construction there is only one typing rule that is applicable, type checking is straightforward. In the introduction, the Cartesian product was presented using a "sugared" syntax. Here, we present the cprod function in the language defined in section 2.

$$\begin{array}{l}\mathsf{letfun}\ \mathsf{cprod}(x, y) = \mathsf{match}\ x\ \mathsf{with} \mid \mathsf{nil} \Rightarrow \mathsf{nil} \\ \qquad\qquad\qquad\qquad\qquad\qquad \mid \mathsf{cons}(hd, tl) \Rightarrow \mathsf{let}\ z_1 \quad\ = \mathsf{pairs}(hd, y) \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\ \ \mathsf{in\ let}\ z_2 = \mathsf{cprod}(tl, y) \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\ \ \mathsf{in\ append}(z_1, z_2)\end{array}$$

Functions pairs and append are assumed to be defined in the core syntax of the language as well. Hence, Σ contains the following types:

$$\begin{array}{ll}\Sigma(\mathsf{append}) & = \mathsf{L}_n(\alpha) \times \mathsf{L}_m(\alpha) \to \mathsf{L}_{n+m}(\alpha) \\ \Sigma(\mathsf{pairs}) & = \alpha \times \mathsf{L}_m(\alpha) \to \mathsf{L}_m(\mathsf{L}_2(\alpha)) \\ \Sigma(\mathsf{cprod}) & = \mathsf{L}_n(\alpha) \times \mathsf{L}_m(\alpha) \to \mathsf{L}_{n*m}(\mathsf{L}_2(\alpha))\end{array}$$

To type-check $\mathsf{cprod} : \mathsf{L}_n(\alpha) \times \mathsf{L}_m(\alpha) \to \mathsf{L}_{n*m}(\mathsf{L}_2(\alpha))$ means to check:

Prove: $x{:}\mathsf{L}_n(\alpha), y{:}\mathsf{L}_m(\alpha) \ \vdash_\Sigma e_{\mathsf{cprod}}{:}\mathsf{L}_{n*m}(\mathsf{L}_2(\alpha))$,

where e_{cprod} is the function body. This is demanded by the first branch of the LetFun-rule. Applying the Match-rule branches the proof:

$$\begin{array}{ll}
\textsc{Nil}: & n=0;\ y:\mathsf{L}_m(\alpha)\ \vdash_\Sigma \mathsf{nil}:\mathsf{L}_{n*m}(\mathsf{L}_2(\alpha)) \\
\textsc{Cons}: & hd:\alpha,\ x:\mathsf{L}_n(\alpha),\ tl:\mathsf{L}_{n-1}(\alpha),\ y:\mathsf{L}_m(\alpha)\ \vdash_\Sigma \\
& \qquad \mathsf{let}\ z_1 \quad = \mathsf{pairs}(hd,y):\mathsf{L}_{n*m}(\mathsf{L}_2(\alpha)) \\
& \qquad \mathsf{in\ let}\ z_2 = \mathsf{cprod}(tl,y) \\
& \qquad \mathsf{in}\ \mathsf{append}(z_1,z_2)
\end{array}$$

Applying the NIL-rule to the NIL-branch gives $n = 0 \vdash n * m = 0$, which is trivially true. The CONS-branch is proved by applying the LET-rule twice. This results in three proof obligations:

$$\begin{array}{ll}
\textsc{Bind-z1}: & hd:\alpha,\ y:\mathsf{L}_m(\alpha)\ \vdash_\Sigma \mathsf{pairs}(hd,y):\tau_1 \\
\textsc{Bind-z2}: & tl:\mathsf{L}_{n-1}(\alpha),\ y:\mathsf{L}_m(\alpha)\ \vdash_\Sigma \mathsf{cprod}(tl,y):\tau_2 \\
\textsc{Body}: & z_1:\tau_1, z_2:\tau_2\ \vdash_\Sigma \mathsf{append}(z_1,z_2):\mathsf{L}_{n*m}(\alpha)
\end{array}$$

From the applications of the FUNAPP-rule to BIND-Z1 and BIND-Z2 it follows that τ_1 should be $\mathsf{L}_m(\mathsf{L}_2(\alpha))$ and τ_2 should be $\mathsf{L}_{(n-1)*m}(\mathsf{L}_2(\alpha))$. Lastly, applying the FUNAPP-rule to BODY yields the proof obligation $\vdash (n-1)*m+m = n*m$, which is true in the axiomatics.

