Chapter 3

Size Analysis of Algebraic Data Types

Alejandro Tamalet, Olha Shkaravska, Marko van Eekelen[1]
Category: Research

Abstract: We present a size-aware type system for a first-order functional language with algebraic data types, where types are annotated with polynomials over size variables. We define how to generate typing rules for each data type, provided its user defined size function meets certain requirements. As an example, a program for balancing binary trees is type checked. The type system is shown to be sound with respect to the operational semantics in the class of shapely functions. Type checking is shown to be undecidable, however, decidability for a large subset of programs is guaranteed.

3.1 INTRODUCTION

Embedded systems or server applications often have limited resources available. Therefore, it can be important to know in advance how much time or memory a computation is going to take, for instance, to determine how much memory should at least be put in a system to enable all desired operations. This helps to prevent abrupt termination on small devices like mobile phones and Java cards as well as on powerful computers running memory exhaustive computations like GRID applications and model generation. Analysing resource usage is also interesting for optimisations in compilers, in particular optimisations of memory allocation and garbage collection techniques. An accurate estimation of heap usage enables preallocation of larger chunks of memory instead of allocating memory cells separately when needed, leading to a better cache performance. Size verification can

[1]All authors are members of the Digital Security Section, Institute for Computing and Information Sciences, Radboud University, Nijmegen, The Netherlands; {A.Tamalet, O.Shkaravska, M.vanEekelen}@cs.ru.nl. This work is sponsored by the Netherlands Organisation for Scientific Research (NWO) under grant nr. 612.063.511.

be used to avoid memory exhaustion which helps to prevent attacks that exploit it, like some "Denial of Service" attacks. Size-aware type systems can also be used to prove termination of finite computations or progression of infinite ones (see the related work section).

Decisions regarding these (and related) problems should be based on formally verified upper bounds of resource consumption. A detailed analysis of these bounds requires knowledge of the sizes of the data structures used throughout the program (see [11]).

As part of the AHA project, we study in this paper a *type-and-effect system* for a strict first-order functional language with algebraic data types, where types are annotated with size information. We focus on *shapely* function definitions in this language, where shapely means that the size of their output is polynomial with respect to the sizes of its arguments. Formally, if $size_{\tau_i}\colon \tau_i \to \mathcal{N}$ are the size functions of the types τ_i for $i = 1..k+1$, a function $f\colon \tau_1 \times \ldots \times \tau_k \to \tau_{k+1}$ is shapely if there exists a polynomial p on k variables such that

$$\forall x_1\colon \tau_1, \ldots, x_k\colon \tau_k \,.\, size_{\tau_{k+1}}(f(x_1, \ldots, x_k)) = p(size_{\tau_1}(x_1), \ldots, size_{\tau_k}(x_k))$$

For instance, if we take for lists their length to be their size, then appending two lists is shapely because the size of the output is the sum of the sizes of the inputs. However, a function that conditionally deletes an element from a list is not shapely because the size of the output can be the same as the size of the input or one less, which can not be expressed with a unique polynomial. The definition can be easily extended to size functions that return tuples of natural numbers.

We have previously shown for a basic language (whose only types are integers and lists) and a simplified size-aware type system, that type checking is undecidable in general, but decidable under a syntactical restriction [9]. Type inference through a combination of dynamic testing and type checking was developed in [12]. A demonstrator for type checking and type inference is available at `www.aha.cs.ru.nl`.

In this paper we extend this analysis to algebraic data types. We show a procedure to generate size-aware typing rules for an algebraic data type, provided its size function has a given form. Furthermore, for any data type we define a *canonical size function* which is used in case no size function is defined by the user. We prove soundness of the type system with respect to its operational semantics, which allows sharing. In the presence of sharing, the size annotations can be interpreted as an upper bound on the amount of memory used to allocate the result. Type checking is shown to be undecidable, however, the syntactic restriction introduced in [9] can be used to guarantee decidability. We also give an example that shows that the type system is incomplete.

This paper is organised as follows. In Section 3.2 we define the language and the type system, and we give generic typing rules for user defined size functions. In Section 3.3 we deal with soundness, decidability and completeness issues. Section 3.4 discusses a possible extension to the language and future work. Section 3.5 comments on related work and Section 3.6 draws conclusions.

3.2 SIZE-AWARE TYPE SYSTEM WITH ALGEBRAIC DATA TYPES

We start this section by introducing the working language and types with size annotations followed by an example with binary trees in 3.2.2. Subsection 3.2.3 shows how to obtain typing rules from a size function that meets the requirements stated in 3.2.1.

3.2.1 Language and Types

We define a type and effect system in which types are annotated with polynomial size expressions:

$$p ::= c \mid n \mid p+p \mid p-p \mid p*p$$

where c is a rational number and n denotes a size variable that ranges over natural numbers. A zero-order type can be one of the primitive data types (boolean and integers), a type variable or size annotated algebraic data type:

$$\tau ::= \texttt{Bool} \mid \texttt{Int} \mid \alpha \mid T^{p_1,\ldots,p_n}(\tau_1,\ldots,\tau_m)$$

An algebraic data type is annotated with a tuple of polynomials. This allows one to measure different aspects of an element of that type, for instance, the number of times each constructor is used. To simplify the presentation we will usually write just $T^{\mathsf{p}}(\overline{\tau})$.

The sets $FV(\tau)$ and $FSV(\tau)$ of the free type and free size variables of τ, are defined inductively in the obvious way. Let τ° denote a zero-order type whose size annotation contains just constants or size variables. First-order types are assigned to shapely functions over values of τ°-types.

$$\begin{array}{rl} \tau^f ::= & \tau_1^\circ \times \ldots \times \tau_k^\circ \to \tau_{k+1} \\ & \text{such that } FSV(\tau_{k+1}) \subseteq FSV(\tau_1^\circ) \cup \cdots \cup FSV(\tau_k^\circ) \end{array}$$

We work with a fairly simple first-order language over these types. The following grammar defines the syntax of the language, where b ranges over booleans and ι over integers, x denotes a program variable of a zero-order type, C stands for a constructor name and f for a function name.

