Call Arity

Joachim Breitner
Karlsruhe Institute of Technology
breitner@kit.edu

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, but the existing analyses are not always precise enough to allow that. In particular, this has prevented the implementation of foldl via foldr, which would allow foldl to take part in list fusion.

Call Arity is an analysis that eta-expands functions based on their uses, instead of their definitions, and is very precise in the presence of recursion. Its inclusion in GHC now allows foldl-based combinators to take part in list fusion.

1. Introduction

After more than two decades of development of Haskell compilers, one has become slightly spoiled by the quality and power of optimizations applied by the compiler. For example, list fusion allows us to write concise and easy to understand code using 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 take part in list fusion. In particular, left folds like foldl, foldl′, length and derived functions like sum are not fusing, and the expression sum (filter f [42 . . 2014]) still allocates and traverses one list.

The issue is that in order to take part in list fusion, these need to be expressed as right folds, which requires higher-order parameters like in

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

The resulting fused code would be allocating and calling function closures on the heap and prevent good code from being generated ([7]).

Andrew Gill already noted that eta-expansion based on an arity analysis would help here [5]. Existing arity analyses however are not precise enough to allow for a fusing foldl.

Why is this so hard? Consider the slightly contrived example in Figure 1: Our goal is to eta-expand the definition of tA. For that, 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 with two arguments, and under

```
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)
```

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

that assumption calls itself with two arguments as well, and therefore always calls tA with one argument – done.

But tA is a thunk – an expression not in head normal form – and even if there are many calls to tA, the call to f a is only evaluated once. If we were to eta-expand tA we would be duplicating that possibly expensive work! So we are only allowed to eta-expand it if we know that it is called at most once. This is tricky: It is called from a recursive function goB, which again is called from the mutual recursion consisting of goA and tB, and that recursion is started multiple times!

Nevertheless we know that tA is evaluated at most once: tB is a thunk, so although it will be evaluated 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 this reasoning and correctly detects that tA can be eta-expanded. It is a combination of a standard forward call arity analysis ([5], [13]) with a novel *co-call analysis*. The latter determines for an expression and two variables whether one 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 like Figure 1.

In particular, we make the following contributions:

- We present a new forward arity analysis (Section 3) based on a co-call analysis.
- The analysis is conservative: No sharing is lost as the result of the eta-expansion.
- The analysis is more precise than existing analyses: It can detect that a variable is called once even in the presence of recursion. We explain why co-call analysis is required for this level of precision. (Section 2).
- The analysis is free of heuristics: No arbitrary choices need to be made by an implementation of the analysis.
- We have implemented the analysis, it is included it in the development version of GHC (Section 4).
- It is now beneficial to implement foldl as a good consumer for list fusion (Section 5.1), as demonstrated by performance measurements (Section 5.2).

2. The need for co-call analysis

The main contribution of this paper is the discovery of the co-call analysis and its importance for arity analysis. This section motivates the analysis based on a sequence of ever more complicated arity analysis puzzles.

2.1 A syntactical analysis

The simplest such exercise is the following code:

```
let f x = ...
in f 1 2 + f 3 4.
```

Are we allowed to eta-expand f by another argument? Yes! How would we find out about it? We'd analyze each expression of the syntax tree and collect this information:

> For each free variable, what is a lower bound on the number of arguments passed to it?

This will tell us that f is always called with two arguments, so we eta-expand it.

2.2 Incoming arity

Here is a slightly more difficult puzzle:

```
let f x = ...
    g y = f (y + 1)
in g 1 2 + g 3 4.
```

Are we still allowed to eta-expand f? The previous syntactic approach fails, as the right-hand side of g mentions f with only one argument. However, g itself can be eta-expanded, and once that is done we would see that g's right hand side is called with one argument more. We could run the previous analysis, simplify the code, and run the analysis once more, but we can do better by asking, for every expression:

> If this expression is called with n arguments, for each free variable, what is a lower bound on the number of arguments passed to it?

The body of the **let** will report to call g with two arguments. This allows us to analyze the right-hand side of g (which consumes only one argument) with an *incoming arity* of 1, and thus report that f is always called with two arguments.

For recursive functions this is more powerful than just running the simpler variant multiple times:

```
let f x = ...
    g y = if y > 10 then f y else g (y + 1)
in g 1 2 + g 3 4.
```

A purely syntactical approach will never be able to eta-expand g or f. But by assuming an incoming arity we can handle the recursive case: The body of the **let** reports that g is called with two arguments. We assume that is true for all calls to g. Next we analyze the right-hand side of g and will learn – under our assumption – that it calls g with two arguments, too, so our assumption was justified and we can proceed.

The assumption can also be refuted when analyzing the recursive function:

```
let f x = ...
    g y = if y > 10 then f y else foo (g (y + 1))
in g 1 2 + g 3 4.
```

The body still reports that it calls g with two arguments, but – even under that assumption – the right-hand side of g calls g with only one argument. So we have to re-analyze g with one argument, which in turn calls f with one argument and no eta-expansion is possible here.

This corresponds to the analysis outlined in [5].

2.3 Called-once information

So far we have only eta-expanded functions; for these the last analysis is sufficient. But there is also the case of *thunks*: If the expression bound to a variable x is not in head-normal form, i.e. the outermost syntactic construct is a function call, case expression or let-binding, but not a lambda, then the work done by this expression is shared between multiple calls to x.

If we were to eta-expand the expression, though, the expensive operation is hidden behind a lambda and will therefore be evaluated for every call to x. Therefore it is crucial that thunks are only eta-expanded if they are going to be called at most once. So we need to distinguish the situation

```
let t = foo x
in if x then t 1 else t 2
```

where t is called at most once and eta-expansion is allowed from

```
let t = foo x
in t 1 + t 2
```

where t is called multiple times and must not be eta-expanded.

An analysis that could help us here would be answering this question:

> If this expression is called once with n arguments, for each free variable, what is a lower bound on the number of arguments passed to it, and are we calling it at most once?

In the first example, both branches of the **if** would report to call t only once (with one argument), so the whole body of the **let** calls t only once and we can eta-expand t. In the second example the two subexpressions t 1 and t 2 are both going to be evaluated. Combined they call t twice and we cannot eta-expand t.

2.4 Mutually exclusive calls

What can we say in the case of a thunk that is called from a recursion:

```
let t = foo x
in let g y = if y > 10 then t else g (y + 1)
   in g 1 2.
```

Clearly t is called at most once, but the current state of the analysis does not see that: The right-hand side of g reports to call t and g at most once. But we would get the same result from

```
let t = foo x
in let g y = if y > 10 then id else g (y + t)
   in g 1 2
```

as well, where t is called many times!

