STUDIA UNIV. BABEŞ–BOLYAI, INFORMATICA, Volume **LIX**, Special Issue 1, 2014
10th Joint Conference on Mathematics and Computer Science, Cluj-Napoca, May 21-25, 2014

EMBEDDED RESOURCE TOOL IN HASKELL

ATTILA GÓBI, TAMÁS KOZSIK, AND BARNABÁS KRÁLIK

Abstract. In our previous work [4], we have created a way to check size annotations of higher-order polymorphic functional programs supporting nested lists. By extending the lambda-calculus, these annotations are able to express the relations between sizes of arguments and those of the corresponding results of functions. These relations are exact, and can be non-linear and non-monotonic polynomials. We provided a way for verification condition generation as well. This paper focuses on how it is possible to implement this extended lambda calculus as an embedded DSL.

1. Introduction

It is a well-known fact that the vast majority of processors are used in embedded systems, where resource consumption of software is severely limited. Software development techniques, which enable reasoning about resource use, are extremely useful for such applications. Analyses on resource consumption of programs typically rely on size information on program values, such as length of lists.

There are different approaches to work with size information. A widespread approach is to use sized types [6] and subtypes, which allows us to compute an upper bound for the possible sizes of values assigned to a variable.

Previously, in [4] a size calculus has been defined to express relations between sizes of expressions, namely the arguments and result of functions. With this hypothetical size expression on a function, it is possible to generate verification conditions using the body of the function. If these verification conditions can be discharged, the function is proved to confirm to the hypothetical size expression.

Received by the editors: May 1, 2014.

2010 *Mathematics Subject Classification.* 68N18, 68N30.

1998 *CR Categories and Descriptors.* D.1.1 [**Programming Techniques**]: Applicative (Functional) Programming – *embedding a functional language into a functional language*; D.3.1 [**Programming Languages**]: Formal Definitions and Theory – *syntax, semantics*.

Key words and phrases. resource analysis, embedded domain-specific language.

Supported by the Ministry of Human Resources of Hungary, contract No. 18370-9/2013/TUDPOL.

Our calculus considerably differs from the sized types approach. In our calculus we can express exact size relations, even in non-linear, non-monotonic cases. For instance, we can describe the exact size for nested lists, in contrast to sized types, where the upper bound is determined by the length of the longest element list.

The size calculus can be introduced as a domain-specific language (DSL), the primitives of which allow us to describe computations in an extension of the typed lambda calculus, as well as size information attached to potentially recursive function definitions. This DSL can be implemented as an embedding into the non-strict pure functional language Haskell [7].

This embedding exposes a number of challenges. Firstly, we need two different sublanguages, both extending the lambda calculus, and hence, sharing certain constructs. Secondly, the typing of these two sublanguages are interdependent. Thirdly, function calls should be observable: this is essential for e.g. the proper generation of verification conditions. Fourthly, we need a design that facilitates extension of the DSL.

The rest of the paper is structured as follows. In section 2, our earlier work is revisited to give some insight into the domain specific language defining the size calculus. Section 3 explains how this DSL can be embedded into Haskell. Section 4 summarizes related work, and section 5 concludes the paper.

2. DSL for describing resource relations

One of the key constructs in our language is the size expression. Size expressions are used to describe exact relations between size of return value and those of the parameters of a function. These expressions are an extension of the lambda calculus with constructs for expressing the sizes of lists. The abstract syntactical structure of size expressions can be seen on Figure 1.

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

Figure 1. Syntax of size expressions

The simplest of them all is the Unsized – this is the size of the primitive values such as values of type `Int`. The empty list corresponds to the size expression List 0 ($\lambda i.\bot$). This means that the empty list has 0 elements and every element has size $\bot$. The bottom is used as a placeholder: it does not equal to anything else, and its usual interpretation is an error. We saw that

the size expression for `Int` is Unsized, so one possible size expression for the list containing a single integer is List 1 ($\lambda i.$ Unsized). The second parameter of the List expression is a function which maps a size to each index, where the index is a natural number, and the index of the last item of the list is zero. This notation might seem unintuitive at first sight, however in [4] we have given a detailed explanation for this choice. The remaining syntactic constructs can be best understood by going through the reduction rules of the size calculus, as shown in Figure 2.

