Operation fusion and deforestation for Scala

Dmytro Petrashko
I&C, EPFL

***Abstract*—Fusion and deforestation have been a topic of research for last 4 decades. An extensive research in this topic was done in the context of for Haskell programming language. This report examines three techniques explored in the Haskell community: *foldr/build*, *unbuild/unfoldr* and stream fusion, all of which try to represent computations with a small set of combinators that are easy to reason about and optimize. We outline strong and weak points of these approaches, providing a starting point for researching deforestation and fusion for Scala.**

***Index Terms*—Streams, deforestation, pipelining, program optimization, program transformation, program fusion, functional programming**

I. Introduction

FUSION is a technique that aims to transform a program that is easy to understand into one that is efficient. As an example consider a program in Haskell[SPJ'03]:

$$sumSq\ xs = sum(map\ sq\ xs)$$
$$sum\ [\,] = 0$$
$$sum\ (x\ :\ xs) = x + sum\ xs$$
$$map\ f\ [\,] = [\,]$$
$$map\ f\ (x : xs) = f\ x\ :\ map\ f\ xs$$
$$sq\ x = x * x$$

This program is written in an elegant and composable way, combining reusable functions that operate on lists. Naive evaluation of *sumSq* will first construct the intermediate list, a result of mapping, and then traverse it, calculating sum of elements in this list. However, this is suboptimal. Consider the following definition of sumSq:

$$sumSq\ [\,] = 0$$
$$sumSq\ (x\ :\ xs) = x * x + sumSq\ xs$$

This definition does not require the construction of an intermediate data structure, requiring less memory and less time to execute. Yet, the second implementation of sumSq trades clarity and modularity for performance. Fusion improves the programming experience by allowing the programmer to write clear and modular code which is then transformed automatically to efficient code by avoiding the creation of redundant intermediate data structures.

This paper is structured as follows:

- Section II outlines early developments in deforestation.
- Section III presents ideas shared by Shortcut fusion.
- Section IV examines system proposed in paper "Cheap deforestation in practice: An optimiser for Haskel"[Gill'94].
- Section V goes in detail in demonstrating system from paper "Shortcut fusion for accumulating parameters & zip-like functions"[Svenningsson'02].
- Section VI describes more recent developments from paper "Stream fusion: from lists to streams to nothing at all"[Coutts'07] that we feel as a more practical system, both for users and for implementors.
- Section VII summarizes weak points of all those systems.
- Section VIII names several possible research directions as well as Scala perspective.

Proposal submitted to committee: June 25th, 2014; Candidacy exam date: July 2th, 2014; Candidacy exam committee: Prof. Christoph Koch, Prof. Martin Odersky, Prof. George Candea.

This research plan has been approved:

Date: _____

Doctoral candidate: _____
(name and signature)

Thesis director: _____
(name and signature)

Doct. prog. director: _____
(B. Falsafi) (signature)

II. Pioneering works

The pioneering paper in fusion is the "A transformation system for developing recursive programs" by Burnstall dated 1977[Burstall'77]. It proposes a method that can be used by programmers to transform a program into a faster one, even potentially improving the asymptotic complexity.

The proposed method is not an automatic technique that can be applied by the compiler: the developer needs to design and implement representations as well as choose which rewriting rules to apply.

Being a programmer guided compiler transformation, it heavily depended on 'eureka' step, in which programmer cames up with better representations. The paper [Burstall'77] presents the function computing Fibonacci numbers as an example. This is an example that illustrates the distinguishing features of their technique. They start with original recursive definition that takes exponential time to evaluate:

$$\begin{aligned} f\ 0 &= 0 \\ f\ 1 &= 1 \\ f\ (n+2) &= f\ (n+1) + f\ n \end{aligned}$$

Then the programmer should perform an 'eureka' step, comming up with auxiliary definition:

$$g\ n = (f(n+1), f(n))$$

Then in terms of this definition the original function can be defined:

$$\begin{aligned} f\ 0 &= 0 \\ f\ 1 &= 1 \\ f\ (n+2) &= u + v, \quad \text{where} (u, v) = g\ n \\ g\ 0 &= (1, 0) \\ g\ (n+1) &= (u + v, u), \ \text{where} (u, v) = g\ n \end{aligned}$$

The techique presented in this paper allowed to achive a different pattern of recursion and have improved running time from exponential to linear. Unfortunately, none of the techniques that we will consider later in this report can change the asymptotic complexity in this example.

This is why we will illustrate all those techniques on a simple example of summing either squares of values in a list or filtered list of values.

Wadler's algorithm

Most of the research that followed [Burstall'77] tried to find restrictions and abstractions that allow an automatic algorithm to run without help of programmer. This led to first algorithm by Wadler [Wadler'90], that was limited to removing tree structures in a special case of compositions of functions in "treeless form". The algorithm was proven to never increase number of allocation and examples suggested that in many cases allocations do decrease.