How can we extend our analysis to distinguish these two cases? The crucial difference is that in the first code, g calls *either* t *or* g, while the second one calls both of them together. So we would like to know, for each expression:

> If this expression is called once with n arguments, for each free variable, what is a lower bound on the number of arguments passed to it? Additionally, what set of variables is called mutually exclusively and at most once?

In the first example, the right-hand side would report to call $\{t, g\}$ mutually exclusively and this allows us to see that the call to t lies not on the recursive path, so there will be at most one call to t in every run of the recursion. We also need the information that the body of the **let** (which reports $\{g\}$) and the right-hand side of g both call g at most once; if the

recursion were started multiple times, or were not linear, then we'd get many calls to t as well.

2.5 Co-Call Analysis

The final puzzle in this sequence shows the shortcomings of the previous iteration and the strength of the co-call analysis. Consider the code

```
let t1 = foo x
in let g x = if x > 10
             then t1
             else let t2 = bar x
                  in let h y = if y > 10
                               then g (t2 y)
                               else h (y + 1)
                     in h 1 x
   in g 1 2.
```

Both recursions are well-behaved: They are entered once and each recursive function calls either itself once or calls the thunk t1 resp. t2 once. So we would like to see both t1 and t2 eta-expanded. Unfortunately, with the analysis above we can only get one of them.

The problematic subexpression is g (t2 y): We need to know that g is called at most once *and* that t2 is called at most once. But we cannot return $\{g, t2\}$ as that is a lie – they are not mutually exclusive – and we have to arbitrarily return $\{g\}$ or $\{t2\}$.

To avoid this dilemma we extend the analysis one last time, allowing it to preserve all valuable information. We now ask, for each expression:

> If this expression is called once with n arguments, for each free variable, what is a lower bound on the number of arguments passed to it, and for each pair of free variables, can both be called during the same execution of the expression?

The latter tells us, as a special case, whether one variable may be called multiple times.

So for the problematic expression g (t2 y) we would find out that g might be called together with t2, but neither of them is called twice. For the right-hand side of h the analysis would tell us that either h is called at most once and on its own, or g and t2 are called together, but each at most once. The whole inner **let** therefore calls t2 and g at most once, and we get to eta-expand t2 as well as learn that the outer recursion is well-behaved.

3. The Call Arity analysis

Thus having motivated needs for the Call Arity analysis we devote this section to a precise and formal description of it. We use the simplified lambda calculus given in Figure 2. Although the implementation works on GHC's typed intermediate language Core, the types are not relevant for the analysis itself, so we consider an untyped calculus. Also, data type constructors and pattern matching play no role here and we use $e\,?\,e_1 : e_2$ as a simple representative for more complicated case constructs. We also assume that all bound variables are distinct and do not hold us up with naming issues.

Like Core we distinguish between the non-recursive **let** and the (possibly mutually recursive) **letrec**, and we assume that some earlier compiler pass has identified the strongly connected components of the bindings' dependency graph and transformed the code so that all **letrec**-bound groups are indeed mutually recursive.

v, x, y, z: Var	variables
e: Expr	expressions
$e ::= x$	variable
$\mid e_1\ e_2$	application
$\mid \lambda x.\, e$	lambda abstraction
$\mid e\,?\,e_1 : e_2$	case analysis
$\mid$ **let** $x = e_1$ **in** e_2	non-recursive binding
$\mid$ **letrec** $\overline{x_i = e_i}$ **in** e	mutually recursive bindings

Figure 2. A simple lambda calculus

3.1 The specification

The goal of this analysis is to find out the *call arity* of every variable v, written n_v. The specification for the call arity is: The compiler can replace the binding **let** v = e by **let** v = $\lambda x_1 \ldots x_{n_v}.\, e\ x_1 \ldots x_{n_v}$ without losing any sharing.

The analysis traverses the syntax tree in a bottom-up manner. Each expression e is analyzed under the assumption of an *incoming arity* n – , which is the number of arguments the expression is currently being applied to – in order to determine with at least how many arguments e calls its free variables, and which free variables can be called together. Separating these two aspects into two functions for this formal description, we have

A_n: Expr $\to$ (Var $\rightharpoonup \mathbb{N}$)	arity analysis
C_n: Expr $\to$ Graph(Var)	co-call analysis

where $\rightharpoonup$ denotes a partial map and Graph(Var) is the type of undirected graphs (with self-edges) over the set of variables.

The specifications for A_n and C_n are

- If $A_n(e)[x] = m$, then every call from e (applied to n arguments) to x passes at least m arguments.
- If x_1—$x_2 \notin C_n(e)$, then no execution of e (applied to n arguments) will call both x_1 and x_2. In particular, if x—$x \notin C_n(e)$, then x will be called at most once.

We can define a partial order on the analyses' results that expresses the notion of precision: If x is correct and $x \sqsubseteq y$, then y is also correct, but possibly less precise. In particular for A^1, A^2: (Var $\rightharpoonup \mathbb{N}$) we have

$$A^1 \sqsubseteq A^2 \iff \forall x \in \mathrm{dom}(A^1).\, A^1[x] \geq A^2[x]$$

(note the contravariance), because it is always safe to assume that x is called with fewer arguments. For C^1, C^2: Graph(Var), we have

$$C^1 \sqsubseteq C^2 \iff C^1 \subseteq C^2,$$

because it is always safe to assume that two variables are called both, or to assume that one variable is called multiple times.

Thus the top of the lattice, i.e. the always-correct and least useful result, maps every variable to 0 (making no statements about their number of arguments), and has the complete graph on all variables as the co-call graph (allowing everything to be called with everything else).

