Logical Methods in Computer Science
Vol. 5 (2:10) 2009, pp. 1–35
www.lmcs-online.org

Submitted Jan. 4, 2006
Published May 25, 2009

POLYNOMIAL SIZE ANALYSIS OF FIRST-ORDER SHAPELY FUNCTIONS *

OLHA SHKARAVSKA [a], MARKO VAN EEKELEN [b], AND RON VAN KESTEREN [c]

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

[b] Institute for Computing and Information Sciences, Radboud University Nijmegen *and* Faculty of Information Science, Open University of the Netherlands
e-mail address: marko@cs.ru.nl *and* marko.vaneekelen@ou.nl

[c] Alten Nederland, Consulting and Engineering in Advanced Technology
e-mail address: ronvankesteren@gmail.com

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 implementations of matrix multiplication and the Cartesian product of two lists.

The type system is proved to be sound w.r.t. the operational semantics of the language. The type checking problem 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, we have shown that the type-inference problem is at least semi-decidable (under this restriction). We have implemented a procedure that combines run-time testing and type-checking to automatically obtain size dependencies. It terminates on total typable function definitions.

1998 ACM Subject Classification: F.4.1[Mathematical logic and formal languages]: Mathematical logic – Lambda calculus and related systems, Logic and constraint programming; F.2.2 [Analysis of algorithms and problem complexity]: Non-numerical algorithms and problems; D.1.1 [Programming techniques]: Applicative (functional) programming. General Terms: Algorithms, Verification.

Key words and phrases: Shapely Functions, Size Analysis, Type Checking, Type Inference, Diophantine equations, Polynomial Interpolation.

* This paper is an extended version of the paper [ShvKvE07b], presented at the TLCA conference in 2007. The paper is extended by the full soundness proof and the full presentation of the type-inference procedure from [vKShvE07]. Moreover, in the presented version we consider rational size polynomials instead of integer ones.

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

LOGICAL METHODS IN COMPUTER SCIENCE

DOI:10.2168/LMCS-5 (2:10) 2009

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.

1.1. **Variety of resource analysis techniques.**

This research is a part of the Amortised Heap Space Usage Analysis (AHA) project [vEShvK07]. Estimating heap consumption is an active research area as it becomes more and more of an issue in many applications, including programming for small devices, e.g. smart cards, mobile phones, embedded systems and distributed computing.

Amortization is a promising technique to obtain accurate bounds of resource consumption and gain. An amortised estimate of a resource does not target a single operation but a sequence of operations. One assigns some amortised cost to an operation. This amortised cost may be higher or lower than the operation's actual cost. For the sequence considered, it is important that its overall amortised cost covers its overall actual cost. An amortised cost of the sequence lies between its actual cost and the simple multiplication of the worst-case of one operation by the length of the sequence. An amortised cost of the sequence is in many cases easier to compute than its actual cost and it is obviously better than the worst-case estimate.

Combining amortization with type theory allows to infer linear heap-consumption bounds for functional programs with explicit memory deallocation [HofJost03]. The *AHA* project aims to adapt this method for *non-linear* bounds within (lazy) functional programs and transfer the results to the object-oriented programming. Contrary to linear amortised bounds, to obtain non-linear heap estimates one does need to know sizes of structures that takes part in computation, see, for instance [vEShvK07].

The AHA project seems to be part of an emerging trend since a growing number of works are addressing resource analysis. Here we mention some of them.

In [AmZil] the authors develop new method to statically (polynomially) bound the resources needed for the execution of systems of concurrent threads. The method generalises an approach designed for first-order functional languages that relies on a combination of standard termination techniques for term rewriting systems and an analysis of the size of the computed values based on the notion of a polynomial quasi-interpretation. Quasi-interpretations were applied to size analysis firstly in [BonMarMoy05b]. In [AvMoSch08] the authors describe a fully automated tool that implements a few techniques that directly classify run-time complexity (i.e. techniques that use the number of rewrite steps as complexity measure), including polynomial quasi-interpretations.

Several groups have studied programming languages with *implicit computational complexity* (ICC) properties. This line of research is motivated both by the perspective of automated complexity analysis, and by foundational goals, in particular to give natural characterisations of complexity classes, like PTIME or PSPACE. In [Gir92] characterisation of PTIME is given in terms of bounded linear logic. In [GabMarRon08] one proposes a characterization of PSPACE by means of an extension of (soft affine) typed lambda calculus. For this extension, the authors design a call-by-name evaluation machine in order to compute programs in polynomial space. In [AtBailTer07] one addresses the problem of typing lambda-terms in a variant of second-order light linear logic. The authors give a procedure which, starting with a term typed in system F, determines whether it is typable in the logic. It is shown that the procedure can be run in time polynomial in the size of the original Church typed system F term.

Resource analysis may be performed within a *Proof Carrying Code* framework. In [AsMcK06] one introduces the notion of a resource policy for mobile code to be run on smart devices. Such a resource policy is integrated in a proof-carrying code architecture. Two forms of policy are used: guaranteed policies which come with proofs and target policies which describe limits of the device.

In [AlArGenPuebZan07] one describes resource consumption for Java bytecode by means of Cost Equation Systems (CESs), which are similar to, but more general than recurrence equations. CESs express the cost of a program in terms of the size of its input data. In a further step, a closed form (i.e., non-recursive) solution or upper bound can sometimes be found by using existing Computer Algebra Systems, such as Maple and Mathematica. This work is continued by the authors in [AlArGenPueb08], where mechanisms of constructing solutions of CESs and upper bounds are studied closely. They consider monotonic cost expressions only.

In [Ben01] the author describes the Automated Complexity Analysis Prototype (ACAp) system for automated time analysis of functional programs. Symbolic evaluation of recursive programs generates systems of multi-variable difference equations, which are solved using Mathematica.

In [GuMeCh09] the authors describe a technique for computing symbolic bounds on the number of statements a procedure executes in terms of its inputs and user defined size functions. The technique is based on multiple counter instrumentation that allows to compute linear bounds individually for each counter. The bounds on these counters are then composed to generate total bounds that are non-linear and disjunctive.

1.2. **Exploring size dependencies.** In this paper 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 and inference procedure.

A typical example of a shapely function in this language is cprod that computes the Cartesian product of two sets, stored as lists. It is given below. The auxiliary function pairs creates pairs of a single value and the elements of a list. To get a Cartesian product the function 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:

$$\begin{array}{lcll} \mathsf{cprod}(l_1, l_2) & = & \mathsf{match}\ l_1\ \mathsf{with} & \mid \mathsf{nil} \Rightarrow \mathsf{nil} \\ & & & \mid \mathsf{cons}(hd, tl) \Rightarrow \mathsf{append}(\mathsf{pairs}(hd, l_2),\ \mathsf{cprod}(tl, l_2)) \end{array}$$

where

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

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 is expressed by the type $\mathsf{L}_n(\alpha) \times \mathsf{L}_m(\alpha) \rightarrow \mathsf{L}_{n*m}(\mathsf{L}_2(\alpha))$.

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. We need such bounds for investigating amortised resource bounds in the AHA project. Nonlinear amortised resource consumption relies on the size of input data, and its gain is calculated based on the size of output.

In this paper our only concern is in sizes of input and output. For instance, the time and space complexity of a function definition with a polynomial input-output size dependency may exceed polynomial space and time consumption due to internal structures and computations.

1.3. **Related work on size analysis.** Information about input-output size dependencies is applied to 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. Amortised heap space analysis has been developed for linear bounds by Hofmann and Jost [HofJost03]. 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 [Chat90].

The analysis of (exact) input-output size dependencies of functions itself has been explored in a series of works. Some interesting work on shape analysis has been done by Jay and Sekanina [JaySek97]. 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 [HerLen01], or through a type inference procedure, as presented by Vasconcelos and Hammond [VasHam03]. 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 [Par98], 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- [BonMarMoy05b] and sup-interpretations [MarPech] one studies max-polynomial upper bounds. The checking and inference rely on real arithmetic. In general, (inference) synthesis procedures are exponential w.r.t. the size of a program. For *multilinear* polynomials in *max-plus*-algebra it is shown to be of polynomial complexity [Am05].

Our approach differs two-fold. Firstly, quasi-interpretations give monotonic bounds. With non-monotonic size dependencies polynomial quasi-interpretations may lead to significant over-estimations. Secondly, to get exact bounds we use rational arithmetic instead of real arithmetic. Our motivation for this choice lies in the fact that one should use decidability procedures in reals with care, if one applies them to integers or naturals. For instance, $x^2 \leq x^3$ holds in naturals, but not in reals, since it does not hold on $0 < x < 1$.

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.

1.4. **Content 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).

In subsection 3.1 we define zero-order types and their set-theoretic semantics. In subsections 3.2 and 3.3 we define first-order types and give typing rules respectively. The soundness of type system w.r.t. the operational semantics of the language is studied in subsection 3.4. The type system is not complete in the class of all shapely functions, and no such complete system exists (subsection 3.5).

In section 4 we show that type checking is reduced to the entailment checking over Diophantine equations. Type checking is shown to be undecidable in general (subsection 4.2). However, type-checking is decidable under certain syntactic condition for function bodies (subsection 4.3).

We define in detail a method for type inference in section 5. It terminates on a non-trivial class of shapely functions. It does not terminate when either the function under consideration does not terminate, or it is not shapely, or its correct size dependency is rejected by the type-checker due type-system's incompleteness.

Finally, in section 6 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 (the example in the introduction used a sugared version of this syntax):

$$
\begin{array}{llll}
\mathit{Basic} & b & ::= & c \mid x\,\mathsf{binop}\,y \mid \mathsf{nil} \mid \mathsf{cons}(z,l) \mid f(z_1,\ldots,z_n) \\
\mathit{Expr} & e & ::= & b \\
 & & & \mid \mathsf{let}\ z = b\ \mathsf{in}\ e_1 \\
 & & & \mid \mathsf{if}\ x\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 \\
 & & & \mid \mathsf{match}\ l\ \mathsf{with}\ \mid \mathsf{nil} \Rightarrow e_1 \\
 & & & \qquad\qquad\qquad \mid \mathsf{cons}(z,l') \Rightarrow e_2 \\
 & & & \mid \mathsf{letfun}\ f(z_1,\ldots,z_n) = e_1\ \mathsf{in}\ e_2 \\
 & & & \mid \mathsf{letextern}\ f(z_1,\ldots,z_n)\ \mathsf{in}\ e_1
\end{array}
$$

where c ranges over integer constants, z, x, y, l denote zero-order program variables (x and y range over integer variables, l possibly decorated with sub- ans superscripts, ranges over lists and z ranges over program variables when their types are not relevant), binop is one of the four integer binary operations: $+$, $-$, div, mod, and f denotes a function name.

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: $FV(e_1) \subseteq \{z_1, \ldots, z_n\}$. The operational semantics is standard, therefore the definition is postponed until it is used to prove soundness (section 3.4).

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.

For practical reasons and in order to support modularity, we introduce a letextern *declaration*, which makes it possible to call functions implemented in other modules that may be defined in other languages.

3. Type System

We consider a type system, constituted from zero- and first-order types, corresponding typing rules for program constructs and Peano arithmetic extended to rational numbers as (classes of equivalence of) pairs of integers, rational addition and multiplication[1].

3.1. Zero-order types and their semantics.

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:

$$\mathit{SizeExpr} \quad p \quad ::= \quad \mathcal{Q} \mid n \mid p + p \mid p - p \mid p * p$$

where $\mathcal{Q}$ denotes rational numbers, and 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 $n * \ldots * n$.

Size expressions are rational polynomials that map natural numbers into natural numbers. For instance, the polynomial $p(n) = \dfrac{n(n+1)}{2}$ represents the size dependency of the function $\mathsf{progression}$:

$$\begin{array}{lcl}\mathsf{progression}(l) & = & \mathsf{match}\ l\ \mathsf{with} \mid \mathsf{nil} \Rightarrow \mathsf{nil} \\ & & \qquad\qquad\quad\; \mid \mathsf{cons}(hd, tl) \Rightarrow \mathsf{append}(\mathsf{progression}(tl),\ l)\end{array}$$

For example, it maps [1, 2, 3] on [3, 2, 3, 1, 2, 3]. The output size dependency is given by the arithmetic progression $0 + 1 + \ \ldots\ + (n-1) + n$, where n is the size of an input. This explains the name of the function [vKShvE07].

Zero-order types are assigned to program values, which are interpreted as integer numbers and finite lists. A list type is annotated with a size expression that represents the length of the list:

$$\mathit{Types} \quad \tau \quad ::= \quad \mathtt{Int} \mid \alpha \mid \mathsf{L}_p(\tau)$$

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

[1] Rational addition is defined as $\dfrac{a}{b} + \dfrac{c}{d} = \dfrac{ad+cb}{bd}$. Rationals with their addition and multiplication form a field, more precisely a field of integer fractions.

It is easy to see that for all m the types $\mathsf{L}_0(\mathsf{L}_m(\mathtt{Int}))$ are equal, because they represent the singleton containing $[\,]$. The same holds for $\mathsf{L}_0(\mathsf{L}_m(\alpha))$. This induces a 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 equivalence expresses the fact that the size of a list is not relevant when such a list does not exist, because an outer list is empty. Now, we define formally an entailment $D \vdash \tau = \tau'$, where D is a conjunction of equations between polynomials. The definition is inductive on τ. The entailment $D \vdash \tau = \tau'$ holds if and only if

- $\tau = \tau' = \mathtt{Int}$ or $\tau = \tau' = \alpha$ for some type variable α;
- $\tau = \mathsf{L}_p(\tau'')$ and $\tau' = \mathsf{L}_{p'}(\tau''')$ have the same underlying type (i.e. the type with annotations omitted) and
  (1) $D \vdash p = p'$, and
  (2) $D \vdash p = 0$ or $D \vdash \tau'' = \tau'''$,

with $D \vdash p = q$ being an arithmetical entailment, meaning $\forall\, \bar{n}.D(\bar{n}) \to p(\bar{n}) = q(\bar{n})$, where $\bar{n}$ is the collection of all size variables taken from D, q and p. For instance,

$$\begin{array}{l} m = 0 \vdash \mathsf{L}_{n+m}(\alpha) = \mathsf{L}_n(\alpha) \quad \text{and} \\ m - 1 = 0,\, n = 0 \vdash \mathsf{L}_{n+m-1}(\mathsf{L}_2(\alpha)) = \mathsf{L}_n(\mathsf{L}_3(\alpha)) \end{array}$$

hold, whereas $n = 0 \vdash \mathsf{L}_{n+m-1}(\mathsf{L}_2(\alpha)) = \mathsf{L}_{m-1}(\mathsf{L}_3(\alpha))$ does not.