The main downside of Wadler's deforestation algorithm was limitation was the strict shape it was imposing on the program: variables were to be used linearly and no intermediate data structures were allowed.

III. Shortcut fusion

The deforestation idea did not look practical until *Shortcut fusion* was presented in [Gill'93], which was implemented directly in Glasgow Haskell Compiler.

The biggest advantage of this approach was it's simplicity, as only local rewrite rules are required from the compiler. The simplest example of rule would be:

$$\forall f,\ g \quad map(f) \cdot map(g) = map(f \cdot g)$$

On this simple example of a rule we can illustrate most requirements for such optimizing rules:

- rewriting must be proven to be equivalent to original code
- rewriting must be proven to terminate
- rewriting should be shown to be an optimization

Proving this independently for every rule is as a simpler approach and most systems tried to follow this general idea. The fact that the shortcut fusion approach only uses local transformation is a key advantage in comparison with a more general deforestation algorithm. Having proved those statements for every rewriting equation, complete correctness of algorithm is fairly straightforward.

Supporting shortcut fusion required adding support for pluggable rules in the GHC. Rewrite rules can be added to a source file and are automatically used by the compiler to optimize the program. This allowed experimenting and made subsequent research easier. In current syntax of GHC rules language, the *map/map* rule can be written as

$$\{-\#\ RULES\ "map/map"\ \forall\ f\ g.\ map\ f\ .\ map\ g = map\ (f.g)\ \#-\}$$

As naive generalizations of rules similar to *map/map* rule, taking into account more functions such as *filter*, *mapConcat*(`flatMap` in Scala terms) leads to quadratic explosion in the number of rules required. Thus most systems have found a small set of combinators used to express a large class of functions. As main goal of deforestation was eliminating intermediate data structures, intuitively, these combinators should capture patterns such as constructing, consuming or transforming the intermediate data structure.

This led to second requirement, advantage in terms of simplicity and limitation for users: "Shortcut" deforestation approach is based on idea that, instead of considering general definitions, it only attempts to improve definitions, which are themselves defined using particular designated functions - fusion combinators.

Multiple *Shortcut fusion* systems were proposed, but the most notorious and the only incorporated in a mainline compiler is "*foldr/build*" system.

IV. Cheap deforestation in practice: An optimiser for Haskell.

The shortcut fusion system proposed in [Gill'94], also known as "*foldr/build*" defines two combinator functions: *foldr* and *build*.

$$foldr :: (a \to b \to b) \to b \to [a] \to b$$

$$build :: ((a \to b \to b) \to b \to b) \to [a]$$

The foldr function is known to Scala programmers as *foldRight* method:

$$foldr\ f\ z\ [\,] = z$$
$$foldr\ f\ z\ (x : xs) = f\ x(foldr\ f\ z\ xs)$$

It can be thought as an general pattern of recursion on lists and can express a huge range of common functions on list:s

$$sum = foldr\ (+)\ 0$$
$$xs + +ys = foldr\ (::)\ ys\ xs$$
$$map\ f = foldr(\lambda\ x\ xs \rightarrow f\ x : xs)[\,]$$
$$filter\ p = foldr(\lambda\ x\ xs \rightarrow if\ p\ x\ then\ x : xs\ else\ xs)[\,]$$
$$foldl\ f\ v\ xs = foldr(\lambda\ x\ g \rightarrow (\lambda\ a \rightarrow g(f\ a\ x)))id\ xs\ v$$
$$dropWhile\ p = fst \cdot foldr\ f([\,],[\,]) \quad where\ f\ x(ys, xs) = (if\ p\ x\ then\ ys\ else\ x : xs, x : xs)$$

Hutton [Hutton'99] illustrates how many common functions can be rewritten in terms of *foldr*. The last two are the most surprising, as they illustrate that that usage of higher order functions or pair types can extend expressiveness beyond commonly used patterns.

The second combinator function *build* is not common and less obvious:

$$build :: ((a \rightarrow b \rightarrow b) \rightarrow b \rightarrow b) \rightarrow [a]$$
$$build\ g = g(:)[\,]$$

While *fold* is to be used to consume lists, *build* can be used to construct them.

The key and the only rule of "*foldr/build*" system is

$$\forall f\ g\ z.\ foldr\ f\ z(build\ g) = g\ f\ z$$

On the left hand side we see *build* producing a list and *foldr* immediately consuming it. The right hand side does not create lists at all. Thus if one explains list operation in terms of *build* and *foldr* this local rewrite rule can be used to eliminate intermediate lists.

Being implemented in GHC, the "*foldr/build*" system is very beneficient in many practical cases, it is basis for performance of list comprehensions in Haskell([SPJ'03]Section 3.11).

