Type Checking and Weak Type Inference for Polynomial Size Analysis of First-Order Functions

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

Technical Report ICIS-R07004
Institute for Computing and Information Sciences
Radboud University Nijmegen

Abstract. We present a size-aware type system for first-order shapely functions. Here, a function is called *shapely* when the size of the result is determined exactly by a polynomial in the sizes of the arguments. Examples of shapely functions are matrix multiplication and the Cartesian product of two lists.
The type checking problem for the type system is shown to be undecidable in general. We define a natural syntactic restriction such that the type checking becomes decidable, even though size polynomials are not necessarily linear. Furthermore, an algorithm for weak type inference for this system is given.[1]

Keywords: Shapely Programs, Size Analysis, Type Checking, Diophantine equations

1 Introduction

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

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

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

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

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

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 can be expressed in a type by $\mathsf{L}_n(\alpha) \times \mathsf{L}_m(\alpha) \rightarrow \mathsf{L}_{n*m}(\mathsf{L}_2(\alpha))$.

1.1 Related work

Information about input-output size dependencies is applied in time and space analysis and optimization, because run time and heap-space consumption obviously depend on the sizes of the data structures involved in the computations. Knowledge of the exact size of data structures can be used to improve heap space analysis for programs with destructive pattern matching. Amortized heap space analysis has been developed for linear bounds by Hofmann and Jost [HJ03]. Precise knowledge of sizes is required to extend this approach to non-linear bounds [EJPS05]. 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 [CBF91].

The analysis of (exact) input-output size dependencies of functions itself has been explored in a series of work. Some interesting work on shape analysis has been done by Jay and Sekanina [JS97]. In this work, a shapely program is translated into a corresponding abstract program 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 this expression 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 Hermann and Lengauer [HL01], or through a type inference procedure, as presented by Vasconcelos and Hammond [VK04]. Both results can be applied to non-shapely programs, 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.

To our knowledge, the only work yielding closed-form solutions for size dependencies is limited to linear dependencies. For instance, in the well-known work of Pareto [Par98], where *non-strict* sized types are used to prove termination. Linearity is a sufficient condition for the type checking procedure to be decidable.

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 linear dependencies.

1.2 Contents of the paper

In this work, we go beyond linearity and consider a type checking procedure for a first-order functional programming language (section 2) with polynomial size dependencies (section 3). We show that type checking is reduced to the entailment checking over Diophantine equations. Type checking is shown to be undecidable in general, but decidable under a natural syntactic condition ("no-let-before-match", section 4). We give a procedure for weak type inference in section 5. In section 6 we define a heap-aware semantics of types and expressions and sketch the proof of the soundness statement with respect to this semantics. Finally, in section 7 we overview the results and discuss further work.

2 Language

The typing system is designed for a first-order functional language over integers and (polymorphic) lists. The syntax of language expressions is defined by the following grammar, where *Int* denote the sets of integer constants, x and y denote zero-order program variables, and f denotes a function name (the example in the introduction used a sugared version of this syntax):

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

The syntax distinguishes between zero-order let-binding of variables and first-order letfun-binding of functions. In a function body, the only free program variables that may occur are its parameters: $FV(e_1) \subseteq \{x_1, \ldots, x_n\}$. The operational semantics is standard, therefore the definition is postponed until it is used to prove soundness (section 6.1).

3 Type System

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

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

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

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

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

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

Abusing notation, we denote via $FV(-)$ free variables of an expression e and free type variables in a type (scheme) τ.

Let $FVS(-)$ denote the free size variables in zero-order types:

$$\begin{array}{ll} FVS(\mathtt{Int}) & = \emptyset \\ FVS(\alpha) & = \emptyset \\ FVS(\mathsf{L}_p(\tau)) & = FVS(p) \cup FVS(\tau) \end{array}$$

where, $FVS()$ "overloaded" for size expressions is:

$$\begin{array}{ll} FVS(i) & = \emptyset \\ FVS(n) & = \{n\} \\ FVS(p_1 + p_2) = FVS(p_1 - p_2) = FVS(p_1 * p_2) & = FVS(p_1) \cup FVS(p_2) \end{array}$$

For instance, $FVS(\mathsf{L}_{(n-3)^2}(\mathsf{L}_{n+m}(\alpha))) = \{n,\ m\}$.

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

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

For instance, $\mathsf{L}_{(n-3)^2}(\mathsf{L}_{n+m}(\alpha))$ is not a ground type, whereas $\mathsf{L}_{(1-3)^2}(\mathsf{L}_{1+4}(\mathtt{Int})) = \mathsf{L}_4(\mathsf{L}_5(\mathtt{Int}))$ – yes. The axiomatics of the integer ring holds for size expressions. It is used in the typing rules as well (see below).

One does not want free size variables on the r.h.s. of first-order types and typing judgements to appear"out of the blue", that is not from the typing contexts, but as wild cards. Size wild cards may appear due to nested empty lists:

$$\mathsf{free\_sv} = \ \mathsf{let}\ x = \mathsf{nil}\ \mathsf{in}\ \mathsf{copy\_nested} :\ \mathtt{Int} \to \mathsf{L}_0(\mathsf{L}_{?m}(\alpha))$$

where $\mathsf{copy\_nested} : \mathsf{L}_n(\mathsf{L}_m(\alpha)) \to \mathsf{L}_n(\mathsf{L}_m(\alpha))$,

In other words one can succesfully type-check "everything":
$\mathsf{free\_sv} :\ \mathtt{Int} \to \mathsf{L}_0(\mathsf{L}_0(\alpha))$,
$\mathsf{free\_sv} :\ \mathtt{Int} \to \mathsf{L}_0(\mathsf{L}_{35}(\alpha))$,
$\mathsf{free\_sv} :\ \mathtt{Int} \to \mathsf{L}_0(\mathsf{L}_{2007}(\alpha))$,
...

However, it is easy to see that sets $\mathsf{L}_0(\mathsf{L}_m(\mathtt{Int}))$ are equal and contain the single element $[\,]$ for all instantiations of m. The similar holds for $\mathsf{L}_0(\mathsf{L}_m(\alpha))$. This induces the natural equivalence relation on types. For instance $\mathsf{L}_q(\mathsf{L}_0(\mathsf{L}_p(\alpha))) \equiv \mathsf{L}_q(\mathsf{L}_0(\mathsf{L}_{p'}(\alpha)))$. The *canonical representative* of this class is $\mathsf{L}_q(\mathsf{L}_0(\mathsf{L}_0(\alpha)))$.

So, the freeness of size variable should be checked w.r.t. canonical forms. This is taken into account in the definition of 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: Diff. with TLCA

$FTypes\tau^f ::= \tau_1^\circ \times \ldots \times \tau_k^\circ \to \tau_{k+1}$ such that
$FVS(canon(*(\tau_{k+1}))) \subseteq FVS(canon(*(\tau_1^\circ))) \cup \cdots \cup FVS(canon(*(\tau_k^\circ)))$
for all substitutions $*$
of the size variables from $\tau_1^\circ, \ldots \tau_k^\circ$, τ_{k+1} with size expressions

The following lemma shows that it is possible to validate in finite number of steps, if a given arrow-type is a well-formed first-order type.

Lemma 1. *Inclusion*

$FVS(canon(*(\tau_{k+1}))) \subseteq FVS(canon(*(\tau_1^\circ))) \cup \cdots \cup FVS(canon(*(\tau_k^\circ)))$
for all substitutions $*$
of the size variables from $\tau_1^\circ, \ldots \tau_k^\circ$, τ_{k+1} *with size expressions*

holds if and only if for any substitution $*'$ *such that it is the ground zero substitution on some subset of the size variables and it is the identity on its complement, we have*

$$FVS(canon(*'(\tau_{k+1}))) \subseteq FVS(canon(*'(\tau_1^\circ))) \cup \cdots \cup FVS(canon(*'(\tau_k^\circ)))$$

It easy to see, that there are finitely many substitutions $*'$, namely $\Sigma_{i=0}^N C_N^i = 2^N$, where N is the amount of size variables from $\tau_1^\circ, \ldots \tau_k^\circ$, τ_{k+1} and i denotes the amount of variables substituted to zero.

Proof. Fix an arrow type $\tau_1^\circ \times \ldots \times \tau_k^\circ \to \tau_{k+1}$. Let $\tau_i^\circ = \mathsf{L}_{n_{i1}}(\ldots \mathsf{L}_{n_{il_i}}(\alpha)\ldots)$ and $\tau_{k+1} = \mathsf{L}_{q_1}(\ldots \mathsf{L}_{q_l}(\alpha)\ldots)$.

Let for any substitution $*'$ such that it is the ground zero substitution on some subset of the size variables and it is the identity on its complement, we have

$$FVS(canon(*'(\tau_{k+1}))) \subseteq FVS(canon(*'(\tau_1^\circ))) \cup \cdots \cup FVS(canon(*'(\tau_k^\circ)))$$

Show that the definition of the well-formed first-order type holds. Fix any substitution $*$. Without lost of generality think that all free size variables in the substitution are fresh.

- For the sake of clearness, first consider the trivial case, when $*$ does not substitute any size variable into 0. Then, say, $*(\tau_i^\circ) = *(\mathsf{L}_{n_{i1}}(\ldots \mathsf{L}_{n_{il_i}}(\alpha)\ldots)) = \mathsf{L}_{p_{i1}}(\ldots \mathsf{L}_{p_{il_i}}(\alpha)\ldots)$, where $*(n_{ij}) = p_{ij}$, coincides with its canonical form. Let $m \in FVS(canon(*(\tau_{k+1})))$. Then

$$*(\tau_{k+1}) = \mathsf{L}_{*q_1}(\ldots \mathsf{L}_{*q_{j-1}}(\mathsf{L}_{q_j(\ldots,n:=p,\ldots)}(\mathsf{L}_{*q_{j+1}}(\ldots \mathsf{L}_{*q_l}(\alpha)\ldots)))\ldots).$$

for some $n \in FVS(\tau_{k+1})$ and $p(\ldots, m, \ldots) = *(n) \neq 0$, with $*(q'_j) \neq 0$ for $1 \leq j' \leq j$.
From the assumption of the lemma, letting $*'$ be the identity, we have that $FVS(\tau_{k+1}) \subseteq FVS(\tau_1^\circ) \cup \cdots \cup FVS(\tau_k^\circ)$. From what follows that $n \in FVS(\tau_i^\circ)$ for some i, that is $n = n_{ii'}$ for some i'. Then

$$*(\tau_i^\circ) = \mathsf{L}_{p_{i1}}(\ldots \mathsf{L}_{p_{i\ i'-1}}(\mathsf{L}_{p(\ldots, m, \ldots)}(\mathsf{L}_{p_{i\ i'+1}}(\ldots \mathsf{L}_{p_{il_i}}(\alpha)\ldots)))\ldots),$$

that is $m \in FVS(*(\tau_i^\circ))$ as well.

– Now, let $*$ be any substitution. Let $m \in FVS(canon(*(\tau_{k+1})))$. Then

$$*(\tau_{k+1}) = \mathsf{L}_{*q_1}(\ldots \mathsf{L}_{*q_{j-1}}(\mathsf{L}_{q_j(\ldots, n:=p, \ldots)}(\mathsf{L}_{*q_{j+1}}(\ldots \mathsf{L}_{*q_l}(\alpha)\ldots)))\ldots).$$

for some $n \in FVS(\tau_{k+1})$ and $p(\ldots, m, \ldots) = *(n) \neq 0$, and not equal to zero size expressions $*q_1, \ldots, *q_j = q_j(\ldots, n := p(\ldots, m \ldots), \ldots)$.

Let $*'$ be the identity on the size variables, where $*$ is non-zero, and $(*')(n') = 0$ once $*(n') = 0$. Show, that $(*')(q_1), \ldots (*')(q_j)$ are not zero polynomials. Suppose the opposite: $*'q_{j'} = q_{j'}(n_1, \ldots, n_d, 0, \ldots, 0) = 0$ for all $n_1, \ldots, n_d$. Then $*q_{j'} = q_{j'}(p'_1, \ldots, p'_d, 0, \ldots, 0) = 0$ which contradicts the statement above.

