Complexity Analysis by Polymorphic Sized Type Inference and Constraint Solving

Martin Avanzini[1] and Ugo Dal Lago[2,3]

[1] University of Innsbruck
[2] University of Bologna
[3] INRIA Sophia Antipolis

Abstract. 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 various key ingredients, and in particular an abstract index language and index polymorphism at higher ranks. A prototype implementation is available.

1 Introduction

Programs can be incorrect for very different reasons. Modern compilers are able to detect many syntactic errors, including type errors. When the error 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, finally, a program is functionally correct but performs quite poorly in terms of space and runtime behaviour, the bad thing is that even *defining* the property one would like the program to satisfy 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 programs are applied to before being executed. One then obtains WCET techniques [21], 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 its 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 [2,15,1,19].

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 few proposals [15,4,20], about 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* [14], and has been shown to be quite robust and amenable to automation [7]. 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. Estimating the size of intermediate results is also a crucial aspect of complexity analysis, and up to now, the only attempt of using sized types for complexity analysis is due to Vasconcelos [20], 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 [10].

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*, that is, 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 relative 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. [14]. 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 3, and will be developed in Section 4. Nameless functions (i.e. λ-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 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 sized types [14] and similar kinds of type systems.
- The polymorphic sized types 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 itself. Complexity analysis of a program P can however be seen as a size analysis of another program $\hat{\mathsf{P}}$ which computes not only P, but its complexity. This transformation, called the *ticking transformation*, has already been studied in similar settings [12], but this study has never been made systematic.
- Contrarily to many papers from the literature, we have taken care not only of constraint *inference*, but also of constraint *solving*. This has been done by building a prototype which implements type inference, and then relies on an external SMT solver to check the generated constraints for satisfiability. All this, together with some experimental results, are described in detail in Section 6.

An extended version with more details is available [3].

2 Examples and Related Work

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 prototypical system introduces by Hughes et. al. [14] and similar ones [7,20], 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 α. Each such datatype is associated with a set of typed *constructors*, below we will use the constructors $0 :: \mathtt{Nat}$, $\mathtt{Succ} :: \mathtt{Nat} \to \mathtt{Nat}$ for naturals, and the usual constructors $\mathtt{Nil} :: \forall\alpha.\ \mathtt{List}\ \alpha$ and $\mathtt{Cons} :: \forall\alpha.\ \alpha \to \mathtt{List}\ \alpha \to \mathtt{List}\ \alpha$ for lists. Sized types refine each such datatype into a *family* of datatypes indexed by *strictly positive* natural numbers, their *size*. E.g., to $\mathtt{Nat}$ and $\mathtt{List}\ \alpha$ we associate the families $\mathtt{Nat}_1, \mathtt{Nat}_2, \mathtt{Nat}_3, \ldots$ and $\mathtt{List}_1\ \alpha, \mathtt{List}_2\ \alpha, \mathtt{List}_3\ \alpha, \ldots$, respectively. An indexed datatype such as $\mathtt{List}_n\ \mathtt{Nat}_m$ then represents lists of size n, over naturals of size m. Here, the attributed size refers to the number of data constructors of the corresponding datatype. In particular, a list of type $\mathtt{List}_n\ \mathtt{Nat}_m$ is formed from (at most) $n-1$ constructors $\mathtt{Cons}$ and a constructor $\mathtt{Nil}$. Each element of this list, in turn, consists of at most m constructors.

A function $\mathtt{f}$ will then be given a polymorphic type $\forall\boldsymbol{\alpha}.\ \forall\boldsymbol{i}.\ \tau \to \rho$ with size-variables $\boldsymbol{i}$ ranging over sizes. Datatypes occurring in the types τ and ρ will

```
reverse :: ∀α. ∀i.List_i α → List_i α
reverse xs = rev xs Nil

rev :: ∀α. ∀ij.List_i α → List_j α → List_{(i−1)+j} α
rev Nil         ys = ys
rev (Cons x xs) ys = rev xs (Cons x ys)                    .
```

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

be indexed by expressions over the variables $\boldsymbol{i}$. E.g., the append function $(+\!\!+)$ can be attributed the sized type $\forall\alpha.\ \forall ij.\ \texttt{List}_i\ \alpha \to \texttt{List}_j\ \alpha \to \texttt{List}_{(i-1)+j}\ \alpha$.

Soundness of our type-system will guarantee that when $(+\!\!+)$ is applied to a list of size $n+1$ and m respectively, it will yield a list of size $n+m$ (or possibly diverge). As customary in sized types [14], we will also integrate a subtyping relation $\tau \sqsubseteq \rho$ into our system, allowing us to relax size annotations to less precise one. This flexibility will in particular allow us to type conditionals where the branches are attributed different sizes.

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. $\texttt{half} :: \forall i.\texttt{Nat}_{2\cdot i} \to \texttt{Nat}_i$ has to be written as $\texttt{half} :: \forall i.\texttt{Nat}_i \to \texttt{Nat}_{i/2}$. The second restriction prohibits e.g. the type declaration $\texttt{f} :: \forall i.\texttt{Nat}_i \to \texttt{Nat}_i \to \tau$, rather, we have to declare f with a more general type $\forall ij.\texttt{Nat}_i \to \texttt{Nat}_j \to \tau'$. The two restrictions considerably simplify the inference machinery when dealing with pattern matching, and pave the way towards automation. Instead of a complicated unification based mechanism, a matching mechanism suffices.

Abstract Index Language. Unlike in [14], where indices are formed over naturals and addition, we keep the index language abstract. This allows for more flexibility, and ultimately for a better intensionality. Indeed, having the freedom of not adopting a fixed index language is known to lead towards completeness [10].

Polymorphic Recursion over Sizes. Functional programming languages, such as `Haskell` or `OCaml`, allow parametric polymorphism in the form of *let-polymorphism*. Recursive definitions are checked under a monotype, i.e., free type variables cannot be instantiated. Consider e.g. the definition of list reversal from Figure 1. When inferring the polymorphic type for `rev`, the recursive call is assigned the monotype $\texttt{List}\ \alpha \to \texttt{List}\ \alpha \to \texttt{List}\ \alpha$, assuming x and xs, ys have types α and $\texttt{List}\ \alpha$, respectively. On the other hand, if we want to infer the polymorphic sized-type indicated in Figure 1 we will have to overcome let-polymorphism. In the recursive case, the sized-type of the second argument to `rev` will change from

$$
\begin{array}{l}
\texttt{foldr} :: \forall\alpha\beta.\forall jkl.\ (\forall i.\ \alpha \to \texttt{List}_i\ \beta \to \texttt{List}_{i+j}\ \beta) \\
\qquad\qquad\qquad\qquad \to \texttt{List}_k\ \beta \to \texttt{List}_l\ \alpha \to \texttt{List}_{(l-1)\cdot j+k}\ \beta \\
\texttt{foldr}\ f\ b\ \texttt{Nil} \qquad\qquad\ = b \\
\texttt{foldr}\ f\ b\ (\texttt{Cons}\ x\ xs) \ = f\ x\ (\texttt{foldr}\ f\ b\ xs) \\
\\
\texttt{product} :: \forall\alpha\beta.\forall ij.\ \texttt{List}_i\ \alpha \to \texttt{List}_j\ \beta \to \texttt{List}_{(i-1)\cdot(j-1)+1}\ (\alpha \times \beta) \\
\texttt{product}\ ms\ ns = \texttt{foldr}\ (\lambda\ m\ ps.\ \texttt{foldr}\ (\lambda\ n.\ \texttt{Cons}\ (m,n))\ ps\ ns)\ \texttt{Nil}\ ms
\end{array}
$$

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

$\texttt{List}_j\ \alpha$ (the assumed type of ys) to $\texttt{List}_{j+1}\ \alpha$ (the inferred type of $\texttt{Cons}\ x\ ys$). Consequently, in a monomorphic setting the recursive call cannot be typed. To overcome this limitation, 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 traditionally found in sized type systems [14,7].

Higher-ranked Polymorphism over Sizes. In order to remain decidable, classical type inference systems work on polymorphic types in *prenex form* $\forall\boldsymbol{\alpha}.\tau$, where τ 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 following function $\texttt{twice}\ f\ x = f\ (f\ x)$. A strong type-system would allow us to type the expression $\texttt{twice}\ \texttt{Succ}$ with a sized-type $\texttt{Nat}_c \to \texttt{Nat}_{c+2}$. A (specialised) type in prenex form for `twice`, such as

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

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

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 allow

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

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

A Worked Out Example. We conclude this section by giving a nontrivial example, that we will use as a motivating one in the rest of this paper. 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. This, by the way, is an example, that cannot be managed by any of the automatic tools from the literature.[4]

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 taking pattern matching into account.

