Techniques for Improving the Efficiency of Functional Programs

Francisco Ribeiro
HASLab/INESC TEC
University of Minho
Braga, Portugal
francisco.j.ribeiro@inesctec.pt

***Abstract*—Combining different programs or code fragments is a natural way to build larger programs. This allows programmers to better separate a complex problem into simple parts. Furthermore, by writing programs in a modular way, we increase code reusability.**

However, these simple parts need to be connected somehow. These connections are done via intermediate structures that communicate results between the different components.

This review paper compiles different techniques used to remove intermediate structures and multiple traversals from programs written in functional languages.

***Index Terms*—functional programming, deforestation, program fusion, circular programs, lazy evaluation**

I. Introduction

Over the last several years, programming languages have evolved in order to provide powerful abstractions to programmers. Examples of such abstractions are models that represent code abstractions, powerful type systems and recursion patterns allowing the definition of functions that abstract the data type they traverse.

Examples of such recursion patterns are higher-order functions like *map* and *filter*. Composing operations like these ones makes it possible to express long, and sometimes complex, sequences of instructions with little effort.

However, if these mechanisms are not tuned appropriately, they may lead to efficiency problems, either by doing more traversals than necessary or by creating intermediate data structures. For example, the following concise Haskell function `all`,

```
all p xs = and (map p xs)
```

checks if all elements of a list `xs` satisfy a given predicate `p`. As we can see, it is expressed as a composition of functions `and` (conjunction of a list of booleans) and `map`. The `and` function is a fold on lists and, as a result, `all` is the composition of two higher-order functions.

```
all p xs = foldl (&&) True (map p xs)
```

In this definition, an intermediate list is created to communicate the results from one function to another. However, this program can be rewritten in a way which does not make use of an intermediate list.

```
all' p xs = h True xs
    where
        h b [] = b
        h b (x:xs) = h (b && p x) xs
```

Modifying the program's code in order to overcome these issues has the drawback of compromising readability and conciseness.

Furthermore, one does not wish to write programs in this style and, instead, prefers to use a more compositional style such as the first version of `all` provided that there are no performance penalties.

In addition, a more efficient implementation may not necessarily be the most natural solution to a problem, leading to increased difficulty during development and maintenance [1], [2].

In essence, programmers wish to write programs in the style they are most familiar with, not necessarily the most efficient one, and have them perform the best way possible. They want the best of both worlds.

Therefore, there's a need for techniques that automatically perform these kinds of optimizations automatically.

The remainder of this paper is structured in the following way: Sections II and III briefly describe lazy evaluation and deforestation, respectively. These two concepts are important to understand the basis of the techniques that will be explained subsequently. Sections IV, V, VI and VII describe specific fusion techniques used to optimize programs written in a functional style. In fact, these techniques are well known to the functional programming community and constitute key ideas behind important optimizations included in many compilers. Section VIII concludes the paper.

II. Lazy Evaluation

As stated previously, intermediate lists connect the different parts that assemble a program. Therefore, with strict evaluation, a lot of intermediate structures are allocated along the way which do not take part in the final result [3], [4]. The problem with the memory usage of these structures can be overcome with lazy evaluation. This way, because elements are generated as they are needed, there is no requirement for loading the entirety of the intermediate lists.

However, even under this mechanism, each list element still has to be allocated, checked and de-allocated [3].

Therefore, lazy evaluation in itself is not enough to overcome all the disadvantages introduced by the use of intermediate lists.

As such, in order to address this issue, several techniques have been developed throughout the years with the aim of completely eliminating the creation of these intermediate structures.

III. Deforestation

Deforestation is a technique allowing for the elimination of intermediate structures which are created and consumed soon afterward. Although the term may be used as a synonym to ”fusion” in general, it is generally used in order to refer to Philip Wadler's pioneer work [3], in which the author coins the term.

One of the first deforestation algorithms was presented by Philip Wadler [3] and, although it removed intermediate data structures, it had some disadvantages, as Gill et al. [2] state.

