Size Calculus for a Higher-Order Functional Language

Attila Gobi[1⋆], Olha Shkaravska[2], and Marko van Eekelen[2,3⋆⋆]

[1] Faculty of Informatics, Eötvös Loránd University Budapest
[2] Institute for Computing and Information Sciences, Radboud University Nijmegen
[3] Open University of the Netherlands, Heerlen

Abstract. The authors present a lambda-calculus that formalizes the relations between the sizes of arguments and the sizes of the corresponding results of functions in a higher-order polymorphic functional language. On top of usual constructions two operators for finite maps: List, that defines (higher-order) finite maps, and Shift are considered. Intuitively, size expressions are abstract interpretations of programs in the natural arithmetic.
To prove normalization and diamond (modulo integer axiomatics) property of the calculus we show that the calculus can be expressed in System F.

1 Introduction

Size information about program expressions is important, especially in case of resource analysis, where sized types are commonly used to build algorithms to predict resource consumption and termination. A good size analysis is especially important as it determines the precision of the prediction.

Correct sizes are however not easy to obtain. Introducing sized types may lead to the need of polymorphism in the size variables. Consider the following function written in a Haskell-like language.

```
f :: ([a] -> b) -> (b,b)
f g = (g [], g [1])
```

By extending the original type declaration of `f` with size variables, the result would be $(\mathsf{L}_n(a) \to b) \to (b, b)$. Here the type is implicitly quantified at the outer level, resulting a typing error. The reason for this is that two different list types are applied to the function g.

A common solution is to use subtype polymorphism, where $\mathsf{L}_n(a) \leq \mathsf{L}_n(a)$ iff $n \leq m$. This approach solves the problem, but can lead to significant overestimation of the sizes.

⋆ The first author is supported by the European Union and co-financed by the European Social Fund (grant agreement no. TAMOP 4.2.1./B-09/1/KMR-2010-0003).
⋆⋆ The second and the third author are supported by the Artemis Joint Undertaking in the CHARTER project, grant-nr. 100039.

Another solution is to use parametric polymorphism, leading to higher rank types, like $(\forall n.\mathsf{L}_n(a) \to b) \to (b, b)$. However this type is still not generic enough. For example in the result of the expression `f cons` would be a pair of list of different size, which is impossible, because the type of `f` suggested that the two elements of the pair must have the same size (i.e. the result is (b, b)).

It is still possible to generalize the idea by using dependent types and by introducing a type function to handle the return types, but for a complicated function the type would be even more complicated. We believe that the type of a function and the size dependencies of a function reflect different aspects and mixing them causes unnecessary complexity.

We present a calculus that formalizes the relations between the sizes of arguments and the sizes of the corresponding results of functions in a higher-order polymorphic functional language. Informally, the calculus extends the lambda-calculus with arithmetic operations and two operators for finite maps: List, that defines (higher-order) finite maps, and Shift.

The verification conditions we obtain as the result of the syntax-directed stage of type-checking, are (conditional) equations in the combination of three theories: lambda-calculus, integer ring and finite maps.

This calculus is Turing complete so checking the equality of two size expressions cannot be decidable. When trying to check a size expression which is not normalizable, the constraint solver would not terminate. In Section 5 we discuss a type system for size expression which gives a sufficient condition for normalizability.

The ultimate goal is to continue the series of work on size analysis of first-order strict functional languages – in these works annotation inference is based on polynomial interpolation [7]. The set of normalizable size expressions can be seen as an extension of the type checker defined in that paper as well.

This paper is a revised version of our previous work [3]. A proof of concept implementation has been created, and much of the paper has been rewritten to reflect the implementation.

2 Language Syntax

Consider a higher-order strict functional language extended with size expressions. Its syntax is given on Fig. 1. Every top-level binding is annotated with its type and its size given by a size expression.

The underlying type system is System F and it has two predefined types: `Int` for integers and $\mathsf{L}(\tau)$ for lists. We do not cover the type checking of top-level bindings, but in practice a Hindley–Milner style type inference can be used to infer these types in a previous stage of the size checking.

Our language does not allow lambda abstraction and recursive let-expressions, however recursive functions can be defined as top-level bindings. It is easy to extend the language with recursive let, if we annotate let bindings with size expressions. Similarly, lambda abstraction can be enabled if it has an explicit

$$
\begin{array}{lrl}
\text{Program variables} & & \in\ x, y, z, f, g, h \\
\text{Integer literals} & & \in\ m, n \\
\text{Type variables} & & \in\ \alpha \\
\text{Types} & \tau & ::= \mathtt{Int} \mid \mathsf{L}(\tau) \mid \alpha \mid \tau_1 \to \tau_2 \\
\text{Data constructors} & K & ::= \mathtt{nil} \mid \mathtt{cons} \mid n \\
\text{Expressions} & e & ::= x \mid K \mid e_1\ e_2 \mid \mathtt{let}\ x = e_1\ \mathtt{in}\ e_2 \\
 & & \quad \mid\ \mathtt{if}\ e_1\ \mathtt{then}\ e_2\ \mathtt{else}\ e_3 \\
 & & \quad \mid\ \mathtt{match}\ e_1\ \mathtt{with} \mid \mathtt{nil} \Rightarrow e_2 \\
 & & \qquad\qquad\qquad\quad \mid \mathtt{cons\ hd\ tl} \Rightarrow e_3 \\
\text{Programs} & \mathit{prog} & ::= \epsilon \mid \mathtt{f}\ \mathtt{z}_1 \ldots \mathtt{z}_k :: \tau :: \eta = e;\ \mathit{prog}
\end{array}
$$

Fig. 1. Syntax

type annotation. However these restriction does not affect the expressiveness of the language so they are omitted for simplicity.