Let us illustrate this on the recursive equation of `foldr`. Throughout the following, we denote by $s :: \tau$ that the term s has type τ. The type declaration of `foldr` dictates that the left-hand side `foldr` $f\ b$ (`Cons` $x\ xs$) of the recursive equation has type $\texttt{List}_{m\cdot o+n}\ \beta$, under the most general assumptions that $f :: \forall i.\ \alpha \to \texttt{List}_i\ \beta \to \texttt{List}_{i+o}\ \beta$, $b :: \texttt{List}_n$, $x :: \alpha$ and $xs :: \texttt{List}_m\ \alpha$ for arbitrary size-indices o, n, m. Notice that we have taken into account that under these assumptions, the recursion parameter `Cons` $x\ xs$ has size $m + 1$. We now check that the body $f\ x$ (`foldr` $f\ b\ xs$) can be attributed the same sized-type. To this end, we proceed inside out as follows.

1. We instantiate the polymorphic type of `foldr` and derive $\texttt{foldr} :: (\forall i.\ \alpha \to \texttt{List}_i\ \beta \to \texttt{List}_{i+o}\ \beta) \to \texttt{List}_n\ \beta \to \texttt{List}_m\ \alpha \to \texttt{List}_{(m-1)\cdot o+n}\ \beta$;
2. from this and the above assumptions we get $\texttt{foldr}\ f\ b\ xs :: \texttt{List}_{(m-1)\cdot o+n}\ \beta$;
3. instantiating the assumed type of f gives us $f :: \alpha \to \texttt{List}_{((m-1)\cdot o+n)}\ \beta \to \texttt{List}_{m\cdot o+n}\ \beta$;
4. from the last two steps we finally get $f\ x$ (`foldr` $f\ b\ xs$) $:: \texttt{List}_{m\cdot o+n}\ \beta$.

We will not exercise the type checking of the remaining functions. However, we would like to stress two crucial points concerning the type of `foldr`. First of all, we could only suitably type the two occurrences of f in the body of `foldr` since f was given a type polymorphic in the size of its arguments. Secondly, notice that the variable j in the type of `foldr` relates the size of the result of the argument function to the size of the result of `foldr`. This turns out to be a very useful feature in our system, as any expression that can be given a type of the form $\tau \to \texttt{List}_k\ \rho \to \texttt{List}_{k+m}\ \rho$ is applicable to `foldr`, even if m depends on the environment of the call-site. In particular, we will be able to instantiate both λ-abstractions in the definition of `product` to such a type, despite that for the outer abstraction, m depends on the size of the captured variable ns.

3 Applicative Programs and Simple Types

We restrict our attention to a small prototypical, strongly typed functional programming language. For the sake of 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.

[4] Except, our own tool `HoCA`.

Indeed our, implementation (described in Section 7) allows polymorphic function definitions.

Statics. Let $\mathcal{B}$ denote a finite set of base types $\mathtt{B}, \mathtt{C}, \ldots$. The set of *simple types* over $\mathcal{B}$ is inductively generated as follows:

$$\begin{array}{lll} \textbf{(simple types)} & \tau, \rho, \xi ::= \mathtt{B} & \textit{base type} \\ & \mid \tau \times \rho & \textit{pair type} \\ & \mid \tau \to \rho & \textit{function type.} \end{array}$$

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, whereas constructors start with an upper-case letter. Each symbol $s \in \mathcal{X} \cup \mathcal{F} \cup \mathcal{C}$ has a simple type τ, and when we want to insist on that, we write s^τ 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 a natural number $\mathsf{ar}(s) \leq n$, its *arity*. The set of *terms*, *patterns*, *values* and *data values* over functions $\mathtt{f} \in \mathcal{F}$, constructors $\mathtt{C} \in \mathcal{C}$ and variables $x \in \mathcal{X}$ is inductively generated as follows. Here, each term receives implicitly a type, in Church style, and m is strictly smaller than $\mathsf{ar}(\mathtt{f})$. Below, we employ the usual convention that application associates to the left.

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

The presented operators are all standard. The pair destructor $\mathsf{let}\ (x, y) = s\ \mathsf{in}\ t$ 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 *substitution* θ is a finite mapping from variables x^τ to terms s^τ. The substitution mapping $\boldsymbol{x} = x_1, \ldots, x_n$ to $\boldsymbol{s} = s_1, \ldots, s_n$, respectively, is indicated with $\{s_1, \ldots, s_n / x_1, \ldots, x_n\}$ or $\{\boldsymbol{s}/\boldsymbol{x}\}$ for short. The variables $\boldsymbol{x}$ are called the *domain* of θ. We denote by $s\theta$ the application of substitution θ to s, where let-bound variables are renamed to avoid variable capture.

A *program* P over functions $\mathcal{F}$ and constructors $\mathcal{C}$ defines each function $\mathtt{f} \in \mathcal{F}$ through a finite set of *equations* $l^\tau = r^\tau$, where l is of the form $\mathtt{f}\ p_1\ \cdots p_{\mathsf{ar}(f)}$.

We put the usual restriction on equations that each variable occurs at most once in l, 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 $\mathtt{f}$ 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 excluded 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 λ-abstractions from our language. In our setting, abstractions would only complicate the presentation without improving on expressivity. They can always be lifted to the top-level. Similar, 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:

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

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 $\Box^\tau$ in E by s^τ. 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:

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

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

Notice that reduction simply gets stuck if pattern matching in the definition of $\mathtt{f}$ 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. This poses no problem. Programs are *non-ambiguous*. Not only are the results of a computation independent from the reduction order, but also reduction lengths coincide.

Proposition 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 $\mathtt{main}^{\mathtt{B}_1\to\cdots\to\mathtt{B}_k\to\mathtt{B}_n}$, 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 $\mathrm{rc}_{\mathsf{P}} : \mathbb{N} \times \cdots \times \mathbb{N} \to \mathbb{N}^{\infty}$ by

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

We emphasise that the runtime-complexity function defines a cost model that is invariant to traditional models of computation, e.g. Turing machines [11,5].

4 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 3. We have tried to keep the presentation of the relatively involved underlying concepts as simple as possible.

4.1 Indices.

Let $\mathcal{G}$ denote a set of first-order function symbols, the *index symbols*. Any symbol $f \in \mathcal{G}$ is associated with a natural number $\mathsf{ar}(f)$, its *arity*. The set of *index terms* is generated over a countable infinite set of *index variables* $i \in \mathcal{V}$ and index symbols $f \in \mathcal{G}$.

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

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 ϑ 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 $f \in \mathcal{G}$ to a (total) and *weakly monotonic* function $[\![f]\!]_{\mathcal{J}} : \mathbb{N}^{\mathsf{ar}(f)} \to \mathbb{N}$. We suppose that $\mathcal{G}$ always contains three symbols, viz, 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 thus subsumes the one of Hughes et. al. [14], 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 $[\![f(a_1, \ldots, a_k)]\!]^{\alpha}_{\mathcal{J}} := [\![f]\!]_{\mathcal{J}}([\![a_1]\!]^{\alpha}_{\mathcal{J}}, \ldots, [\![a_k]\!]^{\alpha}_{\mathcal{J}})$. We define $a \leq_{\mathcal{J}} b$ if $[\![a]\!]^{\alpha}_{\mathcal{J}} \leq [\![b]\!]^{\alpha}_{\mathcal{J}}$ holds for all assignments α.

The following lemma collects some useful properties of the relation $\leq_{\mathcal{J}}$.

Lemma 2.