$$\begin{array}{lll} d & ::= & \texttt{data}\ T(\overline{\alpha}) = C_1(\overline{\tau}_1(\overline{\alpha})) \mid \ldots \mid C_r(\overline{\tau}_r(\overline{\alpha})) \\ a & ::= & b \mid \iota \mid f(\overline{x}) \mid C(\overline{x}) \\ e & ::= & a \mid \texttt{letfun}\ f(\overline{x}) = e_1\ \texttt{in}\ e_2 \\ & & \mid \texttt{let}\ x = a\ \texttt{in}\ e \mid \texttt{if}\ x\ \texttt{then}\ e_1\ \texttt{else}\ e_2 \\ & & \mid \texttt{match}\ x\ \texttt{with} \mid C_1(\overline{x}_1) \Rightarrow e_{C_1} \mid \ldots \mid C_r(\overline{x}_r) \Rightarrow e_{C_r} \\ pr & ::= & d^*\ e \end{array}$$

On the data type definition we have abused of the notation: only type constructors may have type variables as parameters. Types appearing on the right hand side of the definition of a data type must not have free size variables. We prohibit head-nested let-expressions and restrict subexpressions in function calls

to variables to make type checking straightforward. Program expressions of a general form can be equivalently transformed into expressions of this form. It is useful to think of the this as an intermediate language. We also assume that the language has the typical basic operations on integers and booleans, but their study is omitted since they do not involve size annotations.

In order to add size annotations to an algebraic data type, it must be decided what to measure. Because of polymorphism, one can measure only the outer structure, e.g., since the size of $\mathsf{List}(\alpha)$ must be defined for any α, the size of a $\mathsf{List}(\mathsf{Tree}(\mathtt{Int}))$ will be just the length of the list. But, because the size is part of the type, all the elements of the list must have the same size, which allows the user to compute the total size once the sizes of the trees are known. Another consequence of polymorphism is that one usually needs to count the number of times each constructor is used to build an element. A size function for

$$\mathtt{data}\ \mathsf{TreeAB}(\alpha,\beta) = \mathsf{Empty} \mid \mathsf{Leaf}(\alpha) \mid \mathsf{Node}(\beta, \mathsf{TreeAB}(\alpha,\beta), \mathsf{TreeAB}(\alpha,\beta))$$

should return the number of empties, leaves and nodes. Any size function for these trees that returns a single natural number is losing information and the user will not be able to calculate the total size once α and β are known. One may not want to count the number of times some constructor is used because it can be deduced from the others or it is constant, e.g., any finite list has always one nil constructor cell. Ignoring some constructors can also make a function definition shapely as in the case of a function that can return trees of type TreeAB with different number of empties and leaves, but always the same number of nodes. If all the constructors cells are counted, such a function is not shapely, however, if only nodes are counted, it is shapely.

We require a size function for $T(\overline{\alpha})$ to be total and have the form

$$size_T(C_i(x_{i1},\ldots,x_{ik_i})) = \mathsf{c}_i + \sum_{j=1}^{k_i} \gamma(x_{ij})$$

where $x_{ij} : T_{ij}$, c_i is a non-negative integer or a tuple of non-negative integers and

$$\gamma(x_{ij}) = \begin{cases} size_T(x_{ij}) & \text{if } T_{ij}(\overline{\alpha}) = T(\overline{\alpha}) \\ 0 & \text{otherwise} \end{cases}$$

Henceforth, we will assume that every size function satisfies these requirements. The motivation for this is twofold. On one hand linearity is needed for decidability (see 3.3.2) and on the other hand, requiring the recursive calls of the size function to be applied to (some of) the arguments of the constructors, allows us to relate their sizes with the annotations of the respective types in the context (see 3.2.3).

A *canonical size function* for $T(\overline{\alpha})$ is a size function where each c_i is 1_i^r, the tuple of arity r (the number of constructors of T) with all zeros except for a 1 on the i-th position. It is always possible to obtain a canonical size function for a given algebraic data type, and there is only one way to construct it, thus it is unique for that type. When no size function for a type is provided by the user,

its canonical size function is used. We write s_T for the canonical size function of $T(\overline{\alpha})$. For instance, s_{List} is a function that takes a polymorphic list l and returns $(1, \mathit{length}(l))$, since it is defined as:

$$\begin{array}{ll} s_{\mathsf{List}}(\mathsf{Nil}) & = (1,0) \\ s_{\mathsf{List}}(\mathsf{Cons}(hd, tl)) & = (0,1) + s_{\mathsf{List}}(tl) \end{array}$$

We say that a size function for T is *sensible* if it returns the exact amount of occurrences of each constructor of T in its argument. Recall that an inductive type is an initial algebra of the endofunctor corresponding to its constructors. Because we do not allow free size variables in the definitions of data types, we can always "flatten" any algebraic data type of our language, and obtain an isomorphic *polynomial inductive type*. It is not difficult to prove [10] that the canonical size function of an polynomial inductive type is sensible.

A context Γ is a finite mapping from zero-order variables to zero-order types. A signature Σ is a finite function from function names to first-order types. A typing judgement is a relation of the form $D; \Gamma \vdash_\Sigma e\colon \tau$, where D is a set of *Diophantine* equations (i.e., with integer solutions) that constrains the possible values of the size variables, and Σ contains a type assumption for the function that is going to be type checked along with the signatures of the functions used in its definition. The entailment $D \vdash \mathsf{p} = \mathsf{p}'$ means that $\mathsf{p} = \mathsf{p}'$ is derivable from the equations in D, while $D \vdash \tau = \tau'$ means that τ and τ' have the same underlying type and equality of their size annotations is derivable.

The typing rules for the language, excluding the ones for data types, are shown in Figure 3.1. In the FUNAPP rule, C is used to deal with functions where the input size variables are repeated, as in the type of matrix multiplication: $\mathsf{List}^n(\mathsf{List}^k(\mathtt{Int})) \times \mathsf{List}^k(\mathsf{List}^m(\mathtt{Int})) \to \mathsf{List}^n(\mathsf{List}^m(\mathtt{Int}))$. If such a function is instantiated with lists of type $\mathsf{List}^{p_1}(\mathsf{List}^{p_2}(\mathtt{Int}))$ and $\mathsf{List}^{q_1}(\mathsf{List}^{q_2}(\mathtt{Int}))$, we add the condition $p_2 = q_1$ to C. For more details on the use of this rule see [10].