3.3 Example with Negative Coefficients

In contrast to the system presented by Vasconcelos and Hammond [12], where only subtraction of constants are allowed, our system allows negative coefficients in size expressions. Of course, this is only a valid size expression if the polynomial is non-negative for all values of its variables. Here, we show an example where this is the case. Given two lists, the function "subtracts" elements from lists simultaneously, till one of the lists is empty. Then, the Cartesian product of the rest list with itself is returned:

$$\begin{array}{l}
\mathsf{sqdiff}\ (l_1,\ l_2) = \\
\quad \mathsf{match}\ l_1\ \mathsf{with} \mid \mathsf{nil} \Rightarrow \mathsf{cprod}\ l_2\ l_2 \\
\qquad\qquad\qquad\quad \mid \mathsf{cons}(h,t) \Rightarrow \mathsf{match}\ l_2\ \mathsf{with} \mid \mathsf{nil} \Rightarrow \mathsf{cprod}\ l_1\ l_1 \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \mid \mathsf{cons}(h',t') \Rightarrow \mathsf{sqdiff}\ (t,\ t')
\end{array}.$$

It can be checked that sqdiff has type $\mathsf{L}_n(\alpha) \times \mathsf{L}_m(\alpha) \to \mathsf{L}_{(n^2+m^2-2*n*m)}(\mathsf{L}_2(\alpha))$.

4 Decidability Issues for Type Checking

In the examples above, type checking ends up with a set of entailments like $n = 0 \vdash 0 = n * m$ or $\vdash m + m * (n-1) = m * n$ that have to hold. However, we show that there is no procedure that can check all entailments that possibly arise. To make type checking decidable, we formulate a syntactical condition on the structure of a program expression that ensures the entailments have a trivial form. The idea is to *prohibit pattern-matchings in a let-body.*

4.1 Type Checking in General Is Undecidable

We show that the existence of a procedure that may check all possible entailments at the end of type checking is reduced to Hilbert's tenth problem: whether

there exists a general procedure that given a polynomial with integer coefficients decides if this polynomial has natural roots or not.[2] Matiyasevich [10] has shown that such a procedure does not exist. This means that type checking, in the general case, is undecidable as well.

We show that type checking is reducible to a procedure of checking if arbitrary size polynomials of shapely functions have natural roots. It turns out that the latter is the same as finding natural roots of integer polynomials.

Consider the following expression e_H with free variables $x_1, \ldots, x_k$:

$$\begin{array}{l}\mathsf{let}\ x = f_0(x_1, \ldots, x_k)\ \mathsf{in}\ \mathsf{match}\ x\ \mathsf{with} \mid \mathsf{nil} \Rightarrow f_1(x_1, \ldots, x_k) \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \mid \mathsf{cons}(hd, tl) \Rightarrow f_2(x_1, \ldots, x_k)\end{array}$$

We check if it has the type $\mathsf{L}_{n_1}(\alpha_1) \times \ldots \times \mathsf{L}_{n_k}(\alpha_k) \longrightarrow \mathsf{L}_{p(n_1,\ldots,n_k)}(\alpha)$, given that $f_i : \mathsf{L}_{n_1}(\alpha_1) \times \ldots \times \mathsf{L}_{n_k}(\alpha_k) \longrightarrow \mathsf{L}_{p_i(n_1,\ldots,n_k)}(\alpha)$, with $i = 0, 1, 2$. Then at the end of the type checking procedure we obtain the entailment:

$$p_0(n_1, \ldots, n_k) = 0 \vdash p_1(n_1, \ldots, n_k) = p(n_1, \ldots, n_k).$$

Even if p and p_1 are not equal, say $p_1 = 0$ and $p = 1$, it does not mean that type checking fails; it might not be possible to enter the "bad" nil-branch. To check if the nil-branch is entered means to check if $p_0 = 0$ has a solution in natural numbers. Thus, a type-checker for any size polynomial p_0 must be able to define if it has natural roots or not.