From that fact that $(*')(q_1), \ldots (*')(q_j)(\ldots, n \ldots)$ are not zero polynomials, it follows, that $n \in FVS(canon(*'(\tau_{k+1})))$, and from the assumption $FVS(canon(*'(\tau_{k+1}))) \subseteq FVS(canon(*'(\tau_1^\circ))) \cup \cdots \cup FVS(canon(*'(\tau_k^\circ)))$ we have $n \in FVS(canon(*'(\tau_i^\circ)))$ for some i, that is $n = n_{ii'}$ for some i', and $*'(n_{is}) = n_{is} \neq 0$, $1 \leq s \leq i'$.

Therefore, $*(n_{is}) \neq 0$. Thus, $*(\tau_i^\circ) = \mathsf{L}_{*(n_{i1})}(\ldots \mathsf{L}_{*n_{ii'} = p(\ldots, m, \ldots)}(\ldots)\ldots)$ and $m \in FVS(canon(*(\tau_i^\circ)))$.

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

The typeof a shapely transpose function, if it exists, must be $\mathsf{L}_n(\mathsf{L}_m(\alpha)) \to \mathsf{L}_m(\mathsf{L}_n(\alpha))$ and is not a well-formed first-order type. Indeed, for any ground instantiation $n = 0$, $m = i$, with $i > 0$, we have $\mathsf{L}_0(\mathsf{L}_i(\alpha)) \to \mathsf{L}_i(\mathsf{L}_0(\alpha))$, with $FVS(canon(\mathsf{L}_i(\mathsf{L}_0(\alpha)))) = i$, $FVS(canon(\mathsf{L}_0(\mathsf{L}_i(\alpha)))) = \emptyset$. Note, that we were not able to encode transposition of a matrix without non-exhaustive pattern-matching. We leave adding non-exhaustive pattern matchings to our language for the future work.

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 $FVS(-)$ is straightforwardly extended to contexts:

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

3.1 Typing rules

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

In the typing rules, $D \vdash p = p'$ means that $p = p'$ is derivable from D in equational reasoning in the ring of integers. $D \vdash \tau \equiv \tau'$ means the following:

- $\tau = \tau' = \mathtt{Int}$, or $\tau = \tau' = \alpha$ for some α, or $\tau = \mathsf{L}_{p_1}(\ldots \mathsf{L}_{p_l}(\mathtt{Int})\ldots)$ and $\tau' = \mathsf{L}_{p'_1}(\ldots \mathsf{L}_{p'_l}(\mathtt{Int})\ldots)$, or $\tau = \mathsf{L}_{p_1}(\ldots \mathsf{L}_{p_l}(\alpha)\ldots)$ and $\tau' = \mathsf{L}_{p'_1}(\ldots \mathsf{L}_{p'_l}(\alpha)\ldots)$.
- For the last two cases
  - If exists such i, that $D \vdash p_i = 0$ (or $D \vdash p'_i = 0$) then take the minimal such i and accept the typing if and only if $D \vdash p_j = p'_j$ for all $1 \leq j \leq i$.
  - If no such i exists then $D \vdash p_j = p'_j$ for all $1 \leq j \leq l$.

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

The sugared version (τ is a canonical type):

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

In the rule below $\mathsf{L}_0(\tau_0)$ has the same underlying type as τ, but with all size annotations set to 0. We emphasize that τ is of a list type as well.

$$\frac{D \vdash \tau \equiv \mathsf{L}_0(\tau_0)}{D;\ \Gamma\ \vdash_\Sigma \mathsf{nil}\!:\!\tau}\ \textsc{Nil}$$

The sugared version
(τ is a canonical type, but in this rule canonization is not necessary):

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

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

The sugared ("canonical") version

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

$$\dfrac{\begin{array}{c} D \vdash \tau \equiv \tau' \\ D \vdash \tau \equiv \tau'' \\ \Gamma(x) = \mathtt{Int} \qquad D;\ \Gamma\ \vdash_{\Sigma} e_t : \tau' \qquad D;\ \Gamma\ \vdash_{\Sigma} e_f : \tau'' \end{array}}{D;\ \Gamma\ \vdash_{\Sigma} \mathsf{if}\ x\ \mathsf{then}\ e_t\ \mathsf{else}\ e_f : \tau}\ \textsc{If}$$

The sugared version:

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

$$\dfrac{\begin{array}{c} D \vdash \tau_x \equiv \tau'_x \\ D \vdash \tau \equiv \tau' \\ x \notin dom(\Gamma) \qquad D;\ \Gamma\ \vdash_{\Sigma} e_1 : \tau_x \qquad D;\ \Gamma,\ x : \tau'_x\ \vdash_{\Sigma} e_2 : \tau' \end{array}}{D;\ \Gamma\ \vdash_{\Sigma} \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 : \tau}\ \textsc{Let}$$

The sugared version : $\tau_x,\ \tau$ are canonical

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

$$p = 0,\ D;\ \Gamma,\ x : \mathsf{L}_p(\tau')\ \vdash_{\Sigma} e_{\mathsf{nil}} : \tau_{\mathsf{nil}}$$

$D \vdash \tau \equiv \tau_{\mathsf{nil}}$ (The canonical form w.r.t. not only D,
but w.r.t. $p = 0$ as well,
will be considered when the rule for e_{nil} is applied)

$$\dfrac{\begin{array}{c} D \vdash \tau \equiv \tau_{\mathsf{cons}} \\ D \vdash \tau' \equiv \tau'_1 \\ D \vdash \tau' \equiv \tau'_2 \\ hd, tl \notin dom(\Gamma) \\ D;\ \Gamma, hd : \tau'_1,\ x : \mathsf{L}_p(\tau'),\ tl : \mathsf{L}_{p-1}(\tau'_2)\ \vdash_{\Sigma} e_{\mathsf{cons}} : \tau_{\mathsf{cons}} \end{array}}{D;\ \Gamma,\ x : \mathsf{L}_p(\tau')\ \vdash_{\Sigma} \mathsf{match}\ x\ \mathsf{with}\ \begin{array}{l} \mid \mathsf{nil} \Rightarrow e_{\mathsf{nil}} \\ \mid \mathsf{cons}(hd, tl) \Rightarrow e_{\mathsf{cons}} \end{array} : \tau}\ \textsc{Match}$$

The sugared version : $\tau,\ \tau'$are canonical w.r.t. D

$$\dfrac{\begin{array}{c} p = 0,\ D;\ \Gamma,\ x : \mathsf{L}_p(\tau')\ \vdash_{\Sigma} e_{\mathsf{nil}} : \tau \\ hd, tl \notin dom(\Gamma) \\ D;\ \Gamma, hd : \tau'_1,\ x : \mathsf{L}_p(\tau'),\ tl : \mathsf{L}_{p-1}(\tau'_2)\ \vdash_{\Sigma} e_{\mathsf{cons}} : \tau_{\mathsf{cons}} \end{array}}{D;\ \Gamma,\ x : \mathsf{L}_p(\tau')\ \vdash_{\Sigma} \mathsf{match}\ x\ \mathsf{with}\ \begin{array}{l} \mid \mathsf{nil} \Rightarrow e_{\mathsf{nil}} \\ \mid \mathsf{cons}(hd, tl) \Rightarrow e_{\mathsf{cons}} \end{array} : \tau}\ \textsc{Match}\text{-TLCA}$$

$$\frac{\begin{array}{c}\Sigma(f) = \tau_1^\circ \times \cdots \times \tau_n^\circ \to \tau_{k+1} \\ D \vdash \tau' \equiv \tau'' \\ \mathsf{True};\ x_1 : \tau_1^\circ, \ldots, x_k : \tau_k^\circ\ \vdash_\Sigma e_1 : \tau_{k+1} \qquad D;\ \Gamma\ \vdash_\Sigma e_2 : \tau''\end{array}}{D;\ \Gamma\ \vdash_\Sigma \mathsf{letfun}\ f(x_1, \ldots, x_k) = e_1\ \mathsf{in}\ e_2 : \tau'}\ \textsc{LetFun}$$

The sugared version :

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

In the generic function application rule below if $\tau_i^\circ = \mathsf{L}_{n_{i1}^\circ}(\ldots \mathsf{L}_{n_{il_i}^\circ}(\mathtt{Int})\ldots)$ then $D \vdash \tau_i' \equiv \mathsf{L}_{p_{i1}}(\ldots \mathsf{L}_{p_{il_i'}}(\mathtt{Int})\ldots)$, for some polynomials p_{ij}. Similarly, if $\tau_i^\circ = \mathsf{L}_{n_{i1}^\circ}(\ldots \mathsf{L}_{n_{il_i}^\circ}(\alpha_i)\ldots)$ then $D \vdash \tau_i' \equiv \mathsf{L}_{p_{i1}}(\ldots \mathsf{L}_{p_{il_i'}}(\tau_i'')\ldots)$ for some τ_i''.

Strictly speaking, n_{ij}^o are meta-variables for size variables. They are turned into size variables when considered for the rule for some fixed function. For instance, for the "vector multiplication" above n_{11}^o is n and n_{21}^o is n. Equations C are imposed by syntactical equality of size variables. For instance, when the "vector multiplication" function rule is applied on actual arguments of type $\mathsf{L}_{p_1}(\mathtt{Int})$ and $\mathsf{L}_{p_2}(\mathtt{Int})$, then the set of equations C contains $p_1 = p_2$ (and only such equations).

Similarly, when two arguments contain the same *type* variable, say α lile in the type for "constructor" $\alpha \times \mathsf{L}_p(\alpha) \to \mathsf{L}_{p+1}(\alpha)$, then the correspinding equivalence of types for the actual parameters must be added to E (and E contains only such equivalences). That is, for instance, if $hd : \tau''$ and $tl : \mathsf{L}_{n^2+15}(\tau')$, then to call "constructor" on them one needs $D \vdash \tau'' \equiv \tau'$.

$$\frac{\begin{array}{c}\Sigma(f) = \tau_1^\circ \times \ldots \times \tau_k^\circ \to \tau_{k+1} \\ D \vdash \tau_i' \equiv \mathsf{L}_{p_{i1}}(\ldots \mathsf{L}_{p_{il_i'}}(\mathtt{Int}[\tau_i''])\ldots) \\ D \vdash \tau_{k+1}' \equiv *(\tau_{k+1}) \\ D \vdash C \\ D \vdash E\end{array}}{D;\ \Gamma, x_1 : {\tau_1}', \ldots, x_k : {\tau_n}'\ \vdash_\Sigma f(x_1, \ldots, x_k) : {\tau_{k+1}}'}\ \textsc{FunApp}$$

The sugared version of this rule has the form:

$$\frac{\begin{array}{c}\Sigma(f) = \tau_1^\circ \times \ldots \times \tau_k^\circ \to \tau_{k+1} \\ D \vdash \tau_i' = \mathsf{L}_{p_{i1}}(\ldots \mathsf{L}_{p_{il_i'}}(\mathtt{Int}[\tau_i''])\ldots),\ \text{canonical} \\ D \vdash \tau_{k+1}' = \mathit{canon}(*(\tau_{k+1})) \\ D \vdash C \\ D \vdash E\end{array}}{D;\ \Gamma, x_1 : {\tau_1}', \ldots, x_k : {\tau_n}'\ \vdash_\Sigma f(x_1, \ldots, x_k) : {\tau_{k+1}}'}\ \textsc{FunApp}$$

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

ToDo: In the soundness we need a canonized version! We finish this section with the following two lemmas.

Lemma 2. *For any derivable judgment $D;\ \Gamma\ \vdash_\Sigma e:\tau$ the set $FVS(\Gamma)$ of the free size variables contains the set $FVS(\tau)$ of the free size variables of type τ.*

Proof. The lemma is proved by induction on the (height of the) derivation tree for $D;\ \Gamma\ \vdash_\Sigma e:\tau$.

Axioms If $D;\ \Gamma\ \vdash_\Sigma e:\tau$ is an instance of one of the axioms of the system (IConst, Var, Nil, Cons), then $FVS(\tau) \subseteq FVS(\Gamma)$ holds trivially.

FunApp For $\Sigma(f) = \tau_1^\circ \times \ldots \times \ldots \times \tau_k^\circ \rightarrow \tau_{k+1}$ it holds that $FVS(\tau_{k+1}) \subseteq FVS(\tau_1^\circ) \cup \ldots \cup FVS(\tau_k^\circ)$. From the rule it follows

$$\begin{aligned}&FVS(\tau) = FVS(*(\tau_{k+1})) \subseteq FVS(*(\tau_1^\circ)) \cup \ldots \cup FVS(*(\tau_k^\circ)) = \\ &= FVS(\tau_1) \cup \ldots \cup FVS(\tau_k) \subseteq FVS(\Gamma).\end{aligned}$$

LetFun From the induction hypothesis on the type derivation for e_2 we have that $FVS(\tau) \subseteq FVS(\Gamma)$.

