

Size Analysis with Indexed Families of \max_0 -Polynomials ^{*}

O. Shkaravska, M. van Eekelen, A. Tamalet

Institute for Computing and Information Sciences
Radboud University Nijmegen

Abstract. Our previous work studied a size-aware type system for functional programs with non-monotonic polynomial size dependencies. In that approach output sizes depended only on input sizes. That is rather restrictive since in many cases the size of the output can differ for different input data of the same size.

In this paper we remove that limitation by presenting value-sensitive size dependencies via indexed families of polynomials. When a program has a family of polynomials as its output-on-input size dependency, then for each actual input value there is an index such that the corresponding polynomial in the family precisely defines the size of the output.

We introduce and study a novel size-aware type system in which size annotations are indexed families of \max_0 -polynomials; \max_0 -polynomials extend polynomials with the \max_0 -operation to prevent negative sizes.

We prove soundness of the type system and we give several decidability results for different variants of annotations.

1 Introduction

Estimating heap consumption is an active research area as it becomes more and more of an issue in many applications, including programming for small devices, e.g. smart cards, mobile phones, embedded systems and distributed computing. This work explores typing support for checking output-on-input size dependencies for function definitions (functions for short) in a functional language. We allow higher-order functions, but we consider only those where the size of the output depends just on zero-order arguments. Size dependencies are presented via families of indexed polynomials extended with the operation $\max_0(p) = \max(0, p)$. Indices are delimited by quantifier-free first-order arithmetic predicates. For any evaluation of a function there is a polynomial in the family of its type that represents the exact output-on-input size dependency for that evaluation.

^{*} This work is sponsored by the Netherlands Organisation for Scientific Research (NWO) under grant nr. 612.063.511, and is part of the AHA project.

1.1 Exploring size dependencies

We restrict our attention to a language with polymorphic lists as the only data type. For this language we develop a sound size-aware type system and then we discuss type checking and decidability issues.

To get a global idea of the kind of results we expect from our size analysis, consider a function (say `rel`) that produces all pairs of elements from two argument lists that are related to each other according to a given predicate. For instance `rel(>, [2, 3, 5], [2, 3]) = [[3, 2], [5, 2], [5, 3]]`.

The underlying type of such a function is $(\alpha \rightarrow \alpha \rightarrow \mathbf{Bool}) \times \mathbf{L}(\alpha) \times \mathbf{L}(\alpha) \rightarrow \mathbf{L}(\mathbf{L}(\alpha))$. Given two lists of length n and m , it always returns a list of pairs of length i , where i can have any value between 0 and $n * m$. We express this with the annotated type $(\alpha \rightarrow \alpha \rightarrow \mathbf{Bool}) \times \mathbf{L}_n(\alpha) \times \mathbf{L}_m(\alpha) \rightarrow \mathbf{L}_i^{\exists i. 0 \leq i \leq nm}(\mathbf{L}_2(\alpha))$.

Consider as a second example a function which deletes from a list the first occurrence of an element that satisfies a given predicate. The size of the result may vary between the size n of the list and $n - 1$, but it must be always greater than 0. Using an integer index i we express this as `delete` : $(\alpha \rightarrow \mathbf{Bool}) \times \alpha \times \mathbf{L}_n(\alpha) \rightarrow \mathbf{L}_{\max_0(n-i)}^{\exists i. 0 \leq i \leq 1}(\alpha)$ ¹.

In our setting, many size-bounded functions are well-typed. In particular, preliminary research indicates that most functions in the Haskell list library can be assigned a sized type that precisely describes the possible output sizes.

1.2 Our contribution and contents of the paper

The novelty of this paper lies in the fact that we use *families of non-monotonic* \max_0 -polynomials to express size dependencies and study decidability of size-checking in this setting. We also discuss inference of such dependencies.

The advantage of using families of \max_0 -polynomials is manifold. Firstly, we express dependencies of output sizes on input *values* via indices in polynomial annotations and in this way we avoid full-blown dependent types. Secondly, families of polynomials provide a convenient way to manage *compositionality* of non-monotonic annotations. In general, it is difficult to compute a bound on the size dependency of the composition $f_2(f_1(-))$ of two programs f_1 and f_2 if the given bound for f_2 is non-monotonic. With families of polynomials the composition rule becomes straightforward. Thirdly, for any functional program that has lower and upper bounds, both \max_0 -polynomials, one can express the output size dependency in terms of indexed \max_0 -polynomial families. Let the lower and upper bounds be $p_1(\bar{n})$ and $p_2(\bar{n})$, respectively, with \bar{n} representing the vector of the input sizes. Then the output size can be expressed as $p_1(\bar{n}) + i$ with $0 \leq i \leq \delta(\bar{n})$, where $\delta(\bar{n}) = p_2(\bar{n}) - p_1(\bar{n})$. Finally, if lower and upper \max_0 -polynomial bounds are *reachable*, we believe that it is possible to use testing to infer these bounds. By *reachable* we mean that for any size vector \bar{n} , there is

¹ In classical recursion theory $\max_0(n - m)$ is expressed as $n \dot{-} m$. Using $\dot{-}$ in is equivalent to using \max and \min operations. Indeed, $\max(m, n) = m + (n \dot{-} m)$ and $\min(m, n) = m \dot{-} (m \dot{-} n)$.

an input of size \bar{n} such that the lower bound is reached, i.e. the output has size exactly $p_1(\bar{n})$. The same holds the the upper bound. However, type inference is not covered in this work.

We give several decidability results for different variants of annotations. First, we study decidability when size annotations are rational polynomials (with integer size variables and indices). We show that type checking is undecidable even in the class of one-element \max_0 -polynomial families. (It remains undecidable even with the syntactic restriction from [14]. That restriction made type checking decidable for types annotated with single polynomials.) However, checking is decidable in the class of functions annotated with *finite families of polynomials*. Furthermore, type checking is decidable in the class of functions annotated with *(possibly infinite) families of linear \max_0 -polynomials*.

Type checking is decidable when size annotations are given in the real closed field. However, embedding size expressions and predicates into reals has drawbacks. First, some well-typed in integers functions may be rejected. This is because a counterexample to a numerical predicate may exist in real numbers, but not in naturals. Second, decision procedures in reals, such as cylindrical algebraic decomposition, are very complex w.r.t. the amount of numerical variables. We discuss how, in some situations, these drawbacks can be overcome.

This paper is organised as follows. In Section 2 we define the programming language and in Section 3 its size-aware type system. Section 3 also defines the semantics of program values w.r.t. zero-order types and the operational semantics of the language. Decidability of type checking is discussed in Section 4 and related work in Section 5. Section 6 draws conclusions and gives directions to future work.

In the appendices we give the full soundness proof and examples of type checking in detail. We also show an example of type checking a quadratic size dependency using *Matlab* to solve the constraints. They are included for the reviewers interested in technical details, but are not essential for understanding the paper. They will be included in a technical report.

2 Language

The type system is designed for a strict functional language over integers and (polymorphic) lists. Algebraic data types could added as we did in [15]. Language expressions are defined by the following grammar:

$$\begin{aligned}
 \text{Basic } b &::= c \mid \text{unop } x \mid x \text{ binop } y \mid \text{Nil} \mid \text{Cons}(z, l) \mid f(g_1, \dots, g_l, z_1, \dots, z_k) \\
 \text{Expr } e &::= b \\
 &\quad \mid \text{if } x \text{ then } e_1 \text{ else } e_2 \\
 &\quad \mid \text{let } z = b \text{ in } e_1 \\
 &\quad \mid \text{match } l \text{ with} \mid \text{Nil} \Rightarrow e_1 \\
 &\quad \quad \mid \text{Cons}(z_{\text{hd}}, l_{\text{tl}}) \Rightarrow e_2 \\
 &\quad \mid \text{letfun } f(g_1, \dots, g_l, z_1, \dots, z_k) = e_1 \text{ in } e_2
 \end{aligned}$$

where c ranges over integer and boolean constants `False` and `True`, x and y denote program variables of integer and boolean types, l ranges over lists, z denotes

a program variable of a zero-order type, \mathbf{g} ranges over higher-order program variables, \mathbf{unop} is a unary operation, either $-$ or \neg , \mathbf{binop} is one of the integer or boolean binary operations, and \mathbf{f} denotes a function name. Variables may be decorated with sub- and superscripts.

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

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 may be equivalently transformed into expressions of this form. It is useful to think of this as an intermediate language.

3 Type System

We consider a type system constituted from zero-order and higher-order types and typing rules corresponding to program constructs. Size expressions represent lengths of finite lists and arithmetic operations over these lengths. Formally, size expressions are \max_0 -polynomials in some numerical ring \mathcal{R} . The choice of \mathcal{R} influences decidability of type checking and the set of well-typed programs. We will discuss these issues later in Section 4. Indexed \max_0 -polynomials are defined by the following grammar:

$$\text{SizeExpressions } p ::= c \mid I \mid SV \mid p + p \mid p - p \mid \max_0(p - p) \mid p * p$$

where I is the set of indices and SV is the set of size variables. We denote indices via i and j and size variables via n and m . Indices and size variables may be decorated with sub- and superscripts. They may be evaluated only to non-negative values. Furthermore, we use \vec{i} , \vec{n} to denote finite collections (vectors) of indices and size variables, respectively.

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

$$\text{Types } \tau ::= \text{Int} \mid \text{Bool} \mid \alpha \mid \mathbf{L}_{p(\vec{n}, \vec{i})}^{\exists \vec{i}. Q(\vec{n}, \vec{i})}(\tau),$$

where α is a type variable and $Q(\vec{n}, \vec{i})$ is a quantifier-free first-order arithmetic formula. In the examples we omit explicit non-negativity conditions for size variables and indices.

The sets $TV(\tau)$ and $SV(\tau)$ of the type and size variables of a type τ are defined inductively in the obvious way. Note, that $SV(\mathbf{L}_0(\tau)) = \emptyset$, since all empty lists of the same underlying type represent the same data structure. For instance, $\mathbf{L}_0(\mathbf{L}_m(\text{Int}))$ represent the same structure as $\mathbf{L}_0(\mathbf{L}_0(\text{Int}))$.

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

$$\text{GroundTypes } \tau^\bullet ::= \tau \text{ such that } SV(\tau) = \emptyset \wedge TV(\tau) = \emptyset$$

We define the formal semantics of zero-order types in Section 3.1. Here we give some examples: Int , $\text{L}_5(\text{Bool})$, $\text{L}_i^{\exists i. i \leq 5}(\text{Bool})$ and $\text{L}_i^{\exists i. 1 \leq i \leq 5}(\text{L}_j^{\exists j. j \leq 3}(\text{Int}))$ are ground types, whereas α , $\text{L}_{n+5}(\text{Int})$ and $\text{L}_i^{\exists i. i \leq n}(\text{Bool})$ are not. Examples of inhabitants of the ground types are $[\text{True}, \text{True}, \text{False}]$ for $\text{L}_i^{\exists i. i \leq 5}(\text{Bool})$ and $[[1, 2, 3], [1, 2], []]$ for $\text{L}_i^{\exists i. 1 \leq i \leq 5}(\text{L}_j^{\exists j. j \leq 3}(\text{Int}))$.

Let τ° denote a zero-order type in which the size expressions are all just size variables or constants, like, for instance, $\text{L}_n(\alpha)$ and $\text{L}_n(\text{L}_i^{\exists i. i \leq m}(\alpha))$.

Function types are then defined inductively:

$$\text{FunctionTypes} \quad \tau^f ::= \tau_1^f \times \dots \times \tau_{k'}^f \times \tau_1^\circ \times \dots \times \tau_k^\circ \rightarrow \tau_0$$

where k' may be zero (i.e. the list $\tau_1^f, \dots, \tau_{k'}^f$ is empty) and $SV(\tau_0)$ contains only size variables of $\tau_1^\circ, \dots, \tau_k^\circ$. For instance, one expects $\text{insert} : (\alpha \times \alpha \rightarrow \text{Bool}) \times \alpha \times \text{L}_n(\alpha) \rightarrow \text{L}_{n+1}^{\exists i. i \leq 1}(\alpha)$, $\text{filter} : (\alpha \rightarrow \text{Bool}) \times \text{L}_n(\alpha) \rightarrow \text{L}_i^{\exists i. i \leq n}(\alpha)$.