In order to illustrate this system on a simple example, we show rewrting of such definitions:

$$sumSq\ xs = sum(map\ sq\ xs)$$
$$sq\ x = x * x$$

The rewriting passes will be such

$sumSq\ xs$
$= \{$inlining the definition of *sumSq*$\}$
$sum(map\ sq\ xs)$
$= \{$inlining the definition of *sum*$\}$
$foldr\ (+)\ 0\ (map\ sq\ xs)$
$= \{$inlining the definition of *map*$\}$
$foldr\ (+)\ 0\ (build(\lambda\ c\ n \rightarrow foldr(\lambda\ x\ ys \rightarrow sq\ x\ c\ ys)\ n\ xs))$
$= \{$*foldr/build* rewrite rule$\}$
$= (\lambda\ c\ n \rightarrow foldr(\lambda\ x\ ys \rightarrow sq\ x\ c\ ys)n\ xs)\ (+)\ 0$
$= \{beta$-reduce$\}$
$foldr(\lambda\ x\ ys \rightarrow sq\ x + ys)\ 0\ xs$

Unfortunately, not all common list operations can be written effectively in terms of fold. In particular, *foldl* and *zip* proved to be problematic.

Also, despite the fact that *foldl*(*foldLeft* in terms of Scala) can be expressed in term of *foldr*, and can be fused, the resulting code uses higher order functions and is extremely inefficient.

The *sumSq* example also shows a problem with accumulating parameters. We had to define *sum* as *foldr* while it is more efficient to define it as a *foldl*. Indeed, the resulting function is not tail recursive and uses linear stack space. If *sum* is defined using *foldl* in a standard way, then, while the fusion will succeed, running time will still degrade due to usage of high-order functions.

Those examples illustrate that, while the fusion transformation can be an interesting theoretical improvement we should watch out not to be writing functions in highly sophisticated way just to move memory allocations from one place to another, even degrading performance in some cases.

The *zip* function is a problem for this system, as it cannot be written to consume both lists with a *foldr*. It can be written to produce the result list using *build* and consume single one input list using *foldr*, but not both at the same time.

V. The *unbuild/unfoldr* fusion system

[Svenningsson'02] has proposed a way to overcome the challenge of left folds and zips. It is another shortcut fusion system, using different fusion combinators: unfoldr and unbuild[1]:

$$unfoldr :: (s \rightarrow Maybe(a, s)) \rightarrow s \rightarrow [a]$$
$$unbuild :: ((s \rightarrow Maybe(a, s)) \rightarrow s \rightarrow b) \rightarrow [a] \rightarrow b$$

Unforldr[2] captures a general pattern for constructing lists, it can be seen as an iterator style, when a function is used to generate sequence of elements, passing new state with each new element that would be used to generate subsequent one. Implementation of *unfold* can be such:

$$unfoldr\ next\ s = case\ next\ s\ of\ Nothing- > [\,]\ Just(x, z)- > x : unfoldr\ next\ z$$

[1]Unbuild is sometimes called destroy

[2]*Maybe* is analogous to Scala *Option*, with two possibilities: *Nothing* and *Just(value)*

By passing state with a sequence we can express a wide range of list-producing functions:

$$
\begin{aligned}
iterate\ f &= unfoldr(\lambda x \to Just(x, f\ x)) \\
fromTo\ n\ m &= let\ next\ i | i > m = Nothing \\
&\qquad\qquad | \_ = Just(i, i+1) \\
&\quad in\ unfoldr\ next\ n \\
map\ f &= let\ next[\,] = Nothing \\
&\qquad next(x : xs) = Just(f\ x, xs)) \\
&\quad in\ unfoldr\ next \\
filter\ p &= let\ next[\,] = Nothing \\
&\qquad next(x : xs) | px = Just(x, xs)) \\
&\qquad\qquad | \_ = next\ xs \\
&\quad in\ unfoldr\ next
\end{aligned}
$$

Note, that *map* and *filter* not only produce a list, they also consume one. In order to facilate fusion we need to further rewrite them in terms of second combinator funtion.
Logically, while *unfoldr* is for producing lists, *unbuild* is for consuming them. Given a function that consumes elements in the iterator style, it applies it to an argument, obtaining an equivalent function that consumes lists.

$$unbuild :: ((s \to Maybe(a, s)) \to s \to b) \to [a] \to b$$

$$
\begin{aligned}
unbuild\ g\ xs &= g\ uncons\ xs \\
&\quad where \\
&\qquad uncons :: [a] \to Maybe(a, [a]) \\
&\qquad uncons[\,] = Nothing \\
&\qquad uncons(x : xs) = Just(x, xs)
\end{aligned}
$$

Given this function we can express different consumers, and rewrite transform methods like *map* and *filter* to also consume their input using it.