Let In this case $e = \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2$ for some x, e_1, e_2 and we have $D;\ \Gamma\ \vdash_\Sigma e_1 : \tau'$, $x \notin dom(\Gamma)$ and $D;\ \Gamma,\ x:\tau'\ \vdash_\Sigma e_2:\tau$ for some τ'. By induction we have $FVS(\tau') \subseteq FVS(\Gamma)$ and $FVS(\tau) \subseteq FVS(\Gamma \cup \{x:\tau'\})$. Hence, $FVS(\tau) \subseteq FVS(\Gamma)$.

Match Then, $e = \mathsf{match}\ x\ \mathsf{with}\ \mid \mathsf{nil} \Rightarrow e_{\mathsf{nil}} \mid \mathsf{cons}(hd, tl) \Rightarrow e_{\mathsf{cons}}$ for some x, hd, tl, e_{nil}, and e_{cons}. We also have $x : \mathsf{L}_p(\tau') \in \Gamma$ for some, τ', p, and we know $hd, tl \notin dom(\Gamma)$. By induction we have $FVS(\tau) \subseteq FVS(\Gamma)$ if $p = 0$, so that case is proved. For the cons-branch one has $FVS(\tau) \subseteq FVS(\Gamma \cup \{hd:\tau', tl:\mathsf{L}_{p-1}(\tau')\})$. Because $FVS(\tau') \subseteq FVS(\mathsf{L}_p(\tau'))$ and $FVS(\mathsf{L}_{p-1}(\tau')) \subseteq FVS(\mathsf{L}_p(\tau'))$ and $x : \mathsf{L}_p(\tau') \in \Gamma$ we have $FVS(\Gamma \cup \{hd:\tau', tl:\mathsf{L}_{p-1}(\tau')\}) = FVS(\Gamma)$ and, hence, also $FVS(\tau) \subseteq FVS(\Gamma)$ in this case.

If Then $e = \mathsf{if}\ x\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2$ for some x, e_1, and e_2. By induction we have $FVS(\tau) \subseteq FVS(\Gamma)$. □

Lemma 3 (All free variables in a typed expression have a type). *If $D;\ \Gamma\ \vdash_\Sigma e:\tau$ is derivable in the type system, then $FV(e) \subseteq dom(\Gamma)$.*

Proof. The lemma is proved by induction over the derivation tree for $D;\ \Gamma\ \vdash_\Sigma e:\tau$.

Axioms If $D;\ \Gamma\ \vdash_\Sigma e:\tau$ is an instance of one of the axioms of the system (IConst, Var, Nil, Cons, FunApp), then $FV(e) \subseteq dom(\Gamma)$ holds trivially.

If Then $e = \mathsf{if}\ x\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2$ for some x, e_1, and e_2. By induction we have $FV(e_1) \subseteq \Gamma$ and $FV(e_2) \subseteq \Gamma$ and therefore $FV(e_1) \cup FV(e_2) \subseteq \Gamma$.

Let Then $e = \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2$ for some x, e_1, e_2 and we have $D;\ \Gamma\ \vdash_\Sigma e_1 : \tau'$, $x \notin \mathit{dom}(\Gamma)$ and $D;\ \Gamma,\ x : \tau'\ \ \vdash_\Sigma e_2 : \tau$ for some τ'. By induction we have $FV(e_1) \subseteq \mathit{dom}(\Gamma)$ and $FV(e_2) \subseteq \mathit{dom}(\Gamma,\ x : \tau')$. Therefore, $FV(e) = FV(e_1) \cup (FV(e_2) \setminus \{x\}) \subseteq \mathit{dom}(\Gamma)$.

LetFun By the rule and induction hypothesis $FVS(\mathsf{letfun}\ f((x_1, \ldots, x_n)) = e_1\ \mathsf{in}\ e_2) = FVS(e_2) \subseteq \mathit{dom}(\Gamma)$.

Match Then $e = \mathsf{match}\ x\ \mathsf{with}\ \mid \mathsf{nil} \Rightarrow e_{\mathsf{nil}} \mid \mathsf{cons}(hd, tl) \Rightarrow e_{\mathsf{cons}}$ for some x, hd, tl, e_{nil}, and e_{cons}, $\Gamma = \Gamma' \cup x : \mathsf{L}_p(\tau')$ for some Γ', τ', and $hd, tl \notin \mathit{dom}(\Gamma)$. By induction we have $FV(e_{\mathsf{nil}}) \subseteq \mathit{dom}(\Gamma')$ and $FV(e_{\mathsf{cons}}) \subseteq \mathit{dom}(\Gamma') \cup \{x : \mathsf{L}_p(\tau), hd : \tau', tl : \mathsf{L}_{p-1}(\tau)\}$. Then $FV(e) = FV(e_{\mathsf{nil}}) \cup (FV(e_{\mathsf{cons}}) \setminus \{hd, tl\}) \subseteq \mathit{dom}(\Gamma',\ x : \mathsf{L}_p(\tau))$.

$\square$

3.2 Type-checking algorithm underlying the TLCA paper

Let one need to type-check

$$D;\ \Gamma\ \vdash_\Sigma e : \tau.$$

Pre-requisists: underlying type (on the involved types with erased size annotations) is accepted by a standard type-checking procedure.

The type-checking algorithm is inductive on e.

- $e = c$ (not the expression in a let-binding). Then, since underlying type is succesfully checked, τ must be `Int`.
- $e = x$ (not the expression in a let-binding). Then, since underlying type is succesfully checked, Γ is of the form $\Gamma',\ x : \tau'$ for some τ', and underlying types of τ and τ' are the same. The typing is accepted if and only of $D \vdash \tau \equiv \tau'$ (which is the same as $D \vdash \tau^c = \tau'^c$).
- $e = \mathsf{nil}$ (not the expression in a let-binding). The typing is accepted if and only if $D \vdash \tau \equiv \mathsf{L}_0(\tau_0)$, that is $\tau = \mathsf{L}_p(\tau')$ for some τ' and $D \vdash p = 0$.
- $e = \mathsf{cons}(hd, tl)$ (not the expression in a let-binding). The typing is accpeted if and only if $D \vdash \Gamma(tl) \equiv \mathsf{L}_p(\Gamma(hd))$ for some p and $D \vdash \tau \equiv \mathsf{L}_{p+1}(\Gamma(hd))$.
- $e = \mathsf{f}(x_1, \ldots, x_k)$ (not the expression in a let-binding), and $\Sigma(f) = \tau_1^\circ \times \ldots \times \tau_k^\circ \to \tau_{k+1}$, with $\tau_i^\circ = \mathsf{L}_{n_{i1}^\circ}(\ldots \mathsf{L}_{n_{il_i}^\circ}(\alpha_i) \ldots)$, and $x_i : \tau_i'$ in Γ. Then τ_i' must be of the form $\mathsf{L}_{p_{i1}}(\ldots \mathsf{L}_{p_{il_i'}}(\tau_i'') \ldots)$ for some τ_i''. Accept the type, if and only if $D \vdash *(\tau_{k+1}) \equiv \tau$, $D \vdash C$ and $D \vdash E$ hold. Here $*$ is the instantiation n_{ij}^0 with p_{ij} and for C see the definition of the FUN rule.
- $e = \mathsf{if}\ x\ \mathsf{then}\ e_t\ \mathsf{else}\ e_f$. Accept the typing if and only if $D;\ \Gamma\ \vdash_\Sigma e_t : \tau$ and $D;\ \Gamma\ \vdash_\Sigma e_f : \tau$ are accepted.
- $e = \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2$. Since e_1 may be only a basic expression one immediately instantiates the fresh type variable $?\,\tau_x$ in the let-rule. The typing is accpeted if and only if

  $$D;\ \Gamma, x : \mathit{instantiated}(?\,\tau_x)\ \vdash_\Sigma e_2 : \tau$$

  is accepted and:

  - $e_1 = c$: $\mathit{instantiated}(?\,\tau_x) = $ `Int`,

  - $e_1 = y$: $\mathit{instantiated}(?\,\tau_x) = \Gamma(y)$,
  - $e_1 = \mathsf{nil}$: $\mathit{instantiated}(?\,\tau_x) = \mathsf{L}_0(\ldots \mathsf{L}_0([\mathtt{Int}\ [\alpha]]) \ldots)$, where nestedness of the list is reconstructed by the underlying type-checker,
  - $e_1 = \mathsf{cons}(hd, tl)$: $\mathit{instantiated}(?\,\tau_x) = \mathsf{L}_{p+1}(\Gamma(hd))$, and $D \vdash \Gamma(tl) \equiv \mathsf{L}_p(\Gamma(hd))$ for some p
  - $e_1 = \mathsf{f}(x_1, \ldots, x_k)$: $\mathit{instantiated}(?\,\tau_x) = *(\tau_{k+1})$, where $*$ is the insatntiation of formal-parameters' types with the actual-arguments' types, and $D \vdash C$, $D \vdash E$ hold.

– $e = \mathsf{match}\ x\ \mathsf{with} \mid \mathsf{nil} \Rightarrow e_{\mathsf{nil}}$.
$\qquad\qquad\qquad\qquad \mid \mathsf{cons}(hd, tl) \Rightarrow e_{\mathsf{cons}}$
Then $\Gamma(x) = \mathsf{L}_p(\tau')$ for some τ', p. Accept the typing if and only if $p = 0$, $D;\ \Gamma\ \vdash_\Sigma e_{\mathsf{nil}} : \tau$ and and $D;\ \Gamma,\ hd : \tau',\ tl : \mathsf{L}_{p-1}(\tau')\ \vdash_\Sigma e_f : \tau$ are accepted.

– $e = \mathsf{letfun}\ f(x_1, \ldots, x_k) = e_1\ \mathsf{in}\ e_2$. Accept the typing if and only if

$$\mathsf{True};\ x_1 : \tau_1^\circ, \ldots, x_k : \tau_k^\circ\ \vdash_\Sigma e_1\ :\ \tau_{k+1}$$

and

$$D;\ \Gamma\ \vdash_\Sigma e_2\ :\ \tau$$

are accepted, where $\Sigma(f) = \tau_1^\circ \times \cdots \times \tau_n^\circ \to \tau_{k+1}$.

3.3 Examples of type checking

In this section, some examples of type derivations of typable functions will be given. The functions are: append, multiple copying, pairs (used for Cartesian product), cprod (Cartesian product), sqdiff (quadratic difference), and mmaux (matrix multiplication).

Append Consider the following definition of the append function:

$$\begin{array}{l} \mathsf{append}\ (x, y) = \\ \quad \mathsf{match}\ x\ \mathsf{with} \mid \mathsf{nil} \Rightarrow y \\ \qquad\qquad\qquad \mid \mathsf{cons}(h, t) \Rightarrow \mathsf{let}\ z = (\mathsf{append}(t, y))\ \mathsf{in}\ \mathsf{cons}(h, z) \end{array}$$

According to the LetFun-rule, checking that append has the type $\mathsf{L}_n(\alpha) \times \mathsf{L}_m(\alpha) \to \mathsf{L}_{n+m}(\alpha)$ means deriving $x : \mathsf{L}_n(\alpha), y : \mathsf{L}_n(\alpha)\ \vdash_\Sigma e_{\mathsf{append}} : \mathsf{L}_{n+m}(\alpha)$ where $\Sigma(\mathsf{append}) = \mathsf{L}_{n'}(\alpha) \times \mathsf{L}_{m'}(\alpha) \to \mathsf{L}_{n'+m'}(\alpha)$ and e_{append} is the function body. Applying the Match-rule yields two branches to be proven:

Nil: $\quad n = 0;\ y : \mathsf{L}_m(\alpha)\ \vdash_\Sigma y : \mathsf{L}_{n+m}(\alpha)$
Cons: $h : \alpha,\ x : \mathsf{L}_n(\alpha),\ t : \mathsf{L}_{n-1}(\alpha),\ y : \mathsf{L}_m(\alpha)\ \vdash_\Sigma$
$\qquad\qquad \mathsf{let}\ z = (\mathsf{append}\ t\ y)\ \mathsf{in}\ \mathsf{cons}(h, z) : \mathsf{L}_{n+m}(\alpha)$

Applying the Var-rule to the Nil-branch gives $n = 0 \vdash n + m = m$, which is trivially true. The Cons-branch is proven by applying the Let-rule. This yields two proof obligations:

$\textsc{Binding}$: $t\!:\!\mathsf{L}_{n-1}(\alpha),\ y\!:\!\mathsf{L}_m(\alpha)\ \vdash_\Sigma \mathsf{append}(t\ y)\!:\!\tau$
$\textsc{Expr}$: $h\!:\!\alpha,\ z\!:\!\tau\ \vdash_\Sigma \mathsf{cons}(h,z)\!:\!\mathsf{L}_{n+m}(\alpha)$

The Binding-branch is proven by applying the FunApp-rule, which instantiates τ with $\mathsf{L}_{(n-1)+m}(\alpha)$ due to the tautology $\vdash n-1+m = n-1+m$. Proving the Expr-branch only requires applying the Cons-rule giving $\vdash n+m = (n-1+m)+1$, which is also true.

