Higher Order Patterns for Rewrite Rules

Reinders, Jaro

DOI
10.1145/3677999.3678275

Publication date
2024

Document Version
Final published version

Published in
Haskell 2024

Higher Order Patterns for Rewrite Rules

Jaro Reinders
Delft University of Technology
Delft, Netherlands
J.S.reinders@tudelft.nl

Abstract

GHC's rewrite rules enable programmers to write local program transformation rules for their own functions. The most notable use case are fusion optimizations, which merge multiple traversals of a data structure into one and avoids allocation of intermediate structures. However, GHC's rewrite rules were too limited to express a rewrite rule that is necessary for proper fusion of the `concatMap` function in a variant of fusion called stream fusion.

We introduce higher order patterns as a simple extension of GHC's rewrite rules. Higher order patterns substantially broaden the expressiveness of rewrite rules that involve local variables. In particular, they enable the rewriting of `concatMap` such that it can be properly optimized. We implement a stream fusion framework to replace the existing fusion framework in GHC and evaluate it on GHC's benchmark suite. Our stream fusion framework with higher order patterns shows an average speedup of 7% compared to our stream fusion framework without it. However, our stream fusion framework is not yet able to match the performance of GHC's existing fusion framework.

Additionally, we show that our higher order patterns can be used to simplify GHC's existing mechanism for rolling back failed attempts at fusion and, at the same time, solve a problem that prevented proper optimization in one example program. However, evaluating it on GHC's benchmark suite shows a small regression in performance overall.

CCS Concepts: • **Software and its engineering** → *Compilers*; *Functional languages*; • **General and reference** → *Performance*; • **Theory of computation** → *Rewrite systems*; *Equational logic and rewriting*.

Keywords: Rewrite Rules, Program Transformation, Optimization, GHC, Haskell, Higher Order Matching, Fusion

Haskell '24, September 6–7, 2024, Milan, Italy

ACM ISBN 979-8-4007-1102-2/24/09
https://doi.org/10.1145/3677999.3678275

ACM Reference Format:
Jaro Reinders. 2024. Higher Order Patterns for Rewrite Rules. In *Proceedings of the 17th ACM SIGPLAN International Haskell Symposium (Haskell '24), September 6–7, 2024, Milan, Italy.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3677999.3678275

1 Introduction

Optimizing compilers are not perfect. In particular, many compilers struggle with optimizations across user-defined functions, especially those that are recursive.

Performance-minded programmers are therefore encouraged to write complicated special-purpose functions using primitive constructs manually instead of composing them out of many simple general-purpose functions. For example, a function that sums the first `n` square numbers may be written as one special-purpose function:

```
sumFirstNSquares n = go 0 1 where
  go s i | i > n = s
         | otherwise = go (s + (i * i)) (i + 1)
```

Or it can be written as the combination of `sum`, `map`, and `enumFromTo` functions:

```
sumFirstNSquares' n =
  sum (map (\x -> x * x) (enumFromTo 1 n))
```

The Glasgow Haskell Compiler (GHC) [4] aims to allow programmers to write code that is both readable and efficient by optimizing functions such as `sumFirstNSquares'` to be as efficient as the special-purpose `sumFirstNSquares` function. To achieve this, GHC uses an optimization called shortcut fusion, and in particular the `fold/build` flavor [5].[1]

The main insight that underlies this optimization is that we can write many functions that work on lists as a combination of `foldr` and `build`. The `build` function is designed to be an inverse to `foldr` such that applying a `foldr` to a `build` lets us cancel out both and thus save a traversal and avoid allocating an intermediate data structure.

The knowledge that `foldr` and `build` can be cancelled out is not built-in to GHC. Instead, it is accomplished through a generic mechanism that allows programmers to write their own local program transformations called rewrite rules [14].

Since the introduction of `fold/build` fusion, a new flavor of short-cut fusion, called stream fusion [1], has gained traction in the literature. Stream fusion is based on unfolds rather than folds, which gives it the ability to fuse functions with multiple lists as input, such as `zip`.

[1]We refer to GHC in a broad sense, including its implementation of the standard library "`base`".

However, stream fusion has not replaced `fold/build` fusion in GHC. One particularly important drawback — and perhaps the reason that `fold/build` fusion is still in use — is that it is not easy to fuse `concatMap` using stream fusion.

In its most general form, it is impossible to fuse `concatMap`. However, it is commonly used in a form that is similar to a nested loop, which could be fused efficiently, but detecting this special form has proven difficult. Using a rewrite rule to fuse `concatMap` seems like an obvious solution. However, it turns out GHC's rule matcher was not sufficiently powerful to express the required rewrite rule.

In this paper, we introduce higher order patterns to GHC's rewrite rule matcher which makes it possible for it to handle more complicated rewrite rules, such as the rewrite rule that is required to properly fuse `concatMap`.

Concretely, we make the following contributions:

- We explain higher order patterns in the context of GHC's rewrite rule matcher and describe how they can be implemented in GHC (Section 3).
- We use higher order patterns to implement the `concatMap` rewrite rule for stream fusion. We implement a stream fusion system using this rewrite rule in GHC and evaluate it on GHC's 'nofib' benchmark suite [12] (Section 4).
- We use higher order patterns to simplify rewrite rules for rolling back unfruitful `fold/build` fusion. We show that this enables better optimizations for some programs. We implement these changes in GHC and evaluate them on GHC's nofib benchmark suite (Section 5).

The idea of higher order patterns is not new (Section 6), but they had not been implemented in GHC's rule matcher. We show that higher order patterns are useful and that they can be implemented as a simple incremental enhancement of GHC's rule matcher.

2 Rewrite Rules for Short-Cut Fusion

Let us start by reviewing rewrite rules as introduced by Gill et al. and their application to short-cut fusion. We motivate them in Section 2.1, explain how they are used by GHC in Section 2.3, and in Section 2.4 we put particular emphasis on local variables, which play a central role in Section 3.

2.1 Motivation

Before diving into the details of rewrite rules, let us consider the context in which they are used.

Haskell [6] has relatively few built-in data structures (essentially only numbers and arrays). One of the principles of Haskell's design is that programmers should be able to define as much as possible by themselves. There is, of course, a standard library, that includes lists, but Haskell compilers generally do not need to have much special knowledge about these standard data structures.[2]

In particular, GHC's optimizer treats lists as it does any other algebraic data type. So, if we want to have additional optimization rules for lists, then it should be possible to use the same optimization mechanism for other user-defined algebraic data types. Rewrite rules are such a general optimization mechanism that works for any user-defined function, not just those functions that work on lists.

Furthermore, another design principle of GHC's optimizer is to prefer small local optimizations to complex global optimizations. These small local optimizations are much easier to understand, and their interplay can often achieve the same results as large complex optimizations. A drawback of this approach is that it cannot give strong guarantees: an optimization can get stuck halfway. However, large complex optimizations often have very specific conditions on when they apply.

For example, a precursor to short-cut fusion was Wadler's deforestation technique [16]. Deforestation was able to give strong guarantees, but only applied to programs of a very specific form, called *tree-less form*.

In the end, the combination of user-definable and simple local optimizations has prevailed in the form of rewrite rules in GHC. Rewrite rules power GHC's state of the art list fusion optimization and are also applicable in many other applications. For example, GHC itself uses rewrite rules internally to specialize polymorphic functions to particular types, replacing a general implementation with a type-specific optimization.

2.2 Anatomy of a Rewrite Rule

Rewrite rules are local program transformation rules that can be defined by programmers themselves, but what do they actually look like?