$$
\begin{aligned}
foldr\ f\ z &= unbuild(\lambda next\ s \to \\
&\quad let\ go\ s = case\ next\ s \\
&\qquad Nothing \to z \\
&\qquad Just(x, s) \to f\ x(go\ s) \\
&\quad in\ go\ s) \\
foldl\ f\ a &= unbuild(\lambda next\ s \to \\
&\quad let\ go\ a\ s = case\ next\ s\ of \\
&\qquad Nothing \to a \\
&\qquad Just(x, s) \to go(f\ a\ x)s \\
&\quad in\ go\ a\ s) \\
map\ f &= unbuild(\lambda next\ s \to \\
&\quad let\ next\ s = case\ next\ s\ of \\
&\qquad Nothing \to Nothing \\
&\qquad Just(x, s) \to Just(f\ x, s) \\
&\quad in\ unfoldr\ next\ s)
\end{aligned}
$$

$$
\begin{aligned}
filter\ p &= unbuild(\lambda next\ s \to \\
&\quad let\ next\ s = case\ next\ s\ of \\
&\qquad Nothing \to Nothing \\
&\qquad Just(x, s) | p\ x \to Just(x, s) \\
&\qquad\qquad | otherwise \to next\ s \\
&\quad in\ unfoldr\ next\ s)
\end{aligned}
$$

The main rewriting rule *unbuild/unfoldr* for system is:

$$\forall k\ g\ s.\ unbuild\ g(unfoldr\ k\ s) = g\ k\ s$$

We demonstrate this fusion technique on the same *sumSq* example.

$$sumSq\ xs = sum(map\ sq\ xs)$$

This time, in order to illustrate how this approach handles accumulating parameters, we will define sum in terms of *foldl*, which itself is defined directly in terms of *unbuild*.

$$sum = foldl\ (+)\ 0$$

For the sake of being less verbose we define helper functions:

$$
\begin{aligned}
sum &= unbuild\ sumIter \\
map\ f &= unbuild(mapIter\ f) \\
sumIter\ next_0\ s_0 &= sumGo\ next_0 0 s_0 \\
mapIter\ f\ next_0\ s_0 &= unfoldr(next_{map}\ f\ next_0)s_0 \\
sumGo\ next_0 = &let\ go\ a\ s = \ case\ next_0\ s\ of \\
&\qquad Nothing \to a \\
&\qquad Just(x, s) \to go(a + x)s \\
&\quad in\ go \\
next_{map}\ f\ next_0\ s &= case\ next_0\ s\ of \\
&\qquad Nothing \to Nothing \\
&\qquad Just(x, s) \to Just(fx, s)
\end{aligned}
$$

In such definitions we have

$$
\begin{aligned}
&sum(map\ sq\ xs) \\
&\quad = \ unbuild\ sumIter(unbuild(mapIter\ sq)\ xs)
\end{aligned}
$$

Here we need to introduce one more rule for this system *unbuild/unbuild*:

$$\forall g\ k\ xs.\ unbuild\ g(unbuild\ k\ xs)$$

$$= \ unbuild\ (\lambda next\ s_0 \to unbuild\ g(k\ next\ s_0))\ xs$$

We can now rewrite further:

$$
\begin{aligned}
&unbuildsumIter(unbuild(mapIter\ sq)\ xs) \\
&= \ unbuild(\lambda next\ s_0 \to \\
&unbuild\ sumIter(mapIter\ sq\ next\ s_0))\ xs
\end{aligned}
$$

By inlining the definition of mapIter and applying the *unbuild/unfoldr* rule we will get:

$$= \ unbuild(\lambda next\ s_0 \to sumIter(next_{map}\ sq\ next))$$

Here, we have succeeded in fusing all code, but unfortunately if we leave result as-is we will still get poor runtime performance. The reason is in fact that $sumIter$ and $next_{map}$

pair of functions will be constantly allocating $Maybe$ objects. So by applying fusion we did not reduce number of allocated objects, we just moved allocations from one place into another.

[Svenningsson'02] assures that those $Maybe$ allocations will be eliminated by further compiler transformations. The particular compiler transformation responsible for this is *case-of-case* transformation [SPJ'98]. We can show how this transformation works on this example by inlining more definitions:

$$
\begin{aligned}
&sumIter(next_{map}sq\ next)\ s_0\\
&=\ sumGo(next_{map}sq\ next)\ 0\ s_0\\
&=\ let\ go\ a\ s = case\ next_{map}\ sq\ next\ s\ of\\
&\qquad Nothing \to a\\
&\qquad Just(x,s) \to go(a+x)s\\
&\quad in\ go\ 0\ s_0
\end{aligned}
$$

$$
\begin{aligned}
&=\ let\ go\ a\ s = case(case\ next\ s\ of\\
&\qquad\quad Nothing \to Nothing\\
&\qquad\quad Just(x,s) \to Just(sq\ x,s))\\
&\qquad of\\
&\qquad Nothing \to a\\
&\qquad Just(x,s) \to go(a+x)s\\
&\quad in\ go\ 0s_0
\end{aligned}
$$

$$
\begin{aligned}
&=\ \{\textit{case-of-case}\ \text{transformation}\}\\
&let\ go\ a\ s = case\ next\ s\ of\\
&\qquad Nothing \to a\\
&\qquad Just(x,s) \to go(a+sq\ x)s\\
&\quad in\ go\ 0\ s_0
\end{aligned}
$$