Because for every syntactic construction there is only one typing rule that is applicable, type checking is straightforward.

Multiple Copying Now we consider a program which takes a pair of lists l_1, l_2 and copies the first argument $|l_2|$ times, where $|l_2|$ is the length of l_2:

```
f (l₁, l₂) =
  match l₂ with | nil ⇒ nil
                | cons(h, t) ⇒ let x = f(l₁, t) in append(l₁, x)
```

We will check the body e_f of f within the signature

$$\Sigma = \{\mathsf{append} :\ \mathsf{L}_n(\alpha)\times\mathsf{L}_m(\alpha)\ \rightarrow\ \mathsf{L}_{n+m}(\alpha),\ \mathsf{f} :\ \mathsf{L}_n(\alpha)\times\mathsf{L}_m(\alpha)\ \rightarrow\ \mathsf{L}_{n*m}(\alpha)\}$$

First note that any pattern matching generates an obvious case split – the proof tree splits into two parts, one of which is for zero-size case, another – for nonzero. In our example the split happens w.r.t size variable m. The nil-branch is:

$$\dfrac{\dfrac{m=0\ \vdash\ n*m=0}{m=0;\ l_1\!:\!\mathsf{L}_n(\alpha),\ l_2\!:\!\mathsf{L}_m(\alpha)\ \vdash_\Sigma \mathsf{nil}\!:\!\mathsf{L}_{n*m}(\alpha)}\ \textsc{Nil}}{l_1\!:\!\mathsf{L}_n(\alpha),\ l_2\!:\!\mathsf{L}_m(\alpha)\ \vdash_\Sigma e_f\!:\!\mathsf{L}_{n*m}(\alpha)}\ \textsc{Match}$$

The cons-branch is:

$$\dfrac{\dfrac{\dfrac{\vdash p = n*(m-1)}{l_1\!:\!\mathsf{L}_n(\alpha),\ tl\!:\!\mathsf{L}_{m-1}(\alpha)\ \vdash_\Sigma \mathsf{f}(l_1,tl)\!:\!\mathsf{L}_p(\alpha)}\ \textsc{Fun}\quad (\star)}{\left.\begin{array}{r} l_1\!:\!\mathsf{L}_n(\alpha),\ l_2\!:\!\mathsf{L}_m(\alpha),\\ h\!:\!\alpha, t\!:\!\mathsf{L}_{m-1}(\alpha)\end{array}\right\}\ \vdash_\Sigma \left\{\begin{array}{l}\mathsf{let}\ x=\mathsf{f}((l_1,tl))\\ \mathsf{in\ append}(l_1,\ x)\end{array}\!:\!\mathsf{L}_{n*m}(\alpha)\right\}}\ \textsc{Let}}{l_1\!:\!\mathsf{L}_n(\alpha),\ l_2\!:\!\mathsf{L}_m(\alpha)\ \vdash_\Sigma e_f\!:\!\mathsf{L}_{n*m}(\alpha)}\ \textsc{Match}$$

where $\star$ is:

$$\dfrac{\vdash n*m = n+n*(m-1)}{\left.\begin{array}{r} l_1\!:\!\mathsf{L}_n(\alpha),\\ x\!:\!\mathsf{L}_{n*(m-1)}(\alpha)\end{array}\right\}\ \vdash_\Sigma \mathsf{append}(l_1,x)\!:\!\mathsf{L}_{n*m}(\alpha)}\ \textsc{Fun}$$

Pairs The function pairs is used to define the Cartesian product of two lists. The function, given an element and a list of the same type, creates the list of two-element lists. A pair of the output list consists of the input element and an element of the input list. For instance, $\mathsf{pairs}(1,[1,2,3]) = [[1,1],[1,2],[1,3]]$.

```
pairs $x$ $y$ =
  match $y$ with | nil $\Rightarrow$ nil
                 | cons($hd$, $tl$) $\Rightarrow$ let $z$ = cons($hd$, nil)
                                      in let $z'$ = cons($x$, $z$)
                                        in let $z''$ = pairs($x$, $tl$)
                                          in cons($z'$, $z''$)
```

We type-check $\mathsf{pairs} :\ x\!:\!\alpha \times y\!:\!\mathsf{L}_n(\alpha) \longrightarrow \mathsf{L}_n(\mathsf{L}_2(\alpha))$. The $\textsc{Nil}$-branch is proved by:

$$\dfrac{\dfrac{n = 0 \vdash n = 0}{n = 0;\ x\!:\!\alpha,\ y\!:\!\mathsf{L}_n(\alpha)\ \vdash_\Sigma \mathsf{nil}\!:\!\mathsf{L}_n(\mathsf{L}_2(\alpha))}\ \textsc{Nil}}{x\!:\!\alpha,\ y\!:\!\mathsf{L}_n(\alpha)\ \vdash_\Sigma e_{\mathsf{pairs}}\!:\!\mathsf{L}_n(\mathsf{L}_2(\alpha))}\ \textsc{Match}$$

The (sugared) $\textsc{Cons}$-branch is proved by:

$$\dfrac{\dfrac{\dfrac{\vdash n = 0 + 1}{hd\!:\!\alpha\ \vdash_\Sigma \mathsf{cons}(hd,\mathsf{nil})\!:\!\mathsf{L}_{?n}(\alpha)}\ \textsc{Cons} \qquad (\star)}{x\!:\!\alpha,\ hd\!:\!\alpha,\ y\!:\!\mathsf{L}_n(\alpha),\ t\!:\!\mathsf{L}_{n-1}(\alpha)\ \vdash_\Sigma \mathsf{let}\ z = ...\ \mathsf{in}\ ...\!:\!\mathsf{L}_n(\mathsf{L}_2(\alpha))}\ \textsc{Let}}{x\!:\!\alpha,\ y\!:\!\mathsf{L}_n(\alpha)\ \vdash_\Sigma e_{\mathsf{pairs}}\!:\!\mathsf{L}_n(\mathsf{L}_2(\alpha))}\ \textsc{Match}$$

where $(\star)$ is

$$\dfrac{\dfrac{\vdash m = 1 + 1}{z\!:\!\mathsf{L}_1(\alpha),\ x\!:\!\alpha\ \vdash_\Sigma \mathsf{cons}(x,z)\!:\!\mathsf{L}_m(\alpha)}\ \textsc{Cons} \qquad (\star\star)}{z\!:\!\mathsf{L}_1(\alpha),\ x\!:\!\alpha,\ y\!:\!\mathsf{L}_n(\alpha),\ tl\!:\!\mathsf{L}_{n-1}(\alpha)\ \vdash_\Sigma \mathsf{let}\ z' = ...\ \mathsf{in}\ ...\!:\!\mathsf{L}_n(\mathsf{L}_2(\alpha))}\ \textsc{Let}$$

and $(\star\star)$ is

$$\dfrac{\dfrac{\vdash p_1 = 2\ \wedge\ p_2 := n - 1}{x\!:\!\alpha,\quad tl\!:\!\mathsf{L}_{n-1}(\alpha)\ \vdash_\Sigma (\mathsf{pairs}(x,tl)\!:\!\mathsf{L}_{p_2}(\mathsf{L}_{p_1}(\alpha))}\ \text{FA} \qquad (\star\star\star)}{z'\!:\!\mathsf{L}_2(\alpha),\ x\!:\!\alpha,\ t\!:\!\mathsf{L}_{n-1}(\alpha)\ \vdash_\Sigma \mathsf{let}\ z'' = ...\ \mathsf{in}\ ...\!:\!\mathsf{L}_n(\mathsf{L}_2(\alpha))}\ \textsc{Let}$$

with $(\star\star\star)$:

$$\dfrac{\vdash n = (n-1) + 1}{z''\!:\!\mathsf{L}_{n-1}(\mathsf{L}_2(\alpha)), z'\!:\!\mathsf{L}_2(\alpha)\ \vdash_\Sigma \mathsf{cons}(z',z'')\!:\!\mathsf{L}_n(\mathsf{L}_2(\alpha))}\ \textsc{Cons}$$

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

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

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

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

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

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

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

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

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

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

From the applications of the FunApp-rule to Bind-z1 and Bind-z2 it follows that τ_1 should be $\mathsf{L}_m(\mathsf{L}_2(\alpha))$ and τ_2 should be $\mathsf{L}_{(n-1)*m}(\mathsf{L}_2(\alpha))$. Lastly, applying the FunApp-rule to Body yields the proof obligation $\vdash (n-1)*m+m = n*m$, which is true in the axiomatics.

Quadratic difference Now we will consider the function whose output-size polynomial contains subtraction. The polynomial is $(x-y)^2 = x^2 - 2xy + y^2$. Given two lists, the functions "subtracts" elements from lists simultaneously, till one of the lists is empty. The function return the Cartesian product of the rest list with itself.

```
sqdiff (x, y) =
   match x with | nil ⇒ cprod(y, y)
                | cons(hd, tl) ⇒ match y with | nil ⇒ cprod(x, x)
                                              | cons(hd′, tl′) ⇒ sqdiff (tl, tl′)
```

We want to type-check $\mathsf{sqdiff} : \mathsf{L}_n(\alpha) \times \mathsf{L}_m(\alpha) \longrightarrow \mathsf{L}_{(n^2+m^2-2*n*m)}(\mathsf{L}_2(\alpha))$ using:

$$\begin{aligned} \Sigma(\mathsf{sqdiff}) &= \mathsf{L}_n(\alpha) \times \mathsf{L}_m(\alpha) \to \mathsf{L}_{n^2+m^2-2*n*m}(\mathsf{L}_2(\alpha)) \\ \Sigma(\mathsf{cprod}) &= \mathsf{L}_n(\alpha) \times \mathsf{L}_m(\alpha) \to \mathsf{L}_{n*m}(\mathsf{L}_2(\alpha)) \end{aligned}$$

After applying the LetFun and Match rules in the, by now, familiar way, we are left with two Nil-branches and one Cons-branch. The first nil-branch is proved by:

$$\dfrac{\dfrac{n = 0 \vdash 2 = 2 \wedge n^2 + m^2 - 2*n*m = m^2}{n = 0;\ x\!:\mathsf{L}_n(\alpha),\ y\!:\mathsf{L}_m(\alpha)\ \vdash_\Sigma \mathsf{cprod}(y,y)\!:\mathsf{L}_{(n^2+m^2-2*n*m)}(\mathsf{L}_2(\alpha))}\ \textsc{Fun}}{x\!:\mathsf{L}_n(\alpha),\ y\!:\mathsf{L}_m(\alpha)\ \vdash_\Sigma e_{\mathsf{sqdiff}}\!:\mathsf{L}_{(n^2+m^2-2*n*m)}(\mathsf{L}_2(\alpha))}\ \textsc{Match}$$

The second nil-branch is proved by:

$$\dfrac{\dfrac{m = 0\ \vdash 2 = 2 \wedge n^2 + m^2 - 2*n*m = n^2}{\begin{array}{l} m = 0;\ x\!:\mathsf{L}_n(\alpha), y\!:\mathsf{L}_m(\alpha), \\ hd\!:\alpha,\ tl\!:\mathsf{L}_{n-1}(\alpha) \end{array} \vdash_\Sigma \mathsf{cprod}(x,x)\!:\mathsf{L}_{(n^2+m^2-2*n*m)}(\mathsf{L}_2(\alpha))}\ \textsc{FunApp}}{\begin{array}{l} x\!:\mathsf{L}_n(\alpha), y\!:\mathsf{L}_m(\alpha), \\ hd\!:\alpha,\ t\!:\mathsf{L}_{n-1}(\alpha) \end{array} \vdash_\Sigma \mathsf{match...}\!:\mathsf{L}_{(n^2+m^2-2*n*m)}(\mathsf{L}_2(\alpha))}\ \textsc{Match}$$