There is a number of reserved function names corresponding to integer constants (i.e. 0-ary functions), unary and binary integer operations (unary minus, addition, subtraction, multiplication, division and mod), and the list constructors `nil` and `cons hd tl`.

2.1 Size calculus

Size expressions represent size dependencies of functions. Their grammar is given on Fig. 2. It is a lambda calculus extended with integer arithmetic and combinators to express the size dependencies of lists.

$$
\begin{array}{lrl}
\text{Size variables} & & \in\ s, p \\
\text{Binary operators} & \xi & ::= + \mid\ - \mid * \\
\text{Size expressions} & \eta & ::= \lambda s.\eta \mid \widehat{\lambda}^s_p.\eta \mid \eta_1\eta_2 \mid s \\
 & & \quad \mid\ \mathsf{List} \mid \mathsf{Unsized} \mid \mathsf{Shift} \mid \bot \\
 & & \quad \mid\ \eta_1\, \xi\, \eta_2 \mid n
\end{array}
$$

Fig. 2. Syntax of size expressions

Now a few simple examples are considered to give an idea behind the formalization. Begin with an integer literal, eg. 42. We assume that it does not have size, so the expression $\mathsf{Unsized}$ is assigned to it.

For arguments of functions, abstraction over size expressions are used. The following binding declares a function which maps everything to 42. Its size expression mirrors the fact that the size of the result of the function is $\mathsf{Unsized}$ regardless of the size of the argument.

$$\mathtt{const}\ x :: \alpha \to \mathtt{Int} :: \lambda s_x.\, \mathsf{Unsized} = 42$$

The size of a list is expressed by the combinator List. For instance, the size of the list "[2]" is given by $\mathsf{List}\, 1\ (\lambda i.\, \mathsf{Unsized})$. Here the first argument of List denotes

the length of the list while the second is a lambda abstraction expressing the sizes of the elements of the list. As the only element of this list is 2, which is unsized, one can say that all elements of this list is unsized: $\lambda i.\,\mathsf{Unsized}$.

The λ-bound variable i corresponds to the position of the element in a list. For instance, according to the expression $\mathsf{List}\,n\,(\lambda i.\,e\,i)$, the expression $e\,(n-1)$ represents the size of the head, the expression $e\,(n-2)$ represents the size of the element next to the head, while the expression $e\ 0$ represents the length of the tail element[4]. It means the expression e can be seen as a finite map defined on $0,\ldots,n-1$.

For size expressions of the form $\mathsf{List}\ \ s\ p$, the abstraction $\widehat{\lambda}^s_p$ is used. This abstraction is the dual of the list constructor as it can be seen on the reduction rules (Fig. 3). During reductions capture-avoiding substitutions are assumed, i.e. alpha-renaming or other mechanism.

$$(\lambda p.e_1)(e_2) \to_\beta (e_1[p := e_2])$$
$$(\widehat{\lambda}^s_p.e)(\mathsf{List}\,e_1\,e_2) \to_\beta (e[s := e_1, p := e_2])$$
$$\mathsf{Shift}\,e_1\,s\,e_2 \to_\beta \lambda i.\begin{cases} e_1 i & \text{if } i < s \\ e_2(i-s) & \text{otherwise} \end{cases}$$

Fig. 3. Reduction rules of the size calculus

In this way it is possible to present the size expression for the result of a function in terms of the size expressions of its arguments. The following example the function addone takes its argument $l : L(\mathtt{Int})$ and returns the list $(1 :: l)$. The size expression of the function tells us the size of the list is incremented by one.

$$\mathsf{addone}\ l :: \mathsf{L}(\mathtt{Int}) \to \mathsf{L}(\mathtt{Int}) :: \widehat{\lambda}^s_p.\,\mathsf{List}\,(s+1)(\lambda i.\,\mathsf{Unsized}) = \mathtt{cons}\ 1\ l$$

The combinator Shift is used to concatenate size functions. The expression $\mathsf{Shift}\,e_1\,s\,e_2$ means a size function of the list obtained by inserting the last s elements of e_1 before e_2. With the help of this function it is easy to express the size expression of the predefined functions.

As a more complicated example, the declaration of the function `concat` is shown. This function takes two list arguments and results the concatenation of

[4] Note that this enumeration of list elements "opposes" the traditional in the functional languages enumeration, where the head element has number 0, etc. The enumeration we use is more convenient in our reasoning and, for instance, simplifies significantly the match-rule defined later.

$$\texttt{nil} : \mathsf{List}\ 0\ (\lambda i.\bot) \qquad m : \mathsf{Unsized} \qquad +, -, \ldots : \lambda x.\lambda y.\ \mathsf{Unsized}$$
$$\texttt{cons} : \lambda s_x.\ \widehat{\lambda}^{s_l}_{p_l}.\ \mathsf{List}\ (s_l + 1)\ (\mathsf{Shift}\ p_l\ s_l\ \lambda y.s_x)$$

Fig. 4. Size expressions of the predefined functions

the lists.

$$\begin{aligned}\texttt{concat x y} &:: \mathsf{L}(\alpha) \to \mathsf{L}(\alpha) \to \mathsf{L}(\alpha)\\ &:: \widehat{\lambda}^{s_x}_{p_x}.\widehat{\lambda}^{s_y}_{p_y}.\,\mathsf{List}\ (s_x + s_y)\ (\mathsf{Shift}\ p_y\ s_y\ p_x)\\ &= \texttt{match}\ x\ \texttt{with} \mid \texttt{nil} \Rightarrow y\\ &\qquad\qquad\qquad\quad \mid \texttt{cons hd tl} \Rightarrow \texttt{cons hd}\ (\texttt{concat tl}\ y)\end{aligned}$$

3 Size analysis

Our calculus can be used to check the size relations of top level bindings. The verification consists of two steps – the first part is a verification condition generation and the second part is the verification of the generated conditions.

