# Automating sized-type inference for complexity analysis

Martin Avanzini, Ugo Dal Lago

## ▶ To cite this version:

Martin Avanzini, Ugo Dal Lago. Automating sized-type inference for complexity analysis. Proceedings of the ACM on Programming Languages, ACM, 2017, 1 (ICFP), pp.1 - 29. 10.1145/3110287 . hal-01639200

**HAL Id: hal-01639200**

**https://hal.inria.fr/hal-01639200**

Submitted on 20 Nov 2017

# Automating Sized-Type Inference for Complexity Analysis

MARTIN AVANZINI, University of Innsbruck, Austria
UGO DAL LAGO, University of Bologna, Italy & INRIA Sophia Antipolis, France

This paper introduces a new methodology for the complexity analysis of higher-order functional programs, which is based on three ingredients: a powerful type system for size analysis and a sound type inference procedure for it, a ticking monadic transformation, and constraint solving. Noticeably, the presented methodology can be fully automated, and is able to analyse a series of examples which cannot be handled by most competitor methodologies. This is possible due to the choice of adopting an abstract index language and index polymorphism at higher ranks. A prototype implementation is available.

CCS Concepts: • **Theory of computation → Program analysis**; *Automated reasoning*; *Equational logic and rewriting*; • **Software and its engineering → Functional languages**;

**ACM Reference format:**
Martin Avanzini and Ugo Dal Lago. 2017. Automating Sized-Type Inference for Complexity Analysis. *Proc. ACM Program. Lang.* 1, 1, Article 43 (September 2017), 28 pages.
https://doi.org/10.1145/3110287

## 1 INTRODUCTION

Programs can be incorrect for very different reasons. Modern compilers are able to detect many syntactic errors, including type errors. When the errors are semantic, namely when the program is well-formed but does not compute what it should, traditional static analysis methodologies like abstract interpretation or model checking could be of help. When a program is functionally correct but performs quite poorly in terms of space and runtime behaviour, even *defining* the property of interest is very hard. If the units of measurement in which program performances are measured are close to the physical ones, the problem can only be solved if the underlying architecture is known, due to the many transformation and optimisation layers which are applied to programs. One then obtains WCET techniques [Wilhelm et al. 2008], which indeed need to deal with how much machine instructions cost when executed by modern architectures (including caches, pipelining, etc.), a task which is becoming even harder with the current trend towards multicore architectures.

As an alternative, one can analyse the *abstract* complexity of programs. As an example, one can take the number of evaluation steps to normal form, as a measure of the underlying program's execution time. This can be accurate if the actual time complexity *of each instruction* is kept low, and has the advantage of being independent from the specific hardware platform executing the program at hand, which only needs to be analysed *once*. A variety of verification techniques have indeed been defined along these lines, from type systems to program logics, to abstract interpretation, see [Aspinall et al. 2007; Albert et al. 2013; Sinn et al. 2014; Hoffmann et al. 2017].

If we restrict our attention to higher-order functional programs, however, the literature becomes sparser. There seems to be a trade-off between allowing the user full access to the expressive power of modern, higher-order programming languages, and the fact that higher-order parameter passing is a mechanism which intrinsically poses problems to complexity analysis: how big is a certain (closure representation of a) higher-order parameter? If we focus our attention on automatic techniques for the complexity analysis of higher-order programs, the literature only provides very

2017. 2475-1421/2017/9-ART43
https://doi.org/10.1145/3110287

few proposals [Vasconcelos et al. 2008; Avanzini et al. 2015; Hoffmann et al. 2017], which we will discuss in Section 2 below.

One successful approach to automatic verification of *termination* properties of higher-order functional programs is based on *sized types* [Hughes et al. 1996], and has been shown to be quite robust [Barthe et al. 2008]. In sized types, a type carries not only some information about the *kind* of each object, but also about its *size*, hence the name. This information is then exploited when requiring that recursive calls are done on arguments of *strictly smaller* size, thus enforcing termination. Estimating the size of intermediate results is also a crucial aspect of complexity analysis, but up to now the only attempt of using sized types for complexity analysis is due to Vasconcelos et al. [2008], and confined to space complexity. If one wants to be sound for time analysis, size types need to be further refined, e.g., by turning them into linear dependently types [Dal Lago et al. 2011].

In this paper, we take a fresh look at sized types by introducing a new type system which is substantially more expressive than the traditional one. This is possible due to the presence of *arbitrary rank index polymorphism*: functions that take functions as their argument can be polymorphic in their size annotation. The introduced system is then proved to be a sound methodology for *size* analysis, and a type inference algorithm is given and proved sound and relatively complete. Finally, the type system is shown to be amenable to time complexity analysis by a ticking monadic transformation. A prototype implementation is available, see below for more details. More specifically, this paper's contributions can be summarized as follows:

- We show that size types can be generalised so as to encompass a notion of index polymorphism, in which (higher-order subtypes of) the underlying type can be universally quantified. This allows for a more flexible treatment of higher-order functions. Noticeably, this is shown to preserve soundness (i.e., subject reduction), the minimal property one expects from such a type system. On the one hand, this is enough to be sure that types reflect the size of the underlying program. On the other hand, termination is not enforced anymore by the type system, contrarily to, e.g., the system of Hughes et al. [1996]. In particular, we do not require that recursive calls are made on arguments of smaller size. All this is formulated on a language of applicative programs, introduced in Section 4, and will be developed in Section 5. Nameless functions (i.e., $\lambda$-abstractions) are not considered for brevity, as these can be easily lifted to the top-level.
- The type inference problem is shown to be (relatively) decidable by giving in Section 6 an algorithm which, given a program, produces in output candidate types for the program, together with a set of integer index inequalities which need to be checked for satisfiability. This style of results is quite common in similar kinds of type systems. What is uncommon though, at least in the context of sized types, is that we do not restrict ourselves to a particular algebra in which sizes are expressed. Indeed, many of the more advanced sized type systems are restricted to the successor algebra [Blanqui et al. 2005; Abel et al. 2016]. This is often sufficient in the context of termination analysis, where one is interested in determining which recursion parameters decrease. Here, the programs runtime will be expressed in this algebra, and thus a more expressive algebra is required.
- The polymorphic sized type system, by itself, does not guarantee any complexity-theoretic property on the typed program, except for the *size* of the output being bounded by a function on the size of the input, itself readable from the type. Complexity analysis of a program $\mathsf{P}$ can however be seen as a size analysis of another program $\hat{\mathsf{P}}$ which computes not only $\mathsf{P}$, but its complexity. This transformation, called the *ticking transformation*, has already been studied in similar settings [Danner et al. 2015], but this study has never been automated. The ticking transformation is formally introduced in Section 7.

· Contrarily to many papers from the literature, we spent considerable efforts on devising a system that is susceptible to automation with current technology. Moreover, we have taken care not only of constraint *inference*, but also of constraint *solving*. To demonstrate the feasibility of our approach, we have built a prototype which implements type inference, resulting in a set of constraints. To deal with the resulting constraints, we have also built a constraint solver on top of state-of-the-art SMT solvers. All this, together with some experimental results, are described in detail in Section 8.

An extended version with more details is available as supplementary material.

## 2 A BIRD EYE'S VIEW ON INDEX-POLYMORPHIC SIZED TYPES

In this section, we will motivate the design choices we made when defining our type system through some examples. This can also be taken as a gentle introduction to the system for those readers which are familiar with functional programming and type theory. Our type system shares quite some similarities with the seminal type system introduced by Hughes et al. [1996] and similar ones [Vasconcelos et al. 2008; Barthe et al. 2008], but we try to keep presentation as self-contained as possible.

*Basics.* We work with functional programs over a fixed set of inductive datatypes, e.g. $\mathtt{Nat}$ for natural numbers and $\mathtt{List}\ \alpha$ for lists over elements of type $\alpha$. Each such datatype is associated with a set of typed *constructors*, below we will use the constructors $0 :: \mathtt{Nat}$, $\mathsf{Succ} :: \mathtt{Nat} \to \mathtt{Nat}$ for naturals, and the constructor $[\,] :: \forall\alpha.\ \mathtt{List}\ \alpha$ and the infix constructor $(:) :: \forall\alpha.\ \alpha \to \mathtt{List}\ \alpha \to \mathtt{List}\ \alpha$ for lists. Sized types refine each such datatype into a *family* of datatypes indexed by natural numbers, their *size*. E.g., to $\mathtt{Nat}$ and $\mathtt{List}\ \alpha$ we associate the families $\mathtt{Nat}_0, \mathtt{Nat}_1, \mathtt{Nat}_2, \ldots$ and $\mathtt{List}_0\ \alpha, \mathtt{List}_1\ \alpha, \mathtt{List}_2\ \alpha, \ldots$, respectively. An indexed datatype such as $\mathtt{List}_n\ \mathtt{Nat}_m$ then represents lists of length $n$, over naturals of size $m$.

A function $\mathsf{f}$ will then be given a polymorphic type $\forall\vec{\alpha}.\ \forall\vec{i}.\ \tau \to \zeta$. Whereas the variables $\vec{\alpha}$ range over types, the variables $\vec{i}$ range over sizes. Datatypes occurring in the types $\tau$ and $\zeta$ will be indexed by expressions over the variables $\vec{i}$. E.g., the append function can be attributed the sized type $\forall\alpha.\ \forall ij.\ \mathtt{List}_i\ \alpha \to \mathtt{List}_j\ \alpha \to \mathtt{List}_{i+j}\ \alpha$.

Soundness of our type-system will guarantee that when append is applied to lists of length $n$ and $m$ respectively, it will yield a list of size $n + m$, or possibly diverge. In particular, our type system is not meant to guarantee termination, and complexity analysis will be done via the aforementioned ticking transformation, to be described later. As customary in sized types, we will also integrate a subtyping relation $\tau \sqsubseteq \zeta$ into our system, allowing us to relax size annotations to less precise ones. This flexibility is necessary to treat conditionals where the branches are attributed different sizes, or, to treat higher-order combinators which are used in multiple contexts.

Our type system, compared to those from the literature, has its main novelty in polymorphism, but is also different in some key aspects, addressing intensionality but also practical considerations towards type inference. In the following, we shortly discuss the main differences.

*Canonical Polymorphic Types.* We allow polymorphism over size expressions, but put some syntactic restrictions on function declarations: In essence, we disallow non-variable size annotations directly to the left of an arrow, and furthermore, all these variables must be pairwise distinct. We call such types canonical. The first restriction dictates that e.g. $\mathtt{half} :: \forall i.\mathtt{Nat}_{2\cdot i} \to \mathtt{Nat}_i$ has to be written as $\mathtt{half} :: \forall i.\mathtt{Nat}_i \to \mathtt{Nat}_{i/2}$. The second restriction prohibits e.g. the type declaration $\mathsf{f} :: \forall i.\mathtt{Nat}_i \to \mathtt{Nat}_i \to \tau$, rather, we have to declare $\mathsf{f}$ with a more general type $\forall ij.\mathtt{Nat}_i \to \mathtt{Nat}_j \to \tau'$. The two restrictions considerably simplify the inference machinery when dealing with pattern matching, and pave the way towards automation. Instead of a complicated

```
1 rev :: ∀α. ∀ij. List_i α → List_j α → List_{i+j} α
2 rev []       ys = ys
3 rev (x : xs) ys = rev xs (x : ys)
4 reverse :: ∀α. ∀i. List_i α → List_i α
5 reverse xs = rev xs []
```

Fig. 1. Sized type annotated tail-recursive list reversal function.

unification based mechanism, a matching mechanism suffices. Unlike in [Hughes et al. 1996], where indices are formed over naturals and addition, we keep the index language abstract. This allows for more flexibility, and ultimately we can capture more programs. Indeed, having the freedom of not adopting a fixed index language is known to lead towards completeness [Dal Lago et al. 2011].

*Polymorphic Recursion over Sizes.* Type inference in functional programming languages, such as `Haskell` or `OCaml`, is restricted to parametric polymorphism in the form of *let-polymorphism*. Recursive definitions are checked under a monotype, thus, types cannot change between recursive calls. Recursive functions that require full parametric polymorphism [Mycroft et al. 1984] have to be annotated in general, as type inference is undecidable in this setting.

Let-polymorphism poses a significant restriction in our context, because sized types considerably refine upon simple types. Consider for instance the usual tail-recursive definition of list reversal depicted in Figure 1. With respect to the annotated sized types, in the body of the auxiliary function `rev` defined on line 3, the type of the second argument to `rev` will change from $\mathtt{List}_j\ \alpha$ (the assumed type of $ys$) to $\mathtt{List}_{j+1}\ \alpha$ (the inferred type of $x : ys$). Consequently, `rev` is not typeable under a monomorphic sized type. Thus, to handle even such very simple functions, we will have to overcome let-polymorphism, on the layer of size annotations. To this end, conceptually we allow also recursive calls to be given a type polymorphic over size variables. This is more general than the typing rule for recursive definitions found in more traditional systems [Hughes et al. 1996; Barthe et al. 2008].

*Higher-ranked Polymorphism over Sizes.* In order to remain decidable, classical type inference systems work on polymorphic types in *prenex form* $\forall\vec{\alpha}.\tau$, where $\tau$ is quantifier free. In our context, it is often not enough to give a combinator a type in prenex form, in particular when the combinator uses a functional argument more than once. All uses of the functional argument have to be given then *the same* type. In the context of sized types, this means that functional arguments can be applied only to expressions whose attributed size equals. This happens for instance in recursive combinators, but also non-recursive ones such as the function $\mathtt{twice}\ f\ x = f\ (f\ x)$. A strong type-system would allow us to type the expression `twice Succ` with a sized type $\mathtt{Nat}_c \to \mathtt{Nat}_{c+2}$. A (specialised) type in prenex form for `twice`, such as

$$\mathtt{twice} :: \forall i.\ (\mathtt{Nat}_i \to \mathtt{Nat}_{i+1}) \to \mathtt{Nat}_i \to \mathtt{Nat}_{i+2}\ ,$$

would immediately yield the mentioned sized type for `twice Succ`. However, we will not be able to type `twice` itself, because the outer occurrence of $f$ would need to be typed as $\mathtt{Nat}_{i+1} \to \mathtt{Nat}_{i+2}$, whereas the type of `twice` dictates that $f$ has type $\mathtt{Nat}_i \to \mathtt{Nat}_{i+1}$.

The way out is to allow polymorphic types of rank *higher than* one when it comes to size variables, i.e. to allow quantification of size variables to the left of an arrow at arbitrary depth. Thus, we can declare

$$\mathtt{twice} :: \forall i.\ (\forall j.\mathtt{Nat}_j \to \mathtt{Nat}_{j+1}) \to \mathtt{Nat}_i \to \mathtt{Nat}_{i+2}\ .$$

As above, this allows us to type the expression `twice Succ` as desired. Moreover, the inner quantifier permits the two occurrences of the variable $f$ in the body of `twice` to take types $\mathtt{Nat}_i \to \mathtt{Nat}_{i+1}$ and $\mathtt{Nat}_{i+1} \to \mathtt{Nat}_{i+2}$ respectively, and thus `twice` is well-typed.

```
1 foldr :: ∀αβ. ∀jkl. (∀i. α → List_i β → List_{i+j} β) → List_k β → List_l α → List_{l·j+k} β
2 foldr f b []       = b
3 foldr f b (x : xs) = f x (foldr f b xs)
4 product :: ∀αβ. ∀ij. List_i α → List_j β → List_{i·j} (α × β)
5 product ms ns = foldr (λ m ps. foldr (λ n. (:) (m,n)) ps ns) [] ms
```

Fig. 2. Sized type annotated program computing the cross-product of two lists.

*A Worked Out Example.* We conclude this section by giving a nontrivial example. The sized type annotated program is given in Figure 2. The function product computes the cross-product $[\,(m, n) \mid m \in ms, n \in ns\,]$ for two given lists $ms$ and $ns$. It is defined in terms of two folds. The inner fold appends, for a fixed element $m$, the list $[\,(m, n) \mid n \in ns\,]$ to an accumulator $ps$, the outer fold traverses this function over all elements $m$ from $ms$.

In a nutshell, checking that a function f is typed correctly amounts to checking that all its defining equations are well-typed, i.e. under the assumption that the variables are typed according to the type declaration of f, the right-hand side of the equation has to be given the corresponding return-type. Of course, all of this has to take pattern matching into account. Let us illustrate this on the recursive equation of foldr given in Line 3 in Figure 2. Throughout the following, we denote by $s : \tau$ that the term $s$ has type $\tau$. To show that the equation is well-typed, let us assume the following types for arguments: $f : \forall i.\ \alpha \to \mathtt{List}_i\ \beta \to \mathtt{List}_{i+j}\ \beta$, $b : \mathtt{List}_k\ \beta$, $x : \alpha$ and $xs : \mathtt{List}_m\ \alpha$ for arbitrary size-indices $j, k, m$. Under these assumptions, the left-hand side has type $\mathtt{List}_{(m+1)\cdot j+k}\ \beta$, taking into account that the recursion parameter $x : xs$ has size $m + 1$. To show that the equation is well-typed, we verify that the right-hand side can be attributed the same sized type.

