Stream Fusion

From Lists to Streams to Nothing at All

Duncan Coutts[1]   Roman Leshchinskiy[2]   Don Stewart[2]

[1] Programming Tools Group
Oxford University Computing Laboratory
duncan.coutts@comlab.ox.ac.uk

[2] Computer Science & Engineering
University of New South Wales
{rl,dons}@cse.unsw.edu.au

Abstract

This paper presents an automatic deforestation system, *stream fusion*, based on equational transformations, that fuses a wider range of functions than existing short-cut fusion systems. In particular, stream fusion is able to fuse zips, left folds and functions over nested lists, including list comprehensions. A distinguishing feature of the framework is its simplicity: by transforming list functions to expose their structure, intermediate values are eliminated by general purpose compiler optimisations.

We have reimplemented the Haskell standard List library on top of our framework, providing stream fusion for Haskell lists. By allowing a wider range of functions to fuse, we see an increase in the number of occurrences of fusion in typical Haskell programs. We present benchmarks documenting time and space improvements.

Categories and Subject Descriptors   D.1.1 [*Programming Techniques*]: Applicative (Functional) Programming;   D.3.4 [*Programming Languages*]: Optimization

General Terms   Languages, Algorithms

Keywords   Deforestation, program optimisation, program transformation, program fusion, functional programming

1. Introduction

Lists are the primary data structure of functional programming. In lazy languages, such as Haskell, lists also serve in place of traditional control structures. It has long been recognised that composing list functions to build programs in this style has advantages for clarity and modularity, but that it incurs some runtime penalty, as functions allocate intermediate values to communicate results. Fusion (or deforestation) attempts to remove the overhead of programming in this style by combining adjacent transformations on structures to eliminate intermediate values.

Consider this simple function which uses a number of intermediate lists:

$$f :: Int \rightarrow Int$$
$$f\ n = sum\ [\ k * m\ |\ k\ \leftarrow\ [1..n],\ m\ \leftarrow\ [1..k]\ ]$$

No previously implemented short-cut fusion system eliminates all the lists in this example. The fusion system presented in this paper does. With this system, the Glasgow Haskell Compiler (The GHC Team 2007) applies all the fusion transformations and is able to generate an efficient "worker" function f' that uses only unboxed integers ($Int\#$) and runs in constant space:

```
f' :: Int# → Int#
f' n =
  let go :: Int# → Int# → Int#
      go z k =
          case k > n of
            False → case 1 > k of
              False → to (z + k) k (k + 1) 2
              True → go z (k + 1)
            True → z
      to :: Int# → Int# → Int# → Int# → Int#
      to z k k' m =
          case m > k of
            False → to (z + (k * m)) k k' (m + 1)
            True → go z k'
  in go 0 1
```

Stream fusion takes a simple three step approach:

1. Convert recursive structures into non-recursive *co-structures*;
2. Eliminate superfluous conversions between structures and co-structures;
3. Finally, use general optimisations to fuse the co-structure code.

By transforming pipelines of recursive list functions into non-recursive ones, code becomes easier to optimise, producing better results. The ability to fuse all common list functions allows the programmer to write in an elegant declarative style, and still produce excellent low level code. We can finally write the code we *want* to be able to write without sacrificing performance!

1.1 Short-cut fusion

The example program is a typical high-level composition of list producers, transformers and consumers. However, extensive optimisations are required to transform programs written in this style into efficient low-level code. In particular, naive compilation will produce a number of intermediate data structures, resulting in poor performance. We would like to have the compiler remove these intermediate structures automatically. This problem, deforestation (Wadler 1990), has been studied extensively (Meijer et al. 1991; Gill et al. 1993; Takano and Meijer 1995; Gill 1996; Hu et al. 1996; Chitil 1999; Johann 2001; Svenningsson 2002; Gibbons 2004). To illustrate how our approach builds on previous work on short-cut fusion, we review the main approaches.

build/foldr The most practically successful list fusion system to date is the *build/foldr* system (Gill et al. 1993). It uses two combinators, *foldr* and *build*, and a single fusion rule to eliminate adjacent occurrences of the combinators. Fusible functions must be written in terms of these two combinators. A range of standard list functions, and list comprehensions, can be expressed and effectively fused in this way.

There are some notable exceptions that cannot be effectively fused under *build/foldr*: left folds (*foldl*) (functions such as *sum* that consume a list using an accumulating parameter), and zips (functions that consume multiple lists in parallel).

destroy/unfoldr A more recent proposal (Svenningsson 2002) based on unfolds rather than folds addresses these specific shortcomings. However, as proposed, it does not cover functions that handle nested lists (such as *concatMap*) or list comprehensions, and there are inefficiencies fusing *filter*-like functions, which must be defined recursively.

stream fusion Recently, we proposed a new fusion framework for operations on strict arrays (Coutts et al. 2007). While the performance improvements demonstrated for arrays are significant, this previous work describes fusion for only relatively simple operations: maps, filters and folds. It does not address concatenations, functions on nested lists, or zips.

In this paper we extend stream fusion to fill in the missing pieces. Our main contributions are:

- an implementation of stream fusion for lists (Section 2);
- extension of stream fusion to zips, concats, appends (Section 3) and functions on nested lists (Section 4);
- a translation scheme for stream fusion of list comprehensions (Section 5);
- an account of the compiler optimisations required to remove intermediate structures produced by fusion, including functions on nested lists (Section 7);
- an implementation of stream fusion using compiler rewrite rules and concrete results from a complete implementation of the Haskell list library (Section 8).

2. Streams

The intuition behind *build/foldr* fusion is to view lists as sequences represented by data structures, and to fuse functions that work directly on the natural structure of that data. The *destroy/unfoldr* and stream fusion systems take the opposite approach. They convert operations over the list data structure to instead work over the dual of the list: its unfolding or *co-structure*.

In contrast to *destroy/unfoldr*, stream fusion uses an explicit representation of the sequence co-structure: the *Stream* type. Separate functions, *stream* and *unstream*, are used to convert between lists and streams.

2.1 Converting lists to streams

The first step in order to fuse list functions with stream fusion is to convert a function on list structures to a function on stream co-structures (and back again) using *stream* and *unstream* combinators. The function *map*, for example, is simply specified as:

$$map \ :: \ (a \rightarrow b) \rightarrow [a] \rightarrow [b]$$
$$map \ f \ = \ unstream \cdot map_s \ f \cdot stream$$

which composes a map function over streams, with stream conversion to and from lists.

While the natural operation over a list data structure is a fold, the natural operation over a stream co-structure is an unfold. The *Stream* type encapsulates an unfold, wrapping an initial state and a stepper function for producing elements. It is defined as:

data $Stream \ a = \exists s. \ Stream \ (s \rightarrow Step \ a \ s) \ s$
data $Step \ a \ s \ = Done$
$\qquad\qquad\quad | \ Yield \ a \ s$
$\qquad\qquad\quad | \ Skip \ s$

Note that the type of the stream state is existentially quantified and does not appear in the result type: the stream state is encapsulated. The *Stream* data constructor itself is similar to the standard Haskell list function *unfoldr* (Gibbons and Jones 1998),

$$Stream \ :: \ \forall s \ a. \ (s \rightarrow Step \ a \ s) \qquad \rightarrow s \rightarrow Stream \ a$$
$$unfoldr \ :: \ \forall s \ a. \ (s \rightarrow Maybe \ (a, s)) \rightarrow s \rightarrow [a]$$

Writing functions over streams themselves is relatively straightforward. *map*, for example, simply applies its function argument to each yielded stream element, when the stepper is called:

$$map_s \ :: \ (a \rightarrow b) \rightarrow Stream \ a \rightarrow Stream \ b$$
$$map_s \ f \ (Stream \ next_0 \ s_0) \ = \ Stream \ next \ s_0$$
$$\textbf{where}$$
$$\quad next \ s = \ \textbf{case} \ next_0 \ s \ \textbf{of}$$
$$\qquad Done \qquad \rightarrow \ Done$$
$$\qquad Skip \quad s' \rightarrow \ Skip \qquad s'$$
$$\qquad Yield \ x \ s' \rightarrow \ Yield \ (f \ x) \ s'$$

The *stream* function can be defined directly as a stream whose elements are those of the corresponding list. It uses the list itself as the stream state. It is of course non-recursive yielding each element of the list as it is unfolded:

$$stream \ :: \ [a] \rightarrow Stream \ a$$
$$stream \ xs_0 \ = \ Stream \ next \ xs_0$$
$$\textbf{where}$$
$$\quad next \ [\,] \qquad = \ Done$$
$$\quad next \ (x : xs) = \ Yield \ x \ xs$$

The *unstream* function unfolds the stream and builds a list structure. Unfolding a stream to produce a list is achieved by repeatedly calling the stepper function of the stream, to yield the stream's elements.

$$unstream \ :: \ Stream \ a \rightarrow [a]$$
$$unstream \ (Stream \ next_0 \ s_0) \ = \ unfold \ s_0$$
$$\textbf{where}$$
$$\quad unfold \ s = \ \textbf{case} \ next_0 \ s \ \textbf{of}$$
$$\qquad Done \qquad \rightarrow \qquad [\,]$$
$$\qquad Skip \quad s' \rightarrow \qquad unfold \ s'$$
$$\qquad Yield \ x \ s' \rightarrow \ x : unfold \ s'$$

In contrast to *unfoldr*, the *Stream* stepper function has one other alternative, it can *Skip*, producing a new state but yielding no new value in the sequence. This is not necessary for the semantics but as we shall show later, is crucial for the implementation. In particular it is what allows *all* stepper functions to be non-recursive.