The bottom of the lattice, i.e. the best information, is the empty map and empty graph, and is correct for closed values like $\lambda y.\, y$.

$$
\begin{aligned}
A_n(x) &= \{x \mapsto n\} & C_n(x) &= \{\} \\
A_n(e_1\ e_2) &= A_{n+1}(e_1) \sqcup A_0(e_2) & C_n(e_1\ e_2) &= C_{n+1}(e_1) \cup C_0(e_2) \cup \mathrm{fv}(e_1) \times \mathrm{fv}(e_2) \\
A_{n+1}(\lambda x.\, e) &= A_n(e) & C_{n+1}(\lambda x.\, e) &= C_n(e) \\
A_0(\lambda x.\, e) &= A_0(e) & C_0(\lambda x.\, e) &= (\mathrm{fv}(e))^2 \\
A_n(e\,?\,e_1 : e_2) &= A_0(e) \sqcup A_n(e_1) \sqcup A_n(e_2) & C_n(e\,?\,e_1 : e_2) &= C_0(e) \cup C_n(e_1) \cup C_n(e_2) \cup \mathrm{fv}(e) \times (\mathrm{fv}(e_1) \cup \mathrm{fv}(e_2))
\end{aligned}
$$

Figure 3. The Call Arity analysis equations

3.2 The equations

From the specification we can, for every syntactical construct, derive equations that determine the analysis. In these equations, given in Figures 3 to 5, we use the operators

$$
\begin{aligned}
&\mathrm{fv} \colon \mathrm{Expr} \to \mathcal{P}(\mathrm{Var}) \\
&\sqcup \colon (\mathrm{Var} \rightharpoonup \mathbb{N}) \to (\mathrm{Var} \rightharpoonup \mathbb{N}) \to (\mathrm{Var} \rightharpoonup \mathbb{N}) \\
&\times \colon \mathcal{P}(\mathrm{Var}) \to \mathcal{P}(\mathrm{Var}) \to \mathrm{Graph}(\mathrm{Var}) \\
&{}^2 \colon \mathcal{P}(\mathrm{Var}) \to \mathrm{Graph}(\mathrm{Var})
\end{aligned}
$$

where $\mathcal{P}(S)$ is the type of subsets of S; $\mathrm{fv}(e)$ is the set of free variables of e; $f \sqcup g$ is the union of two partial maps, taking the minimum where both are defined; $S_1 \times S_2 = \{x\text{—}y \mid x \in S_1, y \in S_2\}$ is the complete bipartite graph and $S^2 = S \times S = \{x\text{—}y \mid x \in S, y \in S\}$ is the complete graph on S.

Case 1: Variables

Evaluating a variable with an incoming arity of n yields a call to that variable with n arguments, so the arity analysis returns a singleton map. Because we are interested in the effect of *one* call to the expression, we return x as called at-most once, i.e. the empty graph.

Case 2: Application

Here, the incoming arity is modified: If $e_1\ e_2$ is being called with n arguments, e_1 is passed one more argument. On the other hand we do not know how many arguments e_2 is called with – this analysis is not higher order – so we analyze it with an incoming arity of 0.

The co-call analysis reports all possible co-calls from both e_1 and e_2. Additionally, it reports that everything that may be called by e_1 can be called together with everything called by e_2.

This may be an over-approximation. Consider the expression $e = (e_0\,?\,x_1 : x_2)(e_0\,?\,x_3 : e_4)$. The analysis will return the graph

$$C_0(e) = \begin{matrix} x_1 & \text{—} & x_3 \\ & \times & \\ x_2 & \text{—} & x_4 \end{matrix}$$

although x_1 will only be called together with x_3 and x_2 with x_4, as the conditional e_0 will choose the same branch in both cases.

Case 3: Lambda abstraction

For lambda abstraction, we have to distinguish two cases. The good case is if the incoming arity is nonzero, i.e. we want to know the behavior of the expression when applied once to some arguments. In that case, we know that the body is evaluated once, and applied to one argument less, and the co-call information from the body can be used directly.

If the incoming arity is zero then we have to assume that the lambda abstraction is used as-is, for example as a parameter to a higher-order function, or stored in a data type. In particular, it is possible that it is called multiple times. So while the incoming arity on the body of the **let** stays zero (which is always correct), we cannot obtain any useful co-call results and have to assume that every variable mentioned in e is called with every other.

Example The expression $e = \lambda x.\, (x_0\,?\,x_1 : x_2)$ will, when analyzed with an incoming arity of 1 or more, yield

$$C_1(e) = x_0 \text{—} \begin{matrix} x_1 \\ x_2 \end{matrix},$$

while the same expression, analyzed with arity 0, yields the complete co-call graph

$$C_0(e) = \circlearrowleft x_0 \text{—} \begin{matrix} x_1 \circlearrowleft \\ x_2 \circlearrowleft \end{matrix}.$$

Case 4: Case analysis

The arity analysis of a case expression is straight forward: The incoming arity is fed into each of the alternatives, while the scrutinee is analyzed with an incoming arity of zero; the results are combined using $\sqcup$.

The co-call analysis proceeds likewise, but adds extra co-call edges, connecting everything that may be called by the scrutinee with everything that may be called in the alternatives. This is, as usual when analyzing a case expression, analogous to what happens when analyzing an application.

Case 5: Non-recursive let

This case is slightly more complicated than the previous, so we describe it in multiple equations in Figure 4.

We analyze the body of the let-expression first, using the incoming arity of the whole expression. Based on that we determine our main analysis result, the call arity of the variable. There are two cases:

1. If the right-hand side expression e_1 is a thunk and the body of the **let** may possibly call it twice, i.e. there is a self-loop in the co-call graph, then there is a risk of losing work when eta-expanding e_1, so we do not do that.
2. Otherwise, the call arity is the number of arguments that x is called with in e_2.

Depending on this result we need to adjust the co-call information obtained from e_1. Again, there are two cases:

1. We can use the co-call graph from e_1 if e_1 is evaluated at most once. This is obviously the case if x is called at most once in the first place. It is also the case if e_1 is (and stays!) a thunk, because its result will be shared and further calls to x can be ignored here.

$$n_x = \begin{cases} 0 & \text{if } x\text{—}x \in C_n(e_2) \text{ and } e_1 \text{ not in HNF} \\ A_n(e_2)[x_i] & \text{otherwise} \end{cases}$$

$$C_{\text{rhs}} = \begin{cases} C_{n_x}(e_1) & \text{if } x\text{—}x \notin C_n(e_2) \text{ or } n_x = 0 \\ \text{fv}(e_1)^2 & \text{otherwise} \end{cases}$$

$$E = \text{fv}(e_1) \times \{v \mid v\text{—}x \in C_n(e_2)\}$$

$$A = A_{n_x}(e_1) \sqcup A_n(e_2)$$

$$C = C_{\text{rhs}} \cup C_n(e_2) \cup E$$

$$A_n(\textbf{let } x = e_1 \textbf{ in } e_2) = A \qquad C_n(\textbf{let } x = e_1 \textbf{ in } e_2) = C$$

Figure 4. Equations for **let** $x = e_1$ **in** e_2

2. If e_1 may be evaluated multiple times we cannot get useful co-call information and therefore return the complete graph on everything that is possibly called.