1. We instantiate the polymorphic type of foldr and derive
$$\mathtt{foldr} : (\forall i.\ \alpha \to \mathtt{List}_i\ \beta \to \mathtt{List}_{i+j}\ \beta) \to \mathtt{List}_k\ \beta \to \mathtt{List}_m\ \alpha \to \mathtt{List}_{m\cdot j+k}\ \beta\ ;$$
2. from this and the above assumptions we get $\mathtt{foldr}\ f\ b\ xs : \mathtt{List}_{m\cdot j+k}\ \beta$;
3. by instantiating the quantified size variable $i$ in the assumed type of $f$ with the index term $m \cdot j + k$ we get $f : \alpha \to \mathtt{List}_{m\cdot j+k}\ \beta \to \mathtt{List}_{(m\cdot j+k)+j}\ \beta$;
4. from the last two steps we finally get $f\ x\ (\mathtt{foldr}\ f\ b\ xs) : \mathtt{List}_{(m+1)\cdot j+k}\ \beta$.

We will not explain the type checking of the remaining equations, but revisit this example in Section 8.

## 3 ON RELATED WORK

Since the first inception in the seminal paper of Hughes et al. [1996], the literature on sized types has grown to a considerable extent. Indeed, various significantly more expressive systems have been introduced, with the main aim to enlarge the class of typable (and thus proved terminating) programs. For instance, Blanqui et al. [2005] introduced a novel sized type system on top of the *calculus of algebraic construction.*

Notably, it has been shown that for size indices over the successor algebra, type checking is decidable [Blanqui et al. 2005]. The system is thus capable of expressing additive relations between sizes. In the context of termination analysis, where one would like to statically detect that a recursion parameter decreases in size, this is sufficient. In this line of research falls also more recent work of Abel et al. [2016], where a novel sized type system for termination analysis on top of $\mathsf{F}_\omega$ is proposed. Noteworthy, this system has been integrated in the dependently typed language Agda.

Type systems related to sized types have been introduced and studied not only in the context of termination analysis, but also for size and complexity analysis of programs. One noticeable example is the series of work by Shkaravska et al. [2009], which aims at size analysis but which is limited to first-order programs. Also Crary et al. [2000] use types, like here, to express the runtime of functions.

However, the system is inherently *semi-automatic*. Related to this is also the work by Danielsson et al. [2008], whose aim is again complexity analysis, but which is not fully automatable and limited to linear bounds. If one's aim is complexity analysis of higher-order functional programs, achieving a form of completeness is indeed possible by linear dependent types [Dal Lago et al. 2011, 2014]. While the front-end of this verification machinery is fully-automatable [Dal Lago et al. 2013], the back-end is definitely not, and this is the reason why this paper should be considered a definite advance over this body of work. Our work is also related to that of ?, which uses a combination of runtime and size analysis to reason about the complexity of functional programs expressed as interaction nets.

Our work draws inspiration from Danner et al. [2015]. In this work, the complexity analysis of higher-order functional programs, defined in a system akin to Gödel's T enriched with inductive types, is studied. A ticking transformation is used to instrument the program with a clock, recurrence relations are then extracted from the ticked version that express the complexity of the input program. Conceptually, our ticking transformation is identical to the one defined by Danner et al., and differs only in details to account for the peculiarities of the language that we are considering. In particular, our simulation theorem, Theorem 7.3, has an analogue in [Danner et al. 2015]. The proof in the present work is however more delicate, as our language admits arbitrary recursion and programs may thus very well diverge. To our best knowledge, no attempts have been made so far to automate solving of the resulting recurrences.

In contrast, Hoffmann et al. refine in a series of works the methodology of Jost et al. [2010] based on *Tarjan's amortised resource analysis*. This lead to the development of RAML [Hoffmann et al. 2012], a fully fledged automated resource analysis tool. Similar to the present work, the analysis is expressed as a type system. Data types are annotated by *potentials*, inference generates a set of linear constraints which are then solved by an external tool. This form of analysis can not only deal with non-linear bounds [Hoffmann et al. 2011], but it also demonstrates that type based systems are relatively stable under language features such as parallelism [Hoffmann et al. 2015] or imperative features [Hoffmann et al. 2015]. In more recent work [Hoffmann et al. 2017], the methodology has been lifted to the higher-order case and RAML can now interface with Inria's OCaml compiler. Noteworthy, some of the peculiarities of this compiler are taken into account. The overall approach is in general incomparable to our methodology. Whilst it seems feasible, our method neither takes amortisation into account nor does our prototype interface with a industrial strength compiler. On the other hand, our system can properly account for closures, whereas inherent to the methodology underlying RAML, closures can only be dealt with in a very restricted form. We return to this point in Section 8 within our experimental assessment.

There are also connections to the work of Avanzini et al. [2015], where a complexity preserving transformation from higher-order to first-order programs is proposed. This transformation works by a form of control-flow guided defunctionalisation. Furthermore, a variety of simplification techniques, such as inlining and narrowing, are employed to make the resulting first-order program susceptible to an automated analysis. The complete procedure has been implemented in the tool HoCA, which relies on the complexity analyser TcT [Avanzini et al. 2016] to analyse the resulting first-order program. Unlike for our system, it is unclear whether the overall method can be used to derive precise bounds.

## 4 APPLICATIVE PROGRAMS AND SIMPLE TYPES

We restrict our attention to a small prototypical, strongly typed functional programming language. For the sake of simplifying presentation, we impose a simple, monomorphic, type system on programs, which does not guarantee anything except a form of type soundness. We will only later in this paper introduce sized types proper. Our theory can be extended straightforwardly to an

ML-style polymorphic type setting. Here, such an extension would only distract from the essentials. Indeed, our implementation (described in Section 8) allows polymorphic function definitions.

*Statics.* Let $\mathcal{B}$ denote a finite set of base types $\mathsf{B}, \mathsf{C}, \ldots$ . *Simple types* are inductively generated from $\mathsf{B} \in \mathcal{B}$:

$$\textbf{(simple types)} \qquad \tau, \rho, \xi ::= \mathsf{B} \mid \tau \times \rho \mid \tau \to \rho\ .$$

We follow the usual convention that $\to$ associates to the right. Let $\mathcal{X}$ denote a countably infinite set of *variables*, ranged over by metavariables like $x$, $y$. Furthermore, let $\mathcal{F}$ and $\mathcal{C}$ denote two disjoint sets of symbols, the set of *functions* and *constructors*, respectively, all pairwise distinct with elements from $\mathcal{X}$. Functions and constructors are denoted in `teletype font`. We keep the convention that functions start with a lower-case letter, whereas constructors start with an upper-case letter. Each symbol $s \in \mathcal{X} \cup \mathcal{F} \cup \mathcal{C}$ has a simple type $\tau$, and when we want to insist on that, we write $s^\tau$ instead of just $s$. Furthermore, each symbol $s^{\tau_1 \to \cdots \to \tau_n \to \rho} \in \mathcal{F} \cup \mathcal{C}$ is associated with a natural number $\mathsf{ar}(s) \leq n$, its *arity*. The set of *terms*, *patterns*, *values* and *data values* over functions $\mathsf{f} \in \mathcal{F}$, constructors $\mathsf{C} \in \mathcal{C}$ and variables $x \in \mathcal{X}$ is inductively generated as follows. Here, each term receives implicitly a type, in Church style. Below, we employ the usual convention that application associates to the left.

$$\begin{array}{lll}
\textbf{(terms)} & s, t ::= x^\tau & \textit{variable} \\
& \quad \mid \mathsf{f}^\tau & \textit{function} \\
& \quad \mid \mathsf{C}^\tau & \textit{constructor} \\
& \quad \mid (s^{\tau \to \rho}\ t^\tau)^\rho & \textit{application} \\
& \quad \mid (s^\tau, t^\rho)^{\tau \times \rho} & \textit{pair constructors} \\
& \quad \mid (\mathsf{let}\ (x^\tau, y^\rho) = s^{\tau \times \rho}\ \mathsf{in}\ t^\xi)^\xi & \textit{pair destructor;} \\
\textbf{(patterns)} & p, q ::= x^\tau \mid \mathsf{C}^{\tau_1 \to \cdots \tau_n \to \mathsf{B}}\ p_1^{\tau_1} \cdots p_n^{\tau_n}\ ; & \\
\textbf{(values)} & u, v ::= \mathsf{C}^{\tau_1 \to \cdots \to \tau_n \to \tau}\ u_1^{\tau_1} \cdots u_n^{\tau_n} & \\
& \quad \mid \mathsf{f}^{\tau_1 \to \cdots \to \tau_m \to \tau_{m+1} \to \tau}\ u_1^{\tau_1} \cdots u_m^{\tau_m} & \\
& \quad \mid (u^\tau, v^\rho)^{\tau \times \rho}\ ; & \\
\textbf{(data values)} & d ::= \mathsf{C}^{\mathsf{B}_1 \to \cdots \to \mathsf{B}_{n+1}}\ d_1 \cdots d_n\ . &
\end{array}$$

The presented operators are all standard, except the pair destructor $\mathsf{let}\ (x, y) = s\ \mathsf{in}\ t$ which binds the variables $x$ and $y$ to the two components of the result of $s$ in $t$. The set of *free variables* $\mathsf{FVar}(s)$ of a term $s$ is defined in the usual way. If $\mathsf{FVar}(s) = \varnothing$, we call $s$ *ground*. A term $s$ is called *linear*, if each variable occurs at most once in $s$. A *substitution* $\theta$ is a finite mapping from variables $x^\tau$ to terms $s^\tau$. The substitution mapping $\vec{x} = x_1, \ldots, x_n$ to $\vec{s} = s_1, \ldots, s_n$, respectively, is indicated with $\{s_1, \ldots, s_n / x_1, \ldots, x_n\}$ or $\{\vec{s}/\vec{x}\}$ for short. The variables $\vec{x}$ are called the *domain* of $\theta$. We denote by $s\theta$ the application of $\theta$ to $s$. Let-bound variables are renamed to avoid variable capture.

A *program* $\mathsf{P}$ over functions $\mathcal{F}$ and constructors $\mathcal{C}$ defines each function $\mathsf{f} \in \mathcal{F}$ through a finite set of *equations* $l^\tau = r^\tau$, where $l$ is of the form $\mathsf{f}\ p_1 \cdots p_{\mathsf{ar}(f)}$. We put the usual restriction on equations that each variable occurs at most once in $l$, i.e. that $l$ is linear, and that the variables of the *right-hand side* $r$ are all included in $l$. To keep the semantics short, we do not impose any order on the equations. Instead, we require that left-hand sides defining $\mathsf{f}$ are all pairwise non-overlapping. This ensures that our programming model is deterministic.

Some remarks are in order before proceeding. As standard in functional programming, only values of base type can be destructed by pattern matching. In a pattern, a constructor always needs to be fully applied. We deliberately disallow the destruction of pairs through pattern matching.

This would unnecessarily complicate some key definitions in later sections. Instead, a dedicated destructor $\mathsf{let}\ (x, y) = s\ \mathsf{in}\ t$ is provided. We also excluded $\lambda$-abstractions from our language, for brevity, as these can always be lifted to the top-level. Similarly, conditionals and case-expressions would not improve upon expressivity.

*Dynamics.* We impose a *call-by-value* semantics on programs P. *Evaluation contexts* are defined according to the following grammar:

$$E ::= \square^{\tau} \mid (E^{\tau \to \rho}\ s^{\tau})^{\rho} \mid (s^{\tau \to \rho}\ E^{\tau})^{\rho} \mid (E^{\tau}, s^{\rho})^{\tau \times \rho} \mid (s^{\tau}, E^{\rho})^{\tau \times \rho} \mid (\mathsf{let}\ (x^{\tau}, y^{\rho}) = E^{\tau \times \rho}\ \mathsf{in}\ s^{\xi})^{\xi} \ .$$

As with terms, type annotations will be omitted from evaluation contexts whenever this does not cause ambiguity. With $E[s^{\tau}]$ we denote the term obtained by replacing the *hole* $\square^{\tau}$ in $E$ by $s^{\tau}$. The one-step *call-by-value* reduction relation $\to_{\mathsf{P}}$, defined over ground terms, is then given as the closure over all evaluation contexts, of the following two rules:

$$\frac{\mathsf{f}\ p_1\ \cdots\ p_n = r \in \mathsf{P}}{(\mathsf{f}\ p_1\ \cdots\ p_n)\{\vec{u}/\vec{x}\} \to_{\mathsf{P}} r\{\vec{u}/\vec{x}\}} \qquad \frac{}{\mathsf{let}\ (x, y) = (u, v)\ \mathsf{in}\ t \to_{\mathsf{P}} t\{u, v/x, y\}}$$

We denote by $\to_{\mathsf{P}}^{*}$ the transitive and reflexive closure, and likewise, $\to_{\mathsf{P}}^{\ell}$ denotes the $\ell$-fold composition of $\to_{\mathsf{P}}$.

Notice that reduction simply gets stuck if pattern matching in the definition of $\mathsf{f}$ is not exhaustive. We did not specify a particular reduction order, e.g., left-to-right or right-to-left. Reduction itself is thus non-deterministic, but this poses no problem since programs are *non-ambiguous*: not only are the results of a computation independent from the reduction order, but also reduction lengths coincide.

Proposition 4.1. *All* normalising reductions *of $s$ have the same length and yield the same result, i.e. if $s \to_{\mathsf{P}}^{m} u$ and $s \to_{\mathcal{R}}^{n} v$ then $m = n$ and $u = v$.*

To define the *runtime-complexity* of P, we assume a single entry point to the program via a *first-order* function $\mathtt{main}^{\mathsf{B}_1 \to \cdots \to \mathsf{B}_k \to \mathsf{B}_n}$, which takes as input data values and also produces a data value as output. The (*worst-case*) *runtime-complexity* of P then measures the reduction length of $\mathtt{main}$ in the sizes of the inputs. Here, the size $|d|$ of a data value is defined as the number of constructor in $d$. Formally, the runtime-complexity function of P is defined as the function $\mathsf{rc}_{\mathsf{P}} : \mathbb{N} \times \cdots \times \mathbb{N} \to \mathbb{N}^{\infty}$:

$$\mathsf{rc}_{\mathsf{P}}(n_1, \ldots, n_k) := \sup\{\ell \mid \exists d_1, \ldots, d_k.\ \mathtt{main}\ d_1\ \cdots\ d_k \to_{\mathsf{P}}^{\ell} s \text{ and } |d_i| \leqslant n_i\} \ .$$

We emphasise that the runtime-complexity function defines a cost model that is invariant to traditional models of computation, e.g., Turing machines [Dal Lago et al. 2009; Avanzini et al. 2010].

## 5 SIZED TYPES AND THEIR SOUNDNESS

This section is devoted to introducing the main object of study of this paper, namely a sized type system for the applicative programs that we introduced in Section 4. We have tried to keep the presentation of the relatively involved underlying concepts as simple as possible.

### 5.1 Indices

As a first step, we make the notion of *size index*, with which we will later annotate data types, precise. Let $\mathcal{G}$ denote a set of first-order function symbols, the *index symbols*. Any symbol $\mathsf{f} \in \mathcal{G}$ is associated with a natural number $\mathsf{ar}(\mathsf{f})$, its *arity*. The set of *index terms* is generated over a countable infinite set of *index variables* $i \in \mathcal{V}$ and index symbols $\mathsf{f} \in \mathcal{G}$.

$$\textbf{(index terms)} \qquad a, b ::= i \mid \mathsf{f}(a_1, \ldots, a_{\mathsf{ar}(\mathsf{f})}) \ .$$

We denote by $\mathsf{Var}(a) \subset \mathcal{V}$ the set of variables occurring in $a$. Substitutions mapping index variables to index terms are called *index substitutions*. With $\vartheta$ we always denote an index substitution. We adopt the notions concerning term substitutions to index substitutions from the previous section.