2.2 Eliminating conversions

Writing list functions using compositions of *stream* and *unstream* is clearly inefficient: each function must first construct a new *Stream*, and when it is done, unfold the stream back to a list. This is evident in the definition of *map* from the previous section. Instead of consuming and constructing a list once: *stream* consumes a list, allocating *Step* constructors; *map_s* consumes and allocates more *Step* constructors; finally, *unstream* consumes the *Step* constructors and allocates new list nodes. However, if we compose two functions implemented via streams:

$$map \ f \ \cdot \ map \ g \ =$$
$$unstream \ \cdot \ map_s \ f \ \cdot \ stream \ \cdot \ unstream \ \cdot \ map_s \ g \ \cdot \ stream$$

we immediately see an opportunity to eliminate the intermediate list conversions!

Assuming *stream* $\cdot$ *unstream* as the identity on streams, we obtain the rewrite rule:

$$\langle \textbf{stream/unstream fusion} \rangle \ \forall \ s \ . \ stream \ (unstream \ s) \ \mapsto \ s$$

The Glasgow Haskell Compiler supports programmer-defined rewrite rules (Peyton Jones et al. 2001), applied by the compiler during compilation. We can specify the stream fusion rule as part of the list library source code — without changing the compiler. When the compiler applies this rule to our example, it yields:

$$unstream \cdot map_s\ f \cdot map_s\ g \cdot stream$$

Our pipeline of list transformers has now been transformed into a pipeline of stream transformers. Externally, the pipeline still consumes and produces lists, just as the direct list implementation of $map \cdot map$ does. However, internally the $map_s\ f \cdot map_s\ g$ pipeline is the composition of (simple, non-recursive) stream functions.

It is interesting to note that the $stream/unstream$ rule is not really a classical fusion rule at all. It only eliminates the list allocations that were introduced in converting operations to work over streams.

2.3 Fusing co-structures

Having converted the functions over list structures into functions over stream co-structures, the question now is how to optimise away intermediate $Step$ constructors produced by composed functions on streams.

The key trick is that all stream producers are non-recursive.

Once list functions have been transformed to compositions of non-recursive stepper functions, there is an opportunity for real fusion: the compiler can relatively easily eliminate intermediate $Step$ constructors produced by the non-recursive steppers, using existing general purpose optimisations. We describe this process in detail in Section 7.

3. Writing stream combinators

Figure 1 shows the definitions of several standard algorithms on flat streams which we use throughout the paper. For the most part, these definitions are essentially the same as those presented in our previous work (Coutts et al. 2007). In the following, we discuss some of the combinators and highlight the principles underlying their implementation.

No recursion: filter Similarly to map_s, the stepper function for $filter_s$ is non-recursive which is crucial for producing efficient fused code. In the case of $filter_s$, however, a non-recursive implementation is only possible by introducing $Skip$ in place of elements that are removed from the stream — the only alternative is to recursively consume elements from the input stream until we find one that satisfies the predicate (as is the case for the $filter$ function in the *destroy/unfoldr* system). As we are able to avoid this recursion, we maintain trivial control flow for streams, and thus never have to see through fixed points to optimise, yielding better code.

Consuming streams: fold The only place where recursion is allowed is when we consume a stream to produce a different type. The canonical examples of this are $foldr_s$ and $foldl_s$ which are defined in Figure 1. To understand this it is helpful to see compositions of stream functions simply as descriptions of pipelines which on their own do nothing. They require a recursive function at the end of the pipeline to unroll sequence elements, to actually construct a concrete value.

Recursion is thus only involved in repeatedly pulling values out of the stream. Each step in the pipeline itself requires no recursion. Of course because of the possibility that a single step might skip it may take many steps to actually yield a value.

Complex stream states: append Many operations on streams encode complex control flow by using non-trivial state types. One

$filter_s :: (a \rightarrow Bool) \rightarrow Stream\ a \rightarrow Stream\ a$
$filter_s\ p\ (Stream\ next_0\ s_0) = Stream\ next\ s_0$
   where
     $next\ s =$ **case** $next_0\ s$ **of**
        $Done$ $\rightarrow Done$
        $Skip\ \ \ \ s'$ $\rightarrow Skip\ \ \ \ s'$
        $Yield\ x\ s'\ |\ p\ x$ $\rightarrow Yield\ x\ s'$
               $|\ otherwise \rightarrow Skip\ \ \ \ s'$

$return_s :: a \rightarrow Stream\ a$
$return_s\ x = Stream\ next\ True$
   where
     $next\ True = Yield\ x\ False$
     $next\ False = Done$

$enumFromTo_s :: Enum\ a \Rightarrow a \rightarrow a \rightarrow Stream\ a$
$enumFromTo_s\ l\ h = Stream\ next\ l$
   where
     $next\ n\ |n > h\ \ \ \ = Done$
          $|otherwise = Yield\ n\ (succ\ n)$

$foldr_s :: (a \rightarrow b \rightarrow b) \rightarrow b \rightarrow Stream\ a \rightarrow b$
$foldr_s\ f\ z\ (Stream\ next\ s_0) = go\ s_0$
   where
     $go\ s =$ **case** $next\ s$ **of**
        $Done\ \ \ \ \ \rightarrow z$
        $Skip\ \ \ \ s' \rightarrow \ \ \ \ \ \ \ go\ s'$
        $Yield\ x\ s' \rightarrow f\ x\ (go\ s')$

$foldl_s :: (b \rightarrow a \rightarrow b) \rightarrow b \rightarrow Stream\ a \rightarrow b$
$foldl_s\ f\ z\ (Stream\ next\ s_0) = go\ z\ s_0$
   where
     $go\ z\ s =$ **case** $next\ s$ **of**
        $Done\ \ \ \ \ \rightarrow z$
        $Skip\ \ \ \ s' \rightarrow go\ z\ \ \ \ \ \ \ s'$
        $Yield\ x\ s' \rightarrow go\ (f\ z\ x)\ s'$

$append_s :: Stream\ a \rightarrow Stream\ a \rightarrow Stream\ a$
$append_s\ (Stream\ next_a\ s_{a0})\ (Stream\ next_b\ s_{b0}) =$
 $Stream\ next\ (Left\ s_{a0})$
   where
     $next\ (Left\ s_a)\ \ \ =$
      case $next_a\ s_a$ **of**
        $Done\ \ \ \ \ \ \ \rightarrow Skip\ \ \ (Right\ s_{b0})$
        $Skip\ \ \ \ s'_a \rightarrow Skip\ \ \ (Left\ s'_a)$
        $Yield\ x\ s'_a \rightarrow Yield\ x\ (Left\ s'_a)$
     $next\ (Right\ s_b) =$
      case $next_b\ s_b$ **of**
        $Done\ \ \ \ \ \ \ \rightarrow Done$
        $Skip\ \ \ \ s'_b \rightarrow Skip\ \ \ (Right\ s'_b)$
        $Yield\ x\ s'_b \rightarrow Yield\ x\ (Right\ s'_b)$

$zip_s :: Stream\ a \rightarrow Stream\ b \rightarrow Stream\ (a,\ b)$
$zip_s\ (Stream\ next_a\ s_{a0})\ (Stream\ next_b\ s_{b0}) =$
 $Stream\ next\ (s_{a0},\ s_{b0},\ Nothing)$
   where
     $next\ (sa,\ sb,\ Nothing) =$
      case $next_a\ s_a$ **of**
        $Done\ \ \ \ \ \ \ \rightarrow Done$
        $Skip\ \ \ \ s'_a \rightarrow Skip\ (s'_a,\ s_b,\ Nothing)$
        $Yield\ a\ s'_a \rightarrow Skip\ (s'_a,\ s_b,\ Just\ a)$
     $next\ (s'_a,\ s_b,\ Just\ a) =$
      case $next_b\ s_b$ **of**
        $Done\ \ \ \ \ \ \ \rightarrow Done$
        $Skip\ \ \ \ s'_b \rightarrow Skip\ \ \ \ \ \ \ (s'_a,\ s'_b,\ Just\ a)$
        $Yield\ b\ s'_b \rightarrow Yield\ (a,\ b)\ (s'_a,\ s'_b,\ Nothing)$

Figure 1: Flat stream combinators

$$concatMap_s \ :: \ (a \to Stream \ b) \to Stream \ a \to Stream \ b$$
$$concatMap_s \ f \ (Stream \ next_a \ s_{a0}) = Stream \ next \ (s_{a0}, \ Nothing)$$

where

$next \ (s_a, \ Nothing) =$
  case $next_a \ s_a$ **of**
    $Done \quad\quad \to Done$
    $Skip \quad\ s_a' \to Skip \ (s_a', \ Nothing)$
    $Yield \ a \ s_a' \to Skip \ (s_a', \ Just \ (f \ a))$
$next \ (s_a, \ Just \ (Stream \ next_b \ s_b)) =$
  case $next_b \ s_b$ **of**
    $Done \quad\quad \to Skip \quad (s_a, \ Nothing)$
    $Skip \quad\ s_b' \to Skip \quad (s_a, \ Just \ (Stream \ next_b \ s_b'))$
    $Yield \ b \ s_b' \to Yield \ b \ (s_a, \ Just \ (Stream \ next_b \ s_b'))$

Figure 2: Definition of $concatMap_s$ on streams

example is $append_s$ which produces a single stream by concatenating two independent streams, with possibly different state types. The state of the new stream necessarily contains the states of the two component streams.