Applying a step similar to the last one after inlining the definition of nested $unbuild$ we get a result:

$$
\begin{aligned}
&unbuild\ (\lambda next\ s \to (...))\ xs\\
&=\ (\lambda next\ s_0 \to (...))uncons\ xs\\
&=\ let\ go\ a\ s = case\ uncons\ x\ of\\
&\qquad Nothing \to a\\
&\qquad Just(x,s) \to go(a+sq\ x)\ s\\
&\quad in\ go\ 0\ xs
\end{aligned}
$$

$$
\begin{aligned}
&= \text{inlining the definition of uncons}\\
&= let\ go\ a\ s = case(case\ s\ of\\
&\qquad\quad [\,] \to Nothing\\
&\qquad\quad (x:s) \to Just(x,s))\\
&\qquad of\\
&\qquad Nothing \to a\\
&\qquad Just(x,s) \to go(a+sq\ x)\ s\\
&\quad in\ go\ 0\ xs
\end{aligned}
$$

$$
\begin{aligned}
&=\ \textit{case-of-case}\ \text{transformation}\\
&=\ let\ go\ a\ s = case\ s\ of\\
&\qquad [\,] \to a\\
&\qquad (x:s) \to go(a+sq\ x)\ s\\
&\quad in\ go\ 0\ xs
\end{aligned}
$$

Obviously this transformation has it's own prerequisites: it needs to clearly see how results of one pattern match are driven into another pattern match in order to join them. This works in example with *map*, but unfortunatelly, in case of *filter* they would not be fulfilled even for simple examples like summing only squares of odd numbers. In such example we will be able to get until last step with such code.

$$
\begin{aligned}
&sumOdd\ xs = sum(filter\ odd\ xs)\\
&= ... =\\
&= let\ go\ a\ s = case(let\ fnext\ s\ = case\ next\ s\\
&\qquad\qquad Nothing \to Nothing\\
&\qquad\qquad Just(x,s)|odd\ x \to Just(x,s))\\
&\qquad\qquad\qquad |\_ \to fnext\ s\\
&\qquad in\ fnext\ s)\\
&\quad of\\
&\quad Nothing \to a\\
&\quad Just(x,s) \to go(a+x)s\\
&in\ go\ 0\ xs
\end{aligned}
$$

In this example *case-of-case* optimization is not applicable, as one of the cases is recursive and there's no static way to inline the outer pattern match into inner one without creating a new nested pattern match.

EXPRESSIVENESS OF SHORTCUT FUSION SYSTEMS

The problems with the two shorcut fusion systems presented comes from the fact that restrict fusable functions by requiring that they are written in terms of predefined function combinators.

The *foldr/build* system allows great freedom in expressing producers, but we are restricted on how we write consumers. And while this is certainly true that *foldr* can express most of list consumers, is obviously fails for several commonly used ones (*zip, foldl*).

Situation with *unbuild/unfold* is symmetric, we are given freedom in writing consumers, but enforced to write producers in restrictive manner. Again, while indeed *unforldr* can express most of producer, it fails to do so for some commonly used operations, such as *filter* and *flatMap*.

There's one more limitation imposed by such setup. Those deforestation systems were developed only in attempt to eliminate intermediate data structures, not the ones that are used for storage. This difference can even be seen in names, as next system differentiates itself from systems that only support deforestation but also allowing fusion.

This limitation was a key motivation for Stream Fusion approach to be developed.

VI. Stream fusion

The idea in paper [Coutts'07] comes from extending the *unbuild/unfold* approach, by allowing the stepper function for both *unbuild* and *unfold* to report a *Skip*, meaning that in order to get next element more invokations are required. This removes need for the recursive definition of methods such as *filter*. Intuition comes from the example of *sumOdd*: if we introduce an additional value *Skip*, there will be no need to perform a recursive call in case of failed predicate check, and *case-of-case* will succeed.

