Radboud Repository

Radboud University Nijmegen

PDF hosted at the Radboud Repository of the Radboud University Nijmegen

The following full text is a publisher's version.

For additional information about this publication click this link.
http://hdl.handle.net/2066/72731

A Size-Aware Type System
with Algebraic Data Types

With proofs of soundness and decidability

Alejandro Tamalet, Olha Shkaravska, and Marko van Eekelen
{A.Tamalet, O.Shkaravska, M.vanEekelen}@cs.ru.nl

Technical Report ICIS-R08006**
Institute for Computing and Information Sciences
Radboud University
Heyendaalseweg 135, 6525 AJ, Nijmegen, The Netherlands.

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.

Keywords: Size Analysis, Shapely Functions, Type Checking, Algebraic Data Types, Ordinary Inductive Types

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 be used to avoid memory exhaustion which helps to prevent attacks that exploit it, like some "Denial of Service" attacks. Size-aware

---

** A shorter version of this work entitled "Size Analysis of Algebraic Data Types" was presented in the Ninth Symposium on Trends in Functional Programming, 2008.

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 [21]).

As part of the AHA project, we study in this paper a *type-and-effect system* [17, 15] 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 $size_{\tau_i} : \tau_i \to \mathcal{N}$ are the size functions of the types τ_i for $i = 1..k + 1$, a function $f : \tau_1 \times \ldots \times \tau_k \to \tau_{k+1}$ is shapely if there exists a polynomial p on k variables such that

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

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 [18]. Type inference through a combination of dynamic testing and type checking was developed in [22]. 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 [18] 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 2 we define the language and the type system, and we give generic typing rules for user defined size functions. In Section 4 we deal with soundness, decidability and completeness issues. Section 3 recalls the category-theoretic semantics of inductive types as initial algebras. We show that for parameterised polynomial inductive types the canonical size function is sensible and, moreover, that it is an initiality homomorphism. Section 5

2

discusses the addition of size-parametric data types to the language. Section 6 comments on related work and Section 7 draws conclusions and gives pointers to future work.

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 2.2. Subsection 2.3 shows how to obtain typing rules from a size function that meets the requirements stated in 2.1.

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:

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

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 $T^{\mathbf{p}}(\overline{\tau})$.

Note that the types $\mathsf{List}^0(\mathsf{List}^m(\mathtt{Int}))$ are equivalent for all m because their only inhabitant is the empty list. When counting the occurrences of all constructors, we can generalise this to any algebraic data type by regarding as equivalent all elements that have a size of zero in all the non-nullary constructors of the outer type. For example, elements of type $\mathsf{Tree}^{1,0}(\mathsf{List}^{1,m}(\mathtt{Int}))$ are considered equivalent for any m. The canonical value of this class is $\mathsf{Tree}^{1,0}(\mathsf{List}^{1,0}(\mathtt{Int}))$. In this work τ denotes in fact the canonical representative $\tau_{\equiv}$.

The sets $FV(\tau)$ and $FSV(\tau)$ of the free type and free size variables of τ, are defined inductively in the obvious way. Note, that $FSV(\mathsf{List}^0(\mathsf{List}^m(\alpha))) = \emptyset$, since this type is equivalent to $\mathsf{List}^0(\mathsf{List}^0(\alpha))$. 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.

$$\tau^f ::= \tau_1^\circ \times \dots \times \tau_k^\circ \to \tau_{k+1}$$
$$\text{such that } FSV(\tau_{k+1}) \subseteq FSV(\tau_1^\circ) \cup \dots \cup FSV(\tau_k^\circ)$$

For *total* functions the following condition is necessary: *for all instantiations* $*$ *of size variables with themselves or zeros,* $FSV(*\tau_{n+1}) \subseteq FSV(*\tau_1^\circ) \cup \dots \cup FSV(*\tau_n^\circ)$. Consider, e.g., the first-order type $\mathsf{List}^n(\mathsf{List}^m(\alpha)) \to \mathsf{List}^m(\mathsf{List}^n(\alpha))$.

3

When $n = 0$ the input type degenerates to $\mathsf{List}^0(\mathsf{List}^0(\alpha))$, but in the output, the outer list must have size m, which in this case is unknown. Hence, this first-order type may be accepted without the condition on instantiations, only if a function of this type is *undefined* [1] on empty lists. In the previous example the type may correspond to an $n \times m$-matrix transposition function, in which case undefinedness on Nil would be interpreted as the exception "cannot transpose an empty matrix".

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 $\imath$ over integers, x denotes a program variable of a zero-order type, C stands for a constructor name and f for a function name.

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

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 $\mathsf{List}(\mathsf{Tree}(\mathsf{Int}))$ 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

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

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

---

[1] We use a nonterminating function to express undefinition.

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 $x_{ij} \colon T_{ij}$, 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 4.2) 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 2.3).

A *canonical size function* for $T(\overline{\alpha})$ is a size function where each c_i is $\mathsf{1}_i^r$, 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 $(1, \, length(l))$, since it is defined as:

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

The syntax distinguishes between zero-order let-binding of variables (`let`) and first-order letfun-binding of functions (`letfun`). 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 4).

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, where $vars(D) \subseteq FSV(\tau_1^{\circ} \times \ldots \times \tau_k^{\circ})$, 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. When D is empty we will write $\Gamma \vdash_{\Sigma} e \colon \tau$. 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. We write the

union of the constraints c_1 and c_2 as c_1, c_2, and we write Γ_1, Γ_2 to denote the union of the contexts Γ_1 and Γ_2, provided $dom(\Gamma_1) \cap dom(\Gamma_1) = \emptyset$.

The typing rules for the language, excluding the ones for data types, are shown in Figure 1. The FUNAPP rule needs some comments on its notation:

$$\frac{}{D;\ \Gamma \vdash_\Sigma b\colon \texttt{Bool}} \ \text{BCONST} \qquad \frac{}{D;\ \Gamma \vdash_\Sigma \imath\colon \texttt{Int}} \ \text{ICONST}$$

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

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

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

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

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

Fig. 1. Typing rules excluding the ones for data types.

$\tau[\tau_1^\circ := \tau_1' \ldots \tau_k^\circ := \tau_k']$ is the simultaneous substitution in τ of τ_i° by τ_i' for $i = 1..k$. This is done as follows:

1. Check that the underlying type (i.e., the type without the size annotations) of τ_i° and τ_i' are the same, except for the type variables.
2. Check that every type variable in each τ_i° is substituted by the same zero-order type by each τ_i', and substitute them in τ.
3. Substitute in τ the size variable of τ_i° by the corresponding size expression in τ_i'. It may happen that the same size variable appears in different types τ_i° and τ_j°, and that it is substituted by size expressions p_i and p_j, respectively. In such a case, replace the size variable by p_i and add the equation $\mathsf{p}_i = \mathsf{p}_j$ to C (which is initially empty).

As example, consider the last step in type checking *append* (see [19]).

$$\frac{\begin{array}{c}\Sigma(\mathsf{Cons}) = \alpha' \times \mathsf{List}^{n'}(\alpha') \to \mathsf{List}^{n'+1}(\alpha') \\ D \vdash \mathsf{List}^{n+m}(\alpha) = \mathsf{List}^{n'+1}(\alpha')[\alpha'/\alpha, \mathsf{List}^{n'}(\alpha')/\mathsf{List}^{(n-1)+m}(\alpha)]\end{array}}{h\colon \alpha,\ z\colon \mathsf{List}^{(n-1)+m}(\alpha) \vdash_\Sigma \mathsf{Cons}(h,\ z)\colon \mathsf{List}^{n+m}(\alpha)} \ \text{FUNAPP}$$

To do the substitutions we follow the 3 steps described before. As a first step we check the consistency of the underlying types of the *actuals* and the *formals*.

Since we omit type variables, there is nothing to check in α'/α; and is obvious that $\mathsf{List}^{n'}$ and $\mathsf{List}^{(n-1)+m}$ have the same underlying type, List. Then we check that each type variable is instantiated to same value, which is true since in both cases α' is instantiated to α. We replace α' by α in $\mathsf{List}^{n'+1}(\alpha')$. Finally we replace n' by $(n-1)+m$ to get the entailment $\vdash \mathsf{List}^{n+m}(\alpha) = \mathsf{List}^{(n-1)+m+1}(\alpha)$. Since there is only one substitution of n', the set of equations C is empty.

The implicit side-conditions on FUNAPP, i.e., the checks of consistency of the underlying types made in the first two steps, will be omitted in the following examples because they are part of conventional type checking. We will concentrate on the checking the entailments about size expressions, for instance, in the *append* example, we will just write the entailment $D \vdash n + m = (n - 1) + m + 1$.

The set C is used e,g., when type checking a function to do matrix multiplications: $\mathsf{List}^n(\mathsf{List}^k(\mathtt{Int})) \times \mathsf{List}^k(\mathsf{List}^m(\mathtt{Int})) \to \mathsf{List}^n(\mathsf{List}^m(\mathtt{Int}))$. If the first k is instantiated with p_1 and then the second with p_2, we substitute k with say p_1, but we add the requirement $D \vdash p_1 = p_2$.

In [18] we defined a type system for a similar language, whose only data type were lists and integers. The typing rules for calculating the size of lists were "hardcoded" in the type system by the typing rules in Figure 2.

$$\frac{D \vdash p = 0}{D;\ \Gamma \vdash_\Sigma \mathsf{Nil} \colon \mathsf{List}^p(\tau)}\ \text{N\textsc{il}}$$

$$\frac{D \vdash p = q + 1}{D;\ \Gamma,\ hd \colon \tau,\ tl \colon \mathsf{List}^q(\tau) \vdash_\Sigma \mathsf{Cons}(hd,\ tl) \colon \mathsf{List}^p(\tau)}\ \text{C\textsc{ons}}$$

$$\frac{\begin{array}{c} D,\ p = 0;\ \Gamma,\ x \colon \mathsf{List}^p(\tau) \vdash_\Sigma e_\mathsf{Nil} \colon \tau' \qquad hd,\ tl \notin dom(\Gamma) \\ D;\ \Gamma,\ x \colon \mathsf{List}^p(\tau),\ hd \colon \tau,\ tl \colon \mathsf{List}^{p-1}(\tau) \vdash_\Sigma e_\mathsf{Cons} \colon \tau' \end{array}}{D;\ \Gamma,\ x \colon \mathsf{List}^p(\tau) \vdash_\Sigma \begin{array}{l} \mathtt{match}\ x\ \mathtt{with}\ |\ \mathsf{Nil} \Rightarrow e_\mathsf{Nil} \\ \qquad\qquad\qquad |\ \mathsf{Cons}(hd,\ tl) \Rightarrow e_\mathsf{Cons} \end{array} \colon \tau'}\ \text{M\textsc{atch}}$$

Fig. 2. Typing rules for lists.