The major drawback of this kind of approach to the elimination of intermediate structures is the restriction imposed on the algorithm inputs. In his paper, Wadler presents what he calls a *treeless* form for defining functions which do not use any internal intermediate structures. The algorithm developed transforms a program composed by functions defined in *treeless* form into a single function, also defined in *treeless* form. As one can see, this is where one of the main disadvantages of this technique is evident. By limiting its application to functions defined in a restrictive form, the algorithm has a restricted range of inputs to operate on.

This places boundaries on the style of programming allowed to programmers, compromising code readability and conciseness.

A technique allowing the elimination of intermediate data structures, and thus creating a more efficient version, without sacrificing code clarity was needed.

IV. Short-cut Fusion

Gill et al. [2] present a transformation technique to create more efficient versions of programs through the elimination of intermediate lists. The core idea behind this deforestation technique are the algebraic transformations performed on some functions.

With these algebraic transformations, the authors show it is possible to standardize the way lists are consumed and produced. Furthermore, this algorithm allows every program as input.

In Haskell, one could define the well known list data type as:

```
data List a = Nil | Cons a (List a)
```

`foldr` is a function which behaviour consists of processing a list with an operator and returning the value it constructed along the way (accumulated in an initial value).

This systematic consumption of a list can be thought of as replacing every occurrence of `Cons` with the provided operator and the `Nil` instance with the initial value.

Therefore, many functions that consume lists in a constant way like the one just described can be expressed in terms of `foldr`. That is because this higher-order function encloses that kind of systematic consumption of a list.

Some example implementations of pre-defined functions resorting to `foldr` are:

```
map f xs = foldr (\a b -> f a : b) [] xs

filter f xs = foldr (\a b -> if f a
                             then a:b
                             else b)
                    [] xs
```

However, this standardization of list consumption is not enough to achieve the desired program transformation, as the following example demonstrates.

Supposing a composition of functions like:

```
sum (map f ls)
```

where `map` applies function `f` to each element of `ls` and `sum` performs the addition of every element in the list.

One could modify this program and have:

```
foldr (+) 0 (foldr ((:).f) [] ls)
```

where `foldr` is a higher-order function which consumes a recursive data structure (in this case, a list) by applying a given combining function in a systematic way to all the constituent parts, building a return value in the end.

But there isn't a rule that simplifies occurrences of *foldr/-foldr*. A workaround for this, could be rewriting these kinds of programs in a more specific way, and have the above example transformed in:

```
foldr ((+).f) 0 ls
```

The problem with this approach is that it is not very general. More precisely, it is very difficult to be sure we have sufficient rules. When another combination of functions is encountered, a new rule would need to be written so that that particular case would get simplified.

In the example used to illustrate this, the `foldr` on the outside had no way to know how the `foldr` on the inside was producing its result list. As such, we also need a way to standardize list production.

The abstraction described for list consumption consists in the replacement of every `cons` with a function and the `nil` at the end with a given value. And `foldr` encapsulates this behaviour by receiving a function `f` and an initial value `acc`.

Therefore, if list production is abstracted in terms of `cons` and `nil`, it is possible to obtain `foldr`'s effect if this list-producing abstraction is applied to `f` and `acc`.

As such, a function `build` can be defined like:

```
build g = g (:) []
```

Following the line of thought just described, we come up with the *foldr/build* rule, which can be expressed as:

```
foldr f acc (build g) = g f acc
```

As an example, one can consider the `upto` function which, given two numbers, produces a list that starts from the first one and continues until the second one.