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:

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

In our semantic model a heap is essentially 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 a 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. A heap is a finite partial mapping from locations and fields to program values:

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

We will write $h.\ell.\mathtt{hd}$ and $h.\ell.\mathtt{tl}$ for the results of applications $h\ \ell\ \mathtt{hd}$ and $h\ \ell\ \mathtt{tl}$, which denote the values stored in the heap h at the location ℓ at fields `hd` and `tl`, respectively. Let $h[\ell.\mathtt{hd} := v_h,\ \ell.\mathtt{tl} := v_t]$ denote 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 interpretation with respect to a specific heap h and a ground type τ. It is given via the four-place relation $v \models^h_\tau w$, where integer constants interprets themselves, and locations are interpreted as non-cyclic lists:

$$\begin{array}{llll} c & \models^h_{\mathtt{Int}} & c & \\ \mathtt{NULL} & \models^h_{\mathsf{L}_0(\tau^\bullet)} & \mathtt{[]} & \\ \ell & \models^h_{\mathsf{L}_{n^\bullet}(\tau^\bullet)} & w_{\mathtt{hd}} :: w_{\mathtt{tl}} & \text{iff} \quad n \geq 1, \ell \in dom(h), \\ & & & \qquad h.\ell.\mathtt{hd} \ \models^{h|_{dom(h)\setminus\{\ell\}}}_{\tau^\bullet} \ w_{\mathtt{hd}}, \\ & & & \qquad h.\ell.\mathtt{tl} \ \models^{h|_{dom(h)\setminus\{\ell\}}}_{\mathsf{L}_{n^\bullet-1}(\tau^\bullet)} \ w_{\mathtt{tl}} \end{array}$$

where $n^{\bullet}$ is a natural constant and $h|_{dom(h)\setminus\{\ell\}}$ denotes the heap equal to h everywhere except for ℓ, where it is undefined.

3.2. **First-order types.** 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:

$$\begin{array}{lll} \mathit{FTypes} \quad \tau^{f} & ::= & \tau^{\circ}_{1} \times \ldots \times \tau^{\circ}_{n} \to \tau_{n+1} \\ & & \text{such that } \mathit{FVS}(\tau_{n+1}) \subseteq \mathit{FVS}(\tau^{\circ}_{1}) \cup \cdots \cup \mathit{FVS}(\tau^{\circ}_{n}) \end{array}$$

For instance, one expects that the following function definitions (in the sugared syntax[2]) will be well-typed in the system:

```
append : $\mathsf{L}_n(\alpha) \times \mathsf{L}_m(\alpha) \to \mathsf{L}_{n+m}(\alpha)$
append($l_1$, $l_2$) = match $l_1$ with | nil $\Rightarrow l_2$
                                        | cons($hd$, $tl$) $\Rightarrow$ cons($hd$, append($tl$, $l_2$))

pairs : $\alpha \times \mathsf{L}_n(\alpha) \to \mathsf{L}_n(\mathsf{L}_2(\alpha))$
pairs($x$, $l$) = match $l$ with | nil $\Rightarrow$ nil
                                 | cons($hd$, $tl$) $\Rightarrow$ let $l'$ = cons($x$, cons($hd$, nil))
                                                        in cons($l'$, pairs($x$, $tl$))

cprod : $\mathsf{L}_n(\alpha) \times \mathsf{L}_m(\alpha) \to \mathsf{L}_{n*m}(\mathsf{L}_2(\alpha))$
cprod($l_1$, $l_2$)match $l_1$ with | nil $\Rightarrow$ nil
                                    | cons($hd$, $tl$) $\Rightarrow$ append(pairs($hd$, $l_2$), cprod($tl$, $l_2$))

sqdiff : $\mathsf{L}_n(\alpha) \times \mathsf{L}_m(\alpha) \to \mathsf{L}_{(n^2+m^2-2*n*m)}(\mathsf{L}_2(\alpha))$
sqdiff($l_1$, $l_2$) = match $l_1$ with | nil $\Rightarrow$ cprod($l_2$, $l_2$)
                                        | cons($hd$, $tl$) $\Rightarrow$ match $l_2$ with | nil $\Rightarrow$ cprod($l_1$, $l_1$)
                                                                      | cons($hd'$, $tl'$) $\Rightarrow$ sqdiff ($tl$, $tl'$)
```

For *total* functions the following condition is necessary: *for all instantiations * of size variables with themselves or zeros, the inclusion $\mathit{FVS}(*\tau_{n+1}) \subseteq \mathit{FVS}(*\tau^{\circ}_{1}) \cup \cdots \cup \mathit{FVS}(*\tau^{\circ}_{n})$ holds.* Consider, for instance, the first-order type $\mathsf{L}_n(\mathsf{L}_m(\alpha)) \to \mathsf{L}_m(\mathsf{L}_n(\alpha))$, where on nil input, i.e. with $n = 0$, the input type degenerates to $\mathsf{L}_0(\mathsf{L}_m(\alpha)) \equiv \mathsf{L}_0(\mathsf{L}_0(\alpha))$ but the outer list of the output must have length m. This m becomes unknown being "hidden" in $\mathsf{L}_0(\mathsf{L}_m(\alpha))$. Thus, this first-order type may be accepted without the condition above, once a function of this type is *partial and undefined on empty lists.* Since the type $\mathsf{L}_n(\mathsf{L}_m(\alpha)) \to \mathsf{L}_m(\mathsf{L}_n(\alpha))$ may be assigned to an implementation of $n \times m$-matrix transposition, undefinedness on nil may be interpreted as an exception "cannot transpose an empty matrix".

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.

[2] In the sugared syntax we use $f(g(z))$ for "let $z' = g(z)$ in $f(z')$"

3.3. **Typing rules.** A typing judgement is a relation of the form $D;\ \Gamma\ \vdash_\Sigma e\!:\!\tau$, where D is a conjunction of equations between polynomials. D is used to keep track of 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 called in the expression under consideration. The typing judgement relation is defined by the following rules:

$$\frac{}{D;\ \Gamma\ \vdash_\Sigma c\!:\!\mathtt{Int}}\ \textsc{IConst} \qquad \frac{}{D;\ \Gamma,\ x\!:\!\mathtt{Int},\, y\!:\!\mathtt{Int}\ \vdash_\Sigma x\,\mathsf{binop}\,y\!:\!\mathtt{Int}}\ \textsc{IBinop}$$

$$\frac{D \vdash p = 0}{D;\ \Gamma\ \vdash_\Sigma \mathsf{nil}\!:\!\mathsf{L}_p(\tau)}\ \textsc{Nil} \qquad \frac{D \vdash \tau = \tau'}{D;\ \Gamma,\ z\!:\!\tau\ \vdash_\Sigma z\!:\!\tau'}\ \textsc{Var}$$

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

$$\frac{\begin{array}{c} D;\ \Gamma,\ x\!:\!\mathtt{Int}\ \vdash_\Sigma e_t\!:\!\tau \\ D;\ \Gamma,\ x\!:\!\mathtt{Int}\ \vdash_\Sigma e_f\!:\!\tau \end{array}}{D;\ \Gamma,\ x\!:\!\mathtt{Int}\ \vdash_\Sigma \mathsf{if}\ x\ \mathsf{then}\ e_t\ \mathsf{else}\ e_f\!:\!\tau}\ \textsc{If}$$

$$\frac{\begin{array}{c} z \notin \mathit{dom}(\Gamma) \\ D;\ \Gamma\ \vdash_\Sigma e_1\!:\!\tau_z \\ D;\ \Gamma,\ z\!:\!\tau_z\ \vdash_\Sigma e_2\!:\!\tau \end{array}}{D;\ \Gamma\ \vdash_\Sigma \mathsf{let}\ z = e_1\ \mathsf{in}\ e_2\!:\!\tau}\ \textsc{Let}$$

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

The rule LetFun demands that all letfun-defined functions, including recursive ones, must be in the domain of the signature, and the corresponding first-order type must pass type-checking:

$$\frac{\begin{array}{c} \Sigma(f) = \tau_1^\circ \times \cdots \times \tau_n^\circ \to \tau_{n+1} \\ \mathsf{True};\ z_1\!:\!\tau_1^\circ, \ldots, z_n\!:\!\tau_n^\circ\ \vdash_\Sigma e_1\ :\ \tau_{n+1} \\ D;\ \Gamma\ \vdash_\Sigma e_2\ :\ \tau' \end{array}}{D;\ \Gamma\ \vdash_\Sigma \mathsf{letfun}\ f(z_1, \ldots, z_n) = e_1\ \mathsf{in}\ e_2\!:\!\tau'}\ \textsc{LetFun}$$

However, in practice we do not prohibit calls to functions that are not defined via letfun. If a function coming from a trusty external source together with its first-order type is declared via letextern, one applies the LetExtern rule:

$$\frac{\begin{array}{c} \Sigma(f) = \tau_1^\circ \times \cdots \times \tau_n^\circ \to \tau_{n+1} \\ D;\ \Gamma\ \vdash_\Sigma e\ :\ \tau' \end{array}}{D;\ \Gamma\ \vdash_\Sigma \mathsf{letextern}\ f(z_1, \ldots, z_n)\ \mathsf{in}\ e\!:\!\tau'}\ \textsc{LetExtern}$$

When proving soundness we require all functions to be defined via letfun within an expression under consideration.

In the FunApp-rule, Θ computes the substitution $*$ from its first argument (whose size expressions are always variables since they are taken from the first-order signature of the function) to its 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 if the expressions p and p' are substituted to the same size variable. For instance, if a function $\mathsf{dotprod} : \mathsf{L}_m(\mathtt{Int}) \times \mathsf{L}_m(\mathtt{Int}) \rightarrow \mathtt{Int}$ is called with actual parameters of the types $\mathsf{L}_{n+n'+2}(\mathtt{Int})$ and $\mathsf{L}_{n+3}(\mathtt{Int})$, then C contains the equation $n + n' + 2 = n + 3$.

$$\frac{\begin{array}{c}\langle *, C\rangle = \Theta(\tau_1^\circ \times \cdots \times \tau_n^\circ, {\tau_1}' \times \cdots \times {\tau_n}') \\ \Sigma(f) = \tau_1^\circ \times \ldots \times \tau_n^\circ \rightarrow \tau_{n+1} \qquad D \vdash \tau'_{n+1} = *(\tau_{n+1}) \qquad D \vdash C\end{array}}{D;\ \Gamma, z_1 : {\tau_1}', \ldots, z_n : {\tau_n}' \ \vdash_\Sigma f(z_1, \ldots, z_k) : {\tau_{n+1}}'}\ \textsc{FunApp}$$

In the example with the call of dotprod the equation $n + n' + 2 = n + 3$ holds if D contains $n' - 1 = 0$.

As another example of the FunApp-rule consider the recursive call $\mathsf{append}(tl,\, l_2)$ in the definition of append:

$$\frac{\begin{array}{c}\Sigma(\mathsf{append}) = \mathsf{L}_n(\alpha) \times \mathsf{L}_m(\alpha) \rightarrow \mathsf{L}_{n+m}(\alpha) \\ \vdash \tau = *(\mathsf{L}_{n+m}(\alpha))\end{array}}{tl : \mathsf{L}_{n-1}(\alpha),\ l_2 : \mathsf{L}_m(\alpha)\ \vdash_\Sigma \mathsf{append}(tl,\, l_2) : \tau}\ \textsc{FunApp}$$

Here $\Theta(\mathsf{L}_n(\alpha) \times \mathsf{L}_m(\alpha),\ \mathsf{L}_{n-1}(\alpha) \times \mathsf{L}_m(\alpha)) = \langle *, \emptyset\rangle$ with $*(n) = n - 1$, $*(m) = m$. Thus, $\tau = *(\mathsf{L}_{n+m}(\alpha)) = \mathsf{L}_{n-1+m}(\alpha)$.

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 ensures that this property is preserved throughout (the evaluation of) a well-typed expression.

See subsection 4.1 for examples of type checking in detail.

3.4. **Soundness of the type system.** Informally, soundness of the type system ensures that "well-typed programs will not go wrong". This means that if function arguments have meaningful values according to their types then the result will have a meaningful value of the output type. In section 3.1, we formalized the notion of a meaningful value using a heap-aware semantics of types. Here we give an operational semantics of the language.

We introduce a *frame store* as a mapping from program variables to program values. This mapping is maintained when a function body is evaluated. Before evaluation of the function body starts, the store contains only the actual parameters of the function. During evaluation, the store is extended with the variables introduced by pattern matching or let-constructs. These variables are eventually bound to the actual parameters, thus there is no access beyond the current frame. Formally, a frame store 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 the bodies of the function definitions, and a mapping $\mathcal{E}$ of external function names to the external implementations, the operational semantics of expressions is defined by the following rules:

$$\frac{c \in \mathtt{Int}}{s;\ h;\ \mathcal{C},\ \mathcal{E}\ \vdash\ c\ \leadsto\ c;\ h}\ \textsc{OSIConst}$$

$$\frac{}{s;\ h;\ \mathcal{C},\ \mathcal{E}\ \vdash\ x\ \mathsf{binop}\, y\ \leadsto\ s(x)\mathsf{binop}\, s(y);\ h}\ \textsc{OSIBinop}$$

$$\frac{}{s;\ h;\ \mathcal{C},\ \mathcal{E}\ \vdash\ \mathsf{nil}\ \leadsto\ \mathtt{NULL};\ h}\ \textsc{OSNil} \qquad \frac{}{s;\ h;\ \mathcal{C},\ \mathcal{E}\ \vdash\ z\ \leadsto\ s(z);\ h}\ \textsc{OSVar}$$

$$\frac{s(hd) = v_{\mathtt{hd}} \qquad s(tl) = v_{\mathtt{tl}} \qquad \ell \notin dom(h)}{s;\ h,\ \mathcal{C},\ \mathcal{E}\ \vdash\ \mathsf{cons}(hd, tl)\ \leadsto\ \ell;\ h[\ell.\mathtt{hd} := v_{\mathtt{hd}},\ \ell.\mathtt{tl} := v_{\mathtt{tl}}]}\ \textsc{OSCons}$$

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

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

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

$$\frac{s(l) = \mathtt{NULL} \qquad s;\ h;\ \mathcal{C},\ \mathcal{E}\ \vdash\ e_1\ \leadsto\ v;\ h'}{\begin{array}{l} s;\ h;\ \mathcal{C},\ \mathcal{E}\ \vdash\ \mathsf{match}\ l\ \mathsf{with}\ \mid \mathsf{nil} \Rightarrow e_1 \qquad\quad \leadsto\ v;\ h' \\ \qquad\qquad\qquad\qquad\qquad\quad\ \mid \mathsf{cons}(hd, tl) \Rightarrow e_2 \end{array}}\ \textsc{OSMatch-Nil}$$

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

$$\frac{\begin{array}{c} s;\ h;\ \mathcal{C}[f := ((z_1, \ldots, z_n) \times e_1)],\ \mathcal{E}\ \vdash\ e_2\ \leadsto\ v;\ h' \\ FV(e_1) \subseteq \{z_1,\ \ldots,\ z_n\} \end{array}}{s;\ h;\ \mathcal{C},\ \mathcal{E}\ \vdash\ \mathsf{letfun}\ f(z_1, \ldots, z_n) = e_1\ \mathsf{in}\ e_2\ \leadsto\ v;\ h'}\ \textsc{OSLetFun}$$

$$\frac{\begin{array}{c} s(z_1) = v_1\ \ldots\ s(z_n) = v_n \qquad \mathcal{C}(f) = (z'_1, \ldots, z'_n) \times e_f \\ [z'_1 := v_1, \ldots, z'_n := v_n];\ h;\ \mathcal{C},\ \mathcal{E}\ \vdash\ e_f\ \leadsto\ v;\ h' \\ FV(e_f) \subseteq \{z'_1,\ \ldots,\ z'_n\} \end{array}}{s;\ h;\ \mathcal{C},\ \mathcal{E}\ \vdash\ f(z_1, \ldots, z_n)\ \leadsto\ v;\ h'}\ \textsc{OSFunApp}$$