Verification of those conditions are not covered in this paper, an we assume an external solver is used. However in our prototype implementation, a solver with a very heuristics is able to solve all of our examples.

3.1 Constraint generation

$$\begin{array}{ll}\text{Constraint set} & \mathcal{C} ::= \epsilon \mid \mathcal{C}, D \rightsquigarrow \eta\\ \text{Condition} & D ::= \epsilon \mid D, \eta = 0 \mid D, \eta >= n\end{array}$$
$$\mathcal{C}_1 \ltimes \mathcal{C}_2 = \{D_1 \cup D_2 \rightsquigarrow \eta_1\eta_2 \mid D_1 \rightsquigarrow \eta_1 \in \mathcal{C}_1,\ D_2 \rightsquigarrow \eta_2 \in \mathcal{C}_2\}$$

Fig. 5. Syntax and operations of the assumption system

First of all, an assumption system is defined in Fig. 5. The assumption system consists of elements of the form $D \rightsquigarrow \eta$, where η is a size expression and D is a set of conditions. A $D \rightsquigarrow \eta$ means that the corresponding expression has size η if all of the conditions in D hold.

Before introducing the size checking rules we need a function to generate size expressions containing fresh variables for a given underlying type. The formal definition of the function can be found in Fig. 6, and informally, the following examples show what does the function do.

$$
\begin{aligned}
\text{fresh}(\mathsf{L}(\texttt{Int});\ \emptyset) &= \mathsf{List}\ \ s_1\ \ (\lambda i.\,\mathsf{Unsized}) \\
\text{fresh}(\mathsf{L}(\alpha);\ \emptyset) &= \mathsf{List}\ \ s_1\ \ (\lambda i.s_2 i) \\
\text{fresh}(\mathsf{L}(\mathsf{L}(\texttt{Int}));\ \emptyset) &= \mathsf{List}\ \ s_1\ \ (\lambda i.\,\mathsf{List}\ \ s_2\ \ (\lambda j.\,\mathsf{Unsized})) \\
\text{fresh}(\mathsf{L}(\mathsf{L}(\alpha));\ \emptyset) &= \mathsf{List}\ \ s_1\ \ (\lambda i.\,\mathsf{List}\ \ s_2\ \ (\lambda j.s_3 i j))
\end{aligned}
$$

The fresh function takes a type and a set of free size variables as arguments and results a size expression corresponding to the given type. All free variables in the resulting size expressions are fresh.

$$
\begin{aligned}
\text{fresh} &:: \tau \times [\eta] \to \eta \\
\text{fresh}(\texttt{Int},\ \gamma) &= \mathsf{Unsized} \\
\text{fresh}(\tau_1 \to \tau_2,\ \gamma) &= \lambda\beta.\,\text{fresh}(\tau_2,\ \gamma, \beta) \\
\text{fresh}(\mathsf{L}(\tau),\ \gamma) &= \mathsf{List}\ \ \beta\ \ (\lambda\beta'.\,\text{fresh}(\tau,\ \gamma, \beta')) \\
\text{fresh}(\alpha,\ (\eta :: \gamma)) &= \text{fresh}(\alpha,\ \gamma)\ \eta \\
\text{fresh}(\alpha,\ [\,]) &= \beta
\end{aligned}
$$

(Where β and β' are fresh size variables.)

Fig. 6. Fresh size expression for a given type

The top level bindings are shown in Fig. 7. The judgement $\Gamma \vdash e$ can be read as "in the type environment Γ the program e is well-sized". It is assumed that the function binding is well-typed in the underlying type system. Then the rule creates fresh size expressions for each argument, saves it in the environment and infers an assumption set. The assumption set is then used to create verification conditions.

$$
\frac{FV(\Gamma) = \emptyset}{\Gamma \vdash \epsilon}\ \textsc{Empty}
$$

$$
\frac{\begin{array}{c} \vdash_{UL} \lambda \mathtt{z}_1 \ldots \mathtt{z}_k.e : \tau \\ \bar{\eta} = \text{fresh}(\bar{\tau}, \emptyset) \qquad \Gamma, \mathtt{f} : \{\emptyset \rightsquigarrow \eta'\},\ \bar{z} : \{\emptyset \rightsquigarrow \bar{\eta}\}\ \vdash e : \mathcal{C} \\ \forall D \rightsquigarrow \eta_c \in \mathcal{C} : D\ \Vdash \eta_c = \eta' \bar{\eta} \\ \Gamma, \mathtt{f} : \{\emptyset \rightsquigarrow \eta'\} \vdash prog \end{array}}{\Gamma \vdash \mathtt{f}\ \mathtt{z}_1 \ldots \mathtt{z}_k :: \tau_1 \to \ldots \to \tau_k \to \tau' :: \eta' = e, prog}\ \textsc{Bind}
$$

Fig. 7. Top level bindings

The generation of the assumption set uses the judgement $\Gamma \vdash e : \mathcal{C}$ which means "in the type environment Γ we assume that expression e has size $\mathcal{C}$". The rules to obtain the assumption set can be seen in Fig 8.