$$(\lambda p.\eta_1)(\eta_2) \to_\beta (\eta_1[p := \eta_2])$$
$$(\widehat{\lambda}^s_p.e)(\mathsf{List}\,\eta_1\,\eta_2) \to_\beta (e[s := \eta_1, p := \eta_2])$$
$$\mathsf{Shift}\,\eta_1\,s\,\eta_2\,i \to_\beta \begin{cases} \eta_1 i & \text{if } i < s \\ \eta_2(i-s) & \text{otherwise} \end{cases}$$

FIGURE 2. Reduction rules of the size calculus

The first reduction is the usual beta reduction from lambda calculus. The second rule tells us that $\widehat{\lambda}$ is the dual of List, that is it can be used to pattern match on list sizes. The third rule is used to compute the size of an element of the concatenation of two lists. The combinator Shift is used to concatenate size functions. The size expression for the concatenation of two lists can be given with the following formula.

$$\widehat{\lambda}^{s_x}_{p_x}.\widehat{\lambda}^{s_y}_{p_y}.\,\mathsf{List}\ (s_x + s_y)\ (\mathsf{Shift}\ p_y\,s_y\,p_x)$$

The expression $\mathsf{Shift}\,e_1\,s\,e_2$ gives the size function of the list obtained by inserting the last s elements of e_1 before e_2. That is:

$$\eta_1 = \{0 \mapsto \eta_1^0, 1 \mapsto \eta_1^1, \ldots, n \mapsto \eta_1^n\}$$
$$\eta_2 = \{0 \mapsto \eta_2^0, 1 \mapsto \eta_2^1, \ldots, m \mapsto \eta_2^m\}$$
$$\mathsf{Shift}\,\eta_1\,s\,\eta_2 = \{0 \mapsto \eta_1^0, 1 \mapsto \eta_1^1, \ldots, s-1 \mapsto \eta_1^{s-1}, s \mapsto \eta_2^0, s+1 \mapsto \eta_2^1, \ldots\}$$

Another possible size expression for the singleton integer list can be given with this Shift operator, namely: List 1 (Shift ($\lambda i.\bot$) 0 ($\lambda i.$ Unsized)). This, again, means that the list has one element, and using the reduction rules it is easy to see that the size of this element is Unsized.

This leads us to the definition of size equality of lists – the size of two lists are equal, if their length equals, and all of their elements have the same size.

$$\mathsf{List}\,s_1\,\eta_1 = \mathsf{List}\,s_2\,\eta_2 \Leftrightarrow s_1 = s_2 \wedge \forall i \in \{0, \ldots, s_1 - 1\} :\ \eta_1\,i = \eta_2\,i$$

The other key component of our DSL is the sublanguage which describes the computational aspect of a program. This sublanguage is a variation of a typed lambda calculus, extended with polymorphic homogeneous lists (with the $\mathsf{L}()$ type constructor) and pattern matching on lists (`match-with`). Top-level bindings introduce recursive definitions in the computational sublanguage, and connect the size expression and computational sublanguages, as it can be seen on the following example.

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

Here the `concat` function has two arguments – `x` and `y`; a type declaration – after the first `::`; a size expression declaration – after the second `::`; and a body – after the equality sign.

3. Finally tagless embedding

Now, let us see how to embed our DSL into Haskell. We follow the technique introduced by [2]. In that paper, the lambda calculus is embedded into OCaml and Haskell. The advantages of this embedding are that it is extensible, and interpreting it incurs no significant runtime overhead. This latter property is the result of eliminating type tags during compilation. The higher-order abstract syntax of a lambda calculus looks as follows.

```
class Lambda l where
    lam :: (l a -> l b) -> l (a -> b)
    app :: l (a -> b) -> l a -> l b
    lit :: Int -> l Int
```

The expressions of this lambda calculus can be lambda abstractions, applications and integer literals. Observe that the `lam` function takes a Haskell function that operates on values of the embedded language, and lifts this function into the embedded language.