The main contribution of this work is to extend the type system to cope with other algebraic data types.

2.2 Example: Binary Trees

Consider the following definition of binary trees:

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

The canonical size function for Tree is:

$$s_{\mathsf{Tree}}(\mathsf{Empty}) \qquad = (1,\,0)$$
$$s_{\mathsf{Tree}}(\mathsf{Node}(v,\,l,\,r)) = (0,\,1) + s_{\mathsf{Tree}}(l) + s_{\mathsf{Tree}}(r)$$

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:

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

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.

$$\frac{\begin{array}{c} D,\,(e,\,n) = (1,\,0);\ \Gamma,\,t\colon \mathsf{Tree}^{e,n}(\tau) \vdash_{\Sigma} e_{\mathsf{Empty}}\colon \tau' \\[2pt] D,\,(e,\,n) = (0,\,1) + (e_1,\,n_1) + (e_2,\,n_2);\ \Gamma, \\ t\colon \mathsf{Tree}^{e,n}(\tau),\,v\colon\tau,\,l\colon \mathsf{Tree}^{e_1,n_1}(\tau),\,r\colon \mathsf{Tree}^{e_2,n_2}(\tau) \vdash_{\Sigma} e_{\mathsf{Node}}\colon \tau' \\[2pt] e_1,\,e_2,\,n_1\,n_2 \notin vars(D) \qquad v,\,l,\,r \notin dom(\Gamma) \end{array}}{D;\ \Gamma,\,t\colon \mathsf{Tree}^{e,n}(\tau) \vdash_{\Sigma} \begin{array}{l} \texttt{match}\ t\ \texttt{with}\ |\ \mathsf{Empty} \Rightarrow e_{\mathsf{Empty}} \\ \qquad\qquad\ |\ \mathsf{Node}(v,\,l,\,r) \Rightarrow e_{\mathsf{Node}} \end{array}\colon \tau'}\ \textsc{MTree}$$

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 $\texttt{let}$ 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 $\mathsf{undefined}$ to indicate a non-terminating expression with the required type.

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

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

$$
\dfrac{\dfrac{\begin{array}{l}(e_1, n_1) = \\ (e_{11} + e_{12},\, n_{11} + n_{12} + 1)\end{array} \;\vdash\; \begin{array}{l}(e_1 + e_2,\, n_1 + n_2 + 1) = \\ (e_{11} + e_{12} + e_2,\, n_{11} + (n_{12} + n_2 + 1) + 1)\end{array}}{\begin{array}{l}(e_1, n_1) = \\ (e_{11} + e_{12},\, n_{11} + n_{12} + 1); \\ v,\, v_1\colon \alpha,\, l\colon \mathsf{Tree}^{e_1,n_1}(\alpha), \\ l_1\colon \mathsf{Tree}^{e_{11},n_{11}}(\alpha), \\ r_1\colon \mathsf{Tree}^{e_{12},n_{12}}(\alpha)\end{array} \;\vdash_\Sigma\; \begin{array}{l}\mathsf{Node}(v_1, l_1, \\ \mathsf{Node}(v, r_1, r))\end{array} \colon \mathsf{Tree}^{e_1+e_2,n_1+n_2+1}(\alpha)}\;\text{NODE}}{\begin{array}{l}v\colon \alpha,\, l\colon \mathsf{Tree}^{e_1,n_1}(\alpha), \\ r\colon \mathsf{Tree}^{e_2,n_2}(\alpha)\end{array} \;\vdash_\Sigma\; \mathtt{match}\ l\ldots \colon \mathsf{Tree}^{e_1+e_2,n_1+n_2+1}(\alpha)}\;\text{MTREE}
$$

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

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

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.

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

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

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

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:

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

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:

$$\textit{build\_bal\_tree}(xs) : \mathsf{List}^n(\alpha) \to \mathsf{Tree}^{n+1,n}(\alpha) =$$
$$\mathtt{match}\ xs\ \mathtt{with}\ |\ \mathsf{Nil} \Rightarrow \mathsf{Empty}$$
$$|\ \mathsf{Cons}(hd,\, tl) \Rightarrow \textit{insert}(hd,\, \textit{build\_bal\_tree}(tl))$$

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:

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

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

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

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*.

For Tree it does not make sense to count both constructors because if e and n are the number of Empty and Node constructors in any Tree, respectively, it holds that $e = n + 1$. However, in general there is no such relationship. As an example where counting different constructors is relevant, consider binary trees defined as:

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

Suppose that we annotate Tree with e, l and n representing the number of empties, leaves and nodes, respectively. A function that replaces all empties with a leaf has type $\alpha \times \mathsf{Tree}^{e,l,n} \to \mathsf{Tree}^{0,e+l,n}$.

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

$$\mathtt{data}\ T(\overline{\alpha}) = C_1(\overline{\tau}_1(\overline{\alpha}))\ |\ \dots\ |\ C_r(\overline{\tau}_r(\overline{\alpha}))$$

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

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

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 $T(\overline{\alpha})$:

$$\frac{\begin{array}{l} D,\ \mathsf{p} = \mathsf{c}_i + \sum_{j=1}^{k_i} \mathsf{n}_{ij};\ \Gamma,\ x\colon T^\mathsf{p}(\overline{\tau}), \\ \quad x_{ij}\colon \gamma'_{ij}(T(\overline{\tau}))\ \text{ for } j = 1..k_i \end{array} \vdash_\Sigma e_i\colon \tau'\ \text{ for } i = 1..r \\ \mathsf{n}_{ij} \notin vars(D),\ x_{ij} \notin dom(\Gamma)\ \text{ for } i = 1..r, j = 1..k_i}{\begin{array}{ll} & \texttt{match } x \texttt{ with } \mid C_1(x_{11},\ldots,x_{1k_1}) \Rightarrow e_1 \\ D;\ \Gamma,\ x\colon T^\mathsf{p}(\overline{\tau}) \vdash_\Sigma & \quad\quad\quad\quad\quad\quad : \quad\quad\quad\quad\quad\quad : \tau' \\ & \mid C_r(x_{r1},\ldots,x_{rk_r}) \Rightarrow e_r \end{array}}\ \textsc{MatchT}$$

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.

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.

3 Inductive Types as Initial Algebras

In this section we recall categorical semantics of inductive types [4]. We recapitulate necessary notions from category theory and recall, that an inductive type is an initial algebra of the endofunctor corresponding to its constructors. We will see that the canonical size function of an polynomial inductive type is sensible and, moreover, it is an initiality homomorphism. Here we think of types as of *sets*.

11

Recall the notion of a functor. An *endofunctor* from *Set* to *Set*, with *endo* emphasising that the domain and the codomain of the given functor coincide, sends sets to sets and *set maps* to *set maps*. A functor satisfies three simple properties [4], e.g., $F(f \circ g) = F(f) \circ F(g)$. A simple example is a *list functor* L. It sends a set A to the set $\mathsf{List}(A)$ of lists of elements from A, and a function $f : A \to B$ to the function $L(f) := map(f) : \mathsf{List}(A) \to \mathsf{List}(B)$, where $map : (\alpha \to \alpha') \to \mathsf{List}(\alpha) \to \mathsf{List}(\alpha')$ is the usual lifting of a set map to a list map.

Now we define the notion of an *initial algebra*. First, consider an example. Let A be a set. The set $\mathsf{List}(A)$ of all lists of elements from A is defined inductively by two constructors:

$$\mathsf{Nil} \; : \mathbf{1} \to \mathsf{List}(A)$$
$$\mathsf{Cons} \colon A \times \mathsf{List}(A) \to \mathsf{List}(A)$$

where $\mathbf{1} = \{\star\}$ is a singleton set. So, the set $\mathsf{List}(A)$ may be presented as a pair

$$\Big(\mathsf{List}(A), \; [\mathsf{Nil}, \mathsf{Cons}] : (\mathbf{1} + A \times \mathsf{List}(A)) \to \mathsf{List}(A) \Big),$$

where $+$ denotes a coproduct (a disjoint sum) of two sets. One says that the lists form an *F-algebra of an endofunctor* F, where $F(X) = \mathbf{1} + A \times X$, which sends $f : X \to Y$ to $F(f) = (\mathbf{1} + A \times X) \to (\mathbf{1} + A \times Y)$, so, that $F(f)_{left}(\star) = \star$, and $F(f)_{right}(a, x) = (a, f(x))$.

Similarly, binary trees with nodes from A form an algebra of an endofunctor F', where $F'(X) = \mathbf{1} + A \times X \times X$:

$$\Big(\mathsf{Tree}(A), \; [\mathsf{Empty}, \mathsf{Node}] : (\mathbf{1} + A \times \mathsf{Tree}(A) \times \mathsf{Tree}(A)) \to \mathsf{Tree}(A) \Big)$$

An inductive type is not the only possible F-algebra, where F defined by the collection of its constructors. There are other F-algebras, that is, pairs of the form $\big(X, \; a_X : F(X) \to X \big)$ for some set X. Roughly speaking, an inductive type is the *minimal* F-algebra for F. To make this statement formal, we need to recapitulate the notion of a *homomorphism of F-algebras*. Let $\big(X, \; a_x : F(X) \to X \big)$ and $\big(Y, \; a_Y : F(Y) \to Y \big)$ be F-algebras. A set map $f : X \to Y$, such that:

$$
\begin{array}{ccc}
F(X) & \xrightarrow{\;a_X\;} & X \\
{\scriptstyle F(f)}\big\downarrow & & \big\downarrow{\scriptstyle f} \\
F(Y) & \xrightarrow{\;a_Y\;} & Y
\end{array}
$$

is called a homomorphism.

An inductive type corresponding to F is an *initial* F-algebra (or a least fixed point of F), i.e., *there is a unique homomorphism from it to any other F-algebra*. For example, lists of A form the initial algebra of $F(X) = \mathbf{1} + A \times X$.