In a very straightforward way, one could define this function as:

```
upto x y = if x>y then []
           else x : upto (x+1) y
```

But, as stated before, we can try to abstract the production of the list in terms of `cons` and `nil`, and thus getting the following definition:

```
upto' x y =
    \cons nil →
           if x>y then nil
           else cons x (upto' (x+1) y
                                cons nil)
```

Now, the function `upto` would be written like:

```
upto x y = build (upto' x y)
```

Deforestation is now possible if the list is produced using `build` and consumed using `foldr`:

```
mul (upto x y)
    = foldr (*) 1 (build(upto' x y))
    = upto' x y (*) 1
```

Applying the *foldr/build* rule (key elements highlighted inside red rectangles[1]) allows us to obtain a reduced form of the function `mul`, where no intermediate list is produced, which confirms the effect of deforestation.

V. Circular Programs

Algorithms that perform multiple traversals on the same data structure can be expressed as a single traversal function through a technique called *Circular Program Calculation*.

This kind of approach, first explored by Bird [5], highlights the importance of the lazy evaluation mechanism in functional languages like Haskell. In fact, defining circular programs in this way only works because of lazy evaluation. A circular definition has the consequence of creating a function call containing an argument that is, simultaneously, a result of that same call. Under a strict evaluation mechanism, this can be a problem as an infinite cycle is created because values are demanded before they can be calculated, leading to non-termination.

On the other hand, lazy evaluation allows for the computation of such circular structures. With this strategy, the right evaluation order of the expression is determined at runtime. More specifically, only the elements of the expression to be computed that are necessary to continue are expanded.

Although circular programs avoid unnecessary multiple traversals, they are not necessarily more efficient than their more straightforward counterparts [6] and are even more difficult to write. In fact, even more experienced programmers find it hard to understand programs written in such a way. In his paper, Bird proposes deriving these circular programs from their less efficient (in terms of number of traversals), but more natural, equivalent solutions.

[1]Colours assumed to be available

The example he uses is the function `repmin`, which has become a traditional example for being simple and a good assistant for the explanation of this particular technique.

The problem at hand is going to be the replacement of every leaf value in a tree with the original minimum value of the tree.

First of all, we must define a datatype for the tree. After that, we need a function `replace` and a function `tmin` to swap the tree's leaves for a given value and to calculate the minimum value of a tree, respectively.

With that, we can easily come up with a natural way of expressing the problem, which is implemented by the function `transform`.

```
data LeafTree = Leaf Int
              | Fork (LeafTree, LeafTree)

tmin :: LeafTree → Int
tmin (Leaf n) = n
tmin (Fork (l, r)) = min (tmin l) (tmin r)

replace :: (LeafTree, Int) → LeafTree
replace (Leaf _, m) = Leaf m
replace (Fork (l, r), m)
    = Fork (replace (l, m),
            replace (r, m))

transform :: LeafTree → LeafTree
transform t = replace (t, tmin t)
```

After having a straightforward solution to the problem, one can start applying Bird's proposed technique.

The first step consists of tupling. The functions `tmin` and `replace` both have a similar recursive pattern and operate on the same data structure. Therefore, a function `repmin` can be created by combining the results from the two previous functions in a tuple.

```
repmin (t, m) = (replace (t, m), tmin t)
```

Furthermore, a recursive definition of this function can be created, in which two cases need to be considered:

```
repmin (Leaf n, m)
    = (replace (Leaf n, m), tmin (Leaf n))
    = (Leaf m, n)

repmin (Fork (l, r), m)
    = (replace (Fork (l, r), m),
       tmin (Fork (l, r)))
    = (Fork (replace (l, m), replace (r, m)),
       min (tmin l) (tmin r))
    = (Fork (l', r'), min n1 n2)
```

```
        where (l', n1) = repmin (l, m)
              (r', n2) = repmin (r, m)
```

The final step is where circular programming is used in order to put together the two elements forming the result of `repmin`.

Highlighted inside blue rectangles is the presence of circularity; `m` is being used simultaneously as an argument and a result of the same call.

```
transform :: LeafTree → LeafTree
transform t = nt
    where (nt,m) = repmin (t,m)
```

However, this method for deriving circular programs presents a drawback.