The recursive part of the derivation tree is

$$\dfrac{\dfrac{\vdash\ n^2 + m^2 - 2*n*m = (n-1)^2 + (m-1)^2 - 2*(n-1)*(m-1)}{\begin{array}{l} x\!:\mathsf{L}_n(\alpha), y\!:\mathsf{L}_m(\alpha) \\ hd\!:\alpha,\ tl\!:\mathsf{L}_{n-1}(\alpha), \\ hd'\!:\alpha,\ tl'\!:\mathsf{L}_{m-1}(\alpha) \end{array} \vdash_\Sigma \mathsf{sqdiff}(tl, tl')\!:\mathsf{L}_{(n^2+m^2-2*n*m)}(\mathsf{L}_2(\alpha))}\ \textsc{FunApp}}{\begin{array}{l} x\!:\mathsf{L}_n(\alpha), y\!:\mathsf{L}_m(\alpha), \\ hd\!:\alpha,\ t\!:\mathsf{L}_{n-1}(\alpha) \end{array} \vdash_\Sigma \mathsf{match...}\!:\mathsf{L}_{(n^2+m^2-2*n*m)}(\mathsf{L}_2(\alpha))}\ \textsc{Match}$$

Matrix multiplication For the matrix multiplication we need auxiliary functions The function transpose transposes a matrix; the columns become rows, and the rows become columns. The function inprod computes the scalar product of two integer vectors. We define the matrix multiplications mmult, that leaves the interesting work for newrow, which calculates a single row of the result matrix.

$$\begin{array}{lll}
\mathsf{letfun\ newrow}(x,y) & = \mathsf{match}\ y\ \mathsf{with} & \mid \mathsf{nil} \Rightarrow \mathsf{nil} \\
 & & \mid \mathsf{cons}(hd,tl) \Rightarrow \mathsf{let}\ z \ \ = \mathsf{inprod}(x,hd) \\
 & & \qquad\qquad\quad\ \mathsf{in\ let}\ z' = \mathsf{newrow}(x,tl) \\
 & & \qquad\qquad\quad\ \mathsf{in\ cons}(z,z') \\
\mathsf{in\ letfun\ mmaux}(x,y) & = \mathsf{match}\ y\ \mathsf{with} & \mid \mathsf{nil} \Rightarrow \mathsf{nil} \\
 & & \mid \mathsf{cons}(hd,tl) \Rightarrow \mathsf{let}\ z \ \ = \mathsf{newrow}(hd,y) \\
 & & \qquad\qquad\quad\ \mathsf{in\ let}\ z' = \mathsf{mmaux}(tl,y) \\
 & & \qquad\qquad\quad\ \mathsf{in\ cons}(z,z') \\
\mathsf{in\ letfun\ mmult}(x,y) & = \mathsf{let}\ z = \mathsf{transpose}(y) & \\
 & \ \ \ \mathsf{in\ mmaux}(x,z) &
\end{array}$$

Σ contains the following types:

$$\begin{array}{ll}
\Sigma(\mathsf{transpose}) & = \mathsf{L}_n(\mathsf{L}_m(\alpha)) \to \mathsf{L}_m(\mathsf{L}_n(\alpha)) \\
\Sigma(\mathsf{inprod}) & = \mathsf{L}_n(\mathtt{Int}) \times \mathsf{L}_n(\mathtt{Int}) \to \mathtt{Int} \\
\Sigma(\mathsf{newrow}) & = \mathsf{L}_n(\mathtt{Int}) \times \mathsf{L}_m(\mathsf{L}_n(\mathtt{Int})) \to \mathsf{L}_m(\mathtt{Int}) \\
\Sigma(\mathsf{mmaux}) & = \mathsf{L}_m(\mathsf{L}_n(\mathtt{Int})) \times \mathsf{L}_{m'}(\mathsf{L}_n(\mathtt{Int})) \to \mathsf{L}_m(\mathsf{L}_{m'}(\mathtt{Int})) \\
\Sigma(\mathsf{mmult}) & = \mathsf{L}_m(\mathsf{L}_n(\mathtt{Int})) \times \mathsf{L}_n(\mathsf{L}_{m'}(\mathtt{Int})) \to \mathsf{L}_m(\mathsf{L}_{m'}(\mathtt{Int}))
\end{array}$$

Now, according to the LetFun-rule, checking that mmaux has the type $\mathsf{L}_m(\mathsf{L}_n(\mathtt{Int}))\times \mathsf{L}_{m'}(\mathsf{L}_n(\mathtt{Int})) \to \mathsf{L}_m(\mathsf{L}_{m'}(\mathtt{Int}))$ means deriving $x:\mathsf{L}_m(\mathsf{L}_n(\mathtt{Int})), y:\mathsf{L}_{m'}(\mathsf{L}_n(\mathtt{Int})) \vdash_\Sigma e_{\mathsf{mmaux}}:\mathsf{L}_m(\mathsf{L}_{m'}(\mathtt{Int}))$ where e_{mmaux} is the function body. Applying the Match-rule yields two branches to be proven:

$$\begin{array}{ll}
\textsc{Nil}: & m = 0 \ \vdash_\Sigma \mathsf{nil}:\mathsf{L}_m(\mathsf{L}_{m'}(\alpha)) \\
\textsc{Cons}: & hd:\mathsf{L}_n(\mathtt{Int}),\ tl:\mathsf{L}_{m-1}(\mathsf{L}_n(\mathtt{Int})),\ y:\mathsf{L}_{m'}(\mathsf{L}_n(\mathtt{Int})) \ \vdash_\Sigma \\
 & \quad \mathsf{let}\ z \ \ = \mathsf{newrow}(hd,y):\mathsf{L}_m(\mathsf{L}_{m'}(\alpha)) \\
 & \quad \mathsf{in\ let}\ z' = \mathsf{mmaux}(tl,y) \\
 & \quad \mathsf{in\ cons}(z,z')
\end{array}$$

Applying the Nil-rule to the Nil-branch gives $m = 0 \vdash m = 0$, which is trivially true. The Cons-branch is proven by applying the Let-rule twice. This yields three proof obligations:

$$\begin{array}{ll}
\textsc{Bind-z}: & hd:\mathsf{L}_n(\mathtt{Int}),\ y:\mathsf{L}_p(\mathsf{L}_n(\mathtt{Int})) \ \vdash_\Sigma \mathsf{newrow}(hd,y):\tau_1 \\
\textsc{Bind-z'}: & tl:\mathsf{L}_{m-1}(\mathsf{L}_n(\mathtt{Int})),\ y:\mathsf{L}_p(\mathsf{L}_n(\mathtt{Int})) \ \vdash_\Sigma \mathsf{mmaux}(tl,y):\tau_2 \\
\textsc{Body}: & z:\tau_1, z':\tau_2 \ \vdash_\Sigma \mathsf{cons}(r,t):\mathsf{L}_m(\mathsf{L}_{m'}(\alpha))
\end{array}$$

Function application instantiates τ_1 and τ_2 with $\mathsf{L}_m(\mathtt{Int})$ and $\mathsf{L}_{m-1}(\mathsf{L}_{m'}(\alpha))$.

4 Decidability Issues for Type Checking

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

4.1 Type checking in general is undecidable

We show that the existence of a procedure that may check all possible entailments at the end of type checking is reduced to Hilbert's tenth problem: whether there exists a general procedure that given a polynomial with integer coefficients decides if this polynomial has natural roots or not.[2] Matiyasevich [MJ91] has shown that such a procedure does not exist. This means that type checking, in the general case, is undecidable as well.

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

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

$$\begin{aligned}\mathsf{let}\ x = f_0(x_1, \ldots, x_k)\ \mathsf{in}\ \mathsf{match}\ x\ \mathsf{with}\ &|\ \mathsf{nil} \Rightarrow f_1(x_1, \ldots, x_k)\\ &|\ \mathsf{cons}(hd, tl) \Rightarrow f_2(x_1, \ldots, x_k)\end{aligned}$$

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

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

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

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

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

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

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

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, 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.

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

4.2 Syntactical condition for decidability

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

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

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

Theorem 1. *Let a program expression e satisfy the refined grammar, and its type to check is $\tau_1^o \times \ldots \times \tau_k^o \longrightarrow \tau$.*

Then at the end of type checking procedure for $\emptyset;\ x_1 : \tau_1^o, \ldots, x_k : \tau_k^o \vdash_{\emptyset} e : \tau$ one has to check entailments of the form

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

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

The proof of the theorem is based on the following lemma.

Lemma 4. *Let a variable $x : \mathsf{L}_{q_m}(\ldots \mathsf{L}_{q_1}(\tau_b)\ldots)$ be pattern-matched in an expression e, where τ_b is either* `Int` *or a type variable. Let the type-checking tree for $\emptyset;\ x_1 : \tau_1^o, \ldots, x_k : \tau_k^o \ \vdash_\emptyset e : \tau$ be considered. Then $q_i = n_i - c_i$ for some size variable $n_i \in FVS(\tau_1^o \times \ldots \times \tau_k^o)$ and some constant c_i.*

Proof. (Of the Lemma.) By induction on the length of the path that leads from the root of the tree to the pattern-matching point.

- The path is empty, that is

$$e = \mathsf{match}\ x\ \mathsf{with}\ \begin{array}{l} \mid \mathsf{nil} \Rightarrow e_1 \\ \mid \mathsf{cons}(hd, tl) \Rightarrow e_2 \end{array}$$

  for some hd, tl, e_1, e_2.That is x is free in e and $x : \tau_j^o$ for some j. From the definition of τ^o-type $q_i = n_i$ for some size variables n_i.
- Let the path is not empty. Consider two cases.
  - x is free in e. then, as above, $q_i = n_i$ for some size variables n_i.
  - x is not free in e. We use the fact, that the pattern matching may not be a sub-expression of a let-body. Thus, there is y, such that $e' = \mathsf{match}\ y\ \mathsf{with}\ \begin{array}{l} \mid \mathsf{nil} \Rightarrow e_1 \\ \mid \mathsf{cons}(x, tl) \Rightarrow e_2 \end{array}$ or $e' = \mathsf{match}\ y\ \mathsf{with}\ \begin{array}{l} \mid \mathsf{nil} \Rightarrow e_1 \\ \mid \mathsf{cons}(hd, x) \Rightarrow e_2 \end{array}$ are sub-expressions of e. (The pattern matching for x is a sub-expression of e_2 in both cases.) In the first case $y : \mathsf{L}_q(\mathsf{L}_{q_m}(\ldots \mathsf{L}_{q_1}(\tau_b)\ldots))$ for some q. In the second case $y : \mathsf{L}_{q'_m}(\ldots \mathsf{L}_{q_1}(\tau_b)\ldots)$ for $q_m = q'_m - 1$.
    In the first case $q_i = n_i - c_i$, in the second case – the same but $q_m = n_m - (c_m + 1)$, where n_i, c_i are obtained by induction assumption for y.

Proof. (Of the Theorem.) Consider a path in the type checking tree which ends up with some $D \vdash p' = p$, and let an equation $q = 0$ belong to D. It means that in the path there is the nil-branch of the pattern matching for some $x : \mathsf{L}_{q_m}(\ldots \mathsf{L}_{q_1}(\tau_b)\ldots)$, with $q_m = q$. Apply the lemma to get $q = n - c$ for some $n \in FVS(\tau_1^o \times \ldots \times \tau_k^o)$ and some constant c.

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

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

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

and the expression

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

define the same map of lists.

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

5 Weak Type Inference

Here we discuss weak type inference under the syntactical condition defined in the previous section. Weak type inference is defined as follows: *assuming that a function always terminates and is well-typable, find its type.* Since we consider shapely functions, there is a way to reduce weak type inference to type checking using the well-known fact that a finite polynomial is defined by a finite number of points.

For each size polynomial in the output type of a given function, one assumes a hypothesis about the degree and the variables. Then, to obtain the coefficients, the function is run (preferably in a sand-box) as many times as the number of coefficients the polynomial has. This finite number of input-output size pairs defines a system of linear equations, where the unknowns are the coefficients of the polynomial. When (the sizes of the data-structures in) the set of input data satisfies some criteria known from the polynomial interpolation theory [CL87,Lor92], the system has a unique solution. Input sizes that satisfy these criteria, which are nontrivial for multivariate polynomials, can be determined algorithmically. Section 5.1 explains how.