Let us start by considering a simple rewrite rule, which rewrites a polymorphic function into an equivalent function that is specialized to a particular type. For example, the `fromIntegral` function converts an integral numeric type to any compatible numeric type. If we know that this function is used to convert an `Int` to a `Float` we can rewrite it to a specialized function instead. We can capture that in the following rewrite rule:[3]

```
"fromIntegral/int2Float" fromIntegral = int2Float
```

This rewrite rule consists of three parts: it starts with the name inside quotation marks, then the core of the rule is an equation with a left hand side and a right hand side. The left

[2] Lists do have special syntax, but you could completely replace them with your own user-defined list type if you do not mind slightly more verbose syntax.

[3] The actual rewrite rules in GHC are more complicated to avoid having to define rewrite rules for every pair of types.

$$\frac{\texttt{f} \leftarrow \texttt{g} \qquad \texttt{x} \leftarrow \texttt{y}}{\texttt{f x} \leftarrow \texttt{g y}}\ \textsc{App}$$

$$\frac{\texttt{x} \leftarrow \texttt{y[v := u]}}{\texttt{(\u -> x)} \leftarrow \texttt{(\v -> y)}}\ \textsc{Lam}$$

$$\frac{}{\texttt{v} \leftarrow \texttt{v}}\ \textsc{Var}$$

$$\frac{\texttt{v templ} \qquad \text{flvs}(\texttt{x}) = \emptyset}{\texttt{v} \leftarrow \texttt{x}}\ \textsc{Templ-Var}$$

$$\frac{\texttt{v unfolds to y} \qquad \texttt{x} \leftarrow \texttt{y}}{\texttt{x} \leftarrow \texttt{v}}\ \textsc{Var-Unfold}$$

$$\frac{\text{flvs}(\texttt{y}) = \emptyset \qquad \texttt{x} \leftarrow \texttt{z}}{\texttt{x} \leftarrow \texttt{let v = y in z}}\ \textsc{Let-Float}$$

Figure 1. Basic matching rules used by GHC.

hand side is called the *template* and the right hand side is simply abbreviated to *RHS*.

Note that while the RHS may be an arbitrary Haskell expression, the template must be a top level function that may be applied to several arguments. In Section 2.3, we explain the reason for this limitation.

You might notice that in this case the template is `fromIntegral` without any special type annotation to make sure that it is really used to convert an `Int` to a `Float`. Type signatures are allowed, but they are not necessary because GHC does perform its normal type inference, and it makes sure that the type of the template matches the type of the RHS.

Rewrite rules can get more complicated. In particular, we might want to leave holes in the template that match any Haskell expression (of a suitable type). For example, the well-known fusion property of the `map` function states that sequentially mapping two functions over a list is the same as mapping the composition of those functions. In such a rule we want to be able to match any function and any input list. We can do that as follows:

```
"map fusion" forall f g xs.
  map f (map g xs) = map (f . g) xs
```

This rewrite rule has one additional component: the `forall` followed by a sequence of variable names. These variables are called the *template variables*. Template variables match any Haskell expression that occurs in their place.

There are some further details, but those are not relevant for this paper.

2.3 Matching Rewrite Rules

The primary use of rewrite rules is to find expressions in our programs which match the template and replace them by the RHS in the hope of optimizing our programs. To make it possible to find candidates efficiently, the template of a rewrite rule must always be a top level function which may be applied to one or more arguments. Given this constraint, finding match candidates is just a matter of finding occurrences of that top level function in the program.

Rewrite rules are written in Haskell proper, but matching happens at a point in the compilation pipeline where the program has been desugared into a small intermediate representation called Core. Rewrite rules are desugared as well, so, the matcher only has to cover a few cases, of which we have shown the six most important in Figure 1.

These matching rules prescribe how we can construct a valid match. The judgment $x \leftarrow y$ states that the template x matches the expression y. The meta-variables x, y, z, f, and g stand for expressions (including templates) and u and v stand for variables.

The first three rules are straightforward syntactic matching rules for the λ-calculus. In the Lam rule, we use the notation $[v := u]$ to mean the capture-avoiding substitution of all occurrences of the variable v by u. We omitted the syntactic matching rule for `case` expressions, because it is straightforward and too verbose to include here.

With only structural matching rules, we would be limited to finding exact matches of expression in our program. Often, we are instead interested in finding an expression where only parts match exactly and other parts can be arbitrary expressions. In such cases, we can use template variables, which match any expression.

The Templ-Var rule states that a variable v can match any expression x as long as v is a template variable (v templ) and x contains no free occurrences of variables that are local to the rewrite rule itself ($\text{flvs}(x) = \emptyset$). Such *free local variables* can occur if we have moved under a binder while matching, such as with the Lam rule.

When we find a valid derivation tree, and thus a match, we can read off the substitution of the template variables by looking at the all the uses of the Templ-Var rule. One detail we have omitted from the matching rules in Figure 1 is that GHC's rule matcher supports non-linear patterns which have more than one occurrence of the same template variable in the template. In such cases, we have to additionally check that all these occurrences are actually matched to the equal expressions.

The Var-Unfold rule gives us the ability to look past a variable in the target into its definition. When possible, GHC keeps track of the definition of each variable in its so-called "unfolding". So, if we encounter a variable in the target and the template is not that same variable or a template variable,

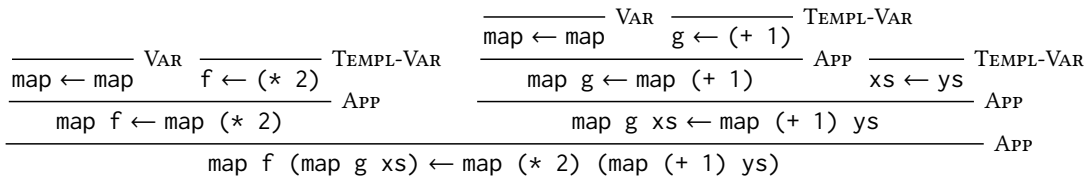

Figure 2. An example matching derivation tree.

then we can still continue matching using the unfolding of that target variable.

The Let-Float rule states we may discard `let` bindings as long as the right hand side of the binding does not contain free local variables. When applying the rewrite to a match that uses the Let-Float rule, we should add the discarded let back around the whole result. Note that the Let-Float rule does not affect the unfoldings

Both Var-Unfold and Let-Float should only be taken as a last resort. In particular, they might be applicable at the same time as the Var or Templ-Var rule, but they would unnecessarily modify the program in those cases.

For the sake of simplicity we omit four more rules which are present in GHC's rule matcher: ticks, types, coercions, and casts.

We can use the matching rules to build derivation trees. For example, given that f, g, and xs are template variables, we might want to show that the following judgment is valid:

```
map f (map g xs) ⟵ map (* 2) (map (+ 1) ys)
```

In Figure 2, we show that this is indeed a valid match, as witnessed by its derivation tree.

From the occurrences of the Templ-Var rule in this derivation tree, we can read off assignment of the template variables. In this way, we can see that the match succeeds with the substitution:

$$[f := \texttt{(* 2)}, g := \texttt{(+ 1)}, xs := \texttt{ys}]$$

We can apply this substitution to the RHS of the rewrite rule, which yields:

```
map ((* 2) . (+ 1)) ys
```

2.4 Local Variables

Templates may contain λ-abstractions and case expressions, and thus introduce variables that are local to the template. The naive matching algorithm described in the previous section cannot express many interesting rewrite rules that involve local variables.

For example, consider this rewrite rule, which fuses `unzip` with a particular use of `map`.

```
"unzip/map1" forall xs.
  unzip (map (\x -> (x, x)) xs) = (xs,xs)
```

This rule is limited to cases where the function we map is exactly `\x -> (x, x)`, which duplicates each element and produces a tuple. If the function we map produces a tuple that contains something other than exactly `(x, x)`, this rewrite rule will not match. This raises the question: can we write a more general rewrite rule that matches regardless of the contents of the tuple?

