Polynomial Size Analysis with Families of Piecewise Polynomials for Functional Programs over Lists *

With soundness proof and examples in detail

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

Institute for Computing and Information Sciences
Radboud University Nijmegen

Abstract. Size analysis can play an important role in optimising memory management and in preventing failure due to memory exhaustion.
Static size analysis can be performed using size-aware type systems. In size-aware type systems types express *output-on-input size dependencies* where sizes of outputs depend on sizes of inputs but they do not depend directly on the actual values.
We present a novel type system for a strict functional language. We introduce size annotations that are *indexed families* of piecewise polynomials. Families are defined using indices. They collect possible different size dependencies of a single function.
With families *compositionality* becomes straightforward, even for *non-monotonic* size dependencies. *Lower and upper* size *bounds* can be expressed in a natural way as members of the family. Furthermore, families make it possible to allow *non matrix-like structures* where inner lists have different lengths.
The type system is proven to be *sound* with respect to a standard operational semantics. Type checking in our type system is not fully decidable. However, we identify two large decidable subclasses. Moreover, the type system admits a semi-automatic inference procedure.

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 strict functional language. Knowing lower and upper bounds of these dependencies one can apply *amortisation* [16] to check and infer tight non-linear bounds on heap consumption [21]. Moreover, information about lower size bounds on outputs helps in optimising memory management. To reduce the number of allocation operations, e.g., one

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

can allocate in advance a chunk of memory equal to the minimal size of an output array.

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 piecewise polynomials. A simple example of a piecewise polynomial is $\max_0(p)$, where p is a polynomial. It denotes the expression that is equal to p if $p \geq 0$, and it is 0 otherwise. For the sake of convenience, in this paper piecewise polynomials are simply called polynomials, unless otherwise noted.

1.1 Exploring size dependencies

We restrict our attention to a strict language with polymorphic lists as the only data type. For this language we develop a sound size-aware type system and 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 insert that given a binary predicate $\mathsf{g}\colon \alpha \times \alpha \to \mathsf{Bool}$, some z of type α and a list l of type $\mathsf{L}(\alpha)$, inserts z into the list if the list does not contain an element z', such that $\mathsf{g}(\mathsf{z}, \mathsf{z}')$ holds:

```
insert(g, z, l) =match l with
                 | Nil ⇒ Cons(z, Nil)
                 | Cons(hd, tl) ⇒if g(z, hd) then l
                                 else Cons(hd, insert(g, z, tl))
```

It is easy to see that the length of the output is equal or one more than the length of the input. Using an integer index i, we express this size dependency with the type $\mathsf{insert} : (\alpha \times \alpha \to \mathsf{Bool}) \times \alpha \times \mathsf{L}_n(\alpha) \to \mathsf{L}_{n+i}^{\exists i.\, 0 \leq i \leq 1}(\alpha)$.

Consider as a second example a function that deletes from a list the first occurrence of an element that satisfies a given predicate if such an element exists:

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

Using integer indices i and j, we express the size dependency for delete with the type $\mathsf{delete} : (\alpha \to \mathsf{Bool}) \times \alpha \times \mathsf{L}_n(\alpha) \to \mathsf{L}_j^{\exists\, j.Q(n,\, j)}(\alpha)$, where $Q(n, j)$ abbreviates $j = n \vee \big((n = 0 \to j = 0) \wedge (n \geq 1 \to j = n - 1)\big)$. This type represents the two-element family of piecewise polynomials consisting of $p_0(n) = n$ and $p_1(0) = 0$, $p_1(n) = n - 1$ for $n \geq 1$. Using $\max_0$ the type for delete can be expressed as $(\alpha \to \mathsf{Bool}) \times \alpha \times \mathsf{L}_n(\alpha) \to \mathsf{L}_{\max_0(n-i)}^{\exists i.\, 0 \leq i \leq 1}(\alpha)$.

Take as a last example the function concat that given a list of lists appends its elements:

```
concat(l) =match l with
           | Nil ⇒ Nil
           | Cons(hd, tl) ⇒ append(hd, concat(tl))
```

Assuming that l has n inner lists that are not necessarily of the same length, but their lengths do not exceed m, the type of concat is $\mathsf{L}_n(\mathsf{L}_i^{\exists\, i.\, 0\le i\le m}(\alpha)) \to \mathsf{L}_j^{\exists\, j.\, 0\le i\le nm}(\alpha)$.

Predicates that delimit indices are of the form $\exists \bar{i}.\, Q(\overline{n}, \bar{i})$, where $\overline{n} = (n_1, .. n_l)$ are the sizes of the inputs, $\bar{i} = (i_1, .. i_k)$ is a vector of indices and Q is a quantifier-free first-order arithmetic formula. The type system we introduce is sound w.r.t. this general form of delimiting predicates. For instance, $\mathsf{L}_i^{\exists i.\, i\ge 0}(\alpha)$ is an admissible output type, although not an informative one from a size point of view. In practice one is interested in predicates that define good approximations of actual size dependencies. Typically, Q is a conjunction of atomic formulae of the form $q(\overline{n}) \ge 0 \to g(\overline{n}, i_1, .. i_{j-1}) \le i_j \le h(\overline{n}, i_1, .. i_{j-1})$ for some polynomials q, g, h.

The principal type of a given function has the most precise size annotation. However, as sketched in Section 3, not every function has a principal type in this type system.

In general, the size of an output of a function depends not only on the sizes of the corresponding inputs, but on their values. Therefore, an expressive size-annotated type system may involve dependent typing. Using families we can express dependencies of output sizes on input values avoiding dependent types. When a program has an indexed family of polynomials as its output-on-input size dependency, *for each input value, there is an index such that the corresponding polynomial in the family precisely describes the size of the output.* The appropriate index is inferred by the type checker, making the dependency on the input value implicit. Values are abstracted to indices.

1.2 Our contribution

Most papers in size analysis study upper bounds for *monotonic* size dependencies (see section on related work). The novelty of our paper lies in the fact that we use *families* of *non-monotonic* polynomials to express size dependencies and study decidability of size checking in this setting.

This work continues a series of papers where we have studied output-on-input polynomial size dependencies, where the polynomials are not necessary monotonic. In [19] we designed a type system where each type is annotated with a single polynomial size expression. It allows to type function definitions where the size of the output depends on the sizes of the inputs, but not on their values. For instance, $\mathsf{append}\colon \mathsf{L}_n(\alpha) \times \mathsf{L}_m(\alpha) \to \mathsf{L}_{n+m}(\alpha)$, whereas insert, delete, concat do not have a type in that system. We also developed a test-based annotation inference procedure for that system in [22].

The advantage of using families of polynomials is at least four-fold:

- families provide a convenient way to manage *compositionality* of non-monotonic annotations;
- if an input of a function is a list of lists (of lists, etc.), then *internal lists are not necessarily of the same length*;
- the system admits a *semi-automatic inference procedure*;

– type-checking reduces to checking predicates in first-order arithmetic, which facilitates studying decidability issues.

Consider these advantages in more detail.

Compositionality of non-monotonic size dependencies In general, it is difficult to obtain a bound on the size dependency of the composition $\mathsf{f}_2(\mathsf{f}_1(-))$ of two programs f_1 and f_2 if the bound for f_2 is non-monotonic. One has to analyse monotonicity areas of f_2 and its extrema. With families of polynomials the composition rule is straightforward. Let, for instance, $p_2(n) = (n-2)^2$ and $1 \leq p_1(n) \leq 4$ be size dependencies for the programs f_2 and f_1, respectively. A naive substitution of the bounds of p_1 into p_2 gives $p_2(1) \leq p_2(p_1(n)) \leq p_2(4)$, i.e. $1 \leq p_2(p_1(n)) \leq 4$, which is incorrect. The reason is that p_2 has a minimum $n_0 = 2$ with $f_2(2) = 0$. On $[0, 2]$ the polynomial p_2 decreases, and it increases on $[2, +\infty)$. The correct bounds for the composition on $[1, 4]$ are $0 \leq p_2(p_1(n)) \leq 4$. By expressing size dependencies via families of polynomials we can avoid the analysis of (multivariate) monotonicity and extrema. Indeed, let $p_1(n) = i$, where $1 \leq i \leq 4$. Then $p_2(p_1(n)) = (i-2)^2$, with $1 \leq i \leq 4$. If needed one can simplify this expression and obtain $p_2(p_1(n)) = i'$ with $0 \leq i' \leq 4$.

Variable length of input lists The presented type system is quite expressive due to the fact that inner lists are not required to have the same length. In previous works [19, 20] we dealt only with functions over matrix-like structures like $\mathsf{multiply}\colon \mathsf{L}_n(\mathsf{L}_m(\mathsf{Int})) \times \mathsf{L}_m(\mathsf{L}_k(\mathsf{Int})) \to \mathsf{L}_n(\mathsf{L}_k(\mathsf{Int}))$.

Idea behind annotation-inference procedure First, note that any program that has polynomials lower and upper size bounds, one can express the output size dependency in terms of indexed polynomial families. Let the lower and upper bounds be $p_1(\overline{n})$ and $p_2(\overline{n})$, respectively, with $\overline{n}$ representing the vector of input sizes. Then the output size can be expressed as $p_1(\overline{n})+i$ with $0 \leq i \leq \delta(\overline{n})$, where $\delta(\overline{n}) = p_2(\overline{n}) - p_1(\overline{n})$. Here we briefly outline the inference procedure presented in a separate paper [18]. It consists of two stages. First, a candidate family is constructed. Given a function body one can construct conditional rewriting rules expressing the size dependency for that function. For instance, the size dependency for delete is given by

$$\begin{gathered} \vdash p(0) \to 0 \\ n \geq 1 \vdash p(n) \to \ n-1 \mid 1 + p(n-1) \end{gathered}$$

where $\vdash$ denotes logical consequence and $\mid$ denotes optional application of rewriting rules. Using such rewriting rules the procedure generates the points lying on the bounds. In the example, from the rewriting rules a user can easily see that $p(0) = 0$. So, (s)he has to find bounds for the branch $n \geq 1$. If the user chooses as parameters for delete a degree 1 of lower and upper bounds, an initial point $n^0 = 1$ and a step 1, the procedure generates $n^0 = 1$, $n^1 = 2$. Next, the rewriting rules are used to calculate the sets $p(n)$. For delete we obtain $p(n^0) = \{0, 1\}$

and $p(n^1) = \{1, 2\}$. The procedure picks up the minimal and maximal values from each of the set and computes the coefficients of the lower and upper polynomial bounds as the solutions of two corresponding linear systems. In the example, the lower bound $p_{\min}(n) = n - 1$ is computed from the nodes $\{(1, 0), (2, 1)\}$, and the upper bound $p_{\max}(n) = n$ from $\{(1, 1), (2, 2)\}$. It follows that the lower bound is a piecewise polynomial $p_{\min}(0) = 0$, $p_{\max}(n) = n - 1$ for $n \geq 1$, and the upper bound is the polynomial $p_{\max}(n) = n$. They define an indexed family of polynomials, which may be presented as $\{p_{\min}(n) + i\}_{i=0}^{\delta(n)}$, where $\delta(n) = p_{\max}(n) - p_{\min}(n)$. Simplifying, we see that for $n = 0$ the family is given by $\{i\}$ with $0 \leq i \leq \delta(0) = 0$, and for $n \geq 1$ it is given by $\{n - 1 + i\}$ with $0 \leq i \leq \delta(n) = 1$. Finally, one obtains the family given for delete earlier in this section. The first stage is completed. The second stage consists of validating the candidate family within the type system presented in this paper. If the type checker rejects the candidate, then the user may repeat the cycle "generate a candidate and check" using other parameters.

Studying decidability In our type system decidability issues amount to decidability in an underlying arithmetic. The choice of the set for possible values of the size variables (e.g. integer or real) influences not only the performance of an implementation but also decidability and the set of admissible types.

We describe *two large classes* of programs for which type-checking is *decidable in integer arithmetic*.

1.3 Contents of the paper

This paper is organised as follows. In Section 2 we define the programming language and its operational semantics. Section 3 defines its size-aware type system, the semantics of program values w.r.t. zero-order types and sketches the proof of soundness. In this section we discuss principal types as well. 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 appendix 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.

2 Language

The type system is designed for a strict functional language over integers and (polymorphic) lists. Algebraic data types could be added as we did in [20]. Language expressions are defined by the following grammar:

$$
\begin{array}{rl}
\mathit{Basic}\ b ::= & c \mid \mathsf{unop\ x} \mid \mathsf{x\ binop\ y} \mid \mathsf{Nil} \mid \mathsf{Cons(z,\ l)} \mid \\
& \mathsf{f}(\mathsf{g}_1, \ldots, \mathsf{g}_l,\ \mathsf{z}_1, \ldots, \mathsf{z}_k) \\
\mathit{Expr}\ \ e ::= & \ b \mid \mathsf{if\ x\ then}\ e_1\ \mathsf{else}\ e_2 \mid \mathsf{let\ z} = b\ \mathsf{in}\ e_1 \\
& \mid \mathsf{match\ l\ with} \mid \mathsf{Nil} \Rightarrow e_1 \\
& \qquad\qquad\qquad \mid \mathsf{Cons}(\mathsf{z}_{\mathsf{hd}},\ \mathsf{l}_{\mathsf{tl}}) \Rightarrow e_2 \\
& \mid \mathsf{letfun\ f}(\mathsf{g}_1, \ldots, \mathsf{g}_l,\ \mathsf{z}_1, \ldots, \mathsf{z}_k) = e_1\ \mathsf{in}\ e_2
\end{array}
$$

where c ranges over integer and boolean constants True and False, x and y denote integer or boolean program variables, l ranges over lists, z denotes a program variable of a zero-order type (it can be a list), g ranges over higher-order program variables, unop is a unary operation, either $-$ or $\neg$, binop is one of the integer or boolean binary operations, and f denotes a function name.

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 our language as an intermediate one.

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. A program value is either an integer or a boolean constant, a location or the null address, and a heap is a finite partial mapping from locations and fields into program values. Formally:

$$
\begin{array}{rlrl}
\mathit{Address} & \mathtt{adr} ::= \ell \mid \mathtt{NULL} & \ell \in \mathit{Loc} & \quad c \in \mathsf{Int} \cup \mathsf{Bool} \\
\mathit{Val} & v ::= c \mid \mathtt{adr} & \mathit{Heap} & \quad h\colon \mathit{Loc} \rightharpoonup \{\mathsf{hd}, \mathsf{tl}\} \rightharpoonup \mathit{Val}
\end{array}
$$

We write $h.\ell.\mathsf{hd}$ and $h.\ell.\mathsf{tl}$ for the applications $h(\ell)(\mathsf{hd})$ and $h(\ell)(\mathsf{tl})$, which denote the values stored in the heap h at the location ℓ at its fields hd and tl, respectively. Let $h.\ell.[\mathsf{hd} := v_h,\ \mathsf{tl} := v_t]$ 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.

We introduce a *frame store* as a finite partial 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, $\mathit{Store}\ s\colon \mathit{ProgramVars} \rightharpoonup \mathit{Val}$.

Using heaps and a frame store, and maintaining a mapping $\mathcal{C}$ from function names to function signatures and bodies, the operational semantics of program expressions is defined by a set of standard rules (see the appendix for more detail).

Here we show three of them:

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

$$\frac{h.s(\mathsf{l}).\mathsf{hd} = v_{\mathsf{hd}} \qquad h.s(\mathsf{l}).\mathsf{tl} = v_{\mathsf{tl}} \qquad s[\mathsf{hd} := v_{\mathsf{hd}}, \mathsf{tl} := v_{\mathsf{tl}}];\, \mathcal{C};\, h \vdash e_2 \rightsquigarrow v;\, h'}{s;\, h;\, \mathcal{C} \vdash \begin{cases}\mathsf{match\ l\ with}\\ \mid\ \mathsf{Nil} \Rightarrow e_1 & \rightsquigarrow v;\, h'\\ \mid\ \mathsf{Cons}(\mathsf{hd},\, \mathsf{tl}) \Rightarrow e_2\end{cases}}\ \textsc{OSMatch-Cons}$$

$$\frac{\begin{array}{c}s(\mathsf{z}'_1) = v_1 \ \ldots\ s(\mathsf{z}'_k) = v_k \qquad \mathcal{C}(\mathsf{f}) = (\mathsf{g}_1, \ldots, \mathsf{g}_{k'},\ \mathsf{z}_1, \ldots, \mathsf{z}_k) \times e_f\\ [\mathsf{z}_1 := v_1, \ldots, \mathsf{z}_k := v_k];\, h;\, \mathcal{C} \vdash\\ e_f[\mathsf{g}_1 := \mathsf{f}_1, \ldots,\, \mathsf{g}_{k'} := \mathsf{f}_{k'}] \rightsquigarrow v;\, h'\end{array}}{s;\, h;\, \mathcal{C} \vdash \mathsf{f}(\mathsf{f}_1, \ldots, \mathsf{f}_{k'},\ \mathsf{z}'_1, \ldots, \mathsf{z}'_k) \rightsquigarrow v;\, h'}\ \textsc{OSFunApp}$$

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 polynomials in some ordered 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 polynomials are defined by the following grammar:

$$\begin{array}{rll}\textit{SizeExpressions} & p ::= & \cancel{c} \mid I \mid SV \mid\\ & & p + p \mid p - p \mid p * p \mid \max_0(p),\end{array}$$

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 and can only be evaluated to non-negative values. Furthermore, we use $\overline{i}$ and $\overline{n}$ 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. Lists are annotated with an expression and a predicate representing its length:

$$\textit{Types} \quad \tau ::= \mathsf{Int} \mid \mathsf{Bool} \mid \alpha \mid \mathsf{L}^{\exists\, \overline{i}.\, Q(\overline{n},\, \overline{i})}_{p(\overline{n},\, \overline{i})}(\tau),$$

where α is a type variable and $Q(\overline{n}, \overline{i})$ is a quantifier-free first-order arithmetic formula. When we speak about a *type annotation* we mean a size expression together with a predicate delimiting its indices. Note that in a type $\mathsf{L}^{\exists\, \overline{i}.\, Q(\overline{n},\, \overline{i})}_{p(\overline{n},\, \overline{i})}(\tau)$ *the scope of the existential quantifier* is composed by Q and p, but not τ. Thus the types $\mathsf{L}^{\exists\, \overline{i}.\, Q(\overline{n},\, \overline{i})}_{p(\overline{n},\, \overline{i})}(\tau)$ and $\mathsf{L}^{\exists\, \overline{j}.\, Q(\overline{n},\, \overline{j})}_{p(\overline{n},\, \overline{j})}(\tau)$ are the same for all τ. For instance, the types $\mathsf{L}^{\exists\, i.\, 0 \le i \le 1}_{n+i}(\mathsf{L}^{\exists\, i.\, 0 \le i \le 1}_{n^2+i}(\alpha))$ and $\mathsf{L}^{\exists\, i'.\, 0 \le i' \le 1}_{n+i'}(\mathsf{L}^{\exists\, i.\, 0 \le i \le 1}_{n^2+i}(\alpha))$ represent the same type.

To ease reading, in the examples we omit explicit non-negativity conditions for size variables and indices, and existential quantifiers for indices. For instance, the type $\mathsf{L}_{n+i}^{\exists\, i.\, 0\leq i\leq 1}(\alpha)$ will be written as $\mathsf{L}_{n+i}^{i\leq 1}(\alpha)$. If p does not depend on indices, we omit the predicate in the type. For instance, we write $\mathsf{L}_{n+m}(\alpha)$ instead of $\mathsf{L}_{n+m}^{True}(\alpha)$.

The sets $TV(\tau)$ and $SV(\tau)$ represent the sets of type and size variables of a type τ, respectively and are defined inductively in the obvious way.

Since all types of the form $\mathsf{L}_0(\mathsf{L}_{p_1(\overline{n},\,\overline{i}_1)}^{Q_1(\overline{n},\,\overline{i}_1)}(\ldots \mathsf{L}_{p_k(\overline{n},\,\overline{i}_1)}^{Q_k(\overline{n},\,\overline{i}_k)}(\alpha)\ldots))$ represent the same data structure (an empty list of lists), we assume that such types are equivalent to a similarly nested type of empty lists: $\mathsf{L}_0(\mathsf{L}_0(\ldots \mathsf{L}_0(\alpha)\ldots))$. For instance, $\mathsf{L}_0(\mathsf{L}_m(\mathsf{Int}))$ represents the same structure as $\mathsf{L}_0(\mathsf{L}_0(\mathsf{Int}))$. Therefore, we assume that $SV(\mathsf{L}_0(\tau)) = \emptyset$, for all τ.

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

$$\mathit{GroundTypes} \quad \tau^{\bullet} ::= \tau \;\text{ such that } SV(\tau) = \emptyset \wedge TV(\tau) = \emptyset\,.$$

For instance, Int, $\mathsf{L}_i^{i\leq 5}(\mathsf{Bool})$ and $\mathsf{L}_i^{1\leq i\leq 5}(\mathsf{L}_j^{j\leq 3}(\mathsf{Int}))$ are ground types, whereas α, $\mathsf{L}_{n+5}(\mathsf{Int})$ and $\mathsf{L}_i^{i\leq n}(\mathsf{Bool})$ are not. Examples of inhabitants of ground types are $[\mathsf{True},\ \mathsf{True},\ \mathsf{False}]$ for $\mathsf{L}_i^{i\leq 5}(\mathsf{Bool})$ and $[\,[1,\ 2,\ 3],\ [1,\ 2],\ [\,]\,]$ for $\mathsf{L}_i^{1\leq i\leq 5}(\mathsf{L}_j^{j\leq 3}(\mathsf{Int}))$. We define the formal semantics of zero-order types in Section 3.2.

