Stream Fusion

From Lists to Streams to Nothing at All

Duncan Coutts | Roman Leshchinskiy | Don Stewart

Programming Tools Group, Oxford University

Programming Languages and Systems, University of New South Wales

LISTS IN HASKELL

List processing can be beautiful:

$$f :: Int \rightarrow Int$$
$$f\ n\ =\ \mathbf{sum}\,[\,k * m \mid k \leftarrow [1..n],\ m \leftarrow [1..k]\,]$$

Concise syntax for complex nested loops.

This is the code we *want* to write.

And this is the code we get...

$$
\begin{array}{l}
f \;::\; Int\# \to Int\# \\
f\ n \;=\; \textbf{sum}\ 0\ (\\
\quad \textbf{case}\ 1 > n\ \textbf{of} \\
\qquad True \to [\,] \\
\qquad False \to \\
\qquad \textbf{let} \\
\qquad\quad go \;::\; Int\# \to [Int] \\
\qquad\quad go\ x \;=\; \textbf{let} \\
\qquad\qquad\qquad ds \;=\; \textbf{case}\ x == n\ \textbf{of} \\
\qquad\qquad\qquad\qquad True \to [\,] \\
\qquad\qquad\qquad\qquad False \to go\ (x+1) \\
\qquad\qquad\quad \textbf{in} \\
\qquad\qquad\quad \textbf{case}\ 1 > x\ \textbf{of} \\
\qquad\qquad\qquad True \to ds \\
\qquad\qquad\qquad False \to \textbf{let} \\
\qquad\qquad\qquad\qquad to\ y \;=\; I\#\ (x * y)\ :\ \textbf{case}\ y == x\ \textbf{of} \\
\qquad\qquad\qquad\qquad\qquad False \to to\ (y+1) \\
\qquad\qquad\qquad\qquad\qquad True \to ds \\
\qquad\qquad\qquad\quad \textbf{in} \\
\qquad\qquad\qquad\quad to\ 1 \\
\qquad \textbf{in} \\
\qquad go\ 1)
\end{array}
$$

Generating better code

Problem:

- ➜ Intermediate list is allocated, only to be immediately consumed!
- ➜ We need to combine the sum and list comprehension loops: *fusion*
- ➜ But some key functions, like $foldl$ (or sum) are hard to fuse using existing systems

We need fusion for $zips$, $foldls$ and $concatMaps$/list comprehensions.

STREAM FUSION

Final code under stream fusion:

$$
\begin{array}{l}
f' \;::\; Int\# \to Int\# \\
f'\; n \;=\; go\; 0\; 1 \\
\quad \textbf{where} \\
\qquad go\; s\; k \qquad = \textbf{case}\; k > n\; \textbf{of} \\
\qquad\qquad\qquad\quad False \to \textbf{case}\; 1 > k\; \textbf{of} \\
\qquad\qquad\qquad\qquad\qquad True \to go\; s\; (k + 1) \\
\qquad\qquad\qquad\qquad\qquad False \to to\; (s + k)\; k\; (k + 1)\; 2 \\
\qquad\qquad\qquad\quad True \to s \\
\qquad to\; s\; k\; j\; m = \textbf{case}\; m > k\; \textbf{of} \\
\qquad\qquad\qquad\quad False \to to\; (\mathbf{s} + (\mathbf{k} * \mathbf{m}))\; k\; j\; (m + 1) \\
\qquad\qquad\qquad\quad True \to go\; s\; j
\end{array}
$$

No intermediate list. Better code. Faster code!

INTRODUCTION TO FUSION

The Big Idea: Deforestation and Fusion

Ideally, pipelines on lists would just make one traversal

We'd write:

```
map f . map g
```

and the compiler would emit:

```
map (f . g)
```

No side-effects, so its ok!

We can teach GHC to do this with *rewrite rules*:

$$\langle \textbf{map/map fusion} \rangle\ \forall\, f\ g\ .$$
$$\textit{map}\ f \cdot \textit{map}\ g \mapsto \textit{map}\ (f \cdot g)$$

But how to fuse other combinations of list functions?