To implement concatenation we notice that at any moment we need only the state of the first stream, or the state of the second. The $next$ function for append thus operates in two modes, either yielding elements from the first stream, or elements from the second.

The two modes can then be encoded as a sum type, $Either \ s_a \ s_b$, tagging which mode the stepper is in: either yielding the first stream, or yielding the second. The modes are thus represented as $Left \ s_a$ or $Right \ s_b$ and there is one clause of $next$ for each. When we get to the end of the first stream we have to switch modes so that we can start yielding elements from the second stream.

This is another example where it is convenient to use $Skip$. Instead of immediately having to yield the first element of the second stream (which is impossible anyway since the second stream may skip) we can just transition into a new state where we will be able to do so. The rule of thumb is in each step to do one thing and one thing only.

What is happening here of course is that we are using state to encode control flow. This is the pattern used for all the more complex stream functions. Section 7.1 explains how code in this style is optimised.

Consuming multiple streams: zip Functions that consume multiple stream in parallel, such as zip_s, also require non-trivial state. Unsurprisingly the definition of zip_s on streams is quite similar to the equivalent definition in the *destroy/unfoldr* system. The main difference is that the stream version has to cope with streams that produce $Skips$, which complicates matters slightly. In particular, it means that the we must cope with a situation where we have an element from the first stream but cannot immediately (i.e., non-recursively) obtain an element from the second one.

So rather than trying to extract an element from one stream, then from another in a single step, we must pull from the first stream, store the element in the state and then move into a new state where we attempt to pull a value from the second stream. Once the second stream has yielded a value, we can return the pair. In each call of the $next$ function we pull from at most one stream. Again we see that in any single step we can do only one thing.

4. Functions on nested streams

The last major class of list functions that we need to be able to fuse are ones that deal with nested lists. The canonical example is $concatMap$, but this class also includes all the list comprehensions. In terms of control structures, these functions represent nested recursion and nested loops.

$$\mathcal{T}[\![\ [E \ | \quad] \ ]\!] \qquad\qquad = \ return \ E$$
$$\mathcal{T}[\![\ [E \ | \ B, Q] \ ]\!] \qquad = \ guard \ B \ (\mathcal{T}[\![\ [E \ | \ Q] \ ]\!])$$
$$\mathcal{T}[\![\ [E \ | \ P \leftarrow L, Q] \ ]\!] = \ \textbf{let} \ f \ P = \ True$$
$$\qquad\qquad\qquad\qquad f \ \_ = \ False$$
$$\qquad\qquad\qquad\qquad g \ P = \ \mathcal{T}[\![\ [E \ | \ Q] \ ]\!]$$
$$\qquad\qquad\qquad\qquad h \ x = \ guard \ (f \ x) \ (g \ x)$$
$$\qquad\qquad\qquad \textbf{in} \ concatMap \ h \ L$$
$$\mathcal{T}[\![\ [E \ | \ \textbf{let} \ decls, Q] \ ]\!] = \ \textbf{let} \ decls \ \textbf{in} \ \mathcal{T}[\![\ [E \ | \ Q] \ ]\!]$$

Figure 3: Translation scheme for list comprehensions

The ordinary list $concatMap$ function has the type:

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

For each element of its input list it applies a function which gives another list and it concatenates all these lists together. To define a list $concatMap$ that is fusible with its input and output list, and with the function that yields a list, we will need a stream-based $concatMap_s$ with the type:

$$concatMap_s \ :: \ (a \to Stream \ b) \to Stream \ a \to Stream \ b$$

To get back the list version we compose $concatMap_s$ with $stream$ and $unstream$ and compose the function argument f with $stream$:

$$concatMap \ f = \ unstream \ . \ concatMap_s \ (stream \ . \ f) \ . \ stream$$

To convert a use of list $concatMap$ to stream form we need a fusible list consumer c and fusible list producers p and f. For c, p and f to be fusible means that they must be defined in terms of $stream$ or $unstream$ and appropriate stream consumers and producers c_s, p_s and f_s:

$$c = \ c_s \ . \ stream$$
$$p = \ unstream \ . \ p_s$$
$$f = \ unstream \ . \ f_s$$

We now compose them, expanding their definitions to reveal the $stream$ and $unstream$ conversions, and then apply the $stream/unstream$ fusion rule three times:

$$\quad c \ \cdot \ concatMap \ f \ \cdot \ p$$
$$= c_s \ \cdot \ stream$$
$$\quad \cdot \ unstream \ \cdot \ concatMap_s \ (stream \ \cdot \ f) \ \cdot \ stream$$
$$\quad \cdot \ unstream \ \cdot \ p_s$$
$$= c_s \ \cdot \ concatMap_s \ (stream \ \cdot \ f) \ \cdot \ p_s$$
$$= c_s \ \cdot \ concatMap_s \ (stream \ \cdot \ unstream \ \cdot \ f_s) \ \cdot \ p_s$$
$$= c_s \ \cdot \ concatMap_s \ f_s \ \cdot \ p_s$$

Actually defining $concatMap_s$ on streams is somewhat tricky. We need to get an element a from the outer stream, then $f \ a$ gives us a new inner stream. We must drain this stream before moving onto the next outer a element.

There are thus two modes: one where we are trying to obtain an element from the outer stream; and another mode in which we have the current inner stream and are pulling elements from it. We can represent these two modes with the state type:

$$(s_a, \ Maybe \ (Stream \ b))$$

where s_a is the state type of the outer stream. The full $concatMap_s$ definition is given in Figure 2.

5. List comprehensions

List comprehensions provide a very concise way of expressing operations that select and combine lists. It is important to fuse them to help achieve our goal of efficiently compiling elegant, declarative programs. Recall our introductory example:

$$f \ n = \ sum \ [\ k * m \ | \ k \ \leftarrow \ [1..n], \ m \ \leftarrow \ [1..k] \ ]$$

There are two aspects to fusion of list comprehensions. One is fusing with list generators. Obviously this is only possible when the generator expression is itself fusible. The other aspect is eliminating any intermediate lists used internally in the comprehension, and allowing the comprehension to be fused with a fusible list consumer.

The *build/foldr* system tackles this second aspect directly by using a translation of comprehensions into uses of *build* and *foldr* that, by construction, uses no intermediate lists. Furthermore, by using *foldr* to consume the list generators it allows fusion there too.

Obviously the *build/foldr* translation, employing *build*, is not suitable for streams. The other commonly used translation (Wadler 1987) directly generates recursive list functions. For streams we either need a translation directly into a single stream (potentially with a very complex state and stepper function) or a translation into fusible primitives. We opt for the second approach which makes the translation itself simpler but leaves us with the issue of ensuring that the expression we build really does fuse.

We use a translation very similar to the translation given in the Haskell language specification (Peyton Jones et al. 2003). However, there are a couple of important differences. The first change is to always translate into list combinators, rather than concrete list syntax. This allows us to delay expansion of these functions and use compiler rewrite rules to turn them into their stream-fusible counterparts.

The second change is to modify the translation so that conditionals do not get in the way of fusion. The Haskell'98 translations for expressions and generators are:

$$\mathcal{T}[\![\ [E \mid B, Q]\ ]\!] \quad = \text{if } B \text{ then } \mathcal{T}[\![\ [E \mid Q]\ ]\!] \text{ else } [\,]$$
$$\mathcal{T}[\![\ [E \mid P \leftarrow L, Q]\ ]\!] = \text{let } ok\ P = \mathcal{T}[\![\ [E \mid Q]\ ]\!]$$
$$ok\ \_ = [\,]$$
$$\text{in } concatMap\ ok\ L$$

Note that for the generator case, P can be any pattern and as such pattern match failure is a possibility. This is why the ok function has a catch-all clause.