Let τ° denote a zero-order type without predicate, whose size expressions are just size variables, e.g. $\mathsf{L}_n(\alpha)$. *Function types* are then defined as

$$\mathit{FunctionTypes} \quad \tau^{f} ::= \tau_1^{f} \times \ldots \times \tau_{k'}^{f} \times \tau_1^{\circ} \times \ldots \times \tau_k^{\circ} \to \tau_0,$$

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

This definition of function types does not admit lists of lists with different lengths as function inputs. This is because the size annotations of input types can only be size variables. For instance, the matrix-like structures $\mathsf{L}_n(\alpha)$ and $\mathsf{L}_n(\mathsf{L}_m(\alpha))$ are allowed input types, whereas $\mathsf{L}_n(\mathsf{L}_i^{i\leq m}(\alpha))$ is not. We will extend this definition in Section 3.1. This extension admits as function inputs not only lists of lists with different length as $\mathsf{L}_n(\mathsf{L}_i^{i\leq m}(\alpha))$, but lists with constants annotations, like $\mathsf{L}_n(\mathsf{L}_2(\alpha))$.

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:

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

3.1 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 size annotations and appears as side conditions in some typing rules. Intuitively, $D \vdash \tau_1 \preccurlyeq \tau_2$ means that under the conditions on D, τ_2 is less restrictive than τ_1. For instance, the following subtyping judgements hold:

$$n - 3 = 0 \vdash \mathsf{L}_i^{i \le n}(\alpha) \preccurlyeq \mathsf{L}_j^{j \le 4}(\alpha) \text{ and}$$
$$n = 0, m = 0 \vdash \mathsf{L}_{n+m}(\mathsf{L}_i^{i \le 5}(\alpha)) \preccurlyeq \mathsf{L}_n(\mathsf{L}_j^{j \le 2}(\alpha))$$

We use commas to separate predicates in D so, for instance, $n = 0$, $m = 0$ in the example above stands for $n = 0 \wedge m = 0$. Formally, an (inductive) relation $D \vdash \tau_1 \preccurlyeq \tau_2$, which is read as τ_1 *is a subtype* of τ_2 under D, holds if an only if

- $\tau_1 = \tau_2$ is a basic type (Int, Bool or a type variable α).
- $\tau_1 = \mathsf{L}_{p(\overline{n},\, \overline{i})}^{\exists\, \overline{i}.\, Q(\overline{n},\overline{i})}(\tau_3)$ and $\tau_2 = \mathsf{L}_{p'(\overline{n},\, \overline{j})}^{\exists\, \overline{j}.\, Q'(\overline{n},\, \overline{j})}(\tau_4)$ have the same underlying type, and $D(\overline{n}) \wedge Q(\overline{n}, \overline{i}) \vdash \exists\, \overline{j}.\ Q'(\overline{n}, \overline{j}) \wedge p(\overline{n}, \overline{i}) = p'(\overline{n}, \overline{j})$ and $\exists\ \overline{n}\ \overline{i}.\ D(\overline{n}) \wedge Q(\overline{n}, \overline{i}) \wedge p(\overline{n}, \overline{i}) \geq 1$ implies $D \vdash \tau_3 \preccurlyeq \tau_4$. The last condition is needed to avoid comparing τ_3 and τ_4 in cases like $\tau_1 = \mathsf{L}_0(\tau_3)$ and $\tau_2 = \mathsf{L}_0(\tau_4)$.

The typing judgement is defined by the following rules:

$$\frac{}{D;\, \Gamma \vdash_\Sigma c\colon \mathsf{Int}}\ \textsc{IConst} \qquad \frac{}{D;\, \Gamma \vdash_\Sigma b\colon \mathsf{Bool}}\ \textsc{BConst}$$

$$\frac{D \vdash \tau \preccurlyeq \tau'}{D;\, \Gamma,\ \mathsf{z}\colon \tau \vdash_\Sigma \mathsf{z}\colon \tau'}\ \textsc{Var} \qquad \frac{D \vdash \mathsf{L}_0(\tau) \preccurlyeq \tau'}{D;\, \Gamma \vdash_\Sigma \mathsf{Nil}\colon \tau'}\ \textsc{Nil}$$

$$\frac{D \vdash \mathsf{L}_{p(\overline{n},\overline{i})+1}^{\exists\, \overline{i}.\, Q(\overline{n},\overline{i})}(\tau_2) \preccurlyeq \tau' \qquad D \vdash \tau_1 \preccurlyeq \tau_2}{D;\, \Gamma,\ \mathsf{hd}\colon \tau_1,\ \mathsf{tl}\colon \mathsf{L}_{p(\overline{n},\overline{i})}^{\exists\, \overline{i}.\, Q(\overline{n},\overline{i})}(\tau_2) \vdash_\Sigma \mathsf{Cons}(\mathsf{hd}, \mathsf{tl})\colon \tau'}\ \textsc{Cons}$$

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

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

$$\frac{\begin{array}{c}\Sigma(f) = \tau_1^f \times \ldots \times \tau_{k'}^f \times \tau_1^\circ \times \cdots \times \tau_k^\circ \to \tau_0 \\ \Sigma(\mathsf{g}_1) = \tau_1^f, \ldots, \Sigma(\mathsf{g}_{k'}) = \tau_{k'}^f \\ \mathsf{z}_1\colon \tau_1^\circ, \ldots, \mathsf{z}_k\colon \tau_k^\circ \vdash_\Sigma e_1\colon \tau_0 \qquad D;\, \Gamma \vdash_\Sigma e_2\colon \tau'\end{array}}{D;\, \Gamma \vdash_\Sigma \mathsf{letfun}\ f(\mathsf{g}_1, \ldots, \mathsf{g}_{k'},\ \mathsf{z}_1, \ldots, \mathsf{z}_k) = e_1\ \mathsf{in}\ e_2\colon \tau'}\ \textsc{LetFun}$$

Function-application rule In this section we give the light-weight function-application rule for functions over matrix-like structures. The general rule will be given later in Section 3.1.

Recall, that functions over matrix-like structures have signatures of the form $\tau_1^f \times \ldots \times \tau_{k'}^f \times \tau_1^\circ \times \ldots \times \tau_k^\circ \to \tau_0$, where τ_l°, with $1 \leq l \leq k$, have only variables as size annotations. The function application rule looks as follows:

$$\frac{\begin{array}{c}\Sigma(f) = \tau_1^f \times \ldots \times \tau_{k'}^f \times \tau_1^\circ \times \ldots \times \tau_k^\circ \to \tau_0 \\ \text{the underlying type of } \mathsf{g}_{l'} \text{ is an instance} \\ \text{of the underlying type of } \tau_{l'}^f, \text{ where } 1 \leq l' \leq k' \\ D \vdash \sigma(\tau_0) \preccurlyeq \tau \\ D \vdash C\end{array}}{D;\, \Gamma, \mathsf{z}_1 \colon \tau_1, \ldots, \mathsf{z}_k \colon \tau_k \vdash_\Sigma \mathsf{f}(\mathsf{g}_1, \ldots, \mathsf{g}_{k'},\ \mathsf{z}_1, \ldots, \mathsf{z}_k) \colon \tau}\ \textsc{FunApp}$$

where σ instantiates type and size variables of the parameter types τ_l°, such that $D \vdash \sigma(\tau_l^\circ) \equiv \tau_l$, and C is a set of equivalences over the argument types τ_l, where an equivalence $\tau \equiv \tau'$ abbreviates $\tau \preccurlyeq \tau' \wedge \tau' \preccurlyeq \tau$. C is used to collect relationships that must hold among the arguments, i.e., if $\mathsf{scalarprod} : \mathsf{L}_m(\mathsf{Int}) \times \mathsf{L}_m(\mathsf{Int}) \to \mathsf{Int}$ is called with arguments $\mathsf{l} \colon \tau$ and $\mathsf{l}' \colon \tau'$, then C will contain a condition requiring τ and τ' to have the same size. Next we give a procedure for computing σ and generating the set C:

1. Map the formal types onto the actual types. This is a trivial step: $\tau_l^\circ \mapsto \tau_l$. Moreover, $C = \{\tau_l \equiv \tau_{l'} \mid \tau_l^\circ = \tau_{l'}^\circ\}$. Let, for instance, a function $\mathsf{f} : \mathsf{L}_m(\alpha) \times \mathsf{L}_m(\alpha) \to \mathsf{L}_{m+j}^{j \leq m}(\alpha)$ be called with arguments $\mathsf{l} \colon \mathsf{L}_{n+i}^{i \leq 1}(\alpha)$ and $\mathsf{l}' \colon \mathsf{L}_{n-n'+i}^{i \leq 1}(\alpha)$. Then $\mathsf{L}_m(\alpha) \mapsto \mathsf{L}_{n+i}^{i \leq 1}(\alpha)$ and $C = \{\mathsf{L}_{n+i}^{i \leq 1}(\alpha) \equiv \mathsf{L}_{n-n'+i}^{i \leq 1}(\alpha)\}$. The condition $D \vdash C$ means that C holds if D holds. In our example, the equivalence holds if D contains $n' = 0$.
2. Define σ on size variables as well: tell explicitly which size variables are mapped into which size annotations. For instance, in the example above, $\sigma(m) = n + i$, where $i \leq 1$.
   In general, if $\tau^\circ = \mathsf{L}(\ldots \mathsf{L}_m(\tau^{\circ\prime}) \ldots)$ is mapped onto $\tau = \mathsf{L}(\ldots \mathsf{L}_{p(\overline{n},\overline{i})}^{\exists\ \overline{i}.\ Q(\overline{n},\overline{i})}(\tau') \ldots)$, then $\sigma(m) = p(\overline{n}, \overline{i})$, where i satisfies $Q(\overline{n}, \overline{i})$.
3. Compute $\sigma(\tau_0)$. For the function f from the example above, σ applied to the formal output type τ_0 gives $\sigma(\mathsf{L}_{m+j}^{j \leq m}(\alpha)) = \mathsf{L}_{n+i+j}^{i \leq 1 \wedge j \leq n+i}(\alpha)$.
   The general case of $\sigma(\tau_0)$ is computed as follows. Let $\sigma(m) = p'(\overline{n}, \overline{i})$, where $Q'(\overline{n}, \overline{i})$ holds. Then

$$\sigma\Big(\mathsf{L}(\ldots \mathsf{L}_{p(m,\overline{j})}^{\exists\, \overline{j}.\, Q(m,\overline{j})}(\ldots \mathsf{L}(\alpha) \ldots) \ldots)\Big) = \mathsf{L}(\ldots \mathsf{L}_{p(p'(\overline{n},\overline{i}),\overline{j})}^{\exists\, \overline{i}\,\overline{j}.\, Q'(\overline{n},\overline{i}) \wedge Q(p'(\overline{n},\overline{i}),\overline{j})}(\ldots \mathsf{L}(\alpha) \ldots) \ldots)$$

Match-rule In the premises of the MATCH rule below, we replace the indices $\overline{i}$ of the type of the matched variable l with fresh size variables $\overline{n}^{\overline{i}} \notin \{\overline{n}\} \cup$

$SV(\Gamma, \tau, \tau')$ (i.e. we *skolemise the indices*). Moreover, skolemising the indices keeps the information that the tail tl is shared with the list l:

$$\frac{\left.\begin{array}{l} Q(\overline{n},\, \overline{n^i}),\, p(\overline{n},\, \overline{n^i}) = 0,\, D;\, \Gamma,\, \mathsf{l}\colon \mathsf{L}_{p(\overline{n},\, \overline{n^i})}(\tau) \vdash_\Sigma e_{\mathsf{Nil}}\colon \tau' \\ Q(\overline{n},\, \overline{n^i}),\, p(\overline{n},\, \overline{n^i}) \geq 1,\, D; \\ \Gamma, \mathsf{hd}\colon \tau,\, \mathsf{l}\colon \mathsf{L}_{p(\overline{n},\, \overline{n^i})}(\tau),\, \mathsf{tl}\colon \mathsf{L}_{p(\overline{n},\, \overline{n^i})-1}(\tau) \end{array}\right\} \vdash_\Sigma e_{\mathsf{Cons}}\colon \tau' \qquad \mathsf{hd}, \mathsf{tl} \notin dom(\Gamma) \qquad \overline{n^i} \text{ are fresh in } \Gamma,\ \tau,\ \tau' \text{and } \overline{n}}{D;\, \Gamma,\, \mathsf{l}\colon \mathsf{L}^{\exists\, \overline{i}.\, Q(\overline{n},\, \overline{i})}_{p(\overline{n},\, \overline{i})}(\tau) \vdash_\Sigma \begin{array}{l} \mathsf{match\ l\ with} \\ \quad \mid \mathsf{Nil} \Rightarrow e_{\mathsf{Nil}} \\ \quad \mid \mathsf{Cons(hd,\, tl)} \Rightarrow e_{\mathsf{Cons}} \end{array} : \tau'}$$

MATCH with $\exists - I$

Sharing arises not only due pattern-matching but also due to let-bindings where the bound variable is passed as argument in several function calls. This will be considered in more detail in Section 3.1. The predicates $p(\overline{n},\, \overline{n^i}) = 0$ and $p(\overline{n},\, \overline{n^i}) \geq 1$ are used at the end of type checking when numerical entailments are to be proved.

Example of type-checking: function insert First, we give the definition of insert in the desugared syntax, where compositions of program expressions are given via let-bindings:

$$\begin{array}{l} \mathsf{insert(g,\, z,\, l)} = \mathsf{match\ l\ with} \\ \qquad \mid \mathsf{Nil} \Rightarrow \mathsf{let\ z'} = \mathsf{Nil\ in\ Cons(z,\, z')} \\ \qquad \mid \mathsf{Cons(hd,\, tl)} \Rightarrow \mathsf{if\ g(z,\, hd)\ then\ l} \\ \qquad\qquad\qquad\qquad \mathsf{else\ let\ l'} = \mathsf{insert(g,\, z,\, tl)} \\ \qquad\qquad\qquad\qquad\quad \mathsf{in\ Cons(hd,\, l')} \end{array}$$

We are going to check the assertion $\mathsf{insert} : (\alpha \times \alpha \to \mathsf{Bool}) \times \alpha \times \mathsf{L}_n(\alpha) \to \mathsf{L}^{i \leq 1}_{n+i}(\alpha)$. Let e_{insert} denote the body of the definition of insert. For this example we will perform type checking in detail, in a backward subgoal-directed style (as used in theorem proving). For simplicity, in the context of a typing judgement, we mention only the variables relevant for the expression at hand.

1. The LETFUN-rule defines the main goal:
   $\mathsf{z}\colon \alpha,\ \mathsf{l}\colon \mathsf{L}_n(\alpha) \vdash_\Sigma e_{\mathsf{insert}}\colon \mathsf{L}^{i \leq 1}_{n+i}(\alpha)$.
2. Now we apply the MATCH-rule. The nil-branch gives the subgoal
   $n = 0;\ \mathsf{z}\colon \alpha \vdash_\Sigma \mathsf{let\ z'} = \mathsf{Nil\ in\ Cons(z,\, z')}\colon \mathsf{L}^{i \leq 1}_{n+i}(\alpha)$.
3. We follow the nil-branch. Applying the LET-rule we obtain two subgoals:
   $n = 0 \vdash_\Sigma \mathsf{Nil}\colon \tau^?$ in the let-binding and
   $n = 0;\ \mathsf{z}\colon \alpha,\ \mathsf{z'}\colon \tau^? \vdash_\Sigma \mathsf{Cons(z,\, z')}\colon \mathsf{L}^{i \leq 1}_{n+i}(\alpha)$ in the let-body.
4. Applying the NIL-rule in the binding we instantiate the unknown type $\tau^?$ with $\mathsf{L}_0(\tau^?_1)$.
5. Now the judgement for the let-body has the form
   $n = 0;\ \mathsf{z}\colon \alpha,\ \mathsf{z'}\colon \mathsf{L}_0(\tau^?_1) \vdash_\Sigma \mathsf{Cons(z,\, z')}\colon \mathsf{L}^{i \leq 1}_{n+i}(\alpha)$.

6. Apply the Cons-rule. Trivially, the unknown type $\tau_1^?$ is instantiated with α and we obtain the subgoal:
   $n = 0 \vdash \mathsf{L}_{0+1}(\alpha) \preccurlyeq \mathsf{L}_{n+i}^{i\leq 1}(\alpha)$.
7. Unfolding the definition of subtyping, we obtain:
   $n = 0 \vdash \exists i.\ 0 \leq i \leq 1 \ \wedge \ 1 = n + i$.
   Trivially $i = 1 - n = 1$.
8. Now we consider the cons-branch:
   $n \geq 1;\ \mathsf{z}\colon \alpha,\ \mathsf{tl}\colon \mathsf{L}_{n-1}(\alpha),\ \mathsf{l}\colon \mathsf{L}_n(\alpha) \vdash_\Sigma$
   $\mathsf{if}\ .\ \mathsf{then}\ .\ \mathsf{else}\ .\colon\ \mathsf{L}_{n+i}^{i\leq 1}(\alpha)$.
9. Apply the If-rule. In the true-branch we obtain the subgoal
   $n \geq 1;\ \mathsf{l}\colon \mathsf{L}_n(\alpha) \vdash_\Sigma \mathsf{l}\colon \mathsf{L}_{n+i}^{i\leq 1}(\alpha)$.
10. Apply the Var-rule. We obtain the subtyping:
    $n \geq 1 \vdash \mathsf{L}_n(\alpha) \preccurlyeq \mathsf{L}_{n+i}^{i\leq 1}(\alpha)$.
11. Unfolding the definition of subtyping, we obtain:
    $n = 0 \vdash \exists i.\ 0 \leq i \leq 1 \ \wedge \ n = n + i$.
    Trivially $i = 0$.
12. In the false-branch we obtain the subgoal
    $n \geq 1;\ \mathsf{z}\colon \alpha,\ \mathsf{hd}\colon \alpha,\ \mathsf{tl}\colon \mathsf{L}_{n-1}(\alpha) \vdash_\Sigma$
    $\mathsf{let}\ \mathsf{l}' = \mathsf{insert}(\mathsf{g}, \mathsf{z}, \mathsf{tl})\ \mathsf{in}\ \mathsf{Cons}(\mathsf{hd}, \mathsf{l}')\colon\ \mathsf{L}_{n+i}^{i\leq 1}(\alpha)$.
13. Apply the Let-rule. In the let-binding we obtain the subgoal: $n \geq 1;\ \mathsf{z}\colon \alpha,\ \mathsf{tl}\colon \mathsf{L}_{n-1}(\alpha) \vdash_\Sigma \mathsf{insert}(\mathsf{g}, \mathsf{z}, \mathsf{tl})\colon \tau^?$.
14. Application of the FunnApp-rule instantiates $\tau^?$ with $\sigma(\mathsf{L}_{n+i}^{i\leq 1}(\alpha))$, which is equal to $\mathsf{L}_{(n-1)+i}^{i\leq 1}(\alpha)$ due to $\sigma(n) = n - 1$.
15. Therefore, in the Let-body we have the subgoal
    $n \geq 1;\ \mathsf{hd}\colon \alpha,\ \mathsf{l}'\colon \mathsf{L}_{n-1+i}^{i\leq 1}(\alpha) \vdash_\Sigma \mathsf{Cons}(\mathsf{hd}, \mathsf{l}')\colon \mathsf{L}_{n+i}^{i\leq 1}(\alpha)$.
16. Applying the Cons-rule generates the subgoal
    $n \geq 1 \vdash \mathsf{L}_{n-1+i+1}^{i\leq 1}(\alpha) \preccurlyeq \mathsf{L}_{n+i}^{i\leq 1}(\alpha)$.
17. Unfolding the definition of subtyping yields the predicate
    $n \geq 1,\ 0 \leq i \leq 1 \vdash \exists\, i'.\ 0 \leq i' \leq 1 \wedge (n-1) + i + 1 = n + i'$. Trivially $i' = i$.