Throughout this section, $\mathcal{G}$ is kept fixed. Meaning is given to index terms through an *interpretation* $\mathcal{J}$, that maps every $k$-ary $\mathsf{f} \in \mathcal{G}$ to a (total) and *weakly monotonic* function $[\![\mathsf{f}]\!]_{\mathcal{J}} : \mathbb{N}^{\mathsf{ar}(\mathsf{f})} \to \mathbb{N}$. We suppose that $\mathcal{G}$ always contains a constant 0, a unary symbol s, and a binary symbol + which we write in infix notation below. These are always interpreted as zero, the successor function and addition, respectively. Our index language encompasses the one of Hughes et al. [1996], where linear expressions over natural numbers are considered. The interpretation of an index term $a$, under an *assignment* $\alpha : \mathcal{V} \to \mathbb{N}$ and an interpretation $\mathcal{J}$, is defined recursively in the usual way: $[\![i]\!]^{\alpha}_{\mathcal{J}} := \alpha(i)$ and $[\![\mathsf{f}(a_1,\ldots,a_k)]\!]^{\alpha}_{\mathcal{J}} := [\![\mathsf{f}]\!]_{\mathcal{J}}([\![a_1]\!]^{\alpha}_{\mathcal{J}},\ldots,[\![a_k]\!]^{\alpha}_{\mathcal{J}})$. We define $a \leq_{\mathcal{J}} b$ if $[\![a]\!]^{\alpha}_{\mathcal{J}} \leq [\![b]\!]^{\alpha}_{\mathcal{J}}$ holds *for all* assignments $\alpha$. The following lemma collects useful properties of the relation $\leq_{\mathcal{J}}$.

LEMMA 5.1.
1. *The relation $\leq_{\mathcal{J}}$ is reflexive, transitive and closed under substitutions, i.e. $a \leq_{\mathcal{J}} b$ implies $a\vartheta \leq_{\mathcal{J}} b\vartheta$.*
2. *If $a \leq_{\mathcal{J}} b$ then $c\{a/i\} \leq_{\mathcal{J}} c\{b/i\}$ for each index term $c$.*
3. *If $a \leq_{\mathcal{J}} b$ then $a\{0/i\} \leq_{\mathcal{J}} b$.*
4. *If $a \leq_{\mathcal{J}} b$ and $i \notin \mathsf{Var}(a)$ then $a \leq_{\mathcal{J}} b\{c/i\}$ for every index term $c$.*

## 5.2 Sized Types Subtyping and Type Checking

The set of *sized types* is given by annotating occurrences of base types in simple types with index terms $a$, possibly introducing quantification over index variables. More precise, the sets of *(sized) monotypes*, *(sized) polytypes* and *(sized) types* are generated from base types B, index variables $\vec{i}$ and index terms $a$ as follows:

$$\textbf{(monotypes)} \quad \tau,\zeta ::= \mathsf{B}_a \mid \tau \times \zeta \mid \rho \to \tau\,, \quad \textbf{(polytypes)} \quad \sigma ::= \forall\vec{i}.\ \rho \to \tau\,, \quad \textbf{(types)} \quad \rho ::= \tau \mid \sigma\,.$$

Types $\mathsf{B}_a$ are called *indexed base types*. We keep the convention that the arrow binds stronger than quantification. Thus in a polytype $\forall\vec{i}.\ \rho \to \tau$ the variables $\vec{i}$ are bound in $\rho$ and $\tau$. We will sometimes write a monotype $\tau$ as $\forall\epsilon.\ \tau$. This way, every type $\rho$ can given in the form $\forall\vec{i}.\ \tau$. The *skeleton* of a type $\rho$ is the simple type obtained by dropping quantifiers and indices. The sets $\mathsf{FVar}^{+}(\cdot)$ and $\mathsf{FVar}^{-}(\cdot)$, of free variables occurring in *positive* and *negative* positions, respectively, are defined in the natural way. The set of free variables in $\rho$ is denoted by $\mathsf{FVar}(\rho)$. We consider types equal up to $\alpha$-equivalence. Index substitutions are extended to sized types in the obvious way, using $\alpha$-conversion to avoid variable capture.

We denote by $\rho \geqslant \tau$ that the monotype $\tau$ is obtained by *instantiating* the variables quantified in $\rho$ with arbitrary index terms, i.e. if $\rho = \forall\vec{i}.\zeta$ then $\tau = \zeta\{\vec{a}/\vec{i}\}$ for some index terms $\vec{a}$. Notice that by our convention $\tau = \forall\epsilon.\ \tau$, we have $\tau \geqslant \tau$ for every monotype $\tau$.

The subtyping relation $\sqsubseteq_{\mathcal{J}}$ is given in Figure 3a. It depends on the interpretation of size indices, but otherwise is defined in the expected way. Subtyping inherits the following properties from the relation $\leq_{\mathcal{J}}$, see Lemma 5.1.

LEMMA 5.2.
1. *The subtyping relation is reflexive, transitive and closed under index substitutions.*
2. *If $a \leq_{\mathcal{J}} b$ then $\rho\{a/i\} \sqsubseteq_{\mathcal{J}} \rho\{b/i\}$ for all index variables $i \notin \mathsf{FVar}^{-}(\rho)$.*

We are interested in certain linear types, namely those in which any index term occurring in negative position is in fact a fresh index variable.

$$\frac{a \leq_{\mathcal{J}} b}{\mathsf{B}_a \sqsubseteq_{\mathcal{J}} \mathsf{B}_b}\ (\sqsubseteq_{\mathsf{B}}) \qquad \frac{\tau_1 \sqsubseteq_{\mathcal{J}} \tau_3 \quad \tau_2 \sqsubseteq_{\mathcal{J}} \tau_4}{\tau_1 \times \tau_2 \sqsubseteq_{\mathcal{J}} \tau_3 \times \tau_4}\ (\sqsubseteq_{\times})$$

$$\frac{\rho_2 \sqsubseteq_{\mathcal{J}} \rho_1 \quad \tau_1 \sqsubseteq_{\mathcal{J}} \tau_2}{\rho_1 \to \tau_1 \sqsubseteq_{\mathcal{J}} \rho_2 \to \tau_2}\ (\sqsubseteq_{\to}) \qquad \frac{\rho_2 \geqslant \tau_2 \quad \tau_1 \sqsubseteq_{\mathcal{J}} \tau_2 \quad \vec{i} \notin \mathsf{FVar}(\rho_2)}{\forall \vec{i}.\tau_1 \sqsubseteq_{\mathcal{J}} \rho_2}\ (\sqsubseteq_{\forall})$$

(a) Subtyping rules.

$$\frac{\rho \geqslant \tau}{\Gamma, x : \rho \vdash^{\mathcal{J}} x : \tau}\ (\textsc{Var}) \qquad \frac{s \in \mathcal{F} \cup C \quad s :: \rho \quad \rho \geqslant \tau}{\Gamma \vdash^{\mathcal{J}} s : \tau}\ (\textsc{Fun})$$

$$\frac{\Gamma \vdash^{\mathcal{J}} s : \tau_1 \times \tau_2 \quad \Gamma, x_1 : \tau_1, x_2 : \tau_2 \vdash^{\mathcal{J}} t : \tau}{\Gamma \vdash^{\mathcal{J}} \mathsf{let}\ (x_1, x_2) = s\ \mathsf{in}\ t : \tau}\ (\textsc{Let}) \qquad \frac{\Gamma \vdash^{\mathcal{J}} s_1 : \tau_1 \quad \Gamma \vdash^{\mathcal{J}} s_2 : \tau_2}{\Gamma \vdash^{\mathcal{J}} (s_1, s_2) : \tau_1 \times \tau_2}\ (\textsc{Pair})$$

$$\frac{\Gamma \vdash^{\mathcal{J}} s : (\forall \vec{i}.\zeta_1) \to \tau \quad \Gamma \vdash^{\mathcal{J}} t : \zeta_2 \quad \zeta_2 \sqsubseteq_{\mathcal{J}} \zeta_1 \quad \vec{i} \notin \mathsf{FVar}(\Gamma\restriction_{\mathsf{FVar}(t)})}{\Gamma \vdash^{\mathcal{J}} s\ t : \tau}\ (\textsc{App})$$

(b) Typing rules

Fig. 3. Typing and subtyping rules, depending on the semantic interpretation $\mathcal{J}$.

*Definition 5.3 (Canonical Sized Type, Sized Type Declaration).*

1. A monotype $\tau$ is *canonical* if one of the following alternatives hold:
   - $\tau = \mathsf{B}_a$ is an indexed base type;
   - $\tau = \zeta_1 \times \zeta_2$ for two canonical monotypes $\zeta_1, \zeta_2$;
   - $\tau = \mathsf{B}_i \to \zeta$ with $i \notin \mathsf{FVar}^-(\zeta)$;
   - $\tau = \sigma \to \zeta$ for a canonical polytype $\sigma$ and canonical type $\zeta$ with $\mathsf{FVar}(\sigma) \cap \mathsf{FVar}^-(\zeta) = \varnothing$.
2. A polytype $\sigma = \forall \vec{i}.\tau$ is *canonical* if $\tau$ is canonical and $\mathsf{FVar}^-(\tau) \subseteq \{\vec{i}\}$.
3. To each function symbol $s \in \mathcal{F} \cup C$, we associate a *closed* and *canonical* type $\rho$ whose skeleton coincides with the simple type of $s$. We write $s :: \rho$ and call $s :: \rho$ the *sized type declaration* of $s$.

Canonicity ensures that pattern matching can be resolved with a simple substitution mechanism, rather than a sophisticated unification based mechanism that takes the semantic interpretation $\mathcal{J}$ into account. Canonical types enjoy the following substitution property.

Lemma 5.4. *Let $\rho$ be a canonical type and suppose that $i \notin \mathsf{FVar}^-(\rho)$. Then $\rho\{a/i\}$ is again canonical.*

In Figure 3b we depict the typing rules of our sized type system. A *(typing) context* $\Gamma$ is a mapping from variables $x$ to types $\rho$ so that the skeleton of $\rho$ coincides with the simple type of $x$. We denote the context $\Gamma$ that maps variables $x_i$ to $\rho_i$ ($1 \leq i \leq n$) by $x_1 : \rho_1, \ldots, x_n : \rho_n$. The empty context is denoted by $\varnothing$. We lift set operations as well as the notion of (positive, negative) free variables and application of index substitutions to contexts in the obvious way. We denote by $\Gamma\restriction_X$ the *restriction* of context $\Gamma$ to a set of variables $X \subseteq \mathcal{X}$. The typing statement $\Gamma \vdash^{\mathcal{J}} s : \tau$ states that under the typing contexts $\Gamma$, the term $s$ has the *monotype* $\tau$, when indices are interpreted with respect to $\mathcal{J}$. The typing rules from Figure 3b are fairly standard. Symbols $s \in \mathcal{F} \cup C \cup \mathcal{X}$ are given instance types of their associated types. This way we achieve the desired degree of polymorphism outlined in Section 2. Subtyping and generalisation are confined to function application, see rule (App). Here, the monotype $\zeta_2$ of the argument term $t$ is weakened to $\zeta_1$, the side-conditions put on index variables $\vec{i}$ allow then a generalisation of $\zeta_1$ to $\forall \vec{i}.\zeta_1$, the type expected by the function $s$. This way,

$$\frac{\mathsf{f} :: \forall \vec{i}.\tau}{\varnothing \vdash_{\mathsf{FP}} \mathsf{f} : \tau}\ (\text{FpFun}) \qquad \frac{\Gamma \vdash_{\mathsf{FP}} t : \rho \to \tau}{\Gamma \uplus \{x : \rho\} \vdash_{\mathsf{FP}} t\ x : \tau}\ (\text{FpAppVar})$$

$$\frac{\begin{array}{c}(\mathsf{FVar}(\Gamma_1) \cup \mathsf{FVar}(\tau)) \cap (\mathsf{FVar}(\Gamma_2) \cup \mathsf{FVar}(\mathsf{B}_a)) = \varnothing \\ \Gamma_1 \vdash_{\mathsf{FP}} s : \mathsf{B}_i \to \tau \qquad \Gamma_2 \vdash_{\mathsf{FP}} t : \mathsf{B}_a \qquad s \notin \mathcal{X}\end{array}}{\Gamma_1 \uplus \Gamma_2 \vdash_{\mathsf{FP}} s\ t : \tau\{a/i\}}\ (\text{FpAppNVar})$$

Fig. 4. Rules for computing the footprint of a term.

the complete system becomes syntax directed. We remark that subtyping is prohibited in the typing of the left spine of applicative terms.

Since our programs are equationally-defined, we need to define when equations are well-typed. In essence, we will say that a program P is *well-typed*, if, for all equations $l = r$, the right-hand side $r$ can be given a subtype of $l$. Due to polymorphic typing of recursion, and since our typing relation integrates subtyping, we have to be careful. Instead of giving $l$ an arbitrary derivable type, we will have to give it a *most general type* that has not been weakened through subtyping. Put otherwise, the type for the equation, which is determined by $l$, should precisely relate to the declared type of the considered function.

To this end, we introduce the restricted typing relation, the *footprint relation*, depicted in Figure 4. The footprint relation makes essential use of canonicity of sized type declaration and the shape of patterns. In particular, $x_1 : \rho_1, \dots, x_n : \rho_n \vdash_{\mathsf{FP}} s : \tau$ implies that all $\rho_i$ and $\tau$ are canonical. The footprint relation can be understood as a function that, given a left-hand side $\mathsf{f}\ p_1 \cdots p_k$, results in a typing context $\Gamma$ and monotype $\tau$. This function is total, for two reasons. First of all, the above lemma confirms that the term $s$ in rule (FpAppNVar) is given indeed a canonical type of the stated form. Secondly, the disjointness condition required by this rule can always be satisfied via $\alpha$-conversion. It is thus justified to define $\mathsf{footprint}(\mathsf{f}\ p_1 \cdots p_k) := (\Gamma, \tau)$ for some (particular) context $\Gamma$ and type $\tau$ that satisfies $\Gamma \vdash_{\mathsf{FP}} \mathsf{f}\ p_1 \cdots p_k : \tau$.

*Definition 5.5.* Let P be a program, such that every function and constructor has a declared sized type. We call a rule $l = r$ from P *well-typed under the interpretation* $\mathcal{J}$ if

$$\Gamma \vdash_{\mathsf{FP}} l : \tau \implies \Gamma \vdash^{\mathcal{J}} r : \zeta \text{ for some monotype } \zeta \text{ with } \zeta \sqsubseteq_{\mathcal{J}} \tau,$$

holds for all contexts $\Gamma$ and types $\tau$. The program P is *well-typed under the interpretation* $\mathcal{J}$ if all its equations are.

## 5.3 Subject Reduction

It is more convenient to deal with subject reduction when subtyping is *not* confined to function application. We thus define the typability relation $\Gamma \vdash^{\mathcal{J}}_{\mathsf{e}} s : \tau$. It is defined in terms of all the rules depicted in Figure 3b, together with the following subtyping rule.

$$\frac{\Gamma \vdash^{\mathcal{J}}_{\mathsf{e}} s : \zeta \qquad \zeta \sqsubseteq_{\mathcal{J}} \tau}{\Gamma \vdash^{\mathcal{J}}_{\mathsf{e}} s : \tau}\ (\text{SubType})$$

As a first step towards subject reduction, we clarify that the footprint correctly accounts for pattern matching. Consider an equation $l = r \in \mathsf{P}$ from a well-typed program P, where $\Gamma \vdash_{\mathsf{FP}} l : \zeta$. If the left-hand side matches a term $s$ of type $\tau$, i.e. $s = l\theta$, then the type $\tau$ is an instance of $\zeta$, or a supertype thereof. Moreover, the images of $\theta$ can all be typed as instances of the corresponding types in the typing context $\Gamma$. More precise:

Lemma 5.6 (Footprint Lemma). *Let $s = \mathsf{f}\ p_1 \cdots p_n$ be a linear term over variables $x_1, \dots, x_m$, and let $\theta = \{t_1, \dots, t_m / x_1, \dots, x_m\}$ be a substitution. If $\vdash_{\mathsf{e}}^{\mathcal{J}} s\theta : \tau$ then there exist a context $\Gamma = x_1 : \rho_1, \dots, x_m : \rho_m$ and a type $\zeta$ such that $\Gamma \vdash_{\mathsf{FP}} s : \zeta$ holds. Moreover, for some index substitution $\vartheta$ we have $\zeta\vartheta \sqsubseteq_{\mathcal{J}} \tau$ and $\vdash_{\mathsf{e}}^{\mathcal{J}} t_n : \tau_n\vartheta$, where $\rho_n = \forall\vec{i}.\tau_n$ $(1 \leqslant n \leqslant m)$.*

The following constitutes the main lemma of this section, the *substitution lemma*:

$$\vdash_{\mathsf{e}}^{\mathcal{J}} s_n : \tau_n\ (1 \leq n \leq m) \text{ and } x_1 : \tau_1, \dots, x_m : \tau_m \vdash_{\mathsf{e}}^{\mathcal{J}} s : \tau \quad \Rightarrow \quad \vdash_{\mathsf{e}}^{\mathcal{J}} s\{s_1, \dots, s_m / x_1, \dots, x_m\} : \tau\ .$$

Indeed, we prove a generalisation.