We reformulate our representations with a new data type:

$$
\begin{aligned}
&data\ Step\ a\ s = Done \\
&\qquad\qquad\quad |Skip\ s \\
&\qquad\qquad\quad |Yield\ a\ s
\end{aligned}
$$

and we define a *Stream* using the new *Step* type:

$$data\ Stream\ a = \exists s.\ Stream\ (s\ \rightarrow Step\ a\ s)\ s$$

While stream is not a list, it is intended to be equivalent. We can define obvious conversion to and from lists:

$$
\begin{aligned}
&stream :: [a] \rightarrow Stream\ a \\
&streamxs = Stream\ uncons\ xs \\
&\quad where \\
&\qquad uncons[\,] = Done \\
&\qquad uncons(x : xs) = Yield\ x\ xs
\end{aligned}
$$

$$
\begin{aligned}
&unstream :: Stream\ a \rightarrow [a] \\
&unstream(Stream\ next\ s_0) = unfold\ next\ s_0 \\
&\quad where \\
&\qquad unfold\ next\ s = case\ next\ s\ of \\
&\qquad\quad Done \rightarrow [\,] \\
&\qquad\quad Skip\ s \rightarrow unfold\ next\ s \\
&\qquad\quad Yield\ x\ s \rightarrow x : unfold\ next\ s
\end{aligned}
$$

Now we can define other operations in terms of *Streams* that can skip:

$$
\begin{aligned}
&map :: (a \rightarrow b) \rightarrow Stream\ a \rightarrow Stream\ b \\
&map\ f(Stream\ next_0\ s_0) = Stream\ next\ s_0 \\
&\quad where \\
&\qquad next\ s = case\ next_0\ s\ of \\
&\qquad\quad Done \rightarrow Done \\
&\qquad\quad Skip\ s \rightarrow Skip\ s \\
&\qquad\quad Yield\ x\ s \rightarrow Yield(f\ x)\ s
\end{aligned}
$$

$$
\begin{aligned}
&foldl :: (b \rightarrow a \rightarrow b) \rightarrow b \rightarrow Stream\ a \rightarrow b \\
&foldl\ f\ a(Stream\ next\ s_0) = go\ a\ s_0 \\
&\quad where \\
&\qquad go\ a\ s = case\ next\ s\ of \\
&\qquad\quad Done \rightarrow a \\
&\qquad\quad Skip\ s \rightarrow go\ a\ s \\
&\qquad\quad Yield\ x\ s \rightarrow go(f\ a\ x)s
\end{aligned}
$$

Now, with the ability to skip, *filter* can be expressed without the need for recursion:

$$
\begin{aligned}
&filter :: (a \rightarrow Bool) \rightarrow Stream\ a \rightarrow Stream\ a \\
&filter\ p\ (Stream\ next_0\ s_0) = Stream\ next\ s_0 \\
&\quad where \\
&\qquad next\ s = case\ next_0\ s\ of \\
&\qquad\quad Done \rightarrow Done \\
&\qquad\quad Skip\ s \rightarrow Skip\ s \\
&\qquad\quad Yield\ x\ s | p\ x \rightarrow Yield\ x\ s \\
&\qquad\qquad\qquad |\_ \rightarrow Skip\ s
\end{aligned}
$$

As can be seen from this examples, ability to Skip allows to express stream producers without need for recursion. This is a key observation, that allows Stream fusion to win in comparison with *unfoldr/unbuildr* system.

Interestingly enough, the only rule required in system with such definitions is trivial:

$$unstream \cdot stream\ xs = xs$$

and this is enough for us to proceed with *sumOdd* example.

$$
\begin{aligned}
&sumOdd\ xs = sum(filter\ odd\ xs) \\
&= foldl\ (+)\ 0(stream(unstream(filter\ odd(stream\ xs)))) \\
&= foldl\ (+)\ 0(filter\ odd(stream\ xs)) \\
&= foldl\ (+)\ 0(filter\ odd(Stream\ uncons\ xs)) \\
&\quad where \\
&\qquad uncons[\,] = Done \\
&\qquad uncons(x : xs) = Yield\ x\ xs
\end{aligned}
$$

$$
\begin{aligned}
&= foldl\ (+)\ 0(Stream\ next\ xs) \\
&\quad where \\
&\qquad next\ s = case\ uncons\ s\ of \\
&\qquad\quad Done \rightarrow Done \\
&\qquad\quad Skip\ s \rightarrow Skip\ s \\
&\qquad\quad Yield\ x\ s | odd\ x \rightarrow Yield\ x\ s \\
&\qquad\qquad\qquad |\_ \rightarrow Skip\ s \\
&\quad uncons[\,] = Done \\
&\quad uncons(x : xs) = Yield\ x\ xs
\end{aligned}
$$

$$
\begin{aligned}
&= \text{inline } \textit{uncons} \text{ and perform } \textit{case-of-case} \text{ transformation}\\
&foldl\ (+)\ 0\ (Stream\ next\ xs)\\
&\quad where\\
&\qquad next\ s = case\ s\ of\\
&\qquad\quad [\,] \to Done\\
&\qquad\quad (x : s) | odd\ x \to Yield\ x\ s\\
&\qquad\qquad\quad | \_ \to Skip\ s\\
&= \text{inline definition of } \textit{foldl}\\
&go\ 0\ xs\\
&\quad where\\
&\qquad go\ a\ s = case\ next\ s\ of\\
&\qquad\quad Done \to a\\
&\qquad\quad Skip\ s \to go\ a\ s\\
&\qquad\quad Yield\ x\ s \to go(a + x)s\\
&\qquad next\ s = case\ s\ of\\
&\qquad\quad [\,] \to\ Done\\
&\qquad\quad (x : s) | odd\ x \to Yield\ x\ s\\
&\qquad\qquad\quad | \_ \to Skip\ s
\end{aligned}
$$