$\exists - I$ rule Skolemisation is needed to save information about *shared* sublists that appear due to let-bindings. To be able to memoise size of shared structures when necessary, we introduce as a standalone $\exists - I$-rule:

$$\frac{Q(\overline{n}, \overline{n^i}),\ D;\ \Gamma,\ \mathsf{l}\colon \mathsf{L}_{p(\overline{n}, \overline{n^i}, \overline{j})}^{\exists\, \overline{j}.\ P(\overline{n}, \overline{j})}(\tau) \vdash_\Sigma e\colon \tau' \qquad \overline{n^i} \text{ are fresh in } \Gamma,\ \tau,\ \tau' \text{ and } \overline{n}}{D;\ \Gamma,\ \mathsf{l}\colon \mathsf{L}_{p(\overline{n}, \overline{i}, \overline{j})}^{\exists\, \overline{i}\, \overline{j}.\ P(\overline{n}, \overline{j}) \wedge Q(\overline{n}, \overline{i})}(\tau) \vdash_\Sigma e\colon \tau'}\ \exists - I$$

To justify introduction of this rule consider a function definition for which type checking rejects an obviously correct type if the rule may not be used, and accepts this type if the rule is used.

```
skolem_demo(z, l) =
  let l' = insert(=, z, l) in
    let l1 = Cons(z, l') in let l2 = Cons(z, l') in diff(l1, l2)
```

where $\mathsf{diff}\colon \mathsf{L}_n(\alpha) \times \mathsf{L}_m(\alpha) \to \mathsf{L}_{\max_0(n-m)}(\alpha)$ is defined by

$$\begin{aligned}\mathsf{diff}(\mathsf{l}_1, \mathsf{l}_2) = &\mathsf{match}\ \mathsf{l}_1\ \mathsf{with}\\ &\mid \mathsf{Nil} \Rightarrow \mathsf{Nil}\\ &\mid \mathsf{Cons}(\mathsf{hd}, \mathsf{tl}) \Rightarrow \mathsf{match}\ \mathsf{l}_2\ \mathsf{with}\\ &\qquad\qquad \mid \mathsf{Nil} \Rightarrow \mathsf{l}_1\\ &\qquad\qquad \mid \mathsf{Cons}(\mathsf{hd}', \mathsf{tl}') \Rightarrow \mathsf{diff}(\mathsf{tl}, \mathsf{tl}')\end{aligned}$$

Let $e_{\mathsf{skolem\_demo}}$ abbreviate the body of the function skolem_demo. Assume that the type of diff is already accepted by the type-checker. Now we want to check $\mathsf{skolem\_demo} : \alpha \times \mathsf{L}_n(\alpha) \to \mathsf{L}_0(\alpha)$. Run the type checking procedure, first, without the $\exists - I$ rule.

1. Due to LETFUN we obtain the main goal:
   $\mathsf{z}\colon \alpha,\ \mathsf{l}\colon \mathsf{L}_n(\alpha) \vdash_\Sigma e_{\mathsf{skolem\_demo}}\colon \mathsf{L}_0(\alpha)$.
2. Apply the LET-rule. In the let-biding for l' we have the subgoal
   $\mathsf{z}\colon \alpha,\ \mathsf{l}\colon \mathsf{L}_n(\alpha) \vdash_\Sigma \mathsf{insert}(=, \mathsf{z}, \mathsf{l})\colon \tau^?$.
3. The FUNAPP-rule instantiates the unknown type $\tau^?$ with $\mathsf{L}^{i\le 1}_{n+i}(\alpha)$.
4. Consider the let-body for l'. It is a let-construct for l_1. Therefore, we apply the LET-rule for l_1. The let-binding for l_1 gives $\mathsf{z}\colon \alpha,\ \mathsf{l}'\colon \mathsf{L}^{i\le 1}_{n+i}(\alpha) \vdash_\Sigma \mathsf{Cons}(\mathsf{z}, \mathsf{l}')\colon \tau^?_1$.
5. Applying the CONS-rule instantiates the unknown type $\tau^?_1$ with $\mathsf{L}^{i\le 1}_{n+i+1}(\alpha)$. This is the type of l_1 in the next let-body.
6. Consider the let-body for l_1. We apply the LET-rule for let-construct for l_2. The let-binding for l_2 gives
   $\mathsf{z}\colon \alpha,\ \mathsf{l}_1\colon \mathsf{L}^{i\le 1}_{n+i+1}(\alpha),\ \mathsf{l}'\colon \mathsf{L}^{i\le 1}_{n+i}(\alpha) \vdash_\Sigma \mathsf{Cons}(\mathsf{z}, \mathsf{l}')\colon \tau^?_2$.
7. Applying the CONS-rule instantiates the unknown type $\tau^?_2$ with $\mathsf{L}^{i\le 1}_{n+i+1}(\alpha)$. This is the type of l_2 in the innermost let-body.
8. The subgoal for the innermost let-body is
   $\mathsf{l}_1\colon \mathsf{L}^{i\le 1}_{n+i+1}(\alpha),\ \mathsf{l}_2\colon \mathsf{L}^{i\le 1}_{n+i+1}(\alpha) \vdash_\Sigma \mathsf{diff}(\mathsf{l}_1, \mathsf{l}_2)\colon \mathsf{L}_0(\alpha)$.
9. Recall, that the scope of the index of l_1 is the size annotation of the type of l_1. The same holds for l_2, so the indices i for both types are not shared. To avoid name clash on the next steps we rename the indices:
   $\mathsf{l}_1\colon \mathsf{L}^{i_1\le 1}_{n+i_1+1}(\alpha),\ \mathsf{l}_2\colon \mathsf{L}^{i_2\le 1}_{n+i_2+1}(\alpha) \vdash_\Sigma \mathsf{diff}(\mathsf{l}_1, \mathsf{l}_2)\colon \mathsf{L}_0(\alpha)$.
10. Now, apply the FUNAPP-rule:
    $\vdash \mathsf{L}^{i_1\le 1 \wedge i_2 \le 1}_{\max_0((n+i_1+1)-(n+i_2+1))}(\alpha) \preccurlyeq \mathsf{L}_0(\alpha)$.
11. Unfold the definition of subtyping:
    $i_1 \le 1,\ i_2 \le 1 \vdash \max_0((n + i_1 + 1) - (n + i_2 + 1)) = 0$.

The last predicate does not hold, since there exist n, i_1 and i_2, such that the equation does not hold: take any n, $i_1 = 1$, $i_2 = 0$. Taking into account the semantics of the computation, we notice that the indices i_1 and i_2 should have been instantiated with the same value, since both lists l_1 and l_2 *share* the same tail l'. However, this sharing information is not available. To handle such sharing, we use the skolemising rule $\exists - I$ after step 3.

1. The same as in the previous proof.

2. The same as in the previous proof.
3. The same as in the previous proof.
4. We are at the let construct $\mathsf{let}\ \mathsf{l}_1 = ...\ \mathsf{in}\ ....$ Apply the $\exists - I$ rule:
   $n^i \leq 1,\ \mathsf{z}\colon \alpha,\ \mathsf{l}\colon \mathsf{L}_n(\alpha),\ \mathsf{l}'\colon \mathsf{L}_{n+n^i}(\alpha) \vdash_\Sigma .$
   $\mathsf{let}\ \mathsf{l}_1 = .\ \mathsf{in}\ .\colon\ \mathsf{L}_0(\alpha)$
5. Now we apply the LET-rule for $\mathsf{let}\ \mathsf{l}_1 = ...\ \mathsf{in}\ ....$ The let-binding for l_1 gives
   $n^i \leq 1;\ \mathsf{z}\colon \alpha,\ \mathsf{l}'\colon \mathsf{L}_{n+n^i}(\alpha) \vdash_\Sigma .$
   $\mathsf{Cons}(\mathsf{z}, \mathsf{l}')\colon\ \tau_1^?$
6. Applying the CONS-rule instantiates the unknown type $\tau_1^?$ with $\mathsf{L}_{n+n^i+1}(\alpha)$.
7. Consider the let-body for l_1, which is the let-construct $\mathsf{let}\ \mathsf{l}_2 = ...\ \mathsf{in}\ ....$ Again, we apply the LET-rule. The let-binding for l_2 gives
   $n^i \leq 1;\ \mathsf{z}\colon \alpha,\ \mathsf{l}_1\colon \mathsf{L}_{n+n^i+1}(\alpha),\ \mathsf{l}'\colon \mathsf{L}_{n+n^i}(\alpha) \vdash_\Sigma .$
   $\mathsf{Cons}(\mathsf{z}, \mathsf{l}')\colon\ \tau_2^?$
8. Applying the CONS-rule instantiates the unknown type $\tau_2^?$ with $\mathsf{L}_{n+n^i+1}(\alpha)$.
9. The subgoal for the innermost let-body is
   $n^i \leq 1;\ \mathsf{l}_1\colon \mathsf{L}_{n+n^i+1}(\alpha), \mathsf{l}_2\colon \mathsf{L}_{n+n^i+1}(\alpha) \vdash_\Sigma \mathsf{diff}(\mathsf{l}_1, \mathsf{l}_2)\colon \mathsf{L}_0(\alpha).$
10. Now, apply the FUNAPP-rule:
    $n^i \leq 1 \vdash \mathsf{L}_{\max_0((n+n^i+1)-(n+n^i+1))}(\alpha) \preccurlyeq \mathsf{L}_0(\alpha).$
11. Unfold the definition of subtyping:
    $n^i \leq 1 \vdash \max_0((n + n^i + 1) - (n + n^i + 1)) = 0.$
    This is trivially true.

Note that on a given context Γ, the rule $\exists - I$ can be applied up to $K|dom(\Gamma)|$ times, where $|dom(\Gamma)|$ is the number of zero-order variables and K is the maximal "nestedness" of list types in the context Γ.

Input lists of lists: variable length of internal lists The FUNNAPP-rule we have given in Section 3.1 is sound for functions over matrix-like structures (of the types $\mathsf{L}_n(\alpha)$, $\mathsf{L}_n(\mathsf{L}_m(\alpha))$, etc.). That rule is a partial case of the general typing rule that is sound for functions with arbitrary first-order annotations (including constants and presentations of lists of lists with different lengths) in their signatures:

$$\frac{\begin{array}{c}\Sigma(f) = \tau_1^f \times \ldots \times \tau_{k'}^f \times \tau_1' \times \ldots \times \tau_k' \to \tau_0 \\ \text{the underlying type of } \mathsf{g}_{l'} \text{ is an instance} \\ \text{of the underlying type of } \tau_l^f, \text{where } 1 \leq l' \leq k' \\ D \vdash \tau_l \preccurlyeq \sigma(\tau_l'),\ \text{where } 1 \leq l \leq k \\ D \vdash \sigma(\tau_0) \preccurlyeq \tau \end{array}}{\left.\begin{array}{l} D;\ \Gamma, \\ \mathsf{z}_1\colon \tau_1, \ldots, \mathsf{z}_k\colon \tau_k \end{array}\right\} \vdash_\Sigma \mathsf{f}(\mathsf{g}_1, \ldots, \mathsf{g}_{k'},\ \mathsf{z}_1, \ldots, \mathsf{z}_k)\colon \tau}\ \text{FUNAPP}$$

where σ is an instantiation of the type and size variables of the types of formal parameters with types and size annotations respectively. Answering the question if such an instantiation, limited to size variables, exists and constructing it, if "yes", amounts to instantiation of existential quantifiers for a finite number of (size) variables in a first-order arithmetic formula. We will see it in the example

concat, where we type-check the annotations for a function over a list of lists with variate lengths.

In general, constructing such instantiations is undecidable in integers. It is decidable, in integers, in the class of functions with linear size annotations. Moreover, for a simple non-linear annotations it may be constructed automatically as well. Type-checking involving functions with complex annotations needs interaction with a user.

Consider as an example type-checking for the function concat:

$$\begin{array}{l}\mathsf{concat(l)} = \mathsf{match\ l\ with}\\ \qquad\qquad \mid \mathsf{Nil} \Rightarrow \mathsf{Nil}\\ \qquad\qquad \mid \mathsf{Cons(hd,\ tl)} \Rightarrow \mathsf{let\ l' = concat(tl)}\\ \qquad\qquad\qquad\qquad\qquad\qquad \mathsf{in\ append(hd,\ l')}\end{array}$$

We want to type-check the type $\mathsf{L}_n(\mathsf{L}_i^{i\le m}(\alpha)) \to \mathsf{L}_j^{j\le mn}(\alpha)$.

1. The main goal is $\mathsf{l}\colon \mathsf{L}_n(\mathsf{L}_i^{i\le m}(\alpha)) \vdash_\Sigma e_{\mathsf{concat}}\colon \mathsf{L}_j^{i\le mn}(\alpha)$.
2. Apply the MATCH-rule. The nil-branch yields the subgoal:
   $n = 0;\ \mathsf{l}\colon \mathsf{L}_n(\mathsf{L}_i^{i\le m}(\alpha)) \vdash_\Sigma \mathsf{Nil}\colon \mathsf{L}_j^{j\le mn}(\alpha)$.
3. We follow the nil-branch. Applying the NIL-rule we obtain a subgoal: $n = 0 \vdash \mathsf{L}_j^{j\le mn}(\alpha) \preccurlyeq \mathsf{L}_0(\alpha)$.
4. Unfold the definition of subtyping and obtain
   $n = 0, 0 \le j \le mn \vdash j = 0$, which is trivially true.
5. In the cons-branch one has to prove the subgoal
   $n \ge 1;\ \mathsf{hd}\colon \mathsf{L}_i^{i\le m}(\alpha),\ \mathsf{tl}\colon \mathsf{L}_{n-1}(\mathsf{L}_i^{i\le m}(\alpha)) \vdash_\Sigma$
   $\mathsf{let\ l'} = .\ \mathsf{in}\ .\colon\ \mathsf{L}_j^{j\le mn}(\alpha)$
6. Apply the LET-rule. The binding yields the subgoal
   $n \ge 1;\ \mathsf{tl}\colon \mathsf{L}_{n-1}(\mathsf{L}_i^{i\le m}(\alpha)) \vdash_\Sigma \mathsf{concat(tl)}\colon \tau^?$.
7. Apply the FUNAPP-rule and obtain the predicates
   $n \ge 1 \vdash \mathsf{L}_{n-1}(\mathsf{L}_i^{i\le m}(\alpha)) \preccurlyeq \sigma(\mathsf{L}_{n'}(\mathsf{L}_{i'}^{i'\le m'}(\alpha')))$ and
   $n \ge 1 \vdash \sigma(\mathsf{L}_{i'}^{i'\le n'm'}(\alpha')) \preccurlyeq \tau^?$.
   We want to find an instantiation σ, for which these subtypings hold, and to reconstruct $\tau^?$.
   To find the instantiations $\sigma(n')$ and $\sigma(m')$ means to find such n'_σ and m'_σ that yield the subtypings
   $n \ge 1 \vdash \mathsf{L}_{n-1}(\mathsf{L}_i^{i\le m}(\alpha)) \preccurlyeq \mathsf{L}_{n'_\sigma}(\mathsf{L}_{i'}^{i'\le m'_{\sigma'}}(\alpha'))$ and
   $n \ge 1 \vdash \mathsf{L}_{i'}^{i'\le n'_\sigma m'_\sigma}(\alpha) \preccurlyeq \tau^?$.
   Unfolding the definition of subtyping for the arguments we obtain the formula
   $\exists\ n'_\sigma\ m'_\sigma.\ n - 1 = n'_\sigma \wedge (i \le m \to \exists\ i'.\ i = i' \wedge i' \le m'_{\sigma'})$
   This is a simple linear predicate. It is easy to see, that we may take $n'_\sigma = n-1$ and $m'_\sigma := m$. This means that we instantiate $\tau^?$ with $\sigma(\mathsf{L}_j^{j\le mn}(\alpha))$, which is equal to $\mathsf{L}_j^{j\le m(n-1)}(\alpha)$.
   In this simple case n_σ and m'_σ may be found automatically. In complex cases a type-checker may fail to find instantiations of the formal size variables in the function call. Then they should be provided by a user. It can be

done as follows. The type-checker asks the user to give the types of the actual parameters in the call according to the pattern given by the formal parameters in the function signature. That is, the user has to instantiate the formal size variables with size expressions. These size expressions are given in terms of the sizes of program variables in the calling context.
For instance, in the example above the type-checker gives the user the type of the formal parameter $\mathsf{L}_{n'}(\mathsf{L}_{i'}^{i'\leq m'}(\alpha))$ and asks to instantiate n' and m'. The user's answer is $n' = \mathit{size}(\mathsf{tl})$ and $m' = \mathit{uppersize}(\mathsf{hd})$.

8. The subgoal in the let-body is

$$\begin{array}{l} n \geq 1;\ \mathsf{hd}\colon \mathsf{L}_{i}^{i\leq m}(\alpha),\ \mathsf{l}'\colon \mathsf{L}_{j'}^{j'\leq m(n-1)}(\alpha) \vdash_{\Sigma} \\ \mathsf{append}(\mathsf{hd},\mathsf{l}')\colon\ \mathsf{L}_{j}^{j\leq mn}(\alpha) \end{array}$$

9. Applying the FunnApp-rule (its "light version" for non-nested lists and matrix-like structures) for append yields the subtyping $n \geq 1 \vdash \mathsf{L}_{i+j'}^{i\leq m\wedge j'\leq m(n-1)}(\alpha) \preccurlyeq \mathsf{L}_{j}^{j\leq mn}(\alpha)$.
10. Unfolding the definition of subtying gives $n \geq 1,\ i \leq m,\ j' \leq m(n-1)\ \vdash\ \exists\, j.\ j \leq mn \wedge i + j' = j$. It is easy to see that $j := i + j' \leq m + m(n-1) = mn$.

3.2 Semantics of typing judgments (soundness)

The set-theoretic semantics of typing judgments is formalised later in this section as the soundness theorem.

First we define the semantics w of a program value v, with respect to a specific heap h and a *ground type* $\tau^{\bullet}$. It is a set-theoretic interpretation given by the four-place relation $v \models^{h}_{\tau^{\bullet}} w$. Integer and boolean constants interpret themselves, and locations are interpreted as *non-cyclic lists*:

$$\begin{array}{lll} c & \models^{h}_{\mathsf{Int}\cup\mathsf{Bool}} & c \\ \mathtt{NULL} & \models^{h}_{\mathsf{L}^{\exists\,\bar{i}.\,Q(\bar{i})}_{p(\bar{i})}(\tau^{\bullet})} & \mathtt{[]}\ \ \textit{iff}\ \ \exists\,\bar{i}.\,Q(\bar{i}) \wedge p(\bar{i}) = 0 \\ \ell & \models^{h}_{\mathsf{L}^{\exists\,\bar{i}.\,Q(\bar{i})}_{p(\bar{i})}(\tau^{\bullet})} & w_{\mathsf{hd}} :: w_{\mathsf{tl}}\ \textit{iff}\ \ \ell \in \mathit{dom}(h) \wedge \\ & & \qquad h.\ell.\mathsf{hd} \models^{h|_{\mathit{dom}(h)\setminus\{\ell\}}}_{\tau^{\bullet}} w_{\mathsf{hd}} \wedge \\ & & \qquad h.\ell.\mathsf{tl} \models^{h|_{\mathit{dom}(h)\setminus\{\ell\}}}_{\mathsf{L}^{\exists\,\bar{i}.\,Q(\bar{i})}_{p(\bar{i})-1}(\tau^{\bullet})} w_{\mathsf{tl}} \end{array}$$

where $h|_{\mathit{dom}(h)\setminus\{\ell\}}$ denotes the heap equal to h everywhere except in ℓ, where it is undefined. Presence of the restriction $h|_{\mathit{dom}(h)\setminus\{\ell\}}$ instead of the full heap h in the recursive part of the definition above ensures that lists are non-cyclic.

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:

$$\begin{array}{l} \mathit{length}\colon \mathit{Heap} \rightharpoonup \mathit{Address} \rightharpoonup \mathcal{N} \\ \mathit{length}_h(\mathtt{NULL}) = 0 \\ \mathit{length}_h(\ell) = 1 + \mathit{length}_{h|_{\mathit{dom}(h)\setminus\{\ell\}}}(h.\ell.\mathsf{tl}) \end{array}$$

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

Lemma 31 (Consistency of the model relation) *Let an address* `adr` *be either an arbitrary location* ℓ *or the null address* `NULL`*. The relation* $\mathtt{adr} \models^h_{\mathsf{L}^{\exists\, \bar{i}.\ Q(\bar{i})}_{p(\bar{i})}(\tau^\bullet)} w$ *implies that there exists* $\bar{i}$ *such that* $Q(\bar{i})$ *holds and* $p(\bar{i}) = \mathit{length}_h(\mathtt{adr})$*.*

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

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 context $\Gamma^\bullet$, in which all the types are ground (a *ground context*):

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

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:

$$\begin{array}{lll}
\mathit{Valuation} & \epsilon : & \mathit{SizeVariables} \to \mathcal{R} \\
\mathit{Instantiation} & \eta : & \mathit{TypeVariables} \to \tau^\bullet
\end{array}$$

Valuations and instantiations extend to types:

$$\begin{aligned}
&\epsilon(\mathsf{L}^{\exists\, \bar{i}.\ Q(n_1,\ldots n_k,\ \bar{i})}_{p(n_1,\ldots n_k,\ \bar{i})}(\tau)) = \mathsf{L}^{\exists\, \bar{i}.\ Q(\epsilon(n_1),\ldots,\epsilon(n_k),\ \bar{i})}_{p(\epsilon(n_1),\ldots,\epsilon(n_k),\ \bar{i})}(\eta(\tau)) \\
&\eta(\mathsf{L}^{\exists\, \bar{i}.\ Q(\overline{n},\bar{i})}_{p(\overline{n},\bar{i})}(\tau)) = \mathsf{L}^{\exists\, \bar{i}.\ Q(\overline{n},\bar{i})}_{p(\overline{n},\bar{i})}(\eta(\tau))
\end{aligned}$$

Now, stating the soundness theorem is straightforward:

Theorem 32 (Soundness) *For any store* s*, heaps* h *and* h'*, closure* $\mathcal{C}$*, expression* e*, value* v*, context* Γ *with size variables* $\overline{n}$*, quantifier-free formula* D*, signature* Σ*, type* τ*, size valuation* ϵ*, and type instantiation* η *such that*

- $\mathit{dom}(s) = \mathit{dom}(\Gamma)$,
- $s;\ h;\ \mathcal{C} \vdash e \rightsquigarrow v;\ h'$,
- $D;\ \Gamma \vdash_\Sigma e : \tau$ *is a node in the derivation tree for some function body,*

the following implication holds:

$$\begin{aligned}
&\forall\, \eta, \epsilon[\ D(\epsilon(\overline{n})) \ \wedge\ \mathit{Valid}_{\mathsf{store}}(\mathit{dom}(s), \eta(\epsilon(\Gamma)), s, h) \implies \\
&\mathit{Valid}_{\mathsf{val}}(v, \eta(\epsilon(\tau)), h')\ ]
\end{aligned}$$

Proof. The proof is done by induction on the size of the derivation tree for the operational-semantics judgement. For the sake of convenience we will denote $D(\epsilon(\overline{n}))$ 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; $\Gamma \vdash_\Sigma e : \tau$. Fix a valuation $\epsilon : SV(\Gamma \cup D \cup \tau) \to \mathcal{R}$, and a type instantiation $\eta : TV(\Gamma \cup \tau) \to \tau^\bullet$ such that the assumptions of the lemma hold. We must show that $\mathit{Valid}_{\mathsf{val}}(v, \tau_{\eta\epsilon}, h')$ holds.