We cannot use this translation directly because in both cases the resulting list is selected on the basis of a test. We cannot directly fuse when the stream producer is not statically known, as is the case when we must make a dynamic choice between two streams. The solution is to push the dynamic choice inside the stream. We use the function *guard*:

$$guard\ ::\ Bool \rightarrow [a] \rightarrow [a]$$
$$guard\ True\ xs\ =\ xs$$
$$guard\ False\ xs\ =\ [\,]$$

This function is quite trivial, but by using a named combinator rather than primitive syntax it enables us to rewrite to a stream fusible implementation:

$$guard_s\ ::\ Bool \rightarrow Stream\ a \rightarrow Stream\ a$$
$$guard_s\ b\ (Stream\ next_0\ s_0)\ =\ Stream\ next\ (b,\ s0)$$
$$\mathbf{where}$$
$$next\ (False,\ \_)\ =\ Done$$
$$next\ (True,\ s)\ =\ \mathbf{case}\ next_0\ s\ \mathbf{of}$$
$$Done\ \quad \rightarrow\ Done$$
$$Skip\ \quad s'\ \rightarrow\ Skip\ \quad (True, s')$$
$$Yield\ x\ s'\ \rightarrow\ Yield\ x\ (True, s')$$

The full translation is given in Figure 3. We can use *guard* directly for the case of filter expressions. For generators we build a function that uses *guard* with a predicate based on the generator's pattern.

We can now use this translation on our example. For the sake of brevity we omit the *guard* functions which are trivial in this example since both generator patterns are simple variables.

$$\mathcal{T}[\![\ [k * m \mid k \leftarrow [1..n], m \leftarrow [1..k]\ ]\ ]\!]$$

$$= concatMap\ (\lambda k\ \rightarrow$$
$$concatMap\ (\lambda m\ \rightarrow$$
$$return\ (k * m))$$
$$(enumFromTo\ 1\ k))$$
$$(enumFromTo\ 1\ n)$$

Next we inline all the list functions to reveal the stream versions wrapped in *stream* / *unstream* and we apply the fusion rule three times:

$$= unstream\ (concatMap_s\ (\lambda k\ \rightarrow\ stream\ ($$
$$unstream\ (concatMap_s\ (\lambda m\ \rightarrow\ stream\ ($$
$$unstream\ (return_s\ (k * m))))$$
$$(stream\ (unstream\ (enumFromTo_s\ 1\ k))))))$$
$$(stream\ (unstream\ (enumFromTo_s\ 1\ n))))$$

$$= unstream\ (concatMap_s\ (\lambda k\ \rightarrow$$
$$concatMap_s\ (\lambda m\ \rightarrow$$
$$return_s\ (k * m))$$
$$(enumFromTo_s\ 1\ k))$$
$$(enumFromTo_s\ 1\ n))$$

Finally, to get our full original example we apply sum_s (which is just $foldl_s\ (+)\ 0$) and repeat the inline and fuse procedure one more time. This gives us a term with no lists left; the entire structure has been converted to stream form.

$$= sum_s\ (concatMap_s\ (\lambda k\ \rightarrow$$
$$concatMap_s\ (\lambda m\ \rightarrow$$
$$return_s\ (k * m))$$
$$(enumFromTo_s\ 1\ k))$$
$$(enumFromTo_s\ 1\ n))$$

6. Correctness

Every fusion framework should come with a rigorous correctness proof. Unfortunately, many do not and ours is not an exception. This might seem surprising at first, as we introduce only one rather simple rewrite rule:

$$\forall\ s.\ stream\ (unstream\ s)\ \mapsto\ s$$

Should it not be easy to show that applying this rule does not change the semantics of a program or, conversely, construct an example where the semantics *is* changed? In fact, a counterexample is easily found for the system presented in this paper: with $s = \bot$, we have:

$$stream\ (unstream\ \bot)\ =\ Stream\ next\ \bot \neq \bot$$

Depending on how we define equivalence on streams, other counterexamples can be derived. In the rest of this section we discuss possible approaches to retaining semantic soundness of stream fusion.

6.1 Strictness of streams

The above counter-example is particularly unfortunate as it implies that we can turn terminating programs into non-terminating ones. In our implementation, we circumvent this problem by not exporting the *Stream* data type and ensuring that we never construct bottom streams within our library. Effectively, this means that we treat *Stream* as an *unlifted* type, even though Haskell does not provide us with the means of saying so explicitly.[1]

Avoiding the creation of bottom streams is, in fact, fairly easy. It boils down to the requirement that all stream-constructing functions be non-strict in all arguments except those of type *Stream* which we can presume not to be bottom. This is always possible, as the

---
[1] Launchbury and Paterson (1996) discuss how unlifted types can be integrated into a lazy language.

arguments can be evaluated in the stepper function. For instance, the combinator *guard* defined in the previous section is lazy in the condition. The latter is not inspected until the stepper function has been called for the first time.

In fact, we can easily change our framework such that the rewrite rule removes bottoms instead of introducing them. For this, it is sufficient to make *stream* strict in its argument. Then, we have $stream\ (unstream\ \bot) = \bot$. However, now we can derive a different counterexample:

$$stream\ (unstream\ (Stream\ \bot\ s)) = \bot \neq Stream\ \bot\ s$$

This is much less problematic, though, as it only means that we turn some non-terminating programs into terminating ones. Unfortunately, with this definition of stream it becomes much harder to implement standard Haskell list functions such that they have the desired semantics. The Haskell 98 Report (Peyton Jones et al. 2003) requires that $take\ 0\ xs = [\ ]$, i.e., *take* must be lazy in its second argument. In our library, *take* is implemented as:

$$take\ ::\ Int \rightarrow [a] \rightarrow [a]$$
$$take\ n\ xs\ =\ unstream\ (take_s\ n\ (stream\ xs))$$

$take_s\ ::\ Int \rightarrow Stream\ a \rightarrow Stream\ a$
$take_s\ n\ (Stream\ next\ s)\ =\ Stream\ next'\ (n, s)$
   where
      $next'\ (0, s)\ =\ Done$
      $next'\ (n,s)\ =\ \mathbf{case}\ next\ s\ \mathbf{of}$
         $Done\ \ \ \ \ \rightarrow\ Done$
         $Skip\ \ \ \ s' \rightarrow\ Skip\ \ \ \ (n,\ \ \ \ s')$
         $Yield\ x\ s' \rightarrow\ Yield\ x\ (n-1,s')$

Note that since $take_s$ is strict in the stream argument, *stream* must be lazy if *take* is to have the required semantics. An alternative would be to make $take_s$ lazy in the stream:

$take_s\ n\ s\ =\ Stream\ next'\ (n, s)$
   where
      $next'\ (0, s)\ \ \ \ \ \ \ \ \ \ \ \ =\ Done$
      $next'\ (n, Stream\ next\ s)\ =$
        $\mathbf{case}\ next\ s\ \mathbf{of}$
          $Done\ \ \ \ \ \rightarrow\ Done$
          $Skip\ \ \ \ s' \rightarrow\ Skip\ \ \ (n,\ \ \ \ \ Stream\ next\ s')$
          $Yield\ x\ s' \rightarrow\ Yield\ x\ (n-1, Stream\ next\ s')$

Here, we embed the entire argument stream in the seed of the newly constructed stream, thus ensuring that it is only evaluated when necessary. Unfortunately, such code is currently less amenable to being fully optimised by GHC. Indeed, efficiency was why we preferred the less safe fusion framework presented in this paper to the one outlined here. We do hope, however, that improvements to GHC's optimiser will allow us to experiment with alternatives in the future.

6.2 Equivalence of streams

Even in the absence of diverging computations, it is not entirely trivial to define a useful equivalence relation on streams. This is mainly due to the fact that a single list can be modeled by infinitely many streams. Even if we restrict ourselves to streams producing different sequences of *Step* values, there is still no one-to-one correspondence — two streams representing the same list can differ in the number and positions of *Skip* values they produce. This suggests that equivalence on streams should be defined modulo *Skip* values. In fact, this is a requirement we place on all stream-processing functions: their semantics should not be affected by the presence or absence of *Skip* values.

6.3 Testing

Although we do not have a formal proof of correctness of our framework, we have tested it quite extensively. It is easy to introduce subtle strictness bugs when writing list functions, either di-

rectly on lists or on streams. Fortunately we have a precise specification in the form of the Haskell'98 report. Comparative testing on total values is relatively straightforward, but to test strictness properties however we need to test on partial values. We were inspired by the approach in StrictCheck (Chitil 2006) of checking strictness properties by generating all partial values up to a certain finite depth. However, to be able to generate partial values at higher order type we adapted SmallCheck (Runciman 2006) to generate all partial rather than total values up to any given depth. We used this and the Chasing Bottoms library (Danielsson and Jansson 2004) to compare our implementations against the Haskell'98 specification and against the standard library used by many Haskell implementations.

This identified a number of subtle bugs in our implementation and a handful of cases where we can argue that the specification is unnecessarily strict. We also identified cases where the standard library differs from the specification. The tests document the strictness properties of list combinators and give us confidence that the stream versions do, in fact, have the desired strictness.