1. *The relation $\leq_{\mathcal{J}}$ is reflexive and transitive.*
2. *The relation $\leq_{\mathcal{J}}$ is closed under substitutions, i.e., $a \leq_{\mathcal{J}} b$ implies $a\vartheta \leq_{\mathcal{J}} b\vartheta$.*
3. *If $a \leq_{\mathcal{J}} b$ then $c\{a/i\} \leq_{\mathcal{J}} c\{b/i\}$ for each index term c.*
4. *If $a \leq_{\mathcal{J}} b$ then $a\{0/i\} \leq_{\mathcal{J}} b$.*
5. *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.*

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

$$\frac{\sigma_2 \sqsubseteq_{\mathcal{J}} \sigma_1 \quad \tau_1 \sqsubseteq_{\mathcal{J}} \tau_2}{\sigma_1 \to \tau_1 \sqsubseteq_{\mathcal{J}} \sigma_2 \to \tau_2}\ (\sqsubseteq_{\to}) \qquad \frac{\sigma_2 \geqslant \tau_2 \quad \tau_1 \sqsubseteq_{\mathcal{J}} \tau_2 \quad \boldsymbol{i} \notin \mathsf{FVar}(\sigma_2)}{\forall \boldsymbol{i}.\tau_1 \sqsubseteq_{\mathcal{J}} \sigma_2}\ (\sqsubseteq_{\forall})$$

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

4.2 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 set of *(sized) types* τ and *schemas* is generated from base types $\mathtt{B} \in \mathcal{B}$ and index terms a as follows:

$$\textbf{(types)} \quad \tau, \rho ::= \mathtt{B}_a \mid \tau \times \rho \mid \sigma \to \tau\ , \quad \textbf{(schemas)} \quad \sigma ::= \mathtt{B}_a \mid \forall \boldsymbol{i}.\ \sigma \to \tau\ ,$$

where $\forall \boldsymbol{i}.\ \sigma$ stands for $\forall i_1 \ldots i_n.\ \sigma$ (and $n \geq 0$). Types $\mathtt{B}_a$ are called *indexed base types*. We keep the convention that the arrow binds stronger than quantification. Thus, in $\forall \boldsymbol{i}.\ \sigma \to \tau$ the variables $\boldsymbol{i}$ are bound in σ and τ. We will sometimes write a type τ as $\forall \epsilon.\tau$, for ϵ the empty sequence. similar fashion, we treat pairs. This way, every type and schema can be given in the form $\forall \boldsymbol{i}.\tau$. The *skeleton* of a type τ (schema σ, respectively) 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 σ is denoted by $\mathsf{FVar}(\sigma)$. We consider types equal up to α-equivalence. Index substitutions are extended to types and schemas in the obvious way, using α-conversion to avoid variable capture.

We denote by $\sigma \geqslant \tau$ that the type τ is obtained by *instantiating* the variables quantified in σ with arbitrary index terms, i.e., $\sigma = \forall \boldsymbol{i}.\rho$ and $\tau = \rho\{\boldsymbol{a}/\boldsymbol{i}\}$ holds for some type ρ and index terms $\boldsymbol{a}$. Notice that instantiation acts precisely on the variables quantified at the outermost level of σ. We also stress that by our convention $\tau = \forall \epsilon.\tau$, we have $\tau \geqslant \tau$ for each type τ.

The subtyping relation $\sqsubseteq_{\mathcal{J}}$ is given in Figure 3. 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 2.

Lemma 3.

1. *The subtyping relation is reflexive and transitive.*
2. *The subtyping relation is closed under index substitutions, i.e.,* $\tau_1 \sqsubseteq_{\mathcal{J}} \tau_2$ *implies* $\tau_1\vartheta \sqsubseteq_{\mathcal{J}} \tau_2\vartheta$.
3. *If* $a \leq_{\mathcal{J}} b$ *then* $\sigma\{a/i\} \sqsubseteq_{\mathcal{J}} \sigma\{b/i\}$ *for all index variables* $i \notin \mathsf{FVar}^{-}(\sigma)$.

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

Definition 4 (Canonical Sized Type, Sized Type Declaration). *A type* τ *is* canonical *if either (i)* $\tau = \mathtt{B}_a$ *is an indexed base type, (ii)* $\tau = \tau_1 \times \tau_2$ *for*

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

$$\frac{\Gamma \vdash^{\mathcal{J}} s : (\forall \boldsymbol{i}.\rho_1) \to \tau \quad \Gamma \vdash^{\mathcal{J}} t : \rho_2 \quad \boldsymbol{i} \notin \mathsf{FVar}(\Gamma\restriction_{\mathsf{FVar}(t)}) \quad \rho_2 \sqsubseteq_{\mathcal{J}} \rho_1}{\Gamma \vdash^{\mathcal{J}} s\ t : \tau}\ (\textsc{App})$$

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

Fig. 4. Syntax directed typing rules.

two canonical types τ_1, τ_2, or (iii) $\tau = \sigma_1 \to \tau_1$ for a canonical schema σ_1 and canonical type τ_1 with $\mathsf{FVar}^+(\sigma_1) \cap \mathsf{FVar}^-(\tau_1) = \varnothing$, where furthermore, if $\sigma_1 = \mathtt{B}_a$ then a is an index variable. A schema $\sigma = \forall \boldsymbol{i}.\tau$ is canonical if τ is canonical and $\mathsf{FVar}^-(\tau) \subseteq \{\boldsymbol{i}\}$. To each function symbol $s^\tau \in \mathcal{F} \cup \mathcal{C}$, we associate a closed *and* canonical *schema σ with skeleton τ. We write $s :: \sigma$ and call $s :: \sigma$ 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. *Let σ be canonical type or schema and suppose that $i \notin \mathsf{FVar}^-(\sigma)$. Then $\tau\{a/i\}$ is again canonical.*

In Figure 4 we depict the typing rules of our sized type system. A *(typing) context* Γ is a mapping from variables x^τ to schemas or types σ with skeleton τ. We denote the context Γ that maps variables x_i to σ_i $(1 \leq i \leq n)$ by $x_1 : \sigma_1, \ldots, x_n : \sigma_n$. 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 Γ to a set of variables $X \subseteq \mathcal{X}$. The typing statement $\Gamma \vdash^{\mathcal{J}} s : \tau$ states that under the typing contexts Γ, the term s has type τ, when indices are interpreted with respect to $\mathcal{J}$. The typing rules from Figure 4 are fairly standard. Symbols $s \in \mathcal{F} \cup \mathcal{C} \cup \mathcal{X}$ are given instance types of their associated schemas. This way we achieve the desired degree of polymorphism outlined in Section 2. Subtyping and generalisation is confined to function application, see rule (App). Here, the type ρ_2 of the argument term t is weakened to ρ_1, the side-conditions put on index variables $\boldsymbol{i}$ allow then a generalisation of ρ_1 to $\forall \boldsymbol{i}.\rho_1$, the type expected by the function s. This way, the complete system becomes syntax directed.

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

$$\frac{\mathtt{f} :: \forall \boldsymbol{i}.\tau}{\varnothing \vdash_{\mathsf{FP}} \mathtt{f} : \tau}\ (\textsc{FpFun}) \qquad \frac{\Gamma \vdash_{\mathsf{FP}} t : \sigma \to \tau}{\Gamma \uplus \{x : \sigma\} \vdash_{\mathsf{FP}} t\ x : \tau}\ (\textsc{FpAppVar})$$

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

Fig. 5. Rules for computing the footprint of a linear term.

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 5. The footprint relation makes essential use of canonicity of sized-type declaration and the shape of patterns. The following tells us that footprints guarantees canonicity of the employed types:

Lemma 6. *If $x_1 : \sigma_1, \ldots, x_n : \sigma_n \vdash_{\mathsf{FP}} s : \tau$ then all σ_i and τ are canonical.*

The footprint relation can be understood as a function that, given a left-hand side $\mathtt{f}\ p_1 \cdots p_k$, results in a typing context Γ and type τ. 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 α-conversion. It is thus justified to define $\mathsf{footprint}(\mathtt{f}\ p_1 \cdots p_k) := (\Gamma, \tau)$ for some (particular) context Γ and type τ that satisfies $\Gamma \vdash_{\mathsf{FP}} \mathtt{f}\ p_1 \cdots p_k : \tau$.

Definition 7. *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*

$$\Gamma \vdash_{\mathsf{FP}} l : \tau \implies \Gamma \vdash^{\mathcal{J}}_{\mathsf{G}} r : \rho \text{ for some type } \rho \text{ with } \rho \sqsubseteq_{\mathcal{J}} \tau,$$

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

4.3 Subject Reduction

As a first step towards subject reduction, we introduce a generalisation of the syntax directed typing rules from Figure 4 by integrating a subtyping rule, see Figure 6. The following is obvious from the definition.