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

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

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

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

n	m	x	y	$\mathsf{cprod}(x, y)$	$p_1(n, m)$	$p_2(n, m)$
0	0	[]	[]	[]	0	?
1	0	[0]	[]	[]	0	?
0	1	[]	[0]	[]	0	?
1	1	[0]	[1]	[[0, 1]]	1	2
2	1	[0, 1]	[2]	[[0, 2], [1, 2]]	2	2
1	2	[0]	[1, 2]	[[0, 1], [0, 2]]	2	2

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

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

The unique solution is $a_{1,1} = 1$ and the rest of coefficients are zero. To verify whether the interpolation is indeed the size polynomial, one checks if cprod : $\mathsf{L}_n(\alpha) \times \mathsf{L}_m(\alpha) \longrightarrow \mathsf{L}_{n*m}(\mathsf{L}_2(\alpha))$. This is the case, as was shown in section 3.3.

As an alternative way of finding the coefficients, one could try to directly solve the systems defined by entailments $D \vdash p = p'$. When the degree is assumed, the unknowns in these systems are the polynomial coefficients. However, the systems are nonlinear in general, see section 5.2. By combining testing with type checking we do not have to solve these nonlinear Diophantine equations anymore.

5.1 Interpolation theory for generating hypotheses

In this section, the conditions under which a set of data points has a unique polynomial interpolation are discussed. The well-known univariate case is used for introduction. The general algorithm is illustrated by a bivariate example.

To define a polynomial $p(x)$ of degree m it suffices to know the values of the polynomial function in $m+1$ *different* points. The set $\big(x_i, p(x_i)\big)$ of pairs of numbers, where $1 \leq i \leq m+1$, determines the system of linear equations w.r.t. the polynomial coefficients $a_0, \ \dots \ , a_m$:

$$a_0 \ + \ a_1\, x_i \ + \ \dots \ + \ a_m\, x_i^m \ = \ p(x_i)$$

for $1 \leq i \leq m+1$. The main matrix of the system is given by:

$$\begin{vmatrix} 1 & x_1 & \dots x_1^{m-1} & x_1^m \\ 1 & x_2 & \dots x_2^{m-1} & x_2^m \\ \dots\dots & & & \\ 1 & x_m & \dots x_m^{m-1} & x_m^m \\ 1 & x_{m+1} & \dots x_{m+1}^{m-1} & x_{m+1}^m \end{vmatrix}$$

Its determinant, the Vandermonde determinant, is equal to $\prod_{i>j}(x_i - x_j)$ and is non-zero for pairwise different points $x_1, \ldots,\ x_{m+1}$. This means that there exists a unique solution for the system of equations.

Similar holds for polynomials of two or more variables. However, the conditions under which multivariate Vandermonde determinants are non-zero are not trivial. They are explored in polynomial interpolation theory [CL87], [Lor92]. In fact, we are looking for the Lagrange interpolation of a complexity function. Since this complexity function is itself a polynomial of the same degree and number of variables as the Lagrange interpolation, the interpolation coincides with the complexity function.

The problem is finding the condition under which a set of data uniquely determines an interpolation of the data to a polynomial. It is stated as follows [CL87]. Let m be the degree, n be the number of variables (dimensionality) and $N^n_m = \binom{m+n}{n}$ be the number of the coefficients. What is needed is the condition on a set of nodes $\{\bar{w}_i\ :\ i = 1, \ldots,\ N^n_m\}$, such that for every data $\{f_i\ :\ i = 1, \ldots,\ N^n_m\}$, there is a unique polynomial $p(\bar{w}) = \Sigma_{0 \leq |j| \leq m} a_j \bar{w}^j$ with total degree m which interpolates the given data at the nodes, that is $p(\bar{w}_i) = f_i$, where $1 \leq i \leq N^n_m$. Here $\bar{w}^j = w_1^{j_1} \ldots w_n^{j_n}$, $|j| = j_1 + \ldots + j_n$ is the usual multivariate notation. The configuration of points which ensures the existence and uniqueness of Lagrange interpolation in general is not trivial.

2 dimensional For the 2-dimensional case of degree m it is stated as follows: there are $m+1$ lines containing $m+1, \ldots, 1$ of the nodes, respectively, and the nodes do not lie on the intersections of the lines. Such a configuration of nodes is presented for parallel lines in figure 1a. The formal definition is as follows:

Definition 1. ***NCA*** *configuration [CL87] for* 2 *variables (*2*-dimensional space): there exist different lines* $\gamma_0, \ldots, \gamma_n$*, such that one poses* $m+1$ *nodes on* γ_m*,* m *nodes on* $\gamma_{m-1} \setminus \gamma_m$*, ... ,* 1 *nodes is placed on* $\gamma_0 \setminus \{\gamma_1 \cup \ldots \cup \gamma_m\}$*.*

Assuming the program terminates on all inputs, there is no problem to find such points for (outer) lists, for example using a triangle of points on parallel lines (figure 1b). However, one must treat with care typings with nested lists, like, for instance, $\mathsf{L}_p(\mathsf{L}_q(\alpha))$. When while testing the outer list is empty, that is $p(x,\ y) = 0$, there is no size value for the inner list (shown by the question marks in the table for cprod). Thus, the points which send p to zero, must be excluded from a testing process. So, one first finds the size polynomial p. There is a finite number of lines $y = i$, where $p(x,\ i) \equiv 0$ and infinitely many other lines (see lemma 5). By the diagonal search one can find as many points $(x,\ y)$ as one wants, such that $p(x, y) \neq 0$ (see, for instance, figure 1c).

Note, that in the first-order signatures we do not consider nested lists with the outer type be a constant zero, like $\mathsf{L}_0(\mathsf{L}_q(\tau))$. This is not a principal type, that is we accept only $\mathsf{L}_0(\alpha)$ for the outer polynomial be a constant zero.

Lemma 5. *A polynomial* $p(x,\ y)$ *of the degree* m *has at most* m *"root" lines,* $y = i$*, such that* $p(x,\ i) \equiv 0$*, or is constant* 0*.*

Proof. Suppose the opposite. Then it is easy to pick up nodes on some $m + 1$ "root" lines, and with these nodes the system of linear equations for the coefficients of p will have the zero-solution, that is all the coefficients of p will be zeros.

The algorithm we are using works not by diagonal search, but uses a limit on the number of the lines $y = i$ and nodes on them that have to be searched at most. Essentially, it just tries to find the triangle shape (as in figure 1b) while skipping all positions that do not give a value for the inner polynomial.

Lemma 6. *Let the result type of a function be* $\mathsf{L}_p(\mathsf{L}_q(\alpha))$ *and* $n = 2$ *be the amount of variables,* m_1 *be the degree of* $p(x, y)$ *and* m_2 *be the degree of* $q(x, y)$*. We want to find test points for* q *at places where* $p(x, y) \neq 0$*. We need to find* $(m_2+1)(m_2+2)/2$ *data points: from* m_2+1 *points lie on the line* γ_{m_2+1} *to 1 point lying on the line* γ_1*. It is sufficient to search the parallel lines* $y = 0, \ldots, y = m_1 + m_2$ *in the square* $[0, \ldots, m_1 + m_2] \times [0, \ldots, m_1 + m_2]$*.*

Proof. We show that in the square $[0, \ldots, m_1 + m_2] \times [0, \ldots, m_1 + m_2]$ there are at least $m_2 + 1$ lines where at least $m_2 + 1$ points do not send p to 0. Indeed, due to lemma 5 there are at most m_1 lines $y = i$ such that $p(x, i) \equiv 0$, so at least $m_2 + 1$ are not "root" lines. All those lines $p(x, i) \not\equiv 0$ have degree m_1, thus each contains at most m_1 nodes (x, i), such that $p(x, i) = 0$. This leaves $m_2 + 1$ non-zero size values.

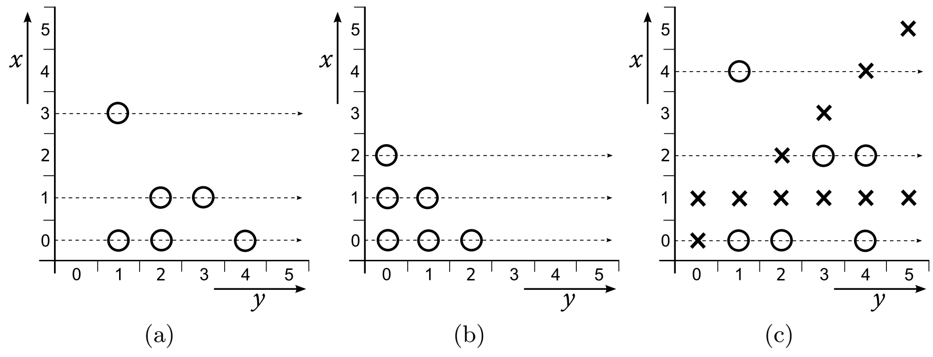

Fig. 1.

Generalization

Definition 2. *The* ***NCA*** *configuration [CL87] for n variables (n-dimensional space) is defined inductively on n.*

Let $\{x_1, \ldots, x_{N^n_m}\}$ *be a set of distinct points in* $\mathcal{R}^n$ *such that there exists* $m + 1$ *hyperplanes* K^n_j*,* $0 \leq j \leq m$ *with*

$$\begin{array}{l} x_{N^n_{m-1}+1}, \ldots, x_{N^n_m} \in K^n_m \\ x_{N^n_{j-1}+1}, \ldots, x_{N^n_j} \in K^n_j \setminus \{K^n_{j+1} \cup \ldots \cup K^n_m\}, \quad \textit{for } 0 \le j \le m-1 \end{array}$$

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

In other words, a hyperplane K^n_j may be viewed as a set, where test points for a polynomial of $n-1$ variable of the degree j lie. There must be $N^{n-1}_j = N^n_j - N^n_{j-1}$ such points.

Thus, in the general algorithm, similarly to lines in a square, parallel hyperplanes in a hypercube are searched.

The 3-dimensional case: condition and solution We consider in more detail the choice of the testing (triples of) lengths for functions with the output type $\mathsf{L}_p(\mathsf{L}_q(\alpha))$, where p and q are polynomials of three variables x, y, z. The procedure is similar tow two-variable case. As in two-dimensional case, m_1 is the degree of p and m_2 is the degree of q.

Lemma 7 (Testing for 3-variable polynomials). *Let $p(x, y, z)$ be not a constant zero. Browsing the integer points in the cube*

$$[0, \ldots, m_1 + m_2] \times [0, \ldots, m_1 + m_2] \times [0, \ldots, m_1 + m_2]$$

*one can find $N^3_{m_2}$ points, which satisfy **NCA** in $\mathcal{R}^3$ and will not be zeros of p.*

Proof. We consider m_1+m_2 planes $z = 0, \ldots m_1+m_2$. Amongst them at most m_1 planes $z = i$ may be zero-planes for $p(x, y, z)$, that is $p(x, y, i) \equiv 0$. (Otherwise $p(x, y, z) \equiv 0$.) Thus, there are at least m_2+1 planes $z = i$, such that $p(x, y, i) \not\equiv 0$. Fix such non zero-plane i and some degree j between 0 and m_2.

As for 2-dimensional case, one shows that on the square $[0, \ldots, m_1 + j] \times [0, \ldots, m_1 + j]$ on the plane $z = i$, one can find N^2_j points that satisfy **NCA** in $\mathcal{R}^2$.

5.2 Direct calculation of coefficients

In weak type inference coefficients of polynomials, at least on the right-hand side of an entailment, are unknown. The task is to find the coefficients, which make two polynomials on the right-hand side equal. For any entailment one equates the coefficients at the monomials of the same degrees. Thus, one obtains the system of Diophantine equations to solve.