Finally we combine the results from the body and the right-hand side, and add the appropriate extra co-call edges. We can be more precise than in the application case because we can exclude variables that are not called together with x from the complete bipartite graph.

Note that we do not clutter the presentation here with removing the local variable from the final analysis results. The implementation removes x from A and C before returning them.

Example Consider the expression

$$e = \textbf{let } v = (x \,?\, (\lambda y.\, x_2) : x_3) \textbf{ in } \lambda z.\, (x_1 \,?\, x_2 : v\ y)$$

with an incoming arity of 1. The co-call graph of the body is

$$C_1(\lambda z.\, (x_1 \,?\, x_2 : v\ y)) = x_1 \text{—} x_2,\ x_1 \text{—} v,\ x_2 \text{—} y$$

and $A_1(\lambda z.\, (x_1 \,?\, x_2 : v\ y))[v] = 1$. The right-hand side is a thunk, so we must be careful when eta-expanding it. But there is no self-loop at v in the graph, so v is called at most once. The call-arity of v is thus $n_v = 1$ and we obtain

$$C_1(x \,?\, (\lambda y.\, x_2) : x_3) = x \text{—} x_2,\ x \text{—} x_3$$

The additional edges E connect all free variables of the right-hand side ($\{x, x_2, x_3\}$) with everything called together with v from the body ($\{x_1, y\}$) and the overall result (skipping the now out-of-scope v) is

$$C_1(e) = x \text{—} x_2,\ x \text{—} x_3,\ x_1 \text{—} x_2,\ x_1 \text{—} x_3,\ x_1 \text{—} y$$

Note that although x_2 occurs in both the body and the right-hand side, there is no self-loop at x_2: The analysis has detected that x_2 is called at most once.

The results are very different if we analyze e with an incoming arity of 0. The body is a lambda abstraction, so may be called many times, and we have

$$C_1(\lambda z.\, (x_1 \,?\, x_2 : v\ y)) = x_1 \text{—} x_2,\ x_1 \text{—} y,\ x_1 \text{—} v,\ x_2 \text{—} y,\ x_2 \text{—} v,\ y \text{—} v$$

This time there is a self-loop at v, and we need to set $n_v = 0$ to be safe. This also means that v stays a thunk and we still get some useful information from the right-hand-side:

$$A = A_n(e) \sqcup \bigsqcup_i A_{n_{x_i}}(e_i)$$

$$C = C_n(e) \cup \bigcup_i C^i \cup \bigcup_i E^i$$

$$n_{x_i} = \begin{cases} 0 & \text{if } e_i \text{ not in HNF} \\ A[x_i] & \text{otherwise} \end{cases}$$

$$C^i = \begin{cases} C_{n_{x_i}}(e_i) & \text{if } x_i\text{—}x_i \notin C \text{ or } n_{x_i} = 0 \\ \text{fv}(e_i)^2 & \text{otherwise} \end{cases}$$

$$E^i = \begin{cases} \text{fv}(e_i) \times N(C_n(e) \cup \bigcup_j C^j) & \text{if } n_{x_i} \neq 0 \\ \text{fv}(e_i) \times N(C_n(e) \cup \bigcup_{j \neq i} C^j) & \text{if } n_{x_i} = 0 \end{cases}$$

$$N(G) = \{v \mid v\text{—}x_i \in G, i = 1 \ldots\}$$

$$A_n(\textbf{letrec } \overline{x_i = e_i} \textbf{ in } e) = A \quad C_n(\textbf{letrec } \overline{x_i = e_i} \textbf{ in } e) = C$$

Figure 5. Equations for **letrec** $\overline{x_i = e_i}$ **in** e

$$C_0(x \,?\, (\lambda y.\, x_2) : x_3) = x \text{—} x_2,\ x \text{—} x_3$$

Due to the lower incoming arity we can no longer rule out that x_2 is called multiple times, as it is hidden inside a lambda abstraction. The final graph now becomes quite large, because everything in the body is potentially called together with v:

$$C_0(e) = x \text{—} x_2,\ x \text{—} x_1,\ x \text{—} x_3,\ x \text{—} y,\ x_2 \text{—} x_1,\ x_2 \text{—} x_3,\ x_2 \text{—} y,\ x_1 \text{—} x_3,\ x_1 \text{—} y,\ x_3 \text{—} y$$

This is almost the complete graph, but it is still possible to derive that x and x_3 are called at most once.

Case 6: Recursive let

The final case is the most complicated. It is also the reason why the figures are labeled "Equations" and not "Definitions": They are also mutually recursive and it is the task of the implementation to find a suitable solution strategy (see Section 4.2).

The complication arises from the fact that the result of the analysis affects its parameters: If the right-hand side of a variable calls itself with a lower arity than the body, we need to use the lower arity as the call arity. Therefore, the result (A and C in the equation) is used to determine the basis for the call-arity and co-call information of the variables.

Thunks aside, we can think of one recursive binding **letrec** $x = e_1$ **in** e_2 as an arbitrarily large number of nested non-recursive bindings

$\ldots$
let $x = e_1$ **in**
$\ldots$
let $x = e_1$ **in** e_2.

The co-call information C can be thought of the co-call information of this expression, and this is how $x_i\text{—}x_i \notin C$ has to interpreted: Not that there are no multiple calls to x_i in the whole recursion (there probably are, given that it is a recursion), but rather that when doing the infinite unrolling of the recursion, there is at most one call to x_i from outside the scope of the outermost non-recursive **let**.

This analogy is flawed for thunks, where multiple nested non-recursive bindings would have a different sharing behavior. Therefore we set $n_{x_i} = 0$ for all thunks; this preserves the sharing.

The formulas for the additional co-calls E^i are a bit more complicated than in the non-recursive case, and differ for thunks and non-thunks. Consider one execution that reaches a call to x_i. What other variables might have been called on the way? If the call came directly from the body e, then we need to consider everything that is adjacent to x_i in $C_n(e)$. But it is also possible that the body has called some other x_j, $j \neq i$ and e_j then has called x_i – in that case, we need to take those variables adjacent to x_j in $C_n(e)$ and those adjacent to x_i in C^j.

In general, every call that can occur together with any recursive call in any of the expressions can occur together with whatever x_i does.

For a thunk we can get slightly better information: A non-thunk can be evaluated multiple times during the recursion, so its free variables can be called together with variables on e_i's own recursive path. A thunk, however, is evaluated at most once, even in a recursive group, so for the calculation of additional co-call edges it is sufficient to consider only the other right-hand sides (and the body of the **let**, of course).