3.2.2 Example: Binary Trees

Consider the following definition of binary trees:

$$\mathtt{data}\ \mathsf{Tree}(\alpha) = \mathsf{Empty} \mid \mathsf{Node}(\alpha, \mathsf{Tree}(\alpha), \mathsf{Tree}(\alpha))$$

The canonical size function for Tree is:

$$\begin{array}{ll} s_{\mathsf{Tree}}(\mathsf{Empty}) & = (1,0) \\ s_{\mathsf{Tree}}(\mathsf{Node}(v, l, r)) & = (0,1) + s_{\mathsf{Tree}}(l) + s_{\mathsf{Tree}}(r) \end{array}$$

Conforming to s_{Tree}, an annotated binary tree has the form $\mathsf{Tree}^{e,n}(\alpha)$, where e is the number of Empty constructors (the leaves of the tree) and n is the number of nodes. We want to obtain typing rules for binary trees that will enable us to statically check the values of e and n when the binary tree is the result of a shapely function. We need one rule per constructor and one rule for pattern matching a binary tree. An empty tree has one leaf and no node, thus:

$$\frac{}{D;\Gamma \vdash_\Sigma b\colon \texttt{Bool}}\ \textsc{BConst} \qquad \frac{}{D;\Gamma \vdash_\Sigma \iota\colon \texttt{Int}}\ \textsc{IConst}$$

$$\frac{D \vdash \tau = \tau'}{D;\Gamma, x\colon \tau \vdash_\Sigma x\colon \tau'}\ \textsc{Var}$$

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

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

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

$$\frac{\begin{array}{c}\Sigma(f) = \tau_1^\circ \times \ldots \times \tau_k^\circ \to \tau_{k+1} \\ D \vdash \tau = \tau_{k+1}[\tau_1^\circ := \tau_1', \ldots, \tau_k^\circ := \tau_k'] \qquad D \vdash C\end{array}}{D;\Gamma, x_1\colon \tau_1', \ldots, x_k\colon \tau_k' \vdash_\Sigma f(x_1,\ldots,x_k)\colon \tau}\ \textsc{FunApp}$$

FIGURE 3.1. Typing rules excluding the ones for data types.

$$\frac{D \vdash (e,n) = (1,0)}{D;\Gamma \vdash_\Sigma \mathsf{Empty}\colon \mathsf{Tree}^{e,n}(\tau)}\ \textsc{Empty}$$

From s_{Tree} we obtain that in a non-empty tree, the number of leaves is equal to the sum of the number of leaves in each subtree and that the number of nodes is one more than the sum of the number of nodes in each subtree. We use variables for the sizes of the subtrees and we relate them accordingly in the premise:

$$\frac{D \vdash (e,n) = (0,1) + (e_1,n_1) + (e_2,n_2)}{D;\Gamma, v\colon \tau, l\colon \mathsf{Tree}^{e_1,n_1}(\tau), r\colon \mathsf{Tree}^{e_2,n_2}(\tau) \vdash_\Sigma \mathsf{Node}(v,l,r)\colon \mathsf{Tree}^{e,n}(\tau)}\ \textsc{Node}$$

Similarly, in the typing rule for pattern matching a binary tree, we introduce fresh variables in the typing context of the premises for the unknown quantities and we add their relationship to the set of conditions.

$$\frac{\begin{array}{c} D, (e,n) = (1,0);\ \Gamma, t\colon \mathsf{Tree}^{e,n}(\tau) \vdash_\Sigma e_{\mathsf{Empty}}\colon \tau' \\ D,\ (e,n) = (0,1) + (e_1,n_1) + (e_2,n_2);\ \Gamma, \\ t\colon \mathsf{Tree}^{e,n}(\tau), v\colon \tau, l\colon \mathsf{Tree}^{e_1,n_1}(\tau), r\colon \mathsf{Tree}^{e_2,n_2}(\tau) \vdash_\Sigma e_{\mathsf{Node}}\colon \tau' \\ e_1, e_2, n_1\, n_2 \notin vars(D) \qquad v, l, r \notin dom(\Gamma)\end{array}}{D;\Gamma, t\colon \mathsf{Tree}^{e,n}(\tau) \vdash_\Sigma \begin{array}{l}\texttt{match}\ t\ \texttt{with} \mid \mathsf{Empty} \Rightarrow e_{\mathsf{Empty}} \\ \qquad\qquad\quad \mid \mathsf{Node}(v,l,r) \Rightarrow e_{\mathsf{Node}}\end{array} : \tau'}\ \textsc{MTree}$$

To see how these rules work in practice, we apply them to a function to balance a (not necessarily ordered) binary tree. To simplify the example we add syntactic

sugar to avoid `let` constructs. It is not our intention to explain the balancing algorithm, but just to show that there are many interesting functions that can be written in our language. We begin with a function for right-rotation of nodes. We use undefined to indicate a non-terminating expression with the required type.

$$\begin{array}{l} r\_rot(v,l,r) : \alpha \times \mathsf{Tree}^{e_1,n_1}(\alpha) \times \mathsf{Tree}^{e_2,n_2}(\alpha) \to \mathsf{Tree}^{e_1+e_2,n_1+n_2+1}(\alpha) = \\ \quad \texttt{match}\ l\ \texttt{with} \mid \mathsf{Empty} \Rightarrow \mathsf{undefined} \\ \qquad\qquad\qquad \mid \mathsf{Node}(v_1, l_1, r_1) \Rightarrow \mathsf{Node}(v_1, l_1, \mathsf{Node}(v, r_1, r)) \end{array}$$

By applying the rule MTREE we get two branches. The branch for the Empty case is undefined and thus we do not need to type check it. The other branch is

$$\dfrac{\dfrac{(e_1,n_1) = (e_{11}+e_{12},\, n_{11}+n_{12}+1) \vdash (e_1+e_2,\, n_1+n_2+1) = (e_{11}+e_{12}+e_2,\, n_{11}+(n_{12}+n_2+1)+1)}{(e_1,n_1) = (e_{11}+e_{12}, n_{11}+n_{12}+1);\ v, v_1 : \alpha, l : \mathsf{Tree}^{e_1,n_1}(\alpha),\ l_1 : \mathsf{Tree}^{e_{11},n_{11}}(\alpha),\ r_1 : \mathsf{Tree}^{e_{12},n_{12}}(\alpha) \vdash_\Sigma \mathsf{Node}(v_1, l_1, \mathsf{Node}(v, r_1, r)) : \mathsf{Tree}^{e_1+e_2,n_1+n_2+1}(\alpha)}\ \textsc{Node}}{v : \alpha,\ l : \mathsf{Tree}^{e_1,n_1}(\alpha),\ r : \mathsf{Tree}^{e_2,n_2}(\alpha) \vdash_\Sigma \texttt{match}\ l \ldots : \mathsf{Tree}^{e_1+e_2,n_1+n_2+1}(\alpha)}\ \textsc{MTree}$$