The Big Idea: Deforestation and Fusion

Ideally, pipelines on lists would just make one traversal

We'd write:

```
map f . map g
```

and the compiler would emit:

```
map (f . g)
```

No side-effects, so its ok!

We can teach GHC to do this with *rewrite rules*:

$$\langle\textbf{map/map fusion}\rangle\ \forall\, f\ g\ .$$
$$\textit{map}\ f \cdot \textit{map}\ g \ \mapsto\ \textit{map}\ (f \cdot g)$$

But how to fuse other combinations of list functions?

The Big Idea: Deforestation and Fusion

Ideally, pipelines on lists would just make one traversal

We'd write:

```
map f . map g
```

and the compiler would emit:

```
map (f . g)
```

No side-effects, so its ok!

We can teach GHC to do this with *rewrite rules*:

$$\langle \textbf{map/map fusion} \rangle \; \forall\, f \;\, g \;.$$
$$\textit{map} \;\, f \cdot \textit{map} \;\, g \;\mapsto\; \textit{map} \;\, (f \cdot g)$$

But how to fuse other combinations of list functions?

FUSION SYSTEMS

A variety of general purpose fusion systems exist

In particular:

$$\langle \textbf{build/foldr fusion} \rangle \ \forall\ g\ k\ z\ .$$
$$\textit{foldr}\ k\ z\ (\textit{build}\ g) \ \mapsto\ g\ k\ z$$

(Gill, Launchbury, Peyton Jones ’93)

$$\langle \textbf{destroy/unfoldr fusion} \rangle \ \forall\ g\ f\ e\ .$$
$$\textit{destroy}\ g\ (\textit{unfoldr}\ f\ e) \ \mapsto\ g\ f\ e$$

(Svenningsson ’02)

Write *your* functions in terms of *these* functions and they will fuse.

Fusion systems

A variety of general purpose fusion systems exist

In particular:

$$\langle \textbf{build/foldr fusion} \rangle \; \forall \, g \; k \; z \; .$$
$$\textit{foldr} \; k \; z \; (\textit{build} \; g) \; \mapsto \; g \; k \; z$$

(Gill, Launchbury, Peyton Jones ’93)

$$\langle \textbf{destroy/unfoldr fusion} \rangle \; \forall \, g \; f \; e \; .$$
$$\textit{destroy} \; g \; (\textit{unfoldr} \; f \; e) \; \mapsto \; g \; f \; e$$

(Svenningsson ’02)

Write *your* functions in terms of *these* functions and they will fuse.

Limitations

But some functions are hard to write using these functions.

The usual suspects:

- zip, $zipWith$ and friends
- $foldl$ and other left folds ($length$, sum, $minimum$)
- nested list functions ($concatMap$, list comprehensions)

And for some other functions that can fuse, we don't get efficient code ($filter$ under *destroy/unfoldr*).

STREAM FUSION

Three steps to better code:

1. Convert functions on recursive list structures into functions on non-recursive *co-structures* (the $Stream$ data type).
2. Eliminate conversions between structures and co-structures
3. Then use general purpose optimisations to fuse the co-structure code

That's all there is!

STEP 1: THE STREAM CO-STRUCTURE

STREAMS: UNFOLDED LISTS

We need an explicit representation of the unfolding of a list:

$$\begin{aligned}
&\textbf{data}\ \mathit{Stream}\ a = \exists s.\ \mathit{Stream}\ (s \to \mathit{Step}\ a\ s)\ \ s \\
&\textbf{data}\ \mathit{Step}\ a\ s\ \ = \mathit{Done} \\
&\qquad\qquad\qquad\quad \mid\ \ \mathit{Yield}\ a\ \ s \\
&\qquad\qquad\qquad\quad \mid\ \ \mathit{Skip}\ s
\end{aligned}$$

The internal state, s, of each stream is hidden.