There are two ways in which we could try to write that generalized rule:

1. We can add extra template variables and use those as elements of the tuple in the template:

   ```
   "unzip/map2" forall a b xs.
     unzip (map (\x -> (a, b)) xs)
      = (map (\x -> a) xs, map (\x -> b) xs)
   ```

   However, the rule matcher will not match template variables to expressions that contain local variables, such as `x` in this case. Otherwise, we run the danger of moving variables out of their scope.
2. Alternatively, we could write a rule with two functions as template variables, which we then apply to `x` as follows:

   ```
   "unzip/map3" forall f g xs.
     unzip (map (\x -> (f x, g x)) xs)
       = (map f xs, map g xs)
   ```

   However, the rule matcher is more syntactic than we might hope in this case. It will only match literal function applications for the applications `f x` and `g x` in the template. In particular, it will no longer match with the original `\x -> (x, x)` we started with.

Neither of these approaches properly generalizes our original rule. In the next section, we introduce a simple extension to the rewrite rule matcher that enables us to write a rule that does properly generalize our original rule.

3 Higher Order Patterns

In this section, we address the shortcoming of rewrite rules that we have identified in Section 2.4. We introduce a new extension of rewrite rule matching that supports more rewrite rules, such as those required for stream fusion of `concatMap` which we will discuss in Section 4. Higher order patterns are not new (Section 6), but their application to program transformations for optimization in industrial compilers is new.

3.1 Revisiting `unzip/map`

In Section 2.4, we presented a rewrite rule for fusing `unzip` and a particular form of `map` and we discussed two potential generalizations that both fell short of actually generalizing the original rule. We now present a solution to that problem.

Firstly, we choose to build on `unzip/map3`, which uses template variables that represent functions that are applied to the local variables. This approach lets us avoid having to deal with scope issues.

Recall that we want to show that our generalized `unzip/map` rule matches at least the same expressions as the simple `unzip/map` rule. Formally, we would like the following judgment to be valid:

```
unzip (map (\x -> (f x, g x)) xs)
            ← unzip (map (\x -> (x, x)) xs)
```

To see what goes wrong, we apply matching rules until we get stuck, thus constructing the partial derivation tree of Figure 3.

We are stuck at the judgments `f x ← x` and `g x ← x`. No rule applies to these judgments, but there is a simple substitution of the template variables f and g that makes both sides semantically equivalent — namely, we can make them both identity functions:

```
[f := (\y -> y), g := (\z -> z)]
```

If we apply this substitution, we get `(\y -> y) x ← x` and `(\z -> z) x ← x`. In a single β-reduction step, we reach a valid judgment: `x ← x`. So, to support a rule that is a proper generalization of `unzip/map`, the rule matcher could assign the identity function to both the `f` and `g` template variables.

However, just recognizing this special case and using the identity function still does not allow the rule to match a tuple with any contents. Consider, for example, the target:

```
unzip (map (\x -> (x + 1, x * x)))
```

If we try to match the `unzip/map3` rule, we encounter the judgments `f x ← x + 1` and `g x ← x * x`. Remember that infix function application is syntactic sugar. In this case, it desugars to: `f x ← (+) x 1` and `g x ← (*) x x`, which makes it more clear which rules might be applicable.

The `f x ← (+) x 1` judgment is stuck, but again, we can substitute in a function for `f` such that a single β-reduction step yields a valid judgment. In this case, that would be the function `(\y -> (+) y 1)`.

The `g x ← (*) x x` judgment is more interesting because the App rule does apply, which would yield two new proof obligations: `g ← (+) x` and `x ← x`. The latter is obviously valid by the Var rule, but the former is problematic because the target contains the local variable `x` so the Templ-Var rule does not apply. So, we should not use the App rule in this case. Instead, we can again use a function, namely `(\y -> (*) y y)`, which makes the judgment valid after one β-reduction step.

Now, we are starting to see a common pattern.

3.2 Generalizing

In the previous section, we have seen several examples of stuck matching judgments that can be made unstuck by instantiating a template variable with a λ-abstraction and considering matching up to one β-reduction step. This puts us in the domain of higher order matching, which we discuss more in Section 3.3, but let us first continue describing our new matching rule, which we call the higher order pattern rule.

To match a higher order pattern, we look for a template variable applied to a local variable. We can assign a function to that template variable to make the match work. In fact, further experimentation would show that this works for template variables applied to any number of distinct local variables. Figure 4 shows the formal higher order pattern rule.

$$\frac{\mathtt{v}_1 \ldots \mathtt{v}_n \text{ local} \qquad \mathtt{v}_1 \ldots \mathtt{v}_n \text{ distinct} \quad \mathtt{f} \text{ templ} \qquad \mathrm{flvs}(\mathtt{x}) \subseteq \{\mathtt{v}_1, \ldots, \mathtt{v}_n\}}{\mathtt{f}\ \mathtt{v}_1 \ldots \mathtt{v}_n \leftarrow \mathtt{x}}\ \text{HOP}$$

Figure 4. The rule for matching higher order patterns.

The HOP rule is an extension of the Templ-Var rule. Instead of just a template variable, the HOP rule allows you to use an application of a template variable to one or more distinct local variables as the template. The local variables you give as arguments to the template variable specify the set of allowed local variables in the target expression.

Similar to the Templ-Var rule, the HOP rule should also be used to read off the substitution necessary to perform the final rewrite. In the case of the HOP rule, this is not completely straightforward.

When we see a match like the following:

$$\mathtt{f}\ \mathtt{v}_1 \ldots \mathtt{v}_n \leftarrow \mathtt{x}$$

Then the template variable `f` should be assigned the expression formed by λ-abstraction over all the local variables with the target `x` as the body:

$$[\mathtt{f} := \backslash \mathtt{v}_1 \ldots \mathtt{v}_n \ \texttt{->}\ \mathtt{x}]$$

Note that a λ-abstraction is added when rewriting a higher order pattern, which may impact performance, but we expect this to be insignificant compared to the potential performance gains of more expressive rewrite rules.

3.3 Uniqueness of Matching

A well known problem of higher order matching is the existence of multiple solutions. For example, the template `f x`, where both `f` and `x` are template variables, can yield multiple valid substitutions when matched against a target such as `0`

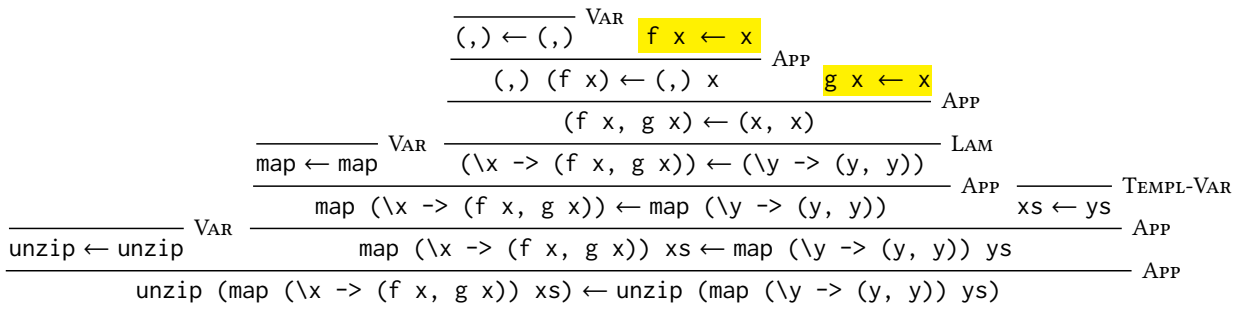

Figure 3. An incomplete matching derivation tree for the `unzip/map` rewrite rule. Note that tuple syntax, e.g. `(f x, g x)`, is sugar for application of the tuple constructor to two arguments, in this case: `(,) (f x) (g x)`.