Although it allows the derivation of a circular program from a more natural and intuitive equivalent, removing the burden of having to come up with such a complicated implementation and creating a circular alternative which makes less traversals on the data structure, this technique does not guarantee termination. Fernandes et al. developed a different technique based on short-cut fusion for deriving circular programs [7]. Circular programs have also been the subject of study in other research works [8]–[11].

VI. Stream Fusion

The work by Coutts et al. [1] in *Stream Fusion* consists of an automatic deforestation system that takes a different approach compared to more traditional short-cut fusion systems.

The approach taken by [2] with the *foldr/build* rule is to fuse functions that work directly over the original structure of the data, that is, lists.

In *Stream Fusion*, the operations over the original list structure are transformed in order to, instead, work over the co-structure of the list.

As Coutts et al. [1] state, the natural operation over a list is a *fold*, while on the other hand, the natural operation over a stream is an *unfold*. Therefore, a list's co-structure is a stream.

The Stream datatype encloses that unfolding behaviour. In order to achieve this, it wraps an initial state and a stepper function which specifies how elements are produced from the stream's state.

```
data Stream a
   = ∃s. Stream (s → Step a s) s
```

The stepper function produces a `Step` element, which permits three possibilities:

```
data Step a s = Done
              | Yield a s
              | Skip s
```

The `Step` datatype allows the co-structure to be non-recursive, thanks to the `Skip` data constructor. This is the key point of the stream fusion system. The `Skip` constructor is what allows the production of a new state without yielding a particular element and this is a crucial point as it permits every stepper function to be non-recursive.

The `Done` and `Yield` alternatives are quite simple as they pinpoint the end of a stream and carry an actual element together with a reference to the rest of the stream's state, respectively.

In order to convert list structures to streams and vice-versa, two functions are needed.

```
stream :: [a] → Stream a
stream xs0 = Stream next xs0
        where
            next [] = Done
            next (x:xs) = Yield x xs

unstream :: Stream a → [a]
unstream (Stream next0 s0) = unfold s0
    where
        unfold s = case next0 s of
            Done → []
            Skip s' → unfold s'
            Yield x s' → x : unfold s'
```

The function `stream` creates a Stream with:

- a stepper function `next` which is non-recursive and yields each element of the stream as it unfolds;
- a state, which consists of the list itself.

On the other hand, the function `unstream` creates a list by unfolding the given stream, repeatedly calling the stream's stepper function.

Implementing functions to perform operations over streams is quite simple. The function intended has to define the particular stepper function for the stream it is going to return as a result. Considering the simple and well known `map` example operating on lists, one would define its stream counterpart as:

```
map_s :: (a → b) → Stream a → Stream b
map_s f (Stream next0 s0) = Stream next s0
    where
        next s = case next0 s of
            Done → Done
            Skip s' → Skip s'
            Yield x s' → Yield (f x) s'
```

What $\texttt{map}_s$ does here is define a stepper function that applies the function given as a parameter of $\texttt{map}_s$ to every yielded element of the stream.

A very simple but important case where one can see the effect of the stream fusion approach is the function $\texttt{filter}_s$. Its implementation allows us to observe the true impact of this technique.

```
filter_s :: (a → Bool) → Stream a → Stream a
filter_s p (Stream next0 s0) = Stream next s0
    where
        next s = case next0 s of
            Done → Done
            Skip s' → Skip s'
            Yield x s' | p x → Yield x s'
                       | otherwise → Skip s'
```

The only way that the function $\texttt{filter}_s$ is non-recursive is because of `Skip`. This constructor, when put in place of the elements that should be removed from the stream, allows us to avoid the recursion otherwise necessary to process every stream element in order to find out which ones satisfy the given predicate.

More precisely, in the last line of the above implementation, `Skip` is introduced whenever an element does not pass the predicate's test.

This way, code can be better optimized by general purpose compiler optimizations.