$$\frac{}{\Gamma,\, a : \mathcal{C} \vdash a : \mathcal{C}}\ \textsc{Var} \qquad \frac{}{\Gamma \vdash m : \{\emptyset \rightsquigarrow \mathsf{Unsized}\}}\ \textsc{Int}$$

$$\frac{}{\Gamma \vdash \texttt{nil} : \{\emptyset \rightsquigarrow \mathsf{List}\ 0\ (\lambda i.\bot)\}}\ \textsc{Nil}$$

$$\frac{}{\Gamma \vdash \texttt{cons} : \lambda s_x.\ \widehat{\lambda}^{s_l}_{p_l}.\ \mathsf{List}\ (s_l + 1)\ (\mathsf{Shift}\ p_l\ s_l\ \lambda y.s_x)}\ \textsc{Cons}$$

$$\frac{\Gamma \vdash e_2 : \mathcal{C}_2 \quad \Gamma \vdash e_3 : \mathcal{C}_3}{\texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 : \mathcal{C}_1 \cup \mathcal{C}_2}\ \textsc{If}$$

$$\frac{\Gamma \vdash e_1 : \mathcal{C}_1 \quad \Gamma \vdash e_2 : \mathcal{C}_2}{\Gamma \vdash e_1 e_2 : \mathcal{C}_1 \ltimes \mathcal{C}_2}\ \textsc{App} \qquad \frac{\Gamma \vdash e_1 : \mathcal{C}_1 \quad \Gamma,\, x : \mathcal{C}_1 \vdash e_2 : \mathcal{C}_2}{\Gamma \vdash \texttt{let}\ x = e_1\ \texttt{in}\ e_2 : \mathcal{C}_2}\ \textsc{Let}$$

$$\frac{\begin{array}{c}\Gamma \vdash e_1 : \{D_1 \rightsquigarrow \eta_1, \ldots, D_n \rightsquigarrow \eta_n\} \qquad \Gamma \vdash e_2 : \mathcal{C}' \\ \forall i = 1 \ldots n :\ \Gamma,\ \texttt{hd} : \{\emptyset \rightsquigarrow (\widehat{\lambda}^s_p.p\,(s-1))\eta_i\}, \\ \qquad\qquad \texttt{tl} : \{\emptyset \rightsquigarrow (\widehat{\lambda}^s_p.\,\mathsf{List}\ (s-1)\ p)\eta_i\} \vdash e_3 : \mathcal{C}_i \\ \mathcal{C}'' = \cup_{i=1}^{n} \left(\left(D_i, (\widehat{\lambda}^s_p.s)\eta_i >= 1 \rightsquigarrow \mathcal{C}_i \right) \cup \left(D_i, (\widehat{\lambda}^s_p.s)\eta_i = 0 \rightsquigarrow \mathcal{C}' \right) \right)\end{array}}{\begin{array}{l}\Gamma \vdash \texttt{match}\ e_1\ \texttt{with} \mid \texttt{nil} \Rightarrow e_2 \qquad\qquad : \mathcal{C}'' \\ \qquad\qquad\qquad\qquad \mid \texttt{cons hd tl} \Rightarrow e_3\end{array}}\ \textsc{Match}$$

Fig. 8. Rules for expressions

4 Examples

concat

$$\begin{array}{l}\texttt{concat x y} :: \mathsf{L}(\alpha) \to \mathsf{L}(\alpha) \to \mathsf{L}(\alpha) \\ \qquad :: \widehat{\lambda}^{s_x}_{p_x}.\widehat{\lambda}^{s_y}_{p_y}.\,\mathsf{List}\ (s_x + s_y)\ (\mathsf{Shift}\ p_y\ s_y\ p_x) \\ \qquad = \texttt{match}\ x\ \texttt{with} \mid \texttt{nil} \Rightarrow y \\ \qquad\qquad\qquad\qquad \mid \texttt{cons hd tl} \Rightarrow \texttt{cons hd}\ (\texttt{concat tl}\ y)\end{array}$$

Here x and y are of type $\mathsf{L}(\alpha)$. According to the Bind rule, we are creating fresh size expressions for the arguments by using the function fresh. Assuming that the fresh size expressions are $\mathsf{List}\ s_x\ (\lambda i.p_x\ i)$ and $\mathsf{List}\ s_y\ (\lambda j.p_y\ j)$ for x and y,

respectively, we need to prove the following entailment:

$$
\begin{array}{l}
\{ \\
\quad \texttt{concat} : \{\emptyset \rightsquigarrow \widehat{\lambda}^{s_x}_{p_x}.\widehat{\lambda}^{s_y}_{p_y}.\,\mathsf{List}\ (s_x + s_y)\ (\mathsf{Shift}\ p_y\, s_y\, p_x)\}, \\
\quad \texttt{x} : \{\emptyset \rightsquigarrow \mathsf{List}\ s_x\ (\lambda i.p_x\ i)\}, \qquad \texttt{y} : \{\emptyset \rightsquigarrow \mathsf{List}\ s_y\ (\lambda j.p_y\ j)\} \\
\} \\
\quad \vdash \texttt{match}\ x\ \texttt{with} \mid \texttt{nil} \Rightarrow y \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad : \mathcal{C}'' \\
\qquad\qquad\qquad\quad \mid \texttt{cons hd tl} \Rightarrow \texttt{cons hd}\ (\texttt{concat tl}\ y)
\end{array}
$$

Next we can calculate the assumed size of the nil branch and the condition using the VAR rule:

$$
\frac{}{\Gamma\ \vdash y : \{\emptyset \rightsquigarrow \mathsf{List}\ s_y\ (\lambda j.p_y\ j)\}} \qquad \frac{}{\Gamma\ \vdash x : \{\emptyset \rightsquigarrow \mathsf{List}\ s_x\ (\lambda i.p_x\ i)\}}
$$

