The HERMIT in the Stream

Fusing Stream Fusion's `concatMap`

Andrew Farmer
Information and Telecommunication Technology Center
The University of Kansas
http://andrewfarmer.name

Christian Höner zu Siederdissen
Department of Theoretical Chemistry, University of Vienna, Austria
Bioinformatics Group, Department of Computer Science, and Interdisciplinary Center for Bioinformatics
Universität Leipzig, Germany
choener@tbi.univie.ac.at

Andy Gill
Information and Telecommunication Technology Center
The University of Kansas
andygill@ittc.ku.edu

Abstract

Stream Fusion, a popular deforestation technique in the Haskell community, cannot fuse the $concatMap$ combinator. This is a serious limitation, as $concatMap$ represents computations on nested streams. The original implementation of Stream Fusion used the Glasgow Haskell Compiler's user-directed rewriting system. A transformation which allows the compiler to fuse many uses of $concatMap$ has previously been proposed, but never implemented, because the host rewrite system was not expressive enough to implement the proposed transformation. In this paper, we develop a custom optimization plugin which implements the proposed $concatMap$ transformation, and study the effectiveness of the transformation in practice. We also provide a new translation scheme for list comprehensions which enables them to be optimized. Within this framework, we extend the transformation to monadic streams. Code featuring uses of $concatMap$ experiences significant speedup when compiled with this optimization. This allows Stream Fusion to outperform its rival, foldr/build, on many list computations, and enables performance-sensitive code to be expressed at a higher level of abstraction.

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Processors—Code generation, Compilers, Optimization

General Terms Languages, Performance

Keywords Deforestation; Functional Programming; GHC; Haskell; Optimization; Program Fusion; Program Transformation; Stream Fusion

1. Introduction

In functional languages, it is often natural to implement sequence-processing pipelines by gluing together reusable combinators, such as $foldr$ and zip. These combinators communicate their results to the next function in the pipeline by means of intermediate data structures, such as lists. If these pipelines are compiled in a straightforward way, the intermediate structures adversely affect performance as they must be allocated, traversed, and subsequently garbage collected.

Many techniques, collectively known as *deforestation* [11, 30] or *fusion*, exist to transform such programs to eliminate these intermediate structures. Intuitively, rather than allow each combinator to transform the entire sequence in turn, the resulting code processes sequence elements in an assembly-line fashion. In many cases, after fusion, no sequence structures need to be allocated at all.

Shortcut (or algebraic) fusion works by expressing sequence computations using a set of primitive producer and consumer combinators, along with rewrite rules that combine, or fuse, consumers and producers. The three most well-known shortcut fusion systems, foldr/build [7], its dual unfoldr/destroy [27], and Stream Fusion [3], each choose a different set of primitive combinators and fusion rules. This choice determines which sequence combinators can be fused by each system[1]. We briefly summarize the tradeoffs here, though an excellent and thorough overview of the three systems can be found in Coutts [2].

The foldr/build system cannot fuse zip-like combinators which consume more than one sequence. It also cannot fuse consumers which make use of accumulating parameters, such as $foldl$, without a subsequent non-trivial arity-raising transformation [5]. Despite these shortcomings, GHC has used foldr/build to fuse list computations for 20 years in part because it performs well on nested sequence computations, such as $concatMap$, which are common in list-heavy code.

The unfoldr/destroy system fuses zip and $foldl$, but cannot fuse $filter$ or $concatMap$. Stream Fusion improves on unfoldr/destroy by fusing $filter$, but it still cannot fuse $concatMap$. Stream Fusion is currently the system of choice for array computations, which tend to heavily use zip, $foldl$, and $filter$.

This paper enhances Stream Fusion such that it handles $concatMap$. This enhancement removes a significant limitation which prevents Stream Fusion from replacing foldr/build as the fusion system of choice for GHC. We accomplish this by adding an additional rewrite rule which transforms calls to $concatMap$ into calls to a similar combinator, $flatten$, which is more amenable to fusion. GHC's current user-directed rewriting system, GHC RULES, cannot express this transformation, so we provide an implementation using HERMIT [4, 24], a GHC plugin for transforming GHC's intermediate language.

[1] There is a distinction between "fusion" and "fusion that results in an optimization". Fusion is only an optimization if it reduces allocation. Fusion may occur, but result in a function which allocates an internal structure equivalent to the eliminated sequence. In this paper, we only care about fusion that results in an optimization, and this is the meaning we intend when we say a particular system "can fuse" a given combinator.

PEPM '14, January 20–21, 2014, San Diego, CA, USA.
Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-2619-3/14/01…$15.00.
http://dx.doi.org/10.1145/2543728.2543736

While the transformation at the heart of this paper has been proposed previously, it has never been implemented in practice. The major contribution of this paper is to explore the practicality and payoff of implementing such a transformation and applying it to real Haskell programs. There are many details, especially regarding simplification and desugaring, that were not obvious when we set out. Specifically, this paper makes the following contributions.

- We describe a transformation from $concatMap$ to $flatten$ which enables fusion (Section 4). We extend this transformation to monadic streams (Section 4.2) so that it may be applied to vector fusion.
- We provide an implementation of the transformation as a custom GHC optimization, using HERMIT, and detail the simplifications necessary to enable the transformation in practice (Section 5).
- We give a novel translation scheme for list comprehensions which results in combinators which are amenable to fusion by our extended Stream Fusion system (Section 5.3).
- We apply our system to the **nofib**[19] suite of benchmark programs, demonstrating its advantage over foldr/build in list-heavy code (Section 6.2).
- We apply our system to the ADPfusion [12] library, which is used to write CYK-style parsers [10, Chapter 4.2] for single- and multi-tape grammars. The library makes heavy use of nested vector computations that need to be fused to achieve high performance. ADPfusion previously made extensive use of $flatten$. We demonstrate that our transformation simplifies the implementation of ADPfusion with no loss of performance (Section 7).

2. Stream Fusion

In this section, we summarize the Stream Fusion technique. Readers familiar with the topic may safely skip ahead, as none of this material is new. More detail can be found in [3] and [2].

The key idea of Stream Fusion is to transform a pipeline of recursive sequence processing functions into a pipeline of non-recursive stream processing functions, terminated by a single recursive function which “runs” the pipeline. The non-recursive functions are known as *producers*, if they produce a stream, or *transformers*, if they transform one stream into another. The recursive function at the end of the pipeline is known as the *consumer*.

The benefit of this transformation is that it enables subsequent local transformations such as inlining and constructor specialization, which are generally useful and thus implemented by the compiler, to *fuse* the producers and transformers into the body of the consumer, yielding a single recursive function which produces no intermediate data structures. Stream Fusion relies on a data type which makes explicit the computation required to generate each element of a given sequence:

```
data Stream a where
  Stream :: (s → Step a s) → s → Stream a
data Step a s = Yield a s | Skip s | Done
```

A Stream is a pair of a *generator function* ($s \rightarrow$ Step a s) and an existentially-quantified state (s). When applied to the state, the generator may give one of three possible responses, embodied in the Step type. Yield returns a single element of the sequence, along with a new state. Skip provides a new state without yielding an element. Done indicates that there are no more elements in the sequence. Generator functions are non-recursive, which allows them to be easily combined by GHC’s optimizer.

We convert to and from this Stream representation using a pair of representation-changing functions. In this section, we use Haskell lists as the sequence type, but the same technique works for other sequence types, such as arrays. The $stream$ function is a producer that converts a list to a Stream:

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

The state of $stream$ is the list of values to which it is applied. The generator function yields the head of the list, returning the tail of the list as the new state.

The $unstream$ function is a consumer that repeatedly applies the generator function to obtain the elements of the list:

```
unstream :: Stream a → [a]
unstream (Stream g s) = go s
  where go s = case g s of
                 Done       → []
                 Skip s′    → go s′
                 Yield x s′ → x : go s′
```

Using $stream$ and $unstream$, list combinators can now be redefined in terms of their Stream counterparts. Consider map:

```
map :: (a → b) → [a] → [b]
map f = unstream ∘ mapS f ∘ stream
mapS :: (a → b) → Stream a → Stream b
mapS f (Stream g s0) = Stream mapStep s0
  where mapStep s = case g s of
                      Done       → Done
                      Skip s′    → Skip s′
                      Yield x s′ → Yield (f x) s′
```

Note that $stream$ and $mapS$, as producer and transformer, respectively, are both non-recursive. Rather than traverse a sequence, $mapS$ simply modifies the generator function. Wherever the original stream would have produced an element x, the new stream produces the value $f\ x$ instead. Subsequent inlining and case reduction will fuse the two generators into a single non-recursive function.

The final, crucial, ingredient is the following rewrite rule, the proof of which can be found in Coutts [2]:

$$stream \circ unstream \equiv id$$

As an example of Stream Fusion in action, consider a simple pipeline consisting of two calls to map.

```
map f ∘ map g
```

Unfolding map yields the underlying stream combinators.

```
unstream ∘ mapS f ∘ stream ∘ unstream ∘ mapS g ∘ stream
```

Applying the rewrite rule eliminates the intermediate conversion.

```
unstream ∘ mapS f ∘ mapS g ∘ stream
```

Inlining the remaining functions, along with their generators, and performing standard local transformations such as case reduction and the case-of-case transformation [23] results in the following recursive function, which produces no intermediate lists.

```
let go []       = []
    go (x : xs) = f (g x) : go xs
in go
```

In this case, Stream Fusion has effectively implemented the $map\ f \circ map\ g \equiv map\ (f \circ g)$ transformation.