Stream fusion seems like a clear champion, as it can easily express *filter* as well as *foldl* and it can express operations that consume multiple sources (*zip*). Unfortunately this approach has it's own disadvantages. While previous two approaches can express and fuse *concatMap*(Scala equivalent is *flatMap*), in current state Stream fusion can not. There are other functions that cannot be optimized by stream fusion, as they return multiple streams: such functions are:

$$
\begin{aligned}
&unzip && :: [(a, b)] \to ([a], [b])\\
&splitAt && :: Int \to [a] \to ([a], [a])\\
&partition && :: (a \to Bool) \to [a] \to ([a], [a])\\
&span, break && :: (a \to Bool) \to [a] \to ([a], [a])
\end{aligned}
$$

Still, the biggest practical limitation of Stream fusion is that it's unable to fuse commonly used pattern with *concatMap*:

$$concatMap\ s(\lambda x \to Stream\ next\ b\ (f\ x))$$

The original paper [Coutts'07] proposes several rewrite rules that potentially solve this example, but neither of those rules can be written in current Glasgow Haskell Compiler rules syntax.

VII. Conclusions

From discussed systems, *foldr/buildr* system is the only incorporated in an industrial compiler. It is included in GHC and empowers list comprehensions. As can be seen from code samples writing producer function in this system is pretty straightforward, but writing consumers is a challenging task. For this system, there are no known fusable representations of list consumer functions such as *zip* and *foldl*. *Zip*'s cannot be effectively fused as this system does not support consuming multiple lists at the same time, while fusion of accumulating functions such as *foldl* yields higher-order code, that is slow. In order to solve those problem the *unfoldr/unbuildr* system was proposed, and while it provides flexibility in expressing consumers, both consuming multiple sources (*zip*) and consumer with accumulating parameters (*foldl*), it creates challenges when it comes to producers, most important one being *filter*.
The Stream fusion approach can be seen as a generalization of *unfoldr/unbuildr* technique, where producers can Skip from returning a value, this led to solving many problems of Shortcut fusion systems, such as simplifying code patterns and providing good ground for fusion between different operations on the same data source. Unfortunately, it also introduced complications in case when a function is both a producer and a consumer, and number of elements produced for every consumer is not statically limited. The most notorious function that Stream fusion fails to optimize is a commonly used *concatMap* function, that *foldr/buildr* system is able to improve.
Another problem, common to all systems, is that those systems are proposed as one-directional transformations that are not complete and rely on success of other compiler transformations to remove inefficiencies. If the later fails those systems can lead to degradation of performance, without any way of reporting it.

VIII. Further research, Scala perspective

While all of presented frameworks have functions that they fail to optimize, the inclusion of *foldr/buildr* system in GHC compiler already brings benefit to many use-cases, allowing programmers to use convenient patterns that, if compiled directly, would be highly inefficient. Scala would clearly benefit from existance of similar system. Though this is not a question of simply replicating the results gained from Haskell world on Scala. The two languages have different semantics: in Scala users are allowed to write side-effecting programs, and, as optimizations must not alter the execution semantics, special care has to be taken to maintain all observable effects. The presence of mutable collections makes the problem even more demanding.

A. *Fixing the* concatMap *problem*

The [Coutts'07] paper presented several approaches to fix the *concatMap* problem. Unfortunately, those rules were able to bring this barrier down can not be expressed in terms of GHC rewriting rules language. There's clear possibility of trying those rules in Scala, as in presence of Scala Macros [Burmako'13], the potential range of analysis that can be performed in Scala extends beyond the boundaries of simple local rewrite rules.

B. *Separate compilation*

Neither of presented systems work in separate compilation setting. None of discussed systems has a notion of fusing with code not known in compile time. But as Stream fusion already has notion of state of stepper function s this state can be passed between functions in a generic way, in order to

interleave execution of function generation a collection and a function consuming it. This is not even a work-in-progress, but rather an idea-in-progress. In order to illustrate this idea on a simple example, imagine we have two separately defined library functions

$$def\ mapSquare(xs : List[Int]) = xs.map(\_ * \_)$$
$$def\ sum(xs : List[Int]) = xs.reduce(\_ + \_)$$