What does initiality bring to size analysis? Consider e.g. an initiality homomorphism from lists of A to the F-algebra $(\mathbf{N}, \ a_{\mathbf{N}} : (\mathbf{1} + A \times \mathbf{N}) \to \mathbf{N})$, where

$$a_{\mathbf{N}, \, left}(\star) = 0$$
$$a_{\mathbf{N}, \, right}(a, \ n) = n + 1$$

This homomorphism is the function $length : \mathsf{List}(A) \to \mathbf{N}$:

$$
\begin{array}{ccc}
\mathbf{1} + A \times \mathsf{List}(A) & \xrightarrow{\ [\mathsf{Nil}, \mathsf{Cons}]\ } & \mathsf{List}(A) \\
\Big\downarrow {\scriptstyle F(length)} & & \Big\downarrow {\scriptstyle length} \\
\mathbf{1} + A \times \mathbf{N} & \xrightarrow{\ \ a_{\mathbf{N}}\ \ } & \mathbf{N}
\end{array}
$$

Indeed, one can easily show that the diagram above commutes. On one hand $length(\mathsf{Nil}(\star)) = length(\mathsf{Nil}) = 0$, and $length(\mathsf{Cons}(a, x)) = length(x) + 1$. On the other hand, $a_{\mathbf{N}, left}\big(F(length)_{left}(\star)\big) = a_{\mathbf{N}, left}(\star) = 0$ and similarly for the Cons part: $a_{\mathbf{N}, right}\big(F(length)_{right}(a, \ x)\big) = a_{\mathbf{N}, right}\big(a, \ length(x)\big) = length(x) + 1$.

3.1 The Canonical Size Function of Polynomial Inductive Types

We say that a size function for T is *sensible* if it returns the exact amount of each constructor of T that its argument contains. Recall that an inductive type is an initial algebra of the endofunctor corresponding to its constructors [4]. We will show that the canonical size function of an polynomial inductive type is sensible.

An endofunctor is called *polynomial* if it sends a set X to a finite coproduct of its finite products (with probably some other constant sets). If the endofunctor of an inductive type $T(\overline{\alpha})$ is polynomial, it follows from the construction of the endofunctor that the type (up to isomorphism) has a finite number of constructors (corresponding to the finite coproduct) of finite arity (finite products). Moreover, the types of arguments of constructors are either defined types that do not depend on $T(\overline{\alpha})$, type variables, or $T(\overline{\alpha})$. Such inductive types are called *parameterised polynomial inductive types* [9], where the word "parameterised" refers to polymorphism. It follows from the definition of an polynomial inductive type that its canonical size function is sensible. Indeed, in the definition of a canonical size function all the coefficients a_{ij} at inductive arguments are 1, and at noninductive arguments $a_{ij} = 0$. So, one easily shows by induction that the sensibility condition holds.

Note, that for a polymorphic type its endofunctor is parametric. For instance, in the parametric endofunctor for lists of type α is $F_\alpha(X) = \mathbf{1} + \alpha \times X$. Below we omit subscripts for parameters.

Let $T(\overline{\alpha})$ be a parameterised polynomial inductive type, and F be the corresponding endofunctor. Let the i-th constructor C_i, with $1 \le i \le r$, be of type $\overline{\alpha} \times \big(T(\overline{\alpha})\big)^k$ for some $k \ge 0$. Then the i-th injection of $F(X)$ is $\overline{\alpha} \times X^k$. We

13

want to show, that *the type's canonical size function is an initiality homomorphism* from the type to an F-algebra $\left(\mathbf{N}^r, \; a_{\mathbf{N}^r} : F(\mathbf{N}^r) \to \mathbf{N}^r\right)$. Formally, the i-the injection of set $F(\mathbf{N}^r)$ is $\overline{\alpha} \times (\mathbf{N}^r)^k$ and $a_{\mathbf{N}^r, \; i}$ sends $(\mathsf{a}, \; \mathsf{n}_1, \ldots, \; \mathsf{n}_k)$ to $\mathbf{1}_i + \sum_{j=1}^{k} \mathsf{n}_j$.

To ease reading we consider not the full homomorphism diagram, but only its part for the i-th injection:

$$
\begin{array}{ccc}
\overline{\alpha} \times \left(T(\overline{\alpha})\right)^k & \xrightarrow{\;\; C_i \;\;} & T(\overline{\alpha}) \\[2mm]
{\scriptstyle \overline{\alpha} \times s_T^k} \Big\downarrow & & \Big\downarrow {\scriptstyle s_T} \\[2mm]
\overline{\alpha} \times (\mathbf{N}^r)^k & \xrightarrow{\;\; a_{\mathbf{N}^r, \; i} \;\;} & \mathbf{N}^r
\end{array}
$$

It trivially commutes:

$$s_T(C_i(\mathsf{a}, \; x_1, \ldots, \; x_k)) = \mathbf{1}_i + \sum_{j=1}^{k} s_T(x_j)$$
and
$$a_{\mathbf{N}^r, \; i}\left((\overline{\alpha} \times s_T^k)(\mathsf{a}, \; x_1, \ldots, \; x_k)\right) =$$
$$a_{\mathbf{N}^r, \; i}\left(\mathsf{a}, \; s_T(x_1), \ldots, s_T(x_k)\right) = \mathbf{1}_i + \sum_{j=1}^{k} s_T(x_j)$$

4 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.

4.1 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 [18]. 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. Note that cyclic structures may be studied as, for instance, in the paper of Hofmann and Jost [12] about heap-space analysis for a subset of Java.

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. To avoid technical overhead with semantics of null-ary constructors, we do not consider a null-address NULL as a program value. The problem is that a type may have more than one null-ary constructor. If one had a NULL-address, then, to avoid ambiguity, one of the null-ary constructors would have been associated with NULL, whereas the others would have been placed in some locations. This would have made the proofs non-uniform. 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 $C_{i\_}field_1$. Thus, the definitions and proofs for null-ary and non null-ary constructors will be very similar. The reason to introduce a "fake" field for null-ary constructors is to make the proofs more uniform. Formally:

$$Val\ v\ ::=\ \imath\ |\ b\ |\ \ell \qquad \ell \in Loc \qquad \imath \in \text{Int} \qquad b \in \text{Bool}$$
$$Heap\ h\ :\ Loc \rightharpoonup Field \rightharpoonup Val \qquad ObjHeap\ oh\ :\ Loc \rightharpoonup Constructor$$

We will write $h[\ell.field := v]$ 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 $v \models_{\tau^\bullet}^{h;oh} w$. Integer and boolean constants interpret themselves, and locations are interpreted as non-cyclic structures:

$\imath \models_{\text{Int}}^{h;oh} \imath \qquad\qquad\qquad b \models_{\text{Bool}}^{h;oh} b$

$\ell \models_{T^c(\tau^\bullet)}^{h;oh} C \qquad\qquad$ if C is a null-ary constructor of T, $\ell \in dom(h)$, $oh(\ell) = C$
$\qquad\qquad\qquad\qquad\qquad$ and the constant vector c is the size of C

$\ell \models_{\tau^\bullet}^{h;oh} C(w_1, \ldots, w_k)$ if $\ell \in dom(h)$, $oh(\ell) = C$
$\qquad\qquad\qquad\qquad C: \tau_1^\bullet \times \ldots \times \tau_k^\bullet \rightarrow \tau^\bullet$ (i.e. it is a ground instance),
$\qquad\qquad\qquad\qquad \tau^\bullet = T^{n^0}(\overline{\tau^{\bullet\prime}})$ for some $\overline{\tau^{\bullet\prime}}$, $n^0 = size_T(C(w_1, \ldots, w_k))$,
$\qquad\qquad\qquad\qquad$ and for all $1 \leq j \leq k$: $h.\ell.C_{\_}field_j \models_{\tau_j^\bullet}^{h|_{dom(h)\backslash\{\ell\}};\ oh|_{dom(oh)\backslash\{\ell\}}} w_j$

where $h|_{dom(h)\backslash\{\ell\}}$ 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

15

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, $\mathcal{C} \vdash e \rightsquigarrow v$; h'; oh' informally means that at a store s, a heap h, its object heap oh and with the set $\mathcal{C}$ 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 $\mathcal{C}$ from function names to bodies for the functions definitions encountered, the operational semantics of expressions is defined by the following rules:

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

$$\frac{C_i \text{ is a 0-ary constructor} \qquad \ell \notin dom(h)}{s;\, h;\, oh,\, \mathcal{C} \vdash C_i \rightsquigarrow \ell;\, h[\ell.C_{i\_}\,field_1 := i];\, oh[\ell := C_i]} \text{ OSCons} - 0$$

$$\frac{s(x_1) = v_1, \ldots, s(x_k) = v_k \qquad \ell \notin dom(h)}{s;\, h;\, oh,\, \mathcal{C} \vdash C(x_1,\ldots,x_k) \rightsquigarrow \ell;\, h[\ell.C\_\,field_1 := v_1,\ldots,\ell.C\_\,field_k := v_k];\quad oh[\ell := C]}$$
$$\text{OSCons}$$

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

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

$$\frac{s;\, h;\, oh,\, \mathcal{C} \vdash e_1 \rightsquigarrow v_1;\, h_1;\, oh_1 \qquad s[x := v_1];\, h_1;\, oh_1,\, \mathcal{C} \vdash e_2 \rightsquigarrow v;\, h';\, oh'}{s;\, h;\, oh,\, \mathcal{C} \vdash \text{let } x = e_1 \text{ in } e_2 \rightsquigarrow v;\, h';\, oh'} \text{ OSLet}$$

$$\frac{\begin{array}{c} oh(s(x)) = C_i \\ C_i \text{ is a 0-ary constructor in the collection } C_{i'},\, 1 \leq i' \leq r \\ s;\, h;\, oh,\, \mathcal{C} \vdash e_i \rightsquigarrow v;\, h';\, oh' \end{array}}{s;\, h;\, oh,\, \mathcal{C} \vdash \left\{\begin{array}{l} \texttt{match } x \texttt{ with } \mid C_1(x_{11},\ldots,x_{1k_1}) \Rightarrow e_1 \\ \qquad\qquad \vdots \\ \mid C_r(x_{r1},\ldots,x_{rk_r}) \Rightarrow e_r \end{array}\right\} \rightsquigarrow v;\, h';\, oh'} \text{ OSMatch} - C_i - 0$$

$$\frac{\begin{array}{c} oh(s(x)) = C_i \qquad h.s(x).C_{i\_}\,field_1 = v_1,\,..\,,h.s(x).C_{i\_}\,field_{k_i} = v_{k_i} \\ s[x_1 := v_1,\ldots,x_{k_i} := v_{k_i}];\, h;\, oh,\, \mathcal{C} \vdash e_i \rightsquigarrow v;\, h';\, oh' \end{array}}{\begin{array}{ll} & \texttt{match } x \texttt{ with } \mid C_1(x_{11},\ldots,x_{1k_1}) \Rightarrow e_1 \\ s;\, h;\, oh,\, \mathcal{C} \vdash & \qquad\qquad \vdots \qquad\qquad\qquad\qquad \rightsquigarrow v;\, h';\, oh' \\ & \mid C_r(x_{r1},\ldots,x_{rk_r}) \Rightarrow e_r \end{array}} \text{ OSMatch} - C_i$$

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

$$\frac{\mathcal{C}(f) = (y_1,\ldots,y_n) \times e_f \qquad FV(e_f) \subseteq \overline{y} \\ [y_1 := s(x_1),\ldots,y_n := s(x_n)];\ h;\ oh,\ \mathcal{C}\ \vdash\ e_f\ \rightsquigarrow\ v;\ h';\ oh'}{s;\ h;\ oh,\ \mathcal{C}\ \vdash\ f(x_1,\ldots,x_n)\ \rightsquigarrow\ v;\ h';\ oh'}\ \text{OSFunApp}$$

Let a valuation $\epsilon : SizeVar \to \mathcal{Z}$ map size variables to concrete sizes (integer numbers) and an instantiation $\eta : TypeVar \to \tau^\bullet$ 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: $\epsilon(p[+,-,*]p) = \epsilon(p)[+,-,*]\epsilon(p)$ and $\eta(\epsilon(T^{\mathsf{p}}(\tau))) = T^{\epsilon(\mathsf{p})}(\eta(\tau))$.