In order to use the stream fusion approach on lists, one has to convert lists to streams and back again. This way, functions from the Stream setting (like the previous $\texttt{map}_s$ and $\texttt{filter}_s$ examples) can be applied, as they are intended to operate on streams. This is accomplished by using functions `stream` and `unstream`. As an example, function `map` on lists is specified in the following way:

```
map :: (a → b) → [a] → [b]
map f = unstream . map_s f . stream
```

This way, each function has to construct a Stream, perform its task and then build a list. Doing this for every function in a stream pipeline would be very inefficient. Considering the example of composing a $\texttt{filter}_s$ and a $\texttt{map}_s$, the following is obtained:

```
filter p . map g =
     unstream . filter_s p . stream .
     unstream . map_s g . stream
```

Communicating the results from $\texttt{map}_s$ to $\texttt{filter}_s$ builds an intermediate list (generated by `unstream`), which gets consumed right away (`stream`). But there is a chance to eliminate this intermediate list.

`stream . unstream` is the identity on streams and, as a result, it can be removed. Formalizing, this originates the *stream/unstream fusion* rule:

```
∀s . stream (unstream s) ↦ s
```

The Glasgow Haskell Compiler allows us to write rules that will then be used while compiling our programs. These "custom rules" can be expressed through pragmas, which are instructions that can be given to the compiler. Expressing the previous rule through these pragmas will make the example be transformed into:

```
unstream . filter_s p . map_s g . stream
```

The *stream/unstream fusion* rule is not a traditional fusion rule, as it merely eliminates lists that got created while converting operations.

In fact, the method documented so far has a very curious and important implication. As previously stated, one can define pragmas in order to extend the compiler. Some algebraic transformations can be expressed through these pragmas. For example:

```
map f (map g xs) = map (f.g) xs
```

This algebraic rule expresses that a composition of `maps` is equivalent to the `map` of the composition of the two functions. This rule allows for the generated code to be more efficient.

However, there is a multitude of possible function combinations and, as a consequence, one could never be certain of the number of rules necessary to cover all cases.

This is a point where the work by Coutts et al. [1] plays an important role. As presented earlier, when writing different stream combinators (like $\texttt{map}_s$ and $\texttt{filter}_s$), the outcome of each stepper function that is defined depends on the outcome of the previous stream's stepper function.

This way, whenever a stepper function of a stream is called, every stepper function of the streams preceding the current one is going to be executed.

Therefore, functions are fused without the need to explicitly state the rules performing those transformations.

The main goal of program fusion is to eliminate intermediate data structures. However, *Stream Fusion* achieves that at the cost of introducing lots of intermediate `Step` values. These allocations are going to be responsible for a great amount of overhead. This situation is overcome thanks to several optimization techniques included in *GHC* (e.g. case-of-case transformation and constructor specialisation). Therefore, programs are automatically transformed and, in the end, the most efficient solution is obtained (where all the intermediate values mentioned have been eliminated, thus reducing unnecessary allocations).

VII. Hylo System

Program calculation is what is behind the techniques described to transform programs into more efficient versions. These techniques are based on many existing transformation laws. However, these rules only allow us to work with programs by hand, therefore leaving the application of program transformations necessary to obtain more efficient versions to the programmer, and not to the computer. Fusion systems are, as a consequence, not automatic.

Algorithms need to be developed that construct programs based on those transformation laws. This is what the HYLO System by Onoue et al. [12] aims to be: a fusion system applying these transformations in a more universal and regular way than existing ones.

First of all, we need to understand that there are two possible approaches to fusion: search-based fusion and calculational fusion.

The first one, search-based fusion, unfolds recursive definitions of functions to find suitable places inside those expressions to perform folding operations. But to achieve this, this kind of method needs to keep track of the function calls so it can control the unfolding, in order to avoid an infinite process. As this introduces a great overhead, fusion cannot be practically implemented this way.

The work by Onoue et al. [12] focuses on the second kind of fusion, calculational fusion, which has been the object of a lot of investigation over the years.

This approach explores the recursive structure of each component of the program in order to apply fusion through existing transformation laws.