A context Γ is a mapping from zero-order variables to zero-order types. A signature Σ is a mapping from function names to function types. The definition of $SV(-)$ is straightforwardly extended to contexts:

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

3.1 Semantics of zero-order types

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

$$\begin{aligned} \text{Address } \text{adr} &::= \ell \mid \text{NULL} \\ \text{Val } v &::= c \mid \text{adr} \quad \ell \in \text{Loc} \quad c \in \text{Int} \cup \text{Bool} \\ \text{Heap } h &: \text{Loc} \rightarrow \{\text{hd}, \text{tl}\} \rightarrow \text{Val} \end{aligned}$$

We will write $h.l.\text{hd}$ and $h.l.\text{tl}$ for the results of applications $h \ell \text{hd}$ and $h \ell \text{tl}$, which denote the values stored in the heap h at the location ℓ at its fields hd and tl , respectively. Let $h.l.[\text{hd} := v_h, \text{tl} := v_t]$ denote the heap equal to h everywhere but in ℓ , which at the hd -field of ℓ gets the value v_h and at the tl -field of ℓ gets the value v_t .

The semantics w of a program value v , with respect to a specific heap h and a ground type τ^\bullet is a set-theoretic interpretation given via the four-place relation $v \models_{\tau^\bullet}^h w$. Integer and boolean constants interpret themselves, and locations are

interpreted as non-cyclic lists:

$$\begin{array}{l}
c \quad \models_{\text{Int} \cup \text{Bool}}^h c \\
\text{NULL} \models_{\text{L}_{p(\bar{i})}^{\exists \bar{i}. Q(\bar{i})}(\tau \bullet)}^h \square \quad \text{iff } \exists \bar{i}. Q(\bar{i}) \wedge p(\bar{i}) = 0 \\
\ell \quad \models_{\text{L}_{p(\bar{i})}^{\exists \bar{i}. Q(\bar{i})}(\tau \bullet)}^h w_{\text{hd}} :: w_{\text{tl}} \quad \text{iff } \ell \in \text{dom}(h), \\
\qquad \qquad \qquad h.\ell.\text{hd} \models_{\tau \bullet}^{h|_{\text{dom}(h) \setminus \{\ell\}}} w_{\text{hd}}, \\
\qquad \qquad \qquad h.\ell.\text{tl} \models_{\text{L}_{p(\bar{i})-1}^{\exists \bar{i}. Q(\bar{i})}(\tau \bullet)}^{h|_{\text{dom}(h) \setminus \{\ell\}}} w_{\text{tl}}
\end{array}$$

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

It is easy to establish a natural connection between the size annotations in a ground list type and the length of a chain of cons-cells that “implements” its inhabitant in a heap. The length is defined by the function:

$$\begin{array}{l}
\text{length} : \text{Heap} \rightarrow \text{Address} \rightarrow \mathcal{N} \\
\text{length}_h(\text{NULL}) = 0 \qquad \text{length}_h(\ell) = 1 + \text{length}_{h|_{\text{dom}(h) \setminus \{\ell\}}}(h.\ell.\text{tl})
\end{array}$$

Note, that the function $\text{length}_h(-)$ does not take sharing into account, in the sense that the actual total size of allocated shared lists is less than the sum of their lengths. Thus, the sum of lengths of lists provides an upper bound of the actually allocated memory.

Lemma 1 (Consistency of the model relation).

The relation $\text{adr} \models_{\text{L}_{p(\bar{i})}^{\exists \bar{i}. Q(\bar{i})}(\tau \bullet)}^h w$ implies that there exists \bar{i} such that $Q(\bar{i})$ holds and $p(\bar{i}) = \text{length}_h(\text{adr})$.

The proof is done by induction on $\text{length}_h(\text{adr})$.

3.2 Operational semantics of program expressions

The operational semantics is standard. It extends the semantics from [14] with higher-order functions.

We introduce a *frame store* as a mapping from program variables to program values. This mapping is maintained when a function body is evaluated. Before evaluation of the function body starts, the store contains only the actual parameters of the function. During evaluation, the store is extended with the variables introduced by pattern matching or *let*-constructs. These variables are eventually bound to the actual parameters, thus there is no access beyond the current frame. Formally, a frame store is a finite partial map from variables to values:

$$\text{Store } s : \text{ProgramVars} \rightarrow \text{Val}$$

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

$$\begin{array}{c}
\frac{c \in \text{Int} \cup \text{Bool}}{s; h; \mathcal{C} \vdash c \rightsquigarrow c; h} \text{ OSCONS} \quad \frac{}{s; h; \mathcal{C} \vdash z \rightsquigarrow s(z); h} \text{ OSVAR} \\
\frac{}{s; h; \mathcal{C} \vdash \text{Nil} \rightsquigarrow \text{NULL}; h} \text{ OSNIL} \\
\frac{s(\text{hd}) = v_{\text{hd}} \quad s(\text{tl}) = v_{\text{tl}} \quad \ell \notin \text{dom}(h)}{s; h \vdash \text{Cons}(\text{hd}, \text{tl}) \rightsquigarrow \ell; h[\ell.\text{hd} := v_{\text{hd}}, \ell.\text{tl} := v_{\text{tl}}]} \text{ OSCONS} \\
\frac{s(x) = \text{True} \quad s; h; \mathcal{C} \vdash e_1 \rightsquigarrow v; h'}{s; h; \mathcal{C} \vdash \text{if } x \text{ then } e_1 \text{ else } e_2 \rightsquigarrow v; h'} \text{ OSIFTRUE} \\
\frac{s(x) = \text{False} \quad s; h; \mathcal{C} \vdash e_2 \rightsquigarrow v; h'}{s; h; \mathcal{C} \vdash \text{if } x \text{ then } e_1 \text{ else } e_2 \rightsquigarrow v; h'} \text{ OSIFFALSE} \\
\frac{s; h; \mathcal{C} \vdash e_1 \rightsquigarrow v_1; h_1 \quad s[z := v_1]; h_1; \mathcal{C} \vdash e_2 \rightsquigarrow v; h'}{s; h; \mathcal{C} \vdash \text{let } z = e_1 \text{ in } e_2 \rightsquigarrow v; h'} \text{ OSLET} \\
\frac{s(l) = \text{NULL} \quad s; h; \mathcal{C} \vdash e_1 \rightsquigarrow v; h'}{s; h; \mathcal{C} \vdash \text{match } l \text{ with } \begin{array}{l} | \text{Nil} \Rightarrow e_1 \\ | \text{Cons}(\text{hd}, \text{tl}) \Rightarrow e_2 \end{array} \rightsquigarrow v; h'} \text{ OSMATCH-NIL} \\
\frac{h.s(l).\text{hd} = v_{\text{hd}} \quad h.s(l).\text{tl} = v_{\text{tl}} \quad s[\text{hd} := v_{\text{hd}}, \text{tl} := v_{\text{tl}}]; h \vdash e_2 \rightsquigarrow v; h'}{s; h; \mathcal{C} \vdash \text{match } l \text{ with } \begin{array}{l} | \text{Nil} \Rightarrow e_1 \\ | \text{Cons}(\text{hd}, \text{tl}) \Rightarrow e_2 \end{array} \rightsquigarrow v; h'} \text{ OSMATCH-CONS} \\
\frac{s; h; \mathcal{C}[f := ((\mathbf{g}_1, \dots, \mathbf{g}_{k'}, \mathbf{z}_1, \dots, \mathbf{z}_k) \times e_1)] \vdash e_2 \rightsquigarrow v; h'}{s; h; \mathcal{C} \vdash \text{letfun } f(\mathbf{g}_1, \dots, \mathbf{g}_{k'}, \mathbf{z}_1, \dots, \mathbf{z}_k) = e_1 \text{ in } e_2 \rightsquigarrow v; h'} \text{ OSLETFUN} \\
\frac{s(\mathbf{z}'_1) = v_1 \dots s(\mathbf{z}'_k) = v_k \quad \mathcal{C}(f) = (\mathbf{g}_1, \dots, \mathbf{g}_{k'}, \mathbf{z}_1, \dots, \mathbf{z}_k) \times e_f}{[z_1 := v_1, \dots, z_k := v_k]; h; \mathcal{C} \vdash e_f[\mathbf{g}_1 := \mathbf{f}_1, \dots, \mathbf{g}_{k'} := \mathbf{f}_{k'}] \rightsquigarrow v; h'} \text{ OSFUNAPP} \\
\frac{}{s; h; \mathcal{C} \vdash f(\mathbf{f}_1, \dots, \mathbf{f}_{k'}, \mathbf{z}'_1, \dots, \mathbf{z}'_k) \rightsquigarrow v; h'} \text{ OSFUNAPP}
\end{array}$$

3.3 Typing rules

A typing judgement is a relation of the form $D, \Gamma \vdash_{\Sigma} e : \tau$, where D is a quantifier-free arithmetic formula over size variables. The signature Σ contains the type assumptions for the functions that are going to be checked.

Subtyping in our system relates only to type annotations and appears only as a side condition in axiom rules. Essentially it is a quantified entailment in a chosen arithmetic. An (inductive) relation $D \vdash \tau \preceq \tau'$, which is read τ is a *subtype* of τ' under D , holds if and only if

- $\tau = \tau'$ is a basic type (**Int**, **Bool** or a type variable α).
- $\tau = \mathbb{L}_{p(\bar{n}, \bar{i})}^{\exists \bar{i}. Q(\bar{n}, \bar{i})}(\tau'')$ and $\tau' = \mathbb{L}_{p'(\bar{n}, \bar{j})}^{\exists \bar{j}. Q'(\bar{n}, \bar{j})}(\tau''')$ with

$$\forall \bar{n} \forall \bar{i} \exists \bar{j}. D(\bar{n}) \wedge Q(\bar{n}, \bar{i}) \vdash Q'(\bar{n}, \bar{j}) \wedge p(\bar{n}, \bar{i}) = p'(\bar{n}, \bar{j})$$

and $\exists \bar{n} \bar{i}. D(\bar{n}) \wedge Q(\bar{n}, \bar{i}) \wedge p(\bar{n}, \bar{i}) > 0$ implies $D \vdash \tau'' \preceq \tau'''$.