Example Consider the expression

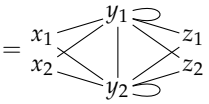
$$\begin{aligned} e = \textbf{letrec } & x_1 = \lambda y.\, (y_1\, \textbf{?}\, x_2\ y : z_1) \\ & x_2 = \lambda y.\, (y_2\, \textbf{?}\, x_1\ y : z_2) \\ \textbf{in } & \lambda y.\, x_1\ y\ y \end{aligned}$$

with an incoming arity of 1. It is an example for a nice tail-call recursion as it is commonly produced by list fusion: The body has one call into the recursive group, and each function in the group also calls at most one of them.

The solution to the equations in Figure 5 in this example is

$$\begin{aligned} C_1(e) &= \{\} \\ n_{x_1} = n_{x_2} &= 2 \\ C^1 = C_2(e_1) &= \{y_1\} \times \{x_2, z_1\} \\ C^2 = C_2(e_2) &= \{y_2\} \times \{x_1, z_2\} \\ E^1 &= \{y_1, x_2, z_1\} \times \{y_1, y_2\} \\ E^2 &= \{y_2, x_1, z_2\} \times \{y_1, y_2\} \end{aligned}$$

and the final result is

$C =$ 

where we see that at most one of z_1 and z_2 is called by the loop, and neither of them twice.

In contrast consider this recursion, which forks in x_2:

$$\begin{aligned} e = \textbf{letrec } & x_1 = \lambda y.\, (y_1\, \textbf{?}\, x_2\ y : z_1) \\ & x_2 = \lambda y.\, (y_2\, \textbf{?}\, x_1\ (x_1\ y\ y) : z_2) \\ \textbf{in } & \lambda y.\, x_1\ y\ y. \end{aligned}$$

We see that now z_1 and z_2 are possibly called together and multiple times. Indeed x_1—$x_1 \in C_2(e_2)$ causes $x_1 \in N(\ldots)$ in the equation for E^i, so especially x_1—$x_1 \in E^2 \subseteq C$. Therefore, $C^1 = \mathrm{fv}(e_1)^2$ and we also have x_2—$x_2 \in C$ and $C^2 = \mathrm{fv}(e_2)^2$. Eventually, we find that the result is the complete graph on all variables, i.e. $E = \{x_1, x_2, y_1, y_2, z_1, z_2\}^2$, and in particular z_1—$z_2 \in E$, as expected.

4. The implementation

The Call Arity analysis is implemented in GHC as a separate Core-to-Core pass, where Core is GHC's typed intermediate language based on System FC. This pass does not do the eta-expansion; it merely annotates let-bound variables with their call arity. A subsequent pass of GHC's simplifier then performs the expansion, using the same code as with the regular, definition-based arity analysis, and immediately applies optimizations made possible by the eta-expansion. This separation of concerns keeps the Call Arity implementation concise and close to the formalization presented here.

GHC Core obviously has more syntactical constructs than our toy lambda calculus, including literals, coercion values, casts, profiling annotations ("ticks"), type lambdas and type applications, but these are irrelevant for our purposes: For literals and coercion values we return the bottom of the lattice; the others are transparent to the analysis. In particular type arguments are not counted towards the arity here, which coincides with the meaning of arity as returned by GHC's regular arity analysis.

We want our analysis to make one pass over the syntax tree (up to the fixpointing for recursive bindings, Section 4.2). So instead of having two functions, one for the arity analysis and one for the co-call analysis, we have one function `callArityAnal` which returns a tuple `(UnVarGraph, VarEnv Arity)`, where the `UnVarGraph` is a data structure for undirected graphs on variable names (see Section 4.4) and `VarEnv` is a partial map from variable names to `Arity`, which is a synonym for `Int`.

The equations refer to $\mathrm{fv}(e)$, the set of free variables of an expression. In the implementation, we do not use GHC's corresponding function `exprFreeIds`, as this would require another traversal of the expression. Instead we use $\mathrm{dom}(A_n(e))$, which by construction happens to be the set of free variables of e, independent of n.

In the sequence of Core-to-Core passes, we inserted Call Arity and its eta-expanding simplifier pass between the simplifier's phase 0, as that is when all the rewrite rules have been active [9], and the strictness analyzer, because we want to have a chance to unbox any new arguments, such as the accumulator in a call to `sum`.

4.1 Interesting variables

The analysis as presented in the previous section would be too expensive if implemented as is. This can be observed when compiling GHC's `DynFlags` module, which defines a record type with 157 elements. The Core code for a setter of one of the fields is

```
setX42 x (DynFlags x1 ... x41 _ x43 ... x157)
    = (DynFlags x1 ... x41 x x43 ... x157).
```

For the body of the function, the analysis would report that 157 variables are called with (at least) 0 arguments, and that all of them are co-called with every other, a graph with 12246 edges. And none of this information is useful: The variables come from function parameters or pattern matches and there is no definition that we can possibly eta-expand!

Therefore we keep track of the set of *interesting variables*, and only return information about them. Interesting variables are all let-bound variables, not interesting are parameters or pattern match results.

4.2 Fixpointing

The equations in the previous section specify the analysis, but do not provide an algorithm: In the case of **letrec** (Figure 5), the equations are mutually recursive and the implementation has to employ a suitable strategy to find a solution.

We find the solution by iteratively approaching the fixpoint, using memorization of intermediate results.

1. Initially, we set $A = A_n(e)$ and $C = C_n(e)$.

2. For every variable $x_i \in \text{dom}\, A$ that has not been analyzed before, or has been analyzed before with different values for n_{x_i} or x_i—$x_i \in C$, we (re-)analyze it, remembering the parameters and memorize the result $A_{n_{x_i}}(e_i)$ and C^i.
3. If any variable has been (re)analyzed in this iteration, recalculate A and C and repeat from step 2.

This process will terminate, as shown by a simple standard argument. The variant that proves this consists of n_{x_i} and whether x_i—$x_i \in C$: The former starts at some natural number and decreases, the latter may start as not true, but once it is true, it stays true. Therefore, these parameters can change only a finite number of times, and we terminate once all of them are unchanged during one iteration. The monotonicity of the parameters follows from the monotonicity of the equations for A_n and C_n: We have that $n \geq n'$ implies $A_n(e) \sqsubseteq A_{n'}(e)$ and $C_n(e) \sqsubseteq C_{n'}(e)$, which is easily verified by case distinction.

4.3 Top-level values

GHC supports modular compilation. Therefore, for exported functions, we do not have the call sites available to analyze. Nevertheless we do want to be able to analyze and eta-expand non-exported top-level functions.