Lemma 8. *If $\Gamma \vdash^{\mathcal{J}} s : \tau$ then also $\Gamma \vdash^{\mathcal{J}}_{\mathsf{G}} s : \tau$ is derivable.*

Proof. The proof follows by induction on the derivation of $\Gamma \vdash^{\mathcal{J}} s : \tau$. The only non-trivial case is when this derivation ends in an application of (App). This case is then handled with rules (AppG) and (SubTypeG) applied on the IHs. □

The following lemma clarifies that the footprint correctly accounts for pattern matching. Suppose $\Gamma \vdash_{\mathsf{FP}} s : \rho$. Then if s matches t of type τ, then this type is a subtype, and instance, of ρ. Moreover, the context Γ suitably accounts for the matched variables.

$$\frac{\sigma \geqslant \tau}{\Gamma, x:\sigma \vdash^{\mathcal{J}}_{\mathsf{G}} x : \tau}\ (\textsc{VarG}) \qquad \frac{s \in \mathcal{F} \cup \mathcal{C} \quad s :: \sigma \quad \sigma \geqslant \tau}{\Gamma \vdash^{\mathcal{J}}_{\mathsf{G}} s : \tau}\ (\textsc{FunG})$$

$$\frac{\Gamma \vdash^{\mathcal{J}}_{\mathsf{G}} s : (\forall \boldsymbol{i}.\rho) \to \tau \quad \Gamma \vdash^{\mathcal{J}}_{\mathsf{G}} t : \rho \quad \boldsymbol{i} \notin \mathsf{FVar}(\Gamma\restriction_{\mathsf{FVar}(t)})}{\Gamma \vdash^{\mathcal{J}}_{\mathsf{G}} s\ t : \tau}\ (\textsc{AppG})$$

$$\frac{\Gamma \vdash^{\mathcal{J}}_{\mathsf{G}} s : \tau_1 \times \tau_2 \quad \Gamma, x:\tau_1, y:\tau_2 \vdash^{\mathcal{J}}_{\mathsf{G}} t : \tau}{\Gamma \vdash^{\mathcal{J}}_{\mathsf{G}} \mathsf{let}\ (x,y) = s\ \mathsf{in}\ t : \tau}\ (\textsc{LPairG})$$

$$\frac{\Gamma \vdash^{\mathcal{J}}_{\mathsf{G}} s_1 : \tau_1 \quad \Gamma \vdash^{\mathcal{J}}_{\mathsf{G}} s_2 : \tau_2}{\Gamma \vdash^{\mathcal{J}}_{\mathsf{G}} (s_1, s_2) : \tau_1 \times \tau_2}\ (\textsc{PairG}) \qquad \frac{\Gamma \vdash^{\mathcal{J}}_{\mathsf{G}} s : \rho \quad \rho \sqsubseteq_{\mathcal{J}} \tau}{\Gamma \vdash^{\mathcal{J}}_{\mathsf{G}} s : \tau}\ (\textsc{SubTypeG})$$

Fig. 6. Non-syntax directed typing rules.

Lemma 9 (Footprint Lemma). *Let $s = \mathtt{f}\ p_1\ \cdots\ p_n$ be a linear term with variables $x_1, \ldots, x_m$, and let θ be a substitution with domain $x_1, \ldots, x_m$. If $\vdash^{\mathcal{J}} s\theta : \tau$ then there exist a context $\Gamma = x_1 : \sigma_1, \ldots, x_m : \sigma_m$ and a type ρ such that $\Gamma \vdash_{\mathsf{FP}} s : \rho$ holds. Moreover, for some index substitution ϑ we have $\rho\vartheta \sqsubseteq_{\mathcal{J}} \tau$ and $\vdash^{\mathcal{J}} x_n\theta : \tau_n\vartheta$, where $\sigma_n = \forall \boldsymbol{i}.\tau_n$ $(1 \leqslant n \leqslant m)$.*

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

$$\left.\begin{array}{r} x_1 : \tau_1, \ldots, x_m : \tau_m \vdash^{\mathcal{J}}_{\mathsf{G}} s : \tau \\ \vdash^{\mathcal{J}} s_n : \tau_n\ (1 \leq n \leq m) \end{array}\right\} \implies \vdash^{\mathcal{J}} s\{s_1, \ldots, s_m/x_1, \ldots, x_m\} : \tau\ .$$

Indeed, we prove a generalisation.

Lemma 10 (Generalised Substitution Lemma). *Let s be a term with free variables $x_1, \ldots, x_m$, let Γ be a context over $x_1, \ldots, x_m$, and let ϑ be an index substitution. If $\Gamma \vdash^{\mathcal{J}}_{\mathsf{G}} s : \tau$ for some type τ and $\vdash^{\mathcal{J}} x_n\theta : \tau_n\vartheta$ holds for the type τ_n with $\Gamma(x_n) = \forall \boldsymbol{i}.\tau_n$ $(1 \leqslant n \leqslant m)$, then $\vdash^{\mathcal{J}} s\theta : \tau\vartheta$.*

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

Theorem 11 (Subject Reduction). *Suppose P is well-typed under the interpretation $\mathcal{J}$. If $\vdash^{\mathcal{J}} s : \tau$ and $s \to_{\mathsf{P}} t$ then $\vdash^{\mathcal{J}} t : \tau$.*

But what does Subject Reduction tells us, besides guaranteeing that types are preserved along reduction? Actually, a lot: If $\vdash^{\mathcal{J}} s : \mathtt{B}_a$, 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 type schema *additive* if it is of the form $\forall \boldsymbol{i}.\ \mathtt{B}_{i_1} \to \cdots \to \mathtt{B}_{i_k} \to \mathtt{B}_{\mathsf{s}(i_1+\cdots+i_k)}$. We thus obtain:

Corollary 12. *Suppose 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 $\forall \boldsymbol{i}.\mathtt{B}_{i_1} \to \cdots \to \mathtt{B}_{i_k} \to \mathtt{B}_a$. Then for all inputs $d_1, \ldots, d_n$, if* `main` $d_1 \cdots d_k$ *reduces to a data value d, then the size of d is bounded by $s(|d_1|, \ldots, |d_k|)$, where s is the function $s(i_1, \ldots, i_k) = [\![a]\!]^{\alpha}_{\mathcal{J}}$.*

Note that the corollary by itself, does not mean much about the *complexity* of evaluating s. We will return on this in Section 6.

5 Sized Types Inference

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.

5.1 First- and Second-order Constraint Problems

As a first step towards inference, we introduce metavariables to our index language. To this end, 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 $f \in \mathcal{G}$ by

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

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 13 (Second-order Constraint Problem, Model). *A* second-order constraint problem Φ *(*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* υ *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* $(\mathcal{J}, \upsilon)$ is a model of Φ*, in notation* $(\mathcal{J}, \upsilon) \vDash \Phi$*, if (i)* $e\upsilon \leq_{\mathcal{J}} f\upsilon$ *holds for all inequalities* $e \leq f \in \Phi$*; and (ii)* $i \notin \mathsf{Var}(\upsilon(E))$ *for each occurrence constraint* $i \notin_{sol} E$*.*

For a sequence of index variables $\boldsymbol{i} = i_1, \ldots, i_m$ and second-order variables $\boldsymbol{E} = E_1, \ldots, E_n$ we abbreviate with $\boldsymbol{i} \notin_{sol} \boldsymbol{E}$ the set of occurrence constraints $\{i_k \notin_{sol} E_l \mid 1 \leq k \leq m, 1 \leq l \leq n\}$. We say that Φ is *satisfiable* if it has a model $(\mathcal{J}, \upsilon)$. The term $\upsilon(E)$ is called the *solution* of E. We call Φ 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 Φ depends only on the semantic interpretation $\mathcal{J}$ of index functions. It is thus justified that FOCPs Φ contain no occurrence constraints. We then write $\mathcal{J} \vDash \Phi$ if $\mathcal{J}$ models Φ.

SOCPs are very much suited to our inference machinery. In contrast, satisfiability of FOCPs is a re-occurring problem, e.g. in termination and complexity analysis [17,8]. Consequently, a variety of techniques for checking satisfiability of FOCPs have been proposed, notably the approach of Contejean et. al. [9] which is capable of inferring polynomial models. 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 sound and complete 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* $f_E(\boldsymbol{i})$, for a unique *skolem function* f_E, and where the sequence of variables $\boldsymbol{i}$ 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 is related to E in an inequality of the SOCP Φ. Based on these observations, skolemization is formally defined as follows.