Consider, for instance, the ("sugared") code for the function $\mathsf{sqdiff}'(x, y)$

```
match $x$ with | nil ⇒ sqr($y$)
               | cons($hd, tl$) ⇒ match $y$ with | nil ⇒ sqr($x$)
                                                 | cons($hd', tl'$) ⇒ $e$
```

where $\mathsf{sqr}(x)$ may be defined as $\mathsf{sqdiff}(x,\ \mathsf{nil})$ and

$$e = \mathsf{sqdiff}(\mathsf{sqdiff}'(tl,\ y)@x@x@x,\ \mathsf{sqdiff}'(x,\ tl')@y@y@y),$$

and @ denotes `append`. In type inference for this code one must obtain a polynomial of the form

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

from the equation

$$p(x+1, y+1) = \big((p(x, y+1) + 3x) - (p(x+1, y) + 3y)\big)^2.$$

Opening parentheses and comparing coefficients at the corresponding monomials gives the following system of quadratic equations:

$$\begin{cases} a_{2,0} & = (2a_{2,0} - a_{1,1} - 3)^2 \\ a_{0,2} & = (2a_{0,2} - a_{1,1} - 3)^2 \\ a_{1,1} & = -2(2a_{0,2} - a_{1,1} - 3)(2a_{2,0} - a_{1,1} - 3) \\ a_{2,0} + a_{1,1} + a_{1,0} & = -2((a_{0,2} - a_{2,0}) + (a_{0,1} - a_{1,0}))(2a_{2,0} - a_{1,1} - 3) \\ a_{0,2} + a_{1,1} + a_{0,1} & = 2((a_{0,2} - a_{2,0}) + (a_{0,1} - a_{1,0}))(2a_{0,2} - a_{1,1} - 3) \\ a_{0,0} + a_{0,1} + a_{1,0}+ & \\ +a_{0,2} + a_{1,1} + a_{2,0} & = ((a_{0,2} - a_{2,0}) + (a_{0,1} - a_{1,0}))^2 \end{cases}$$

The solution is $a_{0,2} = a_{2,0} = 1$, $a_{1,1} = -2$, the rest of coefficients are zero.

We do not know for sure how complex are procedures for solving systems of Diophantine equations at the end of weak type-inference. Moreover, it may be that for some of these systems solving procedures do not exist.

6 Semantics of the Type System

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

In section 6.1, we formalize the notion of a meaningful value using a heap-aware semantics of types and give an operational semantics of the language. Section 6.2 formulates the soundness statement with respect to this semantics and section 6.3 sketches the proof. The system is shown not to be complete in section 6.4.

6.1 Semantics of program values and expressions

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

$$\mathit{Val}\ v \ ::=\ c \mid \ell \mid \texttt{NULL} \qquad \ell \in \mathit{Loc} \qquad c \in \texttt{Int}\ ^{4}$$

$$\mathit{Heap}\ h\ :\ \mathit{Loc} \rightharpoonup \{\texttt{hd}, \texttt{tl}\} \rightharpoonup \mathit{Val}$$

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

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

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

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

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

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

An operational-semantics judgment $s;\ h;\ \mathcal{C}\ \vdash\ e\ \rightsquigarrow\ v;\ h'$ informally means that at a store s and a heap h with a set of closures $\mathcal{C}$ an expression e terminates and evaluates to the value v at the heap h'. Using heaps and frame stores, and maintaining a mapping $\mathcal{C}$ from function names to bodies for the functions definitions encountered, the operational semantics of expressions is defined by the following rules:

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

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

[4] To avoid overhead with notations we treat integer values as integer literals. Ideally, one considers integer values i rather than literals c.

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

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

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

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

$$\dfrac{s(x) = \mathtt{NULL} \qquad s;\, h;\, \mathcal{C} \ \vdash\ e_1 \ \rightsquigarrow\ v;\, h'}{\begin{array}{lll} s;\, h;\, \mathcal{C} \ \vdash\ \mathsf{match}\ x\ \mathsf{with} & \mid \mathsf{nil} \Rightarrow e_1 & \rightsquigarrow\ v;\, h' \\ & \mid \mathsf{cons}(hd, tl) \Rightarrow e_2 & \end{array}}\ \textsc{OSMatch-Nil}$$

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

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

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

6.2 Soundness

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

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

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

$$\begin{aligned}&\epsilon(p+p) = \epsilon(p) + \epsilon(p)\\&\epsilon(p-p) = \epsilon(p) - \epsilon(p)\\&\epsilon(p*p) = \epsilon(p) * \epsilon(p)\\&\eta(\mathsf{L}_p(\tau)) = \mathsf{L}_p(\eta(\tau))\end{aligned}$$

The soundness statement is defined by means of the following two predicates. One indicates if a program value is meaningful with respect to a certain heap and ground type. The other does the same for sets of values and types, taken from a frame store and ground context [5] respectively:

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

Now the soundness statement is straightforward:

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

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

To prove the theorem one needs to discuss some semantic notions and prove a few technical lemmas.

We assume *benign sharing* of variables [HJ03]. It means that evaluation of an expression leaves intact the regions of the heap, accessible from the free variables of the continuation. This condition is not typeable, but may be approximated statically by some type system, such as uniqueness types [BS99]. The discussion on this topic is beyond the scope of this report. Now we can define the operational semantics of let-expressions respecting benign sharing of variables:

To formalize the notion of benign sharing we introduce a function $\mathcal{R} : \mathit{Heap} \times \mathit{Val} \longrightarrow \mathcal{P}(\mathit{Loc})$, which computes the set of locations accessible in a given heap from a given value:

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

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

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

[5] By a ground context we understand the context that maps variable names to ground types.

$$\frac{\begin{array}{l} s;\ h;\ \mathcal{C}\ \vdash\ e_1\ \leadsto v_1;\ h_1 \\ s[x := v_1];\ h_1;\ \mathcal{C}\ \vdash\ e_2\ \leadsto v;\ h' \\ h|_{\mathcal{R}(h,\ s|_{FV(e_2)})} = h_1|_{\mathcal{R}(h,\ s|_{FV(e_2)})} \end{array}}{s;\ h;\ \mathcal{C}\ \vdash\ \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2\ \leadsto v;\ h'}\ \text{OSLet}$$

6.3 Lemmas and soundness proof

One proves by induction on the size of (the domain of) the heap following lemmas:

Lemma 8 (A program value's footprint is in the heap).
$\mathcal{R}(h,\ v) \subseteq dom(h)$.

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

$dom(h) = \emptyset$: Then no $\ell \in dom(h)$ exists and $\mathcal{R}(h,\ c) = \emptyset$ or $\mathcal{R}(h,\ \mathtt{NULL}) = \emptyset$, which is trivially a subset of $dom(h)$.

$dom(h) \neq \emptyset$:

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

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

Lemma 9 (Extending a heap does not change the footprints of program values). *If $\ell \notin dom(h)$ and $h' = h[\ell.\mathtt{hd} := v_{\mathtt{hd}},\ \ell.\mathtt{tl} := v_{\mathtt{tl}}]$, for some $v_{\mathtt{hd}},\ v_{\mathtt{tl}}$ 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.

$dom(h) = \emptyset$: Because $h' = [l.\mathtt{hd}\ = v_{\mathtt{hd}}, l.\mathtt{tl} := v_{\mathtt{tl}}]$ and $v \neq l$ we have $v \notin dom(h')$. Therefore, $\mathcal{R}(h,\ v) = \emptyset = \mathcal{R}(h',\ v))$.

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

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

$v = \ell'$: If $\ell' \notin dom(h)$, then due to $\ell' \neq \ell$, $\ell' \notin dom(h)$ either and $\mathcal{R}(h,\ v) = \emptyset = \mathcal{R}(h',\ v))$.

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

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

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

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

and the induction assumption one has

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

So,

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

Lemma 10 (Validity for Union of Variable Sets). *For all stores s and ground contexts Γ the predicate $Valid_{\mathsf{store}}(vars_1 \cup vars_2, \Gamma, s, h)$ is true if and only if both $Valid_{\mathsf{store}}(vars_1, \Gamma, s, h)$ and $Valid_{\mathsf{store}}(vars_2, \Gamma, s, h)$ are true.*

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

Lemma 11 (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 i and $w = i$, hence $v\ \models^{h'}_{\tau^\bullet}\ w$ by the definition.

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

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

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

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

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

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

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

Lemma 12 (Values only depend on values at their footprints).
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 $\tau^\bullet$.

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

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

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

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

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

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

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

Due to $\mathcal{R}(h|_{\mathit{dom}(h)\setminus\{\ell\}},\ h.\ell.\texttt{hd}) \subseteq \mathit{dom}(h) \setminus \{\ell\}$ (lemma 8) we have

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

Similarly $h|_{\mathit{dom}(h)\setminus\{\ell\}}|_{\mathcal{R}(h|_{\mathit{dom}(h)\setminus\{\ell\}},\ h.\ell.\texttt{tl})} = h|_{\mathcal{R}(h|_{\mathit{dom}(h)\setminus\{\ell\}},\ h.\ell.\texttt{tl})\setminus\{\ell\}}$. Due to $\ell \in \mathcal{R}(h,\ \ell)$, and lemma 11 – with $\mathcal{R}(h|_{\mathit{dom}(h)\setminus\{\ell\}},\ h.\ell.\texttt{hd}) \setminus \{\ell\} \subseteq \mathcal{R}(h,\ h.\ell.\texttt{hd}) \setminus \{\ell\}$, we have

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

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

Lemma 13 (Equality of the "meanings" of a program value in two heaps follows from the eq. of the footprints).
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 12 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 $h'|_{\mathit{dom}(h'|_{\mathcal{R}(h,\ v)})} = h'|_{\mathcal{R}(h,\ v)}$ we may apply lemma 11, which gives $v \models^{h'}_{\tau^\bullet} w$. □

Lemma 14 (ChangeStore).
Given a typing context Γ, store s, heap h, value v, a set of variables vars and a variable $x \notin vars$, s.t. $x \notin dom(s)$, we have $Valid_{\mathsf{store}}(vars, \Gamma, s[x := v], h) \iff Valid_{\mathsf{store}}(vars, \Gamma, s, h)$.

Proof. The lemma follows from the definition of $Valid_{\mathsf{store}}$.

Lemma 15 (SubsetFV).
Given a set of variables $vars_1$, typing context Γ, stack s, and heap h, for any set of variables $vars_2$ such that such that $vars_2 \subseteq vars_1$ we have $Valid_{\mathsf{store}}(vars_1, \Gamma, s, h) \implies Valid_{\mathsf{store}}(vars_2, \Gamma, s, h)$.

Proof. The lemma follows from the definition of $Valid_{\mathsf{store}}$.

The soundness theorem is a partial case of the following lemma:

Lemma 16 (Soundness). *For any s, h, $\mathcal{C}$, e, v, h', a set of equations D, a context Γ, a signature Σ, and a type τ, any size valuation ϵ, 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 is a node in some derivation tree, where all functions called in e are declared via* letfun*,*
- *D holds on size variables valuated by ϵ (i.e. D_ϵ holds)*

if the store is meaningful w.r.t. the context $\eta(\epsilon(\Gamma))$ then the output value is meaningful w.r.t the type $\eta(\epsilon(\tau))$.

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

We prove the statement by induction on the height of the derivation tree for the operational semantics. Given s; h; $\mathcal{C} \ \vdash \ e \ \rightsquigarrow \ v$; h' we fix some Γ, Σ, and τ, such that D; $\Gamma \ \vdash_\Sigma e : \tau$. We fix a valuation $\epsilon \in FV(\Gamma) \cup FV(\tau) \to \mathcal{Z}\rceil\sqcup\dashv$, a type instantiation $\epsilon \in FV(\Gamma) \cup FV(\tau) \to \tau^\bullet$, such that the assumptions of the lemma hold.

We must show that $Valid_{\mathsf{val}}(v, \tau_{\eta\epsilon}, h')$ holds.

ToDo: MOVE TO OP.SEM For $\mathcal{C}$ we have that if $\mathcal{C}(f) = \boldsymbol{x} \times e$ then $FV(e) \subseteq \boldsymbol{x}$.

OSICons: 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 $Valid_{\mathsf{val}}(v, \mathtt{Int}, h')$.

OSNull: In this case $v = \mathtt{NULL}$ and $\tau = \mathsf{L}_0(\tau')$ for some τ', s.t. $FVS(\tau) \subseteq FVS(\Gamma)$ Then, by the definition we have $\mathtt{NULL} \ \models^h_{\mathsf{L}_0(\tau'_{\eta\epsilon})} \ \mathtt{[]}$.