The typing judgement relation is defined by the following rules.

$$\begin{array}{c} \frac{}{D, \Gamma \vdash_{\Sigma} c: \mathbf{Int}} \text{ICONST} \quad \frac{}{D, \Gamma \vdash_{\Sigma} b: \mathbf{Bool}} \text{BCONST} \\ \\ \frac{D \vdash \tau \preceq \tau'}{D, \Gamma, \mathbf{z}: \tau \vdash_{\Sigma} \mathbf{z}: \tau'} \text{VAR} \quad \frac{D \vdash \mathbb{L}_0(\tau) \preceq \tau'}{D, \Gamma \vdash_{\Sigma} \mathbf{Nil}: \tau'} \text{NIL} \\ \\ \frac{D \vdash \mathbb{L}_{p(\bar{n}, \bar{i})+1}^{\exists \bar{i}. Q(\bar{n}, \bar{i})}(\tau_2) \preceq \tau' \quad D \vdash \tau_1 \preceq \tau_2}{D, \Gamma, \mathbf{hd}: \tau_1, \mathbf{tl}: \mathbb{L}_{p(\bar{n}, \bar{i})}^{\exists \bar{i}. Q(\bar{n}, \bar{i})}(\tau_2) \vdash_{\Sigma} \mathbf{Cons}(\mathbf{hd}, \mathbf{tl}): \tau'} \text{CONS} \\ \\ \frac{\Gamma(\mathbf{x}) = \mathbf{Int} \quad D, \Gamma \vdash_{\Sigma} e_t: \tau \quad D, \Gamma \vdash_{\Sigma} e_f: \tau}{D, \Gamma \vdash_{\Sigma} \mathbf{if } \mathbf{x} \mathbf{ then } e_t \mathbf{ else } e_f: \tau} \text{IF} \\ \\ \frac{\mathbf{z} \notin \text{dom}(\Gamma) \quad D, \Gamma \vdash_{\Sigma} e_1: \tau_z \quad D, \Gamma, \mathbf{z}: \tau_z \vdash_{\Sigma} e_2: \tau}{D, \Gamma \vdash_{\Sigma} \mathbf{let } \mathbf{z} = e_1 \mathbf{ in } e_2: \tau} \text{LET} \end{array}$$

To remember that the tail **tl** of a matched list **x** is one element shorter than **x**, we combine the **MATCH**-rule with \exists -introduction:

$$\frac{Q(\bar{n}, \bar{n}^i), p(\bar{n}, \bar{n}^i) = 0, D, \Gamma, \mathbf{l}: \mathbb{L}_{p(\bar{n}, \bar{n}^i)}(\tau) \vdash_{\Sigma} e_{\mathbf{Nil}}: \tau' \quad \mathbf{hd}, \mathbf{tl} \notin \text{dom}(\Gamma) \quad Q(\bar{n}, \bar{n}^i), p(\bar{n}, \bar{n}^i) \geq 1, D, \Gamma, \mathbf{hd}: \tau, \mathbf{l}: \mathbb{L}_{p(\bar{n}, \bar{n}^i)}(\tau), \mathbf{tl}: \mathbb{L}_{p(\bar{n}, \bar{n}^i)-1}(\tau) \vdash_{\Sigma} e_{\mathbf{Cons}}: \tau'}{D, \Gamma, \mathbf{l}: \mathbb{L}_{p(\bar{n}, \bar{i})}^{\exists \bar{i}. Q(\bar{n}, \bar{i})}(\tau) \vdash_{\Sigma} \mathbf{match } \mathbf{l} \mathbf{ with } \left| \begin{array}{l} \mathbf{Nil} \Rightarrow e_{\mathbf{Nil}} \\ \mathbf{Cons}(\mathbf{hd}, \mathbf{tl}) \Rightarrow e_{\mathbf{Cons}} \end{array} \right. : \tau'} \text{MATCH, } \exists - I$$

$$\frac{Q(\bar{n}, \bar{n}^i), D, \Gamma, \mathbf{l}: \mathbb{L}_{p(\bar{n}, \bar{n}^i, \bar{j})}^{\exists \bar{j}. P(\bar{n}, \bar{j})}(\tau) \vdash_{\Sigma} e: \tau}{D, \Gamma, \mathbf{l}: \mathbb{L}_{p(\bar{n}, \bar{i}, \bar{j})}^{\exists \bar{i} \bar{j}. P(\bar{n}, \bar{j}) \wedge Q(\bar{n}, \bar{i})}(\tau) \vdash_{\Sigma} e: \tau} \exists - I$$

Note that on a given context Γ , $\exists - I$ can be applied up to $|\text{dom}(\Gamma)|$ times. It is not applicable to “internal” indices in a nested list type, since one would need as many skolemised constants as internal lists, which is in general unknown. The skolemised indices \bar{n}^i are treated as size variables. To see why one needs skolemisation, consider the expression $\text{let } l_1 = \mathbf{Cons}(\mathbf{z}, \mathbf{l}) \text{ in let } l_2 = \mathbf{Cons}(\mathbf{z}, \mathbf{l}) \text{ in } e$ in the context $\mathbf{l}: \mathbb{L}_{n+j}^{\exists j. j \leq 1}(\mathbf{Int})$. When type checking e the lists l_1 and l_2 will have type $\mathbb{L}_{n+j+1}^{\exists j. j \leq 1}(\mathbf{Int})$ in the context, but this does not keep information that they

are of the same size, the length of l plus 1 (because j is existentially quantified). To prove equality of the sizes one first has to skolemise j .

$$\frac{\begin{array}{c} \Sigma(f) = \tau_1^f \times \dots \times \tau_{k'}^f \times \tau_1^\circ \times \dots \times \tau_k^\circ \rightarrow \tau_0 \\ \Sigma(\mathbf{g}_1) = \tau_1^f, \dots, \Sigma(\mathbf{g}_{k'}) = \tau_{k'}^f \\ \mathbf{z}_1 : \tau_1^\circ, \dots, \mathbf{z}_k : \tau_k^\circ \vdash_{\Sigma} e_1 : \tau_0 \quad \Gamma \vdash_{\Sigma} e_2 : \tau' \end{array}}{\Gamma \vdash_{\Sigma} \text{letfun } f(\mathbf{g}_1, \dots, \mathbf{g}_{k'}, \mathbf{z}_1, \dots, \mathbf{z}_k) = e_1 \text{ in } e_2 : \tau'} \text{LETFUN}$$

$$\frac{\begin{array}{c} \Sigma(f) = \tau_1^f \times \dots \times \tau_{k'}^f \times \tau_1^\circ \times \dots \times \tau_k^\circ \rightarrow \tau_0 \\ \text{the type of } \mathbf{g}_i \text{ is an instance of the type } \tau_i^f; \\ D \vdash \sigma(\tau_0) \preceq \tau \quad D \vdash C(\tau_1, \dots, \tau_k) \end{array}}{D, \Gamma, \mathbf{z}_1 : \tau_1, \dots, \mathbf{z}_k : \tau_k \vdash_{\Sigma} f(\mathbf{g}_1, \dots, \mathbf{g}_{k'}, \mathbf{z}_1, \dots, \mathbf{z}_k) : \tau} \text{FUNAPP}$$

The function application rule computes a substitution σ from the formal size variables to the actual size expressions, and a set C of equations collecting restrictions on the actual input types. These restrictions are of the form $p(\bar{n}) = c$ or $\tau \equiv \tau'$ abbreviating $(\tau \preceq \tau' \wedge \tau' \preceq \tau)$. The equation $p(\bar{n}) = c$ belongs to C if c is a constant from the (formal) input type. The equation $\tau \equiv \tau'$ belongs to C if τ and τ' are substituted into the same type, like, for instance, in the call to the function `scalarprod` : $L_m(\text{Int}) \times L_m(\text{Int}) \rightarrow \text{Int}$ with actual parameters $l : \tau$ and $l' : \tau'$. To see how the substitution σ is applied consider a formal size parameter m with $\sigma(m) = p'(\bar{n}, \bar{j})$, where $Q'(\bar{n}, \bar{j})$ holds. Then

$$\sigma\left(L(\dots L_{\frac{\exists \bar{i}. Q(m, \bar{i})}{p(m, \bar{i})}}(\dots L(\alpha) \dots) \dots)\right) = L(\dots L_{\frac{\exists \bar{j}. Q(p'(\bar{n}, \bar{j}), \bar{i}) \wedge Q'(\bar{n}, \bar{j})}{p(p'(\bar{n}, \bar{j}), \bar{i})}}(\dots L(\alpha) \dots) \dots)$$

As an example of type checking, consider the function `rel` which computes the list of pairs defined by the relation `g` on two sets, given by the input lists:

$$\begin{aligned} \text{rel}(\mathbf{g}, l_1, l_2) = & \text{match } l_1 \text{ with } | \text{Nil} \Rightarrow \text{Nil} \\ & | \text{Cons}(\text{hd}, \text{tl}) \Rightarrow \text{append}(\text{rel_pairs}(\mathbf{g}, \text{hd}, l_2), \text{rel}(\mathbf{g}, \text{tl}, l_2)) \end{aligned}$$

where

$$\begin{aligned} \text{rel_pairs}(\mathbf{g}, x, l) = & \text{match } l \text{ with } | \text{Nil} \Rightarrow \text{Nil} \\ & | \text{Cons}(\text{hd}, \text{tl}) \Rightarrow \text{if } \mathbf{g}(x, \text{hd}) \text{ then} \\ & \quad \text{Cons}(\text{Cons}(x, \text{Cons}(\text{hd}, \text{Nil})), \\ & \quad \quad \text{rel_pairs}(x, \text{tl})) \\ & \quad \text{else } \text{rel_pairs}(x, \text{tl}) \end{aligned}$$

Assuming `rel_pairs` : $(\alpha \times \alpha \rightarrow \alpha) \times \alpha \times L_n(\alpha) \rightarrow L_i^{\exists \bar{i}. i \leq n}(L_2(\alpha))$, type checking the `cons` branch of `rel` reduces to the following predicate:

$\forall n m j i' \exists i. n \geq 1, 0 \leq j \leq m, 0 \leq i' \leq m(n-1) \implies 0 \leq i \leq mn \wedge i = j + i'$. It is easy to see that $i = j + i'$ satisfies the inequality $0 \leq i \leq mn$.

3.4 Semantics of typing judgements (soundness)

The set-theoretic semantics of typing judgements is formalised later in this section as the soundness theorem. The soundness theorem is defined by means of

the following two predicates. One indicates if a program value is *valid* with respect to a certain heap and a ground type. The other does the same for sets of values and types, taken from a frame store and a ground context Γ^\bullet :

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

Let a valuation ϵ map size variables to concrete sizes (numbers from the ring \mathcal{R}) and an instantiation η map type variables to ground types:

$$\begin{aligned} \text{Valuation } \epsilon &: \text{SizeVariables} \rightarrow \mathcal{R} \\ \text{Instantiation } \eta &: \text{TypeVariables} \rightarrow \tau^\bullet \end{aligned}$$

Valuations and instantiations distribute over annotations in the following way:

$$\begin{aligned} \epsilon(p_1 + p_2) &= \epsilon(p_1) + \epsilon(p_2) & \epsilon(p_1 - p_2) &= \epsilon(p_1) - \epsilon(p_2) \\ \epsilon(\text{max}_0(p)) &= \text{max}(0, \epsilon(p)) & \epsilon(p_1 * p_2) &= \epsilon(p_1) * \epsilon(p_2) \\ \epsilon(\exists \bar{i}. Q(\bar{n}, \bar{i})) &= \exists \bar{i}. Q(\bar{i}, \epsilon(\bar{n})) & \eta(\mathbb{L}_p^Q(\tau)) &= \mathbb{L}_p^Q(\eta(\tau)) \end{aligned}$$

Now, stating the soundness theorem is straightforward:

Theorem 1 (Soundness). *For any store s , heaps h and h' , closure \mathcal{C} , expression e , value v , context Γ , quantifier-free formula D , signature Σ , type τ , size valuation ϵ , and type instantiation η such that*

- $\text{dom}(s) = \text{dom}(\Gamma)$
- $s; h; \mathcal{C} \vdash e \rightsquigarrow v; h'$,
- $D, \Gamma \vdash_\Sigma e : \tau$ is a node in the derivation tree for some function body, and all the functions called in e are declared via `letfun`,
- $D(\epsilon(\bar{n}))$ holds, where \bar{n} is the set of size variables from $\text{dom}(\Gamma)$,

the following implication holds:

$$\forall_{\eta, \epsilon} [\text{Valid}_{\text{store}}(\text{dom}(s), \eta(\epsilon(\Gamma)), s, h) \implies \text{Valid}_{\text{val}}(v, \eta(\epsilon(\tau)), h')]$$

Proof. The proof is done by induction on the size of the derivation tree for the operational-semantic judgement. For the sake of convenience we will denote $D(\epsilon(\bar{n}))$ via D_ϵ , $\eta(\epsilon(\tau))$ via $\tau_{\eta\epsilon}$ and $\eta(\epsilon(\Gamma))$ via $\Gamma_{\eta\epsilon}$. Given $s; h; \mathcal{C} \vdash e \rightsquigarrow v; h'$ fix some Γ , Σ , and τ , such that $D, \Gamma \vdash_\Sigma e : \tau$. One can easily check by induction that $TV(\tau) \subseteq TV(\Gamma)$, $SV(\tau) \subseteq SV(\Gamma)$ and $SV(D) \subseteq SV(\Gamma)$, since $D, \Gamma \vdash_\Sigma e : \tau$ is a node in the derivation tree for some function body. Fix a valuation $\epsilon : TV(\Gamma) \rightarrow \mathcal{R}$, and a type instantiation $\eta : TV(\Gamma) \rightarrow \tau^\bullet$ such that the assumptions of the lemma hold. We must show that $\text{Valid}_{\text{val}}(v, \tau_{\eta\epsilon}, h')$ holds.