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 a ground type. The other does the same for sets of values and types, taken from a frame store and a ground context $\Gamma^{\bullet}$, respectively:

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

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

$$\begin{array}{llcl} \mathit{Valuation} & \epsilon & : & \mathit{SizeVar} \to \mathcal{Z} \\ \mathit{Instantiation} & \eta & : & \mathit{TypeVar} \to \tau^{\bullet} \end{array}$$

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

Now, stating the soundness theorem is straightforward:

Theorem 3.1 (Soundness)**.** *Let $s;\ h;\ [\,],\ [\,] \vdash e \rightsquigarrow v;\ h'$ and all functions called in e be 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 for 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 $\eta(\epsilon(\tau))$:*

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

The theorem follows from the following general statement:

Lemma 3.2 (Soundness)**.** *For any s, h, $\mathcal{C}$, e, v, h', a set of equations D, a context Γ, a signature Σ, a type τ, a size valuation ϵ and a type instantiation η such that*

- $s;\ h;\ \mathcal{C},\ [\,] \vdash e \rightsquigarrow v;\ h'$,
- $D;\ \Gamma \vdash_{\Sigma} e : \tau$ *is derivable in the type system and all functions called in e are declared via* letfun*,*

one has

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

The proof is done by induction on the size of the derivation tree for the operational-semantics judgement. For the LET-rule it relies on *benign sharing* [HofJost03] of data structures. With benign sharing, shared heap structures to be used in the let-body are not changed by the let-binding expression of let. To formalize the notion of benign sharing we introduce a function *footprint* $\mathcal{R} : \mathit{Heap} \times \mathit{Val} \longrightarrow \mathcal{P}(\mathit{Loc})$, which computes the set of locations accessible in a given heap from a given value:

$$\begin{array}{ll} \mathcal{R}(h,\ c) & = \emptyset \\ \mathcal{R}(h,\ \mathtt{NULL}) & = \emptyset \\ \mathcal{R}(h,\ \ell) & = \left\{ \begin{array}{l} \emptyset,\ \ \mathit{if}\ \ell \notin \mathit{dom}(h) \\ \{\ell\} \cup \mathcal{R}(h|_{\mathit{dom}(h)\setminus\{\ell\}},\ h.\ell.\mathtt{hd}) \cup \mathcal{R}(h|_{\mathit{dom}(h)\setminus\{\ell\}},\ h.\ell.\mathtt{tl}),\ \mathit{if}\ \ell \in \mathit{dom}(h) \end{array} \right. \end{array}$$

where $f|_X$ denotes the restriction of a (partial) map f to a set X.

We extend $\mathcal{R}$ to stores by $\mathcal{R}(h,\ s) = \bigcup_{z \in \mathit{dom}(s)} \mathcal{R}(h,\ s(z))$. So, the operational-semantics let-rule with benign sharing looks as follows:

$$\frac{\begin{array}{l} s;\ h;\ \mathcal{C},\mathcal{E} \vdash e_1 \rightsquigarrow v_1;\ h_1 \\ s[z := v_1];\ h_1;\ \mathcal{C},\mathcal{E} \vdash e_2 \rightsquigarrow v;\ h' \\ h|_{\mathcal{R}(h,\ s|_{FV(e_2)})} = h_1|_{\mathcal{R}(h,\ s|_{FV(e_2)})} \end{array}}{s;\ h;\ \mathcal{C},\mathcal{E} \vdash \mathsf{let}\ z = e_1\ \mathsf{in}\ e_2 \rightsquigarrow v;\ h'}\ \mathrm{OSL{\small ET}}$$

This semantic condition is not statically typable in general, however, there are type systems that approximate it, e.g. linear typing and uniqueness typing [BarSm96]. Since in our language we have neither destructive pattern matching nor assignments, benign sharing is guaranteed.

Proof. Let everywhere below $s;\, h;\, \mathcal{C} \;\vdash\; e \;\rightsquigarrow\; v;\, h'$ denote the operational-semantics judgement $s;\, h;\, \mathcal{C},\, [\,] \;\vdash\; e \;\rightsquigarrow\; v;\, h'$ with the empty external closure.

In the proof we will use a few technical lemmata about heaps and model relations. They are intuitively clear statements like “extending a heap does not change a model relation”, so we do not prove them in the main part of the paper. The interested reader may find the technical proofs in the appendix.

For the sake of convenience we will denote $\eta(\epsilon(\tau))$ via $\tau_{\eta\epsilon}$, $\eta(\epsilon(\Gamma))$ via $\Gamma_{\eta\epsilon}$ and $\epsilon(D)$ via D_ϵ.

We prove the statement by induction on the height of the derivation tree for the operational-semantics judgement. Given $s;\, h;\, \mathcal{C} \;\vdash\; e \;\rightsquigarrow\; v;\, h'$ fix some Γ, Σ, and τ, such that $D;\; \Gamma \;\vdash_\Sigma e : \tau$. Fix a valuation $\epsilon \in FV(\Gamma) \cup FV(\tau) \to \mathcal{Z}$, a type instantiation $\eta \in FV(\Gamma) \cup FV(\tau) \to \tau^\bullet$, such that D_ϵ and $\mathit{Valid}_{\mathsf{store}}(FV(e), \Gamma_{\eta\epsilon}, s, h)$ hold. We must show that $\mathit{Valid}_{\mathsf{val}}(v, \tau_{\eta\epsilon}, h')$ holds.

OSIConst: In this case $v = c$ for some constant c and $\tau = \mathtt{Int}$. Then, by the definition we have $c \;\models^h_{\mathtt{Int}}\; c$ and $\mathit{Valid}_{\mathsf{val}}(v, \mathtt{Int}, h')$.

OSNull: In this case $v = \mathtt{NULL}$ and $\tau = \mathsf{L}_0(\tau')$ for some τ'. Then, by the definition we have $\mathtt{NULL} \;\models^h_{\mathsf{L}_0(\tau'_{\eta\epsilon})}\; \mathtt{[]}$.

OSVar: From $D \vdash \tau = \tau'$ and D_ϵ it follows that $\tau_{\eta\epsilon} = \tau'_{\eta\epsilon}$. From this and

$$\mathit{Valid}_{\mathsf{store}}(FV(z), \Gamma \cup (z : \tau')_{\eta\epsilon}, h, s)$$

it follows that

$$\mathit{Valid}_{\mathsf{val}}(s(z), \tau_{\eta\epsilon}, h)$$

OSCons: In this case $e = \mathsf{cons}(hd, tl)$, $\tau = \mathsf{L}_p(\tau')$, $\{hd : \tau', tl : \mathsf{L}_{p'}(\tau')\} \subseteq \Gamma$ for some hd, tl, p' and τ'. Since $\mathit{Valid}_{\mathsf{store}}(FV(e), \Gamma_{\eta\epsilon}, s, h)$ there exist $w_{\mathtt{hd}}$ and $w_{\mathtt{tl}}$ such that $s(hd) \;\models^h_{\tau'_{\eta\epsilon}}\; w_{\mathtt{hd}}$ and $s(tl) \;\models^h_{(\mathsf{L}_{p'}(\tau'))_{\eta\epsilon}}\; w_{\mathtt{tl}}$. From the operational semantics judgement we have that $v = \ell$ for some location $\ell \notin \mathit{dom}(h)$, and $h' = h[\ell.\mathtt{hd} := s(hd),\; \ell.\mathtt{tl} := s(tl)]$. Therefore, $h'.\ell.\mathtt{hd} \;\models^h_{\tau'_{\eta\epsilon}}\; w_{\mathtt{hd}}$ and $h'.\ell.\mathtt{tl} \;\models^h_{(\mathsf{L}_{p'}(\tau'))_{\eta\epsilon}}\; w_{\mathtt{tl}}$ hold as well. It is easy to see that $h = h'|_{\mathit{dom}(h')\setminus\{\ell\}}$.

Thus,

$$\begin{aligned} h'.\ell.\mathtt{hd} \;&\models^{h'|_{\mathit{dom}(h')\setminus\{\ell\}}}_{\tau'_{\eta\epsilon}}\; w_{\mathtt{hd}} \\ h'.\ell.\mathtt{tl} \;&\models^{h'|_{\mathit{dom}(h')\setminus\{\ell\}}}_{(\mathsf{L}_{p'}(\tau'))_{\eta\epsilon}}\; w_{\mathtt{tl}} \end{aligned}$$

This and D_ϵ, which implies $p_\epsilon = (p' + 1)_\epsilon$ gives $\ell \;\models^{h'}_{(\mathsf{L}_p(\tau'))_{\eta\epsilon}}\; w_{\mathtt{hd}} :: w_{\mathtt{tl}}$ and thus $\mathit{Valid}_{\mathsf{val}}(\ell, \tau_{\eta\epsilon}, h')$.

OSIfTrue: In this case $e = \mathsf{if}\ x\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2$ for some e_1, e_2, and x. Knowing that $D;\; \Gamma \;\vdash_\Sigma e_1 : \tau$ we apply the induction hypothesis to the derivation of $s;\, h;\, \mathcal{C} \;\vdash\; e_1 \;\rightsquigarrow\; v;\, h'$, with the same η, ϵ to obtain $\mathit{Valid}_{\mathsf{store}}(FV(e_1), \Gamma_{\eta\epsilon}, s, x) \implies \mathit{Valid}_{\mathsf{val}}(v, \tau_{\eta\epsilon}, h')$. From $FV(e_1) \subseteq FV(e)$, $\mathit{Valid}_{\mathsf{store}}(FV(e), \Gamma_{\eta\epsilon}, s, x)$, and lemma 6.7 it follows that $\mathit{Valid}_{\mathsf{val}}(v, \tau_{\eta\epsilon}, h')$.

OSIfFalse: is similar to the true-branch.

OSLetFun: The result follows from the induction hypothesis for

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

with $D;\ \Gamma\ \vdash_\Sigma e_2 : \tau$ and the same η, ϵ, store s and heap h.

OSLet: In this case $e = \mathsf{let}\ z = e_1\ \mathsf{in}\ e_2$ for some z, e_1, and e_2 and we have $s;\ h;\ \mathcal{C}\ \vdash\ e_1\ \rightsquigarrow\ v_1;\ h_1$ and $s[z := v_1];\ h_1;\ \mathcal{C}\ \vdash\ e_2\ \rightsquigarrow\ v;\ h'$ for some v_1 and h_1. We know that $D;\ \Gamma\ \vdash_\Sigma e_1 : \tau'$, $z \notin \Gamma$ and $D;\ \Gamma, z : \tau'\ \vdash_\Sigma e_2 : \tau$ for some τ'. Applying the induction hypothesis to the first branch gives $\mathit{Valid}_{\mathsf{store}}(FV(e_1), \Gamma_{\eta\epsilon}, s, h) \implies \mathit{Valid}_{\mathsf{val}}(v_1, \tau'_{\eta\epsilon}, h_1)$. Since $FV(e_1) \subseteq FV(e_1) \cup (FV(e_2) \setminus \{z\}) = FV(e)$ and

$$\mathit{Valid}_{\mathsf{store}}(FV(e), \Gamma_{\eta\epsilon}, s, h)$$

we have from lemma 6.7 that $\mathit{Valid}_{\mathsf{store}}(FV(e_1), \Gamma_{\eta\epsilon}, s, h)$ holds and hence we have $\mathit{Valid}_{\mathsf{val}}(v_1, \tau'_{\eta\epsilon}, h_1)$.

Now apply the induction hypothesis to the second branch to get

$$\mathit{Valid}_{\mathsf{store}}(FV(e_2), \Gamma_{\eta\epsilon} \cup \{z : \tau'_\epsilon\}, s[z := v_1], h_1) \implies \mathit{Valid}_{\mathsf{val}}(v, \tau_{\eta\epsilon}, h').$$

Now we will show that the l.h.s. of the implication holds. Fix some $z' \in FV(e_2)$. If $z' = z$, then $\mathit{Valid}_{\mathsf{val}}(v_1, \tau'_{\eta\epsilon}, h_1)$ implies $\mathit{Valid}_{\mathsf{val}}(s[z := v_1](z), \tau'_{\eta\epsilon}, h_1)$. If $z' \neq z$, then $s[z := v_1](z') = s(z')$. Because we know that sharing is benign, $h|_{\mathcal{R}(h,\ s(z'))} = h_1|_{\mathcal{R}(h,\ s(z'))}$, applying lemma 6.5 and then 6.7 we have that $s(z')\ \models^{h}_{\Gamma_{\eta\epsilon}(z')}\ w_{z'}$ implies $s(z')\ \models^{h_1}_{\Gamma_{\eta\epsilon}(z')}\ w_{z'}$ implies $s[z := v_1](z')\ \models^{h_1}_{\Gamma_{\eta\epsilon}(z')}\ w_{z'}$ and thus $\mathit{Valid}_{\mathsf{val}}(s[z := v_1](z'), \Gamma_{\eta\epsilon}(z'), h_1)$. Hence, $\mathit{Valid}_{\mathsf{store}}(FV(e_2), \Gamma_{\eta\epsilon} \cup \{z : \tau'_{\eta\epsilon}\}, s[z := v_1], h_1)$. Therefore, $\mathit{Valid}_{\mathsf{val}}(v, \tau_{\eta\epsilon}, h')$.

OSMatch-Nil: In this case $e = \mathsf{match}\ l\ \mathsf{with}\ \mid \mathsf{nil} \Rightarrow e_1 \mid \mathsf{cons}(hd, tl) \Rightarrow e_2$ for some l, hd, tl, e_1, and e_2. The typing context has the form $\Gamma = \Gamma' \cup \{l : \mathsf{L}_p(\tau')\}$ for some Γ', τ', p. The operational-semantics derivation gives $s(l) = \mathtt{NULL}$, hence validity for $s(l)$ gives $l : \mathsf{L}_0(\tau')$ and thus $\epsilon(p) = 0$. From the typing derivation for $D;\ \Gamma\ \vdash_\Sigma e : \tau$ we then know that $p = 0$, $D;\ \Gamma'\ \vdash_\Sigma e_1 : \tau$. Applying the induction hypothesis, with $p = 0 \wedge D$ then yields $\mathit{Valid}_{\mathsf{store}}(FV(e_1), \Gamma'_{\eta\epsilon}, s, h) \implies \mathit{Valid}_{\mathsf{val}}(v, \tau_{\eta\epsilon}, h')$. From $FV(e_1) \subseteq FV(e)$, $\mathit{Valid}_{\mathsf{store}}(FV(e), \Gamma_{\eta\epsilon}, s, h)$, $\epsilon(p) = 0 \wedge D_\epsilon$ and lemma 6.7 it follows that $\mathit{Valid}_{\mathsf{val}}(v, \tau_{\eta\epsilon}, h')$.