(the 0 literal). We could make `f` the identity function and `x` the value 0:

```
[f := (\y -> y), x := 0]
```

Or we could make `f` the constant function that returns the value 0 (the value of `x` does not matter):

```
[f := (\_ -> 0)]
```

Having two equally valid matches is a problem, because, in the end, we need to choose one and we have no way of knowing what the user intended. Luckily, Miller [8, 9] has shown that higher order patterns, with the restriction that a template variable must be applied to distinct local variables, always produce at most a single match. Millers research was in the context of unification, which is a slightly different problem, but Nipkow has shown that the same holds for higher order patterns in rewrite systems [10, 11].

3.4 Backward Compatibility

The GHC compiler is widely used, so any changes that break compatibility with older versions incur a large cost to the community. We argue that our new higher order pattern rule is partially backward compatible, in the sense that all old rewrite rules that used to match continue to match with exactly the same result.[4]

Using the HOP rewrite rule as presented in Figure 4, this is not necessarily the case. There are situations in which the HOP rule and the App rule both apply. In such cases, we cannot simply say to prefer the App rule over the HOP rule because there are situations where the App rule applies but ultimately leads to a matching failure. We have seen such an example with the judgment `g x ⟵ (*) x x` in Section 3.1. So, to avoid changing the result of existing matches, we have to add an extra condition to the HOP rule:

$$x = g\ y \implies y \neq v_n \vee v_n \in \text{flvs}(g)$$

This condition captures the fact that the App rule only applies when the target is the application of a function to an argument, and then it can still fail in two ways: either arguments do not match, or the functions do not match. Since the argument in the template is the local variable v_n, it is easy to check for inequality. Furthermore, we know that the function in the template must itself be either a higher order pattern or a bare template variable. If we would choose the App rule then the only thing that changes is that the set of allowed free local variables shrinks. So it will only fail if the variable v_n is a free local variable of the function in the target.

In this way, we ensure that the HOP rule only applies when the App rule would lead to a matching failure.

Note, however, that our addition of the higher order pattern rule can cause existing rules to match more often. We could prevent this by adding special syntax to mark these higher order patterns, thus ensuring that old rules will be completely unaffected. However, this would complicate the implementation in GHC as it would require changing the rewrite rule parser and we deemed it unlikely that existing rewrite rules depend critically on avoiding a match in cases where the higher order pattern rule applies.

4 Case Study: Stream Fusion

The need for a more powerful rule matcher was first observed by Coutts et al. [1]. They found that the `concatMap` function is not fusible in general using stream fusion. Instead, a rewrite rule could be used to recognize cases where it can be fused and rewrite them to enable this fusion. However, the rule matcher was not powerful enough to write that rule.

In this section, we give a brief introduction to stream fusion in Section 4.1 and the problem of fusing `concatMap` in Section 4.2. Furthermore, we describe how we modified GHC to replace `fold/build` fusion with stream fusion in Section 4.3 and evaluate these changes on the standard `nofib` benchmark suite in Section 4.4.

4.1 A Crash Course on Stream Fusion

Stream fusion [1] is a form of short-cut fusion similar to `fold/build` fusion, but, whereas `fold/build` fusion is based

[4] Unless it now overlaps with a new match which uses the HOP rule, although we do not expect that to occur in practice.

on fusion of consumers, stream fusion is based on fusion of producers. Like `fold/build` fusion, stream fusion is generally applicable to inductive data types, but for the sake of simplicity we will consider only list fusion.

A stream is a pair of a state and a step function which can produce a new element, update the state, or signify the end of the stream:

```
data Stream a = ∃ s. Stream (s -> Step a s) s
data Step a s = Done | Yield a s | Skip s
```

It is straightforward to implement conversion functions to and from lists, but we omit their implementation for the sake of brevity:

```
stream :: [a] -> Stream a
unstream :: Stream a -> [a]
```

Using these conversion functions, we can define functions that operate on lists in terms of functions that operate on streams.

```
mapS :: (a -> b) -> Stream a -> Stream b
map :: (a -> b) -> [a] -> [b]
map f = unstream . mapS f . stream
```

If we compose two such functions and inline their definitions, we see that there is an intermediate step where the stream is converted to a list and immediately back to a stream:

```
unstream . mapS f . stream . unstream
         . mapS g . stream
```

We can avoid these redundant conversions with a simple rewrite rule:

```
"stream/unstream" stream . unstream = id
```

Applying this rule yields a pipeline that streams the input list, applies a sequence of stream transformations, and finally unstreams back into a list.

```
unstream . mapS f . mapS g . stream
```

At a glance, this might not seem inherently more efficient than the composition of two `map` functions that we started with. The difference is that the only recursive function in this pipeline is `unstream`, so GHC can inline all the other functions and apply further optimizations to optimize this whole pipeline to a single efficient loop. A detailed account of these further optimizations can be found in Section 7 of [1].

4.2 Fusing `concatMap`

So far, we have only considered fusion of a simple linear pipeline of list functions. The same approach extends to functions that join together multiple streams, such as `(++)` (append) and `zip`. However, there are also functions like `concatMap` that have a nested structure.

The `concatMap` function takes as input a list and a function that maps each element of that list to a new list. To avoid confusion, we will call the former list the *outer* list and the lists produced by the function *inner* lists. The result of `concatMap` is the concatenation of all the inner lists.

We can use the same approach of implementing the function on lists in terms of the function on streams and adding `stream` and `unstream` in the appropriate places:

```
concatMap :: (a -> [b]) -> [a] -> [b]
concatMap f xs =
  unstream . concatMapS (stream . f) . stream
```

If used properly, all the `stream` and `unstream` functions will be removed by the `stream/unstream` rewrite rule. However, a problem hides in the implementation of `concatMapS` (see Appendix B). Given a function that produces the inner streams from elements of the outer stream, it needs to produce the concatenation of all the inner streams. To achieve that, the whole inner stream, including its step function, needs to be part of the state of the output stream of `concatMapS`.

In this way, the step function of the inner stream is obfuscated to the compiler, which makes it hard for the compiler to perform the optimizations that stream fusion requires.

There are optimization techniques such as Call Pattern Specialization [13] and the Static Argument Transformation [15, Section 7.1], but these are both complicated and not universally applicable. This observation resonates with the drawbacks of complicated global optimizations for fusion, which we discussed in Section 2.1.

Instead, we should try to find a simple local transformation to perform this optimization, and that is exactly what we can use higher order patterns for. The problem is that the step function of the inner stream can change completely depending on the value of the elements of the outer stream. However, in practice, it seems common that the step function of the inner stream does not depend on the value of the elements of the outer stream. We can express that pattern as the following rewrite rule:

```
"concatMapS" forall next f.
  concatMapS (\x -> Stream (next x) (f x)) =
    concatMapS' next f
```

This rewrite rule was proposed by Coutts et al. [1], but it contains two higher order patterns that GHC's rules did not support at the time, so they could not use it.

The `concatMapS'` function used in this rule is very similar to the `concatMapS` function, except that it uses the same inner step function throughout the whole stream. The implementation of `concatMapS'` can be found in Appendix B.

4.3 Implementation

With the rewrite rule for `concatMap` in hand, we can start implementing stream fusion in GHC. Unfortunately, GHC is currently geared toward `fold/build` fusion, which shows up in several places. Ideally, we would perform an extensive

refactoring, however due to time constraints, we decided to focus on the most important parts.

Most of the basic list functions are located in the `ghc-internal` package in the module `GHC.Internal.Data.List`. There are some fundamental list functions, and in particular `foldr` and `build`, in the module `GHC.Internal.Data.Base`. Finally, the enumeration function for integers lives in the module `GHC.Internal.Data.Enum`. We have disabled the old `fold/build` fusion mechanism and associated rewrite rules, and we have placed stream-based alternatives where possible.