Checking if any size polynomial has roots in natural numbers, is the same as checking whether an arbitrary polynomial has roots or not. For polynomials $q(n_1, \ldots, n_k) = 0$ if and only if $q^2(n_1, \ldots, n_k) = 0$ so it is sufficient to prove that the square of any polynomial is a size polynomial for some shapely function. First, note that any polynomial q may be presented as the difference $q_1 - q_2$ of two polynomials with non-negative coefficients[3]. So, $q^2 = (q_1 - q_2)^2$ is a size polynomial, obtained by superposition of sqdiff with q_1 and q_2. Here q_1 and q_2 are size polynomials with positive coefficients for corresponding compositions of cprod and append functions.

So, existence of a general type-checker reduces to solving Hilbert's tenth problem. Hence, type checking is undecidable.

We can show this in a more constructive way using the stronger form of the undecidability of Hilbert's tenth problem: for any type-checking procedure $\mathcal{I}$ one can construct an expression, for which $\mathcal{I}$ fails to give the correct answer. We will use the result of Matiyasevich who has proved the following: there is a one-parameter Diophantine equation $W(a, n_1, \ldots, n_k) = 0$ and an algorithm which for given algorithm $\mathcal{A}$ produces a number $a_{\mathcal{A}}$ such that $\mathcal{A}$ fails to give the correct answer for the question whether equation $W(a_{\mathcal{A}}, n_1, \ldots, n_k) = 0$ has a solution in $(n_1, \ldots, n_k)$. So, if in the example above one takes the function

[2] The original formulation is about integer roots. However, both versions are equivalent and logicians consider natural roots.

[3] If $q = \Sigma a_{i_1,\ldots,i_k} x_1^{i_1} \ldots x_k^{i_k}$, then $q_1 = \Sigma_{a_{i_1,\ldots,i_k} \geq 0} a_{i_1,\ldots,i_k} x_1^{i_1} \ldots x_k^{i_k}$, and $q_2 = \Sigma_{a_{i_1,\ldots,i_k} < 0} |a_{i_1,\ldots,i_k}| x_1^{i_1} \ldots x_k^{i_k}$.

f_0 such that its size polynomial p_0 is the square of the $W(a_{\mathcal{I}}, n_1, \ldots, n_k)$ and $p = 1$, $p_1 = 0$, then the type checker $\mathcal{I}$ fails to give the correct answer for e_H.

For checking a particular expression it is sufficient to solve the corresponding sets of Diophantine equations. Type checking depends on decidability of Diophantine equations from D in any entailment $D \vdash p = p'$, where p is not equal to p' in general (but might be if the equations from D hold). If we have a solution for D we can substitute this solution in p and p'. A solution over variables $n_1, \ldots, n_m, n_{m+1}, \ldots, n_k$ is a set of equations $n_i = q_i(n_{m+1}, \ldots, n_k)$ where $1 \leq i \leq m$. The expressions for n_i are substituted into $p = p'$ and one trivially checks the equality of the two polynomials over $n_{m+1}, \ldots, n_k$ in the axiomatics of the integers' ring. Recall that two polynomials are equal if and only if the coefficient at monomials with the same degrees of variables are equal.

4.2 Syntactical Condition for Decidability

The most simple way to ensure decidability is to require that all equations in D have the form $n = c$, where c is a constant. This would in particular exclude the example e_H from above. As we will see below, this requirement can be fulfilled by imposing a syntactical condition on program expressions: "no let before match".

The refined grammar of the language is defined as the main grammar where the let-construct in e is replaced by $\mathsf{let}\ x = b\ \mathsf{in}\ e_{nomatch}$ with

$$\begin{array}{rl} e_{nomatch} := b & \mid \mathsf{letfun}\ f(x_1, \ldots, x_n) = e\ \mathsf{in}\ e' \\ \mid \mathsf{let}\ x = b\ \mathsf{in}\ e_{nomatch} & \mid \mathsf{if}\ x\ \mathsf{then}\ e'_{nomatch}\ \mathsf{else}\ e'_{nomatch} \end{array}$$

Theorem 1. *Let a program expression e satisfy the refined grammar, and let us check the judgment* $\mathsf{True};\ x_1 : \tau_1^o, \ldots, x_k : \tau_k^o \vdash_\Sigma e : \tau$. *Then, at the end of the type-checking procedure one has to check entailments of the form*

$$D \vdash p' = p,$$

where D is a set of equations of the form $n - c = 0$ for some $n \in FVS(\tau_1^o \times \ldots \times \tau_k^o)$ and constant c and p, p' are polynomials in $FVS(\tau_1^o \times \ldots \times \tau_k^o)$.