It is not necessary, but possible to reduce the size expressions during the inference. To save space we will do it so. Using the APP and VAR rules multiple times, we can infer the following:

$$
\begin{array}{l}
\{ \\
\quad \texttt{concat} : \{\emptyset \rightsquigarrow \widehat{\lambda}^{s_x}_{p_x}.\widehat{\lambda}^{s_y}_{p_y}.\,\mathsf{List}\ (s_x + s_y)\ (\mathsf{Shift}\ p_y\, s_y\, p_x)\}, \\
\quad \texttt{x} : \{\emptyset \rightsquigarrow \mathsf{List}\ s_x\ (\lambda i.p_x\ i)\}, \qquad \texttt{y} : \{\emptyset \rightsquigarrow \mathsf{List}\ s_y\ (\lambda j.p_y\ j)\} \\
\quad \texttt{hd} : \{\emptyset \rightsquigarrow p_x\ (s_x - 1)\}, \quad \texttt{tl} : \{\emptyset \rightsquigarrow \mathsf{List}\ (s_x - 1)\ (\lambda i.p_x\ i)\} \\
\} \\
\quad \vdash (\texttt{concat tl}\ y) : \{\emptyset \rightsquigarrow \mathsf{List}\ (s_x - 1 + s_y)(\mathsf{Shift}\ \ p_y\ \ s_y\ \ p_x)\} \\
\qquad\qquad\qquad\qquad\qquad\qquad \ldots
\end{array}
$$

$$
\begin{array}{l}
\{\ldots\} \\
\vdash \texttt{cons hd}\ (\texttt{concat tl}\ y) : \{\emptyset \rightsquigarrow \\
\quad \mathsf{List}\ (s_x - 1 + s_y + 1)\ (\mathsf{Shift}\ (\mathsf{Shift}\ \ p_y\ \ s_y\ \ p_x)\ (s_x - 1 + s_y)\ (\lambda i.\,(p_x(s-1)))) \\
\}
\end{array}
$$

Calculating $(\widehat{\lambda}^{s}_{p}.s)\eta_1$ which can be reduced to s_x, we can finish the inference of the assumption set:

$$
\begin{array}{l}
\mathrm{C}'' = \{ \\
\qquad \{s_x = 0\} \rightsquigarrow \mathsf{List}\ \ s_y\ \ (\lambda j.p_y\ j) \\
\qquad \{s_x >= 1\} \rightsquigarrow \mathsf{List}\ (s_x - 1 + s_y + 1) \\
\qquad\qquad (\mathsf{Shift}\ (\mathsf{Shift}\ \ p_y\ \ s_y\ \ p_x)\ (s_x - 1 + s_y)\ (\lambda i.\,(p_x(s-1)))) \\
\quad \}
\end{array}
$$

Finally, the verification conditions from the BIND rule:

$$
\begin{array}{l}
s_x = 0\ \Vdash \mathsf{List}\ \ s_y\ \ (\lambda j.p_y\ j) = \mathsf{List}\ (s_x + s_y)\ (\mathsf{Shift}\ p_y\, s_y\, p_x) \\
s_x >= 1\ \Vdash \mathsf{List}\ (s_x - 1 + s_y + 1) \\
\qquad (\mathsf{Shift}\ (\mathsf{Shift}\ \ p_y\ \ s_y\ \ p_x)\ (s_x - 1 + s_y)\ (\lambda i.\,(p_x(s-1)))) = \\
\qquad \mathsf{List}\ (s_x + s_y)\ (\mathsf{Shift}\ p_y\, s_y\, p_x)
\end{array}
$$

Now, using the rules of the verification conditions:

$$s_x = 0 \ \Vdash s_y = s_x + s_y \tag{1}$$
$$s_x = 0, s_i < s_x - 1 + s_y + 1 \ \Vdash p_y\, s_i = \mathsf{Shift}\ p_y\, s_y\, p_x\, s_i \tag{2}$$
$$s_x >= 1 \ \Vdash s_x - 1 + s_y + 1 = s_x + s_y \tag{3}$$
$$\begin{aligned} s_x >= 1, s_i < s_x - 1 + s_y + 1 \ \Vdash \\ \mathsf{Shift}\ (\mathsf{Shift}\ p_y\ s_y\ p_x)\ (s_x - 1 + s_y)\ (\lambda i.\,(p_x(s-1))) \\ = \mathsf{Shift}\ p_y\, s_y\, p_x \end{aligned} \tag{4}$$

Here, (1) and (3) is true, and (2) can be reduced as follows.

$$s_x = 0, s_i < s_x - 1 + s_y + 1, s_i < s_y \ \Vdash p_y\, s_i = p_y\, s_i \tag{5}$$
$$s_x = 0, s_i < s_x - 1 + s_y + 1, s_i >= s_y \ \Vdash p_y\, s_i = p_x\,(s_i - s_y) \tag{6}$$

Where (5) is true and in the conditions of (6) is not satisfiable. Equation (4) can be resolved similarly.

t3 The most interesting question is how can the size expressions handle such a polymorphism when an argument can be a list and even a function. To demonstrate this case we define the following function:

$$\mathsf{t3}\ f\ x :: (\alpha \to \alpha) \to \alpha \to \alpha :: \lambda s.\lambda p.s(s(sp)) = f(f(f\,x))$$

It is easy to check the type of this function so it is left for the reader. The interesting part is when we use this function in different kinds of expressions:

$$\mathsf{t}\ x :: \mathsf{L}(\alpha) \to \mathsf{L}(\alpha) :: \eta = (\mathsf{t3}\ \mathsf{t3})\ \mathsf{addone} x$$

In this example the assumed size for $t3\,t3$ is

$$\begin{gathered} \{\emptyset \rightsquigarrow \big(\lambda f.\lambda x.f(f(fx))\big)\big(\lambda f.\lambda x.f(f(fx))\big)\} \to^* \\ \{\emptyset \rightsquigarrow \lambda f.\lambda x.\underbrace{f(f(f\ldots(fx)\ldots)}_{27 \text{ applications of } f}\} \end{gathered}$$