OSMatch-Cons: In this case $e = \mathsf{match}\ l\ \mathsf{with}\ \mid \mathsf{nil} \Rightarrow e_1 \mid \mathsf{cons}(hd, tl) \Rightarrow e_2$ for some l, hd, tl, e_1, e_2. The typing context has the form $\Gamma = \Gamma' \cup \{l : \mathsf{L}_p(\tau')\}$ for some Γ', τ', p. From the operational semantics we know that $h.s(l).\mathtt{hd} = v_{\mathtt{hd}}$ and $h.s(l).v_{\mathtt{tl}}$ for some $v_{\mathtt{hd}}$ and $v_{\mathtt{tl}}$ – that is $s(l) \neq \mathtt{NULL}$ – hence, due to validity of $s(l)$, we have $l : \mathsf{L}_p(\tau')$ for some τ' and $\epsilon(p) \geq 1$. From the typing derivation of e we obtain that $D;\ \Gamma',\ l : \mathsf{L}_p(\tau'),\ hd : \tau',\ tl : \mathsf{L}_{p-1}(\tau')\ \vdash_\Sigma e_2 : \tau$ Applying the induction hypothesis yields

$$\mathit{Valid}_{\mathsf{store}}(FV(e_2), \left\{ \begin{array}{l} \Gamma'_{\eta\epsilon} \cup \\ \cup \{l : (\mathsf{L}_p(\tau'))_{\eta\epsilon}\} \cup \\ \cup \{hd : \tau'_{\eta\epsilon}\} \cup \\ \cup \{tl : \mathsf{L}_e(\tau')\}_{\eta\epsilon}\} \end{array} \right\}, s \left[\begin{array}{l} hd := v_{\mathtt{hd}}, \\ tl := v_{\mathtt{tl}} \end{array} \right], h) \implies$$
$$\implies \mathit{Valid}_{\mathsf{val}}(v, \tau_{\eta\epsilon}, h').$$

Show that the l.h.s. of the implication holds. From $\mathit{Valid}_{\mathsf{store}}(FV(e), \Gamma_{\eta\epsilon}, s, h)$, $(FV(e_2) \setminus \{hd,\ tl\}) \subseteq FV(e)$, and lemma 6.7 we obtain

$$\mathit{Valid}_{\mathsf{store}}(FV(e_2) \setminus \{hd,\ tl\}, \Gamma_{\eta\epsilon}, s, h)$$

Due to $hd, tl \notin dom(s)$ we can apply lemma 6.6 and get

$$Valid_{\mathsf{store}}(FV(e_2) \setminus \{hd,\ tl\}, \Gamma_{\epsilon}, s[hd := v_{\mathtt{hd}}, tl := v_{\mathtt{tl}}], h)$$

From the validity $s(l) \models^{h}_{(\mathsf{L}_p(\tau'))_{\eta\epsilon}} w_{\mathtt{hd}} :: w_{\mathtt{tl}}$, and obvious $\epsilon(p-1) = \epsilon(p) - 1$ the validity of $v_{\mathtt{hd}}$ and $v_{\mathtt{tl}}$ follows: $v_{\mathtt{hd}} \models^{h}_{\tau'_{\eta\epsilon}} w_{\mathtt{hd}}$, $v_{\mathtt{tl}} \models^{h}_{(\mathsf{L}_{p-1}(\tau'))_{\eta\epsilon}} w_{\mathtt{tl}}$.

Now $Valid_{\mathsf{store}}(FV(e_2), \Gamma_{\eta\epsilon} \cup \{hd : \tau', tl : \mathsf{L}_{p-1}(\tau')\}_{\eta\epsilon}, s[hd := v_{\mathtt{hd}},\ tl := v_{\mathtt{tl}}], h)$ and, hence,

$$Valid_{\mathsf{val}}(v, \tau_{\eta\epsilon}, h').$$

OSFunApp: We want to apply the induction assumption to

$$[z'_1 := v_1, \ldots, z'_n := v_n];\ h;\ \mathcal{C}\ \vdash\ e_f\ \rightsquigarrow v;\ h'.$$

Let $\Sigma(f) = \tau^{\circ}_1 \times \ldots \times \tau^{\circ}_n \to \tau'$, the types τ°_i of the formal parameters be $\mathsf{L}_{n_{i1}}(\ldots \mathsf{L}_{n_{ik_i}}(\alpha_i)\ldots)$ respectively, and the types $\Gamma(z_i)$ of the actual parameters z_i be $\mathsf{L}_{p_{i1}}(\ldots \mathsf{L}_{p_{ik_i}}(\tau_{\alpha_i})\ldots)$, where $1 \leq i \leq n$. According to the typing rule $D \vdash \tau = \tau'[\ldots \alpha_i := \tau_{\alpha_i} \ldots]\,[\ldots n_{ij} := p_{ij} \ldots]$.

Since all called in e functions are defined via letfun, there must be a node in the derivation tree with True, $z'_1 : \tau^{\circ}_1, \ldots, z'_n : \tau^{\circ}_n \vdash_{\Sigma} e_f : \tau'$.

We take η' and ϵ', such that

- $\eta'(\alpha_i) = \eta(\tau_{\alpha_i})$,
- $\epsilon'(n_{ij}) = \epsilon(p_{ij})$.

Thus, $\Gamma(z_i)_{\eta\epsilon} = (\tau^{\circ}_i)_{\eta'\epsilon'}$, since

$$(\tau^{\circ}_i)_{\eta'\epsilon'} = \mathsf{L}_{\epsilon'(n_{i1})}(\ldots \mathsf{L}_{\epsilon'(n_{ik_i})}(\eta'(\alpha_i))\ldots) = \mathsf{L}_{\epsilon(p_{i1})}(\ldots \mathsf{L}_{\epsilon(p_{ik_i})}(\eta(\tau_{\alpha_i})\ldots) = (\Gamma(z_i))_{\eta\epsilon}$$

True ("no conditions") holds trivially on ϵ'. From the induction assumption we have

$$Valid_{\mathsf{store}}((z'_1, \ldots\ z'_n), (z'_1 : \tau^{\circ}_{1\,\eta'\epsilon'}, \ldots,\ z'_n : \tau^{\circ}_{n,\ \eta'\epsilon'}), [z'_1 := v_1, \ldots, z'_n := v_n], h)$$
$$\implies\ Valid_{\mathsf{val}}(v, \tau'_{\eta'\epsilon'}, h')$$

Show that the l.h.s. holds. From $Valid_{\mathsf{store}}(FV(e), \Gamma_{\eta\epsilon}, s, h)$ we have validity of the values of the actual parameters: $v_i \models^{h}_{\Gamma_{\eta\epsilon}(z_i)} w_i$ for some w_i, where $1 \leq i \leq k$. Since $\Gamma_{\eta\epsilon}(z_i) = (\tau^{\circ}_i)_{\eta'\epsilon'}$, the left-hand side of the implication holds, and one obtains $Valid_{\mathsf{val}}(v, \tau'_{\eta'\epsilon'}, h')$.

Now, D_{ϵ} implies $\tau_{\eta\epsilon} = \tau'[\ldots \alpha_i := \tau_{\alpha_i} \ldots][\ldots n_{ij} := p_{ij} \ldots]_{\eta\epsilon}$. Then from the construction for η' and ϵ' it follows $\tau'[\ldots \alpha_i := \tau_{\alpha_i} \ldots][\ldots n_{ij} := p_{ij} \ldots]_{\eta\epsilon} = \tau'[\ldots \alpha_i := \eta(\tau_{\alpha_i}) \ldots][\ldots n_{ij} := \epsilon(p_{ij}) \ldots] = \tau'_{\eta'\epsilon'}$

Thus, we have $Valid_{\mathsf{val}}(v, \tau_{\eta\epsilon}, h')$.

□

3.5. **Completeness of the type system.** Recall, that the system we consider is constituted from zero- and first-order types, typing rules, and Peano arithmetic extended to rationals.

The system is not complete in the class of shapely function definitions: there are shapely functions for which shapeliness may not be proved by means of the typing rules and the arithmetic. In other words, their annotated type cannot be checked by the system. For instance consider the following expression e:

$$\begin{array}{l} \mathsf{let}\, l = f(z_1,\ldots,\, z_k)\ \mathsf{in} \\ \qquad \mathsf{let}\ x = \mathsf{length}(l)\ \mathsf{in\ if}\ x\ \mathsf{then\ cons}(1,\mathsf{nil})\ \mathsf{else\ nil} \end{array}$$

where $\mathsf{length}(x)$ returns the length of list x. Let $p_f(n_1,\ldots,\, n_k)$ denote the polynomial size dependency for the shapely function definition f. If f never outputs an empty list, then the expression e defines a shapely function, with a polynomial size dependency $p(n_1,\ldots,\, n_k) = 1$. Otherwise $p(n_1,\ldots,\, n_k) = 0$ when f outputs nil. Suppose, there exists a procedure, that for any instantiation of the expression with f, produces its shapely type, when it is shapely, or rejects it otherwise. Then this procedure is capable to solve *10th Hilbert problem*: whether there exists a general procedure that given a polynomial with integer coefficients decides if this polynomial has natural roots or not.[3] Matiyasevich [Mat91] has shown that such a procedure does not exist. A similar problem is connected with match-construct.

We study constructions like above in more detail in section 4.2, devoted to decidability of type-checking. In particular, in lemma 4.1 we show, that for any integer polynomial q there is a shapely function definition f such that its size polynomial $p_f(n_1,\ldots,n_k)$ is equal to $q^2(n_1,\ldots,n_k)$ and thus p_f has roots if and only if q has roots.

In fact, this example shows that not only our system, but any system using integer arithmetic, is not complete in the class of shapely function definitions.

4. Type Checking

Because for every syntactic construction there is only one typing rule that is applicable, type checking is straightforward. The procedure parses a given function body and reduces to proving equations for rational polynomials. Consider some examples.

4.1. Examples.

4.1.1. *Cartesian product.* In the introduction, the Cartesian product was implemented using a "sugared" syntax. Here, we present the cprod function in the language defined in section 2.

$$\begin{array}{lll} \mathsf{letfun\ cprod}(l_1,\, l_2) = \mathsf{match}\ l_1\ \mathsf{with} & \mid \mathsf{nil} \Rightarrow \mathsf{nil} & \\ & \mid \mathsf{cons}(hd,\, tl) \Rightarrow & \mathsf{let}\ l' \quad = \mathsf{pairs}(hd,\, l_2) \\ & & \mathsf{in\ let}\ l'' = \mathsf{cprod}(tl, y) \\ & & \mathsf{in\ append}(l',\, l'') \\ \mathsf{in}\ \ldots & & \end{array}$$

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

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

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

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

PROVE: $l_1 : \mathsf{L}_n(\alpha), l_2 : \mathsf{L}_m(\alpha) \ \vdash_\Sigma e_{\mathsf{cprod}} : \mathsf{L}_{n*m}(\mathsf{L}_2(\alpha))$,

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:

$$\begin{array}{ll}
\text{NIL:} & n = 0;\ l_2 : \mathsf{L}_m(\alpha) \ \vdash_\Sigma \mathsf{nil} : \mathsf{L}_{n*m}(\mathsf{L}_2(\alpha)) \\
\text{CONS:} & hd : \alpha,\ l_1 : \mathsf{L}_n(\alpha),\ tl : \mathsf{L}_{n-1}(\alpha),\ l_2 : \mathsf{L}_m(\alpha) \ \vdash_\Sigma \\
& \qquad\qquad\qquad \left.\begin{array}{l} \mathsf{let}\ l' \quad = \mathsf{pairs}(hd, l_2) \\ \mathsf{in\ let}\ l'' = \mathsf{cprod}(tl, l_2) \\ \mathsf{in\ append}(l',\ l'') \end{array}\right\} : \mathsf{L}_{n*m}(\mathsf{L}_2(\alpha))
\end{array}$$

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:

$$\begin{array}{ll}
\text{BIND-L':} & hd : \alpha,\ l_2 : \mathsf{L}_m(\alpha) \ \vdash_\Sigma \mathsf{pairs}(hd,\ l_2) : \tau_1 \\
\text{BIND-L'':} & tl : \mathsf{L}_{n-1}(\alpha),\ l_2 : \mathsf{L}_m(\alpha) \ \vdash_\Sigma \mathsf{cprod}(tl,\ l_2) : \tau_2 \\
\text{BODY:} & l' : \tau_1, l'' : \tau_2 \ \vdash_\Sigma \mathsf{append}(l', l'') : \mathsf{L}_{n*m}(\alpha)
\end{array}$$

From the applications of the FUNAPP-rule to BIND-L' and BIND-L" 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 $\vdash n * m = m + (n-1) * m$, which is true in the axiomatics.

4.1.2. *Example with negative coefficients.* In contrast to the system presented by Vasconcelos and Hammond [VasHam03], where only subtraction of constants are allowed, our system allows negative coefficients in size expressions. Of course, this is only a valid size expression (yielded by a total function) if the polynomial maps naturals into naturals. 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 remaining list with itself is returned:

$$\begin{array}{l}
\mathsf{sqdiff}\ (l_1,\ l_2) = \\
\quad \mathsf{match}\ l_1\ \mathsf{with}\ \begin{array}{l} \mid \mathsf{nil} \Rightarrow \mathsf{cprod}(l_2,\ l_2) \\ \mid \mathsf{cons}(hd, tl) \Rightarrow \mathsf{match}\ l_2\ \mathsf{with}\ \begin{array}{l} \mid \mathsf{nil} \Rightarrow \mathsf{cprod}(l_1,\ l_1) \\ \mid \mathsf{cons}(hd', tl') \Rightarrow \mathsf{sqdiff}\ (tl,\ tl') \end{array} \end{array} .
\end{array}$$

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

4.2. Type checking in general is undecidable (even for total function definitions).

In the examples above, type checking ends up with a set of entailments like $n = 0 \vdash n*m = 0$ or $\vdash n*m = m+m*(n-1)$ that have to hold. However, we show that there is no procedure to check all possible entailments that may 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 condition is as follows: *given a function body, allow pattern-matching only on the function parameters or variables bound to them by other pattern-matchings.* Thus, we prohibit expressions like

$$\mathsf{let}\ l = f_0(x_1,\ \ldots,\ x_k)\ \mathsf{in}\ \ \mathsf{match}\ l\ \mathsf{with}\ \begin{array}{l} \mid \mathsf{nil} \Rightarrow e_1 \\ \mid \mathsf{cons}(hd, tl) \Rightarrow e_2 \end{array}$$

Pattern-matching like

```
match $l$ with | nil $\Rightarrow e_1$
               | cons$(hd, tl) \Rightarrow$  match $tl$ with | nil $\Rightarrow e_1'$
                                                  | cons$(hd', tl') \Rightarrow e_2$
```

is allowed. Below we explain the reason for this restriction.

We show that the existence of a procedure that checks all possible entailments at the end of type checking is reduced to Hilbert's tenth problem. Type checking is reducible to a procedure for 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 $l_1, \ldots, l_k$:

```
let $l = f_0(l_1, \ldots, l_k)$ in  match $l$ with | nil $\Rightarrow f_1(l_1, \ldots, l_k)$
                                       | cons$(hd, tl) \Rightarrow f_2(l_1, \ldots, l_k)$
```

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 decide if it has natural roots or not.

Checking if any size polynomial has roots in natural numbers, is as difficult as checking whether an arbitrary polynomial has roots or not. First, we prove the following lemma.

Lemma 4.1. *For any polynomial q there is a total shapely function definition f such that its size dependency $p_f(n_1, \ldots, n_k)$ is equal to $q^2(n_1, \ldots, n_k)$.*

Proof. First, note that any polynomial q may be presented as the difference $q_1 - q_2$ of two polynomials with non-negative coefficients[4]. 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 append and $\mathsf{copyfirst}$: $\mathsf{L}_n(\alpha) \times \mathsf{L}_m(\alpha) \to \mathsf{L}_{n*m}(\alpha)$ (see subsection 5.1) functions. □

Summing up the constructions above we obtain the following statement:

Lemma 4.2. *If there exists a type-checker that for any function definition and its type annotation is able to accept or reject the annotated type correctly, then there exists a procedure that for any integer polynomial $q(n_1, \ldots, n_k)$ decides if it has natural roots or not.*

Proof. Suppose that such type checker exists. Consider the expression e_H above with f_0, f_1, f_2 defined as follows. Using lemma 4.1, construct a function definition f_0 that has a size dependency $q^2(n_1, \ldots, n_k)$. Now let f_1 be defined by the expression nil and let f_2 be defined by $\mathsf{cons}(1, \mathsf{nil})$.