During early tests, we found that the rewrite rule for `concatMapS` could get inhibited by `let` or single-branch `case` expressions. The Let rule can sometimes float out `let` bindings when they are encountered during matching, but that is not possible when local variables are captured by the `let` binding. This confirms earlier findings by Farmer et al. [3] (Section 5) and we followed their advice of instead floating such bindings further inwards if floating out is not possible. This might duplicate work, but the performance gain of the rewrite rule seems to usually be worth it. They even propose a technique to deal with `case` expressions with multiple branches, but their utility is still an open question, so we did not try to implement that.

One final challenge was GHC's desugaring of list comprehensions, which is currently only efficient under `fold/build` fusion because it uses `build` and `foldr` directly. We have modified this desugaring to use `concatMap` instead, which will require more optimization steps, but it should produce the same optimized code under both `fold/build` fusion and stream fusion with our new higher order rewrite rule.

4.4 Evaluation

We have evaluated our implementation of stream fusion in GHC on the "Enum" and "Nested" example programs from [3] and the `three_partition` example from [14], and confirmed successful fusion by manually inspecting the Core output produced by our modified GHC.

Furthermore, we have evaluated our implementation of stream fusion with higher order patterns on GHC's standard "nofib" benchmark suite [12]. We have compared the results to two different baselines: the existing `fold/build` fusion system in GHC, and our new stream fusion framework without our higher order pattern rule.

4.4.1 Compared to `fold/build`. Our first comparison is to the existing `fold/build` system. The elapsed time increased on average by 7.99% (geometric mean) with a minimum of -53.0% (so a decrease in elapsed time) for the `spectral/knights` benchmark and a maximum of 282% for the `imaginary/gen_regexps` benchmark. The full elapsed time differences are shown in Figure 5. We have labeled the x-axis using numbers to save space, and refer the reader to Appendix A for the full names of all the nofib benchmarks. Note that the `imaginary/gen_regexps` (282%) and `imaginary/paraffins` (277%) benchmarks are off the charts.

The allocations also increased on average by 30.4% (geometric mean), with a minimum of -28.7% (a decrease in allocations) for the `spectral/fft2` benchmark and a maximum of 127000% for the `imaginary/x2n1` benchmark. The full allocation differences are shown in Figure 6. Several benchmarks are off the charts, so we itemize them here:

- `imaginary/x2n1` (127000%)
- `imaginary/queens` (1600%),
- `imaginary/wheel-sieve1` (1400%)
- `spectral/puzzle` (825%)
- `imaginary/gen_regexps` (605%)
- `imaginary/paraffins` (447%)
- `spectral/cryptarithm2` (359%)

These results show that stream fusion can cause a significant speedup for some benchmarks, but most benchmarks regress in performance.

4.4.2 Comparing the Effect of Our Higher Order Patterns. To measure the effect of our `concatMapS` rewrite rule that uses higher order patterns, we have also compared it to our stream fusion framework without the HOP rule. The elapsed time decreased on average by 7.63% (geometric mean), where the largest decrease was 61.0% for the `real/pic` benchmark, but the elapsed time did increase for some benchmarks up to 17.6% for the `real/parser` benchmark. The full elapsed time differences are shown in Figure 7.

The allocations also decreased on average by 13.3% (geometric mean), where the largest decrease was 71.7% for the `imaginary/rfib` benchmark, but for some benchmarks the allocations increased up to 49.1% for the `spectral/puzzle` benchmark. The full allocation differences are shown in Figure 8.

4.4.3 Threats to Validity. Replacing `fold/build` fusion with stream fusion turned out to be more of a challenge than we had hoped. Just implementing the rewrite rule for `concatMap` using higher order patterns was not sufficient to get proper fusion. During early experiments, we solved some issues which we described in Section 4.3. In this section, we list several more issues that we have not yet managed to solve.

Due to time constraints, we have not been able to perform an in depth analysis of the benchmark results.

We have only modified the three most primitive list modules and not derived functions elsewhere.

Furthermore, it has become common practice to write `foldr` in-line in functions and expect it to fuse, but under stream fusion, this is no longer a good practice. Any `foldr` in the standard library or in the benchmarks themselves may cause unnecessary performance regressions. Furthermore, stream fusion might open up new opportunities for fusion which we have not been able to take advantage of.

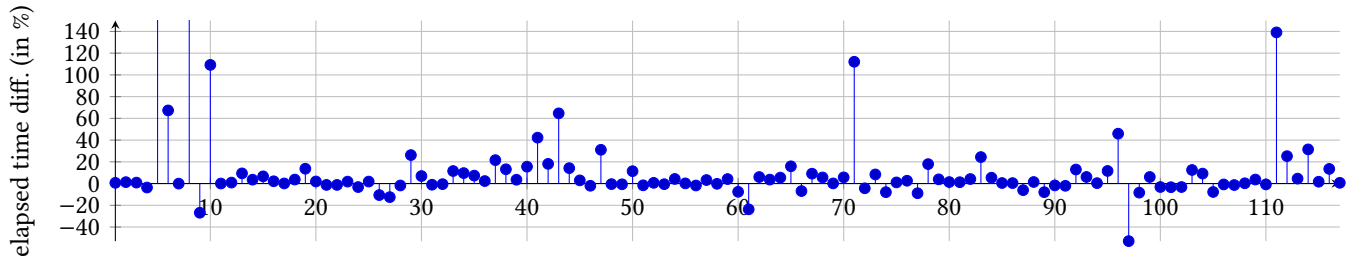

Figure 5. Percentage difference of elapsed time between the existing `fold/build` and our new stream fusion implementation. Lower is better. The X-axis is numbered, corresponding names can be found in Appendix A.

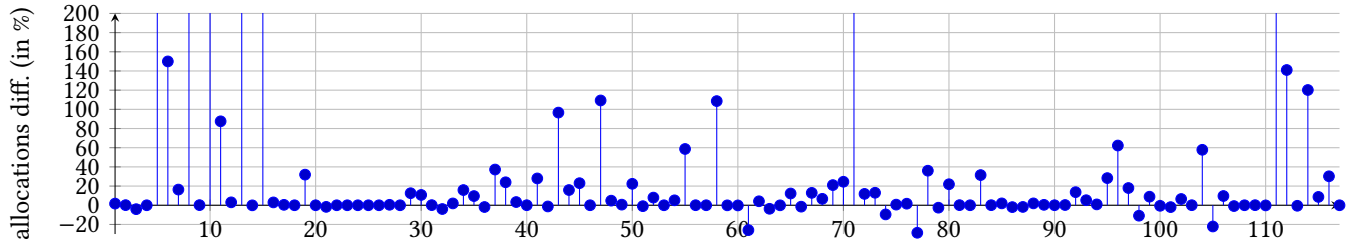

Figure 6. Percentage difference of allocations between the existing `fold/build` and our new stream fusion implementation. Lower is better. The X-axis is numbered, corresponding names can be found in Appendix A.

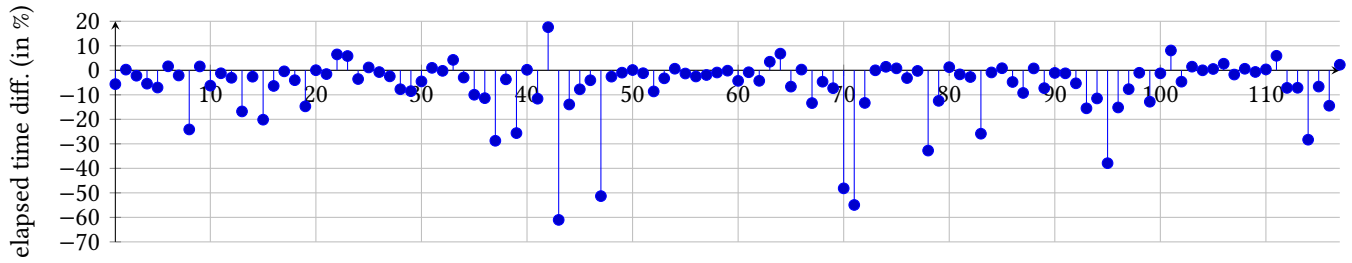

Figure 7. Percentage difference of elapsed time after enabling the `concatMap` rewrite rule. Lower is better. The X-axis is numbered, corresponding names can be found in Appendix A.