To solve this elegantly we treat a module

```
module (foo) where
bar = ...
foo = ...
```

like a sequence of let-bindings

```
let bar = ...
    foo = ...
in e
```

where e represents the external code for which we assume the worst: It calls all exported variables (foo here) with 0 arguments and the co-call graph is the complete graph. This prevents unwanted expansion of foo, but still allows us to eta-expand bar based on how it is called by foo.

4.4 The graph data structure

The data type UnVarGraph used in the implementation is specifically crafted for our purposes. The analysis often builds complete bipartite graphs and complete graphs between set of variables. A usual graph representation like adjacency lists would therefore be quadratic in size and too inefficient for our uses.

Therefore we store graphs symbolically, as multisets of complete bipartite and complete graphs. This allows for very quick creation and combination of graphs. The important query operation (the set of neighbors of a node) is done by traversing the generating subgraphs.

One disadvantage of this data structure is that it does not normalize the representation. In particular, the union of a graph with itself is twice as large. This had to be taken into account when implementing the fixpointing: It would be very inefficient to update C by unioning it with the new results in each iteration. Instead, C is recalculated from $C_n(e)$ and the – new or memorized – results from the bound expressions.

We experimented with simplifying the graph representation using identities like $S_1 \times S_2 \cup S_1^2 \cup S_2^2 = (S_1 \cup S_2)^2$, but it did not pay off, especially as deciding set equality can be expensive.

5. Discussion

5.1 Call Arity and list fusion

As hinted at in the introduction, Call Arity was devised mainly to allow for a fusing foldl, i.e. a definition of foldl in terms of foldr that takes part in list fusion while still producing good code. How exactly does Call Arity help here?

Consider the code

```
sum (filter f [42..2014])
```

Previously, only filter would fuse with the list comprehension, eliminating one intermediate list, but the call to sum, being a left-fold, would remain: Compiled with previous versions of GHC, this produces code roughly equivalent to

```
let go = λx. let r = if x == 2014 then [] else go (x + 1)
             in if f x then x : r else r
in foldl (+) 0 (go 42).
```

If we now change the definition of foldl to use foldr, as in

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

all lists are completely fused and we obtain the code

```
let go = λx. let r = if x == 2014 then id else go (x + 1)
             in if f x thenλa. r (a + x) else r
in go 42 0.
```

Without Call Arity, this is the final code, and as such quite inefficient: The recursive loop go has become a function that takes one argument, then allocates a function closure for r on the heap, and finally returns another heap-allocated function closure which will pop the next argument from the stack – not the fastest way to evaluate a simple program.

With Call Arity the compiler detects that go and r can both be eta-expanded with another argument, yielding the code

```
let go = λx a. let r = λa. if x == 2014 then a
                                        else go (x + 1) a
               in if f x then r (a + x) else r a
in go 42 0
```

where the number of arguments passed matches the number of lambdas that are manifest on the outer level. This avoids allocations of function closures and allows the runtime to do fast calls [7], or even tail-recursive jumps.

5.1.1 Limitations

A particularly tricky case is list fusion with generators with multiple (or non-linear) recursion. This arises when flattening a tree to a list. Consider the code

```
data Tree = Tip Int | Bin Tree Tree

toList :: Tree → [Int]
toList tree = build (toListFB tree)

toListFB root cons nil = go root nil
  where
    go (Tip x)  rest = cons x rest
    go (Bin l r) rest = go l (go r rest)
```

which is a good producer; for example filter f (toList t) is compiled to

```
let go = λt rest. case t of
      Tip x  → if f x then x : rest else rest
      Bin l r → go l (go r rest)
in go t [].
```

If we add a foldr implemented left-fold to the pipe, i.e. foldl (+) 0 (filter f (toList t)), the resulting code (before Call Arity) is

```
let go = λt fn. case t of
      Tip x  → if f x then λa. fn (x + a) else fn
      Bin l r → go l (go r fn)
in go t id 0.
```

	Allocs				Runtime			
Arity Analysis	✓	✓		✓	✓	✓		✓
Co-Call Analysis	✓	✓			✓	✓		
foldl via foldr		✓	✓	✓		✓	✓	✓
`anna`	-1.3%	**-1.4%**	+0.0%	+0.0%				
`bernouilli`	-0.0%	**-4.9%**	+3.7%	+3.7%				
`calendar`	-0.1%	**-0.2%**	-0.1%	-0.1%	+4.7%	**+0.8%**	+0.8%	+2.3%
`fft2`	-0.0%	**-79.0%**	-78.9%	-78.9%				
`gen_regexps`	0.0%	**-53.9%**	+33.8%	+33.8%	-1.2%	**-8.9%**	+223.6%	+224.8%
`hidden`	-0.3%	**-6.3%**	+1.2%	+1.2%	-3.3%	**-3.3%**	0.0%	0.0%
`integrate`	-0.0%	**-61.7%**	-61.7%	-61.7%	-6.0%	**-48.7%**	-48.7%	-48.7%
`minimax`	0.0%	**-15.6%**	+4.0%	+4.0%				
`rewrite`	-0.0%	**-0.0%**	-0.0%	-0.0%	+0.9%	**+6.1%**	+3.5%	+0.9%
`simple`	0.0%	**-9.4%**	+8.1%	+8.1%				
`x2n1`	-0.0%	**-77.4%**	+84.0%	+84.0%				
...and 89 more								
Min	-1.3%	**-79.0%**	-78.9%	-78.9%	-6.0%	**-48.7%**	-48.7%	-48.7%
Max	+0.0%	**+0.0%**	+84.0%	+84.0%	+4.7%	**+6.1%**	+223.6%	+224.8%
Geometric Mean	-0.0%	**-5.2%**	-1.5%	-1.5%	-0.2%	**-1.4%**	+1.0%	+1.2%

Table 1. Nofib results

Although go is always being called with three arguments, our analysis does not see this. For that it would have to detect that go calls its second parameter with one argument; as it is a forward analysis (in the nomenclature of [13]) it cannot do that.

Even if the arity analyzer were smart enough to eta-expand go, the code would not be much better; we'd still be passing a function closure to go.

The code that we would like to see is

```
let go = λt a. case t of
      Tip x  → if f x then a + x else a
      Bin l r → go l (go r a)
in go t 0
```

where just the accumulator is passed through the recursion. This transformation requires much more than just eta-expansion. Notably the worker-wrapper extension to list fusion proposed by Takano (Section 6.4) is able to produce this good code using just rewrite rules and without the help of special compiler optimizations.