The type checker accepts e_H with the type annotation $p \equiv 1$ if and only if the nil-branch is not entered, that is if and only if $q^2(n_1, \ldots, n_k)$ has no roots. Trivially, $q^2(n_1, \ldots, n_k)$ has roots if and only if $q(n_1, \ldots, n_k)$ does. □

[4] 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 $q_2 = \Sigma_{a_{i_1,\ldots,i_k} < 0} |a_{i_1,\ldots,i_k}| x_1^{i_1} \ldots x_k^{i_k}$.

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 a program 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 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.

An anonymous reviewer pointed out that the construction from lemma 4.1 demonstrates a problem with real arithmetic, when it is used to check numerical entailments, generated by the type checker. Suppose we want to omit the syntactic restriction and type check the expression e_H where the size dependency for f_0 is $p_0(n) = (n^2 - 2)^2$. A real-arithmetic-based version of the checker rejects e_H, since there is a real root for p_0 and in this abstract interpretation the nil-branch with $1 = 0$ must be considered. In fact, the expression is well-typed with annotation $p \equiv 1$, since there is no natural roots for p_0 and the nil-branch is never entered.

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'. If 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 $1 \leq i \leq m$, then the expressions for n_i can be 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 rational field. Recall that two polynomials are equal if and only if the coefficient at monomials with the same degrees of variables are equal.

4.3. **Syntactical condition for decidability.** The simplest 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 the syntactical condition for program expressions, *prohibiting pattern matching on variables other than function parameters and bounded to them by other pattern matchings.*

It is easy to see that any function body that satisfies the syntactic condition may be encoded in the language defined by the *refined grammar* where the let-construct in e is replaced by $\mathsf{let}\ x = b\ \mathsf{in}\ e_{nomatch}$:

$$\begin{array}{lcll}
\textit{Basic} & b & ::= & c \mid x\,\mathsf{binop}\,y \mid \mathsf{nil} \mid \mathsf{cons}(z, l) \mid f(z_1, \ldots, z_n) \\
\textit{Expr} & e & ::= & b \\
 & & & \mid \mathsf{let}\ z = b\ \mathsf{in}\ e_{nomatch} \\
 & & & \mid \mathsf{if}\ x\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 \\
 & & & \mid \mathsf{match}\ l\ \mathsf{with}\ \mid \mathsf{nil} \Rightarrow e_1 \\
 & & & \qquad\qquad\qquad \mid \mathsf{cons}(hd, tl) \Rightarrow e_2 \\
 & & & \mid \mathsf{letfun}\ f(z_1, \ldots, z_n) = e_1\ \mathsf{in}\ e_2 \\
 & & & \mid \mathsf{letextern}\ f(z_1, \ldots, z_n)\ \mathsf{in}\ e_1
\end{array}$$

with

$$\begin{array}{rcl} e_{nomatch} & := & b \\ & & \mid \mathsf{let}\ z = b\ \mathsf{in}\ e'_{nomatch} \\ & & \mid \mathsf{if}\ x\ \mathsf{then}\ e'_{nomatch}\ \mathsf{else}\ e''_{nomatch} \\ & & \mid \mathsf{letfun}\ f(z_1, \ldots, z_n) = e\ \mathsf{in}\ e'_{nomatch} \\ & & \mid \mathsf{letextern}\ f(z_1, \ldots, z_n)\ \mathsf{in}\ e'_{nomatch} \end{array}$$

The grammar is more restrictive than the syntactic condition. However, any function body that satisfies the condition may be encoded in this grammar. For instance, an expression

$$\begin{array}{l} \mathsf{let}\ l' = f_0(z)\ \mathsf{in}\ \ \mathsf{match}\ l\ \mathsf{with} \mid \mathsf{nil} \Rightarrow f_1(l, l') \\ \qquad\qquad\qquad\qquad\qquad\qquad \mid \mathsf{cons}(hd, tl) \Rightarrow f_2(l,\ l') \end{array}$$

and the expression

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

define the same map of lists.

For this reason we call the refined grammar the "no-let-before-match" grammar, and roughly refer to the syntactic conditions as to the "no-let-before-match" condition. The demo version of the type checker, accessible from `www.aha.cs.ru.nl`, uses the "no-let-before-match" grammar.

Theorem 4.3. *Let a program expression e satisfy the refined grammar, and let us check the judgement* $\mathsf{True};\ x_1 : \tau^o_1, \ldots, x_k : \tau^o_k\ \vdash_\Sigma e : \tau$. *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 $n \in FVS(\tau^o_1 \times \ldots \times \tau^o_k)$ and constant c and p, p' are polynomials in $FVS(\tau^o_1 \times \ldots \times \tau^o_k)$.

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 $q = 0$ belongs to D. It means that in the path there is the nil-branch of the pattern matching for some $l : \mathsf{L}_q(\tau)$.

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 follows from the syntactic condition: the program variables which are not free in a program expression and pattern-matched may be introduced only by another pattern-matching, but not a let-binding. The technical report [ShvKvE07a] contains the full proof.

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 constants and substituted into $p_1 = p$. Another case when it is easy to check if there are natural roots for $p_0 = 0$ or not (and find them if "yes") is when p_0 is a 1-variable polynomial. We leave relaxations of the condition for future work.

5. 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. The procedure presented in this section was sketched by us in [ShvKvE07b] and given in details and evaluated with a series of measurements in [vKShvE07].

For each size dependency from the output type of a given function definition one assumes that it is a polynomial and one guesses its degree. Then, to obtain the coefficients of the polynomial of this degree, the function definition is evaluated (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 input data satisfy some criteria known from polynomial interpolation theory [Chui87, Lor92] (see the subsections below for more detail), the system has a unique solution. Input sizes that satisfy these criteria, which are nontrivial for multivariate polynomials, can be determined algorithmically.

In this way we find using interpolation theory the interpolating polynomial for the size dependency. If the size dependency is a polynomial function and the hypothesis about its degree is correct, then it coincides with its interpolating polynomial. To check if this is the case, the interpolating polynomial is given to the type checking procedure. If it passes, it is correct. Otherwise, one repeats the procedure for a higher degree of the size dependency. Starting with degree zero[5], the method iteratively constructs the interpolating polynomials until the correct polynomial is found. It does not terminate when

(1) the function under consideration does not terminate on test data,
(2) the function is non-shapely,
(3) the function is shapely but the type-checker rejects it due to the type-system's incompleteness (see section 3.5).

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 underlying type $\mathsf{L}(\alpha) \times \mathsf{L}(\alpha) \longrightarrow \mathsf{L}(\mathsf{L}(\alpha))$. 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:

$$p_1(n,m) = a_{0,0} + a_{0,1}n + a_{1,0}m + a_{1,1}nm + a_{0,2}n^2 + a_{2,0}m^2$$

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

n	m	l_1	l_2	$\mathsf{cprod}(l_1,l_2)$	$p_1(n,m)$	$p_2(n,m)$
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

The first three rows of the table are examples of *incomplete measurements*, where the size of the inner list is unknown, because the outer list is empty. The last three rows are *complete measurements*.

[5] On can also start with a higher degree. If the degree of the solution happens to be lower than the initial degree, the solution will still be found since the found coefficients will be zero at the right places.

The test table defines the following linear system for the outer 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 4.1.

As an alternative way of finding the coefficients, one could try to solve directly the (recurrence) equations defined by entailments $D \vdash p = p'$ that arise during construction of the type-inference tree for a function definition. As we will see in subsection 5.1, it amounts to solving systems that are nonlinear in general. By combining testing with type checking we bypass nonlinear systems [vKShvE07].

However, test-based inference has a drawback: it is not fully static. The procedure has dynamic aspects, since it is done not only in the underlying logic of the type system (i.e. Peano arithmetic), but it involves executing the interpreter of the programming language. A consequence of it may be that inference for function definitions with external calls is based on the semantics of another language. When the size dependency of the external function is known, this can be avoided by

- modifying the interpreter of our language in such a way, that in the case of an external call it creates a "fake" object of the right size (the size of the result of "this" external call), or
- leaving the interpreter in intact, and creating for any external function from its sized type a "fake" function body in our language with the same size dependency as the external function.

From an engineering point of view, the advantage of the second approach is that a standard interpreter can be used directly. We discuss the mechanism of generating "fake" functions in 5.8.

Ideally, one would like to remove all dynamic aspects from type inference. In our current research towards fully static inference we consider a modification of the method where instead of the interpreter of the programming language one uses an abstract interpreter in the form of a term-rewriting system of which the rewriting rules will correspond to equations in Peano arithmetic. For instance, progression is interpreted as $p(n) \rightarrow n + p(n-1)$ together with $p(0) \rightarrow 0$. We have presented preliminary results in the technical report [ShvE0T8].

5.1. **Motivation for test-based inference.**

Consider, as an example of the complexity of systems generated by conventional type inference, the system for a function definition nonlinear with auxiliary functions:

$$\begin{array}{ll} \mathsf{copy:} & \mathsf{L}_n(\alpha) \to \mathsf{L}_n(\alpha) \\ \mathsf{copyfirst:} & \mathsf{L}_{n_1}(\alpha) \times \mathsf{L}_{n_2}(\alpha) \to \mathsf{L}_{n_1 * n_2}(\alpha) \\ \mathsf{sqdiffaux:} & \mathsf{L}_{n_1}(\alpha) \times \mathsf{L}_{n_2}(\alpha) \to \mathsf{L}_{n_1^2 + n_2^2 - 2*n_1*n_2}(\alpha) \end{array}$$

where (in the sugared syntax[6])

```
letfun copy(l) = match l with | nil ⇒ nil
                              | cons(hd, tl) ⇒ cons(hd, copy(tl))
in letfun copyfirst(l1, l2) = match l2 with | nil ⇒ nil
                                            | cons(hd, tl) ⇒ l1 ++ copyfirst(l1, tl)
in letfun sqdiffaux(l1, l2) =  match l1 with | nil ⇒ copyfirst(l2, l2)
                                             | cons(hd, tl) ⇒
                                 match l2 with | nil ⇒ copyfirst(l1, l1)
                                               | cons(hd′, tl′) ⇒ sqdiffaux(tl, tl′)
in letfun nonlinear(l1, l2) =  match l1 with | nil ⇒ copyfirst(copyfirst(l2, l2), [1 . . . 4])
                                             | cons(hd, tl) ⇒
                               match l2 with | nil ⇒ copyfirst(copyfirst(l1, l1), [1 . . . 4])
                                             | cons(hd′, tl′) ⇒
                               sqdiffaux(nonlinear(tl, l2) ++ l1, nonlinear(l1, tl′) ++ l2)
                                  ++ copyfirst(copyfirst(l1, l2), [1 . . . 17])
in . . .
```

The inference procedure ends up with the following recurrence system:

$$\left\{\begin{array}{lcl} p(0,n_2) & = & 4n_2^2 \\ p(n_1,0) & = & 4n_1^2 \\ p(n_1,n_2) & = & (p(n_1-1,n_2)+n_1-(p(n_1,n_2-1)+n_2))^2+17n_1n_2 \end{array}\right. \tag{1}$$

The problem is *to find* p, assuming, say, that it is quadratic.

A standard way of solving this problem uses the method of unknown coefficients. A polynomial to find, $p(n_1,n_2)$, is presented in the form $a_{0,0}+a_{0,1}n_1+a_{1,0}n_2+a_{1,1}n_1n_2+a_{0,2}n_1^2+a_{2,0}n_2^2$ and substituted into (1). Equating the corresponding coefficients of the polynomials from the left and right sides of the equations from (1) gives

$$\left\{\begin{array}{lcl} a_{0,0} & = & 0,\;\; a_{1,0}=0,\;\; a_{2,0}=4,\;\; a_{0,1}=0,\;\; a_{0,2}=4 \\ a_{0,2} & = & (a_{1,1}-2a_{0,2}+1)^2 \\ a_{2,0} & = & (2a_{2,0}-a_{1,1}-1)^2 \\ a_{1,1} & = & 2(a_{1,1}-2a_{0,2}+1)(2a_{2,0}-a_{1,1}-1)+17 \\ a_{0,1} & = & 2((a_{1,0}-a_{0,1})+(a_{0,2}-a_{2,0}))(a_{1,1}-2a_{0,2}+1) \\ a_{1,0} & = & 2((a_{1,0}-a_{0,1})+(a_{0,2}-a_{2,0}))(2a_{2,0}-a_{1,1}-1) \\ a_{0,0} & = & ((a_{1,0}-a_{0,1})+(a_{0,2}-a_{2,0}))^2 \end{array}\right.$$

Substituting the coefficients $a_{0,0}=0,\; a_{1,0}=0,\; a_{2,0}=4,\; a_{0,1}=0,\; a_{0,2}=4$ in the remaining equations one obtains the non-linear system

$$\left\{\begin{array}{lcl} a_{1,1}^2-14a_{1,1}+45 & = & 0 \\ 2a_{1,1}^2-27a_{1,1}+81 & = & 0 \end{array}\right.$$

The solution of this quadratic system can be found easily. It is $a_{1,1}=9$.

[6] Recall, that in the sugared syntax we use $f(g(z))$ for "let $z'=g(z)$ in $f(z')$" and, moreover, use $[1\ldots c]$ for c-ary application of cons$(-,-)$ to nil, so that $[1\ldots 3]$ denotes cons(1, cons(2, cons(3, nil))). We also use the infix ++ for append.

In general, non-linear systems may be hard to solve. With the testing approach we avoid solving *nonlinear* systems w.r.t. polynomial coefficients a_{ij}. Instead, we compute the coefficients solving the *linear* system that is generated after testing.

5.2. **Interpolating a polynomial.** A hypothesis for a type is derived automatically by fitting a polynomial to the size data, as it was shown in the example cprod. We are looking for the polynomial that best approaches the data, i.e., the polynomial interpolation. The polynomial interpolation exists and is unique under some conditions on the data, which are explored in polynomial interpolation theory [Chui87, Lor92].