The axioms are sound due to the lemma stating that subtyping preserves the model relation (see the appendix).

The $\exists - I$ rule and the MATCH rule are sound because of Lemma 31 (consistency).

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

Now apply the induction hypothesis to the second branch to get

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

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

Therefore, using the induction assumption above we obtain $\mathit{Valid}_{\mathsf{val}}(v, \tau_{\eta\epsilon}, h')$.

3.3 Annotations for principal types

Intuitively, the annotations in the principal type of an expression e (respectively, a function $\mathtt{f}$ defined by e), define the most precise size dependency for e (respectively $\mathtt{f}$). For instance, for the function insert the type $(\alpha \times \alpha \to \mathsf{Bool}) \times \alpha \times \mathsf{L}_n(\alpha) \to \mathsf{L}^{i \leq \mathbf{2}}_{n+i}(\alpha)$ passes type checking. However, it is less accurate than the one with predicate $i \leq 1$, and thus it is not principal.

Formally, τ is the principal type of e w.r.t. D, Γ and Σ, if the underlying type (i.e. without annotations) of τ is principal, D; $\Gamma \vdash_\Sigma e : \tau$ and for all types τ' such that D; $\Gamma \vdash_\Sigma e : \tau'$, $D \vdash \tau \preccurlyeq \tau'$ holds.

In our setting there are functions with polynomially bounded size dependencies that do not have a principal type. This is the case for the function $\mathsf{log2(l)}$:

```
log2(l) =
      match l with
      | Nil ⇒ Nil
      | Cons(hd, tl) ⇒match tl with
                      | Nil ⇒ Nil
                      | Cons(hd′, tl′) ⇒ let z = divtwo(l) in
                                         if hd = 0 then log2(z)
                                         else Cons(hd, log2(z))
```

where divtwo: $\mathsf{L}_n(\alpha) \to \mathsf{L}^{i\leq 1}_{\max_0(n-i)/2}(\alpha)$, takes a list of length n and returns a list of length $\lfloor n/2 \rfloor$:

```
divtwo(l) =
  match l with
  | Nil ⇒ Nil
  | Cons(hd, tl) ⇒match tl with
                  | Nil ⇒ Nil
                  | Cons(hd′, tl′) ⇒ Cons(hd′, divtwo(tl′))
```

The principal type of log2 would be $\mathsf{L}_n(\mathsf{Int}) \to \mathsf{L}_i^{i\leq\lfloor\log_2(n)\rfloor}(\mathsf{Int})$, but this type is not expressible in our type system. We can approximate it from above with the linear piecewise functions f_k, $k \geq 1$:

$$f_k(n) = \begin{cases} 0 & \text{if } n = 0 \\ l & \text{if } 2^l \leq n < 2^{l+1}, \text{where } 0 \leq l \leq k-1 \\ \dfrac{k}{2^k}n & \text{if } n \geq 2^k \end{cases}$$

It is easy to see that, the bigger k is, the better f_k approximates $\lfloor\log_2\rfloor$ since both functions coincide on more points. One can show that for any $k \geq 1$, the type $\mathsf{L}_n(\mathsf{Int}) \to \mathsf{L}_i^{i\leq f_k(n)}(\mathsf{Int})$ is an admissible type for log2 (see the appendix).

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 single non-piecewise polynomials in the class of functions over non-nested lists and matrix-like structures (see [19]). 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 the possibility to annotate types with piecewise polynomials, in particularly having the $\max_0$ operation, makes type checking undecidable, even for programs satisfying *no-let-before-match*. This can be proven by showing that for any polynomial $p(n_1, \ldots, n_k)$ that maps naturals to naturals, there is a *total* function f_p of type $\mathsf{L}_{n_1}(\mathsf{Int}) \times \ldots \times \mathsf{L}_{n_k}(\mathsf{Int}) \to \mathsf{L}_{\max_0(1-p^2(n_1,\ldots,n_k))}(\mathsf{Int})$. Checking whether this function has the type $\mathsf{L}_{n_1}(\mathsf{Int}) \times \ldots \times \mathsf{L}_{n_k}(\mathsf{Int}) \to \mathsf{L}_0(\mathsf{Int})$

amounts to answering whether p has natural roots or not. If p has natural roots, i.e. there exists $n'_1, \ldots n'_k$ such that $p(n'_1, \ldots n'_k) = 0$, then $\max_0(1 - p^2(n'_1, \ldots n'_k)) = 1$ and the type is rejected. Otherwise, $p^2(n_1, \ldots, n_k) \geq 1$ for all naturals $n_1, \ldots, n_k$, and thus $\max_0(1 - p^2(n_1, \ldots n_k)) = 0$ so the type is accepted. Hence, type checking reduces to solving 10^{th} Hilbert problem, which is known to be undecidable [15].

The function f_p can be defined by $\mathsf{f}_p(\mathsf{l}_1, \ldots, \mathsf{l}_k) = \mathsf{diff}([1], \mathsf{p}^2(\mathsf{l}_1, \ldots, \mathsf{l}_k))$, where $\mathsf{diff} : \mathsf{L}_n(\alpha) \times \mathsf{L}_m(\alpha) \to \mathsf{L}_{\max_0(n-m)}(\alpha)$ and $\mathsf{p}^2 : \mathsf{L}_{n_1}(\alpha) \times \ldots \times \mathsf{L}_{n_k}(\alpha) \to \mathsf{L}_{p^2(n_1,\ldots,n_k)}(\alpha)$. The defition of $\mathsf{diff}(\mathsf{l}_1, \mathsf{l}_2)$ was given in Section 3.1. The function p^2 has been defined by us in [19]. Here, we recapitulate its construction. First, we note that any polynomial $p(n_1, \ldots, n_k)$ can be written as a difference $p_1 - p_2$ of two polynomials with non-negative coefficients. For these p_i one constructs function definitions $\mathsf{p_i}$ whose types are $\mathsf{L}_{n_1}(\alpha) \times \ldots \times \mathsf{L}_{n_k}(\alpha) \to \mathsf{L}_{p_i(n_1,\ldots,n_k)}(\alpha)$, with $i = 1, 2$, respectively. This is done by compositions of $\mathsf{append} : \mathsf{L}_n(\alpha) \times \mathsf{L}_m(\alpha) \to \mathsf{L}_{n+m}(\alpha)$ and $\mathsf{copyfirst} : \mathsf{L}_n(\alpha) \times \mathsf{L}_m(\alpha) \to \mathsf{L}_{nm}(\alpha)$, which copies the first argument m times. Now, we define $\mathsf{p}^2(\mathsf{l}_1, \ldots, \mathsf{l}_k)$ as $\mathsf{sqdiff}(\mathsf{p_1}(\mathsf{l}_1, \ldots, \mathsf{l}_k),\ \mathsf{p_2}(\mathsf{l}_1, \ldots, \mathsf{l}_k))$, where the function $\mathsf{sqdiff} : \mathsf{L}_n(\alpha) \times \mathsf{L}_m(\alpha) \to \mathsf{L}_{(n-m)^2}(\alpha)$ is given by

$$
\begin{array}{l}
\mathsf{match}\ \mathsf{l}_1\ \mathsf{with} \mid \mathsf{Nil} \Rightarrow \mathsf{copyfirst}(\mathsf{l}_2,\ \mathsf{l}_2) \\
\qquad\qquad\qquad \mid \mathsf{Cons}(\mathsf{hd},\ \mathsf{tl}) \Rightarrow \mathsf{match}\ \mathsf{l}_2\ \mathsf{with} \mid \mathsf{Nil} \Rightarrow \mathsf{copyfirst}(\mathsf{l}_1,\ \mathsf{l}_1) \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \mid \mathsf{Cons}(\mathsf{hd}',\ \mathsf{tl}') \Rightarrow \mathsf{sqdiff}(\mathsf{tl},\ \mathsf{tl}')
\end{array}
$$

Note that these functions satisfy *no-let-before-match*.

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 families of non-piecewise polynomials* and function definitions subject to *no-let-before-match*; input types are matrix-like structures. We believe that decidability will hold in the class of all functions annotated with finite families of polynomials. Checking this hypothesis is a topic of our future research.

As an example of the second class of types, consider the function $\mathsf{insert}\colon (\alpha \times \alpha \to \mathsf{Bool}) \times \alpha \times \mathsf{L}_n(\alpha) \to \mathsf{L}^{i\leq 1}_{n+i}(\alpha)$, defined in Section 3. Its size dependency is given by a family of two polynomials: n and $n+1$.The predicates determined by the cons-branch are $n \geq 1 \vdash \exists\, i.\, 0 \leq i \leq 1 \wedge n = n+i$ (in the true-subbranch) and $n \geq 1,\ 0 \leq j \leq 1 \vdash \exists\, i.\, 0 \leq i \leq 1 \wedge (n-1)+j+1 = n+i$ (in the false-subbranch). Trivially, in the true-subbranch we have $i = 0$. In the false-subbranch the index j appears due to the recursive call. For each j from the finite set $\{0, 1\}$ we search for i in the finite set $\{0, 1\}$, such that the corresponding equation holds. If $j = 0$ then $n = n + i$ and $i = 0$. If $j = 1$ then $n + 1 = n + i$ and $i = 1$.

In general, type checking types annotated with finite families of non-piecewise polynomials is similar to checking types annotated with a single polynomial. It

reduces to checking entailments of the form $\forall\, \overline{j} \in N.\, \exists\, \overline{i} \in M.\, p(\overline{n}, \overline{j}) = p'(\overline{n}, \overline{i})$, where the sets N and M are finite and $\overline{n} = (n_1, \ldots, n_k)$ is such that $n_i \geq c_i$ for some integer constant $c_i \geq 0$. One must instantiate $\overline{i} \in M$ for each $\overline{j} \in N$. Given $\overline{j}_0 \in N$, compute a polynomial $p(\overline{n}, \overline{j}_0)$ that becomes a polynomial w.r.t. $\overline{n}$. By finite search in M, check if there is $\overline{i}_0 \in M$ such that $p(\overline{n}, \overline{j}_0) = p'(\overline{n}, \overline{i}_0)$ as polynomials in $\overline{n}$ (i.e. the corresponding coefficients are equal). If not, reject the type. Otherwise, continue with another $\overline{j}_0 \in N$, 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 $\mathsf{copyfirst}(\mathsf{l}, \mathsf{tail}(\mathsf{l}))$, where $\mathsf{tail} : \mathsf{L}_n(\alpha) \to \mathsf{L}_{\max_0(n-1)}(\alpha)$ returns the tail of a list l if it is not empty, and Nil otherwise. The type of the composition is $\mathsf{L}_n(\alpha) \to \mathsf{L}_{n \max_0(n-1)}(\alpha)$. It is possible to construct an integer type checker that accepts the type $\mathsf{L}_n(\alpha) \to \mathsf{L}_{n(n-1)}(\alpha)$, however, a real-arithmetic based 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, the problem of finding whether an unbounded area contains naturals is unsolvable in general. A similar search is done by the *Mathematica* software package, when it applies *cylindrical algebraic decomposition* to solve systems of Diophantine equations.

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, but the approach seems to be impractical in the general case.

5 Related Work

This research continues our work on non-monotonic polynomial size dependencies [19, 22, 20]. The dependencies of our previous work covered exact sizes only. These earlier exact dependencies correspond in the current, more general, work to one-element families of polynomials (without the $\max_0$ operation).

Amortisation [16] 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 (Hofmann and Jost [12]). Brian Campbell [9] extended this approach to

infer bounds on *stack* space usage. The *AHA* project [21] aims to adapt amortisation techniques for *non-linear* bounds within functional programs and transfer the results to the object-oriented programming.

Our approach is close to size analysis with polynomial quasi-interpretations [8, 4]. There a program is interpreted as a monotonic polynomial extended with the max operation. For instance, Cons(hd, 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.

Resource analysis may be performed within a *Proof Carrying Code* (PCC) framework. The Mobile Resource Guarantees (MRG) project [17] developed a PCC infrastructure for certifying resource bounds of mobile code. Key components of this infrastructure are a certifying compiler for *Camelot*, an ML like resource-safe language; a hierarchy of program logics tailored for reasoning about resource consumption, and an embedding of the logics into a theorem prover. However, their type system assumes *linear* consumption of resources. In [5] 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.

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*.

Sized types can also be used to prove termination or to characterise the complexity of recursive functions (see e.g. [1] for an overview). One of the first type systems used to prove termination was introduced by J. Hughes et al. in [13]. The system of Chin and Khoo [10], annotates every type with size annotations and infers a formula of Presburger arithmetic that guarantees termination. In [7], G. Barthe et al. introduce a sized-type system that uses sized products instead of existential quantification, and they are able to type quicksort as a non-size increasing function.

Several papers have studied programming languages with *implicit computational complexity* properties [11, 6]. 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 PSPACE.

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.

In [3] 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 [2], 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 which allows to express indexed families of output-on-input size dependencies of non-monotonic piecewise polynomials.

Compared to earlier work this feature greatly increases the applicability of our size analysis, which was limited to exact sizes. The extra expressibility will form the basis for design choices of a practical implementation.

The innovation comes at a cost: we have crossed the border of decidability. However, we identify large decidable subclasses. Furthermore, the type system admits semi-automatic type inference.

Our next step will be to try to prove our decidability hypothesis about finite families for non matrix-like structures and to assess the quality of the bounds inferred by the inference procedure. Then, we plan to extend the class of data types to ordinary inductive types (based on [20]) and 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, we seem to be ready now to transfer our size analysis results to the world of imperative programs. Of course, there will be problems to solve: like e.g. incorporating object-orientation with shared substructures.

References

1. A. Abel. *A Polymorphic Lambda-Calculus with Sized Higher-Order Types.* PhD thesis, LFE Theoretische Informatik, Ludwig-Maximilians-Universität München, 2006.
2. E. Albert, P. Arenas, S. Genaim, and G. Puebla. Automatic Inference of Upper Bounds for Recurrence Relations in Cost Analysis. In *Static Analysis,* 15^{th} *International Symposium*, volume 5079 of *LNCS*, pages 221–237, 2008.
3. E. Albert, P. Arenas, S. Genaim, G. Puebla, and D. Zanardini. Cost Analysis of Java Bytecode. In 16^{th} *European Symposium on Programming, ESOP'07*, volume 4421 of *LNCS*, pages 157–172. Springer, 2007.
4. R. M. Amadio. Synthesis of max-plus quasi-interpretations. *Fundamenta Informaticae*, 65(1-2):29–60, 2004.
5. 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.
6. 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.

7. G. Barthe, B. Grégoire, and C. Riba. Type-based termination with sized products. In *17th EACSL Annual Conference on Computer Science Logic, 15th-19th Sep. 2008, Bertinoro, Italy*, Lecture Notes in Computer Science. Springer, 2008.
8. G. Bonfante, J.-Y. Marion, and J.-Y. Moyen. Quasi-interpretations, a way to control resources. *Theoretical Computer Science*, 2005.
9. B. Campbell. *Space Cost Analysis Using Sized Types*. PhD thesis, School of Informatics, University of Edinburgh, 2008.
10. W.-N. Chin and S.-C. Khoo. Calculating sized types. *Higher Order Symbol. Comput.*, 14(2-3):261–300, 2001.
11. M. Gaboardi, J.-Y. Marion, and S. Ronchi Della Rocca. A logical account of PSPACE. In 35^{th} *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages POPL 2008, San Francisco, January 10-12, 2008, Proceedings*, pages 121–131, 2008.
12. M. Hofmann and S. Jost. Static prediction of heap space usage for first-order functional programs. *SIGPLAN Not.*, 38(1):185–197, 2003.
13. J. Hughes, L. Pareto, and A. Sabry. Proving the correctness of reactive systems using sized types. In *POPL '96: Proceedings of the* 23^{rd} *ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 410–423, New York, NY, USA, 1996. ACM.
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. J. P. Jones and Y. V. Matijasevič. Proof of recursive unsolvability of Hilbert's tenth problem. *American Mathematical Monthly*, 98(10):689–709, 1991.
16. C. Okasaki. *Purely Functional Data Structures*. Cambridge University Press, 1998.
17. D. Sannella, M. Hofmann, D. Aspinall, S. Gilmore, I. Stark, L. Beringer, H.-W. Loidl, K. MacKenzie, A. Momigliano, and O. Shkaravska. Mobile resource guarantees (project evaluation paper). In M. C. J. D. van Eekelen, editor, *Trends in Functional Programming*, pages 211–226, 2005.
18. O. Shkaravska, M. van Eekelen, and A. Tamalet. Collected Size Semantics for Functional Programs over Lists. In 20^{th} *International Symposium on the Implementation and Application of Functional Languages (IFL'2008)*, LNCS. Springer-Verlag, 2008. to appear.
19. 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)*, volume 4583 of *LNCS*, pages 351–366. Springer, 2007.
20. A. Tamalet, O. Shkaravska, and M. van Eekelen. Size Analysis of Algebraic Data Types. In M. Morazán, editor, *Selected Papers of the* 9^{th} *Internat. Symp. on Trends in Functional Programming (TFP'08)*. Intellect Publishers. 2008, to appear.
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* 8^{th} *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* 16^{th} *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.

7 Examples of type-checking

$\mathsf{insert}(\mathsf{g},\,\mathsf{z},\mathsf{l}) =$
$\mathsf{match\ l\ with}\ |\ \mathsf{Nil} \Rightarrow \mathsf{let\ l'} = \mathsf{Nil\ in\ Cons}(\mathsf{z},\,\mathsf{l'})$
$\qquad\qquad\qquad\quad |\ \mathsf{Cons}(\mathsf{hd},\,\mathsf{tl}) \Rightarrow \mathsf{if\ g}(\mathsf{z},\,\mathsf{hd})\ \mathsf{then\ l\ else\ let\ l'} = \mathsf{insert}(\mathsf{g},\,\mathsf{z},\mathsf{tl})\ \mathsf{in\ Cons}(\mathsf{hd},\,\mathsf{l'})$

Show that

$$\mathsf{insert} : (\alpha \times \alpha \to \mathsf{Bool}) \times \alpha \times \mathsf{L}_n(\alpha) \to \mathsf{L}_{n+i}^{i\leq 1}(\alpha)$$

Step	The subgoal after the step
main goal	$\mathsf{z}\colon \alpha,\ \mathsf{l}\colon \mathsf{L}_n(\alpha) \vdash_\Sigma e_{\mathsf{insert}} \colon \mathsf{L}_{n+i}^{i\leq 1}(\alpha)$
MatchNil	$n = 0,\ \mathsf{z}\colon \alpha,\ \mathsf{l}\colon \mathsf{L}_n(\alpha) \vdash_\Sigma \mathsf{let\ l'} = \mathsf{Nil\ in\ Cons}(\mathsf{z},\,\mathsf{l'}) \colon \mathsf{L}_{n+i}^{i\leq 1}(\alpha)$
Let-bind	$n = 0,\ \mathsf{z}\colon \alpha,\ \mathsf{l}\colon \mathsf{L}_n(\alpha) \vdash_\Sigma \mathsf{Nil} \colon \tau^?$
Nil	$\tau^? := \mathsf{L}_0(\alpha)$
Let-body	$n = 0,\ \mathsf{z}\colon \alpha,\ \mathsf{l}\colon \mathsf{L}_n(\alpha),\ \mathsf{l'}\colon \mathsf{L}_0(\alpha) \vdash_\Sigma \mathsf{Cons}(\mathsf{z},\,\mathsf{l'}) \colon \mathsf{L}_{n+i}^{i\leq 1}(\alpha)$
Cons	$n = 0 \vdash i^? \leq 1 \wedge 1 + 0 = n + i^?$
Subst	$n = 0 \vdash i^? \leq 1 \wedge i^? = 1$
	$i^? := 1$
MatchCons	$n \geq 1,\ \mathsf{z}\colon \alpha,\ \mathsf{l}\colon \mathsf{L}_n(\alpha) \vdash_\Sigma \mathsf{if}\ -\ \mathsf{then}\ -\ \mathsf{else}\ - \colon \mathsf{L}_{n+i}^{i\leq 1}(\alpha)$
IfTrue	$n \geq 1,\ \mathsf{l}\colon \mathsf{L}_n(\alpha) \vdash_\Sigma \mathsf{l}\colon \mathsf{L}_{n+i}^{i\leq 1}(\alpha)$
Var	$n \geq 1 \vdash i^? \leq 1\ \wedge\ n = n + i^?$
	$i^? := 0$
IfFalse	$n \geq 1,\ \mathsf{z}\colon \alpha,\ \mathsf{hd}\colon \alpha,\ \mathsf{tl}\colon \mathsf{L}_{n-1}(\alpha) \vdash_\Sigma \mathsf{let\ l'} = \mathsf{insert}(\mathsf{g},\,\mathsf{z},\,\mathsf{tl})\ \mathsf{in}\ - \colon \mathsf{L}_{n+i}^{i\leq 1}(\alpha)$
Let-bind	$n \geq 1,\ \mathsf{z}\colon \alpha,\ \mathsf{tl}\colon \mathsf{L}_{n-1}(\alpha) \vdash_\Sigma \mathsf{insert}(\mathsf{g},\,\mathsf{z},\,\mathsf{tl}) \colon \tau^?$
Fun	$\tau^? := \mathsf{L}_{(n-1)+j}^{j\leq 1}(\alpha)$
Let-body	$n \geq 1,\ \mathsf{hd}\colon \alpha,\ \mathsf{l'}\colon \mathsf{L}_{(n-1)+j}^{j\leq 1}(\alpha) \vdash_\Sigma \mathsf{Cons}(\mathsf{hd},\,\mathsf{l'}) \colon \mathsf{L}_{n+i}^{i\leq 1}(\alpha)$
Cons	$n \geq 1,\ j \leq 1 \vdash i^? \leq 1 \wedge (n-1) + j + 1 = n + i^?$
simp	$n \geq 1,\ j \leq 1 \vdash i^? \leq 1 \wedge i^? = j$