However, most of the proposed techniques for fusion have the slight inconvenient of forcing the programmer to express the functions in terms of the necessary recursive structure, so that the different transformations can be applied. This is impractical, as it leads the programmer away from more potentially readable and natural implementations.

With this in mind, when briefly explaining their approach, the authors of the HYLO System start by stating that the majority of recursive functions can be expressed in terms of a very specific recursive form: hylomorphism.

An hylomorphism is the composition of an anamorphism (list production) followed by a catamorphism (list consumption).

Consider the following Haskell implementation of the *Fibonacci* function:

```
fib 0 = 0
fib 1 = 1
fib n = fib (n−1) + fib (n−2)
```

The sequence of calls generated by this program could be generalized over a binary tree, which would then collapse in order to calculate the desired n^{th} *Fibonacci* term.

This, in essence, is a hylomorphism, in which the anamorphism corresponds to the generation of the call tree and the catamorphism to its collapse.

Indeed, this program could be rewritten in terms of an `unfold` (anamorphism) followed by a `fold` (catamorphism).

```
fib'_T :: Integer → Integer
fib'_T = fold_T (+) id ∘ unfold_T g
    where
        g 0 = Left 0
        g 1 = Left 1
        g n = Right (n−1, n−2)
```

By expressing the program this way, if the two recursive patterns that compose the hylomorphism get fused, the creation of the intermediate structure (call tree) is avoided.

In order to rewrite the program's recursive components in terms of hylomorphisms, the authors of the HYLO System developed an algorithm to derive such general recursive structures from the recursive definitions of the program.

As such, for the previous example, the algorithm would derive the `fold ∘ unfold` definition from the original `fib` implementation.

Following that, schemes for data production and consumption need to be captured so that the *Acid Rain Theorem* can be applied to hylomorphisms, in order to fuse them.

The final step consists of inlining the resulting hylomorphism into a normal recursive definition, in which the intermediate structures have been eliminated.

All in all, the HYLO System allows programs to be written without the concern of expressing them in terms of specific and more generic recursive structures, as these are derived by an automatic algorithm. Thus, fusion laws can still be applied, leading to more efficient programs without sacrificing so much code readability and without forcing programmers to express functions under certain recursive patterns. This system was incorporated into the Haskell compiler.

VIII. Conclusion

Throughout the years, programming languages have come up with new mechanisms that allow programmers to abstract more complex ideas into simple instructions. However, these abstractions may lead to performance issues. Chaining several *higher order functions* can cause a program to perform extra unnecessary traversals and operations if optimization techniques like *fusion* are not implemented.

Techniques like *deforestation* and *short-cut fusion* aimed to eliminate intermediate structures that were inevitably created as a way to "glue" different functions together. Other approaches, like *circular program calculation*, focused on converting algorithms which performed multiple traversals to programs performing a single one.

Ultimately, *Stream Fusion* accomplishes both. By rewriting Haskell's List library functions in order to adapt them to the *Stream* setting and, together with that, integrating with an existing set of compiler optimization rules, this approach accomplishes some kind of automation when it comes to fusion. Automating this final step is something that previous techniques have missing. In a similar way, the *HYLO System* also tries to perform these transformations automatically by extracting the recursive structure of programs and performing fusion through the application of transformation laws.

In recent years, many languages such as C++, C# and Java started adopting functional constructs as a form of enriching the way they allow people to write programs. As some of these constructs include many of the *higher-order functions* presented previously, the techniques discussed in this paper are of utter importance as the introduction of this "functional flavour" demands the inner workings of these languages to cope with the introduced overhead by somehow mimicking the formerly described optimizations.

Other areas, that are somehow related to functional programming, also benefit from the kind of mechanisms described throughout this paper. Attribute grammars is an example of such an area in which fusion techniques play a very important role [13].