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:

$$Valid_{\mathsf{val}}(v,\ \tau^\bullet,\ h;\ oh) \qquad = \exists\ w.\ v \models_{\tau^\bullet}^{h,\ oh} w$$

$$Valid_{\mathsf{store}}(vars,\ \Gamma,\ s,\ h; oh) = \forall\ x \in vars.\ Valid_{\mathsf{val}}(s(x),\ \Gamma(x),\ h,\ oh)$$

Now, stating soundness of the type system is straightforward:

Theorem 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* $\eta(\epsilon(\Gamma))$ *then the output value is meaningful w.r.t the type* $\eta(\epsilon(\tau))$:

$$\forall\ \eta, \epsilon.\ Valid_{\mathsf{store}}(FV(e),\ \eta(\epsilon(\Gamma)),\ s,\ h,\ oh)\ \Longrightarrow\ Valid_{\mathsf{val}}(v,\ \eta(\epsilon(\tau)),\ h',\ oh')$$

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

We assume *benign sharing* of variables [11]. 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 [3]. The discussion on this topic is beyond the scope of this report.

To formalise the notion of benign sharing we introduce a *footprint* function $\mathcal{R} : Heap \times ObjHeap \times Val \longrightarrow \mathcal{P}(Loc)$, which computes the set of locations accessible in a given heap h, with a corresponding object heap oh, with $dom(oh) = dom(h)$ from a given value:

$$\mathcal{R}(h,\ oh,\ c) = \emptyset$$

$$\mathcal{R}(h,\ oh,\ \ell) = \begin{cases} \emptyset,\ \ if\ \ell \notin dom(h) \\ \{\ell\} \cup \bigcup_{j=1}^{k} \mathcal{R}(h|_{dom(h)\backslash\{\ell\}},\ oh|_{dom(oh)\backslash\{\ell\}},\ h.\ell.C\_field_j), \\ if\ \ oh(\ell) = C \end{cases}$$

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

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

$$
\frac{
\begin{array}{c}
s;\ h;\ oh,\ \mathcal{C}\ \vdash\ e_1\ \rightsquigarrow\ v_1;\ h_1;\ oh_1 \\
s[x := v_1];\ h_1;\ oh_1,\ \mathcal{C}\ \vdash\ e_2\ \rightsquigarrow\ v;\ h';\ oh' \\
h|_{\mathcal{R}(h,\ oh,\ s|_{FV(e_2)})} = h_1|_{\mathcal{R}(h,\ oh,\ s|_{FV(e_2)})} \\
oh|_{\mathcal{R}(h,\ oh,\ s|_{FV(e_2)})} = oh_1|_{\mathcal{R}(h,\ oh,\ s|_{FV(e_2)})}
\end{array}
}{
s;\ h;\ oh,\mathcal{C}\ \vdash\ \texttt{let}\ x = e_1\ \texttt{in}\ e_2\ \rightsquigarrow\ v;\ h';\ oh'
}\ \text{OSLET}
$$

Lemmas and soundness proof.

Lemma 1 (A program value's footprint is in the heap).
$\mathcal{R}(h,\ oh,\ v) \subseteq dom(h)$.

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

$dom(h) = \emptyset$: Then $\mathcal{R}(h,\ oh,\ v) = \emptyset$ is trivially a subset of $dom(h)$.
$dom(h) \neq \emptyset$:

$v = \imath$ Then, $\mathcal{R}(h,\ oh,\ v) = \emptyset$, 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
$\mathcal{R}(h,\ oh,\ \ell) = \{\ell\} \cup \bigcup_{j=1}^k \mathcal{R}(h|_{dom(h) \setminus \{\ell\}},\ oh|_{dom(oh) \setminus \{\ell\}},\ h.l.C\_field_j)$.
Applying the induction hypotheses we derive that

$$\mathcal{R}(h|_{dom(h) \setminus \{\ell\}},\ oh|_{dom(oh) \setminus \{\ell\}},\ h.\ell.C\_field_j) \subseteq dom(h|_{dom(h) \setminus \{\ell\}})$$

for all $1 \leq j \leq k$. Hence, $\mathcal{R}(h,\ oh,\ l) \subseteq dom(h)$. $\qquad\square$

Lemma 2 (Extending a heap does not change the footprints of program values). *If $\ell \notin dom(h)$, $h' = h[\ell.C\_field_1 := v_1, .., \ell.C\_field_k := v_k]$ for some $v_1, .., v_k$ and $oh' = oh[\ell := C]$ then for any $v \neq \ell$ one has $\mathcal{R}(h,\ oh,\ v) = \mathcal{R}(h',\ oh',\ v)$.*

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

$dom(h) = \emptyset$: Because $h' = [\ell.C\_field_1 := v_1, .., \ell.C\_field_k := v_k]$ and $v \neq \ell$ we
have $v \notin dom(h')$. Therefore, $\mathcal{R}(h,\ oh,\ v) = \emptyset = \mathcal{R}(h',\ oh',\ v)$.
$dom(h) \neq \emptyset$: We proceed by case distinction on v.

$v = \imath$ then $\mathcal{R}(h,\ oh,\ v) = \emptyset = \mathcal{R}(h',\ oh',\ v)$.

$v = \ell'$: Let $\ell' \notin dom(h)$. Then $\mathcal{R}(h,\ oh,\ \ell') = \emptyset$ and $\mathcal{R}(h',\ oh',\ \ell') = \emptyset$
because $\ell' \notin dom(h')$ as well, since $\ell \neq \ell'$ and $dom(h') = dom(h) \cup \ell$.
Let $\ell \in dom(h)$. From the definition of $\mathcal{R}$ we get

$$\mathcal{R}(h,\ oh,\ \ell') = \{\ell'\} \cup \bigcup_{j=1}^{k'} \mathcal{R}(h|_{dom(h) \setminus \{\ell'\}},\ oh|_{dom(oh) \setminus \{\ell'\}},\ h.\ell'.C'\_field_j).$$

where $oh(\ell') = C'$.
Due to $h'(\ell') = h(\ell')$ and

$$h'|_{dom(h')\backslash\{\ell'\}} = h|_{dom(h)\backslash\{\ell'\}}[h.\ell.C\_field_1 := v_1, .., h.\ell.C\_field_k := v_k],$$

and the induction assumption one has

$$\mathcal{R}(h|_{dom(h)\backslash\{\ell'\}}, \; oh|_{dom(oh)\backslash\{\ell'\}}, \; h.\ell'.C'\_field_j) = \\ \mathcal{R}(h'|_{dom(h')\backslash\{\ell'\}}, \; oh'|_{dom(oh')\backslash\{\ell'\}}, \; h'.\ell'.C'\_field_j)$$

for all $1 \leq j \leq k'$. So,

$$\mathcal{R}(h', \; oh', \; \ell') = \\ = \{\ell'\} \; \cup \; \bigcup_{j=1}^{k'} \mathcal{R}(h'|_{dom(h')\backslash\{\ell'\}}, \; oh'|_{dom(oh')\backslash\{\ell'\}}, \; h'.\ell'.C'\_field_j) \\ = \{\ell'\} \; \cup \; \bigcup_{j=1}^{k'} \mathcal{R}(h|_{dom(h)\backslash\{\ell'\}}, \; oh|_{dom(oh)\backslash\{\ell'\}}, \; h.\ell'.C'\_field_j) \\ = \mathcal{R}(h, \; oh, \; \ell').$$

Lemma 3 (Validity for Union of Variable Sets). *For all stores s and ground contexts Γ the predicate $Valid_{\mathsf{store}}(vars_1 \cup vars_2, \Gamma, s, h; oh)$ is true if and only if both $Valid_{\mathsf{store}}(vars_1, \Gamma, s, h; oh)$ and $Valid_{\mathsf{store}}(vars_2, \Gamma, s, h; oh)$ are true.*

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

Lemma 4 (Extending heaps preserves model relations).
For all heaps h and h', if $h'|_{dom(h)} = h$ and $oh'|_{dom(h)} = oh$ then $v \models_{\tau^\bullet}^{h;\,oh} w$ implies $v \models_{\tau^\bullet}^{h';\,oh'} w$.

Proof.
The lemma is proved by induction on the definition of $\models$.

$v = \imath$: In this case $\tau^\bullet = \mathtt{Int}$ and $w = \imath$, hence $v \models_{\tau^\bullet}^{h';\,oh'} w$ by the definition.

$v = \ell$ **and a null-ary constructor:**

$$\ell \models_{T^c(\overline{\tau^{\bullet\prime}})}^{h;\,oh} C$$

By the definition we trivially obtain $\ell \models_{T^c(\overline{\tau^{\bullet\prime}})}^{h';\,oh'} C$.