For 1-variable interpolation this condition is well-known. A polynomial $p(z)$ of degree d with coefficients $a_1, \dots, a_{d+1}$ can be written as follows:

$$a_1 \ + \ a_2\, z \ + \ \dots \ + \ a_{d+1}\, z^d = \ p(z)$$

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

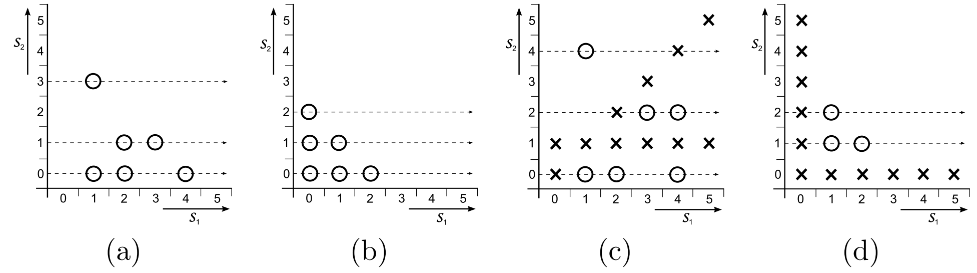
$$\begin{pmatrix} 1 & z_1 & \cdots & z_1^{d-1} & z_1^d \\ 1 & z_2 & \cdots & z_2^{d-1} & z_2^d \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 1 & z_d & \cdots & z_d^{d-1} & z_d^d \\ 1 & z_{d+1} & \cdots & z_{d+1}^{d-1} & z_{d+1}^d \end{pmatrix} \begin{pmatrix} a_1 \\ a_2 \\ \vdots \\ a_d \\ a_{d+1} \end{pmatrix} = \begin{pmatrix} p(z_1) \\ p(z_2) \\ \vdots \\ p(z_d) \\ p(z_{d+1}) \end{pmatrix}$$

The determinant of the left matrix, contains the measurement points, is called a *Vandermonde* determinant. For pairwise different points $z_1, \dots, \ z_{d+1}$ it is non-zero. This means that, as long as the output size is measured for $d+1$ different input sizes, there exists a unique solution for the system of equations and, thus, a unique interpolating polynomial.

The condition under which there exists a unique polynomial that interpolates *multivariate* data is not trivial. We formulate it in the next subsection. Here we introduce the necessary definitions.

Recall that a polynomial of degree d and dimension k (the number of variables) has $N_d^k = \binom{d+k}{k}$ coefficients. Let a set of values f_i of a real function f be given. A set $W = \{\bar{w}_i : i = 1, \dots, \ N_d^k\}$ of points in a real k-dimensional space forms the set of *interpolation nodes* if there is a unique polynomial $p(\bar{z}) = \Sigma_{0 \leq |j| \leq d} a_j \bar{z}^j$ with the total degree d with the property $p(\bar{w}_i) = f_i$, where $1 \leq i \leq N_d^k$. In this case one says that the polynomial p interpolates the function f at the nodes $\bar{w}_i$.

The condition on W, which assures the existence and uniqueness of an interpolating polynomial, is geometrical: it describes a configuration, called **NCA** [Chui87], in which the points from W should be placed in $\mathcal{R}^k$. The multivariate Vandermonde determinant computed from such points is non-zero. Thus, the corresponding system of linear equations w.r.t. the polynomial's coefficients has a unique solution. In the following subsections we show how to generate a collection of *natural-valued* nodes $\bar{w}_i$ in an **NCA** configuration. A Vandermonde determinant is computed by the same formula in reals and naturals, so the system of linear equations based on natural nodes will have a unique (rational) solution.



(a) (b) (c) (d)

Figure 1: (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) Incomplete measurements complicate finding a node configuration (d) Incomplete measurements for the pairs in the output of cprod.

5.3. **Measuring bivariate polynomials.** For a two-dimensional polynomial of degree d, the condition on the nodes that guarantees a unique polynomial interpolation is as follows [Chui87]:

Definition 5.1. N_d^2 nodes forming a set $W \subset \mathcal{R}^2$ lie in a *2-dimensional NCA configuration* if there exist lines $\gamma_1, \ldots, \gamma_{d+1}$ in the space $\mathcal{R}^2$, such that $d+1$ nodes of W lie on γ_{d+1} and d nodes of W lie on $\gamma_d \setminus \gamma_{d+1}$, ..., and finally 1 node of W lies on $\gamma_1 \setminus (\gamma_2 \cup \ldots \cup \gamma_{d+1})$.

An example of such a configuration for integers is given in figure 1a.

Nodes satisfying this condition can be found automatically: if the output type of a given function definition is $\mathsf{L}_{p_1}(\ldots \mathsf{L}_{p_s}(\alpha)\ldots)$, then for the outermost-list size p_1 choose a triangle of nodes on parallel lines, like in figure 1b.

An example of the two dimensional case is the cprod function above. As we have seen, the procedure of reconstructing the size polynomial p_1 for the outer list is straightforward. However, there is a problem for p_2. There are cases in which nodes have no corresponding output size (the question-marks in the table that refer to incomplete measurements). Measurements for p_2 may be incomplete, 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 for complete measurements. As can be seen in figure 1d, for cprod output's outer list is empty when one of the two input lists is empty. In the next section, we show that, despite this, it is always possible to find enough measurements and give an upper bound on the number of natural nodes that have to be searched.

5.4. **Handling incomplete measurements.** In general, for $\mathsf{L}_{p_1}(\ldots \mathsf{L}_{p_s}(\alpha)\ldots)$ 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 using finite search.

First, nested output lists of which the size of the outer list is the constant zero, e.g. $\mathsf{L}_0(\mathsf{L}_{p_2}(\alpha'))$, need special treatment. If a type-checker rejects annotations for $p_1 \equiv 0$ and *arbitrary* p_2 then the outer polynomial p_1 is not a constant zero. (Recall the definition of $D \vdash \tau = \tau'$.)

Now, let the outer polynomial $p_1(x, y)$ be not a constant zero. Then there is a finite number of lines $y = i$, which we will call *root lines*, where $p_1(x,\, i) = 0$.

Lemma 5.2. *A polynomial $p_1(x,\, y)$ of degree d that is not constant* 0 *has at most d root lines $y = i$, such that $p_1(x,\, i) = 0$ for all x.*

Proof. Suppose there are more than d root lines. Then, it is easy to pick $1, \ldots, d+1$ nodes on $d+1$ root lines. They trivially are in **NCA** configuration. With these nodes, at which $p_1(x, y) = 0$, the system of linear equations for the coefficients of p_1 will have the zero-solution, that is, all the coefficients of p_1 will be zeros. This contradicts the assumption that p_1 is not constant 0. □

Using the lemma, we can bound the number of parallel lines $y = i$ and nodes on them that have to be searched. Essentially, we are to find a triangle configuration of nodes, like on figure 1b, skipping all crosses, see 1c.

Lemma 5.3. *When looking for nodes for a polynomial $p_2(x, y)$ that determine a unique polynomial interpolation at places where another polynomial $p_1(x, y) \neq 0$, it is sufficient to search the lines $y = 0, \ldots, y = d_1 + d_2$ in the square $[0, \ldots, d_1 + d_2] \times [0, \ldots, d_1 + d_2]$.*

Proof. For the configuration it is sufficient to have $d_2 + 1$ lines $y = i$ with at least $d_2 + 1$ points where $p_1(x, y) \neq 0$. Due to lemma 5.2 there are at most d_1 lines $y = i$ such that $p_1(x,\, i) = 0$, so at least $d_2 + 1$ are not root lines for p_1. The polynomial $p_1(x,\, j)$, with $y = j$ not a root line, has at most degree d_1, thus $y = j$ contains at most d_1 nodes $(x,\, j)$, such that $p_1(x,\, j) = 0$. Otherwise, it would have been constant zero, and thus a root line. Hence, this leaves at least $d_2 + 1$ points on these lines for which p_1 is not zero. □

This straightforwardly generalizes to all nested types $\mathsf{L}_{p_1}(\ldots \mathsf{L}_{p_s}(\alpha) \ldots)$ with polynomials in two variables. If we want to derive the coefficients of p_i, searching the square of input values $[0, \ldots, \Sigma_{j=1}^{i} d_j] \times [0, \ldots, \Sigma_{j=1}^{i} d_j]$ suffices, where d_j is the degree of p_j. Each p_j has at most d_j root lines, so there are at most $\Sigma_{j=1}^{i-1} d_j$ root lines for $p_1, \ldots, p_{i-1}$. Also, each of the p_j can have at most d_j zeros on a non root line. Hence, since the length of the search interval for p_i is $\Sigma_{j=1}^{i} d_j + 1$, there are always $d_i + 1$ values known.

Eventually, it is enough to search in $[0, \ldots, \Sigma_{j=1}^{s} d_j] \times [0, \ldots, \Sigma_{j=1}^{s} d_j]$.

For cprod there are two size expressions to derive, p_1 for the outer list and p_2 for the inner lists. Deriving that $p_1(n_1, n_2) = n_1 * n_2$ is no problem. Because p_1 has roots for $n_1 = 0$ and for $n_2 = 0$, these nodes should be skipped when measuring p_2 (see figure 1d).

5.5. **Generalizing to k-dimensional polynomials.** The generalization of the condition on nodes for a unique polynomial interpolation to polynomials in k 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 $k-1$ dimensional case. A hyperplane K_j^k may be viewed as a set in which test points for a polynomial of $k-1$ variables of the degree j lie. There must be $N_j^{k-1} = N_j^k - N_{j-1}^k$ such points. The condition on the nodes is defined by:

Definition 5.4. The *NCA configuration for k variables (k-dimensional space)* is defined inductively on k [Chui87]. Let $\{\bar{z}_1, \ldots,\, \bar{z}_{N_d^k}\}$ be a set of distinct points in $\mathcal{R}^k$ such that there exist $d+1$ hyperplanes K_j^k, $0 \leq j \leq d$ with

$$\begin{aligned} &\bar{z}_{N_{d-1}^k+1}, \ldots,\, \bar{z}_{N_d^k} \;\in\; K_d^k \\ &\bar{z}_{N_{j-1}^k+1}, \ldots,\, \bar{z}_{N_j^k} \;\in\; K_j^k \setminus \{K_{j+1}^k \cup \ldots \cup K_d^k\}, \text{for } \; 0 \leq j \leq d-1 \end{aligned}$$

and each of set of points $\bar{z}_{N^k_{j-1}+1}, \ldots, \bar{z}_{N^k_j}$, $0 \leq j \leq d$, considered as points in $\mathcal{R}^{k-1}$ satisfies **NCA** in $\mathcal{R}^{k-1}$.

For instance, given $d = 2$ and $k = 3$ (i.e. interpolating by polynomials of 3 variables of degree 2), the following collection of $N^3_2 = \binom{2+3}{3} = 10$ nodes, placed on parallel planes in $\mathcal{R}^3$, satisfies an **NCA** configuration:

(1) on the plane $x = 0$ take the "triangle" of $N^2_2 = 6$ points (0, 0, 0), (0, 0, 1), (0, 0, 2), (0, 1, 0), (0, 1, 1), (0, 2, 0),
(2) on the plane $x = 1$ take the "triangle" of $N^2_1 = 3$ points (1, 1, 0), (1, 0, 1), (1, 1, 1),
(3) on the plane $x = 2$ take the point (2, 0, 0).

Here the nodes on each of the planes lie in the 2-dimensional **NCA** configurations constructed for degrees 2, 1 and 0 respectively.

Similarly to lines in a square in the two-dimensional case, parallel hyperplanes in $\mathcal{R}^k$ have to be searched while generating hypothesis for a nested type. Using a reasoning similar to the two-dimensional case one can show that it is always sufficient to search a hypercube with sides $[0, \ldots, \Sigma^s_{j=1} d_j]$.

5.6. **Automatically inferring size-aware types: the procedure.** The type checking procedure and the size hypothesis generation can be combined to create an inference procedure. The procedure starts with assuming a fixed degree. The assumptions is that this degree is the maximum degree of all polynomials in the type. If checking rejects the hypothesis generated for this degree, the degree is increased and the test-check cycle is repeated. The procedure is semi-algorithmic: it terminates only when the function is well-typable.

Recently, we have developed a demonstrator for the inference procedure described in [vKShvE07]. It is accessible on `www.aha.cs.ru.nl`.

For any shapely program, the underlying type (the type without size annotations) can be derived by a standard type inference algorithm [Mil78]. After straightforwardly annotating input sizes with size variables and output sizes with size expression variables, we have for example

$$\mathsf{cprod} : \mathsf{L}_{n_1}(\alpha) \times \mathsf{L}_{n_2}(\alpha) \to \mathsf{L}_{p_1(n_1,n_2)}(\mathsf{L}_{p_2(n_1,n_2)}(\alpha))$$

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, from p_1 to p_s. 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 d.

The schema below shows the procedure in pseudo-code. The *TryIncreasingDegrees* function generates (by *GetSizeAwareType*) and checks (by *CheckSizeAwareType*) hypotheses. A size expression is derived by selecting a node configuration (*GetNodeConf*), running the tests for these nodes (*RunTests*), and deriving the size polynomial from the test results (*DerivePolynomial*).

```
Function: TryIncreasingDegrees
Input: a degree d, a function definition f
Output: the size-aware type of that function

TryIncreasingDegrees(d, f) =
    let type  = InferUnderlyingType(f)
        atype = AnnotateWithSizeVariables(type)
        vs    = GetOutputSizeVariables(atype)
        stype = GetSizeAwareType(d, f, atype, vs, [ ])
    in  if (CheckSizeAwareType(stype, f)) then stype
        else TryIncreasingDegrees(d+1, f)
```

```
 Function:   GetSizeAwareType
 Input:      a degree d,
             a function definition f,
             its annotated type,
             a list of unknown size annotations,
             and the polynomials already derived
 Output:     the size-aware type
             of that function if the degree is high enough
GetSizeAwareType(d, f, atype, [ ], ps) =
    AnnotateWithSizeExpressions(atype, ps) // The End
GetSizeAwareType(d, f, atype, v:vs, ps) =
   let nodes   = GetNodeConf(d, atype, ps)
       results = RunTests(f, nodes)
       p       = DerivePolynomial(d, v, atype, nodes, results)
   in  GetSizeAwareType(d, f, atype, vs, p:ps)
```

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 procedure continues for a higher degree. Another possibility is that one of the size expressions is not a polynomial (the function definition is not shapely) or that the type cannot be checked due to incompleteness of the type system. In that case the procedure will not terminate. If the function is well-typable, the procedure will eventually find the correct size-aware type and terminate.

A collection of examples – function definitions together with size measurements – is presented in [vKShvE07].

5.7. **Complexity of hypotheses-generating phase.** Given a function definition, its underlying first-order type and a maximal degree of hypothetical polynomials, the complexity of its hypothesis-generating phase depends on three parameters:

- the nestedness $s \geq 0$ of the output type which may be either $\mathsf{L}_{p_1}(\ldots \mathsf{L}_{p_s}(\mathtt{Int})\ldots)$ or $\mathsf{L}_{p_1}(\ldots \mathsf{L}_{p_s}(\alpha)\ldots)$,
- the fixed maximal degree d of the polynomials $p_1, \ldots, p_s$,
- the number of size variables k defined by the input type of the function.

To generate hypothesis for $p_1(n_1, \ldots, n_k)$ one

(1) generates $N_d^k = \binom{k+d}{k}$ natural-valued nodes inductively on k; it is done by the definition 5.4 of **NCA** configuration for the k-variable case (note that for $k = 1$ it is just the 1-dimensional nodes $0, \dots, d$).
(2) generates a collection of N_d^k concrete inputs with the sizes, defined by the nodes,
(3) evaluates the function body $N_d^k = \binom{k+d}{k}$ times on these inputs,
(4) solves the system of N_d^k linear equations to obtain N_d^k coefficients for p_1.

Generating hypotheses for a p_j, $j > 1$, is similar. However, generating the collection $N_d^k = \binom{k+d}{k}$ nodes is more complicated, since nodes sending some $p_{j'}$, $j' < j$, to zero are excluded. In the worst case, to find correct nodes, one needs to evaluate a k dimensional cube with side $[0, \dots, jd]$, that is to evaluate (to check if it has a zero value) $j-1$ polynomials in at most $(jd+1)^k$ nodes.