We still decided to let foldl take part in list fusion based on the benchmark results, presented in the next section. They indicate that the real-world benefits in the common case of linear recursion are larger than the penalty in the non-linear recursion.

5.2 Measurement

No paper on optimizations without some benchmark results. We compare five variants:

- The baseline is GHC from March 14, 2014 (revision 23eb186), with Call Arity disabled and with the original definition of foldl (commit b63face in the base libraries reverted).
- To measure the effect of Call Arity analysis alone we enable Call Arity, but leave foldl with the original definition.
- The practically important result is the current state of the compiler, with Call Arity enabled and foldl implemented via foldr; this is highlighted in the tables.
- To assess the importance of Call Arity for allowing foldl to take part in list fusion, we also measured GHC without Call Arity, but with foldl via foldr.
- To assess the importance of the co-call analysis, we also measure how well an arity analysis without it, as described in [5], would have fared.

The ubiquitous benchmark suite for Haskell is nofib [8]; our results are shown in Table 1, including individual results for a few interesting benchmarks and the summary. For benchmarks with very short running times we omitted the numbers, as they are too inaccurate. The runtime numbers are generally not very stable, so we'd like to put the focus on the aggregate numbers.

As expected, enabling Call Arity does not increase the number of allocations; if it did something would be wrong. On its own, the analysis rarely has an effect. This is not surprising: Programmers tend to give their functions the right arities in the first place. Only in combination with making foldl a good consumer we can see its strength: Allocation improves considerably and without it, the change to foldl would actually degrade the runtime performance.

Call Arity affects the compile times in two ways: It makes them larger, because the compiler does more work. But it also reduces them, as the compiler itself has been optimized more. Table 2 shows the change in allocations done and time spent by the compiler while running the nofib test suite, and the change in time it takes to compile GHC itself (no allocation numbers are collected for this case). We can see that the latter is indeed happening – the number of allocations is reduced despite the compiler doing more work – but it does not make up for the increase in compilation time.

	Compile Allocs	Compile Time
nofib	-1.8%	+3.0%
GHC		+3.5%

Table 2. Compilation performance

5.3 Future work

5.3.1 Proofs of correctness

The equations of the analysis have grown in complexity beyond the stage of being obviously correct, in particular those for the co-call analysis of recursive bindings. It would therefore be desirable to have a formal proof of correctness. There are various notions of correctness of interest here.

(1) The transformation is semantics preserving.

(2) The specification (Section 3.1) is sufficient, i.e. if we have results adhering to the specification, then eta-expansion based on that does not worsen the program's performance.

(3) Solutions to the equations (Section 3.2) indeed fulfill the specification.

(4) The implementation calculates a correct solution to the equations, and terminates.

For our toy lambda calculus, (1) is trivially true as eta-expansion preserves the semantics with regard to the usual reduction rules for the lambda calculus. If we were to extend the language with side effects, or consider a semantics that models heap space usage, or counts beta-reductions, this is no longer the case; then the result depends on (2).

Point (2) is tricky because the specification is given informally. In order to proof something here, we would have to choose a semantics that allows us to formally express "e does not call both x_1 and x_2", just to be able to define the specification. Then this semantics needs to be related to one that measures performance in some way, and the no-worsen property can be proven.

The former semantics would also be needed to prove (3). We'd annotate the nodes of the syntax tree with the analysis results, assume that the equations hold and proof that in the end the observed calls behave like expected. This is likely the most tractable and also most useful notion of correctness.

A semantics suitable for this might be Launchbury's Natural Semantics for Lazy Evaluation [6], with modifications: In a judgment $\Gamma\colon e \Downarrow \Delta\colon v$, the context of e is lost, so at least the size of the (implicit) stack could be added to the judgments. Furthermore a field recording the calls, as a list of variable-arity pairs, would have to be added to the judgment.

It would be sufficient to annotate bindings with the calculated call-arity, and whether the binding was found to be called at most once, as from this the analysis result can be recovered. If the rules of the semantics put this information onto the heap it can be verified that the calculated arity and one-shotness matches the observed calls.

We believe that formal proofs in computer science should be, if possible, machine checked. A formal development of Launchbury's semantics in Isabelle is available [1] and could be used as a basis for this proof.

The forth notion of correctness would require a verified compiler first; verifying GHC itself is beyond our reach at this point. We can achieve some level of confidence in the correctness of Call Arity from the unit tests for the analysis and from the benchmark suite, where every increase in allocation would be an indication that something went wrong.

5.3.2 Improvements to the analysis

Call Arity does not fully exploit the behavior of thunks in mutual recursion. Consider this example:

```
let go x = if x > 10 then x else go (t1 x)
    t1   = if f (t2 a) then λy. go (y + 1) else λy. go (y + 1)
    t2   = if f b then λy. go (y + 1)
in go 1 2.
```

Currently, Call Arity will refrain from eta-expanding `t1` and `t2`, as they are part of a recursive binding. But `t2` is in fact called at most once! All calls to `t2` are done by `t1`, and `t1`'s result will be shared.

It remains to be seen if such situations occur in the wild and whether the benefits justify the implementation overhead.

6. Related work

Andrew Gill mentions in his thesis [5] on list fusion that eta-expansion is required to make `foldl` a good consumer that produces good code, and outlines a simple arity analysis. It does not consider the issue of thunks and is equivalent to the second refinement in Section 2.

6.1 GHC's arity analyses

The compiler already comes with an arity analysis, which works complementary to Call Arity: It ignores how a function is being used and takes its definition into account. It traverses the syntax tree and returns, for each expression, its arity, i.e. the number of arguments the expression can be applied to before doing any real work. This allows the transformation to turn $x\,?\,(\lambda y.\, e_1)\,:\,(\lambda y.\, e_2)$ into $\lambda y.\, (x\,?\,e_1\,:\,e_2)$ on the grounds that the check whether x is true or false is a negligible amount of work, and it is therefore still better to eta-expand the expression. Call Arity would refrain from doing this unless it knows for sure that the expression is going to be called at most once.

This arity analyzer can make use of one-shot annotations on lambda binders. Such an annotation indicates that the lambda will be called at most once, which allows the analysis to derive greater arities and expand thunks: If the lambdas in $f\ x\,?\,(\lambda y.\, e_1)\,:\,(\lambda y.\, e_2)$ are annotated as one-shot, this would be expanded to $\lambda y.\, (f\ x\,?\,e_1\,:\,e_2)$.