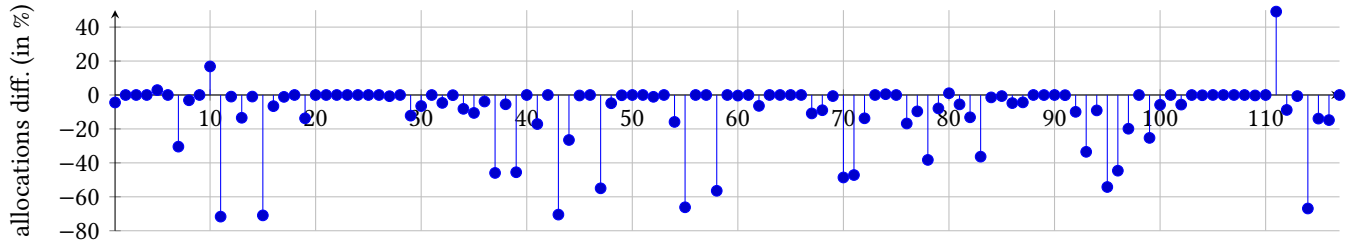

Figure 8. Percentage difference of allocations after enabling the `concatMap` rewrite rule. Lower is better. The X-axis is numbered, corresponding names can be found in Appendix A.

To get a fair comparison, we would have to inspect and potentially modify every function that processes lists. We did not have time to do this and thus leave it to future work 7.1.

Our new desugaring of list comprehensions uses `concatMap` so this will obviously not fuse under stream fusion without our new rewrite rule. We could instead try to desuger list comprehensions directly in terms of `unfoldr` when possible.

That would improve performance, but it would make the desugaring more complicated, and it would be limited to list comprehensions.

5 Case Study: Avoiding Rollback Markers

While our original motivation for introducing higher order patterns was to enable optimizing `concatMap` under stream fusion as discussed in Section 4, we have found another useful application of higher order patterns in rolling back fusion if it is unsuccessful. In this section, we explain how GHC currently rolls back failed fusion and how we can now use higher order patterns instead. In Section 5.1, we show a concrete example where our higher order pattern approach is better than GHC's current approach in Section 5.2, and we evaluate our approach on the nofib benchmark suite in Section 5.3.

5.1 Rolling Back Failed Fusion

If we write a program using `map (\x -> x + 1) ys` and it is not fully fused away, we might end up with an expression like this:[5]

```
foldr (\x xs -> (x + 1) : xs) [] ys
```

This program is already reasonably efficient, but the λ-abstraction does add some overhead, and it is simply larger than the original `map`. We have two options to optimize this further:

1. Inline `foldr` and simplify further, which yields:

   ```
   let go [] = []
       go (x:xs) = (x + 1) : go xs
   in go ys
   ```

   This removes the overhead of the λ-abstraction but it does make the code larger.
2. Rewrite this back to use `map (\x -> x + 1) ys`. This retains the λ-abstraction overhead but shrinks the code size to a minimum.

Today, GHC chooses the second option and rolls back the failed attempt at fusion to reduce the amount of code it generates.

However, we encounter a roadblock when we try to write a rewrite rule to handle this rollback. We can write a naive rule that matches our example exactly:

```
foldr (\x xs -> (x + 1) : xs) [] = map (+ 1)
```

Of course, this only covers the very specific case of adding one to each element. Even the tiniest deviation from this template, e.g. `1 + x`, will make the rule no longer match. This is the same problem as we encountered in Section 2.4.

Instead, GHC uses rollback markers, which are functions whose only purpose is to be detected by a rewrite rule in the future. If that does not happen, they are inlined away. The definition of `map` in terms of a rollback marker is as follows:

```
map f xs = build (\c n -> foldr (mapFB f c) n xs)
```

If fusion does not work out, i.e. `build` is inlined and `foldr` remains in the resulting code, an occurrence of map will be as follows:

```
foldr (mapFB f (:)) []
```

This pattern can be detected by a simple rewrite rule that does not involve local variables:

```
forall f. foldr (mapFB f (:)) [] = map f
```

This is how GHC's rollback mechanism currently works.

With higher order patterns, we can now avoid rollback markers and write a more direct rule:

```
forall f. foldr (\x xs -> f x : xs) [] = map f
```

5.2 Rollback Markers Can Hinder Optimization

An example where rollback markers hinder optimization is the `unlines` function, which takes a list of strings and concatenates it to one long string with newline characters between each of the original strings. The `unlines` function can be implemented as follows:

```
unlines xs = concat (map (++ "\n") xs)
```

However, if run without further optimizations, this will traverse each of the input strings twice: once to append the newline at the end and once to concatenate all the strings.

We would like this implementation of `unlines` to be optimized to be as efficient as a handwritten loop, which could look as follows:

```
unlines [] = ""
unlines (x:xs) = x ++ '\n' : unlines xs
```

Unfortunately, GHC does not manage to reach that point. At some point during the optimization, it will get stuck when the expression is as follows:

```
foldr (mapFB (++ "\n") (++)) [] xs
```

Here, the `mapFB` rollback marker keeps the two invocations of `(++)` separate, which prevents their fusion. From this point the compiler will inline `mapFB`, but it is too late for further fusion to happen.

If we use the higher order pattern rule for rolling back failed fusion, we will instead reach the following expression:

```
foldr (\y ys -> (y ++ "\n") ++ ys) [] xs
```

Which now can be easily fused into the more efficient form, which can be optimized further to the efficient handwritten loop that we wanted to reach:

```
foldr (\y ys -> y ++ ('\n' : ys)) [] xs
```

5.3 Evaluation

We have replaced the rollback markers for `map` and all variants of `zip` (including `zipWith` up to `zipWith7`) and confirmed the effectiveness of these modifications on the `unlines` example by inspecting the resulting Core.

[5] This could be directly due to defining `map` in terms of `foldr` or indirectly due to a rewrite rule that rewrites uses of `map` to `foldr`. GHC uses the latter approach.

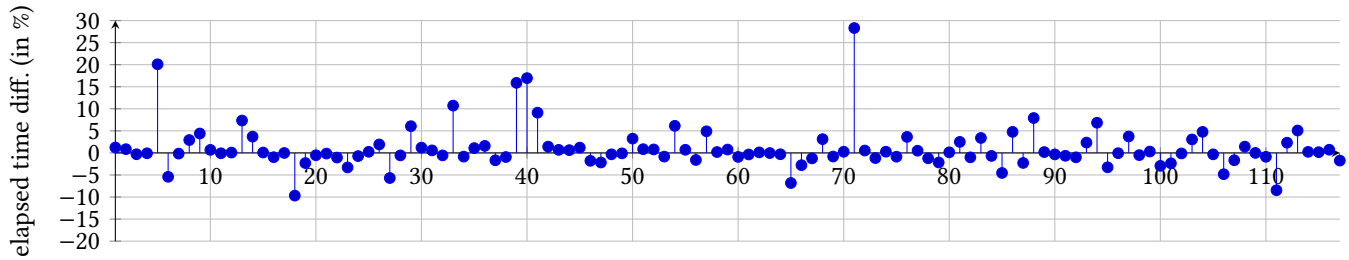

Figure 9. Percentage difference of elapsed time after changing rollback rules to use higher order patterns. Lower is better. The X-axis is numbered, corresponding names can be found in Appendix A.