Lemma 5.7 (Generalised Substitution Lemma). *Let $s$ be a term with free variables $x_1, \dots, x_m$, let $\Gamma$ be a context over $x_1, \dots, x_m$, and let $\vartheta$ be an index substitution. If $\Gamma \vdash_{\mathsf{e}}^{\mathcal{J}} s : \tau$ for some type $\tau$ and $\vdash_{\mathsf{e}}^{\mathcal{J}} x_n\theta : \tau_n\vartheta$ holds for the type $\tau_n$ with $\Gamma(x_n) = \forall\vec{i}.\tau_n$ $(1 \leqslant n \leqslant m)$, then $\vdash_{\mathsf{e}}^{\mathcal{J}} s\theta : \tau\vartheta$.*

The combination of these two lemmas is almost all we need to reach our goal.

Theorem 5.8 (Subject Reduction). *Suppose $\mathsf{P}$ is well-typed under $\mathcal{J}$. If $\vdash_{\mathsf{e}}^{\mathcal{J}} s : \tau$ and $s \to_{\mathsf{P}} t$ then $\vdash_{\mathsf{e}}^{\mathcal{J}} t : \tau$.*

But what does Subject Reduction tells us, besides guaranteeing that types are preserved along reduction? Actually, a lot: If $\vdash_{\mathsf{e}}^{\mathcal{J}} s : \mathsf{B}_a$, we are now sure that the evaluation of $s$, if it terminates, would lead to a value of size at most $[\![a]\!]_{\mathcal{J}}$. Of course, this requires that we give (first-order) *data-constructors* a suitable sized type. To this end, let us call a sized type *additive* if it is of the form $\forall\vec{i}.\ \mathsf{B}_{i_1} \to \cdots \to \mathsf{B}_{i_k} \to \mathsf{B}_{\mathsf{s}(i_1+\cdots+i_k)}$.

Corollary 5.9. *Suppose $\mathsf{P}$ is well-typed under the interpretation $\mathcal{J}$, where data-constructors are given an additive type. Suppose the first-order function `main` has type $\forall\vec{i}.\mathsf{B}_{i_1} \to \cdots \to \mathsf{B}_{i_k} \to \mathsf{B}_a$. Then for all inputs $d_1, \dots, d_n$, if `main` $d_1 \cdots d_k$ reduces to a data value $d$, then the size of $d$ is bounded by $s(|d_1|, \dots, |d_k|)$, where $s$ is the function $s(i_1, \dots, i_k) = [\![a]\!]_{\mathcal{J}}^{\alpha}$.*

As we have done in the preceding examples, the notion of additive sized type could be suited so that constants like the list constructor [ ] are attributed with a size of zero. Thereby, the sized type for lists would reflect the length of lists. Note that the corollary by itself, does not mean much about the *complexity* of evaluating $s$. We will return to this in Section 7.

## 6 SIZED TYPES INFERENCE

The kind of rich type discipline we have just introduced cannot be enforced by requiring the programmer to annotate programs with size types, since this would simply be too burdensome. Studying to which extent types can be inferred, then, is of paramount importance.

We will now describe a type inference procedure that, given a program, produces a set of first-order constraints that are satisfiable *iff* the term is size-typable. At the heart of this procedure lies the idea that we turn the typing system from Figure 3 into a system that, instead of checking, collects all constraints $a \leq b$ put on indices. These constraints are then resolved in a second stage. The so obtained solution can then be used to reconstruct a typing derivation with respect to the system from Figure 3. As with any higher-ranked polymorphic type system, the main challenge here lies in picking suitable types instances from polymorphic types. In our system, this concerns rules (Var) and (Fun). Systems used in practice, such as the one of Peyton Jones et al. [2007], use a combination of forward and backward inference to determine suitable instantiated types. Still, the resulting inference system is incomplete. In our sized type system, higher-ranked polymorphism is confined to size indices. This, in turn, allows us to divert the choice to the solving stage, thereby retaining relative completeness. To this end, we introduce meta variables $E$ in our index language. Whereas in the typing system from Figure 3 index variables $i$ are instantiated by concrete index terms $a$, our inference system uses a fresh meta variable $E$ as placeholder for $a$. A suitable assignments to $E$ will be determined in the constraint solving stage. A minor complication arises as we will have to introduce additional constraints $i \notin_{sol} E$ on meta variables $E$ that condition the set of terms $E$ may represent. This is necessary to deal with the side conditions on free variables, exhibited by the subtyping relation as well as in typing the rule for application. All of this is made precise in the following.

## 6.1 First- and Second-order Constraint Problems

As a first step towards inference, we introduce metavariables to our index language. Let $\mathcal{Y}$ be a countably infinite set of *second-order index variables*, which stand for arbitrary index terms. Second-order index variables are denoted by $E, F, \ldots$ . The set of *second-order index terms* is then generated over the set of index variables $i \in \mathcal{V}$, the set of second-order index variables $E \in \mathcal{Y}$ and index symbols $\mathsf{f} \in \mathcal{G}$ as follows.

$$\textbf{(second-order index terms)} \qquad e, f ::= i \mid E \mid \mathsf{f}(e_1, \ldots, e_{\mathsf{ar}(\mathsf{f})}) \ .$$

We denote by $\mathsf{Var}(e) \subset \mathcal{V}$ the set of (usual) index variables, and by $\mathsf{SoVar}(e) \subset \mathcal{Y}$ the set of second-order index variables occurring in $e$.

*Definition 6.1 (Second-order Constraint Problem, Model).* A *second-order constraint problem* $\Phi$ (*SOCP* for short) is a set of (i) *inequality constraints* of the form $e \leq f$ and (ii) *occurrence constraints* of the form $i \notin_{sol} E$. Let $v$ be a substitution from second-order index variables to first-order index terms $a$, i.e. $\mathsf{SoVar}(a) = \varnothing$. Furthermore, let $\mathcal{J}$ be an interpretation of $\mathcal{G}$. Then $(\mathcal{J}, v)$ *is a model of* $\Phi$, in notation $(\mathcal{J}, v) \models \Phi$, if (i) $ev \leq_{\mathcal{J}} fv$ holds for all inequalities $e \leq f \in \Phi$; and (ii) $i \notin \mathsf{Var}(v(E))$ for each occurrence constraint $i \notin_{sol} E$.

We say that $\Phi$ is *satisfiable* if it has a model $(\mathcal{J}, v)$. The term $v(E)$ is called the *solution* of $E$. We call $\Phi$ a *first-order constraint problem* (*FOCP* for short) if none of the inequalities $e \leq f$ contain a second-order variable. Note that satisfiability of a FOCP $\Phi$ depends only on the semantic interpretation $\mathcal{J}$ of index functions. It is thus justified that FOCPs $\Phi$ contain no occurrence constraints. We then write $\mathcal{J} \models \Phi$ if $\mathcal{J}$ models $\Phi$.

SOCPs are very much suited to our inference machinery. In contrast, satisfiability of FOCPs is a re-occurring problem in various fields. To generate models for SOCPs, we will reduce satisfiability of SOCPs to the one of FOCPs. This reduction is in essence a form of *skolemization.*

*Skolemization.* Skolemization is a technique for eliminating existentially quantified variables from a formula. A witness for an existentially quantified variable can be given as a function in the universally quantified variables, the *skolem function.* We employ a similar idea in our reduction of satisfiability from SOCPs to FOCPs, which substitutes second-order variables $E$ by *skolem term* $\mathsf{f}_E(\vec{i})$, for a unique *skolem function* $\mathsf{f}_E$, and where the sequence of variables $\vec{i}$ over-approximates the index variables of possible solutions to $E$. The over-approximation of index variables is computed by a simple fixed-point construction, guided by the observation that a solution of $E$ contains wlog. an index variable $i$ only when (i) $i$ is related to $E$ in an inequality of the SOCP $\Phi$ and (ii) the SOCP does not require $i \notin_{sol} E$. Based on these observations, skolemization is formally defined as follows.

*Definition 6.2.* Let $\Phi$ be a SOCP.

1. For each second-order variable $F$ of $\Phi$, we define the sets $\mathcal{SV}_F^{\Phi,\leq} \subset \mathcal{V}$ of *index variables related to $F$ by inequalities* as the least set satisfying, for each $(e \leq f) \in \Phi$ with $F \in \mathsf{SoVar}(f)$, (i) $\mathsf{Var}(e) \subseteq \mathcal{SV}_F^{\Phi,\leq}$; and (ii) $\mathcal{SV}_E^{\Phi,\leq} \subseteq \mathcal{SV}_F^{\Phi,\leq}$ whenever $E$ occurs in $e$. The set of *skolem variables* for $F$ is then given by $\mathcal{SV}_F^{\Phi} := \mathcal{SV}_F^{\Phi,\leq} \setminus \{i \mid (i \notin_{sol} F) \in \Phi\}$.

2. For each second-order variable $E$ of $\Phi$, let $\mathsf{f}_E$ be a fresh index symbol, the *skolem function* for $E$. The arity of $\mathsf{f}_E$ is the cardinality of $\mathcal{SV}_E^{\Phi}$. The *skolem substitution* $\upsilon_\Phi$ is given by $\upsilon_\Phi(E) := \mathsf{f}_E(i_1, \dots, i_k)$ where $\mathcal{SV}_E^{\Phi} = \{i_1, \dots, i_k\}$.
3. We define the *skolemization* of $\Phi$ by $\mathsf{skolemize}(\Phi) := \{e\upsilon_\Phi \leq f\upsilon_\Phi \mid e \leq f \in \Phi\}$.

Note that the skolem substitution $\upsilon_\Phi$ satisfies by definition all occurrence constraints of $\Phi$. Thus skolemization is trivially sound: $\mathcal{J} \vDash \mathsf{skolemize}(\Phi)$ implies $(\mathcal{J}, \upsilon_\Phi) \vDash \Phi$. Concerning completeness, the following lemma provides the central observation. Wlog. a solution to $E$ contains only variables of $\mathcal{SV}_E^{\Phi}$:

Lemma 6.3. *Let $\Phi$ be a SOCP with model $(\mathcal{J}, \upsilon)$. Then there exists a restricted second-order substitution $\upsilon_r$ such that $(\mathcal{J}, \upsilon_r)$ is a model of $\Phi$ and $\upsilon_r$ satisfies* $\mathsf{Var}(\upsilon_r(E)) \subseteq \mathcal{SV}_E^{\Phi}$ *for each second-order variable $E$ of $\Phi$.*

Proof. The restricted substitution $\upsilon_r$ is obtained from $\upsilon$ by substituting in $\upsilon(E)$ zero for all non-skolem variables $i \notin \mathcal{SV}_E^{\Phi}$. From the assumption that $(\mathcal{J}, \upsilon)$ is a model of $\Phi$, it can then be shown that $e\upsilon_r \leq_{\mathcal{J}} f\upsilon_r$ holds for each inequality $(e \leq f) \in \Phi$, essentially using the inequalities depicted in Lemma 5.1. As the occurrence constraints are also satisfied under the new model by definition, the lemma follows. □

Theorem 6.4 (Skolemisation — Soundness and Completeness).
1. ***Soundness**: If $\mathcal{J} \vDash$* $\mathsf{skolemize}(\Phi)$ *then $(\mathcal{J}, \upsilon_\Phi) \vDash \Phi$ holds.*
2. ***Completeness**: If $(\mathcal{J}, \upsilon) \vDash \Phi$ then $\hat{\mathcal{J}} \vDash$* $\mathsf{skolemize}(\Phi)$ *holds for an extension $\hat{\mathcal{J}}$ of $\mathcal{J}$ to skolem functions.*

Proof. It suffices to consider completeness. Suppose $(\mathcal{J}, \upsilon) \vDash \Phi$ holds, where wlog. $\upsilon$ satisfies $\mathsf{Var}(\upsilon(E)) \subseteq \mathcal{SV}_E^{\Phi}$ for each second-order variable $E \in \mathsf{SoVar}(\Phi)$ by Lemma 6.3. Let us extend the interpretation $\mathcal{J}$ to an interpretation $\hat{\mathcal{J}}$ by defining $[\![\mathsf{f}_E]\!]_{\hat{\mathcal{J}}}(i_1, \dots, i_k) := [\![\upsilon(E)]\!]_{\mathcal{J}}$, where $\mathcal{SV}_E^{\Phi} = \{i_1, \dots, i_k\}$, for all $E \in \mathsf{SoVar}(\Phi)$. By the assumption on $\upsilon$, $\hat{\mathcal{J}}$ is well-defined. From the definition of $\hat{\mathcal{J}}$, it is then not difficult to conclude that also $(\hat{\mathcal{J}}, \upsilon_\Phi)$ is a model of $\Phi$, and consequently, $\hat{\mathcal{J}}$ is a model of $\mathsf{skolemize}(\Phi)$. □

## 6.2 Constraint Generation

We now define a function obligations that maps a program P to a SOCP $\Phi$. If $(\mathcal{J}, \upsilon)$ is a model of $\Phi$, then P will be well-typed under the interpretation $\mathcal{J}$. Throughout the following, we allow second-order index terms to occur in sized types. If a second-order variable occurs in a type $\rho$, we call $\rho$ a *template type*. The function obligations is itself defined on the two statements $\Phi \vdash_{\mathsf{ST}} \tau \sqsubseteq \zeta$ and $\Phi; \Gamma \vdash_{\mathsf{I}} s : \tau$ that are used in the generation of constraints resulting from the subtyping and the typing relation, respectively. The inference rules are depicted in Figure 5. These are in one-to-one correspondence with those of Figure 3. The crucial difference is that rule ($\sqsubseteq_{\mathsf{B}}$-I) simply records a constraint $a \leq b$, whereas the corresponding rule ($\sqsubseteq_{\mathsf{B}}$) in Figure 3a relies on the semantic comparison $a \leq_{\mathcal{J}} b$. Instantiation of polytypes is resolved by substituting second-order variables, in rule (Var-I) and (Fun-I). For a sequence of index variables $\vec{i} = i_1, \dots, i_m$ and sequence of monotype $\vec{\tau} = \tau_1, \dots, \tau_n$, we use the notation $\vec{i} \notin_{sol} \vec{\tau}$ to denote the collection of occurrence constraints $i_k \notin_{sol} E$ for all $1 \leq k \leq m$ and $E \in \mathsf{SoVar}(\tau_l)$, $1 \leq l \leq n$. Occurrence constraints are employed in rules ($\sqsubseteq_\forall$-I) and (App-I) to guarantee freshness of the quantified index variables also with respect to solutions to second-order index variables.

Notice that the involved rules are again syntax directed. Consequently, a derivation of $\Phi; \Gamma \vdash_{\mathsf{I}} s : \tau$ naturally gives rise to a procedure that, given a context $\Gamma$ and term $s$, yields the SOCP $\Phi$ and template

$$\frac{}{\{a \leq b\} \vdash_{\mathsf{ST}} \mathsf{B}_a \sqsubseteq \mathsf{B}_b}\ (\sqsubseteq_{\mathsf{B}}\text{-I}) \qquad \frac{\Phi_1 \vdash_{\mathsf{ST}} \tau_1 \sqsubseteq \tau_3 \quad \Phi_2 \vdash_{\mathsf{ST}} \tau_2 \sqsubseteq \tau_4}{\Phi_1, \Phi_2 \vdash_{\mathsf{ST}} \tau_1 \times \tau_2 \sqsubseteq \tau_3 \times \tau_4}\ (\sqsubseteq_{\times}\text{-I})$$

$$\frac{\Phi_1 \vdash_{\mathsf{ST}} \rho_2 \sqsubseteq \rho_1 \quad \Phi_2 \vdash_{\mathsf{ST}} \tau_1 \sqsubseteq \tau_2}{\Phi_1, \Phi_2 \vdash_{\mathsf{ST}} \rho_1 \to \tau_1 \sqsubseteq \rho_2 \to \tau_2}\ (\sqsubseteq_{\to}\text{-I}) \qquad \frac{\vec{E}\ \mathit{fresh} \quad \Phi \vdash_{\mathsf{ST}} \tau_1 \sqsubseteq \tau_2\{\vec{E}/\vec{j}\} \quad \vec{i} \notin \mathsf{FVar}(\forall\vec{j}.\tau_2)}{\Phi, \vec{i} \notin_{sol} \tau_1, \vec{i} \notin_{sol} \tau_2 \vdash_{\mathsf{ST}} \forall\vec{i}.\tau_1 \sqsubseteq \forall\vec{j}.\tau_2}\ (\sqsubseteq_{\forall}\text{-I})$$

(a) Subtyping rules.

$$\frac{\vec{E}\ \mathit{fresh}}{\varnothing; \Gamma, x : \forall\vec{i}.\tau \vdash_{\mathsf{I}} x : \tau\{\vec{E}/\vec{i}\}}\ (\textsc{Var-I}) \qquad \frac{x \in \mathcal{F} \cup \mathcal{C} \quad s :: \forall\vec{i}.\tau \quad \vec{E}\ \mathit{fresh}}{\varnothing; \Gamma \vdash_{\mathsf{I}} s : \tau\{\vec{E}/\vec{i}\}}\ (\textsc{Fun-I})$$

$$\frac{\Phi_1; \Gamma \vdash_{\mathsf{I}} s : \tau_1 \times \tau_2 \quad \Phi_2; \Gamma, x_1 : \tau_1, x_2 : \tau_2 \vdash_{\mathsf{I}} t : \tau}{\Phi_1, \Phi_2; \Gamma \vdash_{\mathsf{I}} \mathsf{let}\ (x_1, x_2) = s\ \mathsf{in}\ t : \tau}\ (\textsc{Let-I}) \qquad \frac{\Phi_1; \Gamma \vdash_{\mathsf{I}} s_1 : \tau_1 \quad \Phi_2; \Gamma \vdash_{\mathsf{I}} s_2 : \tau_2}{\Phi_1, \Phi_2; \Gamma \vdash_{\mathsf{I}} (s_1, s_2) : \tau_1 \times \tau_2}\ (\textsc{Pair-I})$$

$$\frac{\Phi_1; \Gamma \vdash_{\mathsf{I}} s : (\forall\vec{i}.\zeta_1) \to \tau \quad \Phi_2; \Gamma \vdash_{\mathsf{I}} t : \zeta_2 \quad \Phi_3 \vdash_{\mathsf{ST}} \zeta_2 \sqsubseteq \zeta_1 \quad \vec{i} \notin \mathsf{FVar}(\Gamma\restriction_{\mathsf{FVar}(t)})}{\Phi_1, \Phi_2, \Phi_3, \vec{i} \notin_{sol} \zeta_1, \vec{i} \notin_{sol} \Gamma\restriction_{\mathsf{FVar}(t)}; \Gamma \vdash_{\mathsf{I}} s\ t : \tau}\ (\textsc{App-I})$$

(b) Typing rules

Fig. 5. Type inference rules, generating a second-order constraint solving problem.

monotype $\tau$, modulo renaming of second-order variables. By imposing an order on how second-order variables are picked in the inference of $\Phi; \Gamma \vdash_{\mathsf{I}} s : \tau$, the resulting SOCP and template type become unique. The function $\mathsf{infer}(\Gamma, s) := (\Phi, \tau)$ defined this way is thus well-defined. In a similar way, we define the function $\mathsf{subtypeOf}(\tau, \zeta) := \Phi$, where $\Phi$ is the SOCP with $\Phi \vdash_{\mathsf{ST}} \tau \sqsubseteq \zeta$.