Thus, for each $1 \leq j \leq s$ the complexity is bounded by $c_{eval\, p_1,\dots,p_{j-1}} + c_{eval\, p_j} + c_{gauss}$, where

- $c_{eval\, p_1,\dots,p_{j-1}} = (j-1) \cdot (jd+1)^k$ evaluations of polynomials,
- $c_{eval\, p_j} = N_d^k = \binom{k+d}{k}$ evaluations of the function definition,
- $c_{gauss} = O({N_d^k}^2)$ is the complexity of Gaussian elimination.

If the results of evaluations of polynomials on the j-th step are memoised, then altogether for $j = 1, \dots s$ one needs at most $(s-1) \cdot (sd+1)^k$ evaluations of polynomials. Thus, the complexity of the hypotheses-generating phase for all $j = 1, \dots s$ together is $(s-1) \cdot (sd+1)^k + s \cdot \binom{k+d}{k} + s \cdot O(\binom{k+d}{k}^2)$.

5.8. **Inhabitants for the types of external functions.** Let $\mathsf{f}_{\mathsf{ext}}$ be an external function. Since the function is external, its code is not present in our language. However, its first-order type may be available. We have to trust this type since we cannot check it.

For inference of types of other functions that somewhere call $\mathsf{f}_{\mathsf{ext}}$, our testing procedure requires the possibility to evaluate within our language the code of the external function. Such code can be made available in our language by constructing an inhabitant of the type of $\mathsf{f}_{\mathsf{ext}}$.

For our demonstrator, an alternative solution would be to create an actual external call for each occurrence of an external function. This may require more implementation effort within the demonstrator. The type inference procedure might take more time because the external function may require more time to execute than the generated inhabitants of the type. Therefore, we prefer to work with inhabitants (which yields the same size dependencies as using external functions directly). For reasons of modularity it might even be worthwhile to also create inhabitants of internal functions (e.g. in the case of using an interface to a huge, time intensive library).

Below, we show how to construct in our language a function f which is an inhabitant of a given type of an external function. It is not necessary to demand that f and the external function are equal as set-theoretic maps. They must have the same size dependency, i.e. the same type.

Let $\mathsf{f}_{\mathsf{ext}}$ have the type $\mathsf{L}_n(\alpha) \to \mathsf{L}_{p(n)}(\alpha)$. We define the body of f by the following program expression:

$$\begin{aligned} \mathsf{match}\ l\ \mathsf{with}\ &|\ \mathsf{nil} \Rightarrow \mathsf{nil} \\ &|\ \mathsf{cons}(hd, tl) \Rightarrow \mathsf{gen}\Big(hd,\ \mathsf{p}(p)(\mathsf{length}(l))\Big) \end{aligned}$$

Now we explain the subexpressions in the nil- and cons-branches. In the nil-branch the expression returns the empty list. This is the only choice, due to the following "folklore" property (which to our knowledge was not published earlier).

Lemma 5.5. *Any total polymorphic function* $g : \mathsf{L}(\alpha) \to \mathsf{L}(\alpha)$ *maps the empty list to the empty list.*

Proof. We prove this property using the "free" theorem $\mathsf{map(a)} \circ g_\alpha = g_{\alpha'} \circ \mathsf{map(a)}$ from [Wad05], which holds for all $\mathsf{a} : \alpha \to \alpha'$. Here $\mathsf{map} : (\alpha \to \alpha') \to \mathsf{L}(\alpha) \to \mathsf{L}(\alpha')$ lifts a to lists, and g_α denotes the instantiation of g with type α. Suppose the opposite: g_α sends nil to $[hd \dots stop]$, and $g_{\alpha'}$ sends nil to $[hd' \dots stop']$. Then $\mathsf{map(a)} \circ g_\alpha$ sends nil to $[\mathsf{a}(hd) \dots \mathsf{a}(stop)]$ and $g_{\alpha'} \circ \mathsf{map(a)}$ sends nil to $[hd' \dots stop']$. It is not the case that for all a one has $\mathsf{a}(hd) = hd'$. □

It is a routine exercise to extend this "property for free" to nested lists.

In the cons-branch we use a straightforwardly defined function $\mathsf{gen}(z, x) : \alpha \times \mathtt{Int} \to \mathsf{L}(\alpha)$ that outputs a list of z-s of length x if x is non-negative and does not terminate otherwise. We also use a function generator p, that given a polynomial p, generate a function definition $\mathsf{p}(p) : \mathtt{Int} \to \mathtt{Int}$ such that $\mathsf{p}(p)(n) = p(n)$. It is easy to see that for any non-empty list l of length n the composition $\mathsf{gen}(\mathsf{hd},\ \mathsf{p}(p)(\mathsf{length}(l)))$ terminates if $\mathsf{f_{ext}}$ terminates. It follows from the fact that if $\mathsf{f_{ext}}$ terminates on l then $p(n) \geq 0$, since $p(n)$ is the length of the corresponding output.

6. 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 run-time 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.

6.1. **Further work.** The system is defined for polymorphic lists. Recently, it has been shown [TaShvE08] how to extend the system to ordinary inductive types (no nested inductive definitions).

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. At present we have been studying checking and inference of size annotations in the form of collections of piecewise polynomials that represent at least all possible size dependencies. For instance, insert is annotated with $\{p(n) = n + i\}_{0 \leq i \leq 1}$, and delete is annotated with $\{p(n) = n \dot{-} i\}_{0 \leq i \leq 1}$. Such collections may be potentially infinite, like in the case of recursive application of insert with $\{p(n, m) = n + i\}_{0 \leq i \leq m}$. Here, involvement of real arithmetic is inevitable in type checking. As for inference, when one is interested in strict ("principal type") and polynomial lower and upper bounds, $p_{\min}$ and $p_{\max}$ respectively, it is possible to extend our testing procedure to obtain them. Then, one checks the hypothesis in the form $\{p_{\min} + i\}_{0 \leq i \leq (p_{\max} - p_{\min})}$.

We plan to allow both unsized integers and adding non-trivial sizes to integers. The size of a non-negative sized integer is taken to be its value. This allows to type such functions as $\mathsf{init} : \mathtt{Int}^n \to \mathsf{L}_n(\mathtt{Int})$, which on the integer n outputs the list of 1 of length n. With sized integers one can type such function definitions without introducing dependent types. 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 e.g. on [VasHam03] and [JaySek97].

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

Application of the methodology to estimate stack and time complexity is considered as a topic for future projects.

Acknowledgments

The authors would like to thank Alejandro Tamalet and the anonymous reviewers for their observations and valuable suggestions for improvement. We thank the students of Radboud University Nijmegen, – Willem Peters, Bob Klaase, Elroy Jumpertz, Jeroen Claassens, Martin van de Goor and Ruben Muijrers – without whom implementation of the on-line demonstrator would have been impossible.

References

[AlArGenPuebZan07] Elvira Albert, Puri Arenas, Samir Genaim, German Puebla, Damiano Zanardini. Cost Analysis of Java Bytecode. *16th European Symposium on Programming, ESOP'07, Lecture Notes in Computer Science* 4421:157–172, 2007.

[AlArGenPueb08] Elvira Albert, Puri Arenas, Samir Genaim, German Puebla. Automatic Inference of Upper Bounds for Recurrence Relations in Cost Analysis. *Static Analysis, 15th International Symposium, Lecture Notes in Computer Science*, 5079: 221–237, 2008.

[Am05] Roberto Amadio. Synthesis of max-plus quasi-interpretations. *Fundamenta Informaticae*, 65(1–2):29–60, 2005.

[AmZil] Roberto Amadio, Silvano Dal Zilio. Resource Control for Synchronous Cooperative Threads. *Theoretical Computer Science*, 358:229–254, 2006.

[AsMcK06] David Aspinall, Kenneth MacKenzie. Mobile Resource Guarantees and Policies. *Proc. Intl. Workshop on Construction and Analysis of Safe, Secure and Interoperable Smart Devices (CASSIS 2005, LNCS*, 3956:16–36, 2006.

[AtBailTer07] Vincent Atassi, Patrick Baillot, Kazushige Terui. Verification of Ptime Reducibility for system F Terms: Type Inference in Dual Light Affine Logic. *Logical Methods in Computer Science*, 32, to appear, 2007.

[AvMoSch08] Martin Avanzini, Georg Moser, Andreas Schnabl. Automated Implicit Computational Complexity Analysis (System Description). *Lecture Notes In Artificial Intelligence. Proceedings of the 4th international joint conference on Automated Reasoning*, 5195: 132–138, 2008.

[Ben01] Ralph Benzinger. Automated complexity analysis of Nuprl extracted programs. *Journal of Functional Programming*, 11, Issue 1: 3–31, 2001.

[BonMarMoy05b] Guillaume Bonfante, Jean-Yves Marion, Jean-Yves Moyen. Quasi-interpretations, a way to control resources. *Theoretical Computer Science*, to appear.

[BarSm96] Erik Barendsen, Sjaak Smetsers. Uniqueness typing for functional languages with graph rewriting semantics. *Mathematical Structures in Computer Science*, 6:579–612, 1996.

[Chat90] Siddhartha Chatterjee, Guy E. Blelloch, Allan L. Fisher. Size and access inference for data-parallel programs. *PLDI '91: Proceedings of the ACM SIGPLAN 1991 conference on Programming language design and implementation*, 130–144, 1991.

[Chui87] C. Chui, H.C. Lai. Vandermonde determinant and Lagrange interpolation in R^s. *Nonlinear and convex analysis*, 23–35, 1987.

[vEShvK07] Marko van Eekelen, Olha Shkaravska, Ron van Kesteren, Bart Jacobs, Erik Poll, Sjaak Smetsers. Amortised Heap Space Usage analysis. *Trends In Functional Programming, ed. by Marco T. Morazan*, 8:36–53, 2007.

[GabMarRon08] Marco Gaboardi, Jean-Yves Marion, Simona Ronchi Della Rocca. A Logical Account of PSPACE. *35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages POPL 2008, San Francisco, January 10–12, 2008, Proceedings*, to appear, 2008.

[Gir92] Jean-Yves Girard, Andre Scedrov, Phillip Scott. Bounded linear logic: a modular approach to polynomial-time computability. *Theoretical Computer Science*, 97(1):1–66, 1992.

[GuMeCh09] Sumit Gulwani, Krishna K. Mehra, Trishul M. Chilimbi. SPEED: precise and efficient static estimation of program computational complexity. *ACM Conference Principles of Programming Languages, POPL'09*: 127–139, 2009.

[HerLen01] Christoph A. Herrmann, Christian Lengauer. A transformational approach which combines size inference and program optimization. Walid Taha, editor, *Semantics, Applications, and Implementation of Program Generation (SAIG'01), Lecture Notes in Computer Science*, 2196:199–218, 2001.

[HofJost03] Martin Hofmann, Steffen Jost. Static prediction of heap space usage for first-order functional programs. *SIGPLAN Not.*, 38(1):185–197, 2003.

[JaySek97] C. Barry Jay, Milan Sekanina. Shape checking of array programs. *Computing: the Australasian Theory Seminar, Proceedings, Australian Computer Science Communications*, 19:113–121, 1997.

[vKShvE07] Ron van Kesteren, Olha Shkaravska, Marko van Eekelen. Inferring static non-monotonically sized types through testing. *In Proceedings of 16th International Workshop on Functional and (Constraint) Logic Programming, Paris, WFLP'07*, 2007.

[Lor92] Rudolf A. Lorenz. Multivariate Birkhoff Interpolation. *Lecture Notes in Math.*, 1516, 1992.

[MarPech] Jean-Yves Marion, Romain Pechoux. Resource analysis by sup-interpretations. *Functional and LOgic Programming 8th international Symposium (FLOPS 2006), Lecture notes in Computer Science*, 3945, 2006.

[Mat91] Yuri Matiyasevich, James P. Jones. Proof of recursive unsolvability of Hilbert's tenth problem. *American Mathematical Monthly*, 98(10):689–709, 1991.

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

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

[ShvE0T8] Olha Shkaravska, Marko van Eekelen, Alejandro Tamalet. Collected Size Semantics for Functional Programs. *Technical report: ICIS-R08021, Radboud University Nijmegen*, November 2008.

[ShvKvE07a] Olha Shkaravska, Ron van Kesteren, Marko van Eekelen. Polynomial size analysis of first-order functions. *Technical Report ICIS-R07004, Radboud University Nijmegen*, January 2007.

[ShvKvE07b] Olha Shkaravska, Ron van Kesteren, Marko van Eekelen. Polynomial size analysis of first-order functions. *Typed Lambda Calculi end Applications, TLCA'07, Lecture Notes in Computer Science*, 4583:351–365, 2007.

[TaShvE08] Alejandro Tamalet, Olha Shkaravska, Marko van Eekelen. Size Analysis of Algebraic Data Types. *Selected Papers of the* 9^{th} *International Symposium on Trends in Functional Programming (TFP'08)*. (Ed). Marco Morazán, *Intellect Publishers*, 2008, to appear.

[VasHam03] Pedro Baltazar Vasconcelos, Kevin Hammond. Inferring cost equations for recursive, polymorphic and higher-order functional programs. 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, Lecture Notes in Computer Science*, 3145:86–101, 2004.

[Wad05] Philip Wadler. Theorems for Free! (1989). *Proceedings 4th Int. Conf.on Funct. Prog.Languages and Computer Arch., FPCA'89, London, UK, 11–13 Sept*, 1989.

Appendix: auxiliary lemmata for soundness proof

Lemma 6.1 (A program value's footprint is in the heap). $\mathcal{R}(h,\ v) \subseteq \mathit{dom}(h)$.

Proof. The lemma is proved by induction on the size of the (domain of the) heap h.

$\mathit{dom}(h) = \emptyset$**:** Then no $\ell \in \mathit{dom}(h)$ exists and $\mathcal{R}(h,\ v) = \emptyset$.

$\mathit{dom}(h) \neq \emptyset$**:** $v = c$ **or** $v = \texttt{NULL}$**:** Then $\mathcal{R}(h,\ v) = \emptyset$, which is trivially a subset of $\mathit{dom}(h)$.

$v = \ell$ **and** $\mathit{dom}(h) = (\mathit{dom}(h) \setminus \{\ell\}) \cup \{\ell\}$**:** From the definition of $\mathcal{R}$ we get $\mathcal{R}(h,\ \ell) = \{\ell\} \cup \mathcal{R}(h|_{\mathit{dom}(h)\setminus\{\ell\}},\ h.l.\texttt{hd}) \cup \mathcal{R}(h|_{\mathit{dom}(h)\setminus\{\ell\}},\ h.l.\texttt{tl})$. Applying the induction hypotheses we derive that $\mathcal{R}(h|_{\mathit{dom}(h)\setminus\{\ell\}},\ h.\ell.\texttt{hd}) \subseteq \mathit{dom}(h|_{\mathit{dom}(h)\setminus\{\ell\}})$ and $\mathcal{R}(h|_{\mathit{dom}(h)\setminus\{\ell\}},\ h.\ell.\texttt{tl}) \subseteq \mathit{dom}(h|_{\mathit{dom}(h)\setminus\{\ell\}})$. Hence, $\mathcal{R}(h,\ l) \subseteq \mathit{dom}(h)$.