7. Compiling stream code

Ultimately, a fusion framework should eliminate temporary data structures. Stream fusion by itself does not, however, reduce allocation - it merely replaces intermediate lists by intermediate *Step* values. Moreover, when a stream is consumed, additional allocations are necessary to maintain its seed throughout the loop. For instance, *append* allocates an *Either* node in each iteration.

This behaviour is quite similar to programs produced by *destroy/unfoldr* and like the latter, our approach relies on subsequent compiler optimisation passes to eliminate these intermediate values. Since we consider more involved list operations than Svenningsson (2002), in particular nested ones, we necessarily require more involved optimisation techniques than the ones discussed in that work. Still, these techniques are generally useful and not specifically tailored to programs produced by our fusion framework. In this section, we identify the key optimisations necessary to produce good code for stream-based programs and explain why they are sufficient to replace streams by nothing at all.

7.1 Flat pipelines

Let us begin with a simple example: $sum\ (xs\ ++\ ys)$. Our fusion framework rewrites this to:

$$foldl_s\ (+)\ 0\ (append_s\ (stream\ xs)\ (stream\ ys))$$

Inlining the definitions of the stream combinators, we get

```
let next_stream xs =
       case xs of
          []      → Done
          x : xs' → Yield x xs'
    next_append (Left xs) =
       case next_stream xs of
          Done      → Skip  (Right ys)
          Skip  xs' → Skip  (Left xs')
          Yield x xs' → Yield x (Left xs')
    next_append (Right ys) =
       case next_stream ys of
          Done      → Done
          Skip  ys' → Skip  (Right ys')
          Yield y ys' → Yield y (Right ys')
    go z s =
       case next_append s of
          Done      → z
          Skip  s' → go z     s'
          Yield x s' → go (z + x) s'
in go 0 (Left xs)
```

Here, $next_{stream}$ and $next_{append}$ are the stepper functions of the corresponding stream combinators and go the stream consumer of $foldl_s$.

While this loop is rather inefficient, it can be easily optimised using entirely standard techniques such as those described by Peyton Jones and Santos (1998). By inlining $next_{stream}$ into the first branch of $next_{append}$, we get a nested case distinction:

$$next_{append} \ (Left \ xs) \ =$$
$$\textbf{case}$$
$$\quad \textbf{case } xs \textbf{ of}$$
$$\quad\quad [\,] \quad\quad \rightarrow \ Done$$
$$\quad\quad x : xs' \ \rightarrow \ Yield \ x \ xs'$$
$$\textbf{of}$$
$$\quad Done \quad\quad \rightarrow \ Skip \quad (Right \ ys)$$
$$\quad Skip \quad xs' \ \rightarrow \ Skip \quad (Left \ xs')$$
$$\quad Yield \ x \ xs' \ \rightarrow \ Yield \ x \ (Left \ xs')$$

This term are easily improved by applying the *case-of-case transformation* which pushes the outer *case* into the alternatives of the inner *case*:

$$next_{append} \ (Left \ xs) \ =$$
$$\textbf{case } xs \textbf{ of}$$
$$[\,] \quad\quad \rightarrow \ \textbf{case } Done \textbf{ of}$$
$$\quad\quad\quad\quad Done \quad\quad \rightarrow \ Skip \quad (Right \ ys)$$
$$\quad\quad\quad\quad Skip \quad xs' \ \rightarrow \ Skip \quad (Left \ xs')$$
$$\quad\quad\quad\quad Yield \ x \ xs' \ \rightarrow \ Yield \ x \ (Left \ xs')$$
$$x : xs' \rightarrow \ \textbf{case } Yield \ x \ xs' \textbf{ of}$$
$$\quad\quad\quad\quad Done \quad\quad \rightarrow \ Skip \quad (Right \ ys)$$
$$\quad\quad\quad\quad Skip \quad xs' \ \rightarrow \ Skip \quad (Left \ xs')$$
$$\quad\quad\quad\quad Yield \ x \ xs' \ \rightarrow \ Yield \ x \ (Left \ xs')$$

This code trivially rewrites to:

$$next_{append} \ (Left \ xs) \ = \ \textbf{case } xs \textbf{ of}$$
$$\quad\quad [\,] \quad\quad \rightarrow \ Skip \quad (Right \ ys)$$
$$\quad\quad x : xs' \rightarrow \ Yield \ x \ (Left \ xs')$$

The $Right$ branch of $next_{append}$ is simplified in a similar manner, resulting in

$$next_{append} \ (Right \ ys) \ = \ \textbf{case } ys \textbf{ of}$$
$$\quad\quad [\,] \quad\quad \rightarrow \ Done$$
$$\quad\quad y : ys' \rightarrow \ Yield \ y \ (Right \ ys')$$

Note how by inlining, applying the case-of-case transformation and then simplifying we have eliminated the construction (in $next_{stream}$) and inspection (in $next_{append}$) of one $Step$ value per iteration. The good news is that these techniques are an integral part of GHC's optimiser and are applied to our programs automatically. Indeed, the optimiser then inlines $next_{append}$ into the body of go and reapplies the transformations described above to produce:

```
let go z (Left xs)  = case xs of
                        []      →  go z          (Right ys)
                        x : xs' →  go (z + x) (Left xs')
    go z (Right ys) = case ys of
                        []      →  z
                        y : ys' →  go (z + y) (Right ys')
in go 0 (Left xs)
```

While this loop does not use any intermediate $Step$ values, it still allocates $Left$ and $Right$ for maintaining the loop state. Eliminating these requires more sophisticated techniques than we have used so far. Fortunately, *constructor specialisation* (Peyton Jones 2007), an optimisation which has been implemented in GHC for some time, does precisely this. It analyses the shapes of the arguments in recursive calls to go and produces two specialised versions of the function, go_1 and go_2, which satisfy the following equivalences:

$$\forall \ z \ xs. \ go \ z \ (Left \ xs) \ = \ go_1 \ z \ xs$$
$$\forall \ z \ ys. \ go \ z \ (Right \ ys) \ = \ go_2 \ z \ ys$$

The compiler then replaces calls to go by calls to a specialised version whenever possible. The definitions of the two specialisa-

tions are obtained by expanding go once in each of the above two equations and simplifying, which ultimately results in the following program:

```
let go₁ z xs = case xs of
                 []      →  go₂ z       ys
                 x : xs' →  go₁ (z + x) xs'
    go₂ z ys = case ys of
                 []      →  z
                 y : ys' →  go₂ (z + y) ys'
in go₁ 0 xs
```

Note that the original version of go is no longer needed. The loop has effectively been split in two parts — one for each of the two concatenated lists. Indeed, this result is the best we could have hoped for. Not only have all intermediate data structures been eliminated, the loop has also been specialised for the algorithm at hand.

By now, it becomes obvious that in order to compile stream programs to efficient code, all stream combinators *must* be inlined and subsequently specialised. Arguably, this is a weakness of our approach, as this sometimes results in excessive code duplication and a significant increase in the size of the generated binary. However, as discussed in Section 8, our experiments suggest that this increase is almost always negligible.

7.2 Nested computations

So far, we have only considered the steps necessary to optimise fused pipelines of flat operations on lists. For nested operations such as $concatMap$, the story is more complicated. Nevertheless, it is crucial that such operations are optimised well. Indeed, even our introductory example uses $concatMap$ under the hood as described in Section 5.

Although a detailed explanation of how GHC derives the efficient loop presented in the introduction would take up too much space, we can investigate a less complex example which, nevertheless, demonstrates the basic principles underlying the simplification of nested stream programs. In the following, we consider the simple list comprehension $sum \ [m * m \ | \ m \ \leftarrow \ [1..n]]$. After desugaring and stream fusion, the term is transformed to (we omit the trivial $guard$):

$$foldl_s \ (+) \ 0 \ (concatMap_s \ (\lambda m. \ return_s \ (m * m))$$
$$(enumFromTo_s \ 1 \ n))$$

After inlining the definitions of the stream functions, we arrive at the following loop ($next_{enum}$, $next_{cm}$ and $next_{ret}$ are the stepper functions of $enumFromTo_s$, $concatMap_s$ and $return_s$, respectively, as defined in Figures 1 and 2):

```
let next_enum i | i > n    = Done
                | otherwise = Yield i (i + 1)

    next_concatMap (i, Nothing) =
      case next_enum i of
        Done    → Done
        Skip   i' → Skip (i', Nothing)
        Yield x i' → let next_ret True  = Yield (x * x) False
                         next_ret False = Done
                     in
                       Skip (i', Just (Stream next_ret True))
    next_concatMap (i, Just (Stream next s)) =
      case next s of
        Done    → Skip   (i, Nothing)
        Skip   s' → Skip   (i, Just (Stream next s'))
        Yield y s' → Yield y (i, Just (Stream next s'))

    go z s = case next_concatMap s of
               Done    → z
               Skip   s' → go z       s'
               Yield x s' → go (z + x) s'
in go 0 (1, Nothing)
```