*Definition 6.5 (Constraint Generation).* For a program P we define

$$\mathsf{obligations}(\mathsf{P}) = \{\mathsf{check}(\Gamma, r, \tau) \mid l = r \in \mathsf{P} \text{ and } \mathsf{footprint}(l) = (\Gamma, \tau)\},$$

where $\mathsf{check}(\Gamma, s, \tau) = \Phi_1 \cup \Phi_2$ for $(\Phi_1, \zeta) = \mathsf{infer}(\Gamma, s)$ and $\Phi_2 = \mathsf{subtypeOf}(\zeta, \tau)$.

## 6.3 Soundness and Relative Completeness

We will now give a series of soundness and completeness results that will lead us to the main result about type inference, namely Corollary 6.9 below. In essence, we show that a derivation of $\Phi \vdash_{\mathsf{ST}} \tau \sqsubseteq \zeta$ (and $\Phi; \Gamma \vdash_{\mathsf{I}} s : \tau$) together with a model $(\mathcal{J}, \upsilon) \vDash \Phi$ can be turned into a derivation of $\tau\upsilon \sqsubseteq_{\mathcal{J}} \zeta\upsilon$ (and $\Gamma\upsilon \vdash^{\mathcal{J}} s : \tau\upsilon$), and *vice versa.*

Lemma 6.6. *Subtyping inference is sound and complete, more precise:*

1. ***Soundness**: If $\Phi \vdash_{\mathsf{ST}} \tau \sqsubseteq \zeta$ holds for two template types $\tau, \zeta$ then $\tau\upsilon \sqsubseteq_{\mathcal{J}} \zeta\upsilon$ holds for every model $(\mathcal{J}, \upsilon)$ of $\Phi$.*
2. ***Completeness**: If $\tau\upsilon \sqsubseteq_{\mathcal{J}} \zeta\upsilon$ holds for two template types $\tau$ and $\zeta$ and second-order index substitution $\upsilon$ then $\Phi \vdash_{\mathsf{ST}} \tau \sqsubseteq \zeta$ is derivable for some SOCP $\Phi$. Moreover, there exists an extension $\nu$ of $\upsilon$, whose domain coincides with the second-order variables occurring in $\Phi \vdash_{\mathsf{ST}} \tau \sqsubseteq \zeta$, such that $(\mathcal{J}, \nu)$ is a model of $\Phi$.*

Proof. Concerning soundness, we consider a derivation of $\Phi \vdash_{\mathsf{ST}} \tau_1 \sqsubseteq \tau_2$, and fix a second-order substitution $\upsilon$ and interpretation $\mathcal{J}$ such that $(\mathcal{J}, \upsilon) \vDash \Phi$ holds. Then $\tau_1\upsilon \sqsubseteq_{\mathcal{J}} \tau_2\upsilon$ can be proven by induction on the derivation of $\Phi \vdash_{\mathsf{ST}} \tau_1 \sqsubseteq \tau_2$. Concerning completeness we fix a second-order substitution $\upsilon$ and construct for any two types $\tau_1, \tau_2$ with $\tau_1\upsilon \sqsubseteq_{\mathcal{J}} \tau_2\upsilon$ an inference of $\Phi \vdash_{\mathsf{ST}} \tau_1 \sqsubseteq \tau_2$ for some SOCP $\Phi$ together with an extension $\nu$ of $\upsilon$ that satisfies $(\mathcal{J}, \nu) \vDash \Phi$. The construction is done by induction on the proof of $\tau_1 \upsilon \sqsubseteq_{\mathcal{J}} \tau_2 \upsilon$. The substitution $\nu$ extends $\upsilon$ precisely on those fresh variables introduced by rule ($\sqsubseteq_{\forall}$-I) in the constructed proof of $\Phi \vdash_{\mathsf{ST}} \tau_1 \sqsubseteq \tau_2$. □

Lemma 6.7. *Type inference is sound and complete in the following sense:*

1. ***Soundness:*** *If $\Phi; \Gamma \vdash_{\mathsf{I}} s : \tau$ holds for a template type $\tau$ then $\Gamma\upsilon \vdash^{\mathcal{J}} s : \tau\upsilon$ holds for every model $(\mathcal{J}, \upsilon)$ of $\Phi$.*
2. ***Completeness:*** *If $\Gamma \vdash^{\mathcal{J}} s : \tau$ holds for a context $\Gamma$ and type $\tau$ then there exists a template type $\zeta$ and a second-order index substitution $\upsilon$, with $\zeta\upsilon = \tau$, such that $\Phi; \Gamma \vdash_{\mathsf{I}} s : \zeta$ is derivable for some SOCP $\Phi$. Moreover, $(\mathcal{J}, \upsilon)$ is a model of $\Phi$.*

Proof. Concerning soundness, we fix a model $(\mathcal{J}, \upsilon)$ of $\Phi$ and prove then $\Gamma\upsilon \vdash^{\mathcal{J}} s : \tau\upsilon$ by induction on $\Phi; \Gamma \vdash_{\mathsf{I}} s : \tau$. Concerning completeness, we prove the following stronger statement. Let $\upsilon$ be a second-order index substitution, let $\Gamma$ be a context over template schemas and let $\tau$ be a type. If $\Gamma\upsilon \vdash^{\mathcal{J}} s : \tau$ is derivable then there exists an extension $\nu$ of $\upsilon$ together with a template type $\zeta$, where $\zeta\nu = \tau$, such that $\Phi; \Gamma \vdash_{\mathsf{I}} s : \zeta$ holds for some SOCP $\Phi$. Moreover, $(\mathcal{J}, \nu)$ is a model of $\Phi$. The proof of this statement is then carried out by induction on the derivation of $\Gamma\upsilon \vdash^{\mathcal{J}} s : \tau$. Strengthening of the hypothesis is necessary to deal with let-expressions. □

Theorem 6.8 (Inference — Soundness and Relative Completeness). *Let* P *be a program and let* $\Phi = \mathsf{obligations}(\mathsf{P})$.

1. ***Soundness:*** *If $(\mathcal{J}, \upsilon)$ is a model of $\Phi$, then* P *is well-typed under the interpretation $\mathcal{J}$.*
2. ***Completeness:*** *If* P *is well-typed under the interpretation $\mathcal{J}$, then there exists a second-order index substitution $\upsilon$ such that $(\mathcal{J}, \upsilon)$ is a model of $\Phi$.*

Proof. Concerning soundness, let $(\mathcal{J}, \upsilon)$ be a model of $\Phi$. Fix a rule $l = r$ of P, and let $(\Gamma, \tau) = \mathsf{footprint}(l)$. Notice that $(\mathcal{J}, \upsilon)$ is in particular a model of the constraint $\Phi_1 \cup \Phi_2 = \mathsf{check}(\Gamma, r, \tau) \subseteq \Phi$, where $\Phi_1; \Gamma \vdash_{\mathsf{I}} r : \zeta$ and $\Phi_2 \vdash_{\mathsf{ST}} \zeta \sqsubseteq \tau$ for some type $\zeta$. Using that the footprint of $l$ does not contain second-order index variables, Lemma 6.7(1) and Lemma 6.6(1) then prove $\Gamma \vdash^{\mathcal{J}} s : \zeta\upsilon$ and $\zeta\upsilon \sqsubseteq_{\mathcal{J}} \tau$, respectively. Conclusively, the rule $l = r$ is well-typed and the claim follows. Completeness is proven dually, using Lemma 6.7(2) and Lemma 6.6(2). □

This, in conjunction with Theorem 6.4, then yields:

Corollary 6.9. *Let* P *be a program and let* $\Phi = \mathsf{obligations}(\mathsf{P})$.

1. ***Soundness:*** *If $\mathcal{J}$ is a model of* $\mathsf{skolemize}(\Phi)$*, then* P *is well-typed under the interpretation $\mathcal{J}$.*
2. ***Completeness:*** *If* P *is well-typed under the interpretation $\mathcal{J}$, then $\hat{\mathcal{J}}$ is a model of* $\mathsf{skolemize}(\Phi)$*, for some extension $\hat{\mathcal{J}}$ of $\mathcal{J}$.*

## 7 TICKING TRANSFORMATION AND TIME COMPLEXITY ANALYSIS

Our size type system is a sound methodology for keeping track of the size of intermediate results a program needs when evaluated. Knowing all this, however, is not sufficient for complexity analysis. In a sense, we need to be able to reduce complexity analysis to size analysis.

We now introduce the *ticking transformation* mentioned in the Introduction. Conceptually, this transformation takes a program P and translates it into another program $\hat{\mathsf{P}}$ which behaves like P, but additionally computes also the runtime on the given input. The latter is achieved by threading through the computation a counter, the *clock*, which is advanced whenever an equation of P fires. Technically, we lift all the involved functions into a *state monad*,[1] that carries as state the clock. More precise, a $k$-ary function $\mathsf{f} :: \tau_1 \to \cdots \to \tau_k \to \tau$ of P will be modeled in $\hat{\mathsf{P}}$ by a function

[1]We could have achieved a similar effect via a writer monad. We prefer however the more general notion of a state monad, as this allows us to in principle also encode resources that can be reclaimed, e.g., heap space.

```
f x = let x1 = g in
      let x2 = h in
      let x3 = x2 x in
      let x4 = x1 x3 in x4
```

```
f̂1 x z = let (x1,z1) = ĝ0 z in
         let (x2,z2) = ĥ0 z1 in
         let (x3,z3) = x2 x z2 in
         let (x4,z4) = x1 x3 z3 in (x4,T z4)
f̂0 z = (f̂1, z)
```

Fig. 6. Equation f $x$ = g (h $x$) in let-normalform (left) and ticked let-normalform (right).

$\hat{\mathsf{f}}_k :: \langle\tau_1\rangle \to \cdots \to \langle\tau_k\rangle \to \mathsf{C} \to \langle\tau\rangle \times \mathsf{C}$, where $\mathsf{C}$ is the type of the *clock*. Here, $\langle\rho\rangle$ enriches functional types $\rho$ with clocks accordingly. The function $\hat{\mathsf{f}}_k$ behaves in essence like f, but advances the threaded clock suitably. The clock-type $\mathsf{C}$ encodes the running time in unary notation using two constructors $\mathsf{Z}^{\mathsf{C}}$ and $\mathsf{T}^{\mathsf{C}\to\mathsf{C}}$. The size of the clock thus corresponds to its value. Overall, ticking effectively reduces time complexity analysis to a size analysis of the threaded clock.

Ticking of a program can itself be understood as a two phase process. In the first phase, the body $r$ of each equation f $p_1 \cdots p_k = r$ is transformed into a very specific let-normalform:

$$\textbf{(let-normalform)} \qquad e ::= x \mid \mathsf{let}\ x = s\ \mathsf{in}\ e \mid \mathsf{let}\ x_1 = x_2\ x_3\ \mathsf{in}\ e\ ,$$

for variables $x_i$ and $s \in \mathcal{F} \cup C$. This first step makes the evaluation order explicit, without changing program semantics. On this intermediate representation, it is then trivial to thread through a global counter. Instrumenting the program this way happens in the second stage. Each $k$-ary function f is extended with an additional clock-parameter, and this clock-parameter is passed through the right-hand side of each defining equation. The final clock value is then increased by one. This results in the definition of the instrumented function $\hat{\mathsf{f}}_k$. Intermediate functions $\hat{\mathsf{f}}_i$ $(0 \leq i < k)$ deal with partial application. Compare Figure 6 for an example.

Throughout the following, we fix a *pair-free program* P, i.e. P neither features pair constructors nor destructors. Pairs are indeed only added to our small programming language to conveniently facilitate ticking. The following definition introduces the ticking transformation formally. Most important, $\langle s^\tau\rangle^z_K$ simultaneously applies the two aforementioned stages to the term $s$. The variable $z$ presents the initial time. The transformation is defined in continuation passing style. Unlike a traditional definition, the continuation $K$ takes as input not only the result of evaluating $s$, but also the updated clock. It thus receives two arguments, viz two terms of type $\langle\tau\rangle$ and $\mathsf{C}$, respectively.

*Definition 7.1 (Ticking).* Let P be a program over constructors $C$ and functions $\mathcal{F}$. Let $\mathsf{C} \notin \mathcal{B}$ be a fresh base type, the *clock type*.

1. To each simple type $\tau$, we associate the following *ticked type* $\langle\tau\rangle$:
$$\langle\mathsf{B}\rangle := \mathsf{B} \qquad \langle\tau_1 \times \tau_2\rangle := \langle\tau_1\rangle \times \langle\tau_2\rangle \qquad \langle\tau_1 \to \tau_2\rangle := \langle\tau_1\rangle \to \mathsf{C} \to \langle\tau_2\rangle \times \mathsf{C}$$
2. The set $\hat{C}$ of *ticked constructors* contains a symbol $\mathsf{Z}^{\mathsf{C}}$, a symbol $\mathsf{T}^{\mathsf{C}\to\mathsf{C}}$, the *tick*, and for each constructor $\mathsf{C}^{\tau_1\to\cdots\to\tau_k\to\mathsf{B}}$ a new constructor $\hat{\mathsf{C}}^{\langle\tau_1\rangle\to\cdots\to\langle\tau_k\rangle\to\mathsf{B}}$.
3. The set $\hat{\mathcal{F}}$ of *ticked functions* contains for each $s^{\tau_1\to\cdots\to\tau_i\to\tau} \in \mathcal{F} \cup C$ and $0 \leq i \leq \mathrm{ar}(s)$ a new function $\hat{\mathsf{s}}_i^{\langle\tau_1\rangle\to\cdots\to\langle\tau_i\rangle\to\mathsf{C}\to\langle\tau\rangle\times\mathsf{C}}$.
4. For each variable $x^\tau$, we assume a dedicated variable $\hat{x}^{\langle\tau\rangle}$.
5. We define a translation from (non-ground) values $u^\tau$ over $C$ to (non-ground) values $\hat{u}^{\langle\tau\rangle}$ over $\hat{C}$ as follows.
$$\hat{u} := \begin{cases} \hat{x} & \text{if } u = x \in \mathcal{X}, \\ \hat{\mathsf{s}}_k\ \hat{u}_1\ \cdots\ \hat{u}_k & \text{if } u = s\ u_1\ \cdots\ u_k, s \in \mathcal{F} \cup C \text{ and } k < \mathrm{ar}(s), \\ \hat{\mathsf{C}}\ \hat{u}_1\ \cdots \hat{u}_{\mathrm{ar}(\mathsf{C})} & \text{if } u = \mathsf{C}\ u_1\ \cdots\ u_{\mathrm{ar}(\mathsf{C})}. \end{cases}$$

6. We define a translation from terms over $\mathcal{F} \cup C$ to terms in *ticked let-normalform* over $\hat{\mathcal{F}}$ as follows. Let $\mathsf{tick}\ x\ z = (x, \mathsf{T}\ z)$. For a term $s$ and variable $z^{\mathsf{C}}$ we define $\langle s \rangle^z := \langle s \rangle^z_{\mathsf{tick}}$, where

$$\langle s^\tau \rangle^{z_i}_K := \begin{cases} K\ \hat{s}\ z_i & \text{if } s \text{ is a variable,} \\ \mathsf{let}\ (x^{\langle \tau \rangle}, z^{\mathsf{C}}_{i+1}) = \hat{\mathsf{s}}_0\ z_i\ \mathsf{in}\ K\ x\ z_{i+1} & \text{if } s \in \mathcal{F} \cup C, \\ \langle s_1^{\rho \to \tau} \rangle^{z_i}_{K_1} & \text{if } s = s_1^{\rho \to \tau}\ s_2^\rho\ , \end{cases}$$

where in the last clause, $K_1\ x_1^{\langle \rho \to \tau \rangle}\ z_j^{\mathsf{C}} := \langle s_2^\rho \rangle^{z_j}_{(K_2\ x_1)}$ and $K_2\ x_1^{\langle \rho \to \tau \rangle}\ x_2^{\langle \rho \rangle}\ z_k^{\mathsf{C}} := \mathsf{let}\ (x^{\langle \tau \rangle}, z_l^{\mathsf{C}}) = x_1\ x_2\ z_k\ \mathsf{in}\ K\ x\ z_l$. All variables introduced by let-expressions are supposed to be fresh.