We want to prove that the size of t is $\widehat{\lambda}^s_f.\,\mathsf{List}(s+27)(\lambda x.\,\mathsf{Unsized})$, using the fact that addone has size $\{\emptyset \rightsquigarrow f_{1:}\}$ where $f_{1:} = \widehat{\lambda}^s_f.\,\mathsf{List}\,(s+1)\,(\lambda x.\,\mathsf{Unsized})$. To continue with our example we apply fresh variables (eg. $\mathsf{List}\ a\,(\lambda y.\,\mathsf{Unsized})$) to η and also apply the rules to finish the calculation of the assumed sized:

$$\begin{aligned} &(\lambda x.\underbrace{f_{1:}(f_{1:}\ldots(f_{1:}x)\ldots)}_{27 \text{ applications of } f_{1:}})(\mathsf{List}\ a\,(\lambda y.\,\mathsf{Unsized})) \to \\ &\to \underbrace{f_{1:}\Big(f_{1:}\ldots\Big((\widehat{\lambda}^s_f.\,\mathsf{List}\,(l+1)\,(\lambda x.\,\mathsf{Unsized}))(\mathsf{List}\ a\,(\lambda y.\,\mathsf{Unsized}))\Big)\ldots\Big)}_{27 \text{ applications of } f_{1:}} \to \\ &\to \underbrace{f_{1:}\Big(f_{1:}\ldots\Big(\mathsf{List}\,(a+1)\,(\lambda x.\,\mathsf{Unsized})\Big)\ldots\Big)}_{26 \text{ applications of } f_{1:}} \to^* \mathsf{List}\,(a+27)\,(\lambda x.\,\mathsf{Unsized}) \end{aligned}$$

The following two expressions can be checked similarly:

$$\mathtt{t}\ x :: \mathsf{L}(\alpha) \to \mathsf{L}(\alpha) :: \widehat{\lambda}^{s}_{f}.\, \mathsf{List}\,(s+9)\,\lambda y.\, \mathsf{Unsized} = \mathsf{t3}\ (\mathsf{t3}\ \mathsf{addone})\ x$$
$$\mathtt{t}\ x :: \mathtt{Int} \to \mathtt{Int} :: \lambda x.\, \mathsf{Unsized} = \mathsf{t3}\ \mathtt{succ}\ x$$

5 Normalizability

Being an extended lambda calculus, our size expressions are certainly Turing complete. It means with the help of some tricky constructs any function of the language can be translated directly into size expression. On one hand it is good, because it proves that our language is able to express the size dependencies of any function. On the other hand it makes the size analysis undecidable.

However sometimes there is no better way to describe the size dependencies of a function, but with itself. One example is t3 which is detailed in section 4.

Size expressions can be seen as a simplification of the function capturing only the necessary information, it is especially important for a recursive function. For a recursive function it is not possible to write a recursive size expression, but recursion can be expressed by the fixed point combinator ($Y = (\lambda f.(\lambda x.f(xx))(\lambda x.f(xx)))$. For example:

$$\mathsf{fix}\ f :: (\alpha \to \alpha) \to \alpha :: \lambda s.Ys = f\ (\mathsf{fix}\ f)$$

Of course the checker is not able to prove this, as the size expression is not normalizable. To ensure the normalizability property of a size expression, and thus ensure the termination of the size checking, we use a type system. Our choice of type system is System F. We assume the usual `Bool`, `Nat`, `Unit` and product types with the usual operations defined. The following type is also predefined:

$$L_a := \mathtt{Nat} \times (\mathtt{Nat} \to a)$$

This type expresses the fact that a size of a list is a tuple of the length of the list and a map holding the sizes of the elements of the list.

5.1 Types of size operators

The Unsized can be easily represented by the `Unit` type:

$$\mathsf{Unsized} = \mathsf{unit} : \mathtt{Unit}$$

List corresponds to the data constructor of a pair:

$$\begin{aligned}\mathsf{List} = {} & \Lambda A.\lambda s^{\mathtt{Nat}}.\lambda f^{\mathtt{Nat}\to A}.\ <s, f> \\ & : \forall A.\mathtt{Nat} \to (\mathtt{Nat} \to A) \to L_A\end{aligned}$$

If $\widehat{\lambda}^{s}_{f}.e$ is seen as a syntactic sugar for $\mathsf{Unlist}\ A\,B\,(\lambda s f.e)$, where A and B are types. Now it is easy to describe $\widehat{\lambda}^{s}_{f}.$ with the help of the usual projections π_1 and π_2:

$$\begin{aligned}\mathsf{Unlist} = {} & \Lambda A.\Lambda B.\lambda f^{\mathtt{Nat}\to(\mathtt{Nat}\to A)\to B}.\lambda t^{L_A}.f\,(\pi^1 t)\,(\pi^2 t) \\ & : \forall A.\forall B.(\mathtt{Nat} \to (\mathtt{Nat} \to A) \to B) \to L_A \to B\end{aligned}$$

The last thing to do is to define the Shift function:

$$\begin{aligned}\mathsf{Shift} &= \Lambda A.\lambda f^{\mathtt{Nat}\to A}.\lambda n^{\mathtt{Nat}}.\lambda g^{\mathtt{Nat}\to A}.\lambda x^{\mathtt{Nat}}.\mathsf{IF}\,A\,(x<n)\,(f\,x)\,(g\,(x-n))\\ &: \forall(A.\mathtt{Nat}\to A)\to\mathtt{Nat}\to(\mathtt{Nat}\to A)\to\mathtt{Nat}\to A\end{aligned}$$

Because type inference for System F is not decidable we can use HMF [6] or MLF [5] for inference if we can tell the type of the size expression. These types can be obtained from the underlying type using the following function:

$$\begin{aligned}\mathit{SizeType}(\forall\alpha.a) &= \forall\alpha.\mathit{SizeType}(a)\\ \mathit{SizeType}(a\to b) &= \mathit{SizeType}(a)\to\mathit{SizeType}(b)\\ \mathit{SizeType}(\mathsf{L}(a)) &= L_a\\ \mathit{SizeType}(\tau) &= \tau \quad \text{if } \tau \text{ is a type variable}\\ \mathit{SizeType}(\tau) &= \mathtt{Unit} \qquad \text{otherwise}\end{aligned}$$