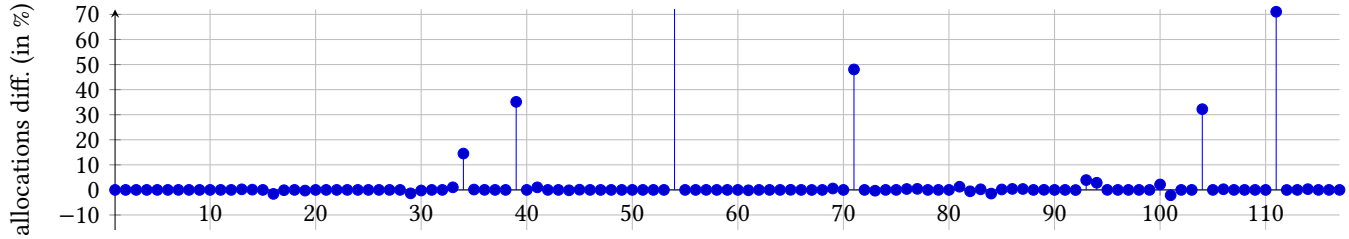

Figure 10. Percentage difference of allocations after changing rollback rules to use higher order patterns. Lower is better. The X-axis is numbered, corresponding names can be found in Appendix A.

Furthermore, we have evaluated our modifications on GHC's nofib benchmark suite [12]. The elapsed time increased on average by 0.43% (geometric mean), with a minimum of -15.9% (a decrease in elapsed time) for the `real/linear` benchmark and a maximum of 28.3% for the benchmark: `spectral/cryptarithm2`. The full elapsed time differences are shown in Figure 9. We have labeled the x-axis using numbers to save space and refer the reader to Appendix A for the full names of all the nofib benchmarks.

The allocations also increased on average by 1.73% (geometric mean), with a minimum of -35.1% (a decrease in allocations) for the `real/linear` benchmark and a maximum of 179% for the `shootout/k-nucleotide` benchmark. The full allocation differences are shown in Figure 10. Note that the `shootout/k-nucleotide` benchmark is off the charts in that figure.

This goes against our expectations because we know of a concrete case where the allocations and running time decreases, namely the `unlines` example. Further analysis of these results is necessary to explain why the results turned out this way.

6 Related Work

Fusion as an optimization in functional programming has a long history. Wadler [16] posed the problem of the needless allocation of intermediate data structures in compositional programs and proposed a custom optimization to eliminate it. Wadler's optimization did not become popular because it was complicated and restricted to certain forms of programs. Gill et al. [5] showed that a similar optimization, short-cut fusion, could be implemented using simple local program transformations, which apply fusion on a best effort basis without restricting the language that can be used to write programs. Later, Simon Peyton Jones et al. [14] incorporated a rewrite rule system into GHC, which allows programmers to define program transformations such as those required for short-cut fusion themselves. Finally, our work was directly inspired by the alternative stream-based short-cut fusion approach of Coutts et al. [1], which showed several potential advantages over Gill et al.'s `fold/build` fusion approach. In particular, it promises better fusion of the `zip` family of functions.

More recently, Kiselyov et al. [7] showed an approach to stream fusion using meta-programming techniques, which does give guarantees about fusion and supports a much larger language than Wadler's original deforestation approach. However, their approach does not automatically speed up any programs that work on lists. Instead, programmers need to explicitly use their streaming library. Furthermore, meta-programming often has syntactic overhead and can degrade compiler diagnostics.

More closely related to this paper is the work of Farmer et al. [3], who have also tackled the problem of `concatMap` stream fusion. They used their own external rewrite engine, including a specialized program transformation to convert suitable uses of `concatMap` to `concatMapS'`.

Miller first discovered higher order patterns in the context of unification in logic programming languages [8, 9].

Nipkow has applied Miller's higher order patterns to rewrite systems [10, 11] (and Nipkow came up with the name "higher order pattern" to the best of our knowledge). Our higher order patterns were independently developed by us, but they are essentially an application of the ideas of Miller and Nipkow to GHC's rewrite rule engine.

A seminal work on the topic of higher order matching is by de Moor and Sittampalam [2]. They develop a matching algorithm that can find higher order matches in general. They note higher order patterns as a special case, but leave investigation of how well their algorithm performs on higher order patterns as future work. As discussed in Section 3.3, in general higher order rules can yield multiple valid matches, which is not practical for rewrite rules in GHC.

7 Conclusion

In this paper, we extended GHC's rewrite rule matcher with higher order patterns. Our higher order patterns lift a restriction on GHC's rewrite rule matcher, which prevents it from matching template variables to expressions with local variables.

We used higher order patterns in a rewrite rule to optimize `concatMap` under stream fusion. We developed a basic stream fusion framework for lists and replaced GHC's current `fold/build` fusion with our stream fusion framework. Our results show that the `concatMap` rewrite rule, powered by higher order patterns, significantly improves the performance of stream fusion and, in some cases, stream fusion performs better than GHC's existing `fold/build` fusion framework. Unfortunately, our framework does not outperform the current `fold/build` fusion system overall.

We showed that higher order patterns can replace the rollback markers for the `map` and `zipWith` functions. We have implemented this in GHC and evaluated it on the standard nofib benchmark suite. Our results show improvements in some cases, but this too shows a performance regression overall, albeit less pronounced.

Nevertheless, we expect both applications of our higher order rewrite rules to have room for improvement.

7.1 Future Work

While we have shown that our `concatMap` rewrite rule with higher order patterns was able to significantly improve the performance of `concatMap` in our stream fusion framework, our implementation of stream fusion was still significantly slower than the existing `fold/build` fusion framework. We believe there are still many opportunities for improving other aspects of our implementation that are not directly related to the higher order patterns in rewrite rules. Further research is needed before we can draw conclusions about the relative performance of these approaches to fusion.

A form of stream fusion is already being used in practice in the `vector` library, however its benchmark suite does not contain programs that use `concatMap`, so we have not been able to meaningfully assess our higher order patterns on their library. Future research could create such a benchmark, perhaps adapting programs from nofib, and measure the performance of our higher order pattern rule for `concatMap`.

```
concatMapS :: (a -> Stream b)
           -> Stream a -> Stream b
concatMapS f (Stream next0 s0)
  = Stream next (s0, Nothing) where
    next (s, Nothing) = case next0 s of
      Done       -> Done
      Skip    s' -> Skip (s', Nothing)
      Yield x s' -> Skip (s', Just (f x))
    next (s, Just (Stream g t)) = case g t of
      Done -> Skip (s, Nothing)
      Skip t' -> Skip (s, Just (Stream g t'))
      Yield x t'
        -> Yield x (s, Just (Stream g t'))

concatMapS' :: (a -> s -> Step s b) -> (a -> s)
            -> Stream a -> Stream b
concatMapS' next1 f (Stream next0 s0)
  = Stream next (s0, Nothing) where
    next (s, Nothing) = case next0 s of
      Done       -> Done
      Skip    s' -> Skip (s', Nothing)
      Yield x s' -> Skip (s', Just (x, (f x)))
    next (s, Just (a, t)) = case next1 a t of
      Done       -> Skip    (s, Nothing)
      Skip    t' -> Skip    (s, Just (a, t'))
      Yield x t' -> Yield x (s, Just (a, t'))
```

Figure 11. Implementation of `concatMapS` and `concatMapS'`.

Acknowledgments

We would like to thank Simon Peyton Jones for helping us with the GHC proposal, the implementation in GHC, and encouraging us to write this paper. We would also like to thank Casper Bach Poulsen and the anonymous Haskell'24 reviewers for their valuable feedback and comments. Lastly, we would like to thank Sebastian Graf for pointing us to related work on pattern unification and Hugo Peters for discovering the example program where rollback markers hinder optimization (Section 5.2).