7. The *ticked* program $\hat{\mathsf{P}}$ consists of the following equations:
   1. For each equation $\mathsf{f}\ p_1 \cdots p_{\mathsf{ar}(\mathsf{f})} = r$ in $\mathsf{P}$, the *translated equation*
   $$\hat{\mathsf{f}}_{\mathsf{ar}(\mathsf{f})}\ \hat{p}_1 \cdots \hat{p}_{\mathsf{ar}(\mathsf{f})}\ z = \langle r \rangle^z\ ,$$
   2. for all $s \in \mathcal{F} \cup C$ and $0 \leq i < \mathsf{ar}(s)$, an *auxiliary equation*
   $$\hat{\mathsf{s}}_i\ x_1 \cdots x_i\ z = (\hat{\mathsf{s}}_{i+1}\ x_1 \cdots x_i, z)\ ,$$
   3. for all $\mathsf{C} \in C$, an *auxiliary equation*
   $$\hat{\mathsf{C}}_{\mathsf{ar}(\mathsf{C})}\ x_1 \cdots x_{\mathsf{ar}(\mathsf{C})}\ z = (\hat{\mathsf{C}}\ x_1 \cdots x_{\mathsf{ar}(\mathsf{C})}, z)\ .$$

   If $s \to_{\hat{\mathsf{P}}} t$, then we also write $s \xrightarrow{\mathsf{t}}_{\hat{\mathsf{P}}} t$ and $s \xrightarrow{\mathsf{a}}_{\hat{\mathsf{P}}} t$ if the step from $s$ to $t$ follows by a translated (case 1) and auxiliary equation (cases 2 and 3), respectively.

Our main theorem from this section states that whenever $\hat{\mathsf{P}}$ is well-type under an interpretation $\mathcal{J}$, thus in particular $\hat{\mathtt{main}}_k$ receives a type $\forall \vec{i} j.\mathsf{B}_{i_1} \to \cdots \to \mathsf{B}_{i_k} \to \mathsf{C}_j \to \mathsf{B}_a \times \mathsf{C}_b$, then the running time of $\mathsf{P}$ on inputs of size $\vec{i}$ is bounded by $[\![ b\{0/j\} ]\!]_{\mathcal{J}}$. This is proven in two steps. In the first step, we show a precise correspondence between reductions of $\mathsf{P}$ and $\hat{\mathsf{P}}$. This correspondence in particular includes that the clock carried around by $\hat{\mathsf{P}}$ faithfully represents the execution time of $\mathsf{P}$. In the second step, we then use the subject reduction theorem to conclude that the index $b$ in turn estimates the size, and thus value, of the threaded clock.

## 7.1 The Ticking Simulation

The ticked program $\hat{\mathsf{P}}$ operates on very specific terms, viz, terms in let-normal form enriched with clocks. The notion of *ticked let-normalforms* over-approximates this set. This set of terms is generated from $s \in \mathcal{F} \cup C$ and $k < \mathsf{ar}(s)$ inductively as follows.

$$\begin{array}{ll} \textbf{(clock terms)} & c ::= z^{\mathsf{C}} \mid \mathsf{Z} \mid \mathsf{T}\ c\ , \\ \textbf{(ticked let-normalform)} & e, f ::= (\hat{u}, c) \mid \hat{\mathsf{s}}_k\ \hat{u}_1 \cdots \hat{u}_k\ c \mid \mathsf{let}\ (x, z) = e\ \mathsf{in}\ f\ . \end{array}$$

Not every term generated from this grammar is legitimate. In a term $\mathsf{let}\ (x, z) = e\ \mathsf{in}\ f$, we require that the two let-bound variables $x, z$ occur exactly once free in $f$. Moreover, the clock variable $z$ occurs precisely in the *head* of $f$. Here, the head of a term in ticked let-normalform is given recursively as follows. In $\mathsf{let}\ (x, z) = e\ \mathsf{in}\ f$, the head position is the one of $e$. In the two other cases, the terms are itself in head position. This ensures that the clock is suitably wired, compare Figure 6. Throughout the following, we assume that every term in ticked let-normalform satisfies these criteria. This is justified, as terms in ticked let-normalform are closed under $\hat{\mathsf{P}}$-reductions, a consequence of the particular shape of right-hand sides in $\hat{\mathsf{P}}$.

As a first step towards the simulation lemma, we define a translation $[e]$ of the term $e$ in ticked let-normalform to a pair, viz, a terms of $\mathsf{P}$ and a clock term. We write $[e]_1$ and $[e]_2$ for the first and second component of $[e]$, respectively. The translation is defined by recursion on $e$ as follows.

$$[e] ::= \begin{cases} (u, c) & \text{if } e = (\hat{u}, c), \\ (s\ u_1\ \cdots\ u_k, c) & \text{if } e = \hat{\mathsf{s}}_k\ \hat{u}_1\ \cdots\ \hat{u}_k\ c \text{ where } s \in \mathcal{F} \cup C, \\ [e_2]\{[e_1]_1/x, [e_1]_2/z\} & \text{if } e = \mathsf{let}\ (x, z) = e_1\ \mathsf{in}\ e_2. \end{cases}$$

Lemma 7.2. *Let $e$ be a term in ticked let-normalform. The following holds:*
1. *$e \xrightarrow{\mathsf{t}}_{\hat{\mathsf{P}}} f$ implies $[e]_1 \to_{\mathsf{P}} [f]_1$ and $[f]_2 = \mathsf{T}\ [e]_2$; and*
2. *$e \xrightarrow{\mathsf{a}}_{\hat{\mathsf{P}}} f$ implies $[e]_1 = [f]_1$ and $[f]_2 = [e]_2$; and*
3. *if $[e]_1$ is reducible with respect to $\mathsf{P}$, then $e$ is reducible with respect to $\hat{\mathsf{P}}$.*

The first two points of Lemma 7.2 immediately yield that given a $\hat{\mathsf{P}}$-reduction, this reduction corresponds to a P-reduction. In particular, the lemma translates a reduction

$$\hat{\mathtt{main}}_k\ \hat{d}_1\ \cdots\ \hat{d}_k\ \mathsf{Z} \xrightarrow{\mathsf{t}}_{\hat{\mathsf{P}}} \cdot \xrightarrow{\mathsf{a}}{}^{*}_{\hat{\mathsf{P}}} e_1 \xrightarrow{\mathsf{t}}_{\hat{\mathsf{P}}} \cdot \xrightarrow{\mathsf{a}}{}^{*}_{\hat{\mathsf{P}}} \cdots \xrightarrow{\mathsf{t}}_{\hat{\mathsf{P}}} \cdot \xrightarrow{\mathsf{a}}{}^{*}_{\hat{\mathsf{P}}} e_\ell\ ,$$

to

$$[\hat{\mathtt{main}}_k\ \hat{d}_1\ \cdots\ \hat{d}_k]_1 = \mathtt{main}\ d_1\ \cdots\ d_k \to_{\mathsf{P}} [e_1]_1 \to_{\mathsf{P}} \cdots \to_{\mathsf{P}} [e_\ell]_1\ ,$$

where moreover, $[e_\ell]_2 = \mathsf{T}^\ell\ \mathsf{Z}$. In the following, let us abbreviate $\xrightarrow{\mathsf{t}}_{\hat{\mathsf{P}}} \cdot \xrightarrow{\mathsf{a}}{}^{*}_{\hat{\mathsf{P}}}$ by $\to_{\mathsf{t/a}}$. This, however, is not enough to show that $\hat{\mathsf{P}}$ simulates P. It could very well be that $\hat{\mathsf{P}}$ gets stuck at $e_\ell$, whereas the corresponding term $[e_\ell]_1$ is reducible. Lemma 7.2(3) verifies that this is indeed not the case. Another, minor, complication that arises is that $\hat{\mathsf{P}}$ is indeed not able to simulate *any* P-reduction. Ticking explicitly encodes a left-to-right reduction, $\hat{\mathsf{P}}$ can thus only simulate left-to-right, call-by-value reductions of P. However, Proposition 4.1 clarifies that left-to-right is as good as any reduction order. To summarise:

Theorem 7.3 (Simulation Theorem — Soundness and Completeness). *Let* P *be a program whose* `main` *function is of arity $k$.*
1. ***Soundness**: If $\hat{\mathtt{main}}_k\ \hat{d}_1\ \cdots\ \hat{d}_k\ \mathsf{Z} \xrightarrow{\mathsf{t/a}}{}^{\ell}_{\hat{\mathsf{P}}} e$ then $\mathtt{main}\ d_1\ \cdots\ d_k \to^{\ell}_{\mathsf{P}} t$ where moreover, $[e]_1 = t$ and $[e]_2 = \mathsf{T}^\ell\ \mathsf{Z}$.*
2. ***Completeness**: If $\mathtt{main}\ d_1\ \cdots\ d_k \to^{\ell}_{\mathsf{P}} s$ then there exists an alternative reduction $\mathtt{main}\ d_1\ \cdots\ d_k \to^{\ell}_{\mathsf{P}} t$ such that $\hat{\mathtt{main}}_k\ \hat{d}_1\ \cdots\ \hat{d}_k\ \mathsf{Z} \xrightarrow{\mathsf{t/a}}{}^{\ell}_{\hat{\mathsf{P}}} e$ where moreover, $[e]_1 = t$ and $[e]_2 = \mathsf{T}^\ell\ \mathsf{Z}$.*

## 7.2 Time Complexity Analysis

As corollary of the Simulation Theorem, essentially through Subject Reduction, we finally obtain our main result.

Theorem 7.4. *Suppose $\hat{\mathsf{P}}$ is well-typed under the interpretation $\mathcal{J}$, where data-constructors, including the clock constructor* T, *are given an additive type and where $\hat{\mathtt{main}}_k :: \forall \vec{ij}.\mathsf{B}_{i_1} \to \cdots \to \mathsf{B}_{i_k} \to \mathsf{C}_j \to \mathsf{B}_a \times \mathsf{C}_b$. The runtime complexity of* P *is bounded from above by $rc(i_1, \ldots, i_k) := [\![ b\{0/j\} ]\!]_{\mathcal{J}}$.*

In the proof of this theorem, we use actually a strengthening of Corollary 5.9. When a term $e$ in ticked let-normal form is given a type $\mathsf{B}_a \times \mathsf{C}_b$, then $b$ accounts for the size of $[e]_2$, even if $e$ is not in normal form.

## 8 PROTOTYPE AND EXPERIMENTAL RESULTS

We have implemented the discussed inference machinery in a prototype, dubbed `HoSA`.[2] This tool performs a fully automatic sized type inference on the typed language given in Section 4, extended with polymorphic types and inductive data type definitions as presented in examples earlier on.

[2]Available from http://cl-informatik.uibk.ac.at/~zini/software/hosa/.

$$\frac{}{\alpha \sqsubseteq_{\mathcal{J}} \alpha} \qquad \frac{a \leq_{\mathcal{J}} b \quad \rho_i \sqsubseteq_{\mathcal{J}} \rho_i' \ (i \text{ in positive position}) \quad \rho_i' \sqsubseteq_{\mathcal{J}} \rho_i \ (i \text{ in negative position})}{\mathsf{D}_a\ \rho_1 \cdots \rho_n \sqsubseteq_{\mathcal{J}} \mathsf{D}_b\ \rho_1' \cdots \rho_n'}$$

Fig. 7. Subtyping rules in the extended system.

These extension are already present in the canonical system from Hughes et al. [1996], and help not only towards modularity of the analysis, but enable also a more fine-grained capture of sizes.

Thus, in our implementation, the language of types is extended with type variables $\alpha$, that range over sized types, and $n$-ary data type constructors $\mathsf{D}$. Each such data constructor is associated with $m$ distinct constructors $\mathsf{C}_i :: \forall\alpha_1 \ldots \alpha_n.\ \tau_1 \to \cdots \tau_{k_i} \to \mathsf{D}\ \alpha_1 \ldots \alpha_n$. To accommodate these extensions to the type language, two main changes are necessary to our type system. First, the subtyping relation has to be adapted, to account for type variables and $n$-ary data type constructors, see Figure 7. Notice that in the second rule, the variance of arguments, given by the types of the corresponding constructors, are taken into account. Second, the type system has to be extended with the usual rule for instantiation of type variables. Also, some auxiliary definitions, noteworthy the one of canonicity, have to be suited in the obvious way.

In the following, we discuss our implementation, and then consider some examples that highlight the strength and limitations of our approach.

## 8.1 Technical Overview on the Prototype

Our tool HoSA is implemented in Haskell. Overall, the tool required just a moderate implementation effort. HoSA itself consists of approximately 2.000 lines of code. Roughly half of this code is dedicated to sized type inference, the other half is related to auxiliary tasks such as parsing etc. Along with HoSA, we have written a constraint solver, called GUBS. GUBS is also implemented in Haskell and weighs also in at around 2.000 lines of code.

In the following, we shortly outline the main execution stages of HoSA. The overall process is also exemplified in Figure 8 on the function prependAll, which prepends a given list to all elements of it second argument, itself a list of lists. This function is defined in terms of map and append, see Figure 8a for the definition.

*Hindley-Milner Type Inference and Specialisation.* As a first step, for each function in the given program a most general polymorphic type is inferred. Should type inference fail, our prototype will abort the analysis with a corresponding error message. As shortly discussed in Section 2, it is not always possible to decorate the most general type for higher-order combinators, such as foldr or map, with size information. Indeed, in the example from Figure 2 on page 5, we have specialised the most general type of foldr. Our implementation performs such a specialisation automatically. Of course, types cannot be specialised arbitrarily. Rather, our implementation computes for each higher-order combinator the least general type that is still general enough to cover all calls to the particular function. Technically, this is achieved via anti-unification and preserves well-typedness of the program. Should specialisation still yield a type that is too general for size annotation, our tool is also capable of duplicating the combinator, introducing a new function per call-site. This will then allow size annotations suitable for the particular call, at the expense of increased program size. With respect to prependAll, our implementation specialises the type of the supplied function in the declaration of map to match the call in Line 8 in Figure 8a.