The following table gives some examples:

$$\begin{array}{rl}\mathtt{Nat}\to\mathtt{Nat} & U\to U\\ \mathsf{L}(\mathtt{Nat})\to\mathsf{L}(\mathtt{Nat}) & L_U\to L_U\\ \mathsf{L}(a)\to\mathsf{L}(a) & \forall a.L_a\to L_a\\ (a\to b)\to\mathsf{L}(a)\to\mathsf{L}(b) & \forall a\,b.(a\to b)\to L_a\to L_b\\ \mathsf{L}(a)\to\mathsf{L}(\mathsf{L}(a)) & \forall a.L_a\to L_{L_a}\end{array}$$

6 Semantics and soundness of the sizing rules

In the presented work *soundness* of the sizing rules means the following. If the size-checking procedure accepts the size annotation of a program expression then this annotation correctly reflects the output-on-input size dependency of the program expression. For instance, the annotation $\widehat{\lambda}^s_p.\,\mathsf{List}\,(s+1)(\lambda i.\,\mathsf{Unsized})$ for the function addone must guarantee that if an input list has the length s then the corresponding output list has the length $s+1$. Moreover, it guarantees that output-list elements are unsized, i.e. they are integer numbers.

Soundness is to be proven w.r.t. some reasonable semantic model, where e.g. the term $\mathsf{List}\,25(\lambda i.\,\mathsf{Unsized})$ is interpreted as the set of all the lists of integers of length 25. One can follow two approaches. The first one would be to consider a denotational semantics a-la the denotational semantics of Haskell, which uses the fixed point semantics for recursive definitions. It allows to make *total-correctness* statements about sizings: if a program passes type-checking then it terminates.

For the time being it is decided to follow the second approach: to consider a state-based semantics, where a program expression in general changes a *state* which stores semantic *values* in locations named by program variables. This semantic allows to consider less challenging *partial-correctness* statements: the annotation of a program represents the correct output-on-input size dependency only on inputs, on which the program terminates. Values are integers, and finite vectors and (higher-order) functions over values, and states are maps from program variables to values:

$$Val := \mathcal{Z} \mid Val^* \mid Val \times \ldots \times Val \to Val$$
$$State : Exp\,Var \to Val$$

A function $f : Val \times \ldots \times Val \to Val$ is understood as an (infinite) table where for each input $\bar{v}$, on which f is defined, there is a corresponding row that gives $f(\bar{v})$. If f is defined algorithmically, it is written as a lambda-term. For instance, adding two integers $\lambda\ x\ y.\ x + y$ represents the table with three columns, where the first two columns list all possible pairs (x, y), and the third one gives the corresponding sum.

Semantics is defined in two steps. First, the total-evaluation semantics $Sem : Expr \to State \to Val$ and, second, partial evaluation semantics $SemPart$ is defined via Sem. Total-evaluation Sem is defined inductively on program expressions and states:

$$\begin{array}{ll}
Sem\ x\ s & = s(x),\ \text{where } x \in dom(s) \\
Sem\ \texttt{nil}\ s & = () \\
Sem(\texttt{cons}\ e_1\ e_2)\ s & = (Sem\ e_1\ s) :: (Sem\ e_2\ s) \\
Sem\ n\ s & = () \\
Sem(e_1\ e_2)\ s & = (Sem\ e_1\ s)(Sem\ e_2\ s),\ \text{if } e_1 \notin FName
\end{array}$$

$$\begin{array}{ll}
Sem(\texttt{f}\ e_1 \ldots e_n)\ s & = Sem\ e_{\texttt{f}}\ [x_i := v_i]_{i=1}^n, \\
 & \text{where } v_i = (Sem\ e_i\ s) \\
 & \text{and } e_{\texttt{f}} \text{ is the body of } \texttt{f} \\
Sem\ (\texttt{let}\ x = e_1\ \texttt{in}\ e_2)\ s & = Sem\ e_2\ s[x := (Sem\ e_1\ s)] \\
Sem\ (\texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3)\ s & = \begin{cases} Sem\ e_2\ s, & \text{if } (Sem\ e_1\ s) > 0 \\ Sem\ e_3\ s, & \text{otherwise} \end{cases} \\
Sem\ (\texttt{match}\ e_1\ \texttt{with} \mid \texttt{nil} \Rightarrow e_2 \mid \texttt{cons hd tl} \Rightarrow e_3)\ s & = \begin{cases} Sem\ \ e_2\ s, & \text{if } (Sem\ e_1\ s) = () \\ Sem\ \ e_3\ s[\texttt{hd} := v_{\texttt{hd}}, \texttt{tl} := v_{\texttt{tl}}] & \\ \qquad \text{if } (Sem\ e_1\ s) = v_{\texttt{hd}} :: v_{\texttt{tl}} & \end{cases}
\end{array}$$

The partial evaluation $SemPart : Expr \to State \to Val$ is given by

$$SemPart\ e\ s := \lambda\ v_1 \ldots v_n.\ (Sem\ e\ s[x_i := v_i]_{i=1}^n)$$

where $[x_i := v_i]_{i=1}^n = FV(e) \setminus dom(s)$.

Now, to make soundness statement, introduce a semantic size M on values:

$$\begin{array}{ll}
M\ m & = () \\
M\ (m_1, \ldots, m_n) & = n \\
M\ (v_1, \ldots, v_n) & = (M\ v_1, \ldots, M\ v_n) \\
M\ f & = \{(M\ \bar{v}) \mapsto \big(M\ f(\bar{v})\big)\}_{\bar{v} \in dom(()f)}
\end{array}$$

The last row implies that semantic functions f must be *shapely*: that is two values $\bar{v}_1$ and $\bar{v}_2$ of the same size must be mapped onto values of the same size: $M\ f(\bar{v}_1) = M\ f(\bar{v}_2)$.