3. Fusing Nested Streams

The $concatMap$ combinator is a means of expressing nested list computations. It accepts a higher-order argument f and a list, which we call the *outer* list. It maps f over each element of the outer list, inducing a list of *inner* lists. It returns the concatenation of the inner lists as its result. Similiar to map in the previous section, $concatMap$ can be implemented in terms of its stream counterpart, $concatMapS$.

```
concatMap :: (a → [b]) → [a] → [b]
concatMap f = unstream ∘ concatMapS (stream ∘ f) ∘ stream
```

The $concatMapS$ function is a non-recursive transformer with a somewhat complicated generator function.

```
concatMapS :: (a → Stream b) → Stream a → Stream b
concatMapS f (Stream g s) = Stream g′ (s, Nothing)
  where
    g′ (s, Nothing) =
        case g s of
          Done      → Done
          Skip s′   → Skip (s′, Nothing)
          Yield x s′ → Skip (s′, Just (f x))
    g′ (s, Just (Stream g″ s″)) =
        case g″ s″ of
          Done      → Skip   (s, Nothing)
          Skip s′   → Skip   (s, Just (Stream g″ s′))
          Yield x s′ → Yield x (s, Just (Stream g″ s′))
```

The state of the resulting stream is a tuple, containing as its first component the state of the outer stream (the second argument to $concatMap$). Its second component is optionally an inner stream.

The generator function g' operates in two modes, determined by whether the inner stream is present in the state (Just) or absent (Nothing). When the inner stream is absent, g' applies the generator for the outer stream to the first component of the state. When this results in a value x, it constructs a new state by applying f to x to get the inner stream.

Subsequent applications of g' will see the Just constructor and operate in the second mode, which applies the generator for the inner stream to its state. When the inner stream is exhausted, it switches back to the first mode by discarding the inner stream state.

Optimizing $concatMapS$, GHC will use *call-pattern specialization* [21] to eliminate the Maybe type, yielding two mutually recursive functions, one for each mode. Unfortunately optimization stops before all Step constructors are fused away.

```
go1 acc s          = ... go2 acc s′ g″ s″ ...
go2 acc s g″ s″ = case g″ s″ of
                      Done      → go1 acc s
                      Skip s′   → go2 acc       s g″ s′
                      Yield x s′ → go2 (acc + x) s g″ s′
```

The problem is that the generator for the inner stream g'' is an argument to $go2$, and therefore not statically known in the body of $go2$. Indeed, we can see this in the original definition of $concatMapS$ above, where g'' is bound by pattern matching on the tuple of states. The fact that g'' is not statically known in $go2$ means it cannot be inlined, thwarting case reduction, which would have eliminated the Step constructors.

The code for g'' is statically known in $go1$. Additionally, we can see that $go2$ always repasses g'' unmodified on recursive calls. We could apply the *static-argument transformation* (SAT) [23] to $go2$ and inline the resulting wrapper into $go1$. This would make the code for g'' statically known at its call site, enabling full fusion.

This approach was suggested in the original Stream Fusion paper [3], but it involves a delicate interaction between call-pattern specialization and the SAT that is difficult to control. Aggressively applying the SAT can have detrimental effects on performance, so GHC is quite conservative in its use. In this case, GHC will not apply the SAT to $go2$ automatically. Even if GHC had a means of targeting the SAT via source annotation, the fact that $go2$ is generated by call-pattern specialization, at compile time, with an auto-generated name, means there is nothing in the source to annotate. Despite considerable effort by GHC developers, successfully applying this solution in the general case has remained elusive.

Stepping back, we can see that this is a consequence of the power of $concatMapS$ itself. The inner stream, including its generator function, is created by applying a function to a value of the outer stream *at runtime*. That function could potentially pick from arbitrarily many *different* inner streams based on the value it is applied to. Each of these streams may have an entirely different generator function. In fact, since the type of the state in a Stream is existentially quantified, the returned streams may not even have the same state type.

An alternative to $concatMapS$ is $flatten$. The type of $flatten$ makes explicit that the generator, and the type of the state, of the inner stream are always the same, regardless of the value present in the outer stream. This means that $flatten$ is readily fused by GHC.

```
flatten :: (a → s)            -- initial state constructor
        → (s → Step b s)    -- generator
        → Stream a → Stream b
```

The disadvantage is that $flatten$ is more difficult to use, as it breaks the abstraction of Stream by exposing the user to Step. Whereas the rest of the Stream Fusion system hides the complexity of state and generator functions from the programmer, providing familiar sequence combinators, $flatten$ requires one to think in terms of generator functions and state. A call to $concatMap$ with a complicated inner stream pipeline can make use of existing stream combinators, while $flatten$ requires the programmer to write a hand-fused, potentially complex generator function.

4. Transforming `concatMap` to `flatten`

In his dissertation, Coutts [2] proposes the following transformation for optimizing common uses of $concatMap$ by transforming them into calls to $flatten$. The advantage of such a transformation is its specificity. Rather than manage a brittle interaction between two general program transformations with potential negative performance consequences, we perform one specific transformation which we know to be advantageous. This is exactly the motivation for GHC RULES.

$$\forall\ g\ s\ .\ concatMapS\ (\lambda x \to \mathsf{Stream}\ g\ s) \Longrightarrow flatten\ (\lambda x \to s)\ g$$

This transformation is only valid if the state type and generator function of the inner streams are independent of the runtime values of the outer stream. That is, the state type and generator function are the same for each inner stream, and statically known. This restriction is exactly what allows the stream to be expressed in terms of $flatten$, and doing so makes this independence explicit.

While this transformation enforces the essential restriction that the value of x does not determine *which* generator and state is selected, it has the undesirable side condition that x cannot be free in g. This side condition severely limits the applicability of the transformation in practice. To see why this is a problem, consider this simple nested enumeration.

$$concatMapS\ (\lambda x \to enumFromToS\ 1\ x)\ (enumFromToS\ 1\ n)$$

As traditionally written, the generator for the inner $enumFromToS$ will necessarily depend on x in order to know when to stop generating additional values. The proposed transformation would fail to apply in this situation.

We could work around this by carefully defining $enumFromToS$ such that it stores its arguments in the stream state. That is, we could place an additional invariant on generator functions that they have no free variables that are not also free in their enclosing stream combinator definition. From a practical perspective, this complicates all stream combinator definitions for the benefit of $concatMap$. More complicated state types are required, which results in higher arity functions after call-pattern specialization, even when $concatMap$ is not present.

In this section, we define a more sophisticated transformation which separates these concerns, permitting g to use x to compute its result, without allowing x to determine which g is selected, and without requiring all stream combinators to be redefined with the additional invariant on their generators. Unfortunately, the GHC RULES system is incapable of expressing such a transformation.

4.1 Non-Constant Inner Streams

The principal limitation to the proposed transformation is the free variable check on the generator function. For any interesting use of $concatMapS$, this will fail. To lift this restriction, we alter the transformation such that it extends the state with the value of the outer stream. The generator function then has access to the value of the outer stream by way of the state. Note this transformation makes intentional use of variable capture (when x is free in g).

$$\begin{array}{l} \forall\ g\ s\ .\ \ concatMapS\ (\lambda x \to \mathsf{Stream}\ g\ s) \\ \qquad \Downarrow \\ \qquad flatten\ (\lambda x \to (x, s))\ (\lambda (x, s) \to fixStep\ x\ (g\ s)) \end{array}$$

Notice we have changed the type of the inner stream state. We can project the original state out of the extended state and apply the original generator, getting a Step result which contains a possible value and new state. This new state is of the *original* state type. We must return a state of the extended type. To do this, we employ $fixStep$ to place x back into the state held by the Step result, thereby lifting it to the extended state type.

$$\begin{array}{ll} fixStep :: a \to \mathsf{Step}\ b\ s \to \mathsf{Step}\ b\ (a, s) & \\ fixStep\ \_\ \mathsf{Done} & = \mathsf{Done} \\ fixStep\ a\ (\mathsf{Skip}\ s) & = \mathsf{Skip}\ (a, s) \\ fixStep\ a\ (\mathsf{Yield}\ b\ s) & = \mathsf{Yield}\ b\ (a, s) \end{array}$$

This improved transformation cannot be implemented as a GHC RULE because it requires manipulating syntactic language constructs such as case expressions. More practically, it is rare that the body of the function argument is in Head-Normal Form (i.e. starting with an explicit Stream constructor). Often the body will involve a call to another stream combinator instead. We use a custom simplification algorithm (described in Section 5) to expose the constructor. Expressing this transformation and applying it automatically during compilation is the main contribution of this paper.