$\mathsf{skolem\_demo}(\mathsf{z}, \mathsf{l}) =$
$\mathsf{let}\ \mathsf{l}' = \mathsf{insert}(=, \mathsf{z}, \mathsf{l})\ \mathsf{in}\ \mathsf{let}\ \mathsf{l}_1 = \mathsf{Cons}(\mathsf{z},\ \mathsf{l}')\ \mathsf{in}\ \mathsf{let}\ \mathsf{l}_2 = \mathsf{Cons}(\mathsf{z},\ \mathsf{l}')\ \mathsf{in}\ \mathsf{diff}(\mathsf{l}_1, \mathsf{l}_2)$

$\mathsf{where}\ \mathsf{diff}(\mathsf{l}_1, \mathsf{l}_2) =$
$\mathsf{match}\ \mathsf{l}_1\ \mathsf{with} \mid \mathsf{Nil} \Rightarrow \mathsf{Nil}$
$\mid \mathsf{Cons}(\mathsf{hd},\ \mathsf{tl}) \Rightarrow \mathsf{match}\ \mathsf{l}_2\ \mathsf{with} \mid \mathsf{Nil} \Rightarrow \mathsf{l}_1$
$\mid \mathsf{Cons}(\mathsf{hd}',\ \mathsf{tl}') \Rightarrow \mathsf{diff}(\mathsf{tl}, \mathsf{tl}')$

It is easy to check that $\mathsf{diff} : \mathsf{L}_n(\alpha) \times \mathsf{L}_m(\alpha) \to \mathsf{L}_{\max_0(n-m)}(\alpha)$. Now we want to check: $\mathsf{skolem\_demo} : \alpha \times \mathsf{L}_n(\alpha) \to \mathsf{L}_0(\alpha)$.

Without $\exists - I$ type-checking fails:

Step	The subgoal after the step
main goal	$\mathsf{z}\colon \alpha,\ \mathsf{l}\colon \mathsf{L}_n(\alpha) \vdash_\Sigma e_{\mathsf{skolem\_demo}}\colon \mathsf{L}_0(\alpha)$
Let-bind 1	$\mathsf{z}\colon \alpha,\ \mathsf{l}\colon \mathsf{L}_n(\alpha) \vdash_\Sigma \mathsf{insert}(=, \mathsf{x},\ \mathsf{tl})\colon \tau^?$
Fun	$\tau^? := \mathsf{L}_{n+i}^{\exists\, i.\, i \le 1}(\alpha)$
Let-bind 2	$\mathsf{z}\colon \alpha,\ \mathsf{l}\colon \mathsf{L}_n(\alpha),\ \mathsf{l}'\colon \mathsf{L}_{n+i}^{\exists\, i.\, i \le 1}(\alpha) \vdash_\Sigma \mathsf{Cons}(\mathsf{z},\ \mathsf{l}')\colon \tau_1^?$
Cons	$\tau_1^? := \mathsf{L}_{n+i+1}^{\exists\, i.\, i \le 1}(\alpha)$
Let-bind 3	$\mathsf{z}\colon \alpha,\ \mathsf{l}\colon \mathsf{L}_n(\alpha),\ \mathsf{l}'\colon \mathsf{L}_{n+i}^{\exists\, i.\, i \le 1}(\alpha) \vdash_\Sigma \mathsf{Cons}(\mathsf{z},\ \mathsf{l}')\colon \tau_2^?$
Cons	$\tau_2^? := \mathsf{L}_{n+i+1}^{\exists\, i.\, i \le 1}(\alpha)$
Let-body	$\mathsf{l}_1\colon \mathsf{L}_{n+i+1}^{\exists\, i.\, i \le 1}(\alpha),\ \mathsf{l}_2\colon \mathsf{L}_{n+i+1}^{\exists\, i.\, i \le 1}(\alpha) \vdash_\Sigma \mathsf{diff}(\mathsf{l}_1, \mathsf{l}_2)\colon \mathsf{L}_0(\alpha)$
Fun	$0 \le i_1 \le 1,\ 0 \le i_2 \le 1 \vdash \max_0((n + i_1 + 1) - (n + i_2 + 1)) = 0$

Apply $\exists - I$ before "Let-bind 2". Then type checking succeeds:

Step	The subgoal after the step
$\exists$-I	$n^i \le 1,\ \mathsf{z}\colon \alpha,\ \mathsf{l}\colon \mathsf{L}_n(\alpha),\ \mathsf{l}'\colon \mathsf{L}_{n+n^i}(\alpha) \vdash_\Sigma \mathsf{let}\ \mathsf{l}_1 = -\ \mathsf{in}\ -\colon \mathsf{L}_0(\alpha)$
Let-bind 2	$n^i \le 1,\ \mathsf{z}\colon \alpha,\ \mathsf{l}\colon \mathsf{L}_n(\alpha),\ \mathsf{l}'\colon \mathsf{L}_{n+n^i}(\alpha) \vdash_\Sigma \mathsf{Cons}(\mathsf{z},\ \mathsf{l}')\colon \tau_1^?$
Cons	$\tau_1^? := \mathsf{L}_{n+n^i+1}(\alpha)$
Let-bind 3	$n^i \le 1,\ \mathsf{z}\colon \alpha,\ \mathsf{l}\colon \mathsf{L}_n(\alpha),\ \mathsf{l}'\colon \mathsf{L}_{n+n^i}(\alpha) \vdash_\Sigma \mathsf{Cons}(\mathsf{z},\ \mathsf{l}')\colon \tau_2^?$
Cons	$\tau_2^? := \mathsf{L}_{n+n^i+1}(\alpha)$
Let-body	$n^i \le 1,\ \mathsf{l}_1\colon \mathsf{L}_{n+n^i+1}(\alpha),\ \mathsf{l}_2\colon \mathsf{L}_{n+n^i+1}(\alpha) \vdash_\Sigma \mathsf{diff}(\mathsf{l}_1, \mathsf{l}_2)\colon \mathsf{L}_0(\alpha)$
Fun	$0 \le n^i \le 1 \vdash \max_0((n + n^i + 1) - (n + n^i + 1)) = 0$

$$\begin{array}{l}\mathsf{rinsert}(\mathsf{g},\, \mathsf{l}_1, \mathsf{l}_2) = \\ \mathsf{match}\ \mathsf{l}_1\ \mathsf{with} \mid \mathsf{Nil} \Rightarrow \mathsf{l}_2 \\ \qquad\qquad\quad \mid \mathsf{Cons}(\mathsf{hd},\, \mathsf{tl}) \Rightarrow \mathsf{let}\ \mathsf{l} = \mathsf{rinsert}(\mathsf{g},\, \mathsf{tl}, \mathsf{l}_2)\ \mathsf{in}\ \mathsf{insert}(\mathsf{g},\, \mathsf{hd}, \mathsf{l})\end{array}$$

Show that

$$\mathsf{rinsert} : (\alpha \times \alpha \to \mathsf{Bool}) \times \mathsf{L}_n(\alpha) \times \mathsf{L}_m(\alpha) \to \mathsf{L}_{m+i}^{i \le n}(\alpha)$$

Step	The subgoal after the step
main goal	$\mathsf{l}_1 : \mathsf{L}_n(\alpha),\ \mathsf{l}_2 : \mathsf{L}_m(\alpha) \vdash_\Sigma e_{\mathsf{rinsert}} : \mathsf{L}_{m+i}^{i \le n}(\alpha)$
MatchNil	$n = 0,\ \mathsf{l}_1 : \mathsf{L}_n(\alpha),\ \mathsf{l}_2 : \mathsf{L}_m(\alpha) \vdash_\Sigma \mathsf{y} : \mathsf{L}_{m+i}^{i \le n}(\alpha)$
Var	$n = 0 \vdash i^? \le n \wedge m = m + i^?$ $i^? := 0$
MatchCons	$n \ge 1,\ \mathsf{l}_1 : \mathsf{L}_n(\alpha),\ \mathsf{hd} : \alpha,\ \mathsf{tl} : \mathsf{L}_{n-1}(\alpha),\ \mathsf{l}_2 : \mathsf{L}_m(\alpha) \vdash_\Sigma$ $\mathsf{let}\ \mathsf{l} = \mathsf{rinsert}(\mathsf{g},\, \mathsf{tl}, \mathsf{l}_2)\ \mathsf{in}\ - :\ \mathsf{L}_{m+i}^{i \le n}(\alpha)$
Let-bind	$n \ge 1,\ \mathsf{tl} : \mathsf{L}_{n-1}(\alpha),\ \mathsf{l}_2 : \mathsf{L}_m(\alpha) \vdash_\Sigma \mathsf{rinsert}(\mathsf{g}, \mathsf{tl}, \mathsf{l}_2) : \tau^?$
Fun	$\tau^? := \mathsf{L}_{m+j}^{j \le n-1}(\alpha)$
Let-body	$n \ge 1,\ \mathsf{l}_2 : \mathsf{L}_m(\alpha),\ \mathsf{hd} : \alpha, \mathsf{tl} : \mathsf{L}_{n-1}(\alpha),\ \mathsf{l} : \mathsf{L}_{m+j}^{j \le n-1}(\alpha) \vdash_\Sigma$ $\mathsf{insert}(\mathsf{hd},\ \mathsf{l}) :\ \mathsf{L}_{m+i}^{i \le n}(\alpha)$
Fun	$n \ge 1,\ j \le n - 1,\ j' \le 1 \vdash i^? \le n \wedge m + j + j' = m + i^?$ $i^? := j + j' \quad (i^? \le n,\ j + j' \le n)$

$$\begin{array}{l}\mathsf{delete}(\mathsf{g},\,\mathsf{z},\,\mathsf{l}) = \\ \mathsf{match}\ \mathsf{l}\ \mathsf{with}\ |\ \mathsf{Nil} \Rightarrow \mathsf{Nil} \\ \qquad\qquad |\ \mathsf{Cons}(\mathsf{hd},\,\mathsf{tl}) \Rightarrow \mathsf{if}\ \mathsf{g}(\mathsf{z},\,\mathsf{hd})\ \mathsf{then}\ \mathsf{tl}\ \mathsf{else}\ \mathsf{let}\ \mathsf{z}' = \mathsf{delete}(\mathsf{g},\,\mathsf{z},\mathsf{tl})\ \mathsf{in}\ \mathsf{Cons}(\mathsf{hd},\,\mathsf{z}')\end{array}$$

$$\mathsf{delete} : (\alpha \times \alpha \to \mathsf{Bool}) \times \alpha \times \mathsf{L}_n(\alpha) \to \mathsf{L}^{\exists\, i.\, i \le 1}_{\max_0(n-i)}(\alpha)$$

Step	The subgoal after the step
main goal	$\mathsf{z}\colon \alpha,\ \mathsf{l}\colon \mathsf{L}_n(\alpha) \vdash_\Sigma e_{\mathsf{delete}} \colon \mathsf{L}^{\exists\, i.\, i \le 1}_{\max_0(n-i)}(\alpha)$
MatchNil	$n = 0;\ \mathsf{z}\colon \alpha,\ \mathsf{l}\colon \mathsf{L}_n(\alpha) \vdash_\Sigma \mathsf{Nil} \colon \mathsf{L}^{\exists\, i.\, i \le 1}_{\max_0(n-i)}(\alpha)$
Nil	$n = 0 \vdash i^? \le 1 \wedge 0 = \max_0(n - i^?)$
	$i^? := 0$
MatchCons	$n \ge 1;\ \mathsf{z}\colon \alpha,\ \mathsf{l}\colon \mathsf{L}_n(\alpha),\ \mathsf{tl}\colon \mathsf{L}_{n-1}(\alpha) \vdash_\Sigma \mathsf{if}\ -\ \mathsf{then}\ -\ \mathsf{else}\ -\colon \mathsf{L}^{\exists\, i.\, i \le 1}_{\max_0(n-i)}(\alpha)$
IfTrue	$n \ge 1;\ \mathsf{tl}\colon \mathsf{L}_{n-1}(\alpha) \vdash_\Sigma \mathsf{tl}\colon \mathsf{L}^{\exists\, i.\, i \le 1}_{\max_0(n-i)}(\alpha)$
Var	$n \ge 1 \vdash i^? \le 1 \wedge n - 1 = \max_0(n - i^?)$
	$i^? := 1$
IfFalse	$n \ge 1;\ \mathsf{z}\colon \alpha,\ \mathsf{hd}\colon \alpha,\ \mathsf{tl}\colon \mathsf{L}_{n-1}(\alpha) \vdash_\Sigma \mathsf{let}\ \mathsf{z}' = -\ \mathsf{in}\ -\colon \mathsf{L}^{\exists\, i.\, i \le 1}_{\max_0(n-i)}(\alpha)$
Let-bind	$n \ge 1;\ \mathsf{z}\colon \alpha,\ \mathsf{tl}\colon \mathsf{L}_{n-1}(\alpha) \vdash_\Sigma \mathsf{delete}(\mathsf{g},\mathsf{z},\,\mathsf{tl})\colon \tau^?$
Fun	$\tau^? := \mathsf{L}^{\exists.\, j \le 1}_{\max_0((n-1)-j)}(\alpha)$
Let-body	$n \ge 1;\ \mathsf{hd}\colon \alpha,\ \mathsf{tl}\colon \mathsf{L}_{n-1}(\alpha),\ \mathsf{z}'\colon \mathsf{L}^{\exists\, j.\, j \le 1}_{\max_0((n-1)-j)}(\alpha) \vdash_\Sigma$ $\mathsf{Cons}(\mathsf{hd},\,\mathsf{z})\colon\ \mathsf{L}^{\exists\, i.\, i \le 1}_{\max_0(n-i)}(\alpha)$
Cons	$n \ge 1,\ j \le 1 \vdash i^? \le 1 \wedge \max_0((n-1) - j) + 1 = \max_0(n - i^?)$
cases for j	$n \ge 1,\ j = 0 \vdash i^? \le 1 \wedge n = \max_0(n - i^?)$
	$n \ge 1,\ j = 1 \vdash i^? \le 1 \wedge \max_0((n-1) - 1) + 1 = \max_0(n - i^?)$
cases	$n \ge 1,\ j = 0 \vdash i^? \le 1 \wedge i^? = 0$
	$n = 1,\ j = 1 \vdash i^? \le 1 \wedge 0 = \max_0(n - i^?)$
	$n \ge 2,\ j = 1 \vdash i^? \le 1 \wedge n - 1 - 1 + 1 = \max_0(n - i^?)$
cases	$n \ge 1,\ j = 0 \vdash i^? \le 1 \wedge i^? = 0$
	$n = 1,\ j = 1 \vdash i^? \le 1 \wedge i^? = 1$
	$n \ge 2,\ j = 1 \vdash i^? \le 1 \wedge i^? = 1$

$$\begin{array}{l}\mathsf{rdelete}(\mathsf{g},\,\mathsf{l}_1,\mathsf{l}_2) = \\ \mathsf{match}\ \mathsf{l}_1\ \mathsf{with}\ |\ \mathsf{Nil} \Rightarrow \mathsf{l}_2 \\ \qquad\qquad\ |\ \mathsf{Cons}(\mathsf{hd},\,\mathsf{tl}) \Rightarrow \mathsf{let}\ \mathsf{l}' = \mathsf{rdelete}(\mathsf{g},\,\mathsf{tl},\mathsf{l}_2)\ \mathsf{in}\ \mathsf{delete}(\mathsf{g},\,\mathsf{hd},\mathsf{l}')\end{array}$$

$$\mathsf{rdelete} : (\alpha \times \alpha \to \mathsf{Bool}) \times \mathsf{L}_n(\alpha) \times \mathsf{L}_m(\alpha) \to \mathsf{L}^{\exists\, i.\, i \le n}_{\max_0(m-i)}(\alpha)$$

Step	The subgoal after the step
main goal	$\mathsf{l}_1 : \mathsf{L}_n(\alpha),\ \mathsf{l}_2 : \mathsf{L}_m(\alpha) \vdash_\Sigma e_{\mathsf{rdelete}} : \mathsf{L}^{\exists\, i. i \le n}_{\max_0(m-i)}(\alpha)$
MatchNil	$n = 0,\ \mathsf{l}_1 : \mathsf{L}_n(\alpha),\ \mathsf{l}_2 : \mathsf{L}_m(\alpha) \vdash_\Sigma \mathsf{l}_2 : \mathsf{L}^{\exists\, i. i \le n}_{\max_0(m-i)}(\alpha)$
Var Simp	$n = 0 \vdash i^? \le n \wedge m = \max_0(m - i^?)$ $i^? := 0$
MatchCons	$n \ge 1;\ \mathsf{l}_1 : \mathsf{L}_n(\alpha),\ \mathsf{hd} : \alpha,\ \mathsf{tl} : \mathsf{L}_{n-1}(\alpha),\ \mathsf{l}_2 : \mathsf{L}_m(\alpha) \vdash_\Sigma$ $\mathsf{let}\ \mathsf{l}' = -\ \mathsf{in}\ - :\ \mathsf{L}^{\exists\, i. i \le n}_{\max_0(m-i)}(\alpha)$
Let-bind	$n \ge 1;\ \mathsf{tl} : \mathsf{L}_{n-1}(\alpha),\ \mathsf{l}_2 : \mathsf{L}_m(\alpha) \vdash_\Sigma \mathsf{rdelete}(\mathsf{g}, \mathsf{tl}, \mathsf{l}_2) : \tau^?$
Fun	$\tau^? := \mathsf{L}^{\exists\, j.\, j \le n-1}_{\max_0(m-j)}(\alpha)$
Let-body	$n \ge 1;\ \mathsf{l}_2 : \mathsf{L}_m(\alpha),\ \mathsf{hd} : \alpha,\ \mathsf{tl} : \mathsf{L}_{n-1}(\alpha),\ \mathsf{l}' : \mathsf{L}^{\exists\, j.\, j \le n-1}_{\max_0(m-j)}(\alpha) \vdash_\Sigma$ $\mathsf{delete}(\mathsf{g},\ \mathsf{hd},\ \mathsf{l}') :\ \mathsf{L}^{\exists\, i.\, i \le n}_{\max_0(m-i)}(\alpha)$
Fun	$n \ge 1,\ j \le n-1,\ j' \le 1 \vdash i^? \le n \wedge$ $\max_0(\max_0(m-j) - j') = \max_0(m - i^?)$
Cases for j, j', find $i^?$	Let P be $n \ge 1,\ j \le n-1,\ j' \le 1$
Case 1 Simp	$P,\ j \le m,\ j' \le m - j \vdash i^? \le n \wedge m - j - j' = \max_0(m - i^?)$ $i^? := j + j' \quad (j + j' \le n)$
Case 2 Simp because	$P,\ j \le m,\ j' > m - j \vdash i^? \le n \wedge 0 = \max_0(m - i^?)$ $i^? := m \quad (i^? \le n)$ $\{(n,\, m,\, j,\, j').\ n \ge 1,\ j \le n-1,\ j \le m,\ 1 \ge j' > m - j\} \subseteq$ $\{(n,\, m,\, j,\, j').\ 1 \ge j' > m - (n-1)\} \subseteq$ $\{(n,\, m,\, j,\, j').\ n > m\}$
Case 3 Simp because	$P,\ j > m \vdash i^? \le n \wedge 0 = \max_0(m - i^?)$ $i^? := m \quad (i^? \le n)$ $\{(n,\, m,\, j,\, j').\ n \ge 1,\ j \le n-1,\ j > m\} \subseteq$ $\{(n,\, m,\, j,\, j').\ n - 1 > m\}$

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

Show that

$$\mathsf{deleteall} : (\alpha \times \alpha \to \mathsf{Bool}) \times \alpha \times \mathsf{L}_n(\alpha) \to \mathsf{L}_{n-i}^{i \le n}(\alpha)$$