Note the Stream constructor is a generalised unfoldr:

$$\begin{aligned}
&\mathit{unfoldr} :: \ \forall s\ a.\ (s \to \mathit{Maybe}\ (a, s)) \to s \to [a] \\
&\mathit{Stream}\ \ :: \ \forall s\ a.\ (s \to \mathit{Step}\ a\ s) \qquad\quad\ \to s \to \mathit{Stream}\ a
\end{aligned}$$

STREAMS: UNFOLDED LISTS

We need an explicit representation of the unfolding of a list:

$$\begin{aligned}
&\mathbf{data}\ \mathit{Stream}\ a = \exists s.\ \mathit{Stream}\ (s \to \mathit{Step}\ a\ s)\ \ s \\
&\mathbf{data}\ \mathit{Step}\ a\ s\ \ = \mathit{Done} \\
&\qquad\qquad\qquad\quad \mid\ \mathit{Yield}\ a\ \ s \\
&\qquad\qquad\qquad\quad \mid\ \mathit{Skip}\ s
\end{aligned}$$

The internal state, s, of each stream is hidden.

Note the Stream constructor is a generalised unfoldr:

$$\begin{aligned}
&\mathit{unfoldr} :: \ \forall s\ a.\ (s \to \mathit{Maybe}\ (a, s)) \to s \to [a] \\
&\mathit{Stream}\ :: \ \forall s\ a.\ (s \to \mathit{Step}\ a\ s) \qquad\quad \to s \to \mathit{Stream}\ a
\end{aligned}$$

Functions on streams

An example:

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

map_s simply applies f to each yielded element.

The key trick is that $next$ is always non-recursive

WRITING LIST FUNCTIONS

Assuming conversion to and from streams, we can write:

$$\begin{aligned} map &:: (a \to b) \to [a] \to [b] \\ map\, f &= unstream \cdot map_s\, f \cdot stream \end{aligned}$$

Easy.

CONVERTION LISTS TO STREAMS

Build a stream by yielding each element of the original list:

$$
\begin{aligned}
&stream :: [a] \to Stream\ a \\
&stream\ xs_0 = Stream\ next\ xs_0 \\
&\quad \mathbf{where} \\
&\qquad next\ [\,] \qquad\quad = Done \\
&\qquad next\ (x : xs) = Yield\ x\ xs
\end{aligned}
$$

Non-recursive stepper function.

CONVERTING STREAMS BACK TO LISTS

$$
\begin{array}{l}
\mathit{unstream} \ :: \ \mathit{Stream}\ a \to [a] \\
\mathit{unstream}\ (\mathit{Stream}\ \ \mathit{next}\ s_0) \ = \ \mathit{unfold}\ s_0 \\
\quad \textbf{where} \\
\qquad \mathit{unfold}\ s \ = \ \textbf{case}\ \ \mathit{next}\ s\ \textbf{of} \\
\qquad\qquad\qquad \begin{array}{lll}
\mathit{Done} & \to & [\,] \\
\mathit{Skip}\quad s' & \to & \mathit{unfold}\ s' \\
\mathit{Yield}\ x\ s' & \to & x : \mathit{unfold}\ s'
\end{array}
\end{array}
$$

- Unfold the stream by calling the stream's next function
- Unlike $\mathit{unfoldr}$, streams can Skip.
- This ensures all steppers are non-recursive.

All recursion is lifted out of the pipeline: no more fixpoints!

STEP 2: REMOVE REDUNDANT CONVERSIONS

One step back...

Now, instead of consuming and producing a list once:

- We consume a list, with $stream$, allocating $Step$ constructors
- Then transform the stream of $Step$ values
- Then, finally, destroy the stream, allocating list nodes ($unstream$)

If we compose two functions:

$$
\begin{aligned}
&map\ f\ \cdot\ map\ g\ = \\
&\quad unstream\ \cdot\ map_s\ f\ \cdot\ stream\ \cdot\ unstream\ \cdot\ map_s\ g\ \cdot\ stream
\end{aligned}
$$

we can immediately see an opportunity to eliminate a conversion!

One step back...