Similarly, we can type check the left-right rotation function. For simplicity we write it in a Haskell-like style of pattern matching.

$$\begin{array}{l} lr\_rot : \alpha \times \mathsf{Tree}^{e_1,n_1}(\alpha) \times \mathsf{Tree}^{e_2,n_2}(\alpha) \to \mathsf{Tree}^{e_1+e_2,n_1+n_2+1}(\alpha) \\ lr\_rot(v, \mathsf{Node}(v_1, l_1, \mathsf{Node}(v_{12}, l_{12}, r_{12})), r) = \\ \quad \mathsf{Node}(v_{12}, \mathsf{Node}(v_1, l_1, l_{12}), \mathsf{Node}(v, r_{12}, r)) \end{array}$$

Now we define the left balance function, which is easily type checked since both branches have the same type. The definitions of *balance* and RightWeight are omitted because they are not needed for out analysis.

$$\begin{array}{l} l\_bal(v,l,r) : \alpha \times \mathsf{Tree}^{e_1,n_1}(\alpha) \times \mathsf{Tree}^{e_2,n_2}(\alpha) \to \mathsf{Tree}^{e_1+e_2,n_1+n_2+1}(\alpha) = \\ \quad \texttt{if}\ balance(l) == \mathsf{RightWeight} \\ \quad \texttt{then}\ lr\_rot(v,l,r) \\ \quad \texttt{else}\ r\_rot(v,l,r) \end{array}$$

Then we type check a function that inserts an element into a balanced binary tree:

$$
\begin{array}{l}
\mathit{insert}(a,t)\colon \alpha \times \mathsf{Tree}^{e,n}(\alpha) \to \mathsf{Tree}^{e+1,n+1}(\alpha) = \\
\quad \texttt{match}\ t\ \texttt{with} \mid \mathsf{Empty} \Rightarrow \mathsf{Node}(a, \mathsf{Empty}, \mathsf{Empty}) \\
\qquad\qquad\qquad \mid \mathsf{Node}(v,l,r) \Rightarrow \mathsf{let}\ l_2 = \mathit{insert}(a,l) \\
\qquad\qquad\qquad\qquad\qquad\qquad \mathsf{in}\ \texttt{if}\ \mathit{height}(l_2) == \mathit{height}(r)+2 \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad \texttt{then}\ \mathit{l\_bal}(v,l_2,r) \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad \texttt{else}\ \mathsf{Node}(v,l_2,r)
\end{array}
$$

Applying MTREE we get two branches. For the Empty branch we get the entailment $(e,n) = (1,0) \vdash (e+1,n+1) = (1+1,0+0+1)$ and for the Node branch we have the judgement:

$$
\begin{array}{l}
(e,n) = (e_1+e_2, n_1+n_2+1);\ t\colon \mathsf{Tree}^{e,n}(\alpha), \\
v\colon \alpha,\ l\colon \mathsf{Tree}^{e_1,n_1}(\alpha),\ r\colon \mathsf{Tree}^{e_2,n_2}(\alpha)
\end{array}
\vdash_\Sigma \mathsf{let}\ l_2 = \ldots\colon \mathsf{Tree}^{e+1,n+1}(\alpha)
$$

Using LET we get $l_2\colon \mathsf{Tree}^{e_1+1,n_1+1}(\alpha)$. Both branches of the `if` have the same type, so we only need to check the entailment it generates:

$$(e,n) = (e_1+e_2, n_1+n_2+1) \vdash (e+1,n+1) = ((e_1+1)+e_2, (n_1+1)+n_2+1)$$

Then we define a function to build a balanced tree from a list:

$$
\begin{array}{l}
\mathit{build\_bal\_tree}(xs) : \mathsf{List}^n(\alpha) \to \mathsf{Tree}^{n+1,n}(\alpha) = \\
\quad \texttt{match}\ xs\ \texttt{with} \mid \mathsf{Nil} \Rightarrow \mathsf{Empty} \\
\qquad\qquad\qquad\quad \mid \mathsf{Cons}(hd,tl) \Rightarrow \mathit{insert}(hd, \mathit{build\_bal\_tree}(tl))
\end{array}
$$

From the Nil branch we get the condition $n = 0 \vdash (n+1,n) = (1,0)$, which is trivially true and for the Cons branch we have:

$$
\frac{\vdash (n+1,n) = ((n-1)+1+1, (n-1)+1)}{hd\colon \alpha,\ tl\colon \mathsf{List}^{n-1}(\alpha) \vdash_\Sigma \mathit{insert}(hd, \mathit{build\_bal\_tree}(tl))\colon \mathsf{Tree}^{n+1,n}(\alpha)}\ \textsc{FunApp}
$$

Finally, we define and type check a function that balances a binary tree:

$$\mathit{balance\_tree}(t)\colon \mathsf{Tree}^{e,n}(\alpha) \to \mathsf{Tree}^{n+1,n}(\alpha) = \mathit{build\_bal\_tree}(\mathit{flatten}(t))$$

where *flatten* is a function with type $\mathsf{Tree}^{e,n}(\alpha) \to \mathsf{List}^n(\alpha)$ that returns a list with the elements of a binary tree. By applying the typing rule for function application twice, we get the trivial entailment $\vdash (n+1,n) = (n+1,n)$. When the tree is flattened, we loose the information about e, thus e does not appear in the resulting type of *balance_tree*.