Step	The subgoal after the step
main goal	$\mathsf{z}\colon \alpha,\ \mathsf{l}\colon \mathsf{L}_n(\alpha) \vdash_\Sigma e_{\mathsf{deleteall}} \colon \mathsf{L}_{n-i}^{i \le n}(\alpha)$
MatchNil	$n = 0,\ \mathsf{z}\colon \alpha,\ \mathsf{l}\colon \mathsf{L}_n(\alpha) \vdash_\Sigma \mathsf{Nil}\colon \mathsf{L}_{n-i}^{i \le n}(\alpha)$
Nil	$n = 0 \vdash i^? \le n \wedge 0 = n - i^?$
	$i^? := 0$
MatchCons	$n \ge 1,\ \mathsf{z}\colon \alpha,\ \mathsf{l}_2\colon \mathsf{L}_n(\alpha) \vdash_\Sigma \mathsf{if}\ -\ \mathsf{then}\ -\ \mathsf{else}\ -\colon \mathsf{L}_{n-i}^{i \le n}(\alpha)$
IfTrue	$n \ge 1,\ \mathsf{l}\colon \mathsf{L}_n(\alpha),\ \mathsf{z}\colon \alpha,\ \mathsf{tl}\colon \mathsf{L}_{n-1}(\alpha) \vdash_\Sigma \mathsf{deleteall}(\mathsf{g}, \mathsf{z}, \mathsf{tl})\colon \mathsf{L}_{n-i}^{i \le n}(\alpha)$
Fun	$n \ge 1,\ j \le n - 1 \vdash i^? \le n \wedge (n-1) - j = n - i^?$
Simp	$n \ge 1,\ j \le n - 1 \vdash i^? \le n \wedge i^? = j + 1$
IfFalse	$n \ge 1,\ \mathsf{z}\colon \alpha,\ \mathsf{hd}\colon \alpha,\ \mathsf{tl}\colon \mathsf{L}_{n-1}(\alpha) \vdash_\Sigma \mathsf{let}\ -\ =\ -\ \mathsf{in}\ -\colon \alpha$
Let-bind	$n \ge 1,\ \mathsf{z}\colon \alpha,\ \mathsf{tl}\colon \mathsf{L}_{n-1}(\alpha) \vdash_\Sigma \mathsf{deleteall}(\mathsf{g}, \mathsf{z}, \mathsf{tl})\colon \tau^?$
Fun	$\tau^? := \mathsf{L}_{(n-1)-j}^{j \le n-1}(\alpha)$
Let-body	$n \ge 1,\ \mathsf{hd}\colon \alpha,\ \mathsf{tl}\colon \mathsf{L}_{n-1}(\alpha),\ \mathsf{l}'\colon \mathsf{L}_{(n-1)-j}^{j \le n-1}(\alpha) \vdash_\Sigma \mathsf{Cons}(\mathsf{hd}, \mathsf{l}')\colon\ \mathsf{L}_{n-i}^{i \le n}(\alpha)$
Cons	$n \ge 1;\ j \le n - 1 \vdash i^? \le n \wedge (n-1) - j + 1 = n - i^?$
	$i^? := j$

$\mathsf{filter(g,\ l)} =$
$\mathsf{match\ l\ with} \mid \mathsf{Nil} \Rightarrow \mathsf{Nil}$
$\mid \mathsf{Cons(hd,\ tl)} \Rightarrow \mathsf{if\ g(hd)\ then\ let\ l' = filter(g,\ tl)\ in\ Cons(hd,\ l')\ else\ filter(g,\ tl)}$

Show that

$$\mathsf{filter} : (\alpha \to \mathsf{Bool}) \times \mathsf{L}_n(\alpha) \to \mathsf{L}_i^{i \le n}(\alpha)$$

Step	The subgoal after the step
main goal	$\mathsf{l}\colon \mathsf{L}_n(\alpha) \vdash_\Sigma e_{\mathsf{filter}} \colon \mathsf{L}_i^{i \le n}(\alpha)$
MatchNil	$n = 0,\ \mathsf{l}\colon \mathsf{L}_n(\alpha) \vdash_\Sigma \mathsf{Nil}\colon \mathsf{L}_i^{i \le n}(\alpha)$
Nil	$n = 0 \vdash i^? \le n \wedge 0 = i^?$
	$i^? := 0$
MatchCons	$n \ge 1,\ \mathsf{l}\colon \mathsf{L}_n(\alpha),\ \mathsf{hd}\colon \alpha,\ \mathsf{tl}\colon \mathsf{L}_{n-1}(\alpha) \vdash_\Sigma$ $\mathsf{if} - \mathsf{then} - \mathsf{else} -\colon\ \mathsf{L}_i^{i \le n}(\alpha)$
IfTrue	$n \ge 1,\ \mathsf{l}\colon \mathsf{L}_n(\alpha),\ \mathsf{hd}\colon \alpha,\ \mathsf{tl}\colon \mathsf{L}_{n-1}(\alpha) \vdash_\Sigma$ $\mathsf{let\ l' = filter(g, tl)\ in\ Cons(hd,\ l')}\colon\ \mathsf{L}_i^{i \le n}(\alpha)$
Let-bind	$n \ge 1,\ \mathsf{tl}\colon \mathsf{L}_{n-1}(\alpha) \vdash_\Sigma \mathsf{filter(g, tl)}\colon \tau^?$
Fun	$\tau^? := \mathsf{L}_j^{j \le n-1}(\alpha)$
Let-body	$n \ge 1,\ \mathsf{l}\colon \mathsf{L}_n(\alpha),\ \mathsf{hd}\colon \alpha,\ \mathsf{l'}\colon \mathsf{L}_j^{j \le n-1}(\alpha) \vdash_\Sigma \mathsf{Cons(hd,\ l')}\colon \mathsf{L}_i^{i \le n}(\alpha)$
Cons	$n \ge 1,\ j \le n-1 \vdash i^? \le n \wedge j + 1 = i^?$
	$i^? := j + 1 \quad (i^? \le n,\ j + 1 \le n)$
IfFalse	$n \ge 1,\ \mathsf{l}\colon \mathsf{L}_n(\alpha),\ \mathsf{hd}\colon \alpha,\ \mathsf{tl}\colon \mathsf{L}_{n-1}(\alpha) \vdash_\Sigma \mathsf{filter(g, tl)}\colon \mathsf{L}_i^{i \le n}(\alpha)$
Fun	$n \ge 1,\ j \le n-1 \vdash i^? \le n \wedge j = i^?$
	$i^? := j \quad (i^? \le n,\ j \le n-1)$

$$
\begin{array}{l}
\mathsf{divtwo(l)} = \\
\mathsf{match\ l\ with} \mid \mathsf{Nil} \Rightarrow \mathsf{Nil} \\
\qquad\qquad \mid \mathsf{Cons(hd,\ tl)} \Rightarrow \mathsf{match\ tl\ with} \mid \mathsf{Nil} \Rightarrow \mathsf{Nil} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mid \mathsf{Cons(hd',\ tl')} \Rightarrow \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathsf{let\ l' = divtwo(tl')\ in\ Cons(hd,\ l')}
\end{array}
$$

Show that

$$\mathsf{divtwo} : \mathsf{L}_n(\alpha) \to \mathsf{L}^{i\le 1}_{\frac{\max_0(n-i)}{2}}(\alpha)$$

Step	The subgoal after the step
main goal	$\mathsf{l}\colon \mathsf{L}_n(\alpha) \vdash_\Sigma e_{\mathsf{divtwo}} \colon \mathsf{L}^{i\le 1}_{\frac{\max_0(n-i)}{2}}(\alpha)$
MatchNil	$n = 0,\ \mathsf{l}\colon \mathsf{L}_n(\alpha) \vdash_\Sigma \mathsf{Nil}\colon \mathsf{L}^{i\le 1}_{\frac{\max_0(n-i)}{2}}(\alpha)$
Nil	$n = 0 \vdash i^? \le 1 \wedge 0 = \max_0(n - i^?)/2$ $i^? := 0$
MatchCons	$n \ge 1,\ \mathsf{l}\colon \mathsf{L}_n(\alpha),\ \mathsf{hd}\colon \alpha,\ \mathsf{tl}\colon \mathsf{L}_{n-1}(\alpha) \vdash_\Sigma \mathsf{match}\ldots\colon \mathsf{L}^{i\le 1}_{\frac{\max_0(n-i)}{2}}(\alpha)$
MatchNil	$n \ge 1,\ n-1 = 0;\ \mathsf{l}\colon \mathsf{L}_n(\alpha),\ \mathsf{hd}\colon \alpha,\ \mathsf{tl}\colon \mathsf{L}_{n-1}(\alpha) \vdash_\Sigma \mathsf{Nil}\colon \mathsf{L}^{i\le 1}_{\frac{\max_0(n-i)}{2}}(\alpha)$
Nil	$n = 1 \vdash i^? \le 1 \wedge 0 = \max_0(n - i^?)/2$
Simp	$i^? := 1$
MatchCons	$n-1 \ge 1,\ \mathsf{l}\colon \mathsf{L}_n(\alpha),\ \mathsf{hd}\colon \alpha,\ \mathsf{tl}\colon \mathsf{L}_{n-1}(\alpha),\ \mathsf{hd'}\colon \alpha,\ \mathsf{tl'}\colon \mathsf{L}_{n-2}(\alpha) \vdash_\Sigma$ $\mathsf{let\ y = divtwo(tl')\ in\ Cons(hd,\ y)}\colon\ \mathsf{L}^{i\le 1}_{\frac{\max_0(n-i)}{2}}(\alpha)$
Let-bind	$n-1 \ge 1;\ \mathsf{tl'}\colon \mathsf{L}_{n-2}(\alpha) \vdash_\Sigma \mathsf{divtwo(tl')}\colon \tau^?$
Fun	$\tau^? := \mathsf{L}^{j\le 1}_{\frac{\max_0((n-2)-j)}{2}}(\alpha)$
Let-body	$n-1 \ge 1;\ \ldots\ \mathsf{hd}\colon \alpha,\ \mathsf{l'}\colon \mathsf{L}^{j\le 1}_{\frac{\max_0((n-2)-i)}{2}}(\alpha) \vdash_\Sigma$ $\mathsf{Cons(hd,\ l')}\colon\ \mathsf{L}^{i\le 1}_{\frac{\max_0(n-i)}{2}}(\alpha)$
Cons	$n-1 \ge 1,\ j \le 1 \vdash i^? \le 1 \wedge$ $\max_0((n-2)-j)/2 + 1 = \max_0(n - i^?)/2$
Simp ($i \le n$)	$n \ge 2,\ j \le 1 \vdash i^? \le 1 \wedge \max_0((n-2)-j)/2 + 1 = (n - i^?)/2$
Cases for j find $i^?$	$n \ge 2,\ j \le 1,\ j \le n-2 \vdash$ $i^? \le 1 \wedge ((n-2)-j)/2 + 1 = (n - i^?)/2$ $n \ge 2,\ j \le 1,\ j \le n-2 \vdash i^? \le 1 \wedge i^? = j$ $i^? := j \quad (i^? \le 1,\ j \le 1)$
	$n \ge 2,\ j \le 1,\ j > n-2 \vdash i^? \le 1 \wedge 1 = (n - i^?)/2$ $n \ge 2,\ j \le 1,\ j > n-2,\ n = 2,\ j = 1 \vdash i^? \le 1 \wedge i^? = 0$ $i^? := 0 \quad (i^? \le 1)$

$$\begin{array}{l}\mathsf{rel\_pairs(g, z, l)} = \mathsf{match\ l\ with} \mid \mathsf{Nil} \Rightarrow \mathsf{Nil} \\ \qquad\qquad\qquad\qquad\qquad\quad \mid \mathsf{Cons(hd,\ tl)} \Rightarrow \mathsf{if\ g(z, hd)\ then} \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathsf{Cons(Cons(z,\ Cons(hd,\ Nil)),} \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathsf{rel\_pairs(z, tl))} \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathsf{else\ rel\_pairs(z, tl)}\end{array}$$

$$\mathsf{rel\_pairs}\colon (\alpha \times \alpha \to \alpha) \times \alpha \times \mathsf{L}_n(\alpha) \to \mathsf{L}_i^{\exists.\, i \le n}(\mathsf{L}_2(\alpha))$$

Step	The subgoal after the step
main goal	$\mathsf{z}\colon \alpha, \mathsf{l}\colon \mathsf{L}_n(\alpha) \vdash_\Sigma e_{\mathsf{rel\_pairs}}\colon \mathsf{L}_i^{i \le n}(\mathsf{L}_2(\alpha))$
MatchNil	$n = 0,\ \mathsf{z}\colon \alpha, \mathsf{l}\colon \mathsf{L}_n(\alpha) \vdash_\Sigma \mathsf{Nil}\colon \mathsf{L}_i^{i \le n}(\mathsf{L}_2(\alpha))$
Nil	$n = 0 \vdash i^? \le n \wedge 0 = i^?$ $i^? := 0$
MatchCons	$n \ge 1,\ \mathsf{l}\colon \mathsf{L}_n(\alpha),\ \mathsf{hd}\colon \alpha,\ \mathsf{tl}\colon \mathsf{L}_{n-1}(\alpha) \vdash_\Sigma$ $\mathsf{if} - \mathsf{then} - \mathsf{else} -\colon \mathsf{L}_i^{i \le n}(\mathsf{L}_2(\alpha))$
IfTrue	$n \ge 1,\ \mathsf{l}\colon \mathsf{L}_n(\alpha),\ \mathsf{hd}\colon \alpha,\ \mathsf{tl}\colon \mathsf{L}_{n-1}(\alpha) \vdash_\Sigma$ $\mathsf{let\ l'} = \mathsf{rel\_pairs(g, z, tl)\ in\ Cons(Cons(z,\ Cons(hd,\ Nil)),\ l')}\colon \mathsf{L}_i^{i \le n}(\mathsf{L}_2(\alpha))$
Let-bind	$n \ge 1,\ \mathsf{tl}\colon \mathsf{L}_{n-1}(\alpha) \vdash_\Sigma \mathsf{rel\_pairs(g, z, tl)}\colon \tau^?$
Fun	$\tau^? := \mathsf{L}_j^{j \le n-1}(\mathsf{L}_2(\alpha))$
Let-body	$n \ge 1,\ \mathsf{l}\colon \mathsf{L}_n(\alpha),\ \mathsf{hd}\colon \alpha,\ \mathsf{l'}\colon \mathsf{L}_j^{j \le n-1}(\mathsf{L}_2(\alpha)) \vdash_\Sigma$ $\mathsf{Cons(Cons(z,\ Cons(hd,\ Nil)),\ l')}\colon \mathsf{L}_i^{i \le n}(\mathsf{L}_2(\alpha))$
Cons	$n \ge 1,\ j \le n-1 \vdash i^? \le n \wedge j + 1 = i^?$ $i^? := j + 1 \quad (i^? \le n,\ j + 1 \le n)$
IfFalse	$n \ge 1,\ \mathsf{z}\colon \alpha, \mathsf{tl}\colon \mathsf{L}_{n-1}(\alpha),\ \mathsf{hd}\colon \alpha \vdash_\Sigma \mathsf{rel\_pairs(g, z, tl)}\colon \mathsf{L}_i^{i \le n}(\mathsf{L}_2(\alpha))$
Fun	$n \ge 1,\ j \le n-1 \vdash i^? \le n \wedge j = i^?$ $n \ge 1,\ j \le n-1, j \ge 1 \vdash 2 = 2$ $i^? := j \quad (i^? \le n,\ j \le n-1)$

$$\mathsf{rel}(\mathsf{g}, \mathsf{l}_1, \mathsf{l}_2) = \mathsf{match}\ \mathsf{l}_1\ \mathsf{with}\ |\ \mathsf{Nil} \Rightarrow \mathsf{Nil} \\ |\ \mathsf{Cons}(\mathsf{hd},\ \mathsf{tl}) \Rightarrow \mathsf{append}(\mathsf{rel\_pairs}(\mathsf{g}, \mathsf{hd}, \mathsf{l}_2),\ \mathsf{rel}(\mathsf{g}, \mathsf{tl}, \mathsf{l}_2))$$

$$\mathsf{rel} : (\alpha \times \alpha \to \mathsf{Bool}) \times \mathsf{L}_n(\alpha) \times \mathsf{L}_m(\alpha) \to \mathsf{L}_i^{i \le nm}(\mathsf{L}_2(\alpha))$$

Step	The subgoal after the step
main goal	$\mathsf{l}_1 : \mathsf{L}_n(\alpha),\ \mathsf{l}_2 : \mathsf{L}_m(\alpha) \vdash_\Sigma e_{\mathsf{rinsert}} : \mathsf{L}_i^{i \le nm}(\mathsf{L}_2(\alpha))$
MatchNil	$n = 0,\ \mathsf{l}_1 : \mathsf{L}_n(\alpha),\ \mathsf{l}_2 : \mathsf{L}_m(\alpha) \vdash_\Sigma \mathsf{Nil} : \mathsf{L}_i^{i \le nm}(\mathsf{L}_2(\alpha))$
Var	$n = 0 \vdash i^? \le nm \wedge 0 = i^?$ $i^? := 0$
MatchCons	$n \ge 1,\ \mathsf{l}_1 : \mathsf{L}_n(\alpha),\ \mathsf{hd} : \alpha,\ \mathsf{tl} : \mathsf{L}_{n-1}(\alpha),\ \mathsf{l}_2 : \mathsf{L}_m(\alpha) \vdash_\Sigma$ $\mathsf{let}\ \mathsf{l}' = \mathsf{rel\_pairs}(\mathsf{g}, \mathsf{hd}, \mathsf{l}_2)\ \mathsf{in}\ - : \mathsf{L}_i^{i \le nm}(\mathsf{L}_2(\alpha))$
Let-bind 1	$n \ge 1,\ \mathsf{hd} : \alpha,\ \mathsf{l}_2 : \mathsf{L}_m(\alpha) \vdash_\Sigma \mathsf{rel\_pairs}(\mathsf{g}, \mathsf{hd}, \mathsf{l}_2) : \tau^?$
Fun	$\tau^? := \mathsf{L}_j^{j \le m}(\mathsf{L}_2(\alpha))$
Let-bind 2	$n \ge 1,\ \mathsf{tl} : \mathsf{L}_{n-1}(\alpha),\ \mathsf{l}_2 : \mathsf{L}_m(\alpha) \vdash_\Sigma \mathsf{rel}(\mathsf{g}, \mathsf{tl}, \mathsf{l}_2) : \tau^?$
Fun	$\tau^? := \mathsf{L}_{j'}^{j' \le (n-1)m}(\mathsf{L}_2(\alpha))$
Let-body	$n \ge 1,\ \mathsf{l}' : \mathsf{L}_j^{j \le m}(\mathsf{L}_2(\alpha)), \mathsf{l}'' : \mathsf{L}_{j'}^{j' \le (n-1)m}(\mathsf{L}_2(\alpha)) \vdash_\Sigma$ $\mathsf{append}(\mathsf{l}',\ \mathsf{l}'') :\ \mathsf{L}_i^{i \le nm}(\mathsf{L}_2(\alpha))$
Fun	$n \ge 1,\ j \le m,\ j' \le n - 1 \vdash i^? \le nm \wedge j + j' = i^?$ $i^? := j + j'$

```
log2(l) =
match l with | Nil ⇒ Nil
             | Cons(hd, tl) ⇒ match tl with | Nil ⇒ Nil
                                            | Cons(hd′, tl′) ⇒ let z = divtwo(l) in
                                                               if hd = 0 then log2(z) else
                                                               Cons(hd, log2(z))
```

We will show that for any $k \geq 1$ the following type passes type-checking:

$\mathsf{log2} : \mathsf{L}_n(\mathsf{Int}) \to \mathsf{L}_i^{i \leq f_k(n)}(\mathsf{Int})$, where f_k is defined as

$$f_k(n) = \begin{cases} 0 & \text{if } n = 0 \\ l & \text{if } 2^l \leq n < 2^{l+1}, \text{where } 0 \leq l \leq k-1 \\ \dfrac{k}{2^k} n & \text{if } n \geq 2^k \end{cases}$$

Hence for each k, f_k can is defined as a picewise polynomial with $k+1$ pieces. To perform type checking we first prove the following lemmata.

Lemma 1. *For all $k \geq 1$ one can prove in the integer arithmetic that the function f_k is non-decreasing.*

Proof. Fix some $m \leq n$. The proof-scheme (works for any k) is done by case distinction on n.

- *Case $n = 0$: then $m = 0$ and $f_k(m) = f_k(n) = 0$.*
- *Case $2^l \leq n < 2^{l+1}$ for some $0 \leq l \leq k-1$: if $m = 0$ then $f_k(m) = 0 \leq l = f_k(n)$. Otherwise, there exists $l' \leq l$ such that $2^{l'} \leq m < 2^{l'+1}$, and then $f_k(m) = l' \leq l = f_k(m)$.*
- *Case $n \geq 2^{k+1}$: if $m = 0$ then $f(m) = 0 < \dfrac{k}{2^k} n = f_k(n)$. If $2^l \leq m < 2^{l+1}$ for some $l \leq k-1$, then $f_k(m) = l \leq k-1 < k = \dfrac{k}{2^k} 2^k \leq \dfrac{k}{2^k} n = f_k(n)$. Otherwise $m \geq 2^k$ and $f_k(m) = \dfrac{k}{2^k} m \leq \dfrac{k}{2^k} n = f_k(n)$.*

Lemma 2. *For all $n \geq 2$ it holds that $\dfrac{1}{2} n + 1 \leq n$.*

Proof. Indeed, $2(\dfrac{1}{2} n + 1) = n + 2 \leq n + n = 2n$ from what follows $\dfrac{1}{2} n + 1 \leq n$.