$v = \ell$ **and a non null-ary constructor:** In this case $\tau^\bullet = T^{\mathsf{n}^0}(\overline{\tau^{\bullet\prime}})$ for some n^0 and

$$\ell \models_{T^{\mathsf{n}^0}(\overline{\tau^{\bullet\prime}})}^{h;\,oh} C(w_1, \ldots, w_k)$$

for some w_j, such that

$$\ell \in dom(h), \quad oh(\ell) = C \\ \mathsf{n}^0 = size_T(C(w_1, \ldots, w_k)) \\ C : \tau_1^\bullet \times \ldots \times \tau_k^\bullet \to \tau^\bullet \\ h.\ell.C\_field_1 \models_{\tau_1^\bullet}^{h|_{dom(h)\backslash\{\ell\}};\,oh|_{dom(oh)\backslash\{\ell\}}} w_1 \\ \ldots \\ h.\ell.C\_field_k \models_{\tau_k^\bullet}^{h|_{dom(h)\backslash\{\ell\}};\,oh|_{dom(oh)\backslash\{\ell\}}} w_k$$

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

$$h'|_{dom(h')\setminus\{\ell\}}|_{dom(h|_{dom(h)\setminus\{\ell\}})}$$
$$= h'|_{dom(h')\setminus\{\ell\}}|_{dom(h)\setminus\{\ell\}}$$
$$= h'|_{dom(h)\setminus\{\ell\}} = h|_{dom(h)\setminus\{\ell\}}$$

Thus, we apply the assumption $oh'|_{dom(h)} = oh$, the induction assumption and $h.\ell = h'.\ell$ to obtain

$$\ell \in dom h', \quad oh'(\ell) = C$$
$$\mathsf{n}^0 = size_T(C(w_1, \ldots, w_k))$$
$$h'.\ell.C\_field_1 \models_{\tau_1^\bullet}^{h'|_{dom(h')\setminus\{\ell\}};\, oh'|_{dom(oh')\setminus\{\ell\}}} w_1$$
$$\ldots$$
$$h'.\ell.C\_field_k \models_{\tau_k^\bullet}^{h'|_{dom(h')\setminus\{\ell\}};\, oh'|_{dom(oh')\setminus\{\ell\}}} w_k$$

Then, $\ell \models_{\tau^\bullet}^{h',\, oh'} C(w_1, \ldots, w_k)$ by the definition. $\qquad\square$

Lemma 5 (Values only depend on values at their footprints).
For v, h, w, and $\tau^\bullet$, the relation $v \models_{\tau^\bullet}^{h,\, oh} w$ implies

$$v \models_{\tau^\bullet}^{h|_{\mathcal{R}(h,\, oh,\, v)},\, oh|_{\mathcal{R}(h,\, oh,\, v)}} w$$

Proof. The lemma is proved by induction on the definition of $\models$.

$v = \imath$: then $w = \imath$ and $v \models_{\tau^\bullet}^{h|_{\mathcal{R}(h,\, oh,\, v)},\, oh|_{\mathcal{R}(h,\, oh,\, v)}} \imath$.

$v = \ell$ **and a null-ary constructor:** then $w = C_i$ is a null-ary constructor of $\tau^\bullet = T^{\mathsf{c}_i}(\overline{\tau^{\bullet}}')$ and

$$v \models_{\tau^\bullet}^{h|_{\mathcal{R}(h,\, oh,\, v)},\, oh|_{\mathcal{R}(h,\, oh,\, v)}} C_i$$

$v = \ell$ **and a non null-ary constructor:** then $\tau^\bullet = T^{\mathsf{n}^0}(\overline{\tau^{\bullet}}')$ for some $\overline{\tau^{\bullet}}'$, that $w = C(w_1, \ldots, w_k)$ for some $w_1, \ldots, w_k$

$$\ell \in dom(h), \quad oh(\ell) = C$$
$$\mathsf{n}^0 = size_T(C(w_1, \ldots, w_k))$$
$$C: \tau_1^\bullet \times \ldots \times \tau_k^\bullet \to \tau^\bullet$$
$$h.\ell.C\_field_1 \models_{\tau_1^\bullet}^{h|_{dom(h)\setminus\{\ell\}},\, oh|_{dom(oh)\setminus\{\ell\}}} w_1,$$
$$\ldots$$
$$h.\ell.C\_field_k \models_{\tau_k^\bullet}^{h|_{dom(h)\setminus\{\ell\}},\, oh|_{dom(oh)\setminus\{\ell\}}} w_k,$$

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

$$\ell \in dom(h), \quad oh(\ell) = C$$
$$C \colon \tau_1^\bullet \times \ldots \times \tau_k^\bullet \to \tau^\bullet$$

$$h.\ell.C\_field_1 \models_{\tau_1^\bullet} \left\{ \begin{array}{l} h|_{dom(h)\backslash\{\ell\}}|_{\mathcal{R}(h|_{dom(h)\backslash\{\ell\}}, \, oh|_{dom(oh)\backslash\{\ell\}}, \, h.\ell.C\_field_1)}, \\ oh|_{dom(oh)\backslash\{\ell\}}|_{\mathcal{R}(h|_{dom(h)\backslash\{\ell\}}, \, oh|_{dom(oh)\backslash\{\ell\}}, \, h.\ell.C\_field_1)} \end{array} \right\} w_1,$$

$$\ldots$$

$$h.\ell.C\_field_k \models_{\tau_k^\bullet} \left\{ \begin{array}{l} h|_{dom(h)\backslash\{\ell\}}|_{\mathcal{R}(h|_{dom(h)\backslash\{\ell\}}, \, oh|_{dom(oh)\backslash\{\ell\}}, \, h.\ell.C\_field_k)}, \\ oh|_{dom(oh)\backslash\{\ell\}}|_{\mathcal{R}(h|_{dom(h)\backslash\{\ell\}}, \, oh|_{dom(oh)\backslash\{\ell\}}, \, h.\ell.C\_field_k)} \end{array} \right\} w_k,$$

Due to $\mathcal{R}(h|_{dom(h)\backslash\{\ell\}}, \, oh|_{dom(oh)\backslash\{\ell\}}, \, h.\ell.C\_field_j) \subseteq dom(h) \backslash \{\ell\}$ (lemma 1) we have

$$h|_{dom(h)\backslash\{\ell\}}|_{\mathcal{R}(h|_{dom(h)\backslash\{\ell\}}, \, oh, \, h.\ell.C\_field_j)} =$$
$$= h|_{\mathcal{R}(h|_{dom(h)\backslash\{\ell\}}, \, oh|_{dom(oh)\backslash\{\ell\}}, \, h.\ell.C\_field_j)} =$$
$$= h|_{\mathcal{R}(h|_{dom(h)\backslash\{\ell\}}, \, oh|_{dom(oh)\backslash\{\ell\}}, \, h.\ell.C\_field_j)\backslash\{\ell\}}.$$

for all $1 \le j \le k$. Due to $\ell \in \mathcal{R}(h, \, oh, \, \ell)$, and lemma 4, with

$$\mathcal{R}(h|_{dom(h)\backslash\{\ell\}}, \, oh|_{dom(oh)\backslash\{\ell\}}, \, h.\ell.C\_field_j) \backslash \{\ell\} \subseteq \mathcal{R}(h, \, oh, \, \ell) \backslash \{\ell\}$$

we have

$$\ell \in dom(h_{\mathcal{R}(h, \, oh, \, \ell)}), \quad oh(\ell) = C$$
$$\mathsf{n}^0 = size_T(C(w_1, \ldots, w_k))$$
$$h|_{\mathcal{R}(h, \, oh, \, \ell)}.\ell.C\_field_1 \models_{\tau_1^\bullet}^{h|_{\mathcal{R}(h, \, oh, \, \ell)\backslash\{\ell\}}, \, oh|_{\mathcal{R}(h, \, oh, \, \ell)\backslash\{\ell\}}} w_1,$$
$$\ldots$$
$$h|_{\mathcal{R}(h, \, oh, \, \ell)}.\ell.C\_field_k \models_{\tau_k^\bullet}^{h|_{\mathcal{R}(h, \, oh, \, \ell)\backslash\{\ell\}}, \, oh|_{\mathcal{R}(h, \, oh, \, \ell)\backslash\{\ell\}}} w_k$$

Thus, $\ell \models_{\tau^\bullet}^{h|_{\mathcal{R}(h, \, oh, \, \ell)}, \, oh|_{\mathcal{R}(h, \, oh, \, \ell)}} C(w_1, \ldots, w_k)$. $\qquad\square$

Lemma 6 (Equality of the "meanings" of a program value in two heaps follows from the equality of the footprints).

If $h|_{\mathcal{R}(h, \, oh, \, v)} = h'|_{\mathcal{R}(h, \, oh, \, v)}$ *and* $oh|_{\mathcal{R}(h, \, oh, \, v)} = oh'|_{\mathcal{R}(h, \, oh, \, v)}$ *then* $v \models_{\tau^\bullet}^{h, \, oh} w$ *implies* $v \models_{\tau^\bullet}^{h', oh'} w$.

Proof. Assume $v \models_{\tau^\bullet}^{h, \, oh} w$. Lemma 5 states that this implies $v \models_{\tau^\bullet}^{h|_{\mathcal{R}(h, \, oh, \, v)}, \, oh|_{\mathcal{R}(h, \, oh, \, v)}} w$. From the assumption of the lemma we get $v \models_{\tau^\bullet}^{h'|_{\mathcal{R}(h, \, oh, \, v)}, \, oh'|_{\mathcal{R}(h, \, oh, \, v)}} w$. Now we apply lemma 4, which gives $v \models_{\tau^\bullet}^{h', oh'} w$. $\qquad\square$

Lemma 7 (ChangeStore).

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

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

Lemma 8 (SubsetFV).

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

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

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

Lemma 9 (Soundness). *For any s, h, oh, $\mathcal{C}$, e, v, h', oh' 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*

- $s;\, h;\, oh,\, \mathcal{C} \vdash e \rightsquigarrow v;\, h';\, oh'$,
- $D;\, \Gamma \vdash_\Sigma e : \tau$ *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 $s;\, h;\, oh,\, \mathcal{C} \vdash e \rightsquigarrow v;\, h';\, oh$, we fix some Γ, Σ, and τ, such that $D;\, \Gamma \vdash_\Sigma e : \tau$. We fix a valuation $\epsilon \in FV(\Gamma) \cup FV(\tau) \to \mathcal{Z}$, 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 $Valid_{\text{val}}(v, \tau_{\eta\epsilon}, h', oh')$ holds.

OSICons: In this case $v = \imath$ for some constant $c\imath$ and $\tau = \text{Int}$. Then, by the definition we have $\imath \models_{\text{Int}}^{h,\, oh} \imath$ and $Valid_{\text{val}}(v, \text{Int}, h' = h, oh' = oh)$.

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

$$Valid_{\text{val}}(s(x), \tau_{\eta\epsilon}, h' = h, oh' = oh)$$

OSCons-0: in this case $e = C_i$, where C_i is a null-ary constructor of some type T, $v = \ell \notin dom(h)$. From the type derivation we have that $\tau = T^{\text{P}}(\overline{\tau}')$ for some $\overline{\tau}'$, and, moreover, $D \vdash p = \mathsf{c}_i$. By the definition of $\models$ relation we have $\ell \models_{T^{c_i}(\overline{\tau}'_{\eta\epsilon})}^{h[\ell.C_i\_field_1 := i],\, oh[\ell := C_i]} C_i$, and therefore

$$D_\epsilon \vdash \ell \models_{T^{\text{P}_\epsilon}(\overline{\tau}'_{\eta\epsilon})}^{h[\ell.C_i\_field_1 := i],\, oh[\ell := C_i]} C_i$$

Thus,

$$Valid_{\text{val}}(v, T^{\text{P}_\epsilon}(\overline{\tau}')_{\eta\epsilon}, h' = h[\ell.C_i\_field_1 := i], oh' = oh[\ell := C_i])$$

OSCons: In this case $e = C(x_1, \ldots, x_k)$, $v = \ell \notin dom(h)$. From the typing rule we have that $\tau = T^{\mathsf{p}}(\overline{\tau}')$ for some $\overline{\tau}'$ and there exist τ_j, such that $x_j : \tau_j \subseteq \Gamma$, and one has the instance of C of type $\tau_1 \times \ldots \times \tau_k \to T^{\mathsf{p}}(\overline{\tau}')$. Moreover, $\tau_j = T_j^{\mathsf{p}_j}(*)$ for some p_j if τ_j takes part in the counting sizes, (otherwise think that it is equal to zero).