*Ticking.* By default, our tool will perform the ticking transformation from Section 7 on the program obtained in the previous step, thereby enabling runtime analysis, the main motivating

```
map :: ∀α. (List α → List α) → List (List α) → List (List α)
map f [] = []
map f (x : xs) = f x : map f xs
append :: ∀α. List α → List α → List α
append []       ys = ys
append (x : xs) ys = x : append xs ys
prependAll :: ∀α. List α → List (List α) → List (List α)
prependAll xs = map (append xs)
```

(a) Function prependAll and auxiliary definitions. Types have been specialised.

$$
\begin{array}{l}
\texttt{map} :: \forall\alpha.\ \forall ijk.\ (\forall l.\ \texttt{List}_l\ \alpha \to \texttt{List}_{\mathsf{f}_1(l,i)}\ \alpha) \to \texttt{List}_k\ (\texttt{List}_j\ \alpha) \to \texttt{List}_{\mathsf{f}_3(i,j,k)}\ (\texttt{List}_{\mathsf{f}_2(i,j,k)}\ \alpha)\\
\texttt{append} :: \forall\alpha.\ \forall ij.\ \texttt{List}_i\ \alpha \to \texttt{List}_j\ \alpha \to \texttt{List}_{\mathsf{f}_4(i,j)}\ \alpha\\
\texttt{prependAll} :: \forall\alpha.\ \forall ijk.\ \texttt{List}_i\ \alpha \to \texttt{List}_k\ (\texttt{List}_j\ \alpha) \to \texttt{List}_{\mathsf{f}_6(i,j,k)}\ (\texttt{List}_{\mathsf{f}_5(i,j,k)}\ \alpha)
\end{array}
$$

(b) Template sized types assigned by HoSA to the main function prependAll and auxiliary functions.

$$
\begin{array}{rrrrr}
\mathsf{f}_1(E_{15},j) \leq \mathsf{f}_1(i,E_{18}) & \mathsf{f}_4(E_7,E_{10}) \leq \mathsf{f}_1(i,E_9) & E_{22} \leq \mathsf{f}_2(i,j,0) & i \leq E_{12} & i \leq E_8\\
E_{21} \leq \mathsf{f}_2(i,j,k+1) & \mathsf{f}_2(E_{18},E_{20},E_{19}) \leq E_{21} & 0 \leq \mathsf{f}_3(i,j,0) & i \leq E_{13} & i \leq E_{10}\\
E_{21}+1 \leq \mathsf{f}_3(i,j,k+1) & \mathsf{f}_3(E_{18},E_{20},E_{19}) \leq E_{21} & j \leq \mathsf{f}_4(0,j) & i \leq E_{17} & i \leq E_{11}\\
E_{14}+1 \leq \mathsf{f}_4(i+1,j) & \mathsf{f}_4(E_{15},i) \leq E_{16} & \mathsf{f}_2(E_9,E_{11},E_{10}) \leq \mathsf{f}_5(i,j,k) & i \leq E_{19} & \\
\mathsf{f}_3(E_9,E_{11},E_{10}) \leq \mathsf{f}_6(i,j,k) & i \leq E_7 & i \leq E_{20} & &
\end{array}
$$

(c) Second-order constraint system generated from HoSA.

$$
\begin{array}{rrrrr}
\mathsf{f}_1(i,j) := i+j & \mathsf{f}_2(i,j,k) := i+j & \mathsf{f}_3(i,j,k) := k & \mathsf{f}_4(i,j) := i+j & \mathsf{f}_5(i,j,k) := i+j\\
\mathsf{f}_6(i,j,k) := k & \mathsf{f}_7(i) := i & \mathsf{f}_8(i) := i & \mathsf{f}_9(i) := i & \mathsf{f}_{10}(i) := i\\
\mathsf{f}_{11}(i) := i & \mathsf{f}_{12}(i) := i & \mathsf{f}_{13}(i) := i & \mathsf{f}_{14}(i,j) := i+j & \mathsf{f}_{15}(i) := i\\
\mathsf{f}_{16}(i,j,k) := i+j & \mathsf{f}_{17}(i) := i & \mathsf{f}_{18}(i) := i & \mathsf{f}_{19}(i) := i & \mathsf{f}_{20}(i) := i\\
\mathsf{f}_{21}(i,j,k,l) := k & \mathsf{f}_{22}() := 0 & & &
\end{array}
$$

(d) Model inferred by GUBS on the generated constraints.

$$
\begin{array}{l}
\texttt{map} :: \forall\alpha.\ \forall ijk.\ (\forall l.\ \texttt{List}_l\ \alpha \to \texttt{List}_{l+i}\ \alpha) \to \texttt{List}_k\ (\texttt{List}_j\ \alpha) \to \texttt{List}_k\ (\texttt{List}_{i+j}\ \alpha)\\
\texttt{append} :: \forall\alpha.\ \forall ij.\ \texttt{List}_i\ \alpha \to \texttt{List}_j\ \alpha \to \texttt{List}_{i+j}\ \alpha\\
\texttt{prependAll} :: \forall\alpha.\ \forall ijk.\ \texttt{List}_i\ \alpha \to \texttt{List}_k\ (\texttt{List}_j\ \alpha) \to \texttt{List}_k\ (\texttt{List}_{i+j}\ \alpha)
\end{array}
$$

(e) Inferred size type obtained by instantiating the template types with the model computed by GUBS.

Fig. 8. Sized type inference carried out by HoSA on prependAll.

application behind this work. For the sake of simplicity, ticking is not performed in the running example though.

*Annotation of Types with Index Terms.* To each function, an abstract, canonical sized type is then assigned by annotating the types inferred in the second stage with index terms. In essence, this is done by annotating polymorphic types $\forall\vec{\alpha}.\ \tau_1 \to \cdots \to \tau_k \to \tau$ (where $\tau$ is not a functional type) as follows: (i) if the argument type $\tau_i$ is a data type then it is annotated with a fresh variable, the arguments are annotated recursively; (ii) if the argument type $\tau_i$ is a functional type we proceed recursively, and close over all index variables occurring in the so obtained sized type, and (iii) we annotate the return type $\tau$ by an index term $\mathsf{f}(\vec{i})$. Here, the index symbol $\mathsf{f}$ is supposed fresh. The variables $\vec{i}$ collect on the one hand the free index variables occurring in argument types $\tau_i$. Moreover, for a functional type in argument position this sequence contains $m$ fresh index variables, for some fixed $m \in \mathbb{N}$, the *extra variables*. Types of constructors $\mathsf{C}$ are annotated similarly, except that the index $\mathsf{f}(\vec{i})$ of its return type is fixed to $\sum_{i \in \vec{i}} i + w$ where $w = 0$ if $\mathsf{C}$ is nullary, and $w = 1$ otherwise. The size index of constructors thus accounts for internal nodes in the tree representation of data values. This measure is seemingly ad-hoc, but turns out favourable, as the number of internal nodes relate to the number of recursive steps for functions defined by structural recursion. For instance, we have $\texttt{[]} :: \forall\alpha.\ \texttt{List}_0\ \alpha$, and $(:) :: \forall\alpha.\ \forall i.\ \alpha \to \texttt{List}_i\ \alpha \to \texttt{List}_{i+1}\ \alpha$.

With respect to `prependAll` and $m = 1$, annotated types are depicted in Figure 8b. Notice that the annotated type of `map` features the extra variable $i$. Extra variables enable the system to deal with *closures*, i.e. functionals that capture part of the environment. Such a closure is for instance created with append $xs$, on Line 8 in Figure 8a. Intuitively, extra variables index the size of the captured environment. We return to this point in a moment. For all of the examples that we considered, taking $m = 1$, i.e. adding a single extra variable in step (ii) above, is sufficient. It would be desirable to statically determine the number of necessary extra variables. This can likely be done with a simple form of data flow analysis, which is however beyond the scope of this work.

*Constraint Generation.* `HoSA` performs type inference as discussed in Section 6 based on the annotated types assigned in the previous step. The extension to the polymorphic type system with inductive data types poses no challenge. The extended subtyping rules from Figure 7 are straight forward to integrate within the machinery discussed in Section 6. It is also not difficult to adapt the rules (Var-I) and (Fun-I) from Figure 5b so that type variables $\alpha$ in polymorphic types are properly instantiated: suitable skeletons are already known at this stage, to turn them into suitable sized types our implementation decorates these with second-order index variables. For instance, suppose $(:) :: \forall\alpha.\ \forall i.\ \alpha \to \texttt{List}_i\ \alpha \to \texttt{List}_{i+1}\ \alpha$ is used to construct a list of naturals. Then this constructor will be typed as $(:) :: \texttt{Nat}_E \to \texttt{List}_j\ \texttt{Nat}_E \to \texttt{List}_{F+1}\ \texttt{Nat}_E$, i.e., the type variable $\alpha$ has been instantiated with $\texttt{Nat}_E$ and the index variable $i$ with $F$. This stage will result in a SOCP, which is then translated to a FOCP by skolemisation. On the function `prependAll`, this results in 22 constraints, see Figure 8c.

*Constraint Solving.* `HoSA` makes use of the external tool `GUBS` to find a suitable model for the FOCP, see Figure 8c. How this is done is explained in a moment. Note that the auxiliary functions $\mathsf{f}_7$—$\mathsf{f}_{22}$ were introduced by skolemization and correspond to $E_7$—$E_{22}$, respectively.

*Concretising Annotated Types.* In this final stage, `HoSA` combines the annotated types with the computed model, by unfolding index functions in template types according to the model. The resulting sized types are decorated by arithmetical expressions only, compare Figure 8e. This final result of the analysis is presented to the user.

Note that the extra variable $i$ in the annotated sized type of `map` is still present in its concrete size type. Indeed, the extra variable is crucial when we want to type the body of `prependAll`. Here, we first derive

$$xs : \texttt{List}_i\ \alpha \vdash \mathsf{map} : (\forall l.\ \texttt{List}_l\ \alpha \to \texttt{List}_{l+i}\ \alpha) \to \texttt{List}_k\ (\texttt{List}_j\ \alpha) \to \texttt{List}_k\ (\texttt{List}_{i+j}\ \alpha), \text{ and}$$
$$xs : \texttt{List}_i\ \alpha \vdash \mathsf{append}\ xs : \texttt{List}_l\ \alpha \to \texttt{List}_{l+i}\ \alpha\ .$$

Therefore, by rule (App) we get

$$xs : \texttt{List}_i\ \alpha \vdash \mathsf{map}\ (\mathsf{append}\ xs) : \texttt{List}_k\ (\texttt{List}_j\ \alpha) \to \texttt{List}_k\ (\texttt{List}_{i+j}\ \alpha)\ ,$$

as demanded by the well-typedness of `appendAll`. In a way, the extra variable in the declaration is used to keep track of the length of the list $xs$ captured by the term append $xs$, which in turn, is relayed through the typing of `map` to the result type $\texttt{List}_k\ (\texttt{List}_{i+j}\ \alpha)$.

## 8.2 Constraint Solving

As for many sized type systems, constraint solving is also a central stage in our approach and appears in the form of model-synthesis for FOCPs. Strength and precision of the overall analysis is directly related to this stage. Sized type inference is undecidable, as a consequence of Corollary 6.9, model-synthesis is in general undecidable too.

Synthesizing functions that obey certain set of constraints, as expressed for instance through FOCPs, is a fundamental task in program analysis. Consequently, the program verification community introduced various techniques in this realm. One popular approach relies on *LP solvers*, compare e.g., [Podelski et al. 2004]. This approach is effective, moreover, yields tight models. However, it is usually restricted to the synthesis of linear functions. This is often sufficient for termination analysis, where one is foremost interested that recursion parameters decrease. In our context however, this rules out the treatment of all programs that admit a non-linear runtime. Another approach rests on solving (non-deterministic) *recurrence relations*. To this end, dedicated tools like PUBS [Albert et al. 2008] have been developed, which are capable of synthesising non-linear functions. Recurrence relations are of limited scope in our context however. For instance, function composition cannot be directly expressed in this formalism.

To overcome these limitations, we have developed the GUBS *upper bound solver* (GUBS for short), an open source tool dedicated to the synthesis of models for FOCPs. This tool is capable of synthesising models formed from linear and non-linear *max-polynomials* over the naturals. GUBS itself is heavily inspired by methods developed in the context of rewriting. The rewriting community pioneered the synthesising of polynomial interpretations, see e.g., [Fuhs et al. 2007, 2008] or the survey of Péchoux et al. [2013] on *sup-interpretations*, a closely related topic. In this line of works, the problem is reduced to the satisfiability in the quantifier-free fragment of the theory of non-linear integer arithmetic. Dedicated to the latter, MiniSmt [Zankl et al. 2010] has been developed. Moreover, state-of-the-art SMT solvers such as Z3 [Mendonça de Moura et al. 2008] can effectively treat quantifier free non-linear integer arithmetic nowadays.

The main novel aspect of GUBS is the modular approach it rest upon, which allowed us to integrate besides the aforementioned reduction various syntactic simplification techniques, and a per-SCC analysis. In what follows, we provide a short outline of two central methods implemented in GUBS.

*Synthesis of Models via SMT.* Conceptually, we follow the method presented by Fuhs et al. [2008]. In this approach, each $k$-ary symbol f is associated with a $k$-ary *template max-polynomial*, through a *template interpretation* $\mathcal{A}$. Here, a template max-polynomial is an expression formed from $k$ variables as well as undetermined coefficient variables $\vec{c}$, the *template coefficients*, and the binary connectives $(+)$, $(\cdot)$ and max, corresponding to addition, multiplication and the maximum function, respectively. For instance, a linear template for a binary symbol f is

$$[\![\mathsf{f}]\!]^{\alpha}_{\mathcal{A}}(x, y) = \max(\mathsf{c}_1 \cdot x + \mathsf{c}_2 \cdot y + \mathsf{c}_3, \mathsf{d}_1 \cdot x + \mathsf{d}_2 \cdot y + \mathsf{d}_3) \ .$$

To find a concrete model for a SOCP $\Phi$ based on the template interpretation $\mathcal{A}$, GUBS is searching for concrete values $\vec{n} \in \mathbb{N}$ for the coefficient variables $\vec{c}$ so that

$$\forall a \le b \in \Phi.\ \forall \alpha : \mathcal{V} \to \mathbb{N}.\ [\![a]\!]^{\alpha}_{\mathcal{A}} \le [\![b]\!]^{\alpha}_{\mathcal{A}} \ ,$$

holds. Once these have been found, an interpretation $\mathcal{J}$ with $\mathcal{J} \models \Phi$ is obtained by substituting $\vec{n}$ for $\vec{c}$ in $\mathcal{A}$. This search is performed itself in two steps. First, the maximum operator is eliminated in accordance to the following two rules. Here, $C$ represents an arbitrary context over max-polynomials.

$$C[\max(a_1, a_2)] \le b \Longrightarrow C[a_1] \le b \wedge C[a_2] \le b \ , \ \ a \le C[\max(b_1, b_2)] \Longrightarrow a \le C[b_1] \vee a \le C[b_2] \ .$$

```
rêv₀ :: C → (α → C → (List α → C → List α × C) × C) × C
rêv₀ z = (rêv₁ , z)

rêv₁ :: List α → C → (List α → C → List α × C) × C
rêv₁ xs z = (rêv₂ xs , z)

rêv₂ :: List α → List α → C → List α × C
rêv₂ []       ys z = (ys , T z)
rêv₂ (x : xs) ys z = let (x₁ , z₁) = rêv₀ z in
                     let (x₂ , z₂) = x₁ xs z₁ in
                     let (x₃ , z₃) = x₂ (x : ys) z₂ in (x₃ , T z₃)

revêrse₀ :: C → (List α → C → List α × C) × C
revêrse₀ z = (revêrse₁ , z)

revêrse₁ :: List α → C → List α × C
revêrse₁ xs z = let (x₁ , z₁) = rêv₀ z in
                let (x₂ , z₂) = x₁ xs z₁ in
                let (x₃ , z₃) = x₂ [] z₄ in (x₃ , T z₃)
```

Fig. 9. Ticked reverse function.

Intuitively, this elimination procedure is sound as we are dealing with weakly monotone expressions only. Once all occurrences of max are eliminated, the resulting formula is reduced to *diophantine constraints* over the coefficient variables $\vec{c}$, via the so-called *absolute positiveness check*, see also the work of Fuhs et al. [2007]. The diophantine constraints are then given to an SMT-solver that support quantifier-free non-linear integer arithmetic, from its assignment and the initially fixed templates GUBS then computes concrete interpretations. To get more precise bounds, GUBS minimises the obtained model by making use of the incremental features of current SMT-solvers, essentially by putting additional constraints on coefficients $\vec{c}$.

The main limitation of this approach is that the shape of interpretations is fixed to that of templates, noteworthy, the degree of the interpretation is fixed in advance. As the complexity of the absolute positiveness check depends not only on the size of the given constraint system but to a significant extent also on the degree of interpretation functions, our implementation searches iteratively for interpretations of increasing degrees. Also notice that our max-elimination procedure is incomplete, for instance, it cannot deal with the constraint $x + y \leq \max(2x, 2y)$, which is reduced to $x + y \leq 2x \vee x + y \leq 2y$. In contrast, Fuhs et al. [2008] propose a complete procedure to eliminate the maximum operator. However, our experimental assessment concluded that this encoding introduces too many auxiliary variables, which turned out as a significant bottleneck.

*Separate SCC Analysis.* Synthesis of models via SMT gets impractical on large constraint systems. To overcome this, GUBS divides the given constraint system $\Phi$ into its *strongly connected components* (*SCCs* for short) $\Phi_1, \ldots, \Phi_n$, topologically sorted bottom-up, and finds a model for each SCC $\Phi_i$ iteratively. Here, the underlying *call graph* is formed as follows. The nodes are given by the constraints in $\Phi$. Let $a_1 \leq b_1$ to $a_2 \leq b_2$ be two constraints in $\Phi$, where wlog. $b_1 = C[\mathsf{f}_1(\vec{c}_1), \ldots, \mathsf{f}_n(\vec{c}_n)]$ for a context $C$ without index symbols. Then there is an edge from $a_1 \leq b_1$ to $a_2 \leq b_2$ if any of the symbols occurring in $\vec{c}_1, \ldots, \vec{c}_n, a_1$ occurs in $b_2$. The intuition is that once we have found a model for all the successors $a_2 \leq b_2$ of $a_1 \leq b_1$, we can interpret the arguments $\vec{b_i}$ and the left-hand side $a_1$ within this model. We can then extend this model by finding a suitable interpretation for $\mathsf{f}_1, \ldots, \mathsf{f}_n$, thereby obtaining a model that satisfies $a_1 \leq b_1$.

