Computer Languages, Systems & Structures 000 (2017) 1–27

Contents lists available at ScienceDirect

Computer Languages, Systems & Structures

journal homepage: www.elsevier.com/locate/cl

Call Arity

Joachim Breitner

University of Pennsylvania, 3330 Walnut Street Philadelphia, PA 19104, USA

ARTICLE INFO

Article history:
Received 24 June 2016
Revised 6 January 2017
Accepted 14 March 2017
Available online xxx

ABSTRACT

Higher order combinators in functional programming languages can lead to code that would be considerably more efficient if some functions' definitions were eta-expanded. Previous analyses were not always precise enough to allow that. In particular, this has prevented `foldl` from efficiently taking part in list fusion.

Call Arity is an analysis that eta-expands functions according to how they are used. By virtue of using a new cardinality analysis based on the notion of co-call graphs, it is sufficiently precise even in the presence of recursion, and its inclusion in the Haskell compiler GHC now enables the fusion of `foldl`-based combinators.

1. Introduction

After more than two decades of development of Haskell compilers, one has become slightly spoiled by the quality and power of optimizations performed by the compiler. For example, list fusion allows us to write concise and easy to understand code using general purpose combinators and list comprehensions and still get the efficiency of a tight loop that avoids allocating the intermediate lists.

Unfortunately, not all list-processing functions took part in list fusion. In particular, left folds like `foldl`, `foldl'`, `length` and derived functions like `sum` did not fuse, and the expression `sum (filter f [42..2016])` still allocated and traversed one list.

The issue is that in order to take part in list fusion, functions need to be expressed as right folds. In the case of `foldl`, this requires higher-order parameters as in

```
foldl k z xs = foldr (λv fn z → fn (k z v)) id xs z.
```

The resulting fused code would be passing around and calling dynamically allocated function closures on the heap, which is rather inefficient.

Gill noted that eta-expansion based on an arity analysis would help here [1]. Previous arity analyses, however, are not precise enough to be able to clean up the code that results from a fusing `foldl`.

What makes eta-expanding function and thunks hard? Too much eta-expansion can destroy sharing, as we explain in greater detail in Section 2.2. A conservative compiler must not suddenly duplicate computational work of unknown extent, and therefore functions may only be eta-expanded as far as it is safe.

Consider the slightly contrived example in Fig. 1: Our goal is to eta-expand the definition of `tA`, which is a thunk returning a function. For that to be safe, we need to ensure that it is always called with one argument, which is not obvious: syntactically, the only use of `tA` is in `goB`, and there it occurs without an argument. But we see that `goB` is initially called

E-mail addresses: joachim@cis.upenn.edu, mail@joachim-breitner.de

http://dx.doi.org/10.1016/j.cl.2017.03.001
1477-8424/

```
let tA = if f a then ... else ...
in let goA x = if f (tB + x) then goA (x+1)
                             else x
       tB    = let goB y = if f y then goB (goA y)
                                  else tA
               in goB 0 1
   in goA (goA 1)
```

Fig. 1. Is it safe to eta-expand tA?.

with two arguments. Furthermore, if two arguments are passed to goB, this function calls itself with two arguments as well. Therefore, we know that goB calls tA always with one argument – done.

But tA is a thunk – i.e. not in head normal form – so even if there are many calls to tA, due to sharing, the predicate f a is only evaluated once. If we were to eta-expand tA, that would no longer be a thunk and that possibly expensive predicate would be evaluated for every call to tA! So we are only allowed to eta-expand tA if we know that it is called at most once. This is tricky: tA is called from a recursive function goB, which itself is called from the mutual recursion consisting of goA and tB, and that recursion is started multiple times!

Nevertheless we can know that tA is evaluated at most once: tB is also a thunk, so although it will be called multiple times by the outer recursion, its right-hand side is only evaluated once. Furthermore, the recursion involving goB is started once and stops when the call to tA happens. Together, this implies that we are allowed to eta-expand tA without losing any work.

We have developed an analysis, dubbed *Call Arity*, that is capable of all of this reasoning and correctly detects that tA can be eta-expanded. It is a combination of a standard forward arity analysis [1,2] with a novel cardinality analysis based on *co-call graphs*. The latter determines for an expression and two variables whether a single evaluation of the expression can possibly call both variables and – as a special case – which variables it calls at most once. We found that this is just the right amount of information to handle tricky cases as in Fig. 1.

In particular, we make the following contributions:

1. We motivate why a precise arity analysis needs a sophisticated cardinality analysis (Section 3).
2. We present the *Call Arity* arity analysis together with the *co-call graph* based cardinality analysis (Section 4).
3. Call Arity is implemented in GHC, and enabled by default since version 7.10.1. We describe a few aspects of its implementation (Section 6).
4. With this analysis, the previously open problem of how to let foldl take part in list fusion while still obtaining well-performing code (Section 2.1) is solved.
5. Our performance measurements (Section 5) show great improvements in some cases, a good overall effect and no increase in allocations due to the addition of Call Arity.

The paper at hand is a revised, corrected and extended version of the conference publication from 2014 [3]. Parts of it forms a chapter of the author's thesis [4], which gives a more formal account of Call Arity, including a formal, machine-checked proof that enabling Call Arity is safe, which is here summarized in Section 4.3.

2. Background

In the interest of a self-contained presentation, this section contains a primer on list fusion, including its implementation in GHC based on rewrite rules. Furthermore, we explain how GHC's runtime implements function calls and why saturated function calls are generally more efficient.

2.1. List fusion

Code like sum (filter f [42..2016]) is a good example for the style of programming that functional programming languages excel at: More complex algorithms are created by composing small, single-purpose building blocks (Fig. 2) in a straightforward way. This style is pleasant to program in and reason about. It is less attractive, though, from a performance point of view: This code, compiled without optimizations, will allocate a list data structure on the heap, traverse it to create a second list of those elements for which the predicate f hold, and then traverse that list to sum the elements. Finally, all the now unused list data structures will be released by the garbage collector.

Fortunately, there exists a well-known program transformation called *list fusion* [5], which can turn the code that we want to write into the code that we want to run, where no lists are allocated.

The central idea is that list-producing code does not allocate the list constructors (:) and [] (which would normally be passed to and deconstructed by the consumer) but rather uses functions provided by the consumer instead, passing the head and the (already processed) tail directly. This requires the producer to abstract over the list constructors, and the consumer to provide these functions at the right type:

Please cite this article as: J. Breitner, Call Arity, Computer Languages, Systems & Structures (2017), http://dx.doi.org/10.1016/j.cl.2017.03.001