Science of Computer Programming 111 (2015) 51–78

Contents lists available at ScienceDirect

Science of Computer Programming

www.elsevier.com/locate/scico

Shape analysis in a functional language by using regular languages [☆]

Manuel Montenegro [*], Ricardo Peña [*], Clara Segura

ARTICLE INFO

Article history:
Received 11 December 2013
Received in revised form 13 November 2014
Accepted 23 December 2014
Available online 5 January 2015

Keywords:
Functional languages
Abstract interpretation
Shape analysis
Points-to analysis
Regular languages

ABSTRACT

Shape analysis is concerned with the compile-time determination of the 'shape' the heap may take at runtime, meaning by this the pointer chains that may happen within, and between, the data structures built by the program. This includes detecting alias and sharing between the program variables.
Functional languages facilitate somehow this task due to the absence of variable updating. Even though, sharing and aliasing are still possible. We present an abstract interpretation-based analysis computing precise information about these relations. In fact, the analysis gives an information more precise than just the existence of sharing. It informs about the paths through which this sharing takes place. This information is critical in order to get a modular analysis and not to lose precision when calling an already analysed function.
The motivation for the analysis in our case is the need of knowing at compile time which variables are at risk of containing dangling pointers at runtime, in a language with explicit memory disposal primitives.
The main innovation with respect to the literature is the use of regular languages to specify the possible pointer paths from a variable to its descendants. This additional information makes the analysis much more precise while still being affordable in terms of efficiency. We have implemented it and give convincing examples of its precision.

1. Motivation

Shape analysis is concerned with statically determining the connections between program variables through pointers in the heap that may occur at runtime. As particular cases, it includes sharing and alias between variables. To know the shape of the heap for every possible program execution is undecidable in general, but the analysis computes an over-approximation of this shape. This means that it may include sharing relations that will never happen at runtime.

Much work has been done in imperative languages (see Section 7), specially for C. There, the sharing detection is aggravated by the fact that variables are mutable, and they may point to different places at different times. We have addressed the problem for a first-order functional language. This simplifies some of the difficulties since variables do not mutate. A consequence is that the inferred relations are immutable considering different parts of the program text. Another consequence is that the heap is never updated. It can only be increased with new data structures, or decreased by the garbage collector. But the latter cannot produce effects in its live part.

[☆] Work partially funded by the projects TIN2013-44742-C4-3-R (CAVI-ART), TIN2008-06622-C03-01/TIN (STAMP), and S2009/TIC-1465 (PROMETIDOS).
[*] Corresponding authors.
E-mail addresses: montenegro@fdi.ucm.es (M. Montenegro), ricardo@sip.ucm.es (R. Peña), csegura@sip.ucm.es (C. Segura).

http://dx.doi.org/10.1016/j.scico.2014.12.006
0167-6423/© 2014 Elsevier B.V. All rights reserved.

```
unshuffle []!     = ([],[])
unshuffle (x:xs)! = (x:ys2, ys1)
                    where (ys1,ys2) = unshuffle xs
merge []!     ys!  = ys
merge (x:xs)! []!  = x:xs
merge (x:xs)! (y:ys)! | x <= y    = x : merge xs     (y:ys)
                      | otherwise = y : merge (x:xs) ys
msort []!  = []
msort [x]! = [x]
msort xs!  = merge (msort xs1) (msort xs2)
             where (xs1, xs2) = unshuffle xs
```

Fig. 1. *mergesort* algorithm in constant heap space.

Our analysis puts the emphasis on three properties: (1) modularity; (2) precision; and (3) efficiency. For the sake of scalability, it is important for the analysis to be modular. The results obtained for a function should summarise the shape information so that the user functions should be able to compute all the sharing produced when calling it. Looked at from outside, and given that the language is functional, a function may only create sharing between its result and its arguments, or between the results themselves, but it can never create new sharing between the arguments. The internal variables become dead after the call, so the result of analysing a function only contains its input–output sharing behaviour. Differently from previous works, we compute the *paths* through which this sharing may occur in a precise way. This information is used to propagate to the caller the sharing created by a call. In this way, large programs need not to be analysed globally, but just a function at a time.

The motivation for our analysis is a safety type system we have developed for a functional language, called *Safe*, with explicit memory disposal [1]. This feature may create dangling pointers at runtime. The language also provides automatically allocated and deallocated heap regions, instead of having a runtime garbage collector. This feature can never create dangling pointers, so it plays no role in the current work and we will not mention it anymore.

The explicit memory disposal is achieved by means of a destructive pattern matching, denoted with symbol !, or a **case**! expression. By applying any of them, we can reuse the cell corresponding to the parameter or variable affected by it. This feature may be used in our language to implement data structures whose updating needs no additional heap space or constant heap space functions over data structures, see [2] for examples. As an example, in Fig. 1 we show an implementation of the mergesort algorithm for sorting a list in constant heap space. Each cell of the original list is disposed by *unshuffle*; lists xs_1 and xs_2 are disposed by the recursive calls to *msort*; and finally the results of the recursive calls are disposed by *merge*.

The type inference algorithm [3] assigns the program variables safety marks: `d` for disposed, `s` for safe and `r` for in-danger variables. Each time a variable is marked as disposed, all those variables that may point to cells belonging to its recursive substructure are marked as in-danger, because they can potentially contain dangling pointers. The type rules propagate the marks and control how the variables are used. For instance, in a **let** expression in-danger and already disposed variables in the let-bound expression cannot be mentioned in the main expression. We have proved that passing successfully the type inference phase gives total guarantee that there will not be dangling pointers.

So, for typechecking a function, it is critical to know at compile time which variables may point to the disposed data structures, and for this we need a sharing analysis. Our prior prototype shape analysis done in [4] was correct but imprecise at some points. In particular, the type system rejected the mergesort algorithm shown in Fig. 1, due to the imprecision of the sharing analysis results. As we will see in more detail in the following section, the reason for this is that it does not suffice knowing that two variables share a common descendant, but we should more precisely know through which paths this sharing occurs, and that is why in this work we introduce regular languages representing paths in the heap.

We believe that the sharing analysis presented here could be equally useful for other purposes, since it provides precise information about the heap shape. Note that some shapes, such as cyclic or doubly linked lists, cannot be created by a functional language, so they are out of the scope of our analysis. But, in some cases, the analysis is capable of asserting that a given structure is a tree, i.e. it does not have internal sharing (see Section 6 for an example).

The main contribution of this paper with respect to [4] is the incorporation of regular languages to our abstract domain. Each word of the language defines a pointer path within a data structure. Having regular languages introduces additional problems such as how to combine them during the analysis, how to compare them, and specially how to guarantee that a fixpoint will be reached after a finite number of iterations. We show that we have increased the precision of our prior analysis, and that the new problems can be tackled with a reasonable efficiency.

The plan of the paper is as follows: Section 2 provides a mild introduction to the analysis via a small example. Then, Sections 3, 4 and 5 contain all the technical material about the abstract domain, abstract interpretation rules, correctness, implementation, widening, and cost of the operations done on regular languages. Section 6 gives abundant examples of the sharing results obtained by our analysis and their corresponding running times. Finally, Section 7 concludes and discuss some related work.

This paper is an extended version of [5]. The additional material here mainly concerns Sections 4, 5, and 6. In the first one, a much detailed proof of correctness is given. In the second one, we compare two alternative implementations (instead of the single one presented in [5]) in which regular languages are represented either by nondeterministic automata or by