Since $Valid_{\mathsf{store}}(FV(e), \Gamma_{\eta\epsilon}, s, h, oh)$ there exist w_j such that $s(x_j) \models_{\tau_{j\,\eta\epsilon}}^{h,\,oh} w_j$. From the operational-semantics judgement we have $h' = h[\ell.C\_field_1 := s(x_1), \ldots, \ell.C\_field_k := s(x_k)]$. Therefore, $h'.\ell.C\_field_j \models_{\tau_{j\,\eta\epsilon}}^{h,\,oh} w_j$. It is easy to see that $h = h'|_{dom\,h'\backslash\{\ell\}}$ and similarly for oh'.

Thus,

$$h'.\ell.C\_field_j \models_{\tau_{j\,\eta\epsilon}}^{h'|_{dom\,h'\backslash\{\ell\}};\; oh'|_{dom\,oh'\backslash\{\ell\}}} w_j$$

From the typing rule we have that D implies $\mathsf{p} = \mathsf{c} + \sum_{j=1}^{k} a_j \mathsf{p}_j$. For the ground valuation we have

$$\mathsf{p}_\epsilon = \mathsf{c} + \sum_{j=1}^{k} a_j \mathsf{p}_{j\,\epsilon}$$

From the definition of $\models$ relation it follows that $\mathsf{p}_{j\,\epsilon} = size_{T_j}(w_j)$. From what follows that $\mathsf{p}_\epsilon = \mathsf{c} + \sum_{j=1}^{k} a_j size_{T_j}(w_j) = size_T(C(w_1, \ldots, w_k))$. This gives $\ell \models_{T^{\mathsf{p}}(\overline{\tau}')_{\eta\epsilon}}^{h',oh'} C(w_1, \ldots, w_k)$ and thus $Valid_{\mathsf{val}}(\ell, \tau_{\eta\epsilon}, h', oh')$.

OSIfFalse: In this case $e = \mathtt{if}\ x\ \mathtt{then}\ e_1\ \mathtt{else}\ e_2$ for some e_1, e_2, and x. Knowing that $D;\ \Gamma \vdash_\Sigma e_2 : \tau$, we apply the induction hypothesis to the derivation of $s;\ h;\ oh,\ \mathcal{C} \vdash e_2 \leadsto v;\ h';\ oh'$ to obtain

$$Valid_{\mathsf{store}}(FV(e_2), \Gamma_{\eta\epsilon}, s, h, oh) \implies Valid_{\mathsf{val}}(v, \tau_{\eta\epsilon}, h', oh')$$

From $FV(e_2) \subseteq FV(e)$, $Valid_{\mathsf{store}}(FV(e), \Gamma_{\eta\epsilon}, s, h, oh)$, and lemma 8 it follows that $Valid_{\mathsf{val}}(v, \tau_{\eta\epsilon}, h', oh')$.

OSIfTrue: In this case $e = \mathtt{if}\ x\ \mathtt{then}\ e_1\ \mathtt{else}\ e_2$ for some e_1, e_2, and x. Knowing that $D;\ \Gamma \vdash_\Sigma e_1 : \tau$, we apply the induction hypothesis to the derivation of $s;\ h;\ oh,\ \mathcal{C} \vdash e_1 \leadsto v;\ h';\ oh'$ to obtain

$$Valid_{\mathsf{store}}(FV(e_1), \Gamma_{\eta\epsilon}, s, h, oh) \implies Valid_{\mathsf{val}}(v, \tau_{\eta\epsilon}, h', oh')$$

From $FV(e_1) \subseteq FV(e)$, $Valid_{\mathsf{store}}(FV(e), \Gamma_{\eta\epsilon}, s, h, oh)$, and lemma 8 it follows that $Valid_{\mathsf{val}}(v, \tau_{\eta\epsilon}, h', oh')$.

OSLetFun: The result follows from the induction hypothesis for

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

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

OSLet: In this case $e = \mathtt{let}\ x = e_1\ \mathtt{in}\ e_2$ for some x, e_1, and e_2 and we have $s;\ h;\ oh,\ \mathcal{C} \vdash e_1 \leadsto v_1;\ h_1;\ oh_1$ and $s[x := v_1];\ h_1;\ oh,\ \mathcal{C} \vdash e_2 \leadsto v;\ h';\ oh$ for some v_1 and h_1. We know that $D;\ \Gamma \vdash_\Sigma e_1 : \tau'$, $x \notin \Gamma$ and $D;\ \Gamma, x : \tau' \vdash_\Sigma$

$e_2 \colon \tau$ for some τ'. Applying the induction hypothesis to the first branch gives $Valid_{\mathsf{store}}(FV(e_1), \Gamma_{\eta\epsilon}, s, h, oh) \implies Valid_{\mathsf{val}}(v_1, \tau'_{\eta\epsilon}, h_1, oh_1)$. Since $FV(e_1) \subseteq FV(e_1) \cup (FV(e_2) \backslash \{x\}) = FV(e)$ and $Valid_{\mathsf{store}}(FV(e), \Gamma_{\eta\epsilon}, s, h, oh)$ we have from lemma 8 that $Valid_{\mathsf{store}}(FV(e_1), \Gamma_{\eta\epsilon}, s, h, oh)$ holds and hence we have $Valid_{\mathsf{val}}(v_1, \tau'_{\eta\epsilon}, h_1, oh_1)$.

Now apply the induction hypothesis to the second branch to get

$$Valid_{\mathsf{store}}(FV(e_2), \Gamma_{\eta\epsilon} \cup \{x \colon \tau'\}_{\eta\epsilon}, s[x := v_1], h_1, oh_1) \implies Valid_{\mathsf{val}}(v, \tau_{\eta\epsilon}, h', oh').$$

Fix some $y \in FV(e_2)$. If $y = x$, then $Valid_{\mathsf{val}}(v_1, \tau'_{\eta\epsilon}, h_1, oh_1)$ implies $Valid_{\mathsf{val}}(s[x := v_1](x), \tau'_{\eta\epsilon}, h_1, oh_1)$. If $y \neq x$, then $s[x := v_1](y) = s(y)$. Because we know that sharing is benign, $h|_{\mathcal{R}(h, oh, s(y))} = h_1|_{\mathcal{R}(h, oh, s(y))}$, applying lemma 6 and then 8 we have that $s(y) \models^{h, oh}_{\Gamma_{\eta\epsilon}(y)} w_y$ implies $s(y) \models^{h_1, oh_1}_{\Gamma_{\eta\epsilon}(y)} w_y$ implies $s[x := v_1](y) \models^{h_1, oh_1}_{\Gamma_{\eta\epsilon}(y)} w_y$ and thus $Valid_{\mathsf{val}}(s[x := v_1](y), \Gamma_{\eta\epsilon}(y), h_1, oh_1)$. Hence, $Valid_{\mathsf{store}}(FV(e_2), \Gamma_{\eta\epsilon} \cup \{x \colon \tau'\}_{\eta\epsilon}, s[x := v_1], h_1, oh_1)$. Therefore, $Valid_{\mathsf{val}}(v, \tau_{\eta\epsilon}, h', oh')$.

OSMatch-0-ary: In this case the expression e has the form

$$e = \left\{ \begin{array}{l} \texttt{match } x \texttt{ with } \mid C_1(x_{11}, \ldots, x_{1k_1}) \Rightarrow e_1 \\ \qquad\qquad \vdots \\ \mid C_r(x_{r1}, \ldots, x_{rk_r}) \Rightarrow e_r \end{array} \right\}$$

The typing context has the form $\Gamma = \Gamma' \cup \{x \colon T^{\mathsf{p}}(\overline{\tau'})\}$ for some Γ', τ', p. The operational-semantics derivation gives $s(x) = \ell$, and C_i is a null-ary constructor of x-s type. Hence validity

$$s(x) \models^{h, oh}_{T^{\mathsf{p}}(\overline{\tau'})_{\eta\epsilon}} C_i$$

gives $\epsilon(\mathsf{p}) = \mathsf{c}_i$. From the typing derivation for $D; \Gamma \vdash_\Sigma e \colon \tau$ we know that $\mathsf{p} = \mathsf{c}_i, D; \Gamma' \vdash_\Sigma e_i \colon \tau$. Applying the induction hypothesis, with $D_\epsilon \wedge \mathsf{p}_\epsilon = \mathsf{c}_i$ then yields $Valid_{\mathsf{store}}(FV(e_i), \Gamma'_{\eta\epsilon}, s, h, oh) \implies Valid_{\mathsf{val}}(v, \tau_{\eta\epsilon}, h', oh')$. From $FV(e_i) \subseteq FV(e)$, $Valid_{\mathsf{store}}(FV(e), \Gamma_{\eta\epsilon}, s, h, oh)$ it follows that

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

OSMatch-C_i: In this case, again, the expression e has the form

$$e = \left\{ \begin{array}{l} \texttt{match } x \texttt{ with } \mid C_1(x_{11}, \ldots, x_{1k_1}) \Rightarrow e_1 \\ \qquad\qquad \vdots \\ \mid C_r(x_{r1}, \ldots, x_{rk_r}) \Rightarrow e_r \end{array} \right\}$$

The operational-semantics derivation gives $oh(s(x)) = C_i$. The typing context has the form $\Gamma = \Gamma' \cup \{x \colon T^{\mathsf{p}}(\overline{\tau'})\}$ for some Γ', $\overline{\tau'}$, p. Hence validity

$$s(x) \models^{h, oh}_{T^{\mathsf{p}}(\overline{\tau'})_{\eta\epsilon}} C_i(w_1, \ldots, w_k)$$

gives $h.s(x).C_i\_field_j \models^{h, oh}_{\tau_{j\,\eta\epsilon}} w_j$.