The working notes in [13] describe this analysis as the *forward arity analysis*. Like Call Arity, it can only determine arities of let-bound expression and will not make any use of arity information on parameters. Consider, for example

```
let g = ...
    s f = f 3
in ... (s g) ...
```

where we would have a chance to find out that `g` is always called with at least one argument. A *backward arity analysis* capable of doing this is also described in [13]. This analysis calculates the arity transformer of a function `f`, which indicates the arities `f` calls its arguments with, given the arity `f` is called with. It is not implemented in GHC as such, but subsumed by the new combined strictness/demand/cardinality analyzer: The function `s` would have a strictness signature of `<C(S)>`. The latest description of this analyzer can be found in [10].

Neither of these two analyses is capable of transforming the bad code from the introduction into the desired form. The former has to abort as the expression `f a` can be expensive; the latter looks at the definition of `goB` before analyzing the body and is therefore unable to make use of the fact that `goB` is always called with two arguments.

An integration of Call Arity into the demand analyzer would be difficult because of the different order they analyze the right-hand side and the body of a let-binding. In such a combined analysis, every **let** expression would require a fixpoint iteration to let information flow in both directions, which is not feasible.

There is, however, a potential for better integration of Call Arity with other analyses and transformations by making use of existing strictness and demand annotation, e.g. on imported identifiers, as well as by passing information to later phases: Thunks that Call Arity has determined to be called at most once can be marked as one-shot, even if they are not eta-expanded, and from the arity a function calls its arguments with, its demand signature could be pre-seeded.

6.2 Explicit one-shot annotation

While pondering the issue of a well-fusing foldl, Call Arity was competing with another way to solve the problem:

In that attempt we created a magic built-in function `oneShot :: (a → b) → a → b`. It is semantically the identity, but the compiler may assume that `oneShot f` is applied to its next parameter at most once. We can use this function when implementing foldl in terms of foldr:

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

This solves our problem with the bad code generated for `sum (filter f [42..2014])` from Section 5.1): The compiler sees

```
let go = λx. let r = if x == 2014 then id else go (x + 1)
             in if f x then oneShot (λa. r (a + x)) else r
in go 42 0
```

and, because the λa is marked as oneShot, the existing arity analysis will happily eta-expand go.

We decided to go the Call Arity route because it turned out to be no less powerful than the explicit annotation, and has the potential to optimize existing user code as well. Furthermore oneShot is unchecked, i.e. the programmer or library author has the full responsibility that the function is really applied only once; with Call Arity the analysis ensures correctness of the transformation.

6.3 unfoldr/destroy and Stream Fusion

There are various contenders to foldr/build-based list fusion, such as unfoldr/destroy [11] and Stream Fusion [3]. They have no problem fusing foldl, but have their own shortcomings, such as difficulties fusing unzip, filter and/or concatMap; a thorough comparison is contained in [2]. After twenty years, this is still an area of active research [4].

These systems are in practical use in array computation libraries like `bytestring` and `vector`. For the typical uses of lists they are inferior to foldr/build-based fusion, and the latter is still the system of choice for the standard Haskell list type.

6.4 Worker-wrapper list fusion

On the GHC mailing list, Akio Takano suggested an extension to foldr/build-based list fusion that will generate good code for left folds directly. The idea is that the consumer not only specifies what the generator should use instead of the list constructors (:) and [], but also a pair of worker-wrapper functions. Slightly simplified, his definition of foldl in terms of the extended foldrW is

```
foldl :: (b → a → b) → b → [a] → b
foldl f z = λxs. foldrW wrap unwrap g id xs z
  where
    wrap s e k a = k (s e a)
    unwrap u = λe a. u e id a
    g x next acc = next (f acc x).
```

The corresponding producer wraps the recursion in the provided wrapper. For example for `[from..to]` the code reads

```
[from..to] = buildW (eftFB from to)
eftFB from to wrap unwrap c n = wrap go from n
  where
    go = unwrap $ λi rest. if i ⩽ to
      then c i (wrap go (i + 1) rest)
      else rest.
```

The fusion rule is analogous to the foldr/buildr rule:

```
foldrW wrap unwrap c n (buildW g)
  = g wrap unwrap c n.
```

This suggestion is currently under evaluation and still fails in some cases where list fusion gives good results, but overall it looks promising: Not only does it produce good code directly, it even does so in tricky cases where eta-expansion is not enough, like fusing foldl with a list produced by treeToList (see Section 5.1.1). A prototype can be found at [12].

Acknowledgments

I would like to thank Simon Peyton Jones for the opportunity to work with him at Microsoft Research in Cambridge, for setting me on the task of making foldl a good consumer and for comments on this paper. I would also like to thank Andreas Lochbihler and Rebecca Schwerdt for helpful proof-reading and comments. This work is partly sponsored by the Deutsche Telekom Stiftung.

References

[1] J. Breitner. The correctness of Launchbury's natural semantics for lazy evaluation. *Archive of Formal Proofs*, Jan. 2013. http://afp.sf.net/entries/Launchbury.shtml, Formal proof development.

[2] D. Coutts. *Stream Fusion: Practical shortcut fusion for coinductive sequence types*. PhD thesis, University of Oxford, 2010.

[3] D. Coutts, R. Leshchinskiy, and D. Stewart. Stream fusion. from lists to streams to nothing at all. In *ICFP*, 2007.

[4] A. Farmer, C. Höner zu Siederdissen, and A. Gill. The HERMIT in the Stream: Fusing Stream Fusion's concatmap. In *PEPM*, 2014.

[5] A. J. Gill. *Cheap deforestation for non-strict functional languages*. PhD thesis, University of Glasgow, 1996.

[6] J. Launchbury. A natural semantics for lazy evaluation. In *POPL*, 1993.

[7] S. Marlow and S. L. Peyton Jones. Making a fast curry: push/enter vs. eval/apply for higher-order languages. *Journal of Functional Programming*, 16(4-5):415–449, 2006.

[8] W. Partain. The nofib benchmark suite of haskell programs. In *Functional Programming*, 1992.

[9] S. Peyton Jones, A. Tolmach, and T. Hoare. Playing by the rules: rewriting as a practical optimisation technique in GHC. In *Haskell Workshop*, 2001.

[10] I. Sergey, D. Vytiniotis, and S. Peyton Jones. Modular, Higher-order Cardinality Analysis in Theory and Practice. POPL, 2014.

[11] J. Svenningsson. Shortcut fusion for accumulating parameters & zip-like functions. In *ICFP*, 2002.

[12] A. Takano. Worker-wrapper fusion. https://github.com/takano-akio/ww-fusion, Prototype.

[13] D. N. Xu and S. Peyton Jones. Arity analysis, 2005. Working Notes.