4.2 Monadic Streams

The transformation we have described works on pure streams. The vector streams we target in Section 7 are parameterized on a monad, permitting generator and state construction functions to perform monadic effects. This leads to the following definition of the stream datatypes.

$$\begin{array}{l} \mathbf{data}\ \mathsf{Stream} :: (* \to *) \to * \to * \mathbf{where} \\ \quad \mathsf{Stream} :: (s \to m\ (\mathsf{Step}\ s\ a)) \to s \to \mathsf{Stream}\ m\ a \end{array}$$

$$\begin{array}{l} concatMapM :: \mathsf{Monad}\ m \\ \qquad\qquad \Rightarrow (a \to m\ (\mathsf{Stream}\ m\ b)) \\ \qquad\qquad \to \mathsf{Stream}\ m\ a \to \mathsf{Stream}\ m\ b \end{array}$$

$$\begin{array}{l} flattenM :: \mathsf{Monad}\ m \\ \qquad \Rightarrow (a \to m\ s) \\ \qquad \to (s \to m\ (\mathsf{Step}\ s\ b)) \\ \qquad \to \mathsf{Stream}\ m\ a \to \mathsf{Stream}\ m\ b \end{array}$$

The Stream constructor of the inner stream is now wrapped in a monadic context. The simplest such context is $return$.

$$concatMapM\ (\lambda x \to return\ (\mathsf{Stream}\ g\ s))$$

However, the monadic context may also have an arbitrary number of binds which scope over the inner stream. The transformation must collect the bound values and store them in the state, like it does for the outer stream binder x. Here we denote the monadic context as $\mathcal{M} \ll \ldots \gg$. Since the context is executed once per element of the outer stream, it can safely be moved to the state construction function of $flattenM$.

$$\begin{array}{l} \forall\ g\ s\ .\ \ concatMapM\ (\lambda x \to \mathcal{M} \ll \mathsf{Stream}\ g\ s \gg) \\ \qquad \Downarrow \\ \qquad flattenM\ (\lambda x \to \mathcal{M} \ll ((x, b_1 \mathinner{..} b_n), s) \gg) \\ \qquad\qquad\qquad (\lambda ((x, b_1 \mathinner{..} b_n), s) \to \\ \qquad\qquad\qquad\quad liftM\ (fixStep\ (x, b_1 \mathinner{..} b_n))\ (g\ s)) \end{array}$$

Instead of storing x in the state, we now store an n-ary tuple of x and the other binders. The projection is modified to project out of this tuple. As a minor optimization, we only store those binders which appear free in the generator function. Finally, $fixStep$ is lifted over the monadic result of the generator in the normal way.

5. Implementation

To implement the transformation, we use HERMIT [4, 24], a GHC plugin that allows the programmer to interact with GHC's intermediate representation of their program *during compilation*. GHC's intermediate language is called Core. Core is an implementation of System $F_C^{\uparrow}$ [26, 32], which is System F [8, 22] extended with let-bindings, constructors, type coercions and algebraic and polymorphic kinds. GHC's optimizer is written as a pipeline of Core-to-Core passes which manipulate this Core program.

HERMIT is a Core-to-Core pass that runs between GHC's existing passes, applying custom transformations which can be specified at a high level of abstraction using KURE, a strongly typed strategic rewriting domain-specific language [6, 25]. We took advantage of HERMIT's interactive capability to assist in developing and refining our transformation. We do not present code from the implementation here directly, but the interested reader can find it at: `https://github.com/ku-fpg/hermit-streamfusion`. While the HERMIT implementation necessarily operates on Core, for clarity we present the code in this section using Haskell syntax.

5.1 Simplifying

In practice, the body of the function passed to $concatMapS$ is not an explicit Stream constructor. In order to expose the constructor, we simplify the body with Algorithm 1. This simplification is done by HERMIT when it tries to apply our transformation. We explain each step in detail below.

1. Gentle simplification Perform dead-let elimination, case reduction, β-reduction, limited (non-work-duplicating) let substitution, and unfolds Haskell's operators for function composition ($\circ$) and application ($), as well as the identity function (id).

2a. Apply the* `stream/unstream` *rule. Recall the definition of $concatMap$ in terms of $concatMapS$ from Section 2.

$$\begin{array}{l} concatMap :: (a \to [b]) \to [a] \to [b] \\ concatMap\ f = unstream \circ concatMapS\ (stream \circ f) \circ stream \end{array}$$

Algorithm 1 Simplification Algorithm **SIMPLIFY**

In order, repeatedly apply the first transformation that succeeds.

```
1. Gentle simplification
2. Once, in a top-down manner:
   (a) Apply the stream/unstream rule.
   (b) Float a let inwards.
   (c) Eliminate a case.
   (d) Reduce a case on an inner stream.
   (e) Float a case inwards.
3. Unfold an application.
```

After transformation, f will be composed of stream combinators wrapped in an $unstream$ which turns the stream back into a list. When f is inlined, this $unstream$ will unite with $stream$, enabling us to eliminate both. If we were to instead unfold the $stream$ application in search of an explicit Stream constructor, we would commit to having an intermediate list. (Recall that the state type of a Stream produced by $stream$ is a list.)

2b. Float a let inwards. The inner stream will often be wrapped in let bindings, especially if we have unfolded a stream combinator. These bindings will scope over the entire Stream constructor, and may or may not depend on the value of the outer stream, so we cannot reliably float them outwards. We can observe, however, that they will never capture the Stream constructor itself, so we can reliably float them inwards past the constructor.

$$\begin{aligned}\textbf{let}\ b = e\ \textbf{in}\ \mathsf{Stream}\ g\ s \Longrightarrow\ & \mathsf{Stream}\ (\textbf{let}\ b = e\ \textbf{in}\ g)\\ & \qquad (\textbf{let}\ b = e\ \textbf{in}\ s)\end{aligned}$$

Floating inwards necessarily duplicates let bindings. This loss of sharing could result in duplicated allocation and computation. In practice, it appears rare that a let binding is used in both the generator and the state (the two arguments to the Stream constructor), so at least one will usually be eliminated by the next gentle simplification step, which, according to Algorithm 1, will occur directly after this transformation. In addition to applications, lets are floated into case expressions and lambdas.

2c. Eliminate a case. This step eliminates a case with a single alternative when none of the binders of the case are free in the right-hand side of the alternative.

$$\frac{\forall v\ \in \overline{vs}.\ \ v\ \notin \mathit{freeVars\ rhs}}{\begin{array}{ll}\textbf{case}\ e\ \textbf{of} & \Longrightarrow \mathit{rhs}\\ \quad \mathsf{C}\ \overline{vs} \to \mathit{rhs} & \end{array}}$$

One might question the necessity of this rule. This situation most often arises from simplification step 2e (float a case inwards), which itself is primarily caused by strictness annotations. Strictness annotations, and use of the Haskell seq function, are desugared to case expressions. (Operationally, case expressions in Core perform computation.) When we float a case inwards in 2e, we necessarily duplicate it (just as when we float lets inwards). Some of the duplicates may bind values that are never used.

2d. Reduce a case on an inner stream. Nested streams will result in a case expression on the inner stream. Consider:

$$\begin{aligned}& concatMapS\ (\lambda x \to concatMapS\ (\lambda y \to enumFromToS\ 1\ y)\\ & \qquad\qquad\qquad\qquad\qquad\qquad (enumFromToS\ 1\ x))\\ & \qquad\qquad (enumFromToS\ 1\ n)\end{aligned}$$

When transforming the outermost $concatMapS$, the body of the function argument will eventually be:

$$\begin{aligned}& \lambda x \to \textbf{case}\ flatten\ mkS\ g''\ (enumFromTo\ 1\ x)\ \textbf{of}\\ & \qquad \mathsf{Stream}\ g\ s \to \ldots\ \mathsf{Stream}\ g'\ s'\ \ldots\end{aligned}$$

GHC would have unfolded $flatten$ eventually. However, in order to transform the outer $concatMapS$ to $flatten$, we need to do it *now* so as to expose the Stream constructor in the right-hand side of the alternative.

A wrinkle arises if the head of the scrutinee is not a stream combinator. It may be a case expression, let expression, or application; or, more subtly, the $stream$ combinator itself.

$$\begin{aligned}& \lambda x \to \textbf{case}\ stream\ (unstream\ (flatten...))\ \textbf{of}\\ & \qquad \mathsf{Stream}\ g\ s \to \ldots\ \mathsf{Stream}\ g'\ s'\ \ldots\end{aligned}$$

If we were to unfold $stream$, the case would reduce, but we would fall prey to the same problem mentioned in step 2a. That is, we would be commited to a state type that included a list, missing a fusion opportunity.

These issues are all ones Algorithm 1 is designed to handle, so we recursively apply the entire algorithm to the scrutinee. When finished, it will yield an explicit Stream constructor, which will be reduced by the next gentle simplification step.

$$\frac{e \Longrightarrow e'}{\begin{array}{ll}\textbf{case}\ e\ \textbf{of} & \Longrightarrow \textbf{case}\ e'\ \textbf{of}\\ \quad \mathsf{Stream}\ g\ s \to \mathit{rhs} & \quad \mathsf{Stream}\ g\ s \to \mathit{rhs}\end{array}}$$

2e. Float a case inwards. Strictness annotations result in case expressions with a single, always matching default alternative. The right-hand side of the alternative may refer to the bound result of the evaluated scrutinee. For the same reasons we cannot always float lets outward, our only option is to eliminate the case or float it inwards.

We know that we cannot eliminate it wholesale, or step 2c would have done so. Eliminating it involves either changing the case to a let, which would make the program more lazy, or substituting the scrutinee into the right-hand side of the alternative, which could duplicate work. We instead choose to float it inwards.

$$\begin{array}{lll}\textbf{case}\ e\ \textbf{of} & \Longrightarrow & \mathsf{Stream}\ (\textbf{case}\ e\ \textbf{of}\ v \to g)\\ \quad v \to \mathsf{Stream}\ g\ s & & \qquad\quad (\textbf{case}\ e\ \textbf{of}\ v \to s)\end{array}$$

In most situations, one of these cases is eliminated, avoiding duplicated work, though we have not explicitly quantified how often this happens in practice. Within the context of the overall transformation, strictness remains unaltered. Nominally, if e is $\bot$, it is now possible to force the entire expression to the Stream constructor, where before it would have diverged. However, forcing just the Stream constructor without forcing one of its arguments provides no useful information, so only a pathological stream combinator would do so.