Sketch of the proof. Consider a path in the type checking tree which ends up with some $D \vdash p' = p$ and let an equation $q = 0$ belong to D. It means that in the path there is the nil-branch of the pattern matching for some $x : \mathsf{L}_q(\tau)$.

By induction on the length of the path, one can show that $q = n - c$ for some size variable $n \in FVS(\tau_1 \times \ldots \times \tau_k)$ and some constant c. This uses the fact, that variables which are not free in an expression and pattern-matched may be introduced only by another pattern-matching, but not a let-binding. The technical report contains the full proof [11].

Note, that prohibiting pattern matching in `let`-bodies is very natural, since it prohibits "risky" definitions of the form $f(x) = g(f(f_0(x)))$. Here x is a non-nil list, and f_0 is a function over lists, possibly with the property $|f_0(x)| \geq |x|$, with $|\cdot|$ denoting length, so termination of f becomes questionable. In a "shapely world" the condition $|f_0(x)| < |x|$ for all x starting from a certain x_0, which ensures termination, implies $|f_0(x)| = |x| - c$ or $|f(x)| = c$ for some constant c.

In principle, any program expression that does not do pattern matching on a variable bound by a let-expression may be recoded so that it satisfies the refined grammar and defines the same map. For instance, an expression

$$\mathsf{let}\ x' = f_0(y)\ \mathsf{in}\ \mathsf{match}\ x\ \mathsf{with}\ \mid \mathsf{nil} \Rightarrow f_1(x,\, x') \mid \mathsf{cons}(hd, tl) \Rightarrow f_2(x,\ x')$$

and the expression

$$\mathsf{match}\ x\ \mathsf{with}\ \mid \mathsf{nil} \Rightarrow \mathsf{let}\ x' = f_0(y)\ \mathsf{in}\ f_1(x,\, x') \mid \mathsf{cons}(hd, tl) \Rightarrow \mathsf{let}\ x' = f_0(y)\ \mathsf{in}\ f_2(x,\, x')$$

define the same map of lists.

Of course, the syntactical condition of the theorem may be relaxed. One may allow expressions with pattern-matching in a let-body, assuming that functions that appear in let-bindings, like f_0, give rise to solvable Diophantine equations. For instance, when p_0 is a linear function, one of the variables is expressed via the others and the constant and substituted into $p_1 = p$. Or, p_0 is a 1-variable quadratic, cubic or degree 4 equation. We leave relaxations of the condition for future work.

5 Method for Type Inference

Here we discuss type inference under the syntactical condition defined in the previous section. Since we consider shapely functions, there is a way to reduce type inference to type-checking using the well-known fact that a finite polynomial is defined by a finite number of points.

For each size polynomial in the output type of a given function, one assumes a hypothesis about the degree and the variables. Then, to obtain the coefficients, the function is run (preferably in a sand-box) as many times as the number of coefficients the polynomial has. This finite number of input-output size pairs defines a system of linear equations, where the unknowns are the coefficients of the polynomial. When (the sizes of the data-structures in) the set of input data satisfies some criteria known from the polynomial interpolation theory [4,9], the system has a unique solution. Input sizes that satisfy these criteria, which are nontrivial for multivariate polynomials, can be determined algorithmically.

The interpolation theory used in the previous paragraph finds the Lagrange interpolation of a size function. If the hypothesis about the degree and the variables of the size expression was correct, the Lagrange interpolation coincides with that desired size function. To check if this is the case, the interpolation is given to the type checking procedure. If it passes, it is correct. Otherwise, one may repeat the procedure for a higher degree of the polynomial.

The method, that is the sequence of such loops, non-terminates when

- the function under consideration does not terminate on test data,
- the function is non-shapely,
- the function is shapely but the type-checker rejects it due to the type system's incompleteness (see 6.3). Note that no complete algorithm for shapeliness-checking exists, even for functions subject to the syntactical restriction.