3.2.3 Typing Rules for Algebraic Data Types

Below, we give a procedure for obtaining typing rules for an arbitrary algebraic data type. Let $T(\overline{\alpha})$ be an algebraic data type defined as

$$\texttt{data}\ T(\overline{\alpha}) = C_1(\overline{\tau}_1(\overline{\alpha})) \mid \ldots \mid C_r(\overline{\tau}_r(\overline{\alpha}))$$

and let $size_T$ be the size function of $T(\overline{\alpha})$. For each constructor C_i we add a typing rule of the form

$$\frac{D \vdash \mathsf{p} = \mathsf{c}_i + \sum_{j=1}^{k_i} \mathsf{p}_{ij}}{D;\, \Gamma,\, x_{ij} \colon \gamma'_{ij}(T(\overline{\tau}))\ \text{ for } j = 1..k_i \vdash_\Sigma C_i(x_{i1}, \ldots, x_{ik_i}) \colon T^{\mathsf{p}}(\overline{\tau})}\ \mathrm{C}_i\ \text{ for } i = 1..r$$

where c_i and the a_{ij} are taken from the definition of $size_T$, and γ'_{ij} is defined as

$$\gamma'_{ij}(T(\overline{\tau})) = \begin{cases} T^{\mathsf{p}_{ij}}(\overline{\tau}) & \text{if } \tau_{ij}(\overline{\tau}) = T(\overline{\tau}) \\ \tau_{ij}(\overline{\tau}) & \text{otherwise} \end{cases}$$

The idea is that if the type of x_{ij} is $T(\overline{\tau})$, the one we are defining the typing rules for, then it must have a size annotation that we call p_{ij}, otherwise its type is just $\tau_{ij}(\overline{\tau})$. There is a clear correspondence between γ and γ'.

We also add a typing rule for pattern matching an element of type $T(\overline{\alpha})$:

$$\frac{\begin{array}{l} D,\ \mathsf{p} = \mathsf{c}_i + \sum_{j=1}^{k_i} \mathsf{n}_{ij};\ \Gamma,\ x \colon T^{\mathsf{p}}(\overline{\tau}),\ x_{ij} \colon \gamma'_{ij}(T(\overline{\tau}))\ \text{ for } j = 1..k_i \vdash_\Sigma e_i \colon \tau'\ \text{ for } i = 1..r \\ \mathsf{n}_{ij} \notin vars(D),\ x_{ij} \notin dom(\Gamma)\ \text{ for } i = 1..r, j = 1..k_i \end{array}}{\begin{array}{ll} & \texttt{match}\ x\ \texttt{with} \\ D;\, \Gamma,\, x \colon T^{\mathsf{p}}(\overline{\tau}) \vdash_\Sigma & \vdots \qquad\qquad : \tau' \\ & \mid C_r(x_{r1}, \ldots, x_{rk_r}) \Rightarrow e_r \end{array}}\ \textsc{MatchT}$$

Each of the size variables of n_{ij} and the formal parameters of the constructors are assumed to be fresh. Notice that there is one premise per constructor. When $\gamma_{ij}(T(\overline{\tau}))$ is $\tau_{ij}(\overline{\tau})$ we regard n_{ij} as 0, that is, we omit that variable from the sum.

3.3 SOUNDNESS, DECIDABILITY AND COMPLETENESS

This section is devoted to soundness and completeness of the type system and decidability of type checking, extending previous results on these topics to a language with algebraic data types.

3.3.1 Soundness

Set-theoretic heap-aware semantics of a ground algebraic data type (i.e., a type where all size and type variables are instantiated) is an obvious extension of the

semantics of lists that can be found, for instance, in [9]. Intuitively, an instance of a ground type is presented in a heap as a directed tree-like structure, that may overlap with other structures. The only restriction is that it must be acyclic.

Since our type system is not linear, that is, a program variable may be used more than once, a data structure in a heap may consist of overlapping substructures. This is the case, for instance, for a heap representation of $\mathsf{Node}(1,t,t)$, where t is a non-empty tree. In general, in a calculation of the *size* of a structure, a node is counted as many times as it is referenced. Hence, a sensible size function gives an upper bound for the actual amount of constructor cells allocated by the structure. If there is no internal sharing, the sensible size function is equal to the amount of cells actually allocated.

A location is the address of some constructor-cell of a ground type. A *program value* is either an integer or boolean constant, or a location. A *heap* is a finite partial mapping from locations and fields to program values, and an *object heap* is a finite partial map from locations to *Constructor*, the set of (the names of) constructors. Below, we assume that for any heap h, there is an object heap oh such that $dom(h) = dom(oh)$.

Let τ be a type defined by a set of constructors C_i, where $1 \leq i \leq r$. With a constructor C_i of arity $k_i > 0$, we associate a collection of field names $C_i\_field_j$, where $1 \leq j \leq k_i$. Let *Field* be the set of all field names in a given program. We also assume that any null-ary constructor is placed in a location with 1 empty integer field. With a 0-arity constructor C_i we associate the field name $C_i\_field_1$. The reason to introduce a "fake" field for null-ary constructors is to make the proofs more uniform. Formally:

$$\begin{array}{ll} \mathit{Val}\ v\ ::=\ \iota \mid b \mid \ell \qquad \ell \in \mathit{Loc} \qquad \iota \in \mathtt{Int} \qquad b \in \mathtt{Bool} & \\ \mathit{Heap}\ h\ :\ \mathit{Loc} \rightharpoonup \mathit{Field} \rightharpoonup \mathit{Val} \qquad \mathit{ObjHeap}\ oh\ :\ \mathit{Loc} \rightharpoonup \mathit{Constructor} & \end{array}$$

We will write $h[\ell.\mathit{field} := v]$ for the heap equal to h everywhere but in ℓ, which at the field of ℓ named *field* gets the value v.