The axioms are sound due to the lemma stating that subtyping preserves the model relation. The *exist* – *I* rule and the `MATCH`-rule are sound because of Lemma 1 (consistency).

To give an idea of the proof, we consider `let`-construct case in detail. In this case e is `let z = e1 in e2` for some z , e_1 , and e_2 and we have $s; h; \mathcal{C} \vdash e_1 \rightsquigarrow v_1; h_1$ and $s[z := v_1]; h_1; \mathcal{C} \vdash e_2 \rightsquigarrow v; h'$ for some v_1 and h_1 . We know

that $D, \Gamma \vdash_{\Sigma} e_1 : \tau', z \notin \Gamma$ and $D, \Gamma, z : \tau' \vdash_{\Sigma} e_2 : \tau$ for some τ' . Applying the induction hypothesis to the first branch gives $\text{Valid}_{\text{store}}(\text{dom}(s), D, \Gamma_{\eta\epsilon}, s, h) \implies \text{Valid}_{\text{val}}(v_1, \tau'_{\eta\epsilon}, h_1)$.

Now apply the induction hypothesis to the second branch to get

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

Fix some $z' \in \text{dom}(s[z := v_1])$. If $z' = z$, then $\text{Valid}_{\text{val}}(v_1, \tau'_{\eta\epsilon}, h_1)$ implies $\text{Valid}_{\text{val}}(s[z := v_1](z), \tau'_{\eta\epsilon}, h_1)$. If $z' \neq z$, then $s[z := v_1](z') = s(z')$. Sharing of data structures in the heap is benign (no destructive pattern matching or assignments), hence $h|_{\mathcal{R}(h, s(z'))} = h_1|_{\mathcal{R}(h, s(z'))}$. Applying a lemma stating that an equal footprint preserves the model relation, we have that $s(z') \models_{\Gamma_{\eta\epsilon}(z')}^h w'_z$ implies $s(z') \models_{\Gamma_{\eta\epsilon}(z')}^{h_1} w'_z$ implies $s[z := v_1](z') \models_{\Gamma_{\eta\epsilon}(z')}^{h_1} z'_y$ and thus $\text{Valid}_{\text{val}}(s[z := v_1](z'), \Gamma_{\eta\epsilon}(z'), h_1)$. Hence, $\text{Valid}_{\text{store}}(\text{dom}(s[z := v_1]), \Gamma_{\eta\epsilon} \cup \{z : \tau'_{\eta\epsilon}\}, s[z := v_1], h_1)$. To apply the induction assumption we use $\text{dom}(\Gamma, z : \tau') = \text{dom}(\Gamma) \cup \{z\} = \text{dom}(s) \cup \{z\} = \text{dom}(s[z := v_1])$.

4 Decidability of Type checking

4.1 Undecidable in general

Type checking is undecidable if the underlying arithmetic is integer, even when types are annotated with a single polynomial [14]. In that work we provided a syntactic condition called *no-let-before-match* that makes type checking exact size dependencies decidable. This condition requires that no match is done over variables bound by a let construct.

However, adding max_0 makes type checking undecidable, even for programs satisfying *no-let-before-match*. We show that for any polynomial $p(n_1, \dots, n_k)$ there is a *total* function $f_p : \mathbb{L}_{n_1}(\text{Int}) \times \dots \times \mathbb{L}_{n_k}(\text{Int}) \rightarrow \mathbb{L}_{\text{max}_0(1-p^2(n_1, \dots, n_k))}(\text{Int})$. Checking if this function has the type $\mathbb{L}_{n_1}(\text{Int}) \times \dots \times \mathbb{L}_{n_k}(\alpha) \rightarrow \mathbb{L}_0(\text{Int})$ amounts to answering the question whether p has natural roots or not. If p has no natural roots, i.e. $p^2(n_1, \dots, n_k) \geq 1$ for all naturals n_1, \dots, n_k , then the type is accepted. Otherwise, there are inputs for which the length of the output is 1 and the type is rejected. Thus, type checking reduces to solving 10th Hilbert problem, which is known to be undecidable [12].

The function f_p can be defined by $f_p(l_1, \dots, l_k) = \text{diff}([1], \text{p2}(li_1, \dots, l_k))$, where $\text{diff} : \mathbb{L}_n(\alpha) \times \mathbb{L}_m(\alpha) \rightarrow \mathbb{L}_{\text{max}_0(n-m)}(\alpha)$ and $\text{p2} : \mathbb{L}_{n_1}(\alpha) \times \dots \times \mathbb{L}_{n_k}(\alpha) \rightarrow \mathbb{L}_{p^2(n_1, \dots, n_k)}(\alpha)$. The function body of $\text{diff}(l_1, l_2)$ is

$$\begin{aligned} \text{match } l_1 \text{ with } & | \text{Nil} \Rightarrow \text{Nil} \\ & | \text{Cons}(\text{hd}, \text{tl}) \Rightarrow \text{match } l_2 \text{ with } | \text{Nil} \Rightarrow l_1 \\ & | \text{Cons}(\text{hd}', \text{tl}') \Rightarrow \text{diff}(\text{tl}, \text{tl}') \end{aligned}$$

The total function p2 is obtained by the compositions of the functions $\text{append} : \mathbb{L}_n(\alpha) \times \mathbb{L}_m(\alpha) \rightarrow \mathbb{L}_{n+m}(\alpha)$, $\text{copyfirst} : \mathbb{L}_n(\alpha) \times \mathbb{L}_m(\alpha) \rightarrow \mathbb{L}_{nm}(\alpha)$, which copies the first argument m times, and the function $\text{sqdiff} : \mathbb{L}_n(\alpha) \times \mathbb{L}_m(\alpha) \rightarrow \mathbb{L}_{(n-m)^2}(\alpha)$:

$$\begin{array}{l} \text{match } l_1 \text{ with } | \text{Nil} \Rightarrow \text{copyfirst}(l_2, l_2) \\ \quad | \text{Cons}(\text{hd}, \text{tl}) \Rightarrow \text{match } l_2 \text{ with } | \text{Nil} \Rightarrow \text{copyfirst}(l_1, l_1) \\ \quad \quad | \text{Cons}(\text{hd}', \text{tl}') \Rightarrow \text{sqdiff}(\text{tl}, \text{tl}') \end{array}$$

Note that these functions satisfy *no-let-before-match*. Any polynomial p may be expressed in the form $p = p_1 - p_2$, where p_1 and p_2 are polynomials with non-negative coefficients. Furthermore, for any polynomial q with non-negative coefficients, there is a composition of `append` and `copyfirst` that has q as a size dependency. Using these facts we can define `p2` for a particular p .

4.2 Decidable classes

There are two obvious classes of function types where type checking is decidable:

- (a) output types annotated with *linear predicates and size expressions*. Since linearity is preserved by compositions, type checking amounts to checking predicates in decidable linear (Presburger) arithmetic,
- (b) output types annotated with *finite predicates* and with *polynomials*, i.e. without `max0`, and function definitions subject to *no-let-before-match*.

In the second case type checking is similar to checking with one-polynomial annotations. It reduces to checking entailments of the form $\forall \vec{j} \in N \exists \vec{i} \in M. p(\vec{n}, \vec{j}) = p'(\vec{n}, \vec{i})$, where the sets N and M are finite and $\vec{n} = (n_1, \dots, n_k)$ is such that $n_i \geq c_i$ for some integer constant $c_i \geq 0$. One must instantiate $\vec{i} \in M$ for each $\vec{j} \in N$. Given $\vec{j}_0 \in N$, compute a polynomial $p(\vec{n}, \vec{j}_0)$ that becomes a polynomial w.r.t. \vec{n} . By the finite search in M , check if there is $\vec{i}_0 \in M$ such that $p(\vec{n}, \vec{j}_0) = p'(\vec{n}, \vec{i}_0)$ as polynomials in \vec{n} (i.e. if the corresponding coefficients are equal). If not, reject the type. Otherwise, continue with another $\vec{j}_0 \in N$, until the whole N is checked.

4.3 Decidability in real arithmetic

If the underlying numerical ring \mathcal{R} is the real closed field, then type checking is decidable due to the well-known Tarski's result. However, such embedding has two drawbacks.

First, some functions that are well-typed in integers may be rejected. Consider, for instance, the composition `copyfirst(l, tail(l))`, where `tail : Ln(α) → Lmax0(n-1)(α)` returns the tail of a list l if it is not empty, and `Nil` otherwise. The type of the composition is $L_n(\alpha) \rightarrow L_{n \max_0(n-1)}(\alpha)$. It is possible to construct an integer type checker that accepts the type $L_n(\alpha) \rightarrow L_{n(n-1)}(\alpha)$, however, a real type checker must reject it because $n \max_0(n-1) = n(n-1)$ fails for $0 < n < 1$.

In the example above, the predicate arising during the real-based type checking procedure fails on a *bounded area*. In cases like this, it is possible to check if there are natural points within the area. In the example above there are no naturals in $(0, 1)$, so a real type checker equipped with a “browser” of naturals in bounded areas will accept the type annotated with $n(n-1)$. However, if the area is unbounded the problem is unsolvable in general. This approach is used in the

Mathematica software package, which applies *cylindrical algebraic decomposition* to obtain a cylindrical-cell-presentation of the solution.

Second, decision procedures in reals, such as cylindrical algebraic decomposition, are very complex w.r.t. the amount of variables. We believe that in practice, it is possible to use optimisation libraries for quadratic constraints to check quadratic bounds. Experimental results using the optimisation toolbox of *MatLab* are encouraging. It is worth to note that there is an algorithm that decides whether a (multivariate) quadratic equation has a solution in integers. But the bound on such a solution, if it exists, is very high. For instance, for a 4-variable equation it is proportional to H^{84} , where H is the maximum of the absolute values of the polynomial's coefficients [8]. To the best of our knowledge, there is no a similar result for *systems* of quadratic equations.

5 Related Work

This research continues our work on polynomial size dependencies [14, 17, 15]. In our current setting such dependencies correspond to one-element families of polynomials (without the \max_0 operation).

Amortisation [13] is a promising technique to obtain accurate bounds of resource consumption and gain. Combining amortisation with type theory allows to infer linear heap bounds for functional programs with explicit memory deallocation [11]. Brian Campbell [7] extended this approach to infer bounds on *stack* space usage. The *AHA* project aims to adapt amortisation techniques for *non-linear* bounds within functional programs and transfer the results to the object-oriented programming. Contrary to linear amortised bounds, to obtain non-linear heap estimates one needs to know the sizes of the structures that take part in a computation [16].

Our approach is close to size analysis with polynomial quasi-interpretations [6, 3]. There a program is interpreted as a monotonic polynomial extended with the \max operation. For instance, $\text{Cons}(\text{hd}, \text{tl})$ is interpreted as $T + 1$, where T is a numerical variable abstracting tl . Using such interpretations one obtains upper monotonic-polynomial bounds for size dependencies. The main difference with our approach is that we are interested in non-monotonic lower and upper bounds. To our knowledge, non-monotonic quasi-interpretations have not been studied for size analysis, but only for proving termination [10]. In this work one considers some unspecified algorithmically decidable classes of non-negative and negative polynomials and introduces abstract variables for the rest. However, for size analysis we study decidability issues in detail.

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 [18] 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*.

Several papers have studied programming languages with *implicit computational complexity* properties [9, 5]. This line of research is motivated both by

the perspective of automated complexity analysis and by fundamental goals, in particular to give natural characterisations of complexity classes, like PTIME or PSPACE.

Resource analysis may be performed within a *Proof Carrying Code* framework. In [4] the authors introduce resource policies for mobile code to be run on smart devices. Policies are integrated into a proof-carrying code architecture. Two forms of policies are used: *guaranteed policies* which come with proofs and *target policies* which describe limits of the device.