The method infers polynomial size dependencies for a nontrivial class of shapely functions.

For instance, standard type inference for the underlying type system yields that the function cprod has the following underlying type $\mathsf{cprod} : \mathsf{L}(\alpha) \times \mathsf{L}(\alpha) \longrightarrow \mathsf{L}(\mathsf{L}(\alpha))$. Adding size annotations with unknown output polynomials gives $\mathsf{cprod} : \mathsf{L}_n(\alpha) \times \mathsf{L}_m(\alpha) \longrightarrow \mathsf{L}_{p_1}(\mathsf{L}_{p_2}(\alpha))$. We assume p_1 is quadratic so we have to compute the coefficients in its presentation:

$$p_1(x, y) = a_{0,0} + a_{0,1}x + a_{1,0}y + a_{1,1}xy + a_{0,2}x^2 + a_{2,0}y^2$$

Running the function cprod on six pairs of lists of length 0, 1, 2 yields:

n	m	x	y	$\mathsf{cprod}(x, y)$	$p_1(n, m)$	$p_2(n, m)$
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

This defines the following linear system for the external output list:

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

The unique solution is $a_{1,1} = 1$ and the rest of coefficients are zero. To verify whether the interpolation is indeed the size polynomial, one checks if $\mathsf{cprod} : \mathsf{L}_n(\alpha) \times \mathsf{L}_m(\alpha) \longrightarrow \mathsf{L}_{n*m}(\mathsf{L}_2(\alpha))$. This is the case, as was shown in section 3.2.

As an alternative way of finding the coefficients, one could try to directly solve the systems defined by entailments $D \vdash p = p'$. When the degree is assumed, the unknowns in these systems are the polynomial coefficients. However, the systems are nonlinear in general [11]. By combining testing with type checking we do not have to solve these nonlinear Diophantine equations anymore.

6 Semantics of the Type System

Informally, soundness of the type system ensures that "well-typed programs will not go wrong". This is achieved by demanding that, when a function is given meaningful values of the types required as arguments, the result will be a meaningful value of the output type.

In section 6.1, we formalize the notion of a meaningful value using a heap-aware semantics of types and give an operational semantics of the language. Section 6.2 formulates the soundness statement with respect to this semantics and sketches the proof. The system is shown not to be complete in section 6.3.

6.1 Semantics of Program Values and Expressions

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

$$\begin{array}{l} \mathit{Val}\ v\ ::=\ c \mid \ell \mid \texttt{NULL} \qquad \ell \in \mathit{Loc} \qquad c \in \texttt{Int} \\ \mathit{Hp}\ h\ :\ \mathit{Loc} \rightharpoonup \{\texttt{hd}, \texttt{tl}\} \rightharpoonup \mathit{Val} \end{array}$$

We will write $h[\ell.\texttt{hd} := v_h,\ \ell.\texttt{tl} := v_t]$ for the heap equal to h everywhere but in ℓ, which at the `hd`-field of ℓ gets value v_h and at the `tl`-field of ℓ gets value v_t.

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

$$\begin{array}{lll} i & \models^h_{\texttt{Int}} & i \\ \texttt{NULL} & \models^h_{\texttt{L}_0(\tau)} & \texttt{[]} \\ \ell & \models^h_{\texttt{L}_n(\tau)} & w_{\texttt{hd}} :: w_{\texttt{tl}} \ \text{iff}\ n \geq 1, \ell \in \mathit{dom}(h), \\ & & \qquad h.\ell.\texttt{hd} \models^{h|_{\mathit{dom}(h)\setminus\{\ell\}}}_{\tau} w_{\texttt{hd}}, \\ & & \qquad h.\ell.\texttt{tl} \models^{h|_{\mathit{dom}(h)\setminus\{\ell\}}}_{\texttt{L}_{n-1}(\tau)} w_{\texttt{tl}} \end{array}$$