Definition 14. *Let Φ be a SOCP.*

1. *For each second-order variable F of Φ, 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 $\mathcal{SV}_F^{\Phi} := \mathcal{SV}_F^{\Phi,\leq} \setminus \{i \mid (i \not\in_{sol} F) \in \Phi\}$.*
2. *For each second-order variable E of Φ, let f_E be a fresh index symbol, such that the arity of f_E is the cardinality of $\mathcal{SV}_E^{\Phi}$. The* skolem substitution *υ_Φ is given by $\upsilon_\Phi(E) := f_E(i_1, \ldots, i_k)$ where $\mathcal{SV}_E^{\Phi} = \{i_1, \ldots, i_k\}$. Finally, we define the* skolemization *of Φ by $\mathsf{skolemize}(\Phi) := \{e\upsilon_\Phi \leq f\upsilon_\Phi \mid e \leq f \in \Phi\}$.*

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

Lemma 15. *Let Φ be a SOCP with model $(\mathcal{J}, \upsilon)$. Then there exists a restricted second-order substitution υ_r such that $(\mathcal{J}, \upsilon_r)$ is a model of Φ and υ_r satisfies $\mathsf{Var}(\upsilon_r(E)) \subseteq \mathcal{SV}_E^{\Phi}$ for each second-order variable E of Φ.*

Proof. The restricted substitution υ_r is obtained from υ 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 Φ, 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 2. As the occurrence constraints are also satisfied under the new model by definition, the lemma follows. □

Theorem 16 (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 $\mathcal{J}$.*

Proof. It suffices to consider completeness. Suppose $(\mathcal{J}, \upsilon) \vDash \Phi$ holds, where wlog. υ satisfies $\mathsf{Var}(\upsilon(E)) \subseteq \mathcal{SV}_E^{\Phi}$ for each second-order variable $E \in \mathsf{SoVar}(\Phi)$. Let us extend the interpretation $\mathcal{J}$ to an interpretation $\hat{\mathcal{J}}$ by defining $[\![f_E]\!]_{\hat{\mathcal{J}}}(i_1, \ldots, i_k) := [\![\upsilon(E)]\!]_{\mathcal{J}}$, where $\mathcal{SV}_E^{\Phi} = \{i_1, \ldots, i_k\}$, for all $E \in \mathsf{SoVar}(\Phi)$. By the assumption on υ, $\hat{\mathcal{J}}$ 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 Φ, and consequently, $\mathcal{J}$ is a model of $\mathsf{skolemize}(\Phi)$. □

$$\frac{}{\{a \leq b\} \vdash_{\mathsf{ST}} \mathtt{B}_a \sqsubseteq \mathtt{B}_b}\ (\sqsubseteq_{\mathtt{B}}\text{-I})$$

$$\frac{\Phi_1 \vdash_{\mathsf{ST}} \tau_1 \sqsubseteq \tau_3 \quad \Phi_2 \vdash_{\mathsf{ST}} \tau_2 \sqsubseteq \tau_4}{\Phi_1 \cup \Phi_2 \vdash_{\mathsf{ST}} \tau_1 \times \tau_2 \sqsubseteq \tau_3 \times \tau_4}\ (\sqsubseteq_{\times}\text{-I})$$

$$\frac{\Phi_1 \vdash_{\mathsf{ST}} \sigma_2 \sqsubseteq \sigma_1 \quad \Phi_2 \vdash_{\mathsf{ST}} \tau_1 \sqsubseteq \tau_2}{\Phi_1 \cup \Phi_2 \vdash_{\mathsf{ST}} \sigma_1 \to \tau_1 \sqsubseteq \sigma_2 \to \tau_2}\ (\sqsubseteq_{\to}\text{-I})$$

$$\frac{\boldsymbol{E}\ \textit{fresh} \quad \Phi \vdash_{\mathsf{ST}} \tau_1 \sqsubseteq \tau_2\{\boldsymbol{E}/\boldsymbol{j}\} \quad \boldsymbol{i} \notin \mathsf{FVar}(\forall \boldsymbol{j}.\tau_2)}{\Phi \cup \{\boldsymbol{i} \notin_{sol} \mathsf{SoVar}(\tau_1) \cup \mathsf{SoVar}(\tau_2)\} \vdash_{\mathsf{ST}} \forall \boldsymbol{i}.\tau_1 \sqsubseteq \forall \boldsymbol{j}.\tau_2}\ (\sqsubseteq_{\forall}\text{-I})$$

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

$$\frac{\Phi_1; \Gamma \vdash_{\mathsf{I}} s : (\forall \boldsymbol{i}.\rho_1) \to \tau \quad \Phi_2; \Gamma \vdash_{\mathsf{I}} t : \rho_2 \quad \boldsymbol{i} \notin \mathsf{FVar}(\Gamma\restriction_{\mathsf{FVar}(t)}) \quad \Phi_3 \vdash_{\mathsf{ST}} \rho_2 \sqsubseteq \rho_1}{\Phi_1 \cup \Phi_2 \cup \Phi_3 \cup \{\boldsymbol{i} \notin_{sol} \mathsf{SoVar}(\rho) \cup \mathsf{SoVar}(\Gamma\restriction_{\mathsf{FVar}(t)})\}; \Gamma \vdash_{\mathsf{I}} s\ t : \tau}\ (\textsc{App}\text{I})$$

$$\frac{\Phi_1; \Gamma \vdash_{\mathsf{I}} s : \tau_1 \times \tau_2 \quad \Phi_2; \Gamma, x : \tau_1, y : \tau_2 \vdash_{\mathsf{I}} t : \tau}{\Phi_1 \cup \Phi_2; \Gamma \vdash_{\mathsf{I}} \mathsf{let}\ (x, y) = s\ \mathsf{in}\ t : \tau}\ (\textsc{Let}\text{I})$$

$$\frac{\Phi_1; \Gamma \vdash_{\mathsf{I}} s_1 : \tau_1 \quad \Phi_2; \Gamma \vdash_{\mathsf{I}} s_2 : \tau_2}{\Phi_1 \cup \Phi_2; \Gamma \vdash_{\mathsf{I}} (s_1, s_2) : \tau_1 \times \tau_2}\ (\textsc{Pair}\text{I})$$

Fig. 7. Type inference generating a second-order constraint solving problem.

5.2 Constraint Generation

We now define a function $\mathsf{obligations}$ that maps a program P to a SOCP Φ. If $(\mathcal{J}, \upsilon)$ is a model of Φ, 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 or schema σ, we call σ a *template type* or *template schema*, respectively. The function $\mathsf{obligations}$ is itself defined on the two statements $\Phi \vdash_{\mathsf{ST}} \tau \sqsubseteq \rho$ 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 corresponding inference rules are depicted in Figure 7. 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 Γ and term s, yields the SOCP Φ and template type τ, 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 $\mathsf{subtypeOf}(\tau, \rho) := \Phi$, where Φ is the SOCP with $\Phi \vdash_{\mathsf{ST}} \tau \sqsubseteq \rho$.

Definition 17 (Constraint Generation). *For a program* P *we define*

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

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

5.3 Soundness and Relative Completeness

In this section, we will give a series of soundness and completeness results that will lead us to the main result about type inference, namely Corollary **??** below.

Lemma 18. *Subtyping inference is sound and complete, more precise:*

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

Proof. Concerning soundness, we consider a derivation of $\Phi \vdash_{\mathsf{ST}} \tau_1 \sqsubseteq \tau_2$, and fix a second-order substitution υ 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 υ and construct for any two types τ_1 and τ_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 Φ together with an extension ν of υ that satisfies $(\mathcal{J}, \nu) \vDash \Phi$. The construction is then done by induction on the proof of $\tau_1\upsilon \sqsubseteq_{\mathcal{J}} \tau_2\upsilon$. The substitution ν extends υ 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 19. *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 τ then $\Gamma\upsilon \vdash^{\mathcal{J}} s : \tau\upsilon$ holds for every model $(\mathcal{J}, \upsilon)$ of Φ.*
2. ***Completeness**: If $\Gamma \vdash^{\mathcal{J}} s : \tau$ holds for a context Γ and type τ then there exists a template type ρ and a second-order index substitution υ such that $\Phi; \Gamma \vdash_{\mathsf{I}} s : \rho$ is derivable for some SOCP Φ. Moreover, $(\mathcal{J}, \upsilon)$ is a model of Φ.*

Proof. Concerning soundness, we fix a second-order substitution υ and interpretation $\mathcal{J}$ such that and $(\mathcal{J}, \upsilon) \vDash \Phi$ holds, and suppose $\Phi; \Gamma \vdash_{\mathsf{I}} s : \tau$. We prove $\Gamma\upsilon \vdash^{\mathcal{J}} s : \tau\upsilon$ by induction on the derivation $\Phi; \Gamma \vdash_{\mathsf{I}} s : \tau$.