Lemma 3. *For all $k \geq 1$, $n \geq 2$ it holds that $f_k(\dfrac{n}{2}) + 1 \leq f_k(n)$ (and therefore, $f_k(\dfrac{n-1}{2}) + 1 \leq f_k(\dfrac{n}{2}) + 1 \leq f_k(n)$).*

Proof. By case distinction on n.

- Case $2^l \leq n < 2^{l+1}$, $1 \leq l \leq k-1$: then $2^{l-1} \leq \dfrac{n}{2} < 2^l$ and $1 + f_k(\dfrac{n}{2}) = 1 + (l-1) = l = f_k(n)$.

– $2^k \le n < 2^{k+1}$: then $2^{k-1} \le \dfrac{n}{2} < 2^k$ and $f_k(\dfrac{n}{2}) + 1 = (k-1) + 1 = k \le k\dfrac{n}{2^k} = f_k(n)$.

– $n \ge 2^{k+1}$: then $\dfrac{n}{2} \ge 2^k$, $f_k(n) = \dfrac{k}{2^k}n$ and $f_k(\dfrac{n}{2}) = \dfrac{k}{2^{k+1}}n$. Now using Lemma 2 (note that $kn/2^k > 2$), $f_k(\dfrac{1}{2}) + 1 = 1 + \dfrac{kn}{2^{k+1}} \le \dfrac{kn}{2^k} = f_k(n)$.

Step	The subgoal after the step
main goal	$\mathsf{l}\colon \mathsf{L}_n(\mathsf{Int}) \vdash_\Sigma e_{\mathsf{log2}}\colon \mathsf{L}_i^{i\le f_k(n)}(\mathsf{Int})$
MatchNil	$n = 0,\ \mathsf{l}\colon \mathsf{L}_n(\alpha) \vdash_\Sigma \mathsf{Nil}\colon \mathsf{L}_i^{i\le f_k(n)}(\mathsf{Int})$
Nil	$n = 0 \vdash i^? = 0 \wedge i^? \le f_k(n) = 0$ $i^? := 0$
MatchCons	$n \ge 1,\ \mathsf{l}\colon \mathsf{L}_n(\mathsf{Int}),\ \mathsf{hd}\colon \mathsf{Int},\ \mathsf{tl}\colon \mathsf{L}_{n-1}(\mathsf{Int}) \vdash_\Sigma \mathsf{match}\ldots\colon \mathsf{L}_i^{i\le f_k(n)}(\mathsf{Int})$
MatchNil	$n \ge 1,\ n-1 = 0;\ \mathsf{l}\colon \mathsf{L}_n(\mathsf{Int}),\ \mathsf{hd}\colon \mathsf{Int},\ \mathsf{tl}\colon \mathsf{L}_{n-1}(\mathsf{Int}) \vdash_\Sigma \mathsf{Nil}\colon \mathsf{L}_i^{i\le f_k(n)}(\mathsf{Int})$
Nil	$n = 1 \vdash i^? = 0 \wedge i^? \ge f_k(1) = 0$ $i^? := 0$
MatchCons	$n-1 \ge 1,\ \mathsf{l}\colon \mathsf{L}_n(\mathsf{Int}),\ \mathsf{hd}\colon \mathsf{Int},\ \mathsf{tl}\colon \mathsf{L}_{n-1}(\mathsf{Int}),\ \mathsf{hd}'\colon \mathsf{Int},\ \mathsf{tl}'\colon \mathsf{L}_{n-2}(\mathsf{Int}) \vdash_\Sigma$ $\mathsf{let\ l}' = \mathsf{divtwo(l)\ in\ if} - \mathsf{then} - \mathsf{else} -\colon \mathsf{L}_i^{i\le f_k(n)}(\mathsf{Int})$
Let-bind	$n-1 \ge 1;\ \mathsf{l}\colon \mathsf{L}_n(\alpha) \vdash_\Sigma \mathsf{divtwo(l)}\colon \tau^?$
Fun	$\tau^? := \mathsf{L}^{j\le 1}_{\frac{\max_0(n-j)}{2}}(\mathsf{Int})$
Let-body	$n-1 \ge 1;\ \ldots\ \mathsf{hd}\colon \mathsf{Int},\ \mathsf{l}'\colon \mathsf{L}^{j\le 1}_{\frac{\max_0(n-j)}{2}}(\mathsf{Int}) \vdash_\Sigma \mathsf{if} - \mathsf{then} - \mathsf{else} -\colon \mathsf{L}_i^{i\le f_k(n)}(\mathsf{Int})$
True-branch	$n-1 \ge 1;\ \ldots\ \mathsf{hd}\colon \mathsf{Int},\ \mathsf{l}'\colon \mathsf{L}^{j\le 1}_{\frac{\max_0(n-j)}{2}}(\alpha) \vdash_\Sigma \mathsf{log2(l')}\colon \mathsf{L}_i^{i\le f_k(n)}(\mathsf{Int})$
Fun f_k is non-decr. (lemma 1)	$n \ge 2,\ j \le 1,\ i' \le f_k(\frac{\max_0(n-j)}{2}) \vdash i^? = i' \wedge i \le f_k(n)$ $n \ge 2,\ \frac{\max_0(n-j)}{2} \le n \Rightarrow i' \le f_k(\frac{\max_0(n-j)}{2}) \le f_k(n)$
False-branch	$n-1 \ge 1;\ \ldots\ \mathsf{hd}\colon \mathsf{Int},\ \mathsf{l}'\colon \mathsf{L}^{j\le 1}_{\frac{\max_0(n-j)}{2}}(\alpha) \vdash_\Sigma \mathsf{Cons(hd,\ log2(l'))}\colon \mathsf{L}_i^{i\le f_k(n)}(\mathsf{Int})$
Fun, Cons Lemma 3	$n \ge 2,\ j \le 1,\ i' \le f_k(\frac{\max_0(n-j)}{2}) \vdash i^? = 1 + i' \wedge i \le f_k(n)$ $n \ge 2,\ 0 \le j \le 1 \Rightarrow i' \le 1 + f_k(\frac{\max_0(n-j)}{2}) \le f_k(n)$

Now we can show that $\{F_k\}_{k\ge 1}$, where $F_k = \{i \mid 0 \le i \le f_k(n)\}$, is a Cauchy sequence in the space of families of $\max_0$-polynomials. We define the distance between functions on $[0,\ +\infty)$ as an L_p-metric with $p = 1$ and weight $w(x) = 1/(x+1)^3$. This metric is given by $d(g_1, g_2) = (\int_0^\infty |g_1 - g_2|^p w(x)\,dx)^{\frac{1}{p}}$. The distance between the k-th and the $k+m$-th family is defined by $d(F_k,\ F_{k+m}) = \max\{d(i, F_{k+m}) \mid 0 \le i \le f_k\}$.

Lemma 4. *For all possitive integers k and m, and real x,*

1. $f_k(x) = f_{k+m}(x)$ *if* $0 \leq x < 2^k$,
2. $f_k(x) \geq f_{k+m}(x)$ *if* $x \geq 2^k$.

Proof. 1. If $x = 0$ then $f_k(x) = f_{k+m}(x) = 0$. On the other hand, if $0 < x < 2^k$, then there is an $l < k-1$ such that $2^l \leq x < 2^{l+1}$ and thus $f_k(x) = f_{k+m}(x) = l$.

2. Case $2^k \leq x < 2^{k+m}$: then there is an $k \leq l \leq k+m-1$ such that $2^l \leq x < 2^{l+1}$. It is easy to prove by induction that for all $l \geq 2$, $2^l/2^k \geq l/k$. Then for $l \geq 2$, $f_k(x) = \frac{k}{2^k}x \geq k\frac{2^l}{2^k} \geq k\frac{l}{k} = l = f_{k+m}(x)$. If $l = 1$ then $k = 1$ (remember that $k \geq 1$) then $2 \leq x < 4$ and $f_k(x) = x/2 \geq 1 = l = f_{k+m}(x)$. Case $x \geq 2^{k+m}$: By induction on m we can prove that for all $k \geq 1$, $m \geq 1$, $k \geq \frac{k+m}{2^m}$. Thus $f_k(x) = \frac{k}{2^k}x \geq \frac{k+m}{2^{k+m}}x = f_{k+m}(x)$.

Now we can calculate $d(f_k, f_{k+m})$:

$$
\begin{aligned}
d(f_k, f_{k+m}) &= \int_0^\infty |f_k(x) - f_{k+m}(x)|w(x)dx \\
&= \{Lemma\ 4\} \\
&\int_{2^k}^\infty (f_k(x) - f_{k+m}(x))\frac{1}{(x+1)^3}\,dx \\
&\leq \{f_{k+m}(x) \leq k\} \\
&\int_{2^k}^\infty (\frac{k}{2^k}x - k)\frac{1}{(x+1)^3}\,dx \\
&= \{\text{Substitution } y = x+1\} \\
&\int_{2^k+1}^\infty (\frac{k}{2^k}(y-1) - k)\frac{1}{y^3}\,dy \\
&= \frac{k}{2^k}\frac{1}{y}|_\infty^{2^k+1} - (\frac{k}{2^k} + k)\frac{1}{2y^2}|_\infty^{2^k+1} \\
&= \frac{k}{2^k(2^k+1)} - \frac{k}{2^{k+1}(2^k+1)^2} - \frac{k}{2(2^k+1)^2}
\end{aligned}
$$

It is easy to see that $d(f_k, f_{k+m}) \to 0$ as $k \to \infty$, that is $\{F_k\}$ forms a Cauchy sequence. It tends to $\lceil \log_2(n) \rceil$, however this limit may not be presented in the space of families of $\max_0$-polynomials. The size dependency $\mathsf{log2}$ is $\lfloor \log_2(n) \rfloor$, however, such a dependency is not expressible in our type system. We have seen that we can get better approximations to this size dependency by making k bigger. Hence, we have shown that in our type system $\mathsf{log2}$ does not have a principal type.

8 Proofs of heap-aware-semantics lemma and soundness in detail

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

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:

$$\frac{c \in \mathsf{Int} \cup \mathsf{Bool}}{s;\, h;\, \mathcal{C} \vdash c \rightsquigarrow c;\, h}\ \textsc{OSCons} \qquad \frac{}{s;\, h;\, \mathcal{C} \vdash \mathsf{z} \rightsquigarrow s(\mathsf{z});\, h}\ \textsc{OSVar}$$

$$\frac{}{s;\, h;\, \mathcal{C} \vdash \mathsf{Nil} \rightsquigarrow \mathtt{NULL};\, h}\ \textsc{OSNil}$$

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

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

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

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

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

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

$$\frac{s;\, h;\, \mathcal{C}[f := ((\mathsf{g}_1, \ldots, \mathsf{g}_{k'},\ \mathsf{z}_1, \ldots, \mathsf{z}_k) \times e_1)] \vdash e_2 \rightsquigarrow v;\, h'}{s;\, h;\, \mathcal{C} \vdash \mathsf{letfun\ f}(\mathsf{g}_1, \ldots, \mathsf{g}_{k'},\ \mathsf{z}_1, \ldots, \mathsf{z}_k) = e_1\ \mathsf{in}\ e_2 \rightsquigarrow v;\, h'}\ \textsc{OSLetFun}$$

$$\frac{\begin{array}{c} s(\mathsf{z}'_1) = v_1 \ \ldots \ s(\mathsf{z}'_k) = v_k \\ \mathcal{C}(\mathsf{f}) = (\mathsf{g}_1, \ldots, \mathsf{g}_{k'},\ \mathsf{z}_1, \ldots, \mathsf{z}_k) \times e_f \\ [\mathsf{z}_1 := v_1, \ldots, \mathsf{z}_k := v_k];\ h;\ \mathcal{C} \vdash e_f[\mathsf{g}_1 := \mathsf{f}_1, \ldots, \mathsf{g}_{k'} := \mathsf{f}_{k'}] \rightsquigarrow v;\ h' \end{array}}{s;\ h;\ \mathcal{C} \vdash \mathsf{f}(\mathsf{f}_1, \ldots, \mathsf{f}_{k'},\ \mathsf{z}'_1, \ldots, \mathsf{z}'_k) \rightsquigarrow v;\ h'}\ \textsc{OSFunApp}$$

Since there is no destructive pattern matchings and assignments on our language, we deal with *benign sharing* of variables [12]. 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*

$$\mathcal{F} : \mathit{Heap} \times \mathit{Val} \longrightarrow \mathcal{P}(\mathit{Loc})$$

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

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

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

8.1 Heap and model-relation lemma

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

Lemma 5 (A program value's footprint is in the heap).
$\mathcal{F}(h, v) \subseteq \mathit{dom}(h)$.

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

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

dom$(h) \neq \emptyset$**:**

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

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

Lemma 6 (Extending a heap does not change the footprints of program values). *If $\ell \notin dom(h)$ and $h' = h[\ell.\mathsf{hd} := v_{\mathsf{hd}},\ \ell.\mathsf{tl} := v_{\mathsf{tl}}]$, for some $v_{\mathsf{hd}},\ v_{\mathsf{tl}}$ then for any $v \neq \ell$ one has $\mathcal{F}(h, v) = \mathcal{F}(h', v)$.*

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

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

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

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

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

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

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

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

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

and the induction assumption one has

$$\mathcal{F}(h|_{dom(h)\setminus\{\ell'\}}, h.\ell'.\mathsf{hd}) = \mathcal{F}(h'|_{dom(h')\setminus\{\ell'\}}, h'.\ell'.\mathsf{hd})$$
$$\mathcal{F}(h|_{dom(h)\setminus\{\ell'\}}, h.\ell'.\mathsf{tl}) = \mathcal{F}(h'|_{dom(h')\setminus\{\ell'\}}, h'.\ell'.\mathsf{tl})$$

So,

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

Lemma 7 (Extending heaps preserves the model relation). *For all heaps h and h', if $h'|_{dom(h)} = h$ then $v \models^{h}_{\tau^\bullet} w$ implies $v \models^{h'}_{\tau^\bullet} w$.*

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

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

$\tau^\bullet = \mathsf{L}^{Q(\bar{i})}_{p(\bar{i})}(\tau^{\bullet\prime})$**:** Cases of v:

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

$v = \ell$**:** By the definition $\ell \models^{h}_{\mathsf{L}^{Q(\bar{i})}_{p(\bar{i})}(\tau^{\bullet\prime})} w_{\mathsf{hd}} :: w_{\mathsf{tl}}$ for some w_{hd} and w_{tl} such that

$$\begin{array}{l}\ell \in dom(h), \\ h.\ell.\mathsf{hd} \models^{h|_{dom(h)\setminus\{\ell\}}}_{\tau^{\bullet\prime}} w_{\mathsf{hd}}, \\ h.\ell.\mathsf{tl} \models^{h|_{dom(h)\setminus\{\ell\}}}_{\mathsf{L}^{Q(\bar{i})}_{p(\bar{i})-1}(\tau^{\bullet\prime})} w_{\mathsf{tl}}\end{array}$$

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

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

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

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

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

Lemma 8 (Values depend only on their footprints).
For v, h, w, and $\tau^\bullet$, the relation $v \models^h_{\tau^\bullet} w$ implies $v \models^{h|_{\mathcal{F}(h,v)}}_{\tau^\bullet} w$.

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

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

$\tau^\bullet = \mathsf{L}^{Q(\bar{i})}_{p(\bar{i})}(\tau^\bullet)$**:** Cases of v:

- $v = \texttt{NULL}$**:** By the definition $\exists\, \bar{i}.Q(\bar{i}) \wedge p(\bar{i}) = 0$ and $w = \texttt{[]}$. Then $v \models^{h|_{\mathcal{F}(h,v)}}_{\tau^\bullet} w$.
- $v = \ell$**:** By the definition $w = w_{\mathsf{hd}} :: w_{\mathsf{tl}}$ for some w_{hd} and w_{tl}, s.t.

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

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

$$\begin{aligned}&\ell \in dom(h),\\ &h.\ell.\mathsf{hd} \models^{h|_{dom(h)\setminus\{\ell\}}|_{\mathcal{F}(h|_{dom(h)\setminus\{\ell\}},h.\ell.\mathsf{hd})}}_{\tau^{\bullet\prime}} w_{\mathsf{hd}},\\ &h.\ell.\mathsf{tl} \models^{h|_{dom(h)\setminus\{\ell\}}|_{\mathcal{F}(h|_{dom(h)\setminus\{\ell\}},h.\ell.\mathsf{tl})}}_{\mathsf{L}^{Q(\bar{i})}_{p(\bar{i})-1}(\tau^{\bullet\prime})} w_{\mathsf{tl}}\end{aligned}$$

Due to $\mathcal{F}(h|_{dom(h)\setminus\{\ell\}}, h.\ell.\mathsf{hd}) \subseteq dom(h) \setminus \{\ell\}$ (lemma 5) we have

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

Similarly $h|_{dom(h)\setminus\{\ell\}}|_{\mathcal{F}(h|_{dom(h)\setminus\{\ell\}},h.\ell.\mathsf{tl})} = h|_{\mathcal{F}(h|_{dom(h)\setminus\{\ell\}},h.\ell.\mathsf{tl})\setminus\{\ell\}}$.
Due to $\ell \in \mathcal{F}(h,\ell)$, and lemma 7 – with $\mathcal{F}(h|_{dom(h)\setminus\{\ell\}}, h.\ell.\mathsf{hd}) \setminus \{\ell\} \subseteq \mathcal{F}(h, h.\ell.\mathsf{hd}) \setminus \{\ell\}$, we have

$$\begin{array}{l} \ell \in dom(h_{\mathcal{F}(h,\ell)}), \\ h|_{\mathcal{F}(h,\ell)}.\ell.\mathsf{hd} \models^{h|_{\mathcal{F}(h,h.\ell.\mathsf{hd})\setminus\{\ell\}}}_{\tau^{\bullet\prime}} w_{\mathsf{hd}}, \\ h|_{\mathcal{F}(h,\ell)}.\ell.\mathsf{tl} \models^{h|_{\mathcal{F}(h,h.\ell.\mathsf{hd})\setminus\{\ell\}}}_{\mathsf{L}^{Q(\bar{i})}_{p(\bar{i})-1}(\tau^{\bullet\prime})} w_{\mathsf{tl}} \end{array}$$

Thus, $\ell \models^{h|_{\mathcal{F}(h,\ell)}}_{\mathsf{L}^{Q(\bar{i})}_{p(\bar{i})}(\tau^{\bullet\prime})} w_{\mathsf{hd}} :: w_{\mathsf{tl}}$. □

Lemma 9 (instantiation preserves model relation).
For `adr`*,* h*,* w*,* $\bar{i}_0$*, s.t.* $Q(\bar{i}_0)$ *holds and* $p(\bar{i}_0) = length_h(\texttt{adr})$*, the relation* $\texttt{adr} \models^h_{\mathsf{L}^{Q(\bar{i})}_{p(\bar{i})}(\tau^\bullet)} w$ *implies* $\texttt{adr} \models^h_{\mathsf{L}_{p(\bar{i}_0)}(\tau^\bullet)} w$.

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

`adr` $= \texttt{NULL}$**:** By the definition $w = \texttt{[]}$. Then, by the definition $\texttt{adr} \models^h_{\mathsf{L}_{p(\bar{i}_0)}(\tau^\bullet)} w$, since $p(\bar{i}_0) = 0 = length_h(\texttt{adr})$.

`adr` $= \ell$**:** By the definition $w = w_{\mathsf{hd}} :: w_{\mathsf{tl}}$ for some w_{hd} and w_{tl}, s.t.

$$\begin{array}{l} \ell \in dom(h), \\ h.\ell.\mathsf{hd} \models^{h|_{dom(h)\setminus\{\ell\}}}_{\tau^{\bullet\prime}} w_{\mathsf{hd}}, \\ h.\ell.\mathsf{tl} \models^{h|_{dom(h)\setminus\{\ell\}}}_{\mathsf{L}^{Q(\bar{i})}_{p(\bar{i})-1}(\tau^{\bullet\prime})} w_{\mathsf{tl}} \end{array}$$

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

$$\begin{array}{l} \ell \in dom(h), \\ h.\ell.\mathsf{tl} \models^{h|_{dom(h)\setminus\{\ell\}}}_{\mathsf{L}^{Q(\bar{i})}_{p(\bar{i}_0)-1}(\tau^{\bullet\prime})} w_{\mathsf{tl}} \end{array}$$

since $p(\bar{i}_0)-1 = length_h(\ell)-1 = length_{h|_{dom(h)\setminus\{\ell\}}}(h.\ell.\mathsf{tl})$. Thus, $\texttt{adr} \models^h_{\mathsf{L}_{p(\bar{i}_0)}(\tau^\bullet)} w_{\mathsf{hd}} :: w_{\mathsf{tl}}$.

Lemma 10 (Equality of the "meanings" of a program value in two heaps follows from the equality of the footprints).
If $h|_{\mathcal{F}(h,v)} = h'|_{\mathcal{F}(h,v)}$ *then* $v \models^h_{\tau^\bullet} w$ *implies* $v \models^{h'}_{\tau^\bullet} w$.