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

When a function body is evaluated, a frame store maintains the mapping from program variables to values. It only contains the actual function parameters, thus preventing access beyond the caller's frame. Formally, a frame store is a finite partial map from variables to values:

$$\mathit{Store}\ s\ :\ \mathit{ExpVar} \rightharpoonup \mathit{Val}$$

Using heaps and frame stores, and maintaining a mapping $\mathcal{C}$ from function names to bodies for the functions definitions encountered, the operational semantics of expressions is defined by the following rules:

$$\frac{c \in \texttt{Int}}{s;\ h;\ \mathcal{C}\ \vdash\ c\ \rightsquigarrow\ c;\ h}\ \text{OSICons} \qquad \frac{}{s;\ h;\ \mathcal{C}\ \vdash\ x\ \rightsquigarrow\ s(x);\ h}\ \text{OSVar}$$

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

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

$$\frac{s(x) \neq 0 \qquad s;\ h;\ \mathcal{C}\ \vdash\ e_1\ \rightsquigarrow\ v;\ h'}{s;\ h;\ \mathcal{C}\ \vdash\ \mathsf{if}\ x\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2\ \rightsquigarrow\ v;\ h'}\ \text{OSIfTrue}$$

$$\frac{s(x) = 0 \qquad s;\, h;\, \mathcal{C} \;\vdash\; e_2 \;\rightsquigarrow\; v;\, h'}{s;\, h;\, \mathcal{C} \;\vdash\; \mathsf{if}\ x\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 \;\rightsquigarrow\; v;\, h'}\ \textsc{OSIfFalse}$$

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

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

$$\frac{s;\, h;\, \mathcal{C}[f := ((x_1, \ldots, x_n) \times e_1)] \;\vdash\; e_2 \;\rightsquigarrow\; v;\, h'}{s;\, h;\, \mathcal{C} \;\vdash\; \mathsf{letfun}\ f(x_1, \ldots, x_n) = e_1\ \mathsf{in}\ e_2 \;\rightsquigarrow\; v;\, h'}\ \textsc{OSLetFun}$$

$$\frac{\begin{array}{c} s(x_1) = v_1 \;\ldots\; s(x_m) = v_n \qquad \mathcal{C}(f) = (y_1, \ldots, y_n) \times e_f \\ [y_1 := v_1, \ldots, y_n := v_n];\, h;\, \mathcal{C} \;\vdash\; e_f \;\rightsquigarrow\; v;\, h' \end{array}}{s;\, h;\, \mathcal{C} \;\vdash\; f(x_1, \ldots, x_n) \;\rightsquigarrow\; v;\, h'}\ \textsc{OSFunApp}$$

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

6.2 Soundness

In this subsection the soundness theorem is formulated and a proof is sketched. The technical report [11] contains the full proof.

Let a valuation ϵ map size variables to concrete (natural) sizes and an instantiation μ map type variables to ground types:

Valuation $\epsilon : \mathit{SizeVar} \to \mathbb{N}$

Instantiation $\eta : \mathit{TypeVar} \to \tau^{\bullet}$

Applied to a type, context, or size equation, valuations (and instantiations) map all variables occurring in it to their valuation (or instantiation) images.

The soundness statement is defined by means of the following two predicates. One indicates if a program value is meaningful with respect to a certain heap and ground type. The other does the same for sets of values and types, taken from a frame store and context respectively:

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

Now the soundness statement is straightforward:

Theorem 2. *Let* $s;\, h;\, [\,] \;\vdash\; e \;\rightsquigarrow\; v;\, h'$ *and all called in* e *functions are defined in it via the let-fun construct. Then for any context* Γ*, signature* Σ *and type* τ*, such that* $\mathsf{True};\ \Gamma \vdash_{\Sigma} e : \tau$ *is derivable in the type system, and any size valuation* ϵ *and type instantiation* η*, it holds that if the store is meaningful w.r.t. the context* $\eta(\epsilon(\Gamma))$ *then the output value is meaningful w.r.t the type* $\eta(\epsilon(\tau))$*:*

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

Sketch of the proof. The proof is done by induction on the length of the operational semantics derivation tree and is presented in the technical report [11]. The proof for the LET-rule relies on the *benign sharing* [5] of data structures. It means that the heap data to be used further are not changed by the head expression in let. There are type systems approximating this semantic condition, e.g. linear typing and uniqueness typing [2] We consider sharing aware type systems separately and combine with the resource aware one afterwords.