As before, we now inline $next_{enum}$ and $next_{concatMap}$ into the body of go and repeatedly apply the case-of-case transformation. Ultimately, this produces the following loop:

```
let go z (i, Nothing) | i > n    = z
                      | otherwise =
        let next_ret True = Yield (i * i) False
            next_ret False = Done
        in
          go z (i + 1, Just (Stream next_ret True))
      go z (i, Just (Stream next s)) =
        case next s of
            Done      → go z       (i, Nothing)
            Skip    s' → go z       (i, Just (Stream next s'))
            Yield x s' → go (z + x) (i, Just (Stream next s'))
in go 0 (1, Nothing)
```

Now we again employ constructor specialisation to split go into two mutually recursive functions go_1 and go_2 such that

$$\forall z\, i. \qquad go\ z\ (i,\ Nothing) \qquad\qquad = go_1\ z\ i$$
$$\forall z\, i\ next\ s.\ go\ z\ (i,\ Just\ (Stream\ next\ s)) = go_2\ z\ i\ next\ s$$

The second specialisation is interesting in that it involves an existential component — the state of the stream. Thus, go_2 must have a polymorphic type which, however, is easily deduced by the compiler. After simplifying and rewriting calls to go, we arrive at the following code:

```
let go_1 z i | i > n    = z
             | otherwise =
        let next_ret True = Yield (i * i) False
            next_ret False = Done
        in
          go_2 z (i + 1) next_ret True
      go_2 z i next s = case next s of
                            Done      → go_1 z        i
                            Skip    s' → go_2 z        i next s'
                            Yield x s' → go_2 (z + x) i next s'
in go_1 0 1
```

The loop has now been split into two mutually recursive functions. The first, go_1, computes the next element i of the enumeration $[1..n]$ and then passes it to go_2 which computes the product and adds it to the accumulator z. However, the nested structure of the original loop obscures this simple algorithm. In particular, the stepper function $next_{ret}$ of the stream produced by $return_s$ has to be passed from go_1, where it is defined, to go_2, where it is used. If we are to produce efficient code, we must remove this indirection and allow $next_{ret}$ to be inlined in the body of go_2. In the following, we consider two approaches to solving this problem: static argument transformation and specialisation on partial applications.

Static argument transformation   It is easy to see that $next$ and i are *static* in the definition of go_2, i.e., they do not change between iterations. An optimising compiler can take advantage of this fact and eliminate the unnecessary arguments:

```
go_2 z i next s =
    let go_2' z s = case next s of
                        Done      → go_1 z i
                        Skip    s' → go_2' z s'
                        Yield x s' → go_2' (z + x) s'
    in go_2' z s
```

With this definition, go_2 can be inlined in the body of go_1. Subsequent simplification binds $next$ to $next_{ret}$ and allows the latter to be inlined in go_2':

```
go_1 z i | i > n    = z
         | otherwise =
    let go_2' z True = go_2' (z + i * i) False
        go_2' z False = go_1 z (i + 1)
    in
      go_2' z True
```

The above can now be easily rewritten to the optimal loop:

```
go_1 z i | i > n    = z
         | otherwise = go_1 (z + i * i) (i + 1)
```

Note how the original nested loop has been transformed into a flat one. This is only possible because in this particular example, the function argument of *concatMap* was not itself recursive. More complex nesting structures, in particular nested list comprehensions, are translated into nested loops if the static argument transformation is employed. For instance, our introductory example would be compiled to

```
let go_1 z k | k > n    = z
             | otherwise =
        let go_2 z m | m > k    = go_1 z (k + 1)
                     | otherwise = go_2 (z + k * m) (m + 1)
        in go_2 z 1
in go_1 0 1
```

Specialisation   An alternative approach to optimising the program is to lift the definition of $next_{ret}$ out of the body of go_1 according to the algorithm of Johnsson (1985):

```
next_ret i True    = Yield (i * i) False
next_ret i False    = Done

go_1 z i | i > n    = z
         | otherwise = go_2 z (i + 1) (next_ret i) True
```

Now, we can once more specialise go_2 for this call; but this time, in addition to constructors we also specialise on the *partial application* of the now free function $next_{ret}$, producing a go_3 such that:

$$\forall z\ i\ j.\ go_2\ z\ j\ (next_{ret}\ i)\ True = go_3\ z\ j\ i$$

After expanding go_2 once in the above equation, we arrive at the following unoptimised definition of go_3:

```
go_3 z j i = case next_ret i True of
                Done      → go_1 z j
                Skip    s' → go_2 z        j (next_ret i) s'
                Yield x s' → go_2 (z + x) j (next_ret i) s'
```

Note that the stepper function is now statically known and can be inlined which allows all case distinctions to be subsequently eliminated, leading to a quite simple definition:

```
go_3 z j i = go_2 (z + (i * i)) j (next_ret i) False
```

The above call gives rise to yet another specialisation of go_2:

$$\forall z\ i\ j.\ go_2\ z\ j\ (next_{ret}\ i)\ False = go_4\ z\ j\ i$$

Again, we rewrite go_4 by inlining $next_{ret}$ and simplifying, ultimately producing:

```
go_1 z i | i > n    = z
         | otherwise = go_3 z (i + 1) i
go_3 z j i           = go_4 (z + (i * i)) j i
go_4 z j i           = go_1 z j
```

This is trivially rewritten to exactly the same code as has been produced by the static argument transformation:

```
go_1 z i | i > n    = z
         | otherwise = go_1 (z + i * i) (i + 1)
```

This convergence of the two optimisation techniques is, however, only due to the simplicity of our example. For the more deeply nested program from the introduction, specialisation would produce two mutually recursive functions:

```
go_1 z k     | k > n    = z
             | otherwise = go_2 z k (k + 1) 1

go_2 z k k' m | m > k    = go_1 z k'
              | otherwise = go_2 (z + k * m) k k' (m + 1)
```

This is essentially the code of f' from the introduction; the only difference is that GHC's optimiser has unrolled go_2 once and unboxed all loop variables. This demonstrates the differences between the two approaches nicely. The static argument transformation translates nested computations into nested recursive functions. Specialisation on partial applications, on the other hand, produces *flat* loops with several mutually recursive functions. The state of such a loop is maintained in explicit arguments.

Unfortunately, GHC currently supports neither of the two approaches — it only specialises on constructors but not on partial applications of free functions and does not perform the static argument transformation. Although we have extended GHC's optimiser with both techniques, our implementation is quite fragile and does not always produce the desired results. Indeed, missed optimisations are at least partly responsible for many of the performance problems discussed in Section 8. At this point, the implementation must be considered merely a proof of concept. We are, however, hopeful that GHC will be able to robustly optimise stream-based programs in the near future.

8. Results

We have implemented the entire Haskell standard List library, including enumerations and list comprehensions, on top of our stream fusion framework. Stream fusion is implemented via equational transformations embedded as rewrite rules in the library source code. We compare time, space, code size and fusion opportunities for programs in the *nofib* benchmark suite (Partain 1992), when compared to the existing *build/foldr* system. To ensure a fair comparison, both frameworks have been benchmarked with our extensions to GHC's optimiser (cf. Section 7) enabled. For the *build/foldr* framework, these extensions do not significantly affect the running time and allocation behaviour, usually improving them slightly, and without them, nested *concatMap*'s under stream fusion risk not being optimised.

8.1 Time

Figure 4 presents the relative speedups for Haskell programs from the *nofib* suite, when compared to the existing *build/foldr* system. On average, there is a 3% improvement when using stream fusion, with 6 of the test programs more than 15% faster, and one, the 'integer' benchmark, more than 50% faster. One program, 'paraffins', ran 24% slower, due to remnant *Stream* data constructors not statically removed by the compiler.

In general we can divide the results into three classes of programs:

1. those for which there is plenty of opportunity for fusion which is under-exploited by *build/foldr*;
2. programs for which there is little fusion or for which the fusion is in a form handled by *build/foldr*;
3. and thirdly, programs such as 'paraffins' with deeply nested list computations and comprehensions which overtax our extensions to GHC's optimiser.

For the first class or programs, those using critical left folds and zip, stream fusion can be a big win. 10% (and sometimes much more) improvement is not uncommon. This corresponds to around 15% of programs tested.

In the second case, the majority of programs covered, there is either little available fusion, or the fusion is in the form of right folds, and list comprehensions, already well handled by *build/foldr*. Only small improvements can be expected here.

Finally, the third class, corresponds to some 5% of programs tested. These programs have available fusion, but in deeply nested form, which can lead to *Step* constructors left behind by limitations

in current GHC optimisations, rather than being removed statically. These programs currently will run dramatically, and obviously, worse.

For large multi-module programs, the results are less clear, with just as many programs speeding up as slowing down. We find that for larger programs, GHC has a tendency to miss optimisation opportunities for stream fusible functions across module boundaries, which is the subject of further investigation.

8.2 Space

Figure 5 presents the relative reduction in total heap allocations for stream fusion programs compared to the existing *build/foldr* system. The results can again be divided into the same three classes as for the time benchmarks: those with under-exploited fusion opportunities, those for which *build/foldr* already does a good job, and those for which *Step* artifacts are not statically eliminated by the compiler.