Now, instead of consuming and producing a list once:

- ➜ We consume a list, with $stream$, allocating $Step$ constructors
- ➜ Then transform the stream of $Step$ values
- ➜ Then, finally, destroy the stream, allocating list nodes ($unstream$)

If we compose two functions:

$$
\begin{aligned}
& map\ f \ \cdot\ map\ g\ = \\
& \quad unstream\ \cdot\ map_s\ f\ \cdot\ stream\ \cdot\ unstream\ \cdot\ map_s\ g\ \cdot\ stream
\end{aligned}
$$

we can immediately see an opportunity to eliminate a conversion!

THE “FUSION” RULE

Assuming $stream \cdot unstream$ is the identity on streams, we obtain:

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

And now GHC knows about this too – thanks to rewrite rules.

ELIMINATING CONVERSIONS BY THE RULES

Give the stream fusion rule, we have:

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

$$\{stream\ fusion\} \Rightarrow$$
$$unstream \cdot map_s\, f \cdot map_s\, g \cdot stream$$

- ➜ The pipeline is now the composition of non-recursive stream functions
- ➜ *Not* recursive list functions!
- ➜ $unstream$ runs the loop that results.

ELIMINATING CONVERSIONS BY THE RULES

Give the stream fusion rule, we have:

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

$$\{stream\ fusion\} \Rightarrow$$
$$unstream \cdot map_s\, f \cdot map_s\, g \cdot stream$$

- ➜ The pipeline is now the composition of non-recursive stream functions
- ➜ *Not* recursive list functions!
- ➜ $unstream$ runs the loop that results.

Step 3: Combining stream functions

FUSING CO-STRUCTURES

Now we need to fuse the stream co-structure functions, to eliminate intermediate $Step$ values.

But, because all stream steppers are non-recursive:

The compiler will eliminate the intermediate values on its own!

Needs:

- Inlining
- case-of-case
- constructor specialisation (new)
- Nested code needs a couple more optimisations (see the paper)

And that's it!

EXAMPLES

Fusible filters and enumerations

$$
\begin{array}{l}
\mathit{filter} \;::\; (a \to \mathit{Bool}) \to \mathit{Stream}\ a \to \mathit{Stream}\ a \\
\mathit{filter}\ p\ (\mathit{Stream}\ \mathit{next}_0\ s_0) \;=\; \mathit{Stream}\ \mathit{next}\ s_0 \\
\quad \textbf{where} \\
\qquad \mathit{next}\ s = \;\textbf{case}\ \mathit{next}_0\ s\ \textbf{of} \\
\qquad\qquad\qquad \begin{array}{lll}
\mathit{Done} & \to & \mathit{Done} \\
\mathit{Skip}\quad s' & \to & \mathit{Skip}\quad s' \\
\mathit{Yield}\ x\ s' \mid p\ x & \to & \mathit{Yield}\ x\ s' \\
\qquad\quad\;\; \mid \mathit{otherwise} & \to & \mathit{Skip}\quad s'
\end{array}
\end{array}
$$

Skip here means a non-recursive filter.

Right folds

$$
\begin{array}{l}
\mathit{foldr} \ :: \ (a \to b \to b) \to b \to \mathit{Stream}\ a \to b \\
\mathit{foldr}\ f\ z\ (\mathit{Stream}\ \mathit{next}\ s_0) \ = \ \mathit{go}\ s_0 \\
\quad \mathbf{where} \\
\qquad \mathit{go}\ s \ = \ \mathbf{case}\ \mathit{next}\ s\ \mathbf{of} \\
\qquad\qquad\qquad \begin{array}{lll}
\mathit{Done} & \to & z \\
\mathit{Skip}\quad s' & \to & \qquad \mathit{go}\ s' \\
\mathit{Yield}\ x\ s' & \to & f\ x\ (\mathit{go}\ s')
\end{array}
\end{array}
$$

Folds consume streams, and are thus recursive.

Left folds

$$
\begin{aligned}
&\mathit{foldl} \;::\; (b \to a \to b) \to b \to \mathit{Stream}\ a \to b\\
&\mathit{foldl}\ f\ z\ (\mathit{Stream}\ \mathit{next}\ s_0) \;=\; \mathit{go}\ z\ s_0\\
&\quad \textbf{where}\\
&\qquad \mathit{go}\ z\ s \;=\; \textbf{case}\ \mathit{next}\ s\ \textbf{of}\\
&\qquad\qquad \begin{array}{lll}
\mathit{Done} & \to & z\\
\mathit{Skip}\quad\ \ s' & \to & \mathit{go}\ z\ s'\\
\mathit{Yield}\ x\ s' & \to & \mathit{go}\ (f\ z\ x)\ s'
\end{array}
\end{aligned}
$$

Easy.