In [2] the authors describe resource consumption for Java bytecode by means of Cost Equation Systems (CESs), which are similar to, but more general than recurrence equations. CESs express the cost of a program in terms of the size of its input data. In a further step, a closed form (i.e., non-recursive) solution or upper bound can sometimes be found by using existing Computer Algebra Systems, such as *Maple* and *Mathematica*. This work is continued by the authors in [1], where mechanisms for solving and upper bounding CESs are studied. They consider monotonic cost expressions only.

6 Conclusions and Future Work

This paper presents a size-aware type system capable of describing size-relations between the inputs and the output of a function. It allows to express a family of output sizes via indexed non-monotonic polynomials augmented with the \max_0 operation. This feature greatly increases the applicability of our size analysis, which was limited to exact sizes. The extra expressibility comes at a cost: we have crossed the border of decidability. However this does not make the analysis infeasible in practice: we have presented several decidability results and we have performed practical experiments. Together they will form the basis for design choices of a practical implementation.

Our next step will be to study type inference for the type system here presented, inspired by the test-based ideas from [17] and assess the quality of the bounds inferred. Then, we plan to extend our prototype implementation, available via www.aha.cs.ru.nl, to cope with different output sizes and use it in some case studies. Furthermore, as part of the AHA project, we will transfer our size analysis results to the world of imperative programs.

References

1. E. Albert, P. Arenas, S. Genaim, and G. Puebla. Automatic Inference of Upper Bounds for Recurrence Relations in Cost Analysis. In *Static Analysis, 15th International Symposium*.
2. E. Albert, P. Arenas, S. Genaim, G. Puebla, and D. Zanardini. Cost Analysis of Java Bytecode. In *16th European Symposium on Programming, ESOP'07*, volume 4421 of *LNCS*, pages 157–172. Springer, 2007.
3. R. M. Amadio. Synthesis of max-plus quasi-interpretations. *Fundamenta Informaticae*, 65(1-2):29–60, 2004.

4. D. Aspinall and K. MacKenzie. Mobile Resource Guarantees and Policies. In G. Barthe, B. Grégoire, M. Huisman, and J.-L. Lanet, editors, *CASSIS 2005*, volume 3956 of *LNCS*, pages 16–36. Springer, 2006.
5. V. Atassi, P. Baillot, and K. Terui. Verification of Ptime Reducibility for System F Terms: Type Inference in Dual Light Affine Logic. *Logical Methods in Computer Science*, 3(4), 2007.
6. G. Bonfante, J.-Y. Marion, and J.-Y. Moyen. Quasi-interpretations, a way to control resources. *Theoretical Computer Science*, 2005.
7. B. Campbell. *Space Cost Analysis Using Sized Types*. PhD thesis, School of Informatics, University of Edinburgh, 2008.
8. R. Dietmann. Small solutions of quadratic diophantine equations. In *Proceedings of the London Mathematical Society*, volume 86, pages 545–582, 2003.
9. M. Gaboardi, J.-Y. Marion, and S. Ronchi Della Rocca. A logical account of PSPACE. In *35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages POPL 2008, San Francisco, January 10-12, 2008, Proceedings*, pages 121–131, 2008.
10. N. Hirokawa and A. Middeldorp. Polynomial interpretations with negative coefficients. In *Artificial Intelligence and Symbolic Computation*, volume 3249 of *LNCS*, 2004.
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. J. P. Jones and Y. V. Matijasevič. Proof of recursive unsolvability of Hilbert’s tenth problem. *American Mathematical Monthly*, 98(10):689–709, 1991.
13. C. Okasaki. *Purely Functional Data Structures*. Cambridge University Press, 1998.
14. 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.
15. A. Tamalet, O. Shkaravska, and M. van Eekelen. Size Analysis of Algebraic Data Types. In M. Morazán, editor, *Selected Papers of the 9th International Symposium on Trends in Functional Programming (TFP’08)*. Intellect Publishers. 2008, to appear.
16. M. van Eekelen, O. Shkaravska, R. van Kesteren, B. Jacobs, E. Poll, and S. Smetters. 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.
17. 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.
18. P. B. Vasconcelos. *Space Cost Analysis Using Sized Types*. PhD thesis, School of Computer Science, University of St. Andrews, August 2008.

A Proofs of heap-aware-semantics lemmata and soundness in detail

In this appendix we present proofs of heap-aware semantics lemmata and the soundness theorem in detail. For the sake of readability, we omit existential quantifiers in annotations when it does not clutter proofs.

Informally, soundness of the type system ensures that “well-typed programs will not go wrong”. This is achieved by demanding that, when evaluation of the function starts with a valid w.r.t its input types store, the result will be a valid value of the output type.

Since there is no destructive pattern matchings and assignments on our language, we deal with *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.

To reason about heap we introduce the function *footprint*

$$\mathcal{R} : \text{Heap} \times \text{Val} \longrightarrow \mathcal{P}(\text{Loc})$$

that computes the set of locations accessible in a given heap from a given value:

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

where $f|_X$ denotes the restriction of a (partial) map f to a set X . We extend \mathcal{R} to stores by $\mathcal{R}(h, s) = \bigcup_{x \in \text{dom}(s)} \mathcal{R}(h, s(x))$.

A.1 Heap and model-relation lemmata

One proves by induction on the size of (the domain of) the heap following lemmata.

Lemma 2 (A program value’s footprint is in the heap).

$\mathcal{R}(h, v) \subseteq \text{dom}(h)$.

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

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

$\text{dom}(h) \neq \emptyset$:

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

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

Lemma 3 (Extending a heap does not change the footprints of program values). *If $\ell \notin \mathbf{dom}(h)$ and $h' = h[\ell.\mathbf{hd} := v_{\mathbf{hd}}, \ell.\mathbf{tl} := v_{\mathbf{tl}}]$, for some $v_{\mathbf{hd}}, v_{\mathbf{tl}}$ then for any $v \neq \ell$ one has $\mathcal{R}(h, v) = \mathcal{R}(h', v)$.*

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

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

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

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

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

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

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

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

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

and the induction assumption one has

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

So,

$$\begin{aligned} \mathcal{R}(h', \ell') &= \\ &= \{\ell'\} \cup \mathcal{R}(h'|_{\mathbf{dom}(h') \setminus \{\ell'\}}, h'.\ell'.\mathbf{hd}) \cup \mathcal{R}(h'|_{\mathbf{dom}(h') \setminus \{\ell'\}}, h'.\ell'.\mathbf{tl}) = \\ &= \{\ell'\} \cup \mathcal{R}(h|_{\mathbf{dom}(h) \setminus \{\ell'\}}, h.\ell'.\mathbf{hd}) \cup \mathcal{R}(h|_{\mathbf{dom}(h) \setminus \{\ell'\}}, h.\ell'.\mathbf{tl}) = \\ &= \mathcal{R}(h, \ell'). \end{aligned}$$

Lemma 4 (Extending heaps preserves the model relation).

For all heaps h and h' , if $h'|_{\mathbf{dom}(h)} = h$ then $v \models_{\tau}^h w$ implies $v \models_{\tau}^{h'} w$.

Proof.

The lemma is proved by induction on the structure of τ .

$\tau = \mathbf{Int} \cup \mathbf{Bool}$: In this case, v is a constant c and $w = c$, hence $v \models_{\tau}^{h'} w$ by the definition.

$\tau = \mathbf{L}_{p(\bar{i})}^{Q(\bar{i})}(\tau')$: We proceed by induction on the length of the chain of constructors.

$v = \text{NULL}$: In this case, $\exists \bar{i}. Q(\bar{i}) \wedge p(\bar{i}) = 0$ and $w = []$, hence $v \models_{\tau^\bullet}^{h'} w$ by the definition.

$v = \ell$: By the definition $\ell \models_{\perp_{p(\bar{i})-1}^{Q(\bar{i})}(\tau^\bullet)}^h w_{\text{hd}} :: w_{\text{tl}}$ for some w_{hd} and w_{tl} such that

$$\begin{aligned} \ell &\in \text{dom}(h), \\ h.\ell.\text{hd} &\models_{\tau^\bullet}^{h|_{\text{dom}(h)\setminus\{\ell\}}} w_{\text{hd}}, \\ h.\ell.\text{tl} &\models_{\perp_{p(\bar{i})-1}^{Q(\bar{i})}(\tau^\bullet)}^{h|_{\text{dom}(h)\setminus\{\ell\}}} w_{\text{tl}} \end{aligned}$$

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

$$\begin{aligned} &h'|_{\text{dom}(h')\setminus\{\ell\}}|_{\text{dom}(h|_{\text{dom}(h)\setminus\{\ell\}})} \\ &= h'|_{\text{dom}(h')\setminus\{\ell\}}|_{\text{dom}(h)\setminus\{\ell\}} \\ &= h'|_{\text{dom}(h)\setminus\{\ell\}} = h|_{\text{dom}(h)\setminus\{\ell\}} \end{aligned}$$

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

$$\begin{aligned} \ell &\in \text{dom}(h'), \\ h'.\ell.\text{hd} &\models_{\tau^\bullet}^{h'|_{\text{dom}(h')\setminus\{\ell\}}} w_{\text{hd}}, \\ h'.\ell.\text{tl} &\models_{\perp_{p(\bar{i})-1}^{Q(\bar{i})}(\tau^\bullet)}^{h'|_{\text{dom}(h')\setminus\{\ell\}}} w_{\text{tl}} \end{aligned}$$

Then, $\ell \models_{\perp_{p(\bar{i})-1}^{Q(\bar{i})}(\tau^\bullet)}^{h'} w_{\text{hd}} :: w_{\text{tl}}$ by the definition. \square

Lemma 5 (Values depend only on their footprints).

For v, h, w , and τ^\bullet , the relation $v \models_{\tau^\bullet}^h w$ implies $v \models_{\tau^\bullet}^{h|\mathcal{R}(h, v)} w$.

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

$\tau^\bullet = \text{Int} \cup \text{Bool}$: By the definition, v is a constant c and thus $w = c$. Then $v \models_{\tau^\bullet}^{h|\mathcal{R}(h, v)} w$.

$\tau^\bullet = \perp_{p(\bar{i})}^{Q(\bar{i})}(\tau^\bullet)$: We proceed by induction on the length of the chain of constructors.

$v = \text{NULL}$: By the definition $\exists \bar{i}. Q(\bar{i}) \wedge p(\bar{i}) = 0$ and $w = []$. Then $v \models_{\tau^\bullet}^{h|\mathcal{R}(h, v)} w$.