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

Theorem 20 (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 Φ, 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 υ such that $(\mathcal{J}, \upsilon)$ is a model of Φ.*

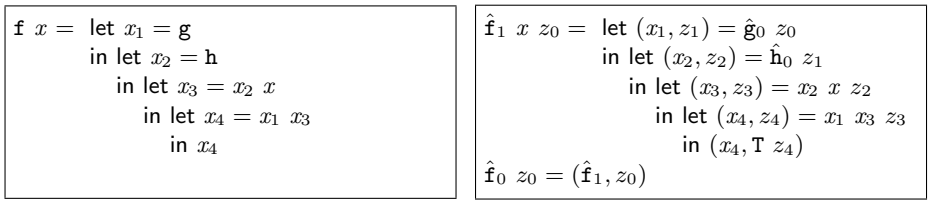

Fig. 8. Equation $\mathtt{f}\ x = \mathtt{g}\ (\mathtt{h}\ x)$ in let-normalform (left) and ticked let-normalform (right).

Proof. We consider soundness first. To this end, let $(\mathcal{J}, \upsilon)$ be a model of Φ. 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 $\Phi_1; \Gamma \vdash_{\mathsf{I}} r : \rho$ and $\Phi_2 \vdash_{\mathsf{ST}} \rho \sqsubseteq \tau$ for some type ρ. Using that the footprint of l does not contain second-order index variables, Lemma 19(1) and Lemma 18(1) then prove $\Gamma \vdash^{\mathcal{J}} s : \rho\upsilon$ and $\rho\upsilon \sqsubseteq_{\mathcal{J}} \tau$, respectively. Conclusively, the rule $l = r$ is well-typed and the claim follows. Completeness is proven dual, using Lemma 19(2) and Lemma 18(2). □

This, in conjunction with Theorem 16 then yields:

Corollary 21. *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* $\mathcal{J}$*.*

6 Ticking Transformation and Time Complexity 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. Technically, the latter is achieved by threading through the computation a counter, the *clock*, which is advanced whenever an equation of P fires. A k-ary function $\mathtt{f} :: \tau_1 \to \cdots \to \tau_k \to \tau$ of P will be modeled in $\hat{\mathsf{P}}$ by a function $\hat{\mathtt{f}}_k :: \langle\tau_1\rangle \to \cdots \to \langle\tau_k\rangle \to \mathtt{C} \to \langle\tau\rangle \times \mathtt{C}$, where $\mathtt{C}$ is the type of the *clock*. Here, $\langle\rho\rangle$ enriches functional types ρ with clocks accordingly. The function $\hat{\mathtt{f}}_k$ behaves in essence like $\mathtt{f}$, but advances the threaded clock suitably. The clock-type $\mathtt{C}$ encodes the running time in unary notation using two constructors $\mathtt{Z}^{\mathtt{C}}$ and $\mathtt{T}^{\mathtt{C}\to\mathtt{C}}$. 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 $\mathtt{f}\ p_1 \cdots p_k = r$ 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 $s \in \mathcal{F} \cup \mathcal{C}$. 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 $\mathtt{f}$ 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 $\hat{\mathtt{f}}_k$. Intermediate functions $\hat{\mathtt{f}}_i$ $(0 \leq i < k)$ deal with partial application. Compare Figure 8 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. The continuation K thus receives two arguments, viz two terms of type $\langle \tau \rangle$ and $\mathtt{C}$, respectively.

Definition 22 (Ticking). *Let* P *be a program over constructors* $\mathcal{C}$ *and functions* $\mathcal{F}$. *Let* $\mathtt{C} \notin \mathcal{B}$ *be a fresh base type.*

1. *To each simple type* τ, *we associate the following* ticked type $\langle \tau \rangle$*:*

$$\langle \mathtt{B} \rangle := \mathtt{B} \quad \langle \tau_1 \times \tau_2 \rangle := \langle \tau_1 \rangle \times \langle \tau_2 \rangle \quad \langle \tau_1 \to \tau_2 \rangle := \langle \tau_1 \rangle \to \mathtt{C} \to \langle \tau_2 \rangle \times \mathtt{C}$$

2. *The set* $\hat{\mathcal{C}}$ *of* ticked constructors *contains a symbol* $\mathtt{Z}^{\mathtt{C}}$, *a symbol* $\mathtt{T}^{\mathtt{C} \to \mathtt{C}}$, *the* tick, *and for each constructor* $\mathtt{C}^{\tau_1 \to \cdots \to \tau_k \to \mathtt{B}}$ *a new constructor* $\hat{\mathtt{C}}^{\langle \tau_1 \rangle \to \cdots \to \langle \tau_k \rangle \to \mathtt{B}}$.
3. *The set* $\hat{\mathcal{F}}$ *of* ticked functions *contains for each* $s^{\tau_1 \to \cdots \to \tau_i \to \tau} \in \mathcal{F} \cup \mathcal{C}$ *and* $0 \leq i \leq \mathsf{ar}(s)$ *a new function* $\hat{s}_i^{\langle \tau_1 \rangle \to \cdots \to \langle \tau_i \rangle \to \mathtt{C} \to \langle \tau \rangle \times \mathtt{C}}$.
4. *For each variable* x^τ, *we assume a dedicated variable* $\hat{x}^{\langle \tau \rangle}$.
5. *We define a translation from (non-ground) values* s^τ *over* $\mathcal{C}$ *to (non-ground) values* $\hat{u}^{\langle \tau \rangle}$ *over* $\hat{\mathcal{C}}$ *as follows.*

$$\hat{u} := \begin{cases} \hat{x} & \text{if } u = x, \\ \hat{s}_{k+1}\ \hat{u}_1\ \cdots\ \hat{u}_k & \text{if } u = s\ u_1\ \cdots\ u_k,\ s \in \mathcal{F} \cup \mathcal{C} \text{ and } k < \mathsf{ar}(s), \\ \hat{\mathtt{C}}\ \hat{u}_1\ \cdots \hat{u}_{\mathsf{ar}(\mathtt{C})} & \text{if } u = \mathtt{C}\ u_1\ \cdots\ u_{\mathsf{ar}(\mathtt{C})}. \end{cases}$$

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

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

where in the last clause, $K_1\ x_1^{\langle \rho \to \tau \rangle}\ z_j^{\mathtt{C}} = \langle s_2^\rho \rangle^{z_j}_{(K_2\ x_1)}$ *and* $K_2\ x_1^{\langle \rho \to \tau \rangle}\ x_2^{\langle \rho \rangle}\ z_k^{\mathtt{C}} = \mathsf{let}\ (x^{\langle \tau \rangle}, z_l) = x_1\ x_2\ z_k\ \mathsf{in}\ K\ x\ z_l$. *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* $\mathtt{f}\ p_1 \cdots p_{\mathsf{ar}(\mathtt{f})} = r$ *in* P*, the* translated equation

$$\hat{\mathtt{f}}_{\mathsf{ar}(\mathtt{f})}\ \hat{p}_1 \cdots \hat{p}_{\mathsf{ar}(\mathtt{f})}\ z = \langle r \rangle^z\ ,$$

   2. *for all* $s \in \mathcal{F} \cup \mathcal{C}$ *and* $0 \leq i < \mathsf{ar}(s)$*, an auxiliary equation*

$$\hat{s}_i\ x_1 \cdots x_i\ z = (\hat{s}_{i+1}\ x_1 \cdots x_i, z)\ ,$$

   3. *for all* $\mathtt{C} \in \mathcal{C}$*, an auxiliary equation*

$$\hat{\mathtt{C}}_{\mathsf{ar}(\mathtt{C})}\ x_1 \cdots x_{\mathsf{ar}(\mathtt{C})}\ z = (\hat{\mathtt{C}}\ x_1 \cdots x_{\mathsf{ar}(\mathtt{C})}, z)\ .$$

   If $s \to_{\hat{\mathsf{P}}} t$*, then we also write* $s \xrightarrow{\mathtt{t}}_{\hat{\mathsf{P}}} t$ *and* $s \xrightarrow{\mathtt{a}}_{\hat{\mathsf{P}}} t$ *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

$$\forall \boldsymbol{ij}.\mathtt{B}_{i_1} \to \cdots \to \mathtt{B}_{i_k} \to \mathtt{C}_j \to \mathtt{B}_a \times \mathtt{C}_b\ ,$$

then the running time of P on inputs of size $\boldsymbol{i}$ 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 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 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.

6.1 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 $s \in \mathcal{F} \cup \mathcal{C}$ and $k < \mathsf{ar}(s)$ inductively as follows.