From the typing derivation for D; $\Gamma \vdash_\Sigma e \colon \tau$ we know that

$$D, \; \mathsf{p} = \mathsf{c}_i + \sum_{j=1}^{k_i} a_{ij}\mathsf{n}_{ij}, \; \Gamma', \; x \colon T^{\mathsf{p}}(\overline{\tau'}), \; x_{ij} \colon T^{n_{ij}} \vdash_\Sigma e_i \colon \tau$$

To apply the induction hypothesis we must extend the valuation ϵ to n_{ij} (call this extension ϵ') so that

$$D_{\epsilon'} \wedge \mathsf{p}_{\epsilon'} = \mathsf{c}_i + \sum_{j=1}^{k_i} a_{ij}\mathsf{n}_{ij_{\epsilon'}} \; \text{holds}$$

We assign $n_{ij} = size_{T_{ij}}(w_j)$, taken from the definition of $\models$-relation for $h.s(x).C_i\_field_j$. Then from the definition of $\models$-relation for $s(x)$ it follows that $p_{\epsilon'} = p_\epsilon = size_T(C(w_1, \ldots, w_k)) = \mathsf{c}_i + \sum_{j=1}^{k_i} a_{ij}\, size_{T_{ij}}(w_j) = \mathsf{c}_i + \sum_{j=1}^{k_i} a_{ij}\mathsf{n}_{ij_{\epsilon'}}$.

Applying the induction hypothesis, with $D \wedge \mathsf{p} = \mathsf{c}_i + \sum_{j=1}^{k_i} a_{ij}\mathsf{n}_{ij}$ with ϵ', η then yields

$Valid_{\mathsf{store}}(FV(e_i), (\Gamma'\; x \colon T^{\mathsf{p}}(\overline{\tau'}), \; x_{ij} \colon T^{n_{ij}})_{\eta\epsilon'}, \; s[..x_{ij} := h.s(x).C_i\_field_j, ..], \; h, \; oh)$
$\implies \; Valid_{\mathsf{val}}(v, \tau_{\eta\epsilon'}, h', oh')$

We must show that

$Valid_{\mathsf{store}}(FV(e_i), (\Gamma'\; x \colon T^{\mathsf{p}}(\overline{\tau'}), \; x_{ij} \colon T^{n_{ij}})_{\eta\epsilon'}, \; s[..x_{ij} := h.s(x).C_i\_field_j, ..], \; h, \; oh)$

It is easy to see, that $FV e_i) \subseteq FV(e) \cup \{x_{i1}, \ldots, x_{ik_i}\}$. We trivially have that $Valid_{\mathsf{store}}(FV(e), (\Gamma'\; x \colon T^{\mathsf{p}}(\overline{\tau'}))_{\eta\epsilon'}, \; s, \; h, \; oh)$. Further, from the model relations above we have that
$Valid_{\mathsf{val}}(s[..x_{ij} := h.s(x).C_i\_field_j, ..](x_{ij}), \; \tau_{j\,\eta\epsilon'}, \; h, \; oh)$. So, the store for evaluation of e_i is meaningful as well.
We apply the induction hypothesis and get $Valid_{\mathsf{val}}(v, \tau_{\eta\epsilon'}, h', oh')$ for the valuation ϵ'. The last step is to show that $Valid_{\mathsf{val}}(v, \tau_{\eta\epsilon}, h', oh')$ for the initial valuation ϵ. This is trivially true, because τ has only free size variables from $dom(\epsilon)$, where ϵ and ϵ' coincide.

OSFun: We want to apply the induction assumption to

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

Since the original typing judgement 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, $y_1 : \tau_1^\circ, \ldots, y_k : \tau_k^\circ \vdash_\Sigma e_f \colon \tau'$.
We take η' and ϵ', such that
- $\eta'(\alpha) = \tau_{\alpha\eta\epsilon}$, 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.
- $\epsilon'(n_{ij}) = p_{ij_\epsilon}$, 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

$$Valid_{\text{store}}((y_1, \ldots y_k), (y_1 : \tau^\circ_{1\ \eta'\epsilon'}, \ldots, y_k : \tau^\circ_{k,\ \eta'\epsilon'}), [y_1 := v_1, \ldots, y_n := v_n], h;\ oh)$$
$$\implies Valid_{\text{val}}(v, \tau'_{\eta'\epsilon'}, h';\ oh')$$

From $Valid_{\text{store}}(FV(e), \Gamma_{\eta\epsilon}, s, h;\ oh)$ we have validity of the values of the actual parameters: $v_j \models^{h;\ oh}_{\Gamma_{\eta\epsilon}(x_j)} w_j$ for some w_j, where $1 \leq j \leq k$. Since $\Gamma_{\eta\epsilon}(x_j) = \tau^\circ_{j\ \eta'\epsilon'}$, the left-hand side of the implication holds, and one obtains $Valid_{\text{val}}(v, \tau'_{\eta'\epsilon'}, h';\ oh')$. Since D_ϵ implies $\tau'[\ldots \alpha := \tau_\alpha \ldots][\ldots n_{ij} := p_{ij} \ldots]_{\eta\epsilon} = \tau_{\eta\epsilon}$, and by the definition of η', ϵ' we have $\tau'_{\eta'\epsilon'} = \tau'[\ldots \alpha := \tau_{\alpha\eta\epsilon} \ldots][\ldots n_{ij} := p_{ij_\epsilon} \ldots]$ one easily obtains $\tau_{\eta\epsilon} = \tau'_{\eta'\epsilon'}$ and, eventually, $Valid_{\text{val}}(v, \tau'_{\eta\epsilon}, h';\ oh')$.

Q.E.D.
The *soundness proof* is finished. :-)

4.2 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 10. *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).

```
let x = f_0(x_1, ..., x_k) in match x with | Nil ⇒ f_1(x_1, ..., x_k)
                                            | Cons(hd, tl) ⇒ f_2(x_1, ..., x_k)
```

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). This is necessary to type check, for instance, a function definition where $p \equiv 0$ and $p_1 \equiv 1$. In [18] 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

26

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:

$$f(x, \overline{y}) = \mathtt{match}\ x\ \mathtt{with}\ |\ \mathsf{Nil} \Rightarrow g(\overline{y})$$
$$|\ \mathsf{Cons}(hd,\ tl) \Rightarrow h(hd,\ tl,\ \overline{y},\ f(tl,\ \overline{y}))$$

where g and h are functions already defined, and $\overline{y}$ 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.

This condition can be enforced by a more restrictive grammar where the `let`-construct in e is replaced by `let` $x = b$ `in` $e_{nomatch}$, where

$$
\begin{aligned}
e_{nomatch} := {} & b \\
| {} & \mathtt{let}\ y = b\ \mathtt{in}\ e_{nomatch} \\
| {} & \mathtt{if}\ y\ \mathtt{then}\ e_{nomatch}\ \mathtt{else}\ e_{nomatch} \\
| {} & \mathtt{letfun}\ f(x_1, \ldots, x_n) = e\ \mathtt{in}\ e_{nomatch}
\end{aligned}
$$

For this reason we call the syntactic condition *no-let-before-match*.

We say that a set of constraints is *linear* if each constraint is of the form $\mathsf{n} = \mathsf{c} + \sum_{i=1}^{k} a_i \cdot \mathsf{n}_i$, where the components of n and n'_i are either constants or size variables and c is a tuple of constants.

Lemma 11. *If D is linear then type checking $D; \Gamma \vdash_\Sigma e : \tau$, reduces to checking a set of entailments of the form $D' \vdash \mathsf{p} = \mathsf{q}$, where D' is linear.*

Proof. By lemma 10 we know that the type checking problem terminates with a set of entailments of the form $D' \vdash \mathsf{p} = \mathsf{q}$. We prove the linearity of D' by induction on the structure of the language. Except for the `match` case, the result follows from the induction hypothesis, since $D' = D$.

Assume that e is an expression of the form `match` x `with` $\ldots$ and that x has type T^P in the context. The MATCH generates new judgements where for each constructor C_i, $D' = D$, $\mathsf{p} = \mathsf{c}_i + \sum_{i=1}^{k_i} a_{ij} \cdot \mathsf{n}_{ij}$. Thus, it only remains to prove that p is indeed a constant or a size variable (not any polynomial).

If x is free in e or it is the parameter of a function, then it must have a τ°-type on the context. From the definition of τ°-types, p is a tuple with size variables or constants. Otherwise, due to the syntactic restriction, e can not be a subexpression of a `let`-body and thus x must be bound by another `match`.

Hence, there is a variable y and a superexpression e' of e, such that

$$e' = \texttt{match } y \texttt{ with } \mid C_1(x_{11}, \dots, x_{1k_1}) \Rightarrow e_1$$

$$\vdots$$

$$\mid C_l(x_{l1}, \dots, x, \dots, x_{lk_l}) \Rightarrow e_l$$

$$\vdots$$

$$\mid C_r(x_{r1}, \dots, x_{rk_r}) \Rightarrow e_r$$

being e a subexpression of e_l. But then the match rule applied to the judgement containing e' added a fresh size variable as the size annotation of x, and it is obvious that no rule can change that. Thus, when we get to type checking e, p is a size variable.

The MATCH rule has in its premises judgements where the size of the variable being matched is expressed in D as a linear combination of new variables. Any of these variables can in turn be further subdivided, creating a *tree-decomposition* of a size variable as shown in Figure 4.2. The *no-let-before-match* restriction en-
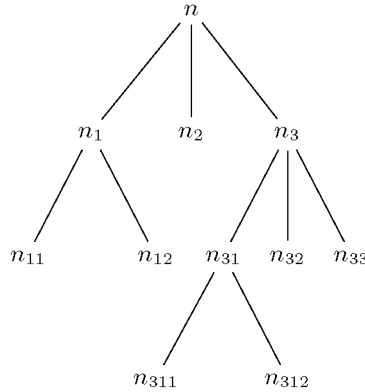

Fig. 3. Example of a tree-decomposition of a size variable. The edges mean that the father is a linear combination of the children.

sures that only size variables, and not polynomials on them, are linearly decomposed. The previous lemma tell us that after applying the typing rules, the set of constraints in the entailments left to check, contain only tree-decomposition of (some of) the free size variables of Γ. With this lemma it is easy to prove decidability of type checking the restricted language.

Theorem 2. *Type checking an expression that conforms to the* no-let-before-match *restriction is decidable.*

28

Proof. Let e be an expression that satisfies the syntactic condition, which we want to type check. At the beginning of the type checking procedure the set of constraints is empty and thus it is trivially linear. By lemma 11, type checking e reduces to checking a set of entailments of the form $D' \vdash \mathsf{p} = \mathsf{q}$, where D' is linear. Then we replace the variables in p and q using the equations in D', following a breadth-first order in the tree-decomposition of each size variable, until we get to an equality of two expression that depend only on the leaves of these trees. But since each variable is substituted by a linear combination, the two expression are polynomials on the leaves of the tree-decompositions. Finally, asserting the value of the equality reduces to comparing the coefficients of the polynomials.