Instances of the type class `Lambda` are the interpreters of the embedded language. As a consequence embedded expressions can be polymorphic global functions, where these functions are polymorphic in the interpreter of the DSL chosen for evaluation. Such an interpreter can be provided by the following instance declaration.

```
newtype Q a = Q { unQ :: a }
instance Lambda Q where
    lit = Q
    lam a = Q (unQ.a.Q)
    app a b = Q $ unQ a (unQ b)
```

```
eval :: Q a -> a
eval = unQ
```

The `newtype` keyword introduces a tagless algebraic data type (one with a single possible data constructor). The compiler can optimize away the `Q` constructor and the `unQ` selector. The entry point of the interpreter is the `eval` function, which is defined surprisingly simply. It creates a Haskell function from an expression of the embedded language.

We can now turn our attention to finer details, such as operators. We introduced syntactic categories for infix operators with different associativity. This technique is useful for later extension of the embedded language with new custom binary operators. Note that we used the qualified name of the operators defined in Prelude, and Prelude is imported with the `qualified` qualifier to avoid name clashes.

```
class (Lambda l) => LOps l where
    infixop  :: String -> Int -> (a -> b -> c) -> l a -> l b -> l c
    infixopr :: String -> Int -> (a -> b -> c) -> l a -> l b -> l c
    infixopl :: String -> Int -> (a -> b -> c) -> l a -> l b -> l c

(+) :: (LOps l) => l Int -> l Int -> l Int
(+) = infixopl "+" 6 (Prelude.+)

(-) :: (LOps l) => l Int -> l Int -> l Int
(-) = infixopl "-" 6 (Prelude.-)

(*) :: (LOps l) => l Int -> l Int -> l Int
(*) = infixopl "*" 7 (Prelude.*)
```

Embedding size expressions of the DSL is carried out in the following way. The only difference in the syntax is that we used the `slam` function to represent $\widehat{\lambda}$ in order to simplify typing (no pun intended).

```
class (LOps l) => Size l where
    list :: l Int -> l (Int -> a) -> l [a]
    slam :: (l Int -> l (Int -> a) -> l b) -> l ([a] -> b)
    shift :: l (Int -> a) -> l Int -> l (Int -> a) -> l (Int -> a)
    unsized :: l Unsized
    bottom :: l a
```

To illustrate the use of the size expression embedding, consider the following definition of `concatSize`, which gives the size relation for the usual list concatenation.

```
concatSize :: Size l => l ([a] -> [a] -> [a])
concatSize = slam $ \s1 f1 -> slam $ \s2 f2 ->
    list (s1 + s2) $ shift f1 s1 f2
```

Our goal is to make recursion observable. To achieve this, the `bind` is introduced as follows.

```
class (Exp e, Size (SizeExp e)) => SizedFun e where
  type SizeExp e :: * -> *
  bind :: Infer a b => String -> SizeExp e a -> e b -> e b
```

Here `SizeExp` is an *associated type constructor synonym* – a type function –, which maps actual interpreter `e` of the body to the interpreter of the size expression. Note that due to overlapping instances, type families cannot be used here to describe the mapping. Therefore, we use a workaround with the following `Infer` type class.

```
class Infer a b where
instance (Infer a b, Infer p q) => Infer (a->p) (b->q)
instance Infer a b => Infer [a] [b]
instance (a~b) => Infer a b
instance Infer Unsized Int
```

Note that `bind` corresponds to the type environment by capturing the name, the type, the size and the body of top-level definitions. For instance, if we want to print out a DSL expression, the `bind` function allows us to perform the printing in a context: if a function is to be printed, either the name or the body of the function will be output, depending on whether the function occurs in the outermost bind, or in a nested one.

```
class SContext s => SBContext s where
    bound :: Lens s Bool

instance SBContext s => SizedFun (S s) where
  type SizeExp (S s) = S s
  bind name size exp = S $ \ctxo -> if getL bound ctxo then
      showString name
      else let (s1, s2) = S.split2 (getL supply ctxo)
               ctx = setL bound True ctxo
      in showString name . showString " :: " .
         unS size (updateCtx s1 0 ctx) . showChar '\n'.
         showString name . showString " = " .
         unS exp (updateCtx s2 0 ctx)
```

This approach can avoid recursive expansion of definitions. When we print out `concat` with this approach, the following output is obtained.

```
concat ::  Λa,b.Λc,d.List (a+c) (Shift b a d)
concat = λe.λf.case e of [] => f; (g:h) => g:concat h f
```

The same technique can be used when generating verification conditions from programs written in the EDSL: the binding registers the size expression for a function, and non-binding occurrences of the same function can retrieve

this size expression from the environment, and use it as an assumption in the generated VCs.