The other situation which requires this rule is the desugaring of pattern binders for list comprehensions (Section 5.3). The desugaring results in a case with a single alternative to bind the components of the pattern. Again, we float the case inwards, possibly duplicating computation.

3. Unfold an application. The last resort is to unfold a function application. It is crucial that this step be tried last, for the reasons mentioned in steps 2a and 2d. That is, we want to avoid unfolding the $stream$ combinator when possible, because it commits the program to a list state type, resulting in an intermediate list.

$$\begin{array}{lll}
\textbf{Entry} & \mathcal{TL}\ll comp\gg & \Longrightarrow \mathcal{TS}\ll\mathcal{TI}\ll comp\gg\gg \\[1em]
\textbf{Invariant} & \mathcal{TI}\ll[e \mid g, qs]\gg & \Longrightarrow [e \mid () \leftarrow [()], g, qs] \\
 & \mathcal{TI}\ll other\gg & \Longrightarrow other
\end{array}$$

$$\textbf{Guard}\qquad \frac{y \notin freeVars\ g}{\mathcal{TS}\ll[e \mid p \leftarrow xs, g, qs]\gg \Longrightarrow \mathcal{TS}\ll[e \mid p \leftarrow filter\ (\lambda y \rightarrow \textbf{case}\ y\ \textbf{of}\ p \rightarrow g)\ xs, qs]\gg}$$

$$\textbf{Bind} - \textbf{Irrefutable}\quad \mathcal{TS}\ll[e \mid p \leftarrow xs, qs]\gg \Longrightarrow concatMap\ (\lambda x \rightarrow \textbf{case}\ x\ \textbf{of}\ p \rightarrow \mathcal{TS}\ll[e \mid \mathcal{TI}\ll qs\gg]\gg)\ xs$$

$$\begin{array}{lll}
\textbf{Bind} - \textbf{Fail} & \mathcal{TS}\ll[e \mid \mathsf{C}\ y\ z \leftarrow xs, qs]\gg & \Longrightarrow concatMap\ (\lambda x \rightarrow \textbf{let}\ y = \textbf{case}\ x\ \textbf{of} \\
 & & \qquad\qquad\qquad\qquad \mathsf{C}\ y\ \_ \rightarrow y \\
 & & \qquad\qquad\qquad\qquad \_ \qquad \rightarrow error\ \texttt{"impossible"} \\
 & & \qquad\qquad\qquad z = \textbf{case}\ x\ \textbf{of} \\
 & & \qquad\qquad\qquad\qquad \mathsf{C}\ \_\ z \rightarrow z \\
 & & \qquad\qquad\qquad\qquad \_ \qquad \rightarrow error\ \texttt{"impossible"} \\
 & & \qquad\qquad\quad \textbf{in}\ \mathcal{TL}\ll[e \mid qs]\gg) \\
 & & \qquad (filter\ (\lambda x \rightarrow \textbf{case}\ x\ \textbf{of} \\
 & & \qquad\qquad\qquad\qquad \mathsf{C}\ \_\ \_ \rightarrow \mathsf{True} \\
 & & \qquad\qquad\qquad\qquad \_ \qquad \rightarrow \mathsf{False})\ xs)
\end{array}$$

$$\textbf{Let}\qquad \mathcal{TS}\ll[e \mid \textbf{let}\ B, qs]\gg \Longrightarrow \textbf{let}\ B\ \textbf{in}\ \mathcal{TL}\ll[e \mid qs]\gg$$

$$\textbf{Parallel}\qquad \frac{(x_1 \mathinner{..} x_n)\ \textit{is an n-ary tuple of values bound by}\ qs' \ \text{—}\ (y_1 \mathinner{..} y_n)\ \textit{is an n-ary tuple of values bound by}\ qs''}{\begin{array}{l}\mathcal{TS}\ll[e \mid qs' \mid qs'', qs]\gg \Longrightarrow concatMap\ (\lambda((x_1 \mathinner{..} x_n), (y_1 \mathinner{..} y_n)) \rightarrow \mathcal{TS}\ll[e \mid \mathcal{TI}\ll qs\gg]\gg) \\ \qquad\qquad\qquad\qquad\qquad (zip\ (\mathcal{TL}\ll[(x_1 \mathinner{..} x_n) \mid qs']\gg)\ (\mathcal{TL}\ll[(y_1 \mathinner{..} y_n) \mid qs'']\gg))\end{array}}$$

$$\textbf{Body}\qquad \mathcal{TS}\ll[e \mid]\gg \Longrightarrow [e]$$

Figure 1: Desugaring rules for list comprehensions. We denote the body as e, generators as xs and ys, guards as g, and remaining clauses as qs. The entry point is $\mathcal{TL}$, and multiple equations within a single named rule are matched top-down.

5.2 Multiple Inner Streams

We actually implement a slightly more general version of the transformation in step 2e. We float a case with multiple alternatives inwards if all alternatives share the same constructor and type.

$$\begin{array}{ll}
\textbf{case}\ e\ \textbf{of} & \Longrightarrow \mathsf{Stream}\ (\textbf{case}\ e\ \textbf{of} \\
\quad P \rightarrow \mathsf{Stream}\ g_1\ s_1 & \qquad\qquad P \rightarrow g_1 \\
\quad Q \rightarrow \mathsf{Stream}\ g_2\ s_2 & \qquad\qquad Q \rightarrow g_2) \\
 & \qquad\quad (\textbf{case}\ e\ \textbf{of} \\
 & \qquad\qquad P \rightarrow s_1 \\
 & \qquad\qquad Q \rightarrow s_2)
\end{array}$$

This has the effect of merging two streams whose state type is the same. The strictness argument is the same as for step 2e, though the resulting cases are less likely to be eliminated, so work duplication is an issue. It is easy to construct a small example which would benefit from this rule:

$$\begin{array}{l}
concatMapS\ (\lambda x \rightarrow \textbf{case}\ even\ x\ \textbf{of} \\
\qquad\qquad\qquad\qquad \mathsf{True}\ \ \rightarrow enumFromToS\ 1\ x \\
\qquad\qquad\qquad\qquad \mathsf{False} \rightarrow enumFromToS\ 1\ (x+1))
\end{array}$$

It remains unclear how often this happens in a real program, as we have not quantitatively measured how often the transformation has this effect. It does, however, suggest possible future work on a more involved means of merging streams.

5.3 List Comprehensions

Haskell offers convenient syntax for nested list computations in the form of list comprehensions. GHC desugars list comprehensions in two different ways. Standard Haskell 98 desugaring [20] is used when optimization is disabled or parallel list comprehensions are present. With standard optimizations enabled, comprehensions are desugared to compositions of $foldr$ and $build$ [5].

Neither scheme will result in good fusion for our extended Stream Fusion system, so we provide a third means of desugaring comprehensions (Figure 1). Haskell 98 desugaring, specifically of guards and pattern-match failures, will result in branching case expressions inside the function argument to $concatMap$. These case expressions cannot be merged because their state type differs, blocking our transformation. Our desugaring translation is a novel extension of Haskell 98 desugaring, and is required to get good fusion from our system. We have implemented it in GHC, selectively enabled by a flag. All tests in Section 6.2 enable it when running our optimization.

To refresh, a list comprehension has a body and a series of clauses. A clause can be a generator, a guard, or a let expression. Groups of clauses may be combined with a parallel operator, indicating a parallel list comprehension. Our goal is to desugar the clauses into a pipeline of list combinators over which we can subsequently guarantee fusion. To this end, we desugar generators to $concatMap$, guards to $filter$, parallel comprehensions to zip, and the body to a singleton list. Let expressions remain let expressions.

5.3.1 Guard Invariant

The rules for zips, lets, and binds are all standard. Traditionally, guards are desugared to boolean case expressions [20, 29].

$$\mathcal{TQ}\ll[\,e \mid g, qs\,]\gg \implies \textbf{case } g \textbf{ of} \\ \quad \mathsf{True} \to \mathcal{TQ}\ll[\,e \mid qs\,]\gg \\ \quad \mathsf{False} \to [\,]$$

The problem with this scheme is that we end up with a branching case expression in the body of the function we pass to $concatMap$. Recall that the stream state is existentially quantified. The empty stream in the False alternative will have a different state type than then stream in the True alternative, preventing us from floating the case inwards and merging the streams.

To get semantically equivalent behavior without a branching case expression, we desugar guards into calls to $filter$. In order to do this, we must have a list to filter. This brings about the invariant that guards are always preceeded by a generator. In the case that no generator is present, we insert a unit generator. The unit generator is a list that generates exactly one unit. Stepping through the translation:

$$\begin{array}{l} \mathcal{TL}\ll[\,e \mid g, qs\,]\gg \\ \Downarrow \\ \mathcal{TS}\ll[\,e \mid () \leftarrow [()], g, qs\,]\gg \\ \Downarrow \\ \mathcal{TS}\ll[\,e \mid () \leftarrow filter\ (\lambda() \to g)\ [()], qs\,]\gg \\ \Downarrow \\ concatMap\ (\lambda() \to \mathcal{TL}\ll[\,e \mid qs\,]\gg) \\ \qquad\qquad (filter\ (\lambda() \to g)\ [()]) \end{array}$$

This may seem convoluted, but the ()'s will eventually be discarded by call-pattern specialization. By avoiding a branching case expression which cannot be floated inwards, we enable our entire transformation.