4.3   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 $\mathtt{if}$ 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:

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

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

5   Size-parametric data types

In the definition of algebraic data types of our language, we forbade free size variable. In this section we extend the language to work with a restricted class of types that are parametrised by size variables.

We extend the syntax of our language to allow *size-parametric data types*. The type T being defined is parametrised by a tuple of size variables m, which can be used as sizes for the types of the variables of the constructors, provided they are not T itself. Occurrences of T are again parametrised with m.

Informally, this can be expressed as

$$spd ::= \mathtt{spdata}\ T_\mathsf{m}(\overline{\alpha})\ =\ C_1(\overline{\tau}_1^{[\mathsf{m}]}(\overline{\alpha}))\mid \ldots \mid C_r(\overline{\tau}_r^{[\mathsf{m}]}(\overline{\alpha}))$$

where $(\tau_1, \ldots, \tau_k)^{[\mathsf{m}]} = (\tau_1^{[\mathsf{m}]}, \ldots, \tau_k^{[\mathsf{m}]})$. If $\tau = T$ then $\tau^{[\mathsf{m}]} = T_\mathsf{m}$, otherwise $\tau^{[\mathsf{m}]}$ is a type that can use the variables in m as size annotations.

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.

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

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 MFullTree.

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

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

$$
\frac{
\begin{array}{c}
D,\ (e,\,n) = (1,\,0);\ \Gamma,\ t \colon \mathsf{MFullTree}_m^{e,n}(\tau) \vdash_\Sigma e_{\mathsf{Empty}} \colon \tau' \\[4pt]
D,\ (e,\,n) = (0,1) + m*(e',\,n');\ \Gamma,\ t \colon \mathsf{MFullTree}_m^{e,n}(\tau), \\
v \colon \tau,\ ts \colon \mathsf{List}^m(\mathsf{MFullTree}_m^{e',n'}(\tau)) \qquad \vdash_\Sigma e_{\mathsf{Node}} \colon \tau' \\[4pt]
e',\ n' \notin vars(D) \qquad v,\ ts \notin dom(\Gamma)
\end{array}
}{
D;\ \Gamma,\ t \colon \mathsf{MFullTree}_m^{e,n}(\tau) \vdash_\Sigma
\begin{array}{l}
\texttt{match } t \texttt{ with} \mid \mathsf{Empty} \Rightarrow e_{\mathsf{Empty}} \\
\qquad\qquad\qquad \mid \mathsf{Node}(v,\, ts) \Rightarrow e_{\mathsf{Node}}
\end{array} \colon \tau'
} \ \text{MMFTree}
$$

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

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

To get the size of the subtrees we must first get an element of the list by doing a pattern match. 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 add a parameter to the size function representing the size of the subtrees.

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

We can generalise the procedure for defining size functions in the following way. Suppose that $T_m(\alpha) = C_1(\overline{\tau}_1^{[m]}(\overline{\alpha})) \mid \ldots \mid C_r(\overline{\tau}_r^{[m]}(\overline{\alpha}))$ and let d be the maximum number of occurrences of $T_m(\alpha)$ within $\overline{\tau}_i$, for all $i = 1..r$. Then the size

---

[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.

function for T_m will take d extra parameters for the sizes of the subexpressions of type T. A constructor may ignore some of these arguments, as was the case for Empty. A canonical size function for a size-parametric data type can be defined by multiplying the size of the types other than T by the size of its elements. When the size of a subexpression of type T is needed, it is replaced it with the respective argument. Note that a size-parametric algebraic data type defines a *family* of ordinary inductive data types.

The size-aware typing rules are then obtained as we did for non-parametric algebraic data types, with the only exception that now the extra parameters of the size function are bound to the respective sizes in the typing context. Consider two functions to re-shape an m-ary full tree, one that prunes the first branch of each subtree (assuming $m > 1$) and one to add a new level at the bottom. Note that in a MFullTree we a leaf is represented by m Empty's. For the sake of simplicity, assume we are counting just the number of nodes.

$$prune(t)\colon \mathsf{MFullTree}_m^n(\alpha) \to \mathsf{MFullTree}_{m-1}^{f(n,m)}(\alpha) =$$
$$\mathtt{match}\ t\ \mathtt{with}\ |\ \mathsf{Empty} \Rightarrow \mathsf{undefined}$$
$$|\ \mathsf{Node}(v,\ ts) \Rightarrow \mathsf{Node}(v,\ tail(prune(ts)))$$

$$add\_level(v,\ t)\colon \alpha \times \mathsf{MFullTree}_m^n(\alpha) \to \mathsf{MFullTree}_m^{g(n,m)}(\alpha) =$$
$$\mathtt{match}\ t\ \mathtt{with}\ |\ \mathsf{Empty} \Rightarrow \mathsf{Node}(v,\ list\_of\_empties(m))$$
$$|\ \mathsf{Node}(v_1,\ ts) \Rightarrow \mathtt{match}\ ts\ \mathtt{with}$$
$$|\ \mathsf{Nil} \Rightarrow \mathsf{Node}(v_1,\ list\_of\_trees(v,\ m))$$
$$|\ \mathsf{Cons}(hd,\ tl) \Rightarrow \mathsf{Node}(v_1,\ \mathsf{Cons}(hd,\ add\_level(v,\ tl)))$$

where $list\_of\_empties(m)$ is a list of $m > 0$ Empty's and $list\_of\_trees(v,\ m) = \mathsf{Node}(v,\ list\_of\_empties(m))$.

Since the number of nodes in an m-ary full tree depends on its height, any function that re-shapes the tree will have size annotations involving logarithms and will not be polynomial. In our examples, the size functions are

$$f(n,\ m) = \begin{cases} log_2(n+1) - 1 & \text{if } m = 2 \\ \frac{(m-1)^{log_m(n*(m-1)+1)} - 1}{m-2} & \text{if } m > 2 \end{cases}$$

$$g(n,\ m) = \begin{cases} n+1 & \text{if } m = 1 \\ \frac{m^{log_m(n*(m-1)+1)} - 1}{m-1} & \text{if } m > 1 \end{cases}$$

To avoid the logarithms we can define the size of m-ary full trees as their size. Then

$$prune \quad \colon \mathsf{MFullTree}_m^h(\alpha) \to \mathsf{MFullTree}_{m-1}^h(\alpha)$$
$$add\_level \colon \alpha \times \mathsf{MFullTree}_m^h(\alpha) \to \mathsf{MFullTree}_m^{h+1}(\alpha)$$

6 Related Work

Amortised heap space analysis has been developed for linear bounds by Hofmann and Jost [11]. Precise knowledge of sizes is required to extend this approach to non-linear bounds [21]. Brian Campbell [6] extended this approach to infer bounds on *stack* space usage.

A type system based on amortised complexity analysis of heap-space requirements for a Java-like language with explicit deallocation is studied by Hofmann and Jost in [12]. Their approach is use *views* to assign a potential to each possible path expression, avoiding explicit manipulation of size expressions. They cope with inheritance and aliasing and circular data structures, but they do not treat type inference and the potential is an over-approximation. Another type system for an object-oriented language with a deallocation primitive is presented by Chin et al. [8], which incorporates an alias control via usage aspects.

Some interesting initial work on inferring size relations within the output of XML transformations has been done by Su and Wassermann [20]. Although this work does not yield input-output dependencies, it is able to infer size relations within the output type, for instance if two branches have the same number of elements. Herrmann and Lengauer presented a size analysis for functional programs over nested lists [10]. However, they do not solve recurrence equations in their size expressions, as this is not important for their goal of program parallelisation.

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 [16]. 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 [24], but this work leaves recurrence equations unsolved and is limited to monotonic dependencies. The work on quasi-interpretations by Bonfante et al. [5] 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 [23] 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 [14]. 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 [13], 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.

An 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 [7].

7 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 current 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. E. Barendsen and S. Smetsers. Uniqueness typing for functional languages with graph rewriting semantics. *Mathematical Structures in Computer Science*, 6:579–612, 1996.
4. M. Barr and C. Wells. *Category theory for computing science*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1990.
5. G. Bonfante, J.-Y. Marion, and J.-Y. Moyen. Quasi-interpretations, a way to control resources. *Theoretical Computer Science*, 2005.
6. B. Campbell. *Space Cost Analysis Using Sized Types*. PhD thesis, School of Informatics, University of Edinburgh, 2008.
7. S. Chatterjee, G. E. Blelloch, and A. L. Fischer. Size and access inference for data-parallel programs. In *Proceedings of the ACM SIGPLAN 1991 conference on Programming Language Design and Implementation (PLDI'91)*, pages 130–144, New York, USA, 1991. ACM Press.
8. W.-N. Chin, H. H. Nguyen, S. Qin, and R. Martin. Memory Usage Verification for OO Programs. In C. Hankin and I. Siveroni, editors, *Intl Symposium on Static Analysis (SAS 2005)*, volume 3672 of *LNCS*, pages 70–86. Springer Berlin Heidelberg, 2005.
9. P. Dybjer. Inductive and Recursive Definitions in Constructive Type Theory. TYPES Summer School, Göteborg, Aug. 2005.

10. C. A. Herrmann and C. Lengauer. A Transformational Approach which Combines Size Inference and Program Optimization. In W. Taha, editor, *Semantics, Applications, and Implementation of Program Generation (SAIG'01)*. Springer-Verlag.

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

12. M. Hofmann and S. Jost. Type-Based Amortised Heap-Space Analysis (for an Object-Oriented Language). In P. Sestoft, editor, *Proceedings of the 15th European Symposium on Programming (ESOP), Programming Languages and Systems*, volume 3924 of *LNCS*, pages 22–37. Springer, 2006.

13. 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.

14. 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.

15. F. Nielson and H. R. Nielson. Type and Effect Systems. In *Correct System Design, Recent Insight and Advances, (to Hans Langmaack on the occasion of his retirement from his professorship at the University of Kiel)*, pages 114–136, London, UK, 1999. Springer-Verlag.

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

17. B. C. Pierce. *Advanced Topics in Types and Programming Languages*. MIT Press, 2004.

18. 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.

19. O. Shkaravska, R. van Kesteren, and M. van Eekelen. Polynomial size analysis of first-order functions. Technical Report ICIS-R07004, Radboud University Nijmegen, January 2007.

20. Z. Su and G. Wassermann. Type-based Inference of Size Relationships for XML Transformations. Technical Report CSE-2004-8, UC Davis, Apr. 2004.

21. 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 8th International Symposium on Trends in Functional Programming (TFP'07), New York, USA*, pages 36–53. Intellect Publishers, UK, 2007.

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

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

24. 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: 15th International Workshop, IFL 2003, Edinburgh, UK, September 8–11, 2003. Revised Papers*, volume 3145 of *LNCS*, pages 86–101. Springer-Verlag, 2004.