4. Related work

Papers [1, 3] discuss methods of resource analysis that are the closest to our approach; however, their style of implementation is more of a library than that of a domain-specific language.

Brady and Hammond present a dependently typed core language called TT and define a framework within this language for representing size metrics and properties thereof. Dependent type systems lend a hand when formulating type-level statements about the resource usage of a given part of a program. Here, not only types but values as well can be used as parameters to a type constructor. The authors exploit this property by encoding size – as a natural number – into the types as just another type parameter.

Danielsson attacks the problem of execution time analysis in a similar manner. He expresses 'ticks' required by each function as type parameters of the individual functions' return types. Regular Agda is then used as a vehicle of implementation.

A drawback of using general-purpose lazy functional programming languages is that the behaviour of the garbage collector is very difficult to predict and that lazy evaluation might skip possible execution paths whatsoever; thus, the aforementioned implementations only give a not-too-tight upper bound on resource consumption. In the case of embedded systems, having explicit resource management and clearly visible execution paths might even pose as an advantage from the viewpoint of the engineers. Hughes and Pareto [5] use a minimal version of ML as a basis for their language embedded ML. This language is then extended using the notion of 'regions', which can be thought of short-lived heaps, explicitly introduced and disposed of by the programmer.

5. Conclusion

In this paper we describe a technique to embed a size calculus DSL into Haskell. This DSL can attach size expressions to functions of a computational language, and provides reduction rules to simplify such size expressions. Both the computational language and the size expression language are extensions of lambda calculus with lists and pattern matching.

The embedding, on the one hand, is tagless, and therefore fairly efficient, since the host language compiler translates EDSL programs into pure Haskell functions. On the other hand, it is necessary to explicitly mark function applications to achieve observability. However, this explicit application symbol can be turned implicit using Template Haskell [8].

One of the design goals of our embedding was to allow extensions to the EDSL to be added. Hence we used higher-order abstract syntax and type classes in the implementation.

References

[1] Edwin Brady and Kevin Hammond. A dependently typed framework for static analysis of program execution costs. In *In Revised selected papers from IFL 2005: 17th international workshop on implementation and application of functional languages*, pages 74–90. Springer, 2005.

[2] Jacques Carette, Oleg Kiselyov, and Chung-chieh Shan. Finally tagless, partially evaluated. In Zhong Shao, editor, *Programming Languages and Systems*, volume 4807 of *Lecture Notes in Computer Science*, pages 222–238. Springer Berlin Heidelberg, 2007.

[3] Nils Anders Danielsson. Lightweight semiformal time complexity analysis for purely functional data structures. In *Proceedings of the 35th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '08, pages 133–144, New York, NY, USA, 2008. ACM.

[4] Attila Góbi, Olha Shkaravska, and Marko van Eekelen. Higher-order size checking without subtyping. In Hans-Wolfgang Loidl and Ricardo Peña, editors, *Trends in Functional Programming*, volume 7829 of *Lecture Notes in Computer Science*, pages 53–68. Springer Berlin Heidelberg, 2013.

[5] John Hughes and Lars Pareto. Recursion and dynamic data structures in bounded space: Towards embedded ML programming. In *Proceedings of the 4th ACM SIGPLAN International Conference on Functional Programming (ICFP'99)*, pages 70–81, Paris, France, 1999. ACM.

[6] John Hughes, Lars Pareto, and Amr Sabry. Proving the correctness of reactive systems using sized types. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages (POPL'96)*, pages 410–423, St. Petersburg Beach, Florida, USA, 1996. ACM.

[7] Simon Marlow et al. Haskell 2010 language report, 2010.

[8] Tim Sheard and Simon Peyton Jones. Template meta-programming for haskell. In *Proceedings of the 2002 ACM SIGPLAN workshop on Haskell*, pages 1–16. ACM, 2002.

Eötvös Loránd University, Budapest, Hungary
E-mail address: {gobi,kto,kralikba}@elte.hu