5.3.2 Failing Pattern Matches

The translation for generators involves two rules. When the pattern is a variable, or only contains constructors of single-constructor data types, the pattern match cannot fail. In this case, we can translate directly to $concatMap$ using **Bind-Irrefutable**.[2] However, many patterns have the ability to fail. As an example, consider:

$[i \mid \mathsf{Just}\ i \leftarrow [\mathsf{Nothing}, \mathsf{Just}\ 6, \mathsf{Just}\ 7]]$

Desugaring this pattern match in the traditional way will result in a branching case expression with one alternative for the Just constructor and a second alternative for any other constructor. Once again, this will prevent fusion, as the two branches cannot be merged.

$$\begin{array}{l} concatMap\ (\lambda x \to \textbf{case } x \textbf{ of} \\ \qquad\qquad \mathsf{Just}\ i \to [i] \\ \qquad\qquad \mathsf{Nothing} \to [\,]) \\ \qquad [\mathsf{Nothing}, \mathsf{Just}\ 6, \mathsf{Just}\ 7] \end{array}$$

This leads us to the more complicated **Bind-Fail** translation for generators. In essence, we use $filter$ to create a list of elements which can *only succeed* when subsequently scrutinized by the case expressions in the let bindings. The let bindings themselves will be floated inward and/or eliminated by simplification. Again, this duplicates work (and allocation), but the effect is outweighed by the advantages of full fusion.

[2] Note that **Bind-Irrefutable** still uses a case expression with a single alternative to bind components of the pattern.

5.3.3 Modifying SIMPLIFY

Recall again the definition of $concatMap$ in terms of $concatMapS$.

$$\begin{array}{l} concatMap :: (a \to [b]) \to [a] \to [b] \\ concatMap\ f = unstream \circ concatMapS\ (stream \circ f) \circ stream \end{array}$$

The f resulting from desugaring will potentially have a single non-branching case expression to bind the pattern binders. The right-hand side of the alternative will consist of a pipeline of stream combinators ending in a single $unstream$ to turn the result into a list. The case itself will be wrapped in the $stream$ combinator in composition after f. For example:

$$\begin{array}{l} concatMapS\ (\lambda x \to stream\ (\textbf{case } x \textbf{ of} \\ \qquad\qquad\qquad \mathsf{C}\ \overline{vs} \to unstream\ (...))) \\ \qquad (...) \end{array}$$

In order to bring $stream$ and $unstream$ together to enable their elimination by the fusion rule, we augment **SIMPLIFY** with an extra step after gentle simplification which floats the case *outwards* if and only if it enables the rule to fire. This is the only change to the transformation specifically required by list comprehensions.

5.4 Call-Pattern Specialization

Stream Fusion depends crucially on call-pattern specialization to eliminate the constructors in the stream state. GHC allows a data type to be annotated to indicate that it should try to aggressively specialize functions with arguments of the annotated type. In the following example, $sPEC$ is the dummy argument of such an annotated type, and should force GHC to specialize go. Unfortunately GHC tends to bind deeply nested constructors and float them outwards, defeating specialization.

$$\begin{array}{l} \textbf{let}\ s_1 = (,)\ (\mathsf{Left}\ (\mathsf{Left}\ (\mathsf{Left}\ (\mathsf{Just}\ (\mathsf{I\#}\ 3)))))\ \mathsf{Nothing} \\ \quad\ \ go = \ldots\ go\ \ldots \\ \textbf{in}\ \ go\ sPEC\ (\mathsf{I\#}\ 0)\ s_1 \end{array}$$

To counter this, immediately before call-pattern specialization runs, we collect all the non-recursive bindings in the program whose heads are non-recursive data constructors and inline them unconditionally. In practice, this simple heuristic offers a huge improvement in the form of decreased allocation, because specialization completely eliminates the constructors.

5.5 The Plugin

In order to use our transformation on real code, we create a GHC plugin using HERMIT. The plugin consists of two modules.

The first module provides the definitions of the Stream Fusion combinators and the $fixStep$ function (Section 4.1), along with GHC RULES which transform list combinators into their Stream Fusion counterparts. A user wishing to enable Stream Fusion must import this module in their program so the definitions are available during compilation. The need for this module is primarily a HERMIT limitation (though the same limitation applies to any GHC plugin). Future versions of HERMIT may be able to inject such dependencies directly.

Our plugin follows the standard Stream Fusion approach to inlining and forcing call-pattern specialization. Where our plugin differs from tradition is the rewrite rules it provides for list combinators. Traditional Stream Fusion provides an alternative list prelude, in which list combinators were defined directly in terms of stream combinators. We instead provide GHC RULES such as this one:

$$\forall\, f\,.\ \ map\ f \equiv unstream \circ mapS\ f \circ stream$$

Our motivation for the RULES approach is two-fold. First, there is no need to hide or redefine the list combinators in the Prelude. This would be less onerous if Stream Fusion were the default list implementation, but makes selective use more difficult. With this approach, a single extra module import in the code targeted for optimization is all that is required.

Second, and more importantly, the desugaring of list comprehensions requires GHC to assign globally unique identifiers to the combinators it generates, so it can generate proper names even if the combinators themselves are not visible to the compiler. Desugaring directly to stream combinators, or even the alternative prelude combinators, would require the combinator and module names to be hard-coded into GHC itself. A change to the Stream Fusion library would require a corresponding change to GHC. On the other hand, the standard list combinators are stable and not likely to change, and are already assigned unique names by GHC. Desugaring to these standard combinators, then rewriting with RULES, allows us to make minimal changes to GHC itself.

The second module provides the actual GHC plugin, defined using HERMIT. This module is the one specified to GHC using the `-fplugin` flag when compiling the target code. The fact that list combinators already have RULES defined for foldr/build is a complicating factor. Before the first optimization pass runs, we apply all our rules to exhaustion in order to transform all the list combinators before their own RULES get a chance to fire. This is the primary source of increased compilation time, as HERMIT is currently not tuned for performance. Then, before and after every optimization pass, we apply our transformation whenever possible. Finally, before the call-pattern specialization pass, we aggressively inline non-recursive constructor bindings, as discussed in Section 5.4.

To summarize, the programmer must do two things to make use of our transformation. First, she must add a single additional import to the module targeted for optimization. Second, she must pass two additional flags to GHC, specifying the plugin name and module to be targeted.

6. Performance

In this section, we evaluate the performance benefits of the transformation by applying it to both Haskell lists and vectors. We first benchmark the results of the optimization on several microbenchmarks that exercise different aspects of the transformation (Section 6.1). We then apply the transformation to GHC's `nofib` benchmark suite (Section 6.2). Finally, we illustrate how $concatMap$ can sometimes lead to *better* performance than $flatten$ (Section 6.3).

All measurements in this section were performed on a 64-bit 2.3Ghz Intel Core^{TM} i5-2415M, with 4GB RAM, running OS X 10.7.5 and GHC HEAD (as of October 7, 2013).

6.1 Micro-benchmarks

In order to illustrate the performance gap between $flatten$ and $concatMap$, and thus characterize the potential benefit of the transformation, we apply it to the micro-benchmarks listed below. Note that these particular benchmarks characterize *best case* improvements, as they are designed to result in tight loops on unboxed integers.

The graph in Figure 2 summarizes the results. For each benchmark we provide the following measurements, where appropriate:

- **concatMap** Use the `vector` library's $concatMap$ combinator. This represents the current status quo for Stream Fusion.
- **flatten** Use `vector`'s $flatten$ combinator and hand-written generator functions.
- **Optimized** Apply our transformation to **concatMap**.
- **List** Use lists and apply foldr/build.

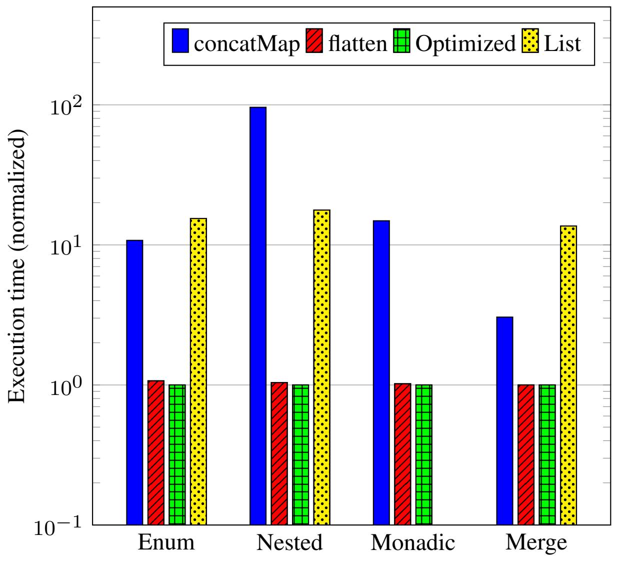

Figure 2: Micro-benchmark performance results.