## 8.3 Experimental Evaluation

We will now look at how HoSA deals with some examples, including those mentioned in the paper. Here, we also relate the strength and precision of tool to that of HoCA [Avanzini et al. 2015] and

```
id :: α → α
id z = z
comp :: (β → γ) → (α → β) → α → γ
comp f g z = f (g z)
walk :: List α → (List α → List α)
walk []     = id
walk (x:xs) = comp (walk xs) ((:) x)
reverse :: List α → List α
reverse xs = walk xs []
```

Fig. 10. List reversal via difference lists. This is the motivating example from Avanzini et al. [2015].

RAML [Hoffmann et al. 2017]. To the best of our knowledge, these constitute the only two state-of-the-art, freely available, tools for the automated resource analysis of higher-order programs.

*Tail-Recursive List Reversal.* Reconsider the version of list reversal presented in Figure 1 on page 4. This is an example that could not be handled by the original sized type system introduced by Hughes et al. [1996]. In Figure 9 we show the corresponding ticked program. For brevity, the auxiliary definitions derived from the list constructors have been inlined. Our tool infers

$$\hat{\mathtt{reverse}}_1 :: \forall\alpha.\ \forall ij.\mathtt{List}_i\ \alpha \to \mathtt{C}_j \to \mathtt{List}_i\ \alpha \times \mathtt{C}_{2+i+j}\ .$$

Thus, by setting the starting clock to zero, i.e. assuming $j = 0$, HoSA derives the bound $2 + i$ on the runtime of reverse. Taking into account that the auxiliary function rev performs $i + 1$ steps on a given list of length $i$, it is clear that the derived runtime bound for reverse is tight. Similar, the derived bound for the size of the returned list is optimal. The optimal linear bound could also be found with HoCA and RAML.

*Reverse with Difference Lists.* In Figure 10 we depict the motivating example from Avanzini et al. [2015]. Here, an alternative definition of list reversal based on *difference lists*, a data structure for representing lists with a constant concatenation operation, is given. In a functional setting, difference lists can be represented as functions $d : \mathtt{List}\ \alpha \to \mathtt{List}\ \alpha$, with $d$ denoting the list $ys$ such that $d\ xs = \mathsf{append}\ ys\ xs$. Difference lists are commonly used in functional programming in order to avoid the unnecessary runtime overhead in expressions such as $(\mathsf{append}\ (\mathsf{append}\ xs\ ys)\ zs)$. On this example, HoSA succeeds with the following declaration

$$\hat{\mathtt{reverse}}_1 :: \forall\alpha.\ \forall ij.\ \mathtt{List}_i\ \alpha \to \mathtt{C}_j\alpha \to \mathtt{List}_i\ \alpha \times \mathtt{C}_{3+2\cdot i+j}\ ,$$

confirming that also this version of reverse exhibits a linear runtime complexity. An asymptotic linear bound can be derived by HoCA, but not by RAML. The latter can be rectified by using a contrived version $\mathsf{comp}'\ f\ x\ g\ y = f\ x\ (g\ y)$ of function composition and suitably adapting the body of walk. Then, RAML can infer the bound $3 + 9 \cdot i$ on the runtime of reverse.

*Product.* Our tool infers

$$\hat{\mathtt{product}}_2 :: \forall ijk.\mathtt{List}_i\ \alpha \to \mathtt{List}_j\ \beta \to \mathtt{C}_k \to (\mathtt{List}_{i\cdot j}\ (\alpha \times \beta)) \times \mathtt{C}_{2+3\cdot i+2\cdot i\cdot j+k}\ ,$$

where $\hat{\mathtt{product}}_2$ corresponds to the ticked version of the function product from Figure 2. The estimated size of the resulting list is precise, the computed runtime is asymptotically precise. Notice that the latter bound takes also the evaluation of the anonymous functions into account. An asymptotic precise bound can be inferred with HoCA, but not with RAML.

```
data Nat = Z | S Nat

gt :: Nat → Nat → Bool
gt Z      y     = False
gt (S x) Z      = True
gt (S x) (S y) = gt x y

insert :: ∀α. (α → α → Bool) → α → List α → List α
insert ord x []       = x : []
insert ord x (y : ys) = if ord x y then y : insert ord x ys else x : y : ys

insertionSort :: ∀α. (α → α → Bool) → List α → List α
insertionSort ord []       = []
insertionSort ord (x : xs) = insert ord x (sort ord xs)

sortNat :: List α → List α
sortNat = insertionSort gt
```

Fig. 11. Insertion-sort on natural numbers.

```
data Queue α = Q [α] [α]

repair :: ∀α. Queue α → Queue α
repair (Q [] r)       = Q (reverse r) []
repair (Q (e : f) r) = Q (e : f) r

push :: ∀α. α → Queue α → Queue α
push x (Q f r) = repair (Q f (x : r))

pop :: ∀α. Queue α → α × Queue α
pop (Q [] r)       = error                 -- queue empty
pop (Q (e : f) r) = (e, repair (Q f r))

fromList :: ∀α. [α] → Queue α
fromList = foldr push (Q [] [])
```

Fig. 12. Functional queues.

*Insertion Sort.* In Figure 11 we present a version of insertion sort that is parameterised by the comparison operation. We have then specialised this function to a comparison on natural numbers. HoSA derives

$$\widehat{\mathtt{sortNat}}_2 :: \forall\alpha.\,\forall ijk.\,\mathtt{List}_i\,\mathtt{Nat}_j \to \mathtt{C}_k \to \mathtt{List}_i\,\mathtt{Nat}_j \times \mathtt{C}_{2+i^2\cdot j+2\cdot i^2+k}\ .$$

The computed runtime bound $2 + i^2 \cdot j + 2 \cdot i^2$ is precise, taking into account that gt is not a constant operation. It is worthy of note that the precise bound could only be inferred since HoSA is capable of inferring that insert *ord x ys*, given $x : \mathtt{Nat}_i$ and $ys : \mathtt{List}_k\ \mathtt{Nat}_j$ produces a list of type $\mathtt{List}_{k+1}\ \mathtt{Nat}_{\max(i,j)}$. This demonstrates that the limitation imposed by the linearity condition on canonical sized types can be overcome with the max operator. Both, HoCA and RAML, can give asymptotic precise bounds on this example. Concerning the former tool the bound $\mathrm{O}(i^3 + j^3)$, concerning the latter a runtime bound $3 - 4 \cdot i \cdot j + 4 \cdot i^2 \cdot j + 8i + 9i^2$, is derived.

*Quicksort.* We have also implemented a version of quicksort. This implementation uses the standard-combinator partition, to partition the given list into elements lesser and greater-equal to the pivot element, respectively. Our tool derives

$$\mathtt{partition} :: \forall\alpha.\,\forall i.\,(\alpha \to \mathtt{Bool}) \to \mathtt{List}_i\ \alpha \to \mathtt{List}_i\ \alpha \times \mathtt{List}_i\ \alpha\ .$$

This is indeed the most precise type that can be given to partition in our system. However, it is not precise enough to prove that quicksort runs in polynomial time. Here, one would need to prove that the length of the two resulting lists sum up to the length of the argument list. On the other hand, both RAML and HoCA can prove a quadratic bound on the runtime of quicksort.

*Functional Queues.* In Figure 12 we give an implementation of queues as defined by Okasaki et al. [1999]. A value `Q` $f$ $r$ represents the queue with initial segment $f$ and reversed remainder $r$. Enqueueing thus simply amounts to consing it to $r$, whereas dequeuing an element amounts to removing the head of $f$, whenever $f$ is non-empty. The latter is ensured by the auxiliary function `repair`, which is called whenever the queue is modified. Notice thus that both adding and removing an element from a queue has a linear worst case complexity, due to the call to `reverse` in the definition of `repair`. However, this cost armortises with the number of pushes. Our system derives

$$\widehat{\mathtt{fromList}}_1 :: \forall\alpha.\ \forall ij.\ \mathtt{List}_i\ \alpha \to \mathtt{C}_j \to \mathtt{Queue}_{1+i}\ \alpha \times \mathtt{C}_{2+i+5\cdot i^2+j}\ ,$$

and thus a quadratic runtime bound on `fromList`. In contrast, both HoCA and RAML derive an asymptotic precise linear bound. Concerning RAML, this is possible because of the underlying amortised analysis. HoCA derives the precise bound for a different reason: `fromList` is translated into two simple recursive definition, that turn a list $[x_1, \dots, x_n]$ directly into `Q` $[x_1]$ $[x_n, \dots, x_2]$, thereby in particular completely eliminating the problematic calls to `reverse` via `repair`.

*Prepend All.* Concerning the function `prependAll` from Figure 8a, HoSA infers

$$\widehat{\mathtt{prependAll}}_2 :: \forall\alpha.\ \forall ijk.\ \mathtt{List}_i\ \alpha \to \mathtt{List}_k\ (\mathtt{List}_j\ \alpha) \to \mathtt{C}_l \to \mathtt{List}_k\ \mathtt{List}_{1+i+j}\ \alpha \times \mathtt{C}_{2+i\cdot k+2\cdot k+l}\ .$$

The runtime of `prependAll` is thus correctly bounded by $2 + i \cdot k + 2 \cdot k$. Evaluating `prependAll` results in $(1 + j)$ calls to `map`, counting the base case and $j$ recursive calls. Each recursive call triggers the evaluation to `append`, itself performing $1 + i$ reduction steps. Taking into account that `prependAll` has to be unfolded first, we see that the inferred bound is indeed optimal.

Worthy of note, the example can also be handled by HoCA. However, HoCA is only able to infer an asymptotic quadratic bound. On the other hand, whereas RAML can produce asymptotic precise bound for `append` and `map`, it fails to analyse `prependAll` itself. RAML does not attribute potentials to functions, thus, it is assumed that the reduction of closures can be solely measured in terms of the formal parameter, but is independent from the captured environment. The compositional nature of the analysis underlying RAML comes at a price.

## 9 CONCLUSIONS

We have described a new system of sized types whose key features are an abstract index language, and higher-rank index polymorphism. This allows for some more flexibility compared to similar type systems from the literature. The introduced type system is proved to enjoy a form of type soundness, and to support a relatively complete type inference procedure, which has been implemented in our prototype tool HoSA. One key motivation behind this work is achieving a form of modular complexity analysis without sacrificing its expressive power. This is achieved by the adoption of a type system, which is modular and composable by definition. This is contrast to other methodologies like program transformations [Avanzini et al. 2015]. Noteworthy, modularity carries to some extent through to constraint solving. The SCCs in the generated constraint problem are in correspondence with the SCC of the call-graph in the input program, and are analysed independently.

Future work definitely includes refinements to our constraint solver GUBS. It would also be interesting to see how our overall methodology applies to different resource measures like heap size etc. Concerning heap size analysis, this is possible by ticking constructor allocations. It could also be worthwhile to integrate a form of amortisation in our system.

## REFERENCES

1984. *Polymorphic Type Schemes and Recursive Definitions.* Springer, Heidelberg, DE, 217–228.

1996. Proving the Correctness of Reactive Systems Using Sized Types. In *Proc. of 23rd POPL (POPL '96).* ACM, New York, NY, 410–423.

1999. *Purely Functional Data Structures.* Cambridge University Press, Oxford, UK.
2000. Resource Bound Certification. In *Proc. of 27th POPL.* ACM, New York, NY, 184–198.
2004. A Complete Method for the Synthesis of Linear Ranking Functions. In *Proc. 5th VMCAI.* ACM, New York, NY, 239–251.
2005. Decidability of Type-Checking in the Calculus of Algebraic Constructions with Size Annotations. In *Proc. of 14th CSL (LNCS)*, Vol. 3634. Springer, Heidelberg, DE, 135–150.
2007. A Program Logic for Resources. *TCS* 389, 3 (2007), 411–445.
2007. Practical Type Inference for Arbitrary-rank Types. *JFP* 17, 1 (2007), 1–82.
2007. SAT Solving for Termination Analysis with Polynomial Interpretations. In *Proc. of 10th SAT (LNCS)*, Vol. 4501. Springer, Heidelberg, DE, 340–354.
2008. Automatic Inference of Upper Bounds for Recurrence Relations in Cost Analysis. In *Proc. of 15th SAS.* Springer, Heidelberg, DE, 221–237.
2008. Lightweight Semiformal Time Complexity Analysis for Purely Functional Data Structures. In *Proc. of 35th POPL.* ACM, New York, NY, 133–144.
2008. Maximal Termination. In *Proc. of 19th RTA*, Vol. 5117. Springer, Heidelberg, DE, 110–125.
2008. *Space Cost Analysis Using Sized Types.* Ph.D. Dissertation. School of Computer Science, University of St Andrews.
2008. The Worst Case Execution Time Problem - Overview of Methods and Survey of Tools. *TECS* 7, 3 (2008), 1–53.
2008. Type-Based Termination with Sized Products. In *Proc. of 17th CSL (LNCS)*, Vol. 5213. Springer, Heidelberg, DE, 493–507.
2008. Z3: An Efficient SMT Solver. In *Proc. of 14th TACAS (LNCS)*, Vol. 4963. Springer, Heidelberg, DE, 337–340.
2009. On Constructor Rewrite Systems and the Lambda-Calculus. In *Proc. of 36th ICALP (LNCS)*, Vol. 5556. Springer, Heidelberg, DE, 163–174.
2009. Polynomial Size Analysis of First-Order Shapely Functions. *LMCS* 5, 2 (2009), 1–35.
2010. Closing the Gap Between Runtime Complexity and Polytime Computability. In *Proc. of 21st RTA (LIPIcs)*, Vol. 6. Dagstuhl, Saarbrücken, DE, 33–48.
2010. Satisfiability of Non-linear (Ir)rational Arithmetic. In *Proc. of 16th LPAR (LNCS)*, Vol. 6355. Springer, Heidelberg, DE, 481–500.
2010. Static Determination of Quantitative Resource Usage for Higher-order Programs. In *Proc. of 37th POPL.* ACM, New York, NY, 223–236.
2011. Linear Dependent Types and Relative Completeness. *LMCS* 8, 4 (2011), 1–44.
2011. Multivariate Amortized Resource Analysis. In *Proc. of 38th POPL.* ACM, New York, NY, 357–370.
2012. Resource Aware ML. In *Proc. of 24th CAV (LNCS)*, Vol. 7358. Springer, Heidelberg, DE, 781–786.
2013. On the Inference of Resource Usage Upper and Lower Bounds. *TOCL* 14, 3 (2013), 22(1–35).
2013. Synthesis of Sup-interpretations: A Survey. *TCS* 467 (2013), 30–52.
2013. The Geometry of Types. In *Proc. of 40th POPL.* ACM, New York, NY, 167–178.
2014. A Simple and Scalable Static Analysis for Bound Analysis and Amortized Complexity Analysis. In *Proc. of 26th CAV (LNCS)*, Vol. 8559. Springer, Heidelberg, DE, 745–761.
2014. Linear Dependent Types in a Call-by-value Scenario. *SCP* 84 (2014), 77–100.
2015. Analysing the Complexity of Functional Programs: Higher-Order Meets First-Order. In *Proc. of 20th ICFP.* ACM, New York, NY, 152–164.
2015. Automatic Static Cost Analysis for Parallel Programs. In *Proc. of 24th ESOP (LNCS)*, Vol. 9032. Springer, Heidelberg, DE, 132–157.
2015. Denotational Cost Semantics for Functional Languages with Inductive Types. In *Proc. of 20th ICFP.* ACM, New York, NY, 140–151.
2015. Type-based Amortized Resource Analysis with Integers and Arrays. *JFP* 25 (2015), e17.
2016. The Complexity of Interaction. In *Proc. of 43rd POPL.* ACM, New York, NY, 243–255.
2016. TcT: Tyrolean Complexity Tool. In *Proc. of 2nd TACAS (LNCS).* Springer, Heidelberg, DE, 407–423.
2016. Well-founded recursion with copatterns and sized types. *JFP* 26 (2016), e2.
2017. Towards Automatic Resource Bound Analysis for OCaml. In *Proc. of 44th POPL.* ACM, New York, NY, 359–373.

## ACKNOWLEDGMENTS

This work is partially supported by the ANR projects 12IS02001 PACE and 14CE250005 ELICA, as well as the FWF project J-3563.