Proof. Assume $v \models^h_{\tau^\bullet} w$. Lemma 8 states that this implies $v \models^{h|_{\mathcal{F}(h,v)}}_{\tau^\bullet} w$. Assuming $h|_{\mathcal{F}(h,v)} = h'|_{\mathcal{F}(h,v)}$ we get $v \models^{h'|_{\mathcal{F}(h,v)}}_{\tau^\bullet} w$. Since $h'|_{dom(h'|_{\mathcal{F}(h,v)})} = h'|_{\mathcal{F}(h,v)}$ we may apply lemma 7, which gives $v \models^{h'}_{\tau^\bullet} w$. □

8.2 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* $\mathtt{adr} \models^{h}_{\mathsf{L}^{Q(\bar{i})}_{p(\bar{i})}(\tau^{\bullet})} w$ *implies that*

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

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

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

length$_h(\mathtt{adr}) > 0$**:** this means that $\mathtt{adr} = \ell$ and, therefore, there exist w_{hd} and w_{tl}, s.t. $w = w_{\mathsf{hd}} : w_{\mathsf{tl}}$ and $h.\ell.\mathsf{hd} \models^{h|_{dom(h)\setminus\{\ell\}}}_{\tau^{\bullet}} w_{\mathsf{hd}}$ and $h.\ell.\mathsf{tl} \models^{h|_{dom(h)\setminus\{\ell\}}}_{\mathsf{L}^{Q(\bar{i})}_{p(\bar{i})-1}(\tau^{\bullet})} w_{\mathsf{tl}}$.

From the definition of *length* it follows that

$$\mathit{length}_h(h.\ell.\mathsf{tl}) = \mathit{length}_{h|_{dom(h)\setminus\{\ell\}}}(\ell) - 1$$

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

8.3 Properties of subtyping

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

Lemma 11 (Arithmetic operations preserving subtyping). *Let* G *be a 1-variable polynomial, such that* $x \geq G(x)$ *(we will need* $G(x) = x - 1$*.) Then* $D(\overline{n}) \vdash \mathsf{L}^{Q(\overline{n},\,\bar{i})}_{p(\overline{n},\,\bar{i})}(\tau) \preccurlyeq \mathsf{L}^{Q'(\overline{n},\,\bar{j})}_{p'(\overline{n},\,\bar{j})}(\tau')$ *implies* $D(\overline{n}) \vdash \mathsf{L}^{Q(\overline{n},\,\bar{i})}_{G(p(\overline{n},\,\bar{i}))}(\tau) \preccurlyeq \mathsf{L}^{Q'(\overline{n},\,\bar{j})}_{G(p'(\overline{n},\,\bar{j}))}(\tau')$.

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

Lemma 12 (Subtyping preserves the model relation). *Let* $\vdash \tau^{\bullet} \preccurlyeq \tau^{\bullet\prime}$*. Then* $v \models^{h}_{\tau^{\bullet}} w$ *implies* $v \models^{h}_{\tau^{\bullet\prime}} w$*.*

Proof. Induction over the relation $\models$.

$v = c$: Then $\tau^\bullet$ and $\tau^{\bullet\prime}$ are both either Int or Bool, and $v \models^h_{(\mathsf{Int}\cup\mathsf{Bool})} w$ is exactly $v \models^h_{\mathsf{Int}\cup\mathsf{Bool}} w$.

$v = \mathtt{NULL}$: Then $\tau^\bullet = \mathsf{L}^{Q(\bar{i}_1)}_{p(\bar{i}_1)}(\tau^{\bullet\prime\prime})$, and $\tau^{\bullet\prime} = \mathsf{L}^{Q'(\bar{j}_1)}_{p'(\bar{j}_1)}(\tau^{\bullet\prime\prime\prime})$ for some $\bar{i}_1$, $\bar{j}_1$. 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. $Q'(\bar{j}^0_1)$ and $p'(\bar{j}^0) = p(\bar{i}^0) = 0$. Therefore $\mathtt{NULL} \models^h_{\mathsf{L}^{Q'(\bar{j}_1)}_{p'(\bar{j}_1)}(\tau^{\bullet\prime})} [\,]$.

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

$$h.\ell.\mathsf{hd} \models^{h|_{dom(h)\setminus\{\ell\}}}_{\tau^{\bullet\prime\prime}} w_{\mathsf{hd}},$$
$$h.\ell.\mathsf{tl} \models^{h|_{dom(h)\setminus\{\ell\}}}_{\mathsf{L}^{Q(\bar{i}_1)}_{p(\bar{i}_1)-1}(\tau^{\bullet\prime\prime})} w_{\mathsf{tl}}$$

for some w_{hd}, w_{tl}. By the induction assumption (we need the lemma 11 to apply it)

$$h.\ell.\mathsf{hd} \models^{h|_{dom(h)\setminus\{\ell\}}}_{\tau^{\bullet\prime\prime\prime}} w_{\mathsf{hd}},$$
$$h.\ell.\mathsf{tl} \models^{h|_{dom(h)\setminus\{\ell\}}}_{\mathsf{L}^{Q'(\bar{j}_1)}_{p'(\bar{j}_1)-1}(\tau^{\bullet\prime\prime\prime})} w_{\mathsf{tl}}$$

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

8.4 Soundness statement

Theorem (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*

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

the following implication holds:

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

Proof. For the sake of convenience we will denote $\eta(\epsilon(\tau))$ via $\tau_{\eta\epsilon}$, $\eta(\epsilon(\Gamma))$ via $\Gamma_{\eta\epsilon}$ and $D(\epsilon(\overline{n}))$ 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 $TV(\tau) \subseteq TV(\Gamma)$, $SV(\tau) \subseteq SV(\Gamma)$, since $D,\, \Gamma \vdash_\Sigma e : \tau$ is a node in the derivation tree for some function body. We fix a valuation $\epsilon \in SV(\Gamma \cup D) \to \mathcal{R}$, a type instantiation $\eta \in TV(\Gamma) \to \tau^\bullet$, such that the assumptions of the lemma hold.

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

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

OSNull: In this case $v = \texttt{NULL}$, and according to the definition of the model relation we have $\texttt{NULL} \models^h_{\mathsf{L}_0(\tau'_{\eta\epsilon})} \texttt{[]}$ for τ' from the typing rule. Now we use the side condition $D \vdash \mathsf{L}_0(\tau') \preccurlyeq \tau$ and the fact that D_ϵ holds to apply the lemma 12 and obtain $\texttt{NULL} \models^h_\tau \texttt{[]}$ with $h = h'$.

OSVar: In this case $v = s(\mathsf{x})$. From $\mathit{Valid}_{\mathsf{store}}(\mathit{dom}(s), \Gamma' \cup (x : \tau')_{\eta\epsilon}, h, s)$ it follows that $s(\mathsf{x}) \models^h_{\tau'} w$ for some w. We use the side condition $D \vdash \tau' \preccurlyeq \tau$ and D_ϵ to apply the lemma 12. We obtain $v \models^h_\tau w$, with $h = h'$.

OSCons: In this case $e = \mathsf{Cons}(\mathsf{hd}, \mathsf{tl})$ and Γ is "Γ', $\mathsf{hd} : \tau_1, \mathsf{tl} : \mathsf{L}^{Q(\overline{n},\overline{i})}_{p(\overline{n},\overline{i})}(\tau_2)$" for some Γ', hd, tl, Q, p, τ_1 and τ_2. Since $\mathit{Valid}_{\mathsf{store}}(\mathit{dom}(s), \Gamma_{\eta\epsilon}, s, h)$ there exist w_{hd} and w_{tl} such that $s(\mathsf{hd}) \models^h_{\tau_{1\,\eta\epsilon}} w_{\mathsf{hd}}$ and $s(\mathsf{tl}) \models^h_{\mathsf{L}^{Q(\epsilon(\overline{n}),\overline{i})}_{p(\epsilon(\overline{n}),\overline{i})}(\tau_{2\,\eta\epsilon})} w_{\mathsf{tl}}$. From the operational semantics judgement we have that $v = \ell$ for some location $\ell \notin \mathit{dom}(h)$, and $h' = h[\ell.\mathsf{hd} := s(\mathsf{hd}),\ \ell.\mathsf{tl} := s(\mathsf{tl})]$. Therefore, $h'.\ell.\mathsf{hd} \models^h_{\tau_{1\,\eta\epsilon}} w_{\mathsf{hd}}$ and $h'.\ell.\mathsf{tl} \models^h_{\mathsf{L}^{Q(\epsilon(\overline{n}),\overline{i})}_{p(\epsilon(\overline{n}),\overline{i})}(\tau_{2\,\eta\epsilon})} w_{\mathsf{tl}}$ also hold. It is easy to see that $h = h'|_{\mathit{dom}(h')\setminus\{\ell\}}$.
Thus,

$$h'.\ell.\mathsf{hd} \models^{h'|_{\mathit{dom}(h')\setminus\{\ell\}}}_{\tau_{1\,\eta\epsilon}} w_{\mathsf{hd}}$$
$$h'.\ell.\mathsf{tl} \models^{h'|_{\mathit{dom}(h')\setminus\{\ell\}}}_{\mathsf{L}^{Q(\epsilon(\overline{n}),\overline{i})}_{p(\epsilon(\overline{n}),\overline{i})}(\tau_{2\,\eta\epsilon})} w_{\mathsf{tl}}$$

Applying $D \vdash \tau_1 \preccurlyeq \tau_2$ and D_ϵ in lemma 12 gives $\ell \models^{h'}_{\mathsf{L}^{Q(\epsilon(\overline{n}),\overline{i})}_{p(\epsilon(\overline{n}),\overline{i})+1}(\tau_{2\,\eta\epsilon})} w_{\mathsf{hd}} :: w_{\mathsf{tl}}$. We use the side condition

$$D \vdash \mathsf{L}^Q_{p+1}(\tau') \preccurlyeq \tau$$

to apply the lemma 12 we obtain $v \models^{h'}_\tau w_{\mathsf{hd}} :: w_{\mathsf{tl}}$.

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

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

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

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

OSLetFun: The result follows from the induction hypothesis for

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

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

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

for some τ'. Applying the induction hypothesis to the first branch gives $\mathit{Valid}_{\mathsf{store}}(\mathit{dom}(s), D, \Gamma_{\eta\epsilon}, s, h) \implies \mathit{Valid}_{\mathsf{val}}(v_1, \tau'_{\eta\epsilon}, h_1)$.
Now apply the induction hypothesis to the second branch to get

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

Fix some $\mathsf{z}' \in \mathit{dom}(s[\mathsf{z} := v_1])$. If $\mathsf{z}' = \mathsf{z}$, then $\mathit{Valid}_{\mathsf{val}}(v_1, \tau'_{\eta\epsilon}, h_1)$ implies $\mathit{Valid}_{\mathsf{val}}(s[\mathsf{z} := v_1](\mathsf{z}), \tau'_{\eta\epsilon}, h_1)$. If $\mathsf{z}' \neq \mathsf{z}$, then $s[\mathsf{z} := v_1](\mathsf{z}') = s(\mathsf{z}')$. Sharing of data structures in the heap is benign (no destructive pattern matching and assignments), hence $h|_{\mathcal{F}(h,s(\mathsf{z}'))} = h_1|_{\mathcal{F}(h,s(\mathsf{z}'))}$. Applying lemma 10 we have that $s(\mathsf{z}') \models^h_{\Gamma_{\eta\epsilon}(\mathsf{z}')} w'_z$ implies $s(\mathsf{z}') \models^{h_1}_{\Gamma_{\eta\epsilon}(\mathsf{z}')} w'_z$ implies $s[\mathsf{z} := v_1](\mathsf{z}') \models^{h_1}_{\Gamma_{\eta\epsilon}(\mathsf{z})} w_{z'}$ and thus $\mathit{Valid}_{\mathsf{val}}(s[\mathsf{z} := v_1](\mathsf{z}'), \Gamma_{\eta\epsilon}(\mathsf{z}'), h_1)$. Hence, $\mathit{Valid}_{\mathsf{store}}(\mathit{dom}(s[\mathsf{z} := v_1]), \Gamma_{\eta\epsilon} \cup \{\mathsf{z}\colon \tau'_{\eta\epsilon}\}, s[\mathsf{z} := v_1], h_1)$. To apply the induction assumption we use $\mathit{dom}(\Gamma, \mathsf{z}\colon \tau') = \mathit{dom}(\Gamma) \cup \{\mathsf{z}\} = \mathit{dom}(s) \cup \{\mathsf{z}\} = \mathit{dom}(s[\mathsf{z} := v_1])$.

OSMatch-Nil: In this case $e = \mathsf{match}\ \mathsf{x}\ \mathsf{with} \mid \mathsf{Nil} \Rightarrow e_1 \mid \mathsf{Cons}(\mathsf{hd}, \mathsf{tl}) \Rightarrow e_2$ for some x, hd, tl, e_1, and e_2. The typing context has the form $\Gamma = \Gamma' \cup \{\mathsf{x}\colon \mathsf{L}^Q_p(\tau')\}$ for some Γ', τ', Q, p. The operational-semantics derivation gives $s(\mathsf{x}) = \mathtt{NULL}$, hence validity for $s(\mathsf{x})$ 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 $\overline{n}^i$ and extend ϵ to ϵ' with $\epsilon'(\overline{n}^i) := \bar{i}_0$. Therefore, $s(\mathsf{x}) \models^h_{\mathsf{L}_{p_{\epsilon'}}(\tau')}$ []. This means that $\mathit{Valid}_{\mathsf{store}}(\mathit{dom}(s), \Gamma'_{\eta\epsilon'}, \mathsf{x}\colon \mathsf{L}_{p_{\epsilon'}}(\tau'), s, h)$. Using lemma 9, we can apply the the induction hypothesis with $D_{\epsilon'} \wedge Q_{\epsilon'} \wedge p_{\epsilon'} = 0$; $\Gamma, \mathsf{x}\colon \mathsf{L}_{p_{\epsilon'}}(\tau') \vdash_\Sigma e\colon \tau$ to obtain $\mathit{Valid}_{\mathsf{val}}(v, \tau_{\eta\epsilon}, h')$, taking into account that τ does don have size variables $\overline{n}^i$ and, hence, $\tau_{\eta\epsilon} = \tau_{\eta\epsilon'}$.

OSMatch-Cons: In this case $e = \mathsf{match}\ \mathsf{x}\ \mathsf{with} \mid \mathsf{Nil} \Rightarrow e_1 \mid \mathsf{Cons}(\mathsf{hd}, \mathsf{tl}) \Rightarrow e_2$ for some x, hd, tl, e_1, e_2. The typing context has the form $\Gamma = \Gamma' \cup \{x\colon \mathsf{L}^Q_p(\tau')\}$ for some Γ', τ', Q, p. From the operational semantics we know that $h.s(\mathsf{x}).\mathsf{hd} = v_{\mathsf{hd}}$ and $h.s(\mathsf{x}).v_{\mathsf{tl}}$ for some v_{hd} and v_{tl}, that is $s(x) \neq \mathtt{NULL}$. Due to validity of $s(\mathsf{x})$ and the lemma 31 we have that there exists $\bar{i}_0$, such that $Q_\epsilon(\bar{i}_0)$ holds and $p_\epsilon(\bar{i}_0) = \mathit{length}_h(s(\mathsf{x})) \geq 1$. We introduce the new size variables $\overline{n}^i$ and extend ϵ to ϵ' with $\epsilon'(\overline{n}^i) := \bar{i}_0$. Therefore, due to lemma 9, $s(\mathsf{x}) \models^h_{\mathsf{L}_{p_{\epsilon'}}(\tau'_{\eta\epsilon'})}$ $w_{\mathsf{hd}} : w_{\mathsf{tl}}$ for the corresponding w_{hd} and w_{tl}.lemma 9
From the validity $s(\mathsf{x}) \models^h_{\mathsf{L}_{p_{\epsilon'}}(\tau'_{\eta\epsilon'})} w_{\mathsf{hd}} : w_{\mathsf{tl}}$ the validities of v_{hd} and v_{tl} follows: $v_{\mathsf{hd}} \models^h_{\tau'_{\eta\epsilon'}} w_{\mathsf{hd}}$, $v_{\mathsf{tl}} \models^h_{(\mathsf{L}_{p_{\epsilon'}-1}(\tau'))_{\eta\epsilon}} w_{\mathsf{tl}}$.
From $\mathit{Valid}_{\mathsf{store}}(\mathit{dom}(s), \Gamma_{\eta\epsilon}, s, h)$ and the results above we obtain

$$\mathit{Valid}_{\mathsf{store}}(\mathit{dom}(s'), \Gamma_{\eta\epsilon'}, \mathsf{x}\colon \mathsf{L}_{p_{\epsilon'}}(\tau'_{\eta\epsilon'}), \mathsf{hd}\colon \tau'_{\eta\epsilon'}, \mathsf{tl}\colon \mathsf{L}_{p_{\epsilon'}-1}(\tau')_{\eta\epsilon'}, s', h)$$

where $s' = s[\mathsf{hd} := v_{\mathsf{hd}}][\mathsf{tl} := v_{\mathsf{tl}}]$. It is easy to see that
$\mathit{dom}(s') = \mathit{dom}(s) \cup \{\mathsf{hd}, \mathsf{tl}\} = \mathit{dom}(\Gamma) \cup \{\mathsf{hd}, \mathsf{tl}\} = \mathit{dom}(\Gamma, \mathsf{x}\colon \mathsf{L}_{p_{\epsilon'}}(\tau'_{\eta\epsilon'}), \mathsf{hd}\colon \tau'_{\eta\epsilon'}, \mathsf{tl}\colon \mathsf{L}_{p_{\epsilon'}-1}(\tau'_{\eta\epsilon'}))$
From the typing derivation of e we obtain that

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

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

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

Now from the induction hypothesis we have

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

OSFun: We prove soundness of the general FunnApp-rule given in Section 3.1. We want to apply the induction assumption to

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

In the typing judgement under consideration we have that Γ is Γ', $\mathsf{x}_1 : \tau_1, \ldots, \mathsf{x}_k : \tau_k$ for some Γ'. Since this 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, $y_1 : \tau^\circ, \ldots, y_k : \tau_k^\circ \vdash_\Sigma e_f : \tau_0$. Trivially, the domains of the frame store $[y_1 := v_1, \ldots, y_k := v_k]$ and the context $y_1 : \tau^\circ, \ldots, y_k : \tau_k^\circ$ coincide. Let σ be the instantiation of the size variables of the formal parameters with size annotations such that assumptions of the lemma holds: $D \vdash \tau_l \preccurlyeq \sigma(\tau_l^\circ)\tau$, for $1 \leq l \leq k$, and $D \vdash \sigma(\tau_0) \preccurlyeq \tau$. Let σ is extended to types as defined in Section 3.
Define the instantiations and valuation for the formal parameters: $\eta'(\alpha) := \eta(\sigma(\alpha))$ and $\epsilon'(n) = \epsilon(\sigma(n))$ where α and n are type and size variables of the types of the formal parameters.
Consider the Letfun-construct with the definition of the body of e. We use this definition for the induction step, in the pair with the function-call rule of the operational semantics. The predicate True ("no conditions") holds trivially on ϵ'. So, we may apply induction and obtain

$$\mathit{Valid}_{\mathsf{store}}((y_1, \ldots\ y_k), (y_1 : \tau^\circ_{1\,\eta'\epsilon'}, \ldots,\ y_k : \tau^\circ_{k,\ \eta'\epsilon'}), [y_1 := v_1, \ldots, y_k := v_k], h)$$
$$\implies \mathit{Valid}_{\mathsf{val}}(v, \tau'_{\eta'\epsilon'}, h')$$

We want to show that the l.h.s of the implication holds. From
$\mathit{Valid}_{\mathsf{store}}(\mathit{dom}(s), (\Gamma' \cup \mathsf{x}_1 : \tau_1, \ldots, \mathsf{x}_l : \tau_l)_{\eta\epsilon}, s, h)$
it follows that for all $1 \leq l \leq k$ we have $v_l \models^h_{\tau_{l\,\eta\epsilon}} w_l$ for some w_l. Apply lemma 12 (subtyping preserves model relation) to the subtyping $D_\epsilon \vdash \eta(\epsilon(\tau_l)) \preccurlyeq \eta(\epsilon(\sigma(\tau_l^\circ))$ the definition of η', ϵ', from which follows $\eta(\epsilon(\sigma(\tau_l^\circ)) = \tau^\circ_{l\,\eta'\epsilon'}$. We obtain $v_l \models^h_{\tau^\circ_{l\,\eta'\epsilon'}} w_l$.
So, the l.h.s of the implication in the induction assumption holds, and therefore, one obtains $\mathit{Valid}_{\mathsf{val}}(v, \tau_{0\,\eta'\epsilon'}, h')$. Again, due to the subtyping $D_\epsilon \vdash \eta(\epsilon(\sigma(\tau_0) \preccurlyeq \eta(\epsilon(\tau))$, the definition of η' and ϵ', and lemma 12, we have $\mathit{Valid}_{\mathsf{val}}(v, \tau_{\eta\epsilon}, h')$.

∃-I: Follows from the lemma 31 and 9.

□

9 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: `x(1)` $:= n$, `x(2)` $:= m$, `x(3)` $:= j$, `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 $p(\mathsf{x}) \leq 0$:

```
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 `x(1)` ≥ 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 the 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 $p(\overline{n}) + i$, where $0 \leq i \leq \delta(\overline{n})$ with quadratic p and δ.