Enum This benchmark characterizes the potential speedup gained by using $flatten$ instead of $concatMap$, and demonstrates that our optimization can fully close the gap.

$$
\begin{aligned}
f\ n = foldl'\ (+)\ 0\ (&concatMap\ (enumFromTo\ 1) \\
&\qquad (enumFromTo\ 1\ n))
\end{aligned}
$$

Nested This benchmark demonstrates the advantage foldr/build normally has on nested list computations. Note that lists outperform vectors despite being slower in **Enum**. In this case, superior fusion for lists is overcoming vector's normally superior data structure.

$$
\begin{aligned}
f\ n = &foldl'\ (+)\ 0 \\
&(concatMap\ (\lambda x \rightarrow concatMap\ (\lambda y \rightarrow enumFromTo\ y\ x) \\
&\qquad\qquad\qquad\qquad (enumFromTo\ 1\ x)) \\
&\qquad (enumFromTo\ 1\ n))
\end{aligned}
$$

Monadic This benchmark exercises the monadic stream transformation. As lists are not parameterized over a monad, they are absent from this comparison.

$$
\begin{aligned}
f\ n = runST\ (&\mathbf{do}\ vec \leftarrow getVector \\
&\quad foldl'\ (+)\ 0 \\
&\qquad (concatMapM\ (\lambda x \rightarrow \mathbf{do} \\
&\qquad\qquad z \leftarrow readVector\ vec\ x \\
&\qquad\qquad return\ (enumFromTo\ 1\ z)) \\
&\qquad\qquad (enumFromTo\ 1\ n)))
\end{aligned}
$$

Merge This benchmark involves merging two streams with the same state type, as discussed in Section 5.2.

$$
\begin{aligned}
f\ n = &foldl'\ (+)\ 0 \\
&(concatMap\ (\lambda x \rightarrow \mathbf{case}\ (odd\ x)\ \mathbf{of} \\
&\qquad\qquad \mathsf{True} \rightarrow enumFromTo\ 1\ x \\
&\qquad\qquad \mathsf{False} \rightarrow enumFromTo\ 2\ x) \\
&\qquad (enumFromTo\ 1\ n))
\end{aligned}
$$

6.2 Nofib Suite

In order to evaluate our transformation on real Haskell programs, we apply it to a subset of GHC's **nofib** benchmarking suite [19]. The **nofib** suite is the standard by which other GHC optimizations are measured before inclusion in the compiler.

Program	Allocs	Runtime	Elapsed	TotalMem
bernouilli	-6.4%	+0.0%	+0.0%	+0.0%
exp3_8	+0.0%	-1.3%	-1.2%	+0.0%
gen_regexps	-44.7%	-45.4%	-45.4%	-56.7%
integrate	-61.3%	-48.2%	-42.6%	+0.0%
kahan	+0.0%	+1.2%	1.4%	+0.0%
paraffins	+129.9%	-11.6%	-12.6%	-29.1%
primes	+2.2%	-15.2%	-15.9%	-33.3%
queens	-17.1%	-7.5%	-6.9%	+0.0%
rfib	+0.0%	+0.0%	+0.0%	+0.0%
tak	+0.0%	-4.7%	-4.7%	+0.0%
wheel-sieve1	+195.2%	+4.6%	+4.7%	-29.6%
wheel-sieve2	-0.6%	-1.2%	-2.4%	+0.0%
x2n1	-77.4%	+0.0%	+0.0%	+0.0%
Min	-77.4%	-48.2%	-45.4%	-56.7%
Max	+195.2%	+4.6%	+4.7%	+0.0%
Geom. Mean	-9.9%	-15.2%	-14.5%	-13.8%

Figure 3: Nofib performance comparison between foldr/build and Stream Fusion with our transformation. Negative figures indicate an improvement over foldr/build.

We target the "imaginary" subset of the suite in this paper. This limits the number of necessary stream combinator implementations (aside from $concatMapS$ and *flattenS*) to those used by this subset. HERMIT itself has not had any performance tuning, so compilation time for the smaller programs is more manageable. These are practical engineering issues that can be solved in a more mature fusion system, and are unrelated to the $concatMap$ transformation itself.

Figure 3 summarizes the results for the selected programs. In summary, programs experience ≈15% speedup over foldr/build on average, with an ≈10% reduction in allocation.

The gains for `bernouilli`, `integrate`, and `x2n1` are due to Stream Fusion itself, and our transformation provides no additional benefit. The `integrate` program features no calls to $concatMap$ at all, but makes heavy use of the $zipWith$ combinator. Similarly, `x2n1` is a true micro-benchmark, consisting of a single mapping operation inside a strict left fold. Though we do not show the results here for brevity, Stream Fusion (without our $concatMap$ transformation) performs significantly worse than foldr/build on each of the other programs. Allocation increases by 45.7% on average, and 836.7% in the worst case. Runtime is equivalent on average, but increases by 66.1% in the worst case. This is the penalty Stream Fusion pays on $concatMap$-heavy code. Our system always outperforms Stream Fusion alone.

The `gen_regexps` program is an ideal case for our system, as it makes heavy use of both $foldl$ and $concatMap$. Previous systems could fuse one of these combinators, but our system can fuse both.

Programs which make use of explicit recursion on lists tend to be slowed by our system. Both `paraffins` and `wheel-sieve1` make use of functions which explicitly accept and return lists, and are also recursive. These recursive functions will not be inlined by GHC, preventing the $stream$ and $unstream$ combinators from coming together and being eliminated. This results in many extra conversions between lists and streams. The resulting allocation is high, even if gains elsewhere improve runtime. To solve this, Stream Fusion systems typically include extra RULES to "back out" unfused stream combinators at the end of the optimization process, converting them back to their list counterparts. Our system currently has no such RULES, but they could be added.

These initial results are promising, and demonstrate the viability of the approach. We plan to apply the optimization to the full `nofib` suite in due course. HERMIT performance needs to be improved to reduce compilation time, currently 5-20× that of `ghc -O2`.

n	*flat*	*cmap*			
		sf	ratio	sf + c→ f	ratio
5000	9.7ms	100.3ms	10.3x	8.8ms	0.91x
10000	39.3ms	395.6ms	10.1x	35.0ms	0.89x
20000	155.9ms	1603.8ms	10.3x	139.3ms	0.89x

Figure 4: Optimizing equivalent stream pipelines for the `vector` package.

6.3 Performance Advantages of `concatMap`

Somewhat surprisingly, $concatMap$ can provide performance advantages over $flatten$. When using $flatten$, the programmer must carefully consider low-level performance issues because they are, in essence, writing a hand-fused inner-loop. By using $concatMap$, we can instead construct our inner loop from existing stream combinators, which presumably are already efficiently implemented. In this case, modularity makes it easier to get good performance.

To illustrate, we benchmark a pair of equivalent `vector` pipelines (one using $concatMap$, the other $flatten$) and examine the resulting Core. Using list combinators, the specification of our pipeline is:

```
spec :: Int → Int
spec n = foldl (+) 0 [i | x ← [1 .. n], i ← [1 .. x]]
```

The `vector` code is morally equivalent, though we have added strictness and elected to use `vector`'s $enumFromStepN$ in place of $enumFromTo$ in order to reflect the sort of code a user of the `vector` library would write in practice.

```
cmap :: Int → Int
cmap n = foldl′ (+) 0
              (concatMap (λ(!x) → enumFromStepN 1 1 x)
                         (enumFromStepN 1 1 n))
flat :: Int → Int
flat !n = foldl′ (+) 0
             (flatten mkS stp Unknown
               (enumFromStepN 1 1 n))
  where
    mkS !x = (1, x)
    stp (!i, !max)
       | i ⩽ max    = Yield i (i + 1, max)
       | otherwise = Done
```

We first benchmark these functions with standard Stream Fusion optimizations (without our transformation) using `criterion`[18] to establish a baseline. As we can see in Figure 4, *cmap* is consistently 10x slower than *flat*. Examining the Core reveals that *flat* results in a single wrapper function to unbox the input and box the result, along with a tight recursive worker on unboxed integers.

Generally, uses of $concatMap$ result in residual Step constructors, indicating fusion is incomplete. GHC actually manages to fuse away the Step constructors in *cmap*, as this code is very simple. However, the resulting inner loop involves both boxed integers and a tuple argument.

Applying our $concatMap$ optimization results in nearly equivalent performance. In fact, *cmap* is now consistently 11% *faster* than *flat*! Examining the Core for the inner loop of each function reveals why. The bodies of the loops consist only of tail calls on unboxed integers, but the bounds test in the inner loop is different. In *cmap*, the iterator is compared to zero, whereas in *flat*, the iterator is compared to a max bound.

Indeed, this behavior is exactly what we wrote in the generator function for *flat*. We can change our generator to implement the same algorithm as $enumFromStepN$, resulting in comparisons to zero.

```
step (!i, !max)
  | max > 0 = Yield i (i + 1, max − 1)
  | otherwise = Done
```

However, this actually makes *flat* 18× slower! Examining the Core again, we can see that the inner loop now involves Integer arguments, a clue to what is happening. The result type of *flat* only constrains the type of i. Since the state of the stream is existentially quantified, and max is no longer compared to i, the type of max is defaulted to Integer rather than Int. Ascribing a type fixes the problem, resulting in *flat* being as fast as *cmap*.

```
step (!i, !max)
  | max > (0 :: Int) = Yield i (i + 1, max − 1)
  | otherwise = Done
```

The fact that the programmer must consider such performance issues each time she writes a generator function is exactly what makes $flatten$ burdensome to use. Using *concatMap* (properly optimized) allows us to take advantage of the hard work done by the `vector` library writers, who have heavily optimized their enumerating stream producer. Thus, in practice, transforming *concatMap* to $flatten$ can result in better performance than direct uses of $flatten$ itself.[3]

7. ADPfusion

ADPfusion [12] is a library designed to simplify the implementation of complex dynamic programming algorithms in Haskell. It targets both single- and multi-tape linear and context-free grammars. ADPfusion can be used to implement complex parsing problems, such as weakly synchronized grammars for machine translation [1] in computational linguistics or interacting ribonucleic acid structure prediction as considered by Huang et al. [14]. As ADPfusion employs CYK-style parsing [10] that lends itself well to a low-level "table-filling" implementation, the resulting programs will perform close to equivalent `C` code, while being implemented at a much higher level of abstraction.

