Automating Size Type Inference and Complexity Analysis

Martin Avanzini Ugo Dal Lago

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

One successful approach to automatic verification of termination properties of higher-order functional programs is based on *sized types* [9], and has been shown to be quite robust and amenable to automation [4]. 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 [10], 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 [6].

Since the first inception in the seminal paper of Hughes et. al. [9] the literature on sized typed has grown to a considerable extend. Indeed, various significantly more expressive systems have been introduced, with the main aim to improve the expressiveness in the context of termination analysis. For instance, Blanqui [5] introduced a novel sized type system on top of the *calculus of algebraic construction*. Notably, it has been shown that for size indices over the successor algebra, type checking is decidable [5]. The system is thus capable of expressing additive relations between sizes. In the context of termination analysis, where one would like to statically detect that a recursion parameter decreases in size, this is sufficient. In this line of research falls also more recent work of Abel and Pientka [1], where a novel sized-type system for termination analysis on top of F_ω is proposed. Noteworthy, this system has been integrated in the dependently typed language `Agda`.[1]

As we will see, capturing only additive relations between value sizes is not enough for our purpose. On the other hand, even slight extensions to the size index language render current methods for type inference, even type checking, intractable. In this paper, we thus take a fresh at sized-systems, with a particular emphasis on a richer index language and feasible automation on existing constraint solving technology. Our system exhibits many similarities with the archetypal system from [9], which itself is based on a Hindley-Milner style system. Although conceptually simple, our system is substantially more expressive than the traditional one. This is possible mainly due to the addition of one ingredient, viz, the presence of *arbitrary rank index polymorphism*. That is, functions that take functions as their argument can be polymorphic in their size annotation. Of course, our sized-type system is proven a sound methodology for *size* analysis. In contrast to existing works, we also introduce a inference machinery that is sound and (relative) complete. Finally, this system

[1] See `http://wiki.portal.chalmers.se/agda`.

system is amenable to time complexity analysis by a ticking monadic transformation. A prototype implementation is available, see below for more details. More specifically, our 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. [5, 1]. In particular, we do not require that recursive calls are made on arguments of smaller size.
- 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. In contrast to existing works, we put only mild assumptions on the index language, and we do not require that the generated inequalities admit a most general solution. Indeed, this enables us to express significantly more complicated size relations between inputs and outputs.
- 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. 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 [7], 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 called `HoSA` which implements type inference and ticking, and then relies on an external tool, dupped `GUBS`, to check the generated constraints for satisfiability. `GUBS` borrows heavily from the advances made over the last decade in the synthesis of *polynomial interpretations*, a form of polynomial ranking function, by the rewriting community. It features also some novel aspects, most importantly, a bottom-up SCC analysis for incremental constraint solving. We thus arrive at a fully automated runtime analysis of higher-order functional programs. Noteworthy, we are able to effectively infer polynomial, not necessarily linear, bounds on the runtime of programs. Both tools are open source and available from the first authors homepage [2]. `HoSA` is able to analyse, fully automatically, a series of examples which cannot be handled by most competitor methodologies. Indeed, it is to our best knowledge up until today the only approach that can fully deal with function closures whose complexity depends on the captured environment,compare for instance the very recent work of Hoffmann et. al. [8]. Dealing with such closures is of crucial importance, e.g., when passing partially applied functions to higher-order combinators, a feature pervasively used in functional programming.

For brevity, in this abstract we can only give some intuitions. An extended version with a full formalisation of the system and all the technical details can be found available [2].

2 Our Type System at a Glance

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. [9] and similar ones [4, 10], but we try to keep presentation as self-contained as possible.

[2] See `https://cl-informatik.uibk.ac.at/users/zini/software`.

$$
\begin{aligned}
&\texttt{reverse} :: \forall\alpha.\ \forall i.\texttt{List}_i\ \alpha \to \texttt{List}_i\ \alpha\\
&\texttt{reverse}\ xs = \texttt{rev}\ xs\ \texttt{Nil}\\
&\\
&\texttt{rev} :: \forall\alpha.\ \forall ij.\texttt{List}_i\ \alpha \to \texttt{List}_j\ \alpha \to \texttt{List}_{(i-1)+j}\ \alpha\\
&\texttt{rev}\ \texttt{Nil}\qquad\quad ys = ys\\
&\texttt{rev}\ (\texttt{Cons}\ x\ xs)\ ys = \texttt{rev}\ xs\ (\texttt{Cons}\ x\ ys)\qquad\qquad .
\end{aligned}
$$

Figure 1: Sized-type annotated tail-recursive list reversal function.