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

Not every term generated from this grammar is legit. 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 8. 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 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.

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

Lemma 23. *Let e be a term in ticked let-normalform. The following holds:*

1. *$e \xrightarrow{\mathtt{t}}_{\hat{\mathsf{P}}} f$ implies $[e]_1 \to_{\mathsf{P}} [f]_1$ and $[f]_2 = \mathtt{T}\ [e]_2$; and*
2. *$e \xrightarrow{\mathtt{a}}_{\hat{\mathsf{P}}} f$ implies $[e]_1 = [f]_1$ and $[f]_2 = [e]_2$; and*
3. *if $[e]_1$ is reducible with respect to P, then e is reducible with respect to $\hat{\mathsf{P}}$.*

The first two points of Lemma 22 immediately yield that given a $\hat{\mathsf{P}}$ reduction, this reduction corresponds to a P reduction. In particular, the lemma translates a reduction

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

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, $[e_\ell]_2 = \mathtt{T}^\ell\ \mathtt{Z}$. In the following, let us abbreviate $\xrightarrow{\mathtt{t}}_{\hat{\mathsf{P}}} \cdot \xrightarrow{\mathtt{a}}{}^*_{\hat{\mathsf{P}}}$ by $\to_{\mathtt{a/t}}$. 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_ℓ, whereas the corresponding term $[e_\ell]_1$ is reducible. Lemma 22(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 1 clarifies that left-to-right is as good as any reduction order. To summarise:

Theorem 24 (Simulation Theorem — Soundness and Completeness). *Let P be a program whose* `main` *function is of arity k.*

1. ***Soundness**: If $\hat{\mathtt{main}}_k\ \hat{d}_1\ \cdots\ \hat{d}_k\ \mathtt{Z} \xrightarrow{\mathtt{t/a}}{}^\ell_{\hat{\mathsf{P}}}\ e$ then $\mathtt{main}\ d_1\ \cdots\ d_k \to^\ell_{\mathsf{P}}\ t$ where moreover, $[e]_1 = t$ and $[e]_2 = \mathtt{T}^\ell\ \mathtt{Z}$.*
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 $\hat{\mathtt{main}}_k\ \hat{d}_1\ \cdots\ \hat{d}_k\ \mathtt{Z} \xrightarrow{\mathtt{t/a}}{}^\ell_{\hat{\mathsf{P}}}\ e$ where moreover, $[e]_1 = t$ and $[e]_2 = \mathtt{T}^\ell\ \mathtt{Z}$.*

6.2 Time Complexity Analysis

As corollary of the Simulation Theorem, essentially through Subject Reduction (Theorem 11), we finally obtain the main result of this work.

Theorem 25. *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 $\hat{\mathtt{main}}_k :: \forall \boldsymbol{ij}.\mathtt{B}_{i_1} \to \cdots \to \mathtt{B}_{i_k} \to \mathtt{C}_j \to \mathtt{B}_a \times \mathtt{C}_b$. The runtime complexity of P is bounded from above by $rc(i_1, \ldots, i_k) := [\![b\{0/j\}]\!]_{\mathcal{J}}$.*

Notice that in the proof of this theorem, we actually use a strengthening of Corollary 12. When a term e in ticked let-normal form is given a type $\mathtt{B}_a \times \mathtt{C}_b$, then b not only accounts for the size of the clock when in normal form, but indeed for $[e]_2$.

7 Prototype and Experimental Results

We have implemented the discussed inference machinery in a small prototype, dubbed `HoSA`.[5] This tool performs a fully automatic sized-type inference on the strongly typed language given in Section 3, extended with polymorphic types. The extension to the polymorphic setting poses no significant problems. Mainly, the subtyping relation and some auxiliary definitions, such as the notion of canonical sized-type, have to be slightly adapted.

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

7.1 Technical Overview on the Prototype

Our tool `HoSA` is implemented in the strongly-typed, functional programming language `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, pretty-printing and Hindley-Milner type-inference. Along with `HoSA`, we have written a small constraint solver, called `GUBS`. `GUBS` is also implemented in `Haskell` and weights in at around 1.000 lines of code.

In the following, we shortly outline the main execution stages of `HoSA`.

Hindley-Milner Inference. 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.

Specialisation of Polymorphic Types. 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 motivating example from Figure 2 on page 5, we have specialised the most general type of `foldr` to the less general type

$$\forall\alpha\beta.(\alpha \to \mathtt{List}\ \beta \to \mathtt{List}\ \beta) \to \mathtt{List}\ \beta \to \mathtt{List}\ \alpha \to \mathtt{List}\ \beta\ .$$

This way, we could give `foldr` more concrete size annotations. Our implementation is performing such a specialisation automatically. Of course, types cannot be specialised arbitrary. 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

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

type that is too general, 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.

Ticking. Optionally, our tool will perform the ticking transformation from Section 6 on the program obtained in the previous step, thus allowing a form of complexity analysis, the main motivating application behind this work.

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, the annotation is performed by: (i) annotating data types that occur in argument position, i.e. to the left of an arrow, with fresh index variables, (ii) annotating data types to the right of an arrow with an index term $f(\boldsymbol{i}, \boldsymbol{j})$, where f is a fresh index symbol, $\boldsymbol{i}$ collects all the index variables from step (i) and where $\boldsymbol{j}$ is a sequence of fresh variables of fixed length $k \in \mathbb{N}$, the *extra variables*; (iii) annotating functional types in argument position recursively this way, and closing then over all index variables in negative position. Note that the extra variables introduced in step (ii) are necessary to deal with calls to higher-order combinators that receive a functional argument whose sized-type depends itself on the environment of the call-site, see e.g. again the sized-type associated to `foldr` in Figure 2. Here, the program would not be typable without the extra variable j, that occurs free in the type of the first argument. For all of the examples that we considered taking $k = 1$, i.e. adding a single extra variable in step (ii) above, is sufficient. It would be desirable to statically determine the number k of extra variables automatically, for each individual type. This, however seems to require some form of data flow analysis and is beyond the scope of this work.

Constraint Generation. `HoSA` performs type checking as discussed in the previous section based on the annotated types assigned in the previous step. This stage will result in a SOCP, which is then translated to a FOCP by skolemisation.

Constraint Solving. In the final stage, `HoSA` relies on the tool `GUBS` to find a suitable model for the generated constraints. `GUBS` will synthesise a weakly-monotone, polynomial interpretation. Constraint solving is handled in a modular fashion by `GUBS`, via a suitable notion of SCC analysis. `GUBS` searches for a model for the individual SCCs in a bottom-up fashion, propagating the so obtained interpretation functions upwards. To find a model for a single SCC, `GUBS` implements the approach described by Contejean et. al. [9]. This approach has the advantage that it is capable of synthesising non-linear interpretation, and is used nowadays by various automated tools [13,16,6]. Each interpretation function is represented as an *abstract polynomial*, i.e., as a sum of monomials with undetermined coefficients of fixed degree. The inequalities from the input system give rise to a set of *diophantine constraints* over the coefficients variables. Such diophantine constraints can be solved by SMT solvers, in our case, `Z3` [18] and `MiniSMT` [22]. From a solution, the abstract polynomials are then turned into concrete ones, by instantiating the undertermined coefficient variables accordingly. `GUBS` also relies on the incremental features of `Z3` to minimize inferred polynomials.

```
rêv₀ :: C → (List 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 Nil ys z = (ys, T z)
rêv₂ (Cons x xs) ys z₀ = let (x₁, z₁) = rêv₀ z₀
                         in let (x₂, z₂) = x₁ xs z₁
                            in let (x₃, z₃) = Cons̀₀ z₂
                               in let (x₄, z₄) = x₃ x z₃
                                  in let (x₅, z₅) = x₄ ys z₄
                                     in let (x₆, z₆) = x₂ x₅ z₅ in (x₆, T z₆)
```

Fig. 9. Ticked reverse function.

GUBS always tries to find linear interpretations. Only if the resulting diophantine constraints cannot be solved, the procedure is iterated by increasing the degree of abstract polynomials.

7.2 Experimental Evaluation

We will now look at how HoSA deals with the examples mentioned in this paper. All mentioned execution times were measured on an Intel® Core™ i7-4600U with 2.10GHz and 12Gb of RAM.