6.3 Completeness

The system is not complete – there are shapely functions that are not well-typed. For instance, the type checking fails for the function $\mathsf{faildueif} : \mathsf{L}_n(\mathtt{Int}) \to \mathsf{L}_n(\mathtt{Int})$ defined by:

$\mathsf{letfun}\ \mathsf{faildueif}(x) = \mathsf{let}\ y = \mathsf{length}(x)\ \mathsf{in}\ \mathsf{if}\ y\ \mathsf{then}\ x\ \mathsf{else}\ \mathsf{nil}$

where $\mathsf{length}(x)$ returns the length of list x. We believe that in some cases program transformations might help to make such functions typable.

7 Conclusion and Further Work

We have presented a natural syntactic restriction such that type checking of a size-aware type system for first-order shapely functions is decidable for polynomial size expressions without any limitations on the degree of the polynomials.

A non-standard, practical method to infer types is introduced. It uses runtime results to generate a set of equations. These equations are linear and hence automatically solvable. The method terminates on a non-trivial class of shapely functions.

7.1 Further Work

The system is defined for polymorphic lists. In principle, the system may be extended so that more general data structures will be allowed. This extension should not influence the approach itself, however it brings additional technical overhead.

An obvious limitation of our approach is that we consider only shapely functions. In practice, one is often interested to obtain upper bounds on space complexity for non-shapely functions. A simple example where for a non-shapely function an upper bound would be useful, is the function to `insert` an element in a list, provided the list does not contain the element. In the future we plan to consider code transformations which, given a non-shapely function `f` with upper bound (worst-case) complexity `c`, translate it into a shapely function `f'` with complexity `c`. Effectively, this will make the analysis applicable to non-shapely functions obtaining upper bounds on the space consumption complexity.

We plan to add non-trivial sizes to integers. At the same time leaving out non-sized integers will result in lists with elements of different sizes. Hence, the

decision how to add sizes to integers is connected to the problem of using sized and non-sized types within the same system. We leave it for future work based on [12] and [7].

Addition of other data structures and extension to non-shapely functions will open the possibility to use the system for an actual programming language.

References

1. Bonfante, G., Marion, J.-Y., Moyen, J.-Y.: Quasi-interpretations, a way to control resources. Theoretical Computer Science (to appear)
2. Barendsen, E., Smetsers, S.: Uniqueness typing for functional languages with graph rewriting semantics. Mathematical Structures in Computer Science 6, 579–612 (1996)
3. Chatterjee, S., Blelloch, G.E., Fischer, A.L.: Size and access inference for data-parallel programs. PLDI '91: Proceedings of the ACM SIGPLAN 1991 conference on Programming language design and implementation, pp. 130–144 (1991)
4. Chui, C., Lai, H.C.: Vandermonde determinant and Lagrange interpolation in R^s. Nonlinear and convex analysis, pp. 23–35 (1987)
5. Hofmann, M., Jost, S.: Static prediction of heap space usage for first-order functional programs. SIGPLAN Not. 38(1), 185–197 (2003)
6. Herrmann, C.A., Lengauer, C.: A transformational approach which combines size inference and program optimization. In: Taha, W. (ed.) SAIG 2001. LNCS, vol. 2196, pp. 199–218. Springer, Heidelberg (2001)
7. Jay, C.B., Sekanina, M.: Shape checking of array programs. In: Computing: the Australasian Theory Seminar, Proceedings. Australian Computer Science Communications, vol. 19, pp. 113–121 (1997)
8. Pareto, L.: Sized Types. Dissertation for the Licentiate Degree in Computing Science. Chalmers University of Technology (1998)
9. Lorenz, R.A.: Multivariate Birkhoff Interpolation. LNCS, vol. 1516. Springer, Heidelberg (1992)
10. Matiyasevich, Y., Jones, J.-P.: Proof or recursive unsolvability of Hilbert's tenth problem. American Mathematical Monthly 98(10), 689–709 (1991)
11. Shkaravska, O., van Kesteren, R., van Eekelen, M.: polynomial size analysis of first-order functions. Technical Report ICIS-R07004, Radboud University Nijmegen (2007)
12. Vasconcelos, P.-B., Hammond, K.: Inferring cost equations for recursive, polymorphic and higher-order functional programs (Revised Papers). In: Trinder, P., Michaelson, G.J., Peña, R. (eds.) IFL 2003. LNCS, vol. 3145, pp. 86–101. Springer, Heidelberg (2004)