As expected, the soundness theorem states that if a well-typed program expression is evaluated on a state s, that maps any variable onto a value of the size

according to a sizing context Γ, then the evaluated value has the size represented by the output sizing term. Formally, if $\Gamma \vdash e :: \tau :: \eta$, then

$$\forall\ s.\ \Big(\big(\forall\ x \in dom(s).\ M\ s(x) = \Gamma(x)\big) \wedge SemPart\ e\ s\ = v\Big) \Longrightarrow (Mv = \eta)$$

Recall, that a term $\mathsf{List}\, t_1\, \big(\lambda i.t_2(i)\big)$ represents a finite map defined on $\{0, \ldots, t_1\}$ that sends i to $t_2(i)$. The theorem is to be proven by induction on the structure of expression e.

7 Related work

The structure of size expressions in our research is close to the approach of A. Abel [1], who has applied sized types for termination analysis of higher-order functional programs. For instance, in his notation sized lists of type A of length $\imath$ are defined as $\lambda\, \imath\, A.\mu^{\imath}.\mathbf{1}+A\times X$ and size expressions are higher-order arithmetic expressions with λ-abstraction as well. The difference is that in that work one uses linear arithmetic over *ordinals*, where ordinals represent zero-order sizes. Moreover, in that research size information is not a stand-alone formalism, but a part of a dependent-type system.

In the paper [8] the authors go beyond linear arithmetic. For a given higher-order functional program, they obtain a set of first-order arithmetic constraints over unknown cost functions f. Solving these constraints w.r.t. f gives the desired costs of the program. The underlying arithmetic is the arithmetic over naturals, extended with undefined ϵ and unbounded ω values, equipped with a natural linear order. Size expressions admit addition $+$, multiplication $*$ and subtraction of a constant $-n$, thus such expressions are monotonic. Function types are annotated with natural numbers (*latencies*), e.g. $\alpha \xrightarrow{l} \beta$, so it may be conveniently interpreted as an increment in cost consumption, like l clock ticks if the resource of interest is time. Our approach is different in a sense that we aim at expressing size dependencies directly in terms of sizes of inputs, bypassing latencies.

In paper [2] the authors approach to complexity analysis of an imperative language, which is a version of Gödel's T. It is done via abstract interpretation of programs in a semiring of matrices. Informally, matrices represent data flow along program variables. The authors give an upper bound for the return values in terms of initial values. However, this is a conjecture and no proof is given. Similarly the conjecture about the existence of an abstract interpretation is not proven.

In recent paper [4] the authors develop amortized cost analysis for a higher-order functional language *Shopenhauer*. The analysis is generic, that is it is applicable to different sorts of resources: heap usage, stack size and the number of function calls. Type-derivation procedure generates linear constraints, solving of which gives desirable upper bounds. The analysis always succeeds, if bounds are linear. So far, the methodology does not support polymorphic recursion.

8 Conclusion and future plans

We have presented a size analysis for higher order functions for a higher-order polymorphic functional language. The calculus is based upon the lambda-calculus extending it with arithmetic operations and special operators for finite maps representing sizes of the elements of lists.

A calculus is introduced which can be used to implement an algorithm to generate verification conditions. The size expressions are generally not normalizable, but we have shown a sufficient condition to ensure normalizability. Normalizability is required to ensure the termination of the verification condition solver.

We are investigating the possibility to use polynomial interpolation [7] to infer size expressions for higher-order functions as well.

References

1. Abel, A.: A Polymorphic Lambda-Calculus with Sized Higher-Order Types. Ph.D. thesis, Ludwig-Maximilians University, Munich (2006)
2. Avery, J., Kristiansen, L., Moyen, J.Y.: Static complexity analysis of higher order programs. In: van Eekelen, M., Shkaravska, O. (eds.) Proceedings of the First international conference on FOundational and Practical Aspects of Resource Analysis (FOPARA). LNCS, vol. 6324, pp. 84–99. Springer-Verlag, Berlin, Heidelberg (2010), http://portal.acm.org/citation.cfm?id=1886124.1886130
3. Góbi, A., Shkaravska, O., van Eekelen, M.: Size analysis of higher-order functions. In: Peña, R., van Eekelen, M. (eds.) Proceedings of the 12th International Symposium on Trends in Functional Programming (TFP2011). pp. 77–91. No. Tech. Rep. SIC-07/11, Madrid, Spain (may 2011)
4. Jost, S., Hammond, K., Loidl, H.W., Hofmann, M.: Static determination of quantitative resource usage for higher-order programs. SIGPLAN Not. 45, 223–236 (January 2010), http://doi.acm.org/10.1145/1707801.1706327
5. Le Botlan, D., Rémy, D.: MLF: raising ml to the power of system f. In: Proceedings of the eighth ACM SIGPLAN international conference on Functional programming. pp. 27–38. ICFP '03, ACM, New York, NY, USA (2003), http://doi.acm.org/10.1145/944705.944709
6. Leijen, D.: HMF: simple type inference for first-class polymorphism. In: Proceeding of the 13th ACM SIGPLAN international conference on Functional programming. pp. 283–294. ICFP '08, ACM, New York, NY, USA (2008), http://doi.acm.org/10.1145/1411204.1411245
7. Shkaravska, O., van Eekelen, M.C.J.D., van Kesteren, R.: Polynomial size analysis of first-order shapely functions. Logical Methods in Computer Science 5(2) (2009)
8. Vasconcelos, P.B., Hammond, K.: Inferring cost equations for recursive, polymorphic and higher-order functional programs. In: Trinder, P., Michaelson, G., Peña, R. (eds.) Revised selected papers of the 15th international symposium on Implementation of Functional Languages (IFL'03). LNCS, vol. 3145, pp. 86–101. Springer-Verlag, Edinburgh, UK, September 8-11, 2003 (2004)