For programs which correctly fuse, in the first class, with new fusion opportunities found by stream fusion, there can be dramatic reductions in allocations (up to 30%). Currently, this is the minority of programs. The majority of programs have modest reductions, with an average decrease in allocations of 4.4%. Two programs have far worse allocation performance, however, due to missed opportunities to remove *Step* constructors in nested code. For large, multi-module programs, we find a small increase in allocations, for similar reasons as for the time benchmarks.

8.3 Fusion opportunities

In Figure 6 we compare the number of fusion sites identified with stream fusion, compared to that with *build/foldr*. In the majority of cases, more fusion sites are identified, corresponding to new fusion opportunities with zips and left folds. Similar results are seen when compiling GHC itself, with around 1500 *build/foldr* fusion sites identified by the compiler, and more than 3000 found under stream fusion.

8.4 Code size

Total compiled binary size was measured, and we find that for single module programs, code size increases by a negligible 2.5% on average. For multi-module programs, code size increases by 11%. 5% of programs increased by more than 25% in size, again due to unremoved specialised functions and *Step* constructors.

9. Further work

9.1 Improved optimisations

The main direction for future work on stream fusion is to improve further the compiler optimisations required to remove *Step* constructors statically, as described in Section 7.

Another possible approach to reliably fusing nested uses of *concatMap* is to define a restricted version of it which assumes that the inner stream is constructed in a uniform way, i.e., using the same stepper function and the same function to construct initial inner-stream states in every iteration of the outer stream. This situation corresponds closely to the forms that we expect to be able to optimise with the static argument transformation.

The aim would be to have a rule that matches the common situation where this restricted *concatMap* can be used. Unfortunately such rules cannot be expressed in the current GHC rules language. A more powerful rule matcher would allow us to write something like:

$$concatMap \ (\lambda \, x \ \rightarrow unstream \ (Stream \ next[\![x]\!] \ s[\![x]\!]))$$
$$= concatMap' \ (\lambda \, y \ \rightarrow next[\![y]\!]) \ (\lambda \, y \ \rightarrow s[\![y]\!])$$
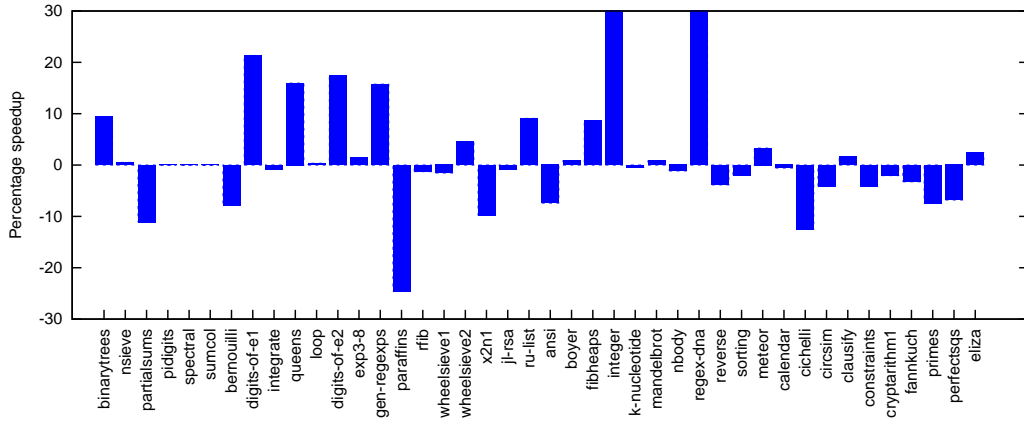
Figure 4: Percentage improvement in running time compared to *build/foldr* fusion
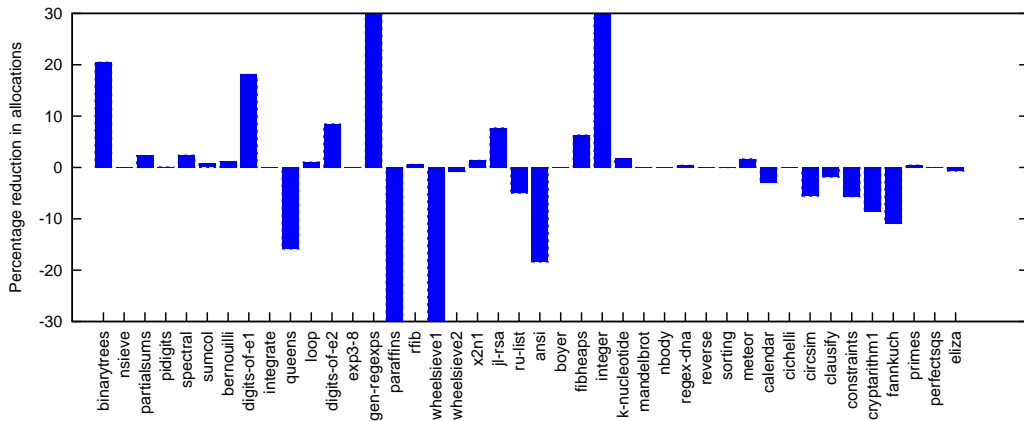

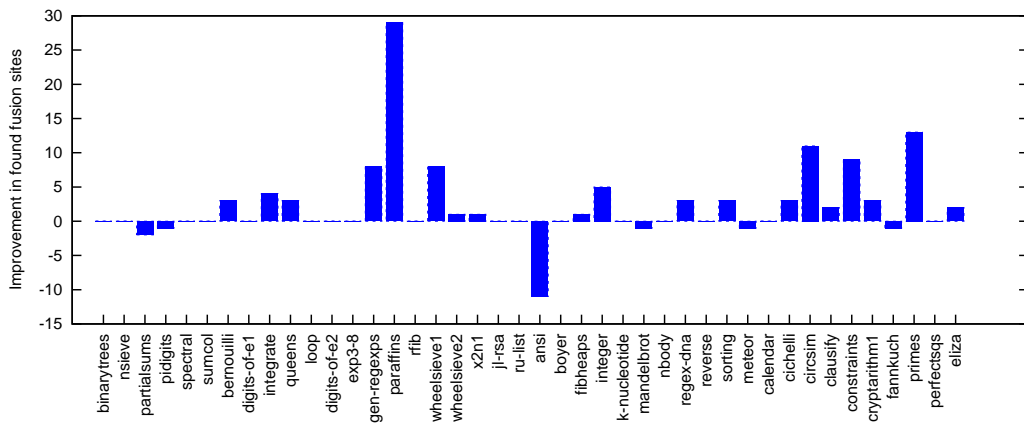Figure 5: Percent reduction in allocations compared to *build/foldr* fusion

Figure 6: New fusion opportunities found when compared to *build/foldr*

Here, $T[\![x]\!]$ matches a term T abstracted over free occurrences of the variable x. In the right hand side the same syntax indicates substitution.

The key point is that the stepper function of the inner stream is statically known in $concatMap'$. This is a much more favourable situation compared to embedding the entire inner stream in the seed of $concatMap$. Indeed, extending GHC's rule matching capabilities in this direction might be easier than robustly implementing the optimisations outlined in Section 7.

9.2 Fusing general recursive definitions

Writing stream stepper functions is not always easy. The representation of control flow as state makes them appear somewhat inside out. One technique we found useful when translating Haskell list functions into stream fusible versions was to first transform the list version to very low level Haskell. From this form, where the precise control flow is clear, there is a fairly direct translation into a stream version.

For example here is a list function written in a very low level style using three mutually tail-recursive functions. Each one has only simple patterns on the left hand side and on the right hand side: an empty list; a call; or a cons and a call.

$$
\begin{array}{l}
intersperse \ :: \ a \ \rightarrow [a] \rightarrow [a] \\
intersperse \ sep \ xs_0 \ = \ init \ xs_0 \\
\quad \textbf{where} \\
\quad\quad init \ xs \ = \ \textbf{case} \ xs \ \textbf{of} \\
\quad\quad\quad [\,] \quad\quad \rightarrow [\,] \\
\quad\quad\quad (x : xs) \rightarrow go \ x \ xs \\
\quad\quad go \ x \ xs = x \ : \ to \ xs \\
\quad\quad to \ xs \quad = \ \textbf{case} \ xs \ \textbf{of} \\
\quad\quad\quad [\,] \quad\quad \rightarrow [\,] \\
\quad\quad\quad (x : xs) \rightarrow sep \ : \ go \ x \ xs
\end{array}
$$