References

[1] D. Coutts, R. Leshchinskiy, and D. Stewart, "Stream fusion: From lists to streams to nothing at all," in *Proceedings of the 12th ACM SIGPLAN International Conference on Functional Programming*, ser. ICFP '07. New York, NY, USA: ACM, 2007, pp. 315–326. [Online]. Available: http://doi.acm.org/10.1145/1291151.1291199

[2] A. Gill, J. Launchbury, and S. L. Peyton Jones, "A short cut to deforestation," in *Proceedings of the Conference on Functional Programming Languages and Computer Architecture*, ser. FPCA '93. New York, NY, USA: ACM, 1993, pp. 223–232. [Online]. Available: http://doi.acm.org/10.1145/165180.165214

[3] P. Wadler, "Deforestation: transforming programs to eliminate trees," *Theoretical Computer Science*, vol. 73, no. 2, pp. 231 – 248, 1990. [Online]. Available: http://www.sciencedirect.com/science/article/pii/030439759090147A

[4] ——, “Listlessness is better than laziness: Lazy evaluation and garbage collection at compile-time,” in *Proceedings of the 1984 ACM Symposium on LISP and Functional Programming*, ser. LFP ’84. New York, NY, USA: ACM, 1984, pp. 45–52. [Online]. Available: http://doi.acm.org/10.1145/800055.802020

[5] R. S. Bird, “Using circular programs to eliminate multiple traversals of data,” *Acta Informatica*, vol. 21, no. 3, pp. 239–250, Oct 1984. [Online]. Available: https://doi.org/10.1007/BF00264249

[6] J. P. Fernandes, J. Saraiva, D. Seidel, and J. Voigtländer, “Strictification of circular programs,” in *Proceedings of the 20th ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation*, ser. PEPM ’11. New York, NY, USA: ACM, 2011, pp. 131–140. [Online]. Available: http://doi.acm.org/10.1145/1929501.1929526

[7] J. P. Fernandes, A. Pardo, and J. Saraiva, “A shortcut fusion rule for circular program calculation,” in *Proceedings of the ACM SIGPLAN Workshop on Haskell Workshop*, ser. Haskell ’07. New York, NY, USA: ACM, 2007, pp. 95–106. [Online]. Available: http://doi.acm.org/10.1145/1291201.1291216

[8] P. Martins, J. P. Fernandes, and J. Saraiva, *Zipper-Based Modular and Deforested Computations*. Cham: Springer International Publishing, 2015, pp. 407–427. [Online]. Available: https://doi.org/10.1007/978-3-319-15940-9_10

[9] A. Pardo, J. P. Fernandes, and J. Saraiva, “Shortcut fusion rules for the derivation of circular and higher-order monadic programs,” in *Proceedings of the 2009 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-based Program Manipulation, PEPM 2009, Savannah, GA, USA, January 19-20, 2009*, G. Puebla and G. Vidal, Eds. ACM, 2009, pp. 81–90. [Online]. Available: http://doi.acm.org/10.1145/1480945.1480958

[10] ——, “Shortcut fusion rules for the derivation of circular and higher-order programs,” *Higher-Order and Symbolic Computation*, vol. 24, no. 1, pp. 115–149, Jun 2011. [Online]. Available: https://doi.org/10.1007/s10990-011-9076-x

[11] ——, “Multiple intermediate structure deforestation by shortcut fusion,” *Science of Computer Programming*, vol. 132, pp. 77 – 95, 2016, selected and extended papers from SBLP 2013. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0167642316300880

[12] Y. Onoue, Z. Hu, M. Takeichi, and H. Iwasaki, “A calculational fusion system hylo,” in *Proceedings of the IFIP TC 2 WG 2.1 International Workshop on Algorithmic Languages and Calculi*. London, UK, UK: Chapman & Hall, Ltd., 1997, pp. 76–106. [Online]. Available: http://dl.acm.org/citation.cfm?id=265779.265797

[13] J. Saraiva and D. Swierstra, “Data Structure Free Compilation,” in *8th International Conference on Compiler Construction, CC/ETAPS’99*, ser. LNCS, Stefan Jähnichen, Ed., vol. 1575. Springer-Verlag, March 1999, pp. 1–16.