The semantics w of a program value v is a set-theoretic interpretation with respect to a specific heap h, its object heap oh and a ground type $\tau^\bullet$, via a five-place relation $v \models^{h;oh}_{\tau^\bullet} w$. Integer and boolean constants interpret themselves, and locations are interpreted as non-cyclic structures:

$$\begin{array}{ll} \iota \models^{h;oh}_{\mathtt{Int}} \iota & b \models^{h;oh}_{\mathtt{Bool}} b \\ \ell \models^{h;oh}_{T^{\mathsf{c}}(\tau^\bullet)} C & \text{if } C \text{ is a null-ary constructor of } T,\ \ell \in dom(h),\ oh(\ell) = C \\ & \text{and the constant vector } \mathsf{c} \text{ is the size of } C \\ \ell \models^{h;oh}_{\tau^\bullet} C(w_1,\ldots,w_k) & \text{if } \ell \in dom(h),\ oh(\ell) = C \\ & C\colon \tau_1^\bullet \times \ldots \times \tau_k^\bullet \to \tau^\bullet \text{ (i.e. it is a ground instance)}, \\ & \tau^\bullet = T^{\mathsf{n}^0}(\overline{\tau^\bullet}') \text{ for some } \overline{\tau^\bullet}',\ \mathsf{n}^0 = \mathit{size}_T(C(w_1,\ldots,w_k)), \\ & \text{and for all } 1 \leq j \leq k\colon h.\ell.C\_field_j \models^{h|_{dom(h)\setminus\{\ell\}};oh|_{dom(oh)\setminus\{\ell\}}}_{\tau_j^\bullet} w_j \end{array}$$

where $h|_{dom(h)\setminus\{\ell\}}$ 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. At the beginning it contains only 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: *Store* s: *ExpVar* $\rightharpoonup$ *Val*.

An operational semantics judgement $s; h; oh, \mathscr{C} \vdash e \rightsquigarrow v; h'; oh'$ informally means that at a store s, a heap h, its object heap oh and with the set $\mathscr{C}$ of function closures (bodies), the evaluation of an expression e terminates with value v in the heap h' and object heap oh'.

Using heaps and frame stores, and maintaining a mapping $\mathscr{C}$ from function names to bodies for the functions definitions encountered, the operational semantics of expressions is defined in a usual way. Here we give the rules for constructors and (non null-ary) pattern-matching. The other rules may be found in [10].

$$\frac{s(x_1) = v_1, \ldots, s(x_k) = v_k \qquad \ell \notin dom(h)}{s; h; oh, \mathscr{C} \vdash C(x_1, \ldots, x_k) \rightsquigarrow \ell; h[\ell.C\_field_1 := v_1, \ldots, \ell.C\_field_k := v_k];\ oh[\ell := C]} \text{ OSCons}$$

$$\frac{\begin{array}{c} oh(s(x)) = C_i \qquad h.s(x).C_i\_field_1 = v_1, \ldots, h.s(x).C_i\_field_{k_i} = v_{k_i} \\ s[x_1 := v_1, \ldots, x_{k_i} := v_{k_i}]; h; oh, \mathscr{C} \vdash e_i \rightsquigarrow v; h'; oh' \end{array}}{\begin{array}{l} s; h; oh, \mathscr{C} \vdash \begin{array}{l} \texttt{match}\ x\ \texttt{with} \mid C_1(x_{11}, \ldots, x_{1k_1}) \Rightarrow e_1 \\ \qquad\qquad \vdots \qquad\qquad\qquad\qquad \rightsquigarrow v; h'; oh' \\ \mid C_r(x_{r1}, \ldots, x_{rk_r}) \Rightarrow e_r \end{array} \end{array}} \text{ OSMatch} - C_i$$

Let a valuation ε : *SizeVar* $\rightarrow \mathscr{Z}$ map size variables to concrete sizes (integer numbers) and an instantiation η : *TypeVar* $\rightarrow \tau^\bullet$ map type variables to ground types. Applied to a type, context, or size expression, valuation and instantiation map all variables occurring in it to their valuation and instantiation images: $\varepsilon(p[+,-,*]p) = \varepsilon(p)[+,-,*]\varepsilon(p)$ and $\eta(\varepsilon(T^{\mathsf{p}}(\tau))) = T^{\varepsilon(\mathsf{p})}(\eta(\tau))$.

The soundness statement is defined by means of the following two predicates. One indicates whether 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:

$$\begin{array}{lcl} \mathit{Valid}_{\mathsf{val}}(v, \tau^\bullet, h; oh) & = & \exists\, w.\ v \models^{h,oh}_{\tau^\bullet} w \\ \mathit{Valid}_{\mathsf{store}}(vars, \Gamma, s, h; oh) & = & \forall\, x \in vars.\ \mathit{Valid}_{\mathsf{val}}(s(x), \Gamma(x), h, oh) \end{array}$$

Now, stating soundness of the type system is straightforward:

Theorem 3.1. *Let* $s; h; oh, [\,] \vdash e \rightsquigarrow v; h'; oh'$. *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* $\eta(\varepsilon(\Gamma))$ *then the output value is meaningful w.r.t the type* $\eta(\varepsilon(\tau))$:

$$\forall\, \eta, \varepsilon.\ \mathit{Valid}_{\mathsf{store}}(FV(e), \eta(\varepsilon(\Gamma)), s, h, oh) \implies \mathit{Valid}_{\mathsf{val}}(v, \eta(\varepsilon(\tau)), h', oh')$$

The proof is a routine done by induction on the operational-semantics tree. It can be found in the technical report [10].

3.3.2 Decidability

Type checking using the type system studied in this work seems to be straightforward because for every syntactic construction of the language there is only one applicable typing rule. The procedure ultimately reduces to proving equations involving rational polynomials.

Lemma 3.2. *The type checking problem $D; \Gamma \vdash_\Sigma e : \tau$ can be reduced to checking a finite number of entailments of the form $D' \vdash \mathsf{p} = \mathsf{q}$, where the variables in D', p and q are either free size variables of Γ or size variables introduced during the type checking procedure.*

Proof. By induction on the structure of the language.

But consider the following expression, where $f_i : \mathsf{List}^{n_1}(\alpha_1) \times \ldots \times \mathsf{List}^{n_k}(\alpha_k) \to \mathsf{List}^{p_i(n_1,\ldots,n_k)}(\alpha)$ for $i = 0, 1, 2$, (assuming we count only the number of elements).

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

When checking whether this expression has type $\mathsf{List}^{n_1}(\alpha_1) \times \ldots \times \mathsf{List}^{n_k}(\alpha_k) \to \mathsf{List}^{p(n_1,\ldots,n_k)}(\alpha)$, in the Nil branch we will get the entailment

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

To validate this entailment we must know whether p_0 has roots or not (that is, whether the Nil branch can be entered at all). In [9] it is shown that for *any* given polynomial q, it is possible to construct a function f_0 whose result has as size annotation the polynomial $p_0 = q^2$, whose roots are exactly the ones of q. Hence, type checking reduces to solving Hilbert's tenth problem and thus it is undecidable.