A Benchmark Names

Table 1 shows the names of all nofib benchmarks.

B `concatMapS` and `concatMapS'`

Figure 11 shows `concatMapS` and `concatMap'`.

Table 1. Names of all nofib benchmarks at the time of writing.

1	imaginary/bernouilli	25	real/eff/S	49	real/symalg	73	spectral/dom-lt	97	spectral/knights
2	imaginary/digits-of-e1	26	real/eff/VS	50	real/veritas	74	spectral/eliza	98	spectral/lambda
3	imaginary/digits-of-e2	27	real/eff/VSD	51	shootout/binary-trees	75	spectral/exact-reals	99	spectral/last-piece
4	imaginary/exp3_8	28	real/eff/VSM	52	shootout/fannkuch-redux	76	spectral/expert	100	spectral/lcss
5	imaginary/gen_regexps	29	real/fem	53	shootout/fasta	77	spectral/fft2	101	spectral/life
6	imaginary/integrate	30	real/fluid	54	shootout/k-nucleotide	78	spectral/fibheaps	102	spectral/mandel
7	imaginary/kahan	31	real/fulsom	55	shootout/n-body	79	spectral/fish	103	spectral/mandel2
8	imaginary/paraffins	32	real/gamteb	56	shootout/pidigits	80	spectral/gcd	104	spectral/mate
9	imaginary/primes	33	real/gg	57	shootout/reverse-compl.	81	spectral/hartel/comp_lab_zift	105	spectral/minimax
10	imaginary/queens	34	real/grep	58	shootout/spectral-norm	82	spectral/hartel/event	106	spectral/multiplier
11	imaginary/rfib	35	real/hidden	59	spectral/ansi	83	spectral/hartel/fft	107	spectral/para
12	imaginary/tak	36	real/hpg	60	spectral/atom	84	spectral/hartel/genfft	108	spectral/power
13	imaginary/wheel-sieve1	37	real/infer	61	spectral/awards	85	spectral/hartel/ida	109	spectral/pretty
14	imaginary/wheel-sieve2	38	real/lift	62	spectral/banner	86	spectral/hartel/listcompr	110	spectral/primetest
15	imaginary/x2n1	39	real/linear	63	spectral/boyer	87	spectral/hartel/listcopy	111	spectral/puzzle
16	real/anna	40	real/maillist	64	spectral/boyer2	88	spectral/hartel/nucleic2	112	spectral/rewrite
17	real/ben-raytrace	41	real/mkhprog	65	spectral/calendar	89	spectral/hartel/parstof	113	spectral/scc
18	real/bspt	42	real/parser	66	spectral/cichelli	90	spectral/hartel/sched	114	spectral/simple
19	real/cacheprof	43	real/pic	67	spectral/circsim	91	spectral/hartel/solid	115	spectral/sorting
20	real/compress	44	real/prolog	68	spectral/clausify	92	spectral/hartel/transform	116	spectral/sphere
21	real/compress2	45	real/reptile	69	spectral/constraints	93	spectral/hartel/typecheck	117	spectral/treejoin
22	real/eff/CS	46	real/rsa	70	spectral/cryptarithm1	94	spectral/hartel/wang		
23	real/eff/CSD	47	real/scs	71	spectral/cryptarithm2	95	spectral/hartel/wave4main		
24	real/eff/FS	48	real/smallpt	72	spectral/cse	96	spectral/integer		

References

[1] 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* (Freiburg, Germany) *(ICFP '07)*. Association for Computing Machinery, New York, NY, USA, 315–326. https://doi.org/10.1145/1291151.1291199

[2] Oege de Moor and Ganesh Sittampalam. 2001. Higher-order matching for program transformation. *Theoretical Computer Science* 269, 1 (2001), 135–162. https://doi.org/10.1016/S0304-3975(00)00402-3

[3] Andrew Farmer, Christian Hoener zu Siederdissen, and Andy Gill. 2014. The HERMIT in the stream: fusing stream fusion's concatMap. In *Proceedings of the ACM SIGPLAN 2014 Workshop on Partial Evaluation and Program Manipulation* (San Diego, California, USA) *(PEPM '14)*. Association for Computing Machinery, New York, NY, USA, 97–108. https://doi.org/10.1145/2543728.2543736

[4] The GHC Team. 2024. The Glasgow Haskell Compiler. https://www.haskell.org/ghc/

[5] Andrew Gill, John Launchbury, and Simon L. Peyton Jones. 1993. A short cut to deforestation. In *Proceedings of the Conference on Functional Programming Languages and Computer Architecture* (Copenhagen, Denmark) *(FPCA '93)*. Association for Computing Machinery, New York, NY, USA, 223–232. https://doi.org/10.1145/165180.165214

[6] The Haskell 2010 committee. 2010. *Haskell 2010 Language Report*. Technical Report. https://www.haskell.org/onlinereport/haskell2010/

[7] Oleg Kiselyov, Aggelos Biboudis, Nick Palladinos, and Yannis Smaragdakis. 2017. Stream fusion, to completeness. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages* (Paris, France) *(POPL '17)*. Association for Computing Machinery, New York, NY, USA, 285–299. https://doi.org/10.1145/3009837.3009880

[8] Dale Miller. 1991. A Logic Programming Language with Lambda-Abstraction, Function Variables, and Simple Unification. *Journal of Logic and Computation* 1, 4 (09 1991), 497–536. https://doi.org/10.1093/logcom/1.4.497

[9] Dale Miller. 1991. *Unification of simply typed lambda-terms as logic programming*. Technical Report MS-CIS-91-24. University of Pennsylvania Department of Computer and Information Science. https://repository.upenn.edu/handle/20.500.14332/7376

[10] Tobias Nipkow. 1991. *Higher-order critical pairs*. Technical Report UCAM-CL-TR-218. University of Cambridge, Computer Laboratory. https://doi.org/10.48456/tr-218

[11] Tobias Nipkow. 1993. Orthogonal higher-order rewrite systems are confluent. In *Typed Lambda Calculi and Applications*, Marc Bezem and Jan Friso Groote (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 306–317. https://doi.org/10.1007/BFb0037114

[12] Will Partain. 1993. The nofib Benchmark Suite of Haskell Programs. In *Functional Programming, Glasgow 1992*, John Launchbury and Patrick Sansom (Eds.). Springer, London, 195–202. https://doi.org/10.1007/978-1-4471-3215-8_17

[13] Simon Peyton Jones. 2007. Call-pattern specialisation for Haskell programs. In *Proceedings of the 12th ACM SIGPLAN International Conference on Functional Programming* (Freiburg, Germany) *(ICFP '07)*. Association for Computing Machinery, New York, NY, USA, 327–337. https://doi.org/10.1145/1291151.1291200

[14] Simon Peyton Jones, Andrew Tolmach, and Tony Hoare. 2001. Playing by the rules: rewriting as a practical optimisation technique in GHC. In *2001 Haskell Workshop* (2001 haskell workshop ed.). ACM SIGPLAN. https://www.microsoft.com/en-us/research/publication/playing-by-the-rules-rewriting-as-a-practical-optimisation-technique-in-ghc/

[15] Andre Santos. 1995. *Compilation by transformation for non-strict functional languages*. Ph. D. Dissertation. University of Glasgow (United Kingdom). https://theses.gla.ac.uk/id/eprint/74568

[16] Philip Wadler. 1988. Deforestation: Transforming programs to eliminate trees. In *ESOP '88*, H. Ganzinger (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 344–358. https://doi.org/10.1016/0304-3975(90)90147-A

Received 2024-06-03; accepted 2024-07-05