We can translate this to an equivalent function on streams by making a data type with one constructor per function. Each constructor holds the arguments to that function except that arguments of type list are replaced by the stream state type. In the body of each function, case analysis on lists is replaced by calling $next_0$ on the stream state. Consing an element onto the result is replaced by uses of $Yield$. Calls are replaced by $Skip$s with the appropriate state data constructor:

$$
\begin{array}{l}
\textbf{data} \ State \ a \ s \ = \ Init \ s \mid Go \ a \ s \mid To \ s \\
\\
intersperse_s \ :: \ a \rightarrow Stream \ a \rightarrow Stream \ a \\
intersperse_s \ sep \ (Stream \ next_0 \ s_0) \ = \ Stream \ next \ (Init \ s_0) \\
\quad \textbf{where} \\
\quad\quad next \ (Init \ s) \ = \ \textbf{case} \ next_0 \ s \ \textbf{of} \\
\quad\quad\quad Done \quad\quad \rightarrow Done \\
\quad\quad\quad Skip \quad s' \rightarrow Skip \ (Init \ s') \\
\quad\quad\quad Yield \ x \ s' \rightarrow Skip \ (Go \ x \ s') \\
\quad\quad next \ (Go \ x \ s) = \ Yield \ x \ (To \ s) \\
\quad\quad next \ (To \ s) \quad = \ \textbf{case} \ next_0 \ s \ \textbf{of} \\
\quad\quad\quad Done \quad\quad \rightarrow Done \\
\quad\quad\quad Skip \quad s' \rightarrow Skip \quad\quad (To \ s') \\
\quad\quad\quad Yield \ x \ s' \rightarrow Yield \ sep \ (Go \ x \ s')
\end{array}
$$

It would be interesting to investigate the precise restrictions on the form which can be translated in this way and whether it can be automated. This might provide a practical way to fuse general recursive definitions over lists: by checking if the list function can be translated to the restricted form and then translating into a stream version. There is some precedent for this approach: Launchbury and Sheard (1995) show that in many common cases it is possible to transform general recursive definitions on lists into a form suitable for use with ordinary *build/foldr* short-cut fusion.

9.3 Fusing more general algebraic data types

It seems straightforward to define a co-structure for any sum-of-products data structure. Consider for example a binary tree type with information in both the leaves and interior nodes:

$$\textbf{data} \ Tree \ a \ b \ = \ Leaf \ a \mid Fork \ b \ (Tree \ a \ b) \ (Tree \ a \ b)$$

The corresponding co-structure would be:

$$
\begin{array}{ll}
\textbf{data} \ Stream \ a \ b \ = & \exists s. \ Stream \ (s \rightarrow Step \ a \ b \ s) \ s \\
\textbf{data} \ Step \quad a \ b \ s = & Leaf_s \ a \mid Fork_s \ b \ s \ s \mid Skip \ s
\end{array}
$$

Of course other short-cut fusion systems can also be generalised in this way but in practice they are not because it requires defining a new infrastructure for each new data structure that we wish to fuse. Automation would be required to make this practical. This problem is somewhat dependent on the ability to generate stream style code from ordinary recursive definitions.

10. Conclusion

It is possible, via stream fusion, to automatically fuse a complete range of list functions, beyond that of previous short-cut fusion techniques. In particular, it *is* possible to fuse left and right folds, zips, concats and nested lists, including list comprehensions. For the first time, details are provided for the range of general purpose optimisations required to generate efficient code for *unfoldr-based* short-cut fusion.

Stream fusion is certainly practical, with a full scale implementation for Haskell lists being implemented, utilising library-based rewrite rules for fusion. Our results indicate there is a greater opportunity for fusion, than under the existing *build/foldr* system, and also show moderate improvements in space and time performance. Further improvements in the specific compiler optimisations required to remove fusion artifacts statically.

The source code for the stream fusion List library, and modified standard Haskell library and compiler, are available online.[2]

References

Olaf Chitil. Type inference builds a short cut to deforestation. In *ICFP '99: Proceedings of the fourth ACM SIGPLAN International Conference on Functional programming*, pages 249–260, New York, NY, USA, 1999. ACM Press.

Olaf Chitil. Promoting non-strict programming. In *Draft Proceedings of the 18th International Symposium on Implementation and Application of Functional Languages, IFL 2006*, pages 512–516, Budapest, Hungary, September 2006. Eotvos Lorand University. URL http://www.cs.kent.ac.uk/pubs/2006/2477.

Duncan Coutts, Don Stewart, and Roman Leshchinskiy. Rewriting Haskell strings. In *Practical Aspects of Declarative Languages 8th International Symposium, PADL 2007*, pages 50–64. Springer-Verlag, January 2007.

Nils Anders Danielsson and Patrik Jansson. Chasing bottoms, a case study in program verification in the presence of partial and infinite values. In Dexter Kozen, editor, *Proceedings of the 7th International Conference on Mathematics of Program Construction, MPC 2004*, volume 3125 of *LNCS*, pages 85–109. Springer-Verlag, July 2004.

---

[2] This material can be found on the website accompanying this paper at http://www.cse.unsw.edu.au/~dons/papers/CLS07.html.

Jeremy Gibbons. Streaming representation-changers. In D. Kozen, editor, *Mathematics of Program Construction*, pages 142–168. Springer-Verlag, 2004. LNCS 523.

Jeremy Gibbons and Geraint Jones. The under-appreciated unfold. In *ICFP '98: Proceedings of the third ACM SIGPLAN International Conference on Functional programming*, pages 273–279, New York, NY, USA, 1998. ACM Press.

Andrew Gill. *Cheap Deforestation for Non-strict Functional Languages*. PhD thesis, University of Glasgow, January 1996.

Andrew Gill, John Launchbury, and Simon Peyton Jones. A short cut to deforestation. In *Conference on Functional Programming Languages and Computer Architecture*, pages 223–232, June 1993.

Zhenjiang Hu, Hideya Iwasaki, and Masato Takeichi. Deriving structural hylomorphisms from recursive definitions. In *Proceedings 1st ACM SIGPLAN International Conference on Functional Programming*, volume 31(6), pages 73–82. ACM Press, New York, 1996.

Patricia Johann. Short cut fusion: Proved and improved. In *SAIG 2001: Proceedings of the Second International Workshop on Semantics, Applications, and Implementation of Program Generation*, pages 47–71, London, UK, 2001. Springer-Verlag.

Thomas Johnsson. Lambda lifting: transforming programs to recursive equations. In *Functional programming languages and computer architecture. Proc. of a conference (Nancy, France, Sept. 1985)*, New York, NY, USA, 1985. Springer-Verlag Inc.

John Launchbury and Ross Paterson. Parametricity and unboxing with unpointed types. In *European Symposium on Programming*, pages 204–218, 1996.

John Launchbury and Tim Sheard. Warm fusion: deriving build-catas from recursive definitions. In *FPCA '95: Proceedings of the seventh international conference on Functional programming languages and computer architecture*, pages 314–323, New York, NY, USA, 1995. ACM Press.

Erik Meijer, Maarten Fokkinga, and Ross Paterson. Functional programming with bananas, lenses, envelopes and barbed wire. In J. Hughes, editor, *Proceedings 5th ACM Conf. on Functional Programming Languages and Computer Architecture, FPCA'91, Cambridge, MA, USA, 26–30 Aug 1991*, volume 523, pages 124–144. Springer-Verlag, Berlin, 1991.

Will Partain. The nofib benchmark suite of Haskell programs. In *Functional Programming*, pages 195–202, 1992.

Simon Peyton Jones. Constructor Specialisation for Haskell Programs, 2007. Submitted for publication.

Simon Peyton Jones and André L. M. Santos. A transformation-based optimiser for Haskell. *Sci. Comput. Program.*, 32(1-3): 3–47, 1998. ISSN 0167-6423.

Simon Peyton Jones, Andrew Tolmach, and Tony Hoare. Playing by the rules: rewriting as a practical optimisation technique in GHC. In Ralf Hinze, editor, *2001 Haskell Workshop*. ACM SIGPLAN, September 2001.

Simon Peyton Jones et al. The Haskell 98 language and libraries: The revised report. *Journal of Functional Programming*, 13(1): 0–255, Jan 2003. `http://www.haskell.org/definition/`.

Colin Runciman. SmallCheck 0.2: another lightweight testing library in Haskell. `http://article.gmane.org/gmane.comp.lang.haskell.general/14461`, 2006.

Josef Svenningsson. Shortcut fusion for accumulating parameters & zip-like functions. In *ICFP '02: Proceedings of the seventh ACM SIGPLAN International Conference on Functional programming*, pages 124–132, New York, NY, USA, 2002. ACM Press.

Akihiko Takano and Erik Meijer. Shortcut deforestation in calculational form. In *Conf. Record 7th ACM SIGPLAN/SIGARCH Int. Conf. on Functional Programming Languages and Computer Architecture, FPCA '95*, pages 306–313. ACM Press, New York, 1995.

The GHC Team. The Glasgow Haskell Compiler (GHC). `http://haskell.org/ghc`, 2007.

Philip Wadler. List comprehensions. In Simon Peyton Jones, editor, *The implementation of functional programming languages*. Prentice Hall, 1987. Chapter 15.

Philip Wadler. Deforestation: transforming programs to eliminate trees. *Theoretical Computer Science, (Special issue of selected papers from 2nd European Symposium on Programming)*, 73(2): 231–248, 1990.