The source of the problem in the previous example was that the pattern match was done over a variable bound by a `let`. We can avoid these cases with a syntactical restriction that we call *no-let-before-match*: given a function body, allow pattern matching only on the function parameters or variables bound by other pattern matchings. Even with this restriction, one can write all shapely primitive recursive functions for our data types because they induce a (polynomial) functor. For instance, the operator for primitive recursion on lists is defined as follows:

$$\begin{array}{l}f(x, \bar{y}) = \texttt{match}\ x\ \texttt{with} \mid \mathsf{Nil} \Rightarrow g(\bar{y}) \\ \qquad\qquad\qquad\qquad\quad \mid \mathsf{Cons}(hd, tl) \Rightarrow h(hd, tl, \bar{y}, f(tl, \bar{y}))\end{array}$$

where g and h are functions already defined, and $\bar{y}$ is a sequence of parameters. It is obvious that f satisfies the syntactic restriction. However, we want to emphasise that this condition is sufficient, but not necessary for decidability.

For the proof of decidability we refer the reader to the technical report [10]. The key step in the proof is to show that type checking can be reduced to a set

of entailments of the form $D \vdash \mathsf{p} = \mathsf{q}$, where each set of constraints D determines *tree-decompositions* of the size variables, i.e., trees of size variables where a father is a linear combination of its children. Then we replace the variables in breadth-first order to obtain two polynomials in terms of the leaves of these trees. Checking the equality of the polynomials is then just a matter of comparing the coefficient of the variables with the same degree.

3.3.3 Completeness

The type system is not complete: there are shapely functions for which shapeliness cannot be proved by means of the typing rules and arithmetic. This comes as no surprise if we consider that the type system subsumes Peano arithmetic. Another reason for incompleteness is that the typing rule for `if` does not keep any size information obtainable from the condition. Consider, for instance, the following schema of expressions, where $f(x)$ is a list of integers:

$$\texttt{let}\, z = f(x)\, \texttt{in}\, \texttt{if}\, \mathit{length}(z) == 0\, \texttt{then}\, z\, \texttt{else}\, \mathsf{Nil}$$

These expressions have type $\mathsf{List}^0(\texttt{Int})$, however, the type checker fails to acknowledge it.

3.4 DISCUSSION AND FUTURE WORK

In this section we discuss a variation of our type system, a possible extension and future work.

First we note that instead of generating one typing rule for each constructor it is possible to derive a size-annotated type for each of them, add these types to the set of signatures Σ and then use the function application rule. This approach is preferred since it results in a type system with fewer rules, however, for presentation purposes, we have chosen to generate typing rules for them because it makes clearer the role that the set of constraints D plays. A typing rule for pattern matching each algebraic data type is still needed.

One possible extension to the language and the type system is to add *size-parametric data types*, i.e., types that are parametrised by a tuple of size variables that can be used as size annotations in the definition of the type.

An m-ary tree is a tree where each node has m subtrees. We say that a tree of height h is *h-full* if all the leaves are at height h. When the height is not relevant, we say that it is *full*. We can define m-ary full trees as a size-parametric data type.

$$\texttt{spdata}\ \mathsf{MFullTree}_m(\alpha) = \mathsf{Empty} \mid \mathsf{Node}(\alpha, \mathsf{List}^m(\mathsf{MFullTree}_m(\alpha)))$$

It is clear that this defines m-ary trees. They are also full because the subtrees at the same level must all have the same size. Assuming that we are counting the occurrences of each constructor[2], it is not hard to come up with typing rules for

[2]Since the number of nodes in an m-ary full tree depends on its height, any function that re-shapes one of these trees will have size annotations involving logarithms. Therefore, for this data structure it would be better to define its size as its height.

MFullTree.

$$\frac{D \vdash (e,n) = (1,0)}{D;\Gamma \vdash_\Sigma \mathsf{Empty} : \mathsf{MFullTree}_m^{e,n}(\tau)}\ \textsc{Empty}$$

$$\frac{D \vdash (e,n) = (0,1) + m * (e',n')}{D;\Gamma, v:\tau, ts: \mathsf{List}^m(\mathsf{MFullTree}_m^{e',n'}(\tau)) \vdash_\Sigma \mathsf{Node}(v,ts) : \mathsf{MFullTree}_m^{e,n}(\tau)}\ \textsc{Node}$$

$$\frac{\begin{array}{c} D,(e,n) = (1,0); \Gamma, t : \mathsf{MFullTree}_m^{e,n}(\tau) \vdash_\Sigma e_{\mathsf{Empty}} : \tau' \\ D,(e,n) = (0,1) + m * (e',n'); \Gamma, t : \mathsf{MFullTree}_m^{e,n}(\tau), \\ v:\tau, ts : \mathsf{List}^m(\mathsf{MFullTree}_m^{e',n'}(\tau)) \end{array} \vdash_\Sigma e_{\mathsf{Node}} : \tau' \qquad e', n' \notin vars(D) \qquad v, ts \notin dom(\Gamma)}{D;\Gamma, t : \mathsf{MFullTree}_m^{e,n}(\tau) \vdash_\Sigma \begin{array}{l} \mathtt{match}\ t\ \mathtt{with} \mid \mathsf{Empty} \Rightarrow e_{\mathsf{Empty}} \\ \qquad\qquad\quad \mid \mathsf{Node}(v,ts) \Rightarrow e_{\mathsf{Node}} \end{array} : \tau'}\ \textsc{MMFTree}$$

A size function for MFullTree counting both constructors is defined below:

$$\begin{array}{lll} size & : \mathsf{MFullTree}_m(\alpha) \to \mathcal{N} \times \mathcal{N} & \\ size(\mathsf{Empty}) & = (1,0) & \\ size(\mathsf{Node}(v,ts)) & = (0,1) + m * \mathtt{match}\ ts\ \mathtt{with} & \mid \mathsf{Nil} \Rightarrow (0,0) \\ & & \mid \mathsf{Cons}(hd,tl) \Rightarrow size(hd) \end{array}$$

But there is no direct relationship between this size function and the previous typing rules. The size function used in the typing rules is simpler because the size of the subtrees can be obtained from the typing context. In order to restore the relationship, we can add a parameter to the size function representing the size of the subtrees.

$$\begin{array}{ll} size & : \mathsf{MFullTree}_m(\alpha) \times (\mathcal{N} \times \mathcal{N}) \to \mathcal{N} \times \mathcal{N} \\ size(\mathsf{Empty}, (e',n')) & = (1,0) \\ size(\mathsf{Node}(v,ts), (e',n')) & = (0,1) + m * (e',n') \end{array}$$

This procedure can be applied to other data types, but the generalisation is not elegant. Although this extension would add some expressiveness to the language, its usefulness is not clear since the added types are quite restricted (note, e.g., that a node at the bottom of an m-ary full tree has list of m Empty subtrees).