$v = \ell$: By the definition $w = w_{\text{hd}} :: w_{\text{tl}}$ for some w_{hd} and w_{tl} , s.t.

$$\begin{aligned} \ell &\in \text{dom}(h), \\ h.\ell.\text{hd} &\models_{\tau^\bullet}^{h|_{\text{dom}(h)\setminus\{\ell\}}} w_{\text{hd}}, \\ h.\ell.\text{tl} &\models_{\perp_{p(\bar{i})-1}^{Q(\bar{i})}(\tau^\bullet)}^{h|_{\text{dom}(h)\setminus\{\ell\}}} w_{\text{tl}} \end{aligned}$$

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

$$\begin{aligned}
& \ell \in \text{dom}(h), \\
& h.\ell.\text{hd} \models_{\tau^\bullet}'^{h|_{\text{dom}(h)\setminus\{\ell\}}|\mathcal{R}(h|_{\text{dom}(h)\setminus\{\ell\}}, h.\ell.\text{hd})} w_{\text{hd}}, \\
& h.\ell.\text{tl} \models_{\perp_{p(\bar{i})-1}^{Q(\bar{i})}(\tau^\bullet)'}^{h|_{\text{dom}(h)\setminus\{\ell\}}|\mathcal{R}(h|_{\text{dom}(h)\setminus\{\ell\}}, h.\ell.\text{tl})} w_{\text{tl}}
\end{aligned}$$

Due to $\mathcal{R}(h|_{\text{dom}(h)\setminus\{\ell\}}, h.\ell.\text{hd}) \subseteq \text{dom}(h) \setminus \{\ell\}$ (lemma 2) we have

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

Similarly $h|_{\text{dom}(h)\setminus\{\ell\}}|\mathcal{R}(h|_{\text{dom}(h)\setminus\{\ell\}}, h.\ell.\text{tl}) = h|_{\mathcal{R}(h|_{\text{dom}(h)\setminus\{\ell\}}, h.\ell.\text{tl})\setminus\{\ell\}}$.
Due to $\ell \in \mathcal{R}(h, \ell)$, and lemma 4 – with $\mathcal{R}(h|_{\text{dom}(h)\setminus\{\ell\}}, h.\ell.\text{hd}) \setminus \{\ell\} \subseteq \mathcal{R}(h, h.\ell.\text{hd}) \setminus \{\ell\}$, we have

$$\begin{aligned}
& \ell \in \text{dom}(h_{\mathcal{R}(h, \ell)}), \\
& h|_{\mathcal{R}(h, \ell)}.\ell.\text{hd} \models_{\tau^\bullet}'^{h|_{\mathcal{R}(h, h.\ell.\text{hd})\setminus\{\ell\}}} w_{\text{hd}}, \\
& h|_{\mathcal{R}(h, \ell)}.\ell.\text{tl} \models_{\perp_{p(\bar{i})-1}^{Q(\bar{i})}(\tau^\bullet)'}^{h|_{\mathcal{R}(h, h.\ell.\text{hd})\setminus\{\ell\}}} w_{\text{tl}}
\end{aligned}$$

Thus, $\ell \models_{\perp_{p(\bar{i})-1}^{Q(\bar{i})}(\tau^\bullet)'}^{h|_{\mathcal{R}(h, \ell)}} w_{\text{hd}} :: w_{\text{tl}}$. □

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

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

Proof. Assume $v \models_{\tau^\bullet}^h w$. Lemma 5 states that this implies $v \models_{\tau^\bullet}^{h|_{\mathcal{R}(h, v)}} w$.
Assuming $h|_{\mathcal{R}(h, v)} = h'|_{\mathcal{R}(h, v)}$ we get $v \models_{\tau^\bullet}^{h'|_{\mathcal{R}(h, v)}} w$. Since $h'|_{\text{dom}(h'|_{\mathcal{R}(h, v)})} = h'|_{\mathcal{R}(h, v)}$ we may apply lemma 4, which gives $v \models_{\tau^\bullet}^{h'} w$. □

A.2 Semantics of zero-order ground types

Lemma 1. (Given a ground list type, the length of the cons-cell chain of its inhabitant is equal to the size polynomial evaluated at some index i .) The relation $\text{adr} \models_{\perp_{p(\bar{i})}^{Q(\bar{i})}(\tau^\bullet)}^h w$ implies that

- there exists i , such that $Q(\bar{i})$ holds and $p(\bar{i}) = \text{length}_h(\text{adr})$, and
- any element of the list pointed by adr is of type τ^\bullet , that is for each $0 \leq l \leq \text{length}_h(\text{adr}) - 1$, the relation $h.\ell.\text{tl}^l.\text{hd} \models_{\tau^\bullet}^h w^l$ is defined for some set-theoretic value w^l .

Proof. By induction on $\text{length}_h(\text{adr})$.

$\text{length}_h(\text{adr}) = 0$: this means that $\text{adr} = \text{NULL}$ and there exists \bar{i} , s.t. $Q(\bar{i})$ holds and $p(\bar{i}) = 0$.

$length_h(\text{adr}) > 0$: this means that $\text{adr} = \ell$ and, therefore, there exist w_{hd} and w_{tl} , s.t. $w = w_{\text{hd}} : w_{\text{tl}}$ and $h.\ell.\text{hd} \models_{\tau^\bullet}^{h|_{\text{dom}(h)\setminus\{\ell\}}} w_{\text{hd}}$ and $h.\ell.\text{tl} \models_{\mathbb{L}_{p(\bar{i})}^{Q(\bar{i})}(\tau^\bullet)}^{h|_{\text{dom}(h)\setminus\{\ell\}}} w_{\text{tl}}$. From the definition of $length$ it follows that

$$length_h(h.\ell.\text{tl}) = length_{h|_{\text{dom}(h)\setminus\{\ell\}}}(\ell) - 1$$

so we may apply induction assumption to obtain \bar{i} , such that $Q(\bar{i})$ holds and $p(\bar{i}) - 1 = length_{h|_{\text{dom}(h)\setminus\{\ell\}}}(h.\ell.\text{tl}) = length_h(\ell) - 1$, and therefore $p(\bar{i}) = length_h(\ell)$. Moreover, for all $0 \leq l \leq length_h(\ell) - 2$ there exists w^l , such that $h.\ell.\text{tl}.\text{tl}^l \models_{\tau^\bullet}^h w^l$.

A.3 Properties of subtyping

Before to show that subtyping preserves the model relation, we prove the following simple auxiliary

Lemma 7 (Arithmetic operations preserving subtyping). *Let G be a 1-variable polynomial, such that $x \geq G(x)$. Then $D(\bar{n}) \vdash \mathbb{L}_{p(\bar{n}, \bar{i})}^{Q(\bar{n}, \bar{i})}(\tau) \preceq \mathbb{L}_{p'(\bar{n}, \bar{j})}^{Q'(\bar{n}, \bar{j})}(\tau')$ implies $D(\bar{n}) \vdash \mathbb{L}_{G(p(\bar{n}, \bar{i}))}^{Q(\bar{n}, \bar{i})}(\tau) \preceq \mathbb{L}_{G(p'(\bar{n}, \bar{j}))}^{Q'(\bar{n}, \bar{j})}(\tau')$.*

Proof. Fix some \bar{n}, \bar{i} , s.t. $D(\bar{n}) \wedge Q(\bar{n}, \bar{i})$ holds. From the condition of the lemma it follows that there exists \bar{j} , s.t. $Q'(\bar{n}, \bar{j}) \wedge p(\bar{n}, \bar{i}) = p'(\bar{n}, \bar{j})$. From this follows that $Q'(\bar{n}, \bar{j}) \wedge G(p(\bar{n}, \bar{i})) = G(p'(\bar{n}, \bar{j}))$. Now, let $\exists \bar{n} \bar{i}. D(\bar{n}) \wedge Q(\bar{n}, \bar{i}) \wedge G(p(\bar{n}, \bar{i})) > 0$. Show that $D \vdash \tau \preceq \tau'$. Indeed, $p(\bar{n}, \bar{i}) \geq G(p(\bar{n}, \bar{i})) > 0$ and from the condition of the lemma it follows that $D \vdash \tau \preceq \tau'$.

Lemma 8 (Subtyping preserves the model relation). *Let $\vdash \tau^\bullet(\bar{i}) \preceq \tau^\bullet(\bar{j})$. Then $v \models_{\tau^\bullet}^h w$ implies $v \models_{\tau^\bullet}^h w$.*

Proof. Induction over the relation \models .

$v = c$: Then τ^\bullet and τ^\bullet are both either Int or Bool , and $v \models_{(\text{Int} \cup \text{Bool})}^h w$ is exactly $v \models_{\text{Int} \cup \text{Bool}}^h w$.

$v = \text{NULL}$: Then $\tau^\bullet = \mathbb{L}_{p(\bar{i}_1)}^{Q(\bar{i}_1)}(\tau^{\bullet\prime\prime})$, with $\bar{i}_1 \subseteq \bar{i}$, and $\tau^\bullet = \mathbb{L}_{p'(\bar{j}_1)}^{Q'(\bar{j}_1)}(\tau^{\bullet\prime\prime\prime})$, with $\bar{j}_1 \subseteq \bar{j}$. From the definition of the model relation one has $\exists \bar{i}^0. Q(\bar{i}^0) \wedge p(\bar{i}^0) = 0$. From the subtyping relation it follows that for this (\bar{i}_0) there exists \bar{j}^0 s.t. $Q'(\bar{j}_1)$ and $p'(\bar{j}^0) = p(\bar{i}^0) = 0$. Therefore $\text{NULL} \models_{\mathbb{L}_{p'(\bar{j}_1)}^{Q'(\bar{j}_1)}(\tau^{\bullet\prime\prime\prime})}^h []$.

$v = \ell$: Then $\tau^\bullet = \mathbb{L}_{p(\bar{i}_1)}^{Q(\bar{i}_1)}(\tau^{\bullet\prime\prime})$ and $\tau^\bullet = \mathbb{L}_{p'(\bar{j}_1)}^{Q'(\bar{j}_1)}(\tau^{\bullet\prime\prime\prime})$, with $\bar{i}_1 \subseteq \bar{i}$ and $\bar{j}_1 \subseteq \bar{j}$. From the definition of the model relation one has

$$\begin{aligned} h.\ell.\text{hd} &\models_{\tau^{\bullet\prime\prime}}^{h|_{\text{dom}(h)\setminus\{\ell\}}} w_{\text{hd}}, \\ h.\ell.\text{tl} &\models_{\mathbb{L}_{p(\bar{i}_1)-1}^{Q(\bar{i}_1)}(\tau^{\bullet\prime\prime})}^{h|_{\text{dom}(h)\setminus\{\ell\}}} w_{\text{tl}} \end{aligned}$$

for some $w_{\text{hd}}, w_{\text{tl}}$. By the induction assumption (we need the lemma 7 to apply it)

$$\begin{aligned} h.\ell.\text{hd} &\models_{\tau \bullet \bullet \bullet}^{h|_{\text{dom}(h) \setminus \{\ell\}}} w_{\text{hd}}, \\ h.\ell.\text{tl} &\models_{\text{L}_{p'(\bar{j}_1)-1}^{Q'(\bar{j}_1)}(\tau \bullet \bullet \bullet)}^{h|_{\text{dom}(h) \setminus \{\ell\}}} w_{\text{tl}} \end{aligned}$$

Therefore, by the definition of the model relation one obtains $\ell \models_{\text{L}_{p'(\bar{j}_1)}^{Q'(\bar{j}_1)}(\tau \bullet \bullet \bullet)}^h w_{\text{hd}} : w_{\text{tl}}$.

A.4 Soundness statement

Theorem 2 (Soundness). *For any $s, h, \mathcal{C}, e, v, h'$, a context Γ , a quantifier-free formula D , a signature Σ , and a type τ , any size valuation ϵ , a type instantiation η such that*

- $\text{dom}(s) = \text{dom}(\Gamma)$
- $s; h; \mathcal{C} \vdash e \rightsquigarrow v; h'$,
- $D, \Gamma \vdash_{\Sigma} e : \tau$ is a node in the derivation tree for some function body, and all the functions called in e are declared via **letfun**,
- $D(\epsilon(\bar{n}))$ holds, where \bar{n} is the set of the size variables from $\text{dom}(\Gamma)$,

the following implication holds:

$$\forall_{\eta, \epsilon} [\text{Valid}_{\text{store}}(\text{dom}(s), \eta(\epsilon(\Gamma)), s, h) \implies \text{Valid}_{\text{val}}(v, \eta(\epsilon(\tau)), h')]$$

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

We prove the statement by induction on the height of the derivation tree for the operational semantics. Given $s; h; \mathcal{C} \vdash e \rightsquigarrow v; h'$ we fix some Γ, Σ , and τ , such that $D, \Gamma \vdash_{\Sigma} e : \tau$. One can easily check by induction that $TV(\tau) \subseteq TV(\Gamma)$, $SV(\tau) \subseteq SV(\Gamma)$ and $SV(D) \subseteq SV(\Gamma)$, since $D, \Gamma \vdash_{\Sigma} e : \tau$ is a node in the derivation tree for some function body. We fix a valuation $\epsilon \in TV(\Gamma) \rightarrow \mathcal{R}$, a type instantiation $\epsilon \in TV(\Gamma) \rightarrow \tau \bullet \bullet \bullet$, such that the assumptions of the lemma hold.

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

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

OSNull: In this case $v = \text{NULL}$, and according to the definition of the model relation we have $\text{NULL} \models_{\text{L}_0(\tau'_{\eta\epsilon})}^h []$ for τ' from the typing rule. Now we use the side condition $D \vdash \text{L}_0(\tau') \preceq \tau$ and the fact that D_{ϵ} holds to apply the lemma 8 and obtain $\text{NULL} \models_{\tau}^h []$.