Basics. We work with functional programs over a fixed set of inductive datatypes, e.g. $\texttt{Nat}$ for natural numbers and $\texttt{List}\ \alpha$ for lists over elements of type α. Each such datatype is associated with a set of typed *constructors*, below we will use the constructors $0 :: \texttt{Nat}$, $\texttt{Succ} :: \texttt{Nat} \to \texttt{Nat}$ for naturals, and the usual constructors $\texttt{Nil} :: \forall\alpha.\ \texttt{List}\ \alpha$ and $\texttt{Cons} :: \forall\alpha.\ \alpha \to \texttt{List}\ \alpha \to \texttt{List}\ \alpha$ for lists. Sized types refine each such datatype into a *family* of datatypes indexed by *strictly positive* natural numbers, their *size*. E.g., to $\texttt{Nat}$ and $\texttt{List}\ \alpha$ we associate the families $\texttt{Nat}_1, \texttt{Nat}_2, \texttt{Nat}_3, \ldots$ and $\texttt{List}_1\ \alpha, \texttt{List}_2\ \alpha, \texttt{List}_3\ \alpha, \ldots$, respectively. An indexed datatype such as $\texttt{List}_n\ \texttt{Nat}_m$ 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 $\texttt{List}_n\ \texttt{Nat}_m$ is formed from (at most) $n-1$ constructors $\texttt{Cons}$ and a constructor $\texttt{Nil}$. Each element of this list, in turn, consists of at most m constructors.

A function $\texttt{f}$ will then be given a polymorphic type $\forall\vec{\alpha}.\ \forall\vec{i}.\ \tau \to \rho$ with size-variables $\vec{i}$ ranging over sizes. Datatypes occurring in the types τ and ρ will be indexed by expressions over the variables $\vec{i}$. E.g., the append function $(+\!\!+)$ can be attributed the sized type $\forall\alpha.\ \forall ij.\ \texttt{List}_i\ \alpha \to \texttt{List}_j\ \alpha \to \texttt{List}_{(i-1)+j}\ \alpha$. 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 [9], 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 $\texttt{f}$ 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 [9], 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 [6].

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 $\texttt{rev}$, 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 $\texttt{rev}$ will change from $\texttt{List}_j\ \alpha$ (the assumed type of ys) to $\texttt{List}_{j+1}\ \alpha$ (the

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

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

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 [9, 4].

Higher-ranked Polymorphism over Sizes. In order to remain decidable, classical type inference systems work on polymorphic types in *prenex form* $\forall\vec{\alpha}.\tau$, where 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 sizes equal each other. 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.

The Ticking Transformation. Our type system as we have described it until now only reflect extensional properties of programs, namely how the size of the output relates to the size of the input. We now introduce the *ticking transformation*, which 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 `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.

3 A Worked Out Example

In this section we give 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.[3]

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 `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 `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.

4 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 [3]. On the other hand, constraint solving, which is inherently not modular, remains in the background.

References

[1] A. Abel and B. Pientka. Well-founded recursion with copatterns and sized types. *JFP*, 26:e2, 2016.

[2] M. Avanzini and U. Dal Lago. Complexity Analysis by Polymorphic Sized Type Inference and Constraint Solving, Extended Version. Technical report, Universities of Bologna and Innsbruck, 2016. Available at `http://cl-informatik.uibk.ac.at/users/zini/CAPSTICS.pdf`.

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

[3] M. Avanzini, U. Dal Lago, and G. Moser. Analysing the Complexity of Functional Programs: Higher-Order Meets First-Order. In *Proc. of 20th ICFP*, pages 152–164. ACM, 2015.

[4] G. Barthe, B. Grégoire, and C. Riba. Type-Based Termination with Sized Products. In *Proc. of 17th CSL*, volume 5213 of *LNCS*, pages 493–507. Springer, 2008.

[5] F. Blanqui. Decidability of Type-Checking in the Calculus of Algebraic Constructions with Size Annotations. In *Proc. of 14th CSL*, volume 3634 of *LNCS*, pages 135–150. Springer, 2005.

[6] U. Dal Lago and M. Gaboardi. Linear Dependent Types and Relative Completeness. *LMCS*, 8(4), 2011.

[7] N. Danner, D. R. Licata, and Ramyaa. Denotational Cost Semantics for Functional Languages with Inductive Types. In *Proc. of 20th ICFP*, pages 140–151. ACM, 2015.

[8] J. Hoffmann, A. Das, and S.-C. Weng. Towards Automatic Resource Bound Analysis for OCaml. In *Proc. of 44th POPL*, pages 359–373. ACM, 2017.

[9] J. Hughes, L. Pareto, and A. Sabry. Proving the Correctness of Reactive Systems Using Sized Types. In *Proc. of 23rd POPL*, POPL '96, pages 410–423. ACM, 1996.

[10] P. Vasconcelos. *Space Cost Analysis Using Sized Types*. PhD thesis, School of Computer Science, University of St Andrews, 2008.