We believe that our recent results on *type inference* for size-annotated lists [12] can be easily extrapolated to ordinary inductive types. Furthermore, we want to extend our current implementation to deal with data types both in the canonical way and by allowing user defined size function. Our long term goals are to study type systems annotated with upper and lower bound sizes and to investigate shapeliness in the context of imperative languages.

3.5 RELATED WORK

Amortised heap space analysis has been developed for linear bounds by Hofmann and Jost [5]. Precise knowledge of sizes is required to extend this approach to non-linear bounds [11]. Brian Campbell [4] extended this approach to infer bounds on *stack* space usage.

Other work on size analysis has been restricted to monotonic dependencies. In *type-based termination* analysis e.g., it is enough to assure that the size (more precisely, an upper bound of it) of a data structure decreases in a recursive call. Research by Pareto has yielded an algorithm to automatically check sized types where linear size expression are upper bounds [8]. In the thesis of Abel [1] ordinals above ω are considered as well (they are used, e.g., for types like streams). The language of (ordinal) size expressions for zero-order types in this work is rather simple: it consists of ordinal variables, ordinal successor, and an ordinal limit (see also [2]). This is enough for termination analysis, however for heap consumption analysis more sophisticated size expressions are needed. Construction of non-linear upper bounds using a traditional type system approach has been presented by Hammond and Vasconcellos [14], but this work leaves recurrence equations unsolved and is limited to monotonic dependencies. The work on quasi-interpretations by Bonfante et al. [3] also requires monotonic dependencies.

The EmBounded project aims to identify and certify resource-bounded code in *Hume*, a domain-specific high-level programming language for real-time embedded systems. In his thesis, Pedro Vasconcelos [13] uses abstract interpretation to automatically infer linear approximations of the sizes of recursive data types and the stack and heap of recursive functions written in a subset of *Hume*.

Exact input-output size dependencies have been explored by Jay and Sekanina [7]. In this work, a shapely program is translated into a program involving sizes. Thus, the relation between sizes is given as a program. However, deriving an arithmetic function from it is beyond the scope of the paper. In a closely related work [6], Jay and Cockett study shapely types, i.e., those whose data and data can be separated in a categorical setting. A notable difference is that we do not consider a type shapely per se, instead its size function determines whether it is shapely.

3.6 CONCLUSIONS

We studied an effect type system with size annotations for a first-order functional language. We provided generic typing rules for algebraic data types based on user defined size functions and we proved soundness of the type system with respect to the operational semantics. Our choice to allow (not necessarily monotonic) polynomials as size annotations brings undecidability to type checking, however, it was shown that for a wide range of programs, decidability of type checking functions with algebraic data types can be ensured. Our experience is that in practice, the entailments obtained while type checking are easily solvable.

Although the practical applicability of this work is limited, it explores the cur-

rent limits of the field. It is also an step towards our goal of providing a practical resource analysis. Its main limitation is that it requires size dependencies to be exact. We are working on an extension of the type system that allows to express lower and upper bounds by specifying a family of indexed polynomials.

REFERENCES

[1] A. Abel. *A Polymorphic Lambda-Calculus with Sized Higher-Order Types*. PhD thesis, LFE Theoretische Informatik, Ludwig-Maximilians-Universitt Mnchen, 2006.

[2] A. Abel. Implementing a Normalizer Using Sized Heterogeneous Types. *Journal of Functional Programming, MSFP'06 special issue*, 2008. to appear.

[3] G. Bonfante, J.-Y. Marion, and J.-Y. Moyen. Quasi-interpretations, a way to control resources. *Theoretical Computer Science*, 2005.

[4] B. Campbell. *Space Cost Analysis Using Sized Types*. PhD thesis, School of Informatics, University of Edinburgh, 2008.

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

[6] B. C. Jay and J. R. B. Cockett. Shapely Types and Shape Polymorphism. In *Programming Languages and Systems - ESOP '94*, pages 302–316. Springer Verlag, 1994.

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

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

[9] O. Shkaravska, R. van Kesteren, and M. van Eekelen. Polynomial Size Analysis for First-Order Functions. In S. R. D. Rocca, editor, *Typed Lambda Calculi and Applications (TLCA'2007), Paris, France*, volume 4583 of *LNCS*, pages 351–366. Springer, 2007.

[10] A. Tamalet, O. Shkaravska, and M. van Eekelen. A Size-Aware Type System with Algebraic Data Types. Technical Report ICIS-R08006, Radboud University Nijmegen.

[11] M. van Eekelen, O. Shkaravska, R. van Kesteren, B. Jacobs, E. Poll, and S. Smetsers. AHA: Amortized Heap Space Usage Analysis. In M. Morazán, editor, *Selected Papers of the 8^{th} International Symposium on Trends in Functional Programming (TFP'07), New York, USA*, pages 36–53. Intellect Publishers, UK, 2007.

[12] R. van Kesteren, O. Shkaravska, and M. van Eekelen. Inferring static non-monotonically sized types through testing. In *Proceedings of* 16^{th} *International Workshop on Functional and (Constraint) Logic Programming (WFLP'07), Paris, France*, volume 216C of *ENTCS*, pages 45–63, 2007.

[13] P. B. Vasconcelos. *Space Cost Analysis Using Sized Types*. PhD thesis, School of Computer Science, University of St. Andrews, August 2008.

[14] P. B. Vasconcelos and K. Hammond. Inferring Cost Equations for Recursive, Polymorphic and Higher-Order Functional Programs. In P. Trinder, G. Michaelson, and R. Peña, editors, *Implementation of Functional Languages:* 15^{th} *International Workshop, IFL 2003, Edinburgh, UK, September 8–11, 2003. Revised Papers*, volume 3145 of *LNCS*, pages 86–101. Springer-Verlag, 2004.