Tail-Recursive List Reversal. Reconsider the tail-recursive version of list reversal, presented in Figure 1 from page 4. This is an example that could not be handled by the original sized-type system introduced by Hughes. et. al. [14]. In Figure 9 we show the corresponding ticked program. For brevity, the auxiliary definitions derived from the list constructors have been omitted. Our tool infers

$$\texttt{rêv}_2 :: \forall ijk.\texttt{List}_i\ \alpha \to \texttt{List}_j\ \alpha \to \texttt{C}_k \to \texttt{List}_{i+j}\ \alpha \times \texttt{C}_{i+k}\ ,$$

in 0.03 seconds. Thus, we are able to derive the precise runtime of `rev` (size of first argument). Similar, the derived bound for the size of the returned list is almost optimal. The optimal size bound could not be derived, as at the time of writing, our constraint solver GUBS does not support negative coefficients.

Product. Our tool infers the sized-type

$$\forall ijk.\texttt{List}_i\ \alpha \to \texttt{List}_j\ \beta \to \texttt{C}_k \to (\texttt{List}_{1+28\cdot i+119\cdot i\cdot j}\ \alpha \times \beta, \texttt{C}_{1+15\cdot i+2\cdot i\cdot j+k})\ ,$$

in 0.27 seconds for the ticked version of the function `product` from Figure 2. The inferred runtime complexity is only asymptotically precise, partly due to the lack of support for negative coefficients, but also, since GUBS implements only rudimentary techniques to minimise the solutions given by the underlying SMT solver. Imprecisions, however small, then multiply along nested SCCs. By disabling separate SCC analysis in GUBS, HoSA derives the more precise sized-type

$$\forall ijk.\texttt{List}_i\ \alpha \to \texttt{List}_j\ \beta \to \texttt{C}_k \to (\texttt{List}_{16+2\cdot i+i\cdot j}\ \alpha \times \beta, \texttt{C}_{1+5\cdot i+2\cdot i\cdot j+k})\ ,$$

however, computation of this type takes then 2.89 seconds. This precision comes at the expense of modularity in constraint solving, and thus at the expense of computation time.

Insertion Sort. Insertion sort, parameterised by a comparison function, can be defined by

$$\texttt{sort} :: (\alpha \to \alpha \to \texttt{Bool}) \to \texttt{List}\ \alpha \to \texttt{List}\ \alpha$$
$$\texttt{sort}\ \mathit{cmp} = \texttt{foldr}\ (\texttt{insert}\ \mathit{cmp})\ \texttt{Nil}$$

where `insert` is defined as expected. Sorting a list of natural numbers is then expressed by $\texttt{sortNat} = \texttt{sort}\ \texttt{leqNat}$. Despite that a suitable sized-type could be derived in our type system, `HoSA` is currently not capable of doing so. `HoSA` will refine the type variable α in the polymorphic type of `sort` to `Nat`, similar, the type of `insert` will be refined.[6] At some point, inference needs to type the result of inserting an element x into a list ys, under the assumption that x has some type $\texttt{Nat}_n$ and ys has some type $\texttt{List}_l\ \texttt{Nat}_m$. The most precise type we can give to this value in our system is $\texttt{List}_{l+1}\ \texttt{Nat}_{\max(m,n)}$. However, `GUBS` is not supporting the max-operator, the most precise type that can be inferred is $\texttt{List}_{l+1}\ \texttt{Nat}_{m+n}$. But under this abstraction, and since insert is defined by recursion, the size of the list would grow too much for `insert` to be feasible.

8 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. To some extent, 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 [4]. On the other hand, constraint solving, which is inherently not modular, remains in the background.

The main topic for future work is certainly the empowerment of our constraint solving which, as explained in Section 7.2 is currently the main bottleneck. This is however out of scope of this paper. Here, we are mainly concerned with the front-end, rather than on the back-end, of our verification machinery.

References

1. Albert, E., Genaim, S., Masud, A.N.: On the Inference of Resource Usage Upper and Lower Bounds. TOCL 14(3), 22(1–35) (2013)

[6] This step is necessary in order to capture the runtime of `leqNat` on two elements of the given list when typing `insert`.

2. Aspinall, D., Beringer, L., Hofmann, M., Loidl, H.W., Momigliano, A.: A Program Logic for Resources. TCS 389(3), 411–445 (2007)
3. Avanzini, M., Dal Lago, U.: Complexity Analysis by Polymorphic Sized Type Inference and Constraint Solving, Extended Version. Tech. rep., Universities of Bologna and Innsbruck (2016), available at `http://cl-informatik.uibk.ac.at/users/zini/CAPSTICS.pdf`.
4. Avanzini, M., Dal Lago, U., Moser, G.: Analysing the Complexity of Functional Programs: Higher-Order Meets First-Order. In: Proc. of 20th ICFP. pp. 152–164. ACM (2015)
5. Avanzini, M., Moser, G.: Closing the Gap Between Runtime Complexity and Polytime Computability. In: Proc. of 21st RTA. LIPIcs, vol. 6, pp. 33–48. Dagstuhl (2010)
6. Avanzini, M., Moser, G.: Tyrolean Complexity Tool: Features and Usage. In: Proc. of 24th RTA. LIPIcs, vol. 21, pp. 71–80. Dagstuhl (2013)
7. Barthe, G., Grégoire, B., Riba, C.: Type-Based Termination with Sized Products. In: Proc. of 17th CSL. LNCS, vol. 5213, pp. 493–507. Springer (2008)
8. Bonfante, G., Marion, J.Y., Moyen, J.Y.: Quasi-interpretations: A Way to Control Resources. TCS 412(25), 2776–2796 (2011)
9. Contejean, E., Marché, C., Tomás, A.P., Urbain, X.: Mechanically Proving Termination Using Polynomial Interpretations. JAR 34(4), 325–363 (2005)
10. Dal Lago, U., Gaboardi, M.: Linear Dependent Types and Relative Completeness. LMCS 8(4) (2011)
11. Dal Lago, U., Martini, S.: On Constructor Rewrite Systems and the Lambda-Calculus. In: Proc. of 36th ICALP. LNCS, vol. 5556, pp. 163–174. Springer (2009)
12. Danner, N., Licata, D.R., Ramyaa: Denotational Cost Semantics for Functional Languages with Inductive Types. In: Proc. of 20th ICFP. pp. 140–151. ACM (2015)
13. Giesl, J., Schneider-Kamp, P., Thiemann, R.: AProVE 1.2: Automatic Termination Proofs in the Dependency Pair Framework. In: Proc. of 3rd IJCAR. LNAI, vol. 4130, pp. 281–286. Springer (2006)
14. Hughes, J., Pareto, L., Sabry, A.: Proving the Correctness of Reactive Systems Using Sized Types. In: Proc. of 23rd POPL. pp. 410–423. POPL '96, ACM (1996)
15. Jost, S., Hammond, K., Loidl, H.W., M.Hofmann: Static Determination of Quantitative Resource Usage for Higher-order Programs. In: Proc. of 37th POPL. pp. 223–236. ACM (2010)
16. Korp, M., Sternagel, C., Zankl, H., Middeldorp, A.: Tyrolean Termination Tool 2. In: Proc. of 20th RTA. LNCS, vol. 5595, pp. 295–304. Springer (2009)
17. Lankford, D.: On Proving Term Rewriting Systems are Noetherian. Tech. Rep. MTP-3, Louisiana Technical University (1979)
18. Mendonça de Moura, L., Bjørner, N.: Z3: An Efficient SMT Solver. In: Proc. of 14th TACAS. LNCS, vol. 4963, pp. 337–340. Springer (2008)
19. Sinn, M., Zuleger, F., Veith, H.: A Simple and Scalable Static Analysis for Bound Analysis and Amortized Complexity Analysis. In: Proc. of 26th CAV. LNCS, vol. 8559, pp. 745–761 (2014)
20. Vasconcelos, P.: Space Cost Analysis Using Sized Types. Ph.D. thesis, School of Computer Science, University of St Andrews (2008)
21. Wilhelm, R., Engblom, J., Ermedahl, A., Holsti, N., Thesing, S., Whalley, D., Bernat, G., Ferdinand, C., Heckmann, R., Mitra, T., Mueller, F., Puaut, I., Puschner, P., Staschulat, J., Stenstrom, P.: The Worst Case Execution Time Problem - Overview of Methods and Survey of Tools. TECS pp. 1–53 (2008)

22. Zankl, H., Middeldorp, A.: Satisfiability of Non-linear (Ir)rational Arithmetic. In: Logic for Programming, Artificial Intelligence, and Reasoning - 16th International Conference, LPAR-16, Dakar, Senegal, April 25-May 1, 2010, Revised Selected Papers. LNCS, vol. 6355, pp. 481–500. Springer (2010)