And we have a user program, being compiled entirely separately from those library functions, but using them:

$$def\ example(xs : List[Int]) = sum(mapSquare(xs))$$

It seems productive to split the library function into fusable parts, that can be used for fusion: the first function will simply return the altered element, while the second one will also return its intermediate state

$$def\ mapSquare\$Iter(el : Int, state : Unit = ()) =$$
$$\quad el * e$$
$$def\ mapSquare(xs : List[Int]) =$$
$$\quad xs.map(mapSquare\$Iter)$$
$$def\ sum\$Init = 0$$
$$def\ sum\$Iter(el : Int, state : Int) =$$
$$\quad el + state$$
$$def\ sum\$Finalize(state : Int) =$$
$$\quad state$$
$$def\ sum(xs : List[Int]) =$$
$$\quad xs.foldLeft(sum\$Init)(sum\$It)$$

Then the user code can take advantage of existence of those 'fusion building blocks' and fuse with them instead of using integral methods.

$$def\ example\$Iter(el : Int, state : Int) =$$
$$\quad sum\$Iter(mapSquare\$Iter(el), state)$$
$$def\ example\$Finalize(state : Int) =$$
$$\quad sum\$Finalize(state)$$
$$def\ example\$Init = sum\$Init$$
$$def\ example(xs : List[Int]) =$$
$$\quad xs.foldLeft(example\$Init)(sum\$Iter)$$

As can be seen in this example, it seems that those 'fusion building blocks' can be maintained across usages. Intuition supports claim that methods can be split into 3 parts, passing state into each other without using it in any other way. This potentially allows the library to change it's underlying implementation without requiring user to recompile. That possibility feels natural in Scala world, but posed a problem for previously discussed approaches.

C. Graceful failure

All presented systems for Haskell define one directional transformations. If the optimization did not succeed at some point, compiler proceeds compilation of code in this intermediate state, potentially slowing down code due to introducing abstractions that were to be expected by successive phases. User can get slower code without any information messages issued by compiler.

As Scala Macros [Burmako'13] allow to preform speculative optimization, if failure is detected the optimization can be reversed, with issuing a compile time messages that can potentially can help user.

REFERENCES

[Burstall'77] R. M. Burstall and John Darlington. A transformation system for developing recursive programs. Journal of the ACM, 24(1):44–67, 1977. doi: 10.1145/321992.321996.

[Gill'94] A. J. Gill and S. L. Peyton Jones. Cheap deforestation in practice: An optimiser for Haskell. In Proc. IFIP, Vol. 1, pages 581-586, Hamburg, Germany, Aug 1994.

[Davis'87] Kei Davis. Deforestation: Transformation of functional programs to eliminate intermediate trees. Master's thesis, Programming Research Group, Oxford University, 1987.

[Wadler'90] Philip Wadler. Deforestation: transforming programs to eliminate trees. Theoretical Computer Science, 73(2):231–248, January 1990b. doi: 10.1016/0304-3975(90)90147-A

[Gill'93] Andrew Gill, John Launchbury, and Simon Peyton Jones. A short cut to deforestation. In Proceedings of the Conference on Functional Programming Languages and Computer Architecture (FPCA '93), pages 223–232. ACM, June 1993. doi: 10.1145/165180.165214.

[SPJ'01] Simon Peyton Jones, Andrew Tolmach, and Tony Hoare. Playing by the rules: rewriting as a practical optimisation technique in GHC. In Ralf Hinze, editor, Preliminary Proceedings of the Haskell Workshop (HW '01), pages 203–233, September 2001. Utrecht University technical report UU-CS-2001-23.

[Hutton'99] Graham Hutton. A tutorial on the universality and expressiveness of fold. Journal of Functional Programming, 9(4):355–372, July 1999. doi: 10.1017/S0956796899003500.

[Svenningsson'02] Josef Svenningsson. Shortcut fusion for accumulating parameters & zip-like functions. In Proceedings of the International Conference on Functional Programming (ICFP '02), pages 124–132. ACM, 2002. doi: 10.1145/581478.581491.

[SPJ'98] Simon Peyton Jones and Andre Santos. A transformation-based optimiser for Haskell. Science of Computer Programming, 32(1–3):3–47, September 1998. doi: 10.1016/ S0167-6423(97)00029-4.

[Coutts'07] Duncan Coutts, Roman Leshchinskiy, and Don Stewart. 2007. Stream fusion: from lists to streams to nothing at all. In Proceedings of the 12th ACM SIGPLAN international conference on Functional programming (ICFP '07). ACM, New York, NY, USA, 315-326. DOI=10.1145/1291151.1291199 http://doi.acm.org/10.1145/1291151.1291199

[Burmako'13] Eugene Burmako. 2013. Scala macros: let our powers combine!: on how rich syntax and static types work with metaprogramming. In Proceedings of the 4th Workshop on Scala (SCALA '13). ACM, New York, NY, USA, , Article 3 , 10 pages. DOI=10.1145/2489837.2489840 http://doi.acm.org/10.1145/2489837.2489840

[SPJ'03] Simon Peyton Jones et al. The Haskell 98 Language and Libraries: The Revised Report. Cambridge University Press, May 2003. ISBN 0-521-82614-4. URL http://www.haskell.org/definition/.