OSVar: In this case $v = s(x)$. From $\text{Valid}_{\text{store}}(\text{dom}(s), (\Gamma' \cup (x : \tau'))_{\eta\epsilon}, h, s)$ for the corresponding τ' it follows that $s(x) \models_{\tau'}^h w$ for some w . We use the side condition $D \vdash \tau' \preceq \tau$ and D_{ϵ} to apply the lemma 8 we obtain $v \models_{\tau}^h w$.

OSCons: In this case $e = \text{Cons}(\text{hd}, \text{tl})$ and Γ is “ $\Gamma', \text{hd} : \tau_1, \text{tl} : \mathbb{L}_{p(\bar{n}, \bar{i})}^{Q(\bar{n}, \bar{i})}(\tau_2)$ ” for some $\Gamma', \text{hd}, \text{tl}, Q, p$ and τ' . Since $\text{Valid}_{\text{store}}(\text{dom}(s), \Gamma_{\eta\epsilon}, s, h)$ there exist w_{hd} and w_{tl} such that $s(\text{hd}) \models_{\tau_1\eta\epsilon}^h w_{\text{hd}}$ and $s(\text{tl}) \models_{\mathbb{L}_{p(\bar{n}, \bar{i})}^{Q(\bar{n}, \bar{i})}(\tau_2\eta\epsilon)}^h w_{\text{tl}}$.

From the operational semantics judgement we have that $v = \ell$ for some location $\ell \notin \text{dom}(h)$, and $h' = h[\ell.\text{hd} := s(\text{hd}), \ell.\text{tl} := s(\text{tl})]$. Therefore, $h'.\ell.\text{hd} \models_{\tau_1\eta\epsilon}^h w_{\text{hd}}$ and $h'.\ell.\text{tl} \models_{\mathbb{L}_{p(\bar{n}, \bar{i})}^{Q(\bar{n}, \bar{i})}(\tau_2\eta\epsilon)}^h w_{\text{tl}}$ also hold. It is easy to see

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

Thus,

$$\begin{aligned} h'.\ell.\text{hd} &\models_{\tau_1\eta\epsilon}^{h'|_{\text{dom}(h') \setminus \{\ell\}}} w_{\text{hd}} \\ h'.\ell.\text{tl} &\models_{\mathbb{L}_{p(\bar{n}, \bar{i})}^{Q(\bar{n}, \bar{i})}(\tau_2\eta\epsilon)}^{h'|_{\text{dom}(h') \setminus \{\ell\}}} w_{\text{tl}} \end{aligned}$$

Applying $D \vdash \tau_1 \preceq \tau_2$ and D_ϵ in lemma 8 gives $\ell \models_{\mathbb{L}_{p(\bar{n}, \bar{i})+1}^{Q(\bar{n}, \bar{i})}(\tau_2\eta\epsilon)}^{h'} w_{\text{hd}} :: w_{\text{tl}}$. We use the side condition

$$D \vdash \mathbb{L}_{p+1}^Q(\tau') \preceq \tau$$

to apply the lemma 8 we obtain $v \models_{\tau}^h w_{\text{hd}} :: w_{\text{tl}}$.

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

$$\text{Valid}_{\text{store}}(\text{dom}(s), \Gamma_{\eta\epsilon}, s, x) \implies \text{Valid}_{\text{val}}(v, \tau_{\eta\epsilon}, h')$$

Due to the condition of the lemma, we have $\text{Valid}_{\text{val}}(v, \tau_{\eta\epsilon}, h')$.

OSIfFalse: Exactly as the true-case, but with e_2 instead of e_1 .

OSLetFun: The result follows from the induction hypothesis for

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

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

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

Now apply the induction hypothesis to the second branch to get

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

Fix some $\mathbf{z}' \in \text{dom}(s[\mathbf{z} := v_1])$. If $\mathbf{z}' = \mathbf{z}$, then $\text{Valid}_{\text{val}}(v_1, \tau'_{\eta\epsilon}, h_1)$ implies $\text{Valid}_{\text{val}}(s[\mathbf{z} := v_1](\mathbf{z}), \tau'_{\eta\epsilon}, h_1)$. If $\mathbf{z}' \neq \mathbf{z}$, then $s[\mathbf{z} := v_1](\mathbf{z}') = s(\mathbf{z}')$. Sharing of data structures in the heap is benign (no destructive pattern matching and assignments), hence $h|_{\mathcal{R}(h, s(\mathbf{z}'))} = h_1|_{\mathcal{R}(h, s(\mathbf{z}'))}$. Applying lemma

6 we have that $s(z') \Vdash_{\Gamma_{\eta^\epsilon(z')}}^h w'_z$ implies $s(z') \Vdash_{\Gamma_{\eta^\epsilon(z')}}^{h_1} w'_z$ implies $s[z := v_1](z') \Vdash_{\Gamma_{\eta^\epsilon(z')}}^{h_1} z'_y$ and thus $\text{Valid}_{\text{val}}(s[z := v_1](z'), \Gamma_{\eta^\epsilon(z')}, h_1)$. Hence, $\text{Valid}_{\text{store}}(\text{dom}(s[z := v_1]), \Gamma_{\eta^\epsilon} \cup \{z: \tau'_{\eta^\epsilon}\}, s[z := v_1], h_1)$. To apply the induction assumption we use $\text{dom}(\Gamma, z: \tau') = \text{dom}(\Gamma) \cup \{z\} = \text{dom}(s) \cup \{z\} = \text{dom}(s[z := v_1])$.

OSMatch-Nil: In this case $e = \text{match } x \text{ with } \begin{cases} \text{Nil} \Rightarrow e_1 \\ \text{Cons}(\text{hd}, \text{tl}) \Rightarrow e_2 \end{cases}$ for some x ,

$\text{hd}, \text{tl}, e_1$, and e_2 . The typing context has the form $\Gamma = \Gamma' \cup \{x: \mathbb{L}_p^Q(\tau')\}$ for some Γ', τ', Q, p . The operational-semantics derivation gives $s(x) = \text{NULL}$, hence validity for $s(x)$ gives that there exists \bar{i}_0 , such that $Q_\epsilon(\bar{i}_0)$ holds and $p_\epsilon(\bar{i}_0) = 0$. We introduce the new size variables \bar{n}^i and extend ϵ to ϵ' with $\epsilon'(\bar{n}^i) := \bar{i}_0$. Therefore, $s(x) \Vdash_{\mathbb{L}_{p_{\epsilon'}}(\tau')}^h []$. This means that

$\text{Valid}_{\text{store}}(\text{dom}(s), \Gamma_{\eta^{\epsilon'}}, x: \mathbb{L}_{p_{\epsilon'}}(\tau'), s, h)$. We can apply the the induction hypothesis with $D_{\epsilon'} \wedge Q_{\epsilon'} \wedge p_{\epsilon'} = 0$; $\Gamma, x: \mathbb{L}_{p_{\epsilon'}}(\tau') \vdash_\Sigma e: \tau$ to obtain $\text{Valid}_{\text{val}}(v, \tau_{\eta^\epsilon}, h')$, taking into account that τ does not have size variables \bar{n}^i and, hence, $\tau_{\eta^\epsilon} = \tau_{\eta^{\epsilon'}}$.

OSMatch-Cons: In this case $e = \text{match } x \text{ with } \begin{cases} \text{Nil} \Rightarrow e_1 \\ \text{Cons}(\text{hd}, \text{tl}) \Rightarrow e_2 \end{cases}$ for some x ,

$\text{hd}, \text{tl}, e_1, e_2$. The typing context has the form $\Gamma = \Gamma' \cup \{x: \mathbb{L}_p^Q(\tau')\}$ for some Γ', τ', Q, p . From the operational semantics we know that $h.s(x).\text{hd} = v_{\text{hd}}$ and $h.s(x).\text{v}_{\text{tl}}$ for some v_{hd} and v_{tl} , that is $s(x) \neq \text{NULL}$. Due to validity of $s(x)$ and the lemma 1 we have that there exists \bar{i}_0 , such that $Q_\epsilon(\bar{i}_0)$ holds and $p_\epsilon(\bar{i}_0) = \text{length}_h(s(x)) \geq 1$. We introduce the new size variables \bar{n}^i and extend ϵ to ϵ' with $\epsilon'(\bar{n}^i) := \bar{i}_0$. Therefore, $s(x) \Vdash_{\mathbb{L}_{p_{\epsilon'}}(\tau'_{\eta^{\epsilon'}})}^h w_{\text{hd}} : w_{\text{tl}}$ for the corresponding w_{hd} and w_{tl} .

From the validity $s(x) \Vdash_{\mathbb{L}_{p_{\epsilon'}}(\tau'_{\eta^{\epsilon'}})}^h w_{\text{hd}} : w_{\text{tl}}$ the validities of v_{hd} and v_{tl} follows: $v_{\text{hd}} \Vdash_{\tau_{\eta^{\epsilon'}}}^h w_{\text{hd}}, v_{\text{tl}} \Vdash_{(\mathbb{L}_{p_{\epsilon'}-1}(\tau'))_{\eta^\epsilon}}^h w_{\text{tl}}$.

From $\text{Valid}_{\text{store}}(\text{dom}(s), \Gamma_{\eta^\epsilon}, s, h)$ and the results above we obtain

$$\text{Valid}_{\text{store}}(\text{dom}(s'), \Gamma_{\eta^{\epsilon'}}, x: \mathbb{L}_{p_{\epsilon'}}(\tau'_{\eta^{\epsilon'}}), \text{hd}: \tau'_{\eta^{\epsilon'}}, \text{tl}: \mathbb{L}_{p_{\epsilon'}-1}(\tau')_{\eta^{\epsilon'}}, s', h)$$

where $s' = s[\text{hd} := v_{\text{hd}}][\text{tl} := v_{\text{tl}}]$. It is easy to see that $\text{dom}(s') = \text{dom}(s) \cup \{\text{hd}, \text{tl}\} = \text{dom}(\Gamma) \cup \{\text{hd}, \text{tl}\} = \text{dom}(\Gamma, x: \mathbb{L}_{p_{\epsilon'}}(\tau'_{\eta^{\epsilon'}}), \text{hd}: \tau'_{\eta^{\epsilon'}}, \text{tl}: \mathbb{L}_{p_{\epsilon'}-1}(\tau')_{\eta^{\epsilon'}})$

From the typing derivation of e we obtain that

$$\Gamma', x: \mathbb{L}_{p_{\epsilon'}}(\tau'_{\eta^{\epsilon'}}), \text{hd}: \tau'_{\eta^{\epsilon'}}, \text{tl}: \mathbb{L}_{p_{\epsilon'}-1}(\tau')_{\eta^{\epsilon'}} \vdash_\Sigma e_2: \tau_{\eta^{\epsilon'}}$$

With $D \wedge Q_{\epsilon'} \wedge p_{\epsilon'} \geq 1$, the induction hypothesis yields

$$\begin{aligned} & \text{Valid}_{\text{store}}(\text{dom}(s'), \left\{ \begin{array}{l} \Gamma_{\eta^\epsilon} \cup \\ \cup \{x: \mathbb{L}_{p_{\epsilon'}}(\tau'_{\eta^{\epsilon'}})\} \cup \\ \cup \{\text{hd}: \tau'_{\eta^{\epsilon'}}\} \cup \\ \cup \{\text{tl}: \mathbb{L}_{p_{\epsilon'}-1}(\tau')_{\eta^\epsilon}\} \end{array} \right\}, s \left[\begin{array}{l} \text{hd} := v_{\text{hd}}, \\ \text{tl} := v_{\text{tl}} \end{array} \right], h) \implies \\ & \implies \text{Valid}_{\text{val}}(v, \tau_{\eta^{\epsilon'}}, h'). \end{aligned}$$

Now from the induction hypothesis we have

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

OSFun: We want to apply the induction assumption to

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

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`, $y_1 : \tau^\circ, \dots, y_k : \tau_k^\circ \vdash_{\Sigma} e_f : \tau_0$. Trivially, the domains of the frame store $[y_1 := v_1, \dots, y_k := v_k]$ the context $y_1 : \tau^\circ, \dots, y_k : \tau_k^\circ$ coincide.