□

Lemma 6.2 (Extending a heap does not change the footprints of program values). *If $\ell \notin \mathit{dom}(h)$ and $h' = h[\ell.\texttt{hd} := v_{\texttt{hd}},\ \ell.\texttt{tl} := v_{\texttt{tl}}]$ for some $v_{\texttt{hd}},\ v_{\texttt{tl}}$ then for any $v \neq \ell$ one has $\mathcal{R}(h,\ v) = \mathcal{R}(h',\ v)$.*

Proof. The lemma is proved by induction on the size of the (domain of the) heap h.

$\mathit{dom}(h) = \emptyset$**:** Since $h' = [\ell.\texttt{hd} = v_{\texttt{hd}}, \ell.\texttt{tl} := v_{\texttt{tl}}]$ and $v \neq \ell$ we have $v \notin \{\ell\} = \mathit{dom}(h')$. Therefore, $\mathcal{R}(h,\ v) = \emptyset = \mathcal{R}(h',\ v)$.

$\mathit{dom}(h) \neq \emptyset$**:** We proceed by case distinction on v.

$v = c$ **or** $v = \texttt{NULL}$**:** Then, $\mathcal{R}(h,\ v) = \emptyset = \mathcal{R}(h',\ v)$.

$v = \ell'$**:** If $\ell' \notin \mathit{dom}(h)$, then due to $\ell' \neq \ell$ we have $\ell' \notin \mathit{dom}(h)$ as well and $\mathcal{R}(h,\ v) = \emptyset = \mathcal{R}(h',\ v)$.

Let $\ell' \in \mathit{dom}(h)$. From the definition of $\mathcal{R}$ we get

$$\mathcal{R}(h,\ \ell') = \{\ell'\}\ \cup\ \mathcal{R}(h|_{\mathit{dom}(h)\setminus\{\ell'\}},\ h.\ell'.\texttt{hd})\ \cup\ \mathcal{R}(h|_{\mathit{dom}(h)\setminus\{\ell'\}},\ h.\ell'.\texttt{tl}).$$

Due to $h'(\ell') = h(\ell')$ and

$$h'|_{\mathit{dom}(h')\setminus\{\ell'\}} = h|_{\mathit{dom}(h)\setminus\{\ell'\}}[\ell.\texttt{hd} := v_{\texttt{hd}},\ \ell.\texttt{tl} := v_{\texttt{tl}}],$$

and the induction assumption one has

$$\begin{array}{l}
\mathcal{R}(h|_{\mathit{dom}(h)\setminus\{\ell'\}},\ h.\ell'.\texttt{hd}) = \mathcal{R}(h'|_{\mathit{dom}(h')\setminus\{\ell'\}},\ h'.\ell'.\texttt{hd}) \\
\mathcal{R}(h|_{\mathit{dom}(h)\setminus\{\ell'\}},\ h.\ell'.\texttt{tl}) = \mathcal{R}(h'|_{\mathit{dom}(h')\setminus\{\ell'\}},\ h'.\ell'.\texttt{tl})
\end{array}$$

So,

$$\begin{array}{l}
\mathcal{R}(h',\ \ell') = \\
= \{\ell'\}\ \cup\ \mathcal{R}(h'|_{\mathit{dom}(h')\setminus\{\ell'\}},\ h'.\ell'.\texttt{hd})\ \cup\ \mathcal{R}(h'|_{\mathit{dom}(h')\setminus\{\ell'\}},\ h'.\ell'.\texttt{tl}) = \\
= \{\ell'\}\ \cup \mathcal{R}(h|_{\mathit{dom}(h)\setminus\{\ell'\}},\ h.\ell'.\texttt{hd})\ \cup\ \mathcal{R}(h|_{\mathit{dom}(h)\setminus\{\ell'\}},\ h.\ell'.\texttt{tl}) = \\
= \mathcal{R}(h,\ \ell').
\end{array}$$

□

Lemma 6.3 (Extending heaps preserves model relations)**.**
For all heaps h and h', if $h'|_{dom(h)} = h$ then $v \models^{h}_{\tau^\bullet} w$ implies $v \models^{h'}_{\tau^\bullet} w$.

Proof.
The lemma is proved by induction on the structure of $\tau^\bullet$.

$\tau^\bullet = \mathtt{Int}$**:** In this case, v is a constant c and $w = c$, hence $v \models^{h'}_{\tau^\bullet} w$ by the definition.

$\tau^\bullet = \mathsf{L}_{n^\bullet}(\tau^{\bullet\prime})$**:** We proceed by induction on $n^\bullet$.

$n^\bullet = 0$**:** In this case, $v = \mathtt{NULL}$ and $w = \mathtt{[]}$, hence $v \models^{h'}_{\tau^\bullet} w$ by the definition.

$n^\bullet = m^\bullet + 1$**:** By the definition v is a location ℓ and $\ell \models^{h}_{\mathsf{L}_{m^\bullet+1}(\tau^{\bullet\prime})} w_{\mathtt{hd}} :: w_{\mathtt{tl}}$ for some $w_{\mathtt{hd}}$ and $w_{\mathtt{tl}}$ such that

$$\begin{array}{l} \ell \in dom(h), \\ h.\ell.\mathtt{hd} \models^{h|_{dom(h)\setminus\{\ell\}}}_{\tau^{\bullet\prime}} w_{\mathtt{hd}}, \\ h.\ell.\mathtt{tl} \models^{h|_{dom(h)\setminus\{\ell\}}}_{\mathsf{L}_{m^\bullet}(\tau^{\bullet\prime})} w_{\mathtt{tl}} \end{array}$$

We want to apply the induction assumption, with heaps $h|_{dom(h)\setminus\{\ell\}}$, $h'|_{dom(h')\setminus\{\ell\}}$ (as "h" and "h'" respectively). The condition of the lemma is satisfied because

$$\begin{array}{l} h'|_{dom(h')\setminus\{\ell\}}|_{dom(h|_{dom(h)\setminus\{\ell\}})} \\ = h'|_{dom(h')\setminus\{\ell\}}|_{dom(h)\setminus\{\ell\}} \\ = h'|_{dom(h)\setminus\{\ell\}} = h|_{dom(h)\setminus\{\ell\}} \end{array}$$

Thus, we apply the induction assumption and with $h.\ell = h'.\ell$ obtain

$$\begin{array}{l} \ell \in dom(h'), \\ h'.\ell.\mathtt{hd} \models^{h'|_{dom(h')\setminus\{\ell\}}}_{\tau^{\bullet\prime}} w_{\mathtt{hd}}, \\ h'.\ell.\mathtt{tl} \models^{h'|_{dom(h')\setminus\{\ell\}}}_{\mathsf{L}_{m^\bullet}(\tau^{\bullet\prime})} w_{\mathtt{tl}} \end{array}$$

Then, $\ell \models^{h'}_{\mathsf{L}_{m^\bullet+1}(\tau^{\bullet\prime})} w_{\mathtt{hd}} :: w_{\mathtt{tl}}$ by the definition.

□

Lemma 6.4 (The model relation for v depends only on values in the footprint of v)**.**
For v, h, w, and $\tau^\bullet$, the relation $v \models^{h}_{\tau^\bullet} w$ implies $v \models^{h|_{\mathcal{R}(h,\ v)}}_{\tau^\bullet} w$.

Proof. The lemma is proved by induction on $\tau^\bullet$.

$\tau^\bullet = \mathtt{Int}$**:** By the definition, v is a constant c and thus $w = c$. Then $v \models^{h|_{\mathcal{R}(h,\ v)}}_{\tau^\bullet} w$.

$\tau^\bullet = \mathsf{L}_{n^\bullet}(\tau^\bullet)$**:** We proceed by induction on $n^\bullet$.

$\tau^\bullet = \mathsf{L}_0(\tau^{\bullet\prime})$**:** By the definition $v = \mathtt{NULL}$ and $w = \mathtt{[]}$. Then $v \models^{h|_{\mathcal{R}(h,\ v)}}_{\tau^\bullet} w$.

$\tau^\bullet = \mathsf{L}_{m^\bullet+1}(\tau^{\bullet\prime})$**:** By the definition $v = \ell$. Then $\ell \models^{h}_{\mathsf{L}_{m^\bullet+1}(\tau^{\bullet\prime})} w$ means that $w = w_{\mathtt{hd}} :: w_{\mathtt{tl}}$ for some $w_{\mathtt{hd}}$ and $w_{\mathtt{tl}}$, and

$$\begin{array}{l} \ell \in dom(h), \\ h.\ell.\mathtt{hd} \models^{h|_{dom(h)\setminus\{\ell\}}}_{\tau^{\bullet\prime}} w_{\mathtt{hd}}, \\ h.\ell.\mathtt{tl} \models^{h|_{dom(h)\setminus\{\ell\}}}_{\mathsf{L}_{m^\bullet}(\tau^{\bullet\prime})} w_{\mathtt{tl}} \end{array}$$

We apply the induction assumption, with the heap $h|_{dom(h)\setminus\{\ell\}}$:

$$\begin{aligned}
&\ell \in \mathit{dom}(h),\\
&h.\ell.\mathtt{hd} \;\models^{h|_{\mathit{dom}(h)\setminus\{\ell\}}|_{\mathcal{R}(h|_{\mathit{dom}(h)\setminus\{\ell\}},\ h.\ell.\mathtt{hd})}}_{\tau^{\bullet\prime}} \; w_{\mathtt{hd}},\\
&h.\ell.\mathtt{tl} \;\models^{h|_{\mathit{dom}(h)\setminus\{\ell\}}|_{\mathcal{R}(h|_{\mathit{dom}(h)\setminus\{\ell\}},\ h.\ell.\mathtt{tl})}}_{\mathsf{L}_{m^\bullet}(\tau^{\bullet\prime})} \; w_{\mathtt{tl}}
\end{aligned}$$

Due to $\mathcal{R}(h|_{\mathit{dom}(h)\setminus\{\ell\}},\ h.\ell.\mathtt{hd}) \subseteq \mathit{dom}(h) \setminus \{\ell\}$ (lemma 6.1) we have

$$\begin{aligned}
&h|_{\mathit{dom}(h)\setminus\{\ell\}}|_{\mathcal{R}(h|_{\mathit{dom}(h)\setminus\{\ell\}},\ h.\ell.\mathtt{hd})} =\\
&= h|_{\mathcal{R}(h|_{\mathit{dom}(h)\setminus\{\ell\}},\ h.\ell.\mathtt{hd})} =\\
&= h|_{\mathcal{R}(h|_{\mathit{dom}(h)\setminus\{\ell\}},\ h.\ell.\mathtt{hd})\setminus\{\ell\}}.
\end{aligned}$$

Similarly $h|_{\mathit{dom}(h)\setminus\{\ell\}}|_{\mathcal{R}(h|_{\mathit{dom}(h)\setminus\{\ell\}},\ h.\ell.\mathtt{tl})} = h|_{\mathcal{R}(h|_{\mathit{dom}(h)\setminus\{\ell\}},\ h.\ell.\mathtt{tl})\setminus\{\ell\}}$.
Due to $\ell \in \mathcal{R}(h,\ \ell)$, and lemma 6.3 – with $\mathcal{R}(h|_{\mathit{dom}(h)\setminus\{\ell\}},\ h.\ell.\mathtt{hd}) \setminus \{\ell\} \subseteq \mathcal{R}(h,\ h.\ell.\mathtt{hd}) \setminus \{\ell\}$, we have

$$\begin{aligned}
&\ell \in \mathit{dom}(h_{\mathcal{R}(h,\ \ell)}),\\
&h|_{\mathcal{R}(h,\ \ell)}.\ell.\mathtt{hd} \;\models^{h|_{\mathcal{R}(h,\ h.\ell.\mathtt{hd})\setminus\{\ell\}}}_{\tau^{\bullet\prime}} \; w_{\mathtt{hd}},\\
&h|_{\mathcal{R}(h,\ \ell)}.\ell.\mathtt{tl} \;\models^{h|_{\mathcal{R}(h,\ h.\ell.\mathtt{hd})\setminus\{\ell\}}}_{\mathsf{L}_{n^\bullet}(\tau^{\bullet\prime})} \; w_{\mathtt{tl}}
\end{aligned}$$

Thus, $\ell \;\models^{h|_{\mathcal{R}(h,\ \ell)}}_{\mathsf{L}_{m^\bullet+1}(\tau^{\bullet\prime})} \; w_{\mathtt{hd}} :: w_{\mathtt{tl}}$.

□

Lemma 6.5 (Equality of footprints implies equivalence of model relations). *If* $h|_{\mathcal{R}(h,\ v)} = h'|_{\mathcal{R}(h,\ v)}$ *then* $v \;\models^{h}_{\tau^\bullet}\; w$ *implies* $v \;\models^{h'}_{\tau^\bullet}\; w$.

Proof. Assume $v \;\models^{h}_{\tau^\bullet}\; w$. Lemma 6.4 states that this implies $v \;\models^{h|_{\mathcal{R}(h,\ v)}}_{\tau^\bullet}\; w$. Assuming $h|_{\mathcal{R}(h,\ v)} = h'|_{\mathcal{R}(h,\ v)}$ we get $v \;\models^{h'|_{\mathcal{R}(h,\ v)}}_{\tau^\bullet}\; w$. Since $\mathit{dom}(h'|_{\mathcal{R}(h,\ v)}) = \mathit{dom}(h|_{\mathcal{R}(h,\ v)}) = \mathcal{R}(h,\ v)$ we have $h'|_{\mathit{dom}(h'|_{\mathcal{R}(h,\ v)})} = h'|_{\mathcal{R}(h,\ v)}$ and we may apply lemma 6.3, which gives $v \;\models^{h'}_{\tau^\bullet}\; w$. □

Lemma 6.6 (Extending a store preserves the validity of the store). *Given a ground context* $\Gamma^\bullet$*, store* s*, heap* h*, value* v*, a set of variables* vars *and a variable* $x \notin \mathit{vars}$*, s.t.* $x \notin \mathit{dom}(s)$*, one has*

$$\mathit{Valid}_{\mathsf{store}}(\mathit{vars}, \Gamma^\bullet, s[x := v], h) \iff \mathit{Valid}_{\mathsf{store}}(\mathit{vars}, \Gamma^\bullet, s, h)$$

Proof. The lemma follows from the definition of $\mathit{Valid}_{\mathsf{store}}$. □

Lemma 6.7 (Weakening for valid stores). *Given a set of variables* vars_1*, ground context* $\Gamma^\bullet$*, stack* s*, and heap* h*, for any set of variables* vars_2 *such that such that* $\mathit{vars}_2 \subseteq \mathit{vars}_1$ *one has*

$$\mathit{Valid}_{\mathsf{store}}(\mathit{vars}_1, \Gamma^\bullet, s, h) \implies \mathit{Valid}_{\mathsf{store}}(\mathit{vars}_2, \Gamma^\bullet, s, h)$$

Proof. The lemma follows from the definition of $\mathit{Valid}_{\mathsf{store}}$. □

Lemma 6.8 (Validity for the disjoint union of sets of variables). *For any store* s *and a ground context* $\Gamma^\bullet$ *one has*

$$\mathit{Valid}_{\mathsf{store}}(\mathit{vars}_1 \cup \mathit{vars}_2, \Gamma^\bullet, s, h) \iff \mathit{Valid}_{\mathsf{store}}(\mathit{vars}_1, \Gamma^\bullet, s, h) \;\wedge\; \mathit{Valid}_{\mathsf{store}}(\mathit{vars}_2, \Gamma^\bullet, s, h)$$

Proof. The lemma follows immediately from the definition of a valid store. □