ADPfusion makes index calculations implicit. Production rules are combined by a small set of combinators, producing a parsing function from an index to a set of (co-)optimal parses. Lifting index calculations from explicit manipulations by the user to combinators makes it less likely that bugs appear, while the type system keeps evaluation functions and production rules in sync.

Questions of how to develop algorithms like these lead to the development of an algebraic framework that allows users to "multiply" dynamic programming algorithms in a meaningful way [13]. The resulting algorithms (grammars) are naturally of the multi-tape variety and the grammar definitions call for an automated embedding in an efficient framework. ADPfusion is used as the target DSL in this case to give efficient code.

Using `concatMap` instead of `flatten`

The availability of *concatMap* helps control the complexity of ADPfusion's underlying Stream Fusion framework by simplifying the design of specialized (non-)terminal symbols for formal grammars.

[3] A more recent build of GHC eliminates the 11% disparity in the original code. We believe this to be the result of new boolean primitive operations which were added after this section was written. While this specific example is no longer strictly true, we keep it because it is illustrative of the general problem with optimizing `flatten` by hand. This issue accounts for the performance gains made by `concatMap` relative to `flatten` in Section 7.

To understand how ADPfusion uses $flatten$, consider the parses for the production rule $S \rightarrow SS$, given in set notation:

$$
\begin{aligned}
{}_iS_j \rightarrow \{(S^{ik}, S^{lj}) \mid\ & k \leftarrow \{i\ \ldots\ j\} \\
, & S^{ik} \leftarrow {}_iS_k \\
, & l \leftarrow \{k\} \\
, & S^{lj} \leftarrow {}_lS_j\}
\end{aligned}
$$

That is, partial parses are generated from left to right for each production rule. All parses, except the final one, make use of the $flatten$ combinator to extend the current stream of partial parses and current index state with the parses for the current symbol. As all indices are already fixed when considering the right-most symbol in a production rule, only a single parse is generated in such a case (denoted $l \leftarrow \{k\}$).

This explanation assumes that, for fixed indices (say ${}_iS_k$), a non-terminal produces only a single parse. When only a single optimal result is required, this is actually the desired behaviour. In cases, where co- or sub-optimal parses are required non-terminals produce multiple results, thereby requiring an *additional* $flatten$ operation for each non-terminal, leading to the full notation above.

Thus, the $flatten$ function is used extensively in ADPfusion. Each new (non-)terminal requires up to two $flatten$ operations. Symbols on the right-hand sides of production rules admit multiple parses. Nesting further, in multi-tape settings, $flatten$ is used to combine parses from individual tapes.

We want to be able to use ADPfusion for any fixed, but arbitrary number of input tapes and allow the user to integrate new (non-) terminal parsers easily with the existing library. The ability to fuse applications of *concatMap*, instead of having to rely on $flatten$, would allow for the replacement of the complex system of recursive calls to $flatten$ with simpler calls to $concatMap$.

As an example, we give the (simplified) code used for multi-tape indices. A *Subword* $(i:.j)$ denotes the substring currently parsed. The highest subword index is removed from the index stack, followed by a recursive call to *tableIx* to calculate inner indices. Using $flatten$, the set of indices is expanded and the index stack extended (with a payload z and a temporary stack a).

```
class TableIx i where
    tableIx :: i → Stream (z, a, i) → Stream (z, a, i)
instance TableIx is ⇒ TableIx (is :. Subword) where
    tableIx (is :. Subword (i :. j))
      = flatten mk step ∘ tableIx is
        ∘ map (λ(z, a, (ns :. n)) → (z, (a :. n), ns))
  where
    mk (z, a :. Subword (k :. l), ns) = (z, a, ns, l, l)
    step (z, a, ns, k, l)
      | l > j       = Done
      | otherwise = Yield (z, a, (ns :. subword k l))
                          (z, a, ns, k, l + 1)
```

A fusable version of *concatMap* simplifies the implementation.

```
instance TableIx is ⇒ TableIx (is :. Subword) where
    tableIx (is :. Subword (i :. j))
      = concatMap f ∘ tableIx
        ∘ map (λ(z, a, (ns :. n)) → (z, (a :. n), ns))
  where
    f (z, a :. Subword (k :. l), ns) =
      map (λm → (z, a, ns :. subword m j))
          (enumFromStepN l 1 (j − l + 1))
```

This simplicity becomes more pronounced as *TableIx* instances statically track additional boundary conditions, maximal yield sizes, and special table conditions which we have omitted here, for clarity.

input length	400	600	800	1 000
ADPfusion$_{\texttt{hermit}}$	0.03s	0.10s	0.22s	0.41s
ADPfusion$_{\texttt{flatten}}$	0.03s	0.10s	0.22s	0.44s
ADPfusion$_{\texttt{concatMap}}$	0.19s	0.64s	1.56s	3.15s
C	0.01s	0.04s	0.09s	0.20s

Table 1: Runtime in seconds for different versions of the `Nussinov78` algorithm using `ADPfusion` and `C`.

Performance of ADPfusion

We have re-implemented ADPfusion using $concatMap$ in order to test the performance of the $concatMap$ transformation on real code. Since ADPfusion is built upon a finite, fixed set of functions (mainly the stream-generating *MkStream* type class), HERMIT optimizations can be targeted to exactly the offending calls.

Table 1 summarizise these results for various input lengths. All applications of $concatMap$ are rewritten, the `Step` data constructors are successfully eliminated, and unboxing (especially of loop counters) of all variables occurs. The HERMIT-optimized version (ADPfusion$_{\texttt{hermit}}$) is on par with the version using `flatten`. Using `concatMap` without HERMIT optimization leads to a slowdown of ≈6-8× compared to both optimized versions. For comparison, we include runtimes for the `C` reference implementation. (Options used: `ghc -O2 -fllvm`, resp. `gcc -O3`, measurements performed on an Intel Core i5-3570K).

8. Related Work

Starting with deforestation work by Wadler [30] a number of approaches to deforestation in Haskell have been developed, including foldr/build [7], unfoldr/destroy [27], and Stream Fusion [3]. Each approach is limited both theoretically and practically. The inability of foldr/build to fuse zip is a theoretical limitation due to the nature of the primitive consumer ($foldr$), which can only traverse a single list at a time. On the other hand, fusing $foldl$ is only a practical limitation to foldr/build, and Gill [5] proposes an *arity-raising* transformation to lift this limitation. As the dual to foldr/build, the unfoldr/destroy approach cannot fuse $unzip$, because the primitive producer ($unfoldr$) cannot produce more than a single list at a time. Stream Fusion extends unfoldr/destroy, overcoming a practical limitation when fusing $filter$. The work in this paper addresses another practical limitation common to unfoldr/destroy and Stream Fusion, fusing $concatMap$.

The development of systems with fine-grained instruction-level parallelism (SIMD extensions) poses new challenges for fusion systems. Normally, streams are defined to yield a single element at a time. This is not conducive to instruction-level parallelism, where multiple elements are fetched and operated on atomically. This has been addressed recently by *Generalized Stream Fusion* [17], which uses a more complicated `Stream` type that features many pairs of state and generator function. Each pair produces a stream with a different granularity of operation, from traditional scalar streams to wide vectors of stream elements. Producers and transformers modify all of the pairs accordingly, but consumers select a single generator and state based on their needs, discarding the rest. Given a hypothetical $flatten$ combinator which accepted all of the pairs, our transformation could be extended to handle Generalized Stream Fusion.

Our extended Stream Fusion system suffers from duplicate loop counters. Extending the state as we do in Section 4.1 necessarily results in extra arguments to the functions resulting from call-pattern specialization. On many occassions, these extra arguments are passed around unmodified, or mutated in lockstep with another argument (hence 'duplicate'). The resulting functions of high arity create register pressure, which has performance implications. A newly developed Flow Fusion system [16], based on Series Expressions [31], attempts to address this problem. However, Flow Fusion targets static, first-order, non-recursive stream pipelines, which necessarily excludes $concatMap$.

Alternatively, the task of eleminating duplicate loop counters as well as further loop optimizations could be delegated to the LLVM [15] backend [28] for GHC. New developments in polyhedral loop optimizations [9] may offer even more opportunities for further optimization of the resulting code.

List comprehensions offer convenient syntax for nested list computations and are common in Haskell programs. Comprehensions are desugared by the compiler to list combinators. The original desugaring scheme of Haskell 98 [20] is simple, but results in many intermediate lists. Wadler [29] improves on this by offering a translation which automatically eliminates some intermediate lists, at the cost of a more complex desugaring scheme. Gill [5] simplifies by desugaring directly to calls of $foldr$ and $build$, which are subsequently fused by the compiler. None of these schemes results in code which is suitable for fusion with our system, as all generate branching case expressions for guards and patterns which can fail. We return to the Haskell 98 translation, modifying it to avoid branching case expressions, which inhibit our transformation.