OSVar: From D_ϵ (and soundness of equational reasoning) it follows that $\tau_{\eta\epsilon} = \tau'_{\eta\epsilon}$. From this and $Valid_{\mathsf{store}}(FV(x), (\Gamma \cup (x : \tau')_{\eta\epsilon}, h, s)$ it follows that

$$Valid_{\mathsf{val}}(s(x), \tau_{\eta\epsilon}, h)$$

OSCons: In this case $e = \mathsf{cons}(hd, tl)$, $\tau = \mathsf{L}_p(\tau')$, $\{hd : \tau', tl : \mathsf{L}_{p'}(\tau')\} \subseteq \mathit{\Gamma}$ for some hd, tl, p' and τ'. Since $\mathit{Valid}_{\mathsf{store}}(FV(e), \mathit{\Gamma}_{\eta\epsilon}, s, h)$ 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 judgment 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 $h'.\ell.\mathtt{tl} \models^{h}_{(\mathsf{L}_{p'}(\tau'))_{\eta\epsilon}} w_{\mathtt{tl}}$ also hold. It is easy to see that $h = h'|_{\mathit{dom}(h')\setminus\{\ell\}}$. Thus,

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

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 $D;\ \mathit{\Gamma}\ \vdash_{\Sigma} e_1 : \tau$ 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 $\mathit{Valid}_{\mathsf{store}}(FV(e_1), \mathit{\Gamma}_{\eta\epsilon}, s, x) \implies \mathit{Valid}_{\mathsf{val}}(v, \tau_{\eta\epsilon}, h')$. From $FV(e_1) \subseteq FV(e)$, $\mathit{Valid}_{\mathsf{store}}(FV(e), \mathit{\Gamma}_{\eta\epsilon}, s, x)$, and lemma 15 it follows that $\mathit{Valid}_{\mathsf{val}}(v, \tau_{\eta\epsilon}, h')$.

OSIfFalse: 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 $D;\ \mathit{\Gamma}\ \vdash_{\Sigma}\ e_2 : \tau$. we apply the induction hypothesis to the derivation of $s;\ h;\ \mathcal{C}\ \vdash\ e_2\ \rightsquigarrow\ v;\ h'$ to obtain $\mathit{Valid}_{\mathsf{store}}(FV(e_2), \mathit{\Gamma}_{\eta\epsilon}, s, x) \implies \mathit{Valid}_{\mathsf{val}}(v, \tau_{\eta\epsilon}, h')$. From $FV(e_2) \subseteq FV(e)$, $\mathit{Valid}_{\mathsf{store}}(FV(e), \mathit{\Gamma}_{\eta\epsilon}, s, x)$, and lemma 15 it follows that $\mathit{Valid}_{\mathsf{val}}(v, \tau_{\eta\epsilon}, h')$.

OSLetFun: The result follows from the induction hypothesis for

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

with $D;\ \mathit{\Gamma}\ \vdash_{\Sigma} e_2 : \tau$ and the same η, ϵ.

OSLet: In this case $e = \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2$ for some x, e_1, and e_2 and we have $s;\ h;\ \mathcal{C}\ \vdash\ e_1\ \rightsquigarrow\ v_1;\ h_1$ and $s[x := v_1];\ h_1;\ \mathcal{C}\ \vdash\ e_2\ \rightsquigarrow\ v;\ h'$ for some v_1 and h_1. We know that $D;\ \mathit{\Gamma}\ \vdash_{\Sigma}\ e_1 : \tau'$, $x \notin \mathit{\Gamma}$ and $D;\ \mathit{\Gamma}, x : \tau'\ \vdash_{\Sigma}\ e_2 : \tau$ for some τ'. Applying the induction hypothesis to the first branch gives $\mathit{Valid}_{\mathsf{store}}(FV(e_1), \mathit{\Gamma}_{\eta\epsilon}, s, h) \implies \mathit{Valid}_{\mathsf{val}}(v_1, \tau'_{\eta\epsilon}, h_1)$. Since $FV(e_1) \subseteq FV(e_1) \cup (FV(e_2) \setminus \{x\}) = FV(e)$ and $\mathit{Valid}_{\mathsf{store}}(FV(e), \mathit{\Gamma}_{\eta\epsilon}, s, h)$ we have from lemma 15 that $\mathit{Valid}_{\mathsf{store}}(FV(e_1), \mathit{\Gamma}_{\eta\epsilon}, s, h)$ 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

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

Fix some $y \in FV(e_2)$. If $y = x$, then $\mathit{Valid}_{\mathsf{val}}(v_1, \tau'_{\eta\epsilon}, h_1)$ implies $\mathit{Valid}_{\mathsf{val}}(s[x := v_1](x), \tau'_{\eta\epsilon}, h_1)$. If $y \neq x$, then $s[x := v_1](y) = s(y)$. Because we know that sharing is benign, $h|_{\mathcal{R}(h,\ s(y))} = h_1|_{\mathcal{R}(h,\ s(y))}$, applying lemma 13 and then 15 we have that $s(y) \models^{h}_{\mathit{\Gamma}_{\eta\epsilon}(y)} w_y$ implies $s(y) \models^{h_1}_{\mathit{\Gamma}_{\eta\epsilon}(y)} w_y$ implies $s[x := v_1](y) \models^{h_1}_{\mathit{\Gamma}_{\eta\epsilon}(y)} w_y$ and thus $\mathit{Valid}_{\mathsf{val}}(s[x := v_1](y), \mathit{\Gamma}_{\eta\epsilon}(y), h_1)$. Hence, $\mathit{Valid}_{\mathsf{store}}(FV(e_2), \mathit{\Gamma}_{\eta\epsilon} \cup \{x : \tau'_{\eta\epsilon}\}, s[x := v_1], h_1)$. Therefore, $\mathit{Valid}_{\mathsf{val}}(v, \tau_{\eta\epsilon}, h')$.

OSMatch-Nil: In this case $e = \mathsf{match}\ x\ \mathsf{with}\ \mid \mathsf{nil} \Rightarrow e_1 \mid \mathsf{cons}(hd, tl) \Rightarrow e_2$ for some x, hd, tl, e_1, and e_2. The typing context has the form $\Gamma = \Gamma' \cup \{x : \mathsf{L}_p(\tau')\}$ for some Γ', τ', p. The operational-semantics derivation gives $s(x) = \mathtt{NULL}$, hence validity for $s(x)$ gives $x : \mathsf{L}_0(\tau')$ 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 $FV(e_1) \subseteq FV(e)$, $\mathit{Valid}_{\mathsf{store}}(FV(e), \Gamma_{\eta\epsilon}, s, h)$, and lemma 15 it follows that $\mathit{Valid}_{\mathsf{val}}(v, \tau_{\eta\epsilon}, h')$.

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

$$\mathit{Valid}_{\mathsf{store}}(FV(e_2), \left\{ \begin{array}{l} \Gamma'_{\eta\epsilon} \cup \\ \cup \{x : (\mathsf{L}_p(\tau'))_{\eta\epsilon}\} \cup \\ \cup \{hd : \tau'_{\eta\epsilon}\} \cup \\ \cup \{tl : \mathsf{L}_{p-1}(\tau')\}_{\eta\epsilon}\} \end{array} \right\}, s \begin{bmatrix} hd := v_{\mathtt{hd}}, \\ tl := v_{\mathtt{tl}} \end{bmatrix}, h) \implies$$
$$\implies \mathit{Valid}_{\mathsf{val}}(v, \tau_{\eta\epsilon}, h').$$

From $\mathit{Valid}_{\mathsf{store}}(FV(e), \Gamma_{\eta\epsilon}, s, h)$, $(FV(e_2) \setminus \{hd,\ tl\}) \subseteq FV(e)$, and lemma 15 we obtain $\mathit{Valid}_{\mathsf{store}}(FV(e_2) \setminus \{hd,\ tl\}, \Gamma_{\eta\epsilon}, s, h)$. Due to $hd, tl \notin \mathit{dom}(s)$ we can apply lemma 14 and get $\mathit{Valid}_{\mathsf{store}}(FV(e_2) \setminus \{hd,\ tl\}, \Gamma_\epsilon, s[hd := v_{\mathtt{hd}}, tl := v_{\mathtt{tl}}], h)$.
From the validity $s(x)\ \models^h_{(\mathsf{L}_p(\tau'))_{\eta\epsilon}}\ w_{\mathtt{hd}} :: w_{\mathtt{tl}}$, and obvious $\epsilon(p-1) = \epsilon(p) - 1$ the validity of $v_{\mathtt{hd}}$ and $v_{\mathtt{tl}}$ follows: $v_{\mathtt{hd}}\ \models^h_{\tau'_{\eta\epsilon}}\ w_{\mathtt{hd}}$, $v_{\mathtt{tl}}\ \models^h_{(\mathsf{L}_{p-1}(\tau'))_{\eta\epsilon}}\ w_{\mathtt{tl}}$ (and thus, again, $\epsilon(p) \geq 1$.)
Now $\mathit{Valid}_{\mathsf{store}}(FV(e_2), \Gamma_{\eta\epsilon} \cup \{hd : \tau', tl : \mathsf{L}_{p-1}(\tau')\}_{\eta\epsilon}, s, h)$ and, hence,

$$\mathit{Valid}_{\mathsf{val}}(v, \tau_\epsilon, h').$$

OSFun: We want to apply the induction assumption to

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

Since the original typing judgment is a node in a derivation tree, where all called in e functions are defined via letfun, there must be a node in the derivation tree with True, $y_1 : \tau^\circ, \ldots, y_k : \tau^\circ\ \vdash_\Sigma e_f : \tau'$.
We take η' and ϵ', such that

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

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

$$\mathit{Valid}_{\mathsf{store}}((y_1, \dots y_k), (y_1 : \tau^{\circ}_{1\,\eta'\epsilon'}, \dots, y_k : \tau^{\circ}_{k,\ \eta'\epsilon'}), [y_1 := v_1, \dots, y_n := v_n], h)$$
$$\Longrightarrow\ \mathit{Valid}_{\mathsf{val}}(v, \tau'_{\eta'\epsilon'}, h')$$

From $\mathit{Valid}_{\mathsf{store}}(FV(e), \mathit{\Gamma}_{\eta\epsilon}, s, h)$ we have validity of the values of the actual parameters: $v_i \models^{h}_{\mathit{\Gamma}_{\eta\epsilon}(x_i)} w_i$ for some w_i, where $1 \leq i \leq k$. Since $\mathit{\Gamma}_{\eta\epsilon}(x_i) = \tau^{\circ}_{i\,\eta'\epsilon'}$, the left-hand side of the implication holds, and one obtains $\mathit{Valid}_{\mathsf{val}}(v, \tau'_{\eta'\epsilon'}, h')$.
Since D_ϵ implies $\tau'[\dots \alpha := \tau_\alpha \dots][\dots n_{ij} := p_{ij} \dots]_{\eta\epsilon} = \tau_{\eta\epsilon}$, and $\tau'[\dots \alpha := \tau_\alpha \dots][\dots n_{ij} := p_{ij} \dots]_{\eta\epsilon} = \tau'_{\eta'\epsilon'}$ we have $\mathit{Valid}_{\mathsf{val}}(v, \tau_\epsilon, h')$.

$\square$

6.4 Completeness

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

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

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

7 Conclusion and Further Work

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

Type inference for this system is shown to be undecidable. A non-standard, practical method to infer types is introduced. It uses run-time results to generate a solvable set of equations. The results of this method are shown to be correct in all cases. Of course, termination of this method cannot be guaranteed since this would be in conflict with the undecidability of type inference.

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

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

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

References

[BS99] Erik Barendsen and Sjaak Smetsers. Graph rewriting aspects of functional programming. In H. Ehrig, G. Engels, H.-J. Kreowski, and G. Rozenberg, editors, *Handbook of Graph Grammars and Computing by Graph Transformation*, pages 63–102. World Scientific, 1999.

[CBF91] S. Chatterjee, G. E. Blelloch, and A. L. Fischer. Size and access inference for data-parallel programs. In *PLDI '91: Proceedings of the ACM SIGPLAN 1991 conference on Programming language design and implementation*, pages 130–144, New York, NY, USA, 1991. ACM Press.

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

[EJPS05] M. van Eekelen, B. Jacobs, E. Poll, and S. Smetsers. AHA: Amortized Heap Space Usage Analysis. NWO project proposal, August 2005. `http://www.cs.ru.nl/ marko/research/aha2005.pdf`.

[HJ03] M. Hofmann and S. Jost. Static prediction of heap space usage for first-order functional programs. *SIGPLAN Not.*, 38(1):185–197, 2003.

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

[JS97] C. B. Jay and M. Sekanina. Shape checking of array programs. In *Computing: the Australasian Theory Seminar, Proceedings, 1997*, volume 19 of *Australian Computer Science Communications*, pages 113–121, 1997.

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

[MJ91] Yu. Matiyasevich and J. P. Jones. Proof or recursive unsolvability of hilbert's tenth problem. *American Mathematical Monthly*, 98(10):689–709, October 1991.

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

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