We take η' and ϵ' , such that

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

`True` (“no conditions”) holds trivially on ϵ' .

From the induction assumption we have

$$\begin{aligned} & \text{Valid}_{\text{store}}((y_1, \dots, y_k), (y_1 : \tau_{1, \eta'\epsilon'}, \dots, y_k : \tau_{k, \eta'\epsilon'}), [y_1 := v_1, \dots, y_n := v_n], h) \\ & \implies \text{Valid}_{\text{val}}(v, \tau'_{\eta'\epsilon'}, h') \end{aligned}$$

From $\text{Valid}_{\text{store}}(\text{dom}(s), \Gamma_{\eta\epsilon}, s, h)$ we have validity of the values of the actual parameters: $v_i \models_{\Gamma_{\eta\epsilon}(x_i)}^h w_i$ for some w_i , where $1 \leq i \leq k$. Since $\Gamma_{\eta\epsilon}(x_i) = \tau_{i, \eta'\epsilon'}$, the left-hand side of the implication holds, and one obtains $\text{Valid}_{\text{val}}(v, \tau_{0, \eta'\epsilon'}, h')$.

We have the subtyping $D \vdash *(\tau_0) \preceq \tau$. It is easy to see that

$$\begin{aligned} \eta\epsilon(*(\tau_0)) &= \\ \eta\epsilon(\tau_0[\dots \alpha := \tau_\alpha \dots][\dots n_{ij} := p_{ij} \dots]) &= \\ \tau_0[\dots \alpha := \eta(\tau_\alpha) \dots][\dots n_{ij} := \epsilon(p_{ij}) \dots] &= \\ \tau_{0, \eta'\epsilon'} & \end{aligned}$$

Therefore, trivially $\text{Valid}_{\text{val}}(v, \eta(\epsilon(*\tau_0)), h')$. We apply the lemma 8 to obtain $v \models_{\tau_{\eta\epsilon}}^h w$.

\exists -I: Follows from the lemmata 1.

□

B Examples of type-checking

```

delete(g, z, l) =
match l with | Nil ⇒ Nil
             | Cons(hd, tl) ⇒ if g(z, hd) then tl else let z' = delete(g, z, tl) in Cons(hd, z')

```

$$\text{delete} : (\alpha \times \alpha \rightarrow \text{Bool}) \times \alpha \times \mathbb{L}_n(\alpha) \rightarrow \mathbb{L}_{n-i}^{\exists i. i \leq 1}(\alpha)$$

Step	The subgoal after the step
main goal	$z : \alpha, l : \mathbb{L}_n(\alpha) \vdash_{\Sigma} e_{\text{delete}} : \mathbb{L}_{n-i}^{\exists i. i \leq 1}(\alpha)$
MATCHNIL NIL	$n = 0; z : \alpha, l : \mathbb{L}_n(\alpha) \vdash_{\Sigma} \text{Nil} : \mathbb{L}_{n-i}^{\exists i. i \leq 1}(\alpha)$ $n = 0 \vdash i^? \leq 1 \wedge 0 = n \dot{-} i^?$ $i^? := 0$
MATCHCONS	$n \geq 1; z : \alpha, l : \mathbb{L}_n(\alpha) \vdash_{\Sigma} \text{if } - \text{ then } - \text{ else } - : \mathbb{L}_{n-i}^{\exists i. i \leq 1}(\alpha)$
IFTRUE VAR	$n \geq 1; l : \mathbb{L}_n(\alpha) \vdash_{\Sigma} \text{tl} : \mathbb{L}_{n-i}^{\exists i. i \leq 1}(\alpha)$ $n \geq 1 \vdash i^? \leq 1 \wedge m = n \dot{-} i^?$ $i^? := 0$
IFFALSE LET-bind	$n \geq 1; z : \alpha, \text{hd} : \alpha, \text{tl} : \mathbb{L}_{n-1}(\alpha) \vdash_{\Sigma} \text{let } z' = - \text{ in } - : \mathbb{L}_{n-i}^{\exists i. i \leq 1}(\alpha)$ $n \geq 1; z : \alpha, \text{tl} : \mathbb{L}_{n-1}(\alpha) \vdash_{\Sigma} \text{delete}(x, \text{tl}) : \tau^?$
FUN	$\tau^? := \mathbb{L}_{(n-1) \dot{-} j}^{\exists j. j \leq 1}(\alpha)$
LET-body	$n \geq 1; \text{hd} : \alpha, \text{tl} : \mathbb{L}_{n-1}(\alpha), z' : \mathbb{L}_{(n-1) \dot{-} j}^{\exists j. j \leq 1}(\alpha) \vdash_{\Sigma}$ $\text{Cons}(\text{hd}, z) : \mathbb{L}_{n-i}^{\exists i. i \leq 1}(\alpha)$
CONS	$n \geq 1, j \leq 1 \vdash i^? \leq 1 \wedge (n-1) \dot{-} j + 1 = n \dot{-} i^?$
CASES for j	$n \geq 1, j = 0 \vdash i^? \leq 1 \wedge n = n \dot{-} i^?$
CASES	$n \geq 1, j = 1 \vdash i^? \leq 1 \wedge (n-1) \dot{-} 1 + 1 = n \dot{-} i^?$ $n \geq 1, j = 0 \vdash i^? \leq 1 \wedge i^? = 0$ $n = 1, j = 1 \vdash i^? \leq 1 \wedge 0 = n \dot{-} i^?$
CASES	$n \geq 2, j = 1 \vdash i^? \leq 1 \wedge n - 1 - 1 + 1 = n \dot{-} i^?$ $n \geq 1, j = 0 \vdash i^? \leq 1 \wedge i^? = 0$ $n = 1, j = 1 \vdash i^? \leq 1 \wedge i^? = 1$ $n \geq 2, j = 1 \vdash i^? \leq 1 \wedge i^? = 1$

$\text{rdelete}(g, l_1, l_2) =$
 $\text{match } l_1 \text{ with } \begin{cases} \text{Nil} \Rightarrow l_2 \\ \text{Cons}(\text{hd}, \text{tl}) \Rightarrow \text{let } l' = \text{rdelete}(g, \text{tl}, l_2) \text{ in } \text{delete}(g, \text{hd}, l') \end{cases}$

$\text{delete} : (\alpha \times \alpha \rightarrow \text{Bool}) \times \mathbb{L}_n(\alpha) \times \mathbb{L}_m(\alpha) \rightarrow \mathbb{L}_{m-i}^{\exists i. i \leq n}(\alpha)$

Step	The subgoal after the step
main goal	$l_1 : \mathbb{L}_n(\alpha), l_2 : \mathbb{L}_m(\alpha) \vdash_{\Sigma} e_{\text{rdelete}} : \mathbb{L}_{m-i}^{\exists i. i \leq n}(\alpha)$
MATCHNIL	$n = 0, l_1 : \mathbb{L}_n(\alpha), l_2 : \mathbb{L}_m(\alpha) \vdash_{\Sigma} l_2 : \mathbb{L}_{m-i}^{\exists i. i \leq n}(\alpha)$
VAR	$n = 0 \vdash i^? \leq n \wedge m = m - i^?$
SIMP	$i^? := 0$
MATCHCONS	$n \geq 1; l_1 : \mathbb{L}_n(\alpha), \text{hd} : \alpha, \text{tl} : \mathbb{L}_{n-1}(\alpha), l_2 : \mathbb{L}_m(\alpha) \vdash_{\Sigma}$ $\text{let } l' = - \text{ in } - : \mathbb{L}_{m-i}^{\exists i. i \leq n}(\alpha)$
LET-bind	$n \geq 1; \text{tl} : \mathbb{L}_{n-1}(\alpha), l_2 : \mathbb{L}_m(\alpha) \vdash_{\Sigma} \text{rdelete}(\text{tl}, l_2) : \tau^?$
FUN	$\tau^? := \mathbb{L}_{m-j}^{\exists j. j \leq n-1}(\alpha)$
LET-body	$n \geq 1; l_2 : \mathbb{L}_m(\alpha), \text{hd} : \alpha, \text{tl} : \mathbb{L}_{n-1}(\alpha), l' : \mathbb{L}_{m-j}^{\exists j. j \leq n-1}(\alpha) \vdash_{\Sigma}$ $\text{delete}(g, \text{hd}, l') : \mathbb{L}_{m-i}^{\exists i. i \leq n}(\alpha)$
FUN	$n \geq 1, j \leq n-1, j' \leq 1 \vdash i^? \leq n \wedge (m-j) - j' = m - i^?$
Cases for j, j' , find $i^?$	Let P be $n \geq 1, j \leq n-1, j' \leq 1$
CASE 1	$P, j \leq m, j' \leq m-j \vdash i^? \leq n \wedge m-j-j' = m - i^?$
SIMP	$i^? := j + j' \quad (j + j' \leq n)$
CASE 2	$P, j \leq m, j' > m-j \vdash i^? \leq n \wedge 0 = m - i^?$
SIMP	$i^? := m \quad (i^? \leq n)$
BECAUSE	$\{(n, m, j, j'). n \geq 1, j \leq n-1, j \leq m, 1 \geq j' > m-j\} \subseteq$ $\{(n, m, j, j'). 1 \geq j' > m - (n-1)\} \subseteq$ $\{(n, m, j, j'). m > n\}$
CASE 3	$P, j > m \vdash i^? \leq n \wedge 0 = m - i^?$
SIMP	$i^? := m \quad (i^? \leq n)$
BECAUSE	$\{(n, m, j, j'). n \geq 1, j \leq n-1, j > m\} \subseteq$ $\{(n, m, j, j'). n-1 > m\}$

C Satisfiability checking via optimisation

In this appendix we show how to use an optimisation toolbox for checking satisfiability of constraints arising at the end of type checking. As an example, consider the constraint

$$\forall n m j i' \exists i. n \geq 1, 0 \leq j \leq m, 0 \leq i' \leq m(n-1) \implies 0 \leq i \leq mn \wedge i = j + i'$$

from the cons branch of the function `rel`. First, note that i is given: $j + i'$. All we have to check is whether $n \geq 1, 0 \leq j \leq m, 0 \leq i' \leq m(n-1) \implies 0 \leq j + i' \leq mn$. We check if the negation of this predicate is satisfiable using the method `fmincon-Constrained nonlinear minimization` from the optimisation toolbox of Matlab.

Start encoding the size variables with Matlab variables: $\mathbf{x}(1) := n, \mathbf{x}(2) := m, \mathbf{x}(3) := j, \mathbf{x}(4) := i'$. Create a file with the objective function to be minimised. Since we are checking satisfiability, it may be just

```
function f = objfun(x)
f = 0;
```

We want to check if there exists n, m, j and i' such that $n \geq 1, 0 \leq j \leq m, 0 \leq i' \leq m(n-1), j + i' \leq -1$ altogether. Create a file with constraints of the form $p(\mathbf{x}) \leq 0$:

```
function [c,ceq] = nonlconstr(x)
c = [-x(1) * x(2) + x(2) + x(4);
      x(3)-x(2);
      x(3) + x(4) + 1;
      -x(2);
      -x(3);
      -x(4)];
ceq = [];
```

The constraint $\mathbf{x}(1) \geq 1$ is presented in the dialog box for linear constraints. Running the tool gives:

```
Optimization running.
Optimization terminated.
Objective function value: 0.0
Optimization terminated: no feasible solution found.
Magnitude of directional derivative in search direction
less than 2*options.TolFun but constraints are not satisfied.
```

Now, we want to check if there exists n, m, j , and i' such that $n \geq 1, 0 \leq j \leq m, 0 \leq i' \leq m(n-1), j + i' \geq mn + 1$ altogether. This constraint implies to one shown below:

```
function [c,ceq] = nonlconstr(x)
c = [- x(1) * x(2) + x(2) + x(4);
      x(3)-x(2);
      x(2) - x(3) + 1];
ceq = [];
```

Running the tool gives the unfeasability result as above. Thus the type is accepted.

This approach can be used for type annotations of the form $p(\bar{n}) + i$, where $0 \leq i \leq \delta(\bar{n})$ with quadratic p and δ .