9. Conclusions and Future Work

In this paper, we used HERMIT to specify a custom GHC plugin which implements a transformation for fusing Stream Fusion's $concatMap$. A key benefit of HERMIT is that it lowers the barrier to implementing such transformations. HERMIT allows a user to rapidly prototype an optimization, interactively exploring the transformation in action *during compilation*.

Our transformation extends the one originally proposed by Coutts [2]. We store the value of the outer stream in a modified inner-stream state so it is available to the inner-stream generator. We also extended the transformation to monadic streams. These extensions require us to manipulate syntactic constructs of the Core representation of the program, something that is currently inexpressible by GHC's RULES rewrite system, and provides evidence that more expressive RULES may be useful in practice.

Several subtleties were uncovered by implementing the transformation itself. Most notably, branching case statements in the body of the function argument to $concatMapS$ result in two streams which cannot be merged, due to their differing (existentially quantified) state types. This motivated a novel translation scheme for list comprehension guards and pattern match failures by way of the $filter$ combinator. Additionally, specific simplification heuristics are required to enable the transformation in practice. Each step in Algorithm 1 (Section 5.1), and their specific ordering, was discovered by exploring the optimization interactively with HERMIT.

Aggressive call-pattern specialization is crucial for Stream Fusion. Our inlining heuristic in Section 5.4 was key to achieving good fusion. This suggests that modifying GHC's implementation of specialization to look through let bindings may be profitable in general, and is worth pursuing as future work.

A number of steps in the transformation have the potential to duplicate work (or allocation). The impact of this duplication remains unquantified, though our performance measurements in Section 6 show the transformation is generally an optimization. Speedups of micro-benchmarks targeted by the transformation are considerable. This is in no small part due to the fact that existing Stream Fusion frameworks perform so poorly on $concatMap$. As Figure 2 illustrates, lists often perform better than vectors in $concatMap$-heavy code because foldr/build is so good at fusing $concatMap$.

Results from the nofib benchmark suite are mostly positive with speedup of ≈15% and a ≈10% reduction in allocation, on average.

Some programs experience large speedups or slowdowns. We are confident that the slowdowns are the result of the limited set of stream combinators we have implemented, along with other practical implementation issues, rather than the *concatMap* transformation itself. Making a production quality system that can be "always on" remains future work.

This performance is already available to users of *flatten*, but at considerable cost in implementation complexity. This complexity often leads to sub-optimal uses of *flatten*. Users of a fully fused *concatMap* can more readily take advantage of the hard work of library writers.

The `ADPfusion` library relies on Stream Fusion on vectors to achieve good performance. The preliminary results we presented in Sec. 7 suggest that using HERMIT to optimize *concatMap* will reduce the implementation complexity of the library considerably, without unduly sacrificing performance.

Acknowledgments

We thank Neil Sculthorpe, Nicolas Frisby, Duncan Coutts, Nancy Retzlaff, and the three anonymous reviewers for feedback on drafts of this paper. This material is based upon work supported by the National Science Foundation under Grant No. 1117569. The second author was funded, in part, by the Austrian *Fonds zur Förderung der wissenschaftlichen Forschung (FWF) SFB F43*, and would like to thank Peter F. Stadler for hosting him for an extended scientific stay.

References

[1] D. Burkett, J. Blitzer, and D. Klein. Joint Parsing and Alignment with Weakly Synchronized Grammars. In *Human Language Technologies: The 2010 Annual Conference of the North American Chapter of the Association for Computational Linguistics*, pages 127–135. Association for Computational Linguistics, 2010.

[2] D. Coutts. *Stream Fusion: Practical Shortcut Fusion for Coinductive Sequence Types*. PhD thesis, University of Oxford, 2010.

[3] D. Coutts, R. Leshchinskiy, and D. Stewart. Stream fusion: From lists to streams to nothing at all. In *Proceedings of the 12th ACM SIGPLAN International Conference on Functional Programming*, pages 315–326, Freiburg, Germany, 2007. ACM.

[4] A. Farmer, A. Gill, E. Komp, and N. Sculthorpe. The HERMIT in the machine: A plugin for the interactive transformation of GHC core language programs. In *Proceedings of the ACM SIGPLAN Haskell Symposium*, Haskell '12, pages 1–12. ACM, 2012. .

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

[6] A. Gill. A Haskell hosted DSL for writing transformation systems. In *Proceedings of the IFIP TC 2 Working Conference on Domain-Specific Languages*, DSL '09, pages 285–309. Springer-Verlag, July 2009.

[7] A. Gill, J. Launchbury, and S. L. Peyton Jones. A short cut to deforestation. pages 223–232. ACM Press, 1993.

[8] J.-Y. Girard. *Interprétation fonctionelle et élimination des coupures de l'arithmétique d'ordre supérieur*. PhD thesis, Université Paris VII, 1972.

[9] T. Grosser, H. Zheng, R. Aloor, A. Simbürger, A. Größlinger, and L.-N. Pouchet. Polly - Polyhedral optimization in LLVM. In *First International Workshop on Polyhedral Compilation Techniques*, 2011.

[10] D. Grune and C. J. Jacobs. *Parsing Techniques: A Practical Guide*. Springer-Verlag New York Inc, 2008.

[11] R. Hinze, T. Harper, and D. W. James. Theory and Practice of Fusion. *Implementation and Application of Functional Languages*, pages 19–37, 2011.

[12] C. Höner zu Siederdissen. Sneaking around concatMap: Efficient combinators for dynamic programming. In *Proceedings of the 17th ACM SIGPLAN International Conference on Functional Programming*, pages 215–226, Copenhagen, Denmark, 2012. ACM.

[13] C. Höner zu Siederdissen, I. L. Hofacker, and P. F. Stadler. How to Multiply Dynamic Programming Algorithms. In *Brazilian Symposium on Bioinformatics (BSB 2013), Lecture Notes in Bioinformatics*, volume 8213. Springer, Heidelberg, 2013.

[14] F. W. Huang, J. Qin, C. M. Reidys, and P. F. Stadler. Partition function and base pairing probabilities for RNA–RNA interaction prediction. *Bioinformatics*, 25(20):2646–2654, 2009.

[15] C. Lattner and V. Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Code Generation and Optimization, 2004. CGO 2004. International Symposium on*, pages 75–86. IEEE, 2004.

[16] B. Lippmeier, M. M. Chakravarty, G. Keller, and A. Robinson. Data Flow Fusion with Series Expressions in Haskell. In *Proceedings of the 2013 ACM SIGPLAN Haskell Symposium*, Haskell '13, pages 93–104. ACM, 2013.

[17] G. Mainland, R. Leshchinskiy, and S. Peyton Jones. Exploiting Vector Instructions with Generalized Stream Fusion. In *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming*, pages 37–48, 2013.

[18] B. O'Sullivan. `http://hackage.haskell.org/package/criterion`.

[19] W. Partain. The nofib Benchmark Suite of Haskell Programs. In *Proceedings of the 1992 Glasgow Workshop on Functional Programming*, pages 195–202, 1993.

[20] S. Peyton Jones, editor. *Haskell 98 Language and Libraries – The Revised Report*. Cambridge University Press, Cambridge, England, 2003.

[21] S. Peyton Jones. Call-pattern Specialisation for Haskell Programs. In *Proceedings of the 12th ACM SIGPLAN International Conference on Functional Programming*, ICFP '07, pages 327–337. ACM, 2007.

[22] J. C. Reynolds. Towards a theory of type structure. In *Colloque sur la Programmation*, volume 19 of *Lecture Notes in Computer Science*, pages 408–423. Springer–Verlag, 1974.

[23] A. Santos. *Compilation by Transformation in Non-Strict Functional Languages*. PhD thesis, University of Glasgow, 1995.

[24] N. Sculthorpe, A. Farmer, and A. Gill. The HERMIT in the tree: Mechanizing program transformations in the GHC core language. In *Implementation and Application of Functional Languages 2012*, volume 8241 of *Lecture Notes in Computer Science*. Springer, 2013.

[25] N. Sculthorpe, N. Frisby, and A. Gill. The Kansas University Rewrite Engine: A Haskell-embedded strategic programming language with custom closed universes. Submitted to the Journal of Functional Programming, 2013.

[26] M. Sulzmann, M. M. T. Chakravarty, S. Peyton Jones, and K. Donnelly. System F with type equality coercions. In *Types in Language Design and Implementaion*, pages 53–66. ACM, 2007.

[27] J. Svenningsson. *Shortcut Fusion for Accumulating Parameters Zip-like Functions*. PhD thesis, 2002.

[28] D. A. Terei and M. M. Chakravarty. An LLVM Backend for GHC. In *Proceedings of the third ACM Haskell Symposium*, pages 109–120. ACM, 2010.

[29] P. Wadler. List comprehensions. In S. Peyton Jones, editor, *The Implementation of Functional Programming Languages*, chapter 7. Prentice Hall International, 1987.

[30] P. Wadler. Deforestation: Transforming programs to eliminate trees. In *Proceedings of the 2nd European Symposium on Programming*, pages 344–358, London, UK, UK, 1988. Springer-Verlag.

[31] R. C. Waters. Automatic transformation of series expressions into loops. *ACM Trans. Program. Lang. Syst.*, 13(1):52–98, Jan. 1991.

[32] B. A. Yorgey, S. Weirich, J. Cretin, S. Peyton Jones, D. Vytiniotis, and J. P. Magalhães. Giving Haskell a promotion. In *Types in Language Design and Implementation*, pages 53–66. ACM, 2012.