Complex stream states: zips

$$
\begin{array}{l}
zip :: Stream\ a \to Stream\ b \to Stream\ (a,\ b)\\
zip\ \ (Stream\ next_a\ s_{a0})\ \ (Stream\ next_b\ s_{b0})\ \ =\ \ Stream\ next\ (s_{a0},\ s_{b0},\ Nothing)\\
\quad \mathbf{where}\\
\qquad next\ (sa,\ sb,\ Nothing) = \ \mathbf{case}\ next_a\ s_a\ \mathbf{of}\\
\qquad\qquad\qquad\qquad\qquad \begin{array}{lll} Done & & \to Done\\ Skip & s'_a & \to Skip\ (s'_a,\ s_b,\ Nothing)\\ Yield & a\ s'_a & \to Skip\ (s'_a,\ s_b,\ Just\ a)\end{array}\\
\qquad next\ (s'_a,\ s_b,\ Just\ a) \ \ = \ \mathbf{case}\ next_b\ s_b\ \mathbf{of}\\
\qquad\qquad\qquad\qquad\qquad \begin{array}{llll} Done & & \to Done & \\ Skip & s'_b & \to Skip & (s'_a,\ s'_b,\ Just\ a)\\ Yield & b\ s'_b & \to Yield\ (a,\ b) & (s'_a,\ s'_b,\ Nothing)\end{array}
\end{array}
$$

To zip two streams, we need a stepper that alternates between each stream.

- ➜ Requires loop state kept in the stream ($Just/Nothing$)
- ➜ This compiles to a loop that builds Maybe values each time around
- ➜ Requires *constructor specialisation* to strip state away, generating direct calls to worker functions instead

NESTED FUNCTIONS: CONCATMAP

$$
\begin{array}{l}
concatMap :: (a \to Stream\ b) \to Stream\ a \to Stream\ b \\
concatMap\ f\ (Stream\ next_a\ s_{a0}) \ = \ Stream\ next\ (s_{a0},\ Nothing) \\
\quad \mathbf{where} \\
\qquad next\ (s_a,\ Nothing)\ = \\
\qquad\quad \mathbf{case}\ next_a\ s_a\ \mathbf{of} \\
\qquad\qquad Done \qquad\ \ \to Done \\
\qquad\qquad Skip \quad\ s'_a \to Skip\ (s'_a,\ Nothing) \\
\qquad\qquad Yield\ a\ s'_a \to Skip\ (s'_a,\ Just\ (f\ a)) \\
\qquad next\ (s_a,\ Just\ (Stream\ next_b\ s_b))\ = \\
\qquad\quad \mathbf{case}\ next_b\ s_b\ \mathbf{of} \\
\qquad\qquad Done \qquad\ \ \to Skip \quad\ (s_a,\ Nothing) \\
\qquad\qquad Skip \quad\ s'_b \to Skip \quad\ (s_a,\ Just\ (Stream\ next_b\ s'_b)) \\
\qquad\qquad Yield\ b\ s'_b \to Yield\ b\ (s_a,\ Just\ (Stream\ next_b\ s'_b))
\end{array}
$$

Fusible with on its input *and* output list:

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

COMPILING AND OPTIMISING STREAM CODE

COMPILING STREAMS CODE

Let's compile this sum of squares code:

$$sum\ [m * m \mid m \leftarrow [1..n]]$$

Desugars to:

$$\begin{aligned} foldl_s\ (+)\ 0\ (concatMap_s\ (\lambda m.\ return_s\ (m * m))& \\ (enumFromTo_s\ 1\ n))& \end{aligned}$$

Using the streams desugaring of comprehensions (see paper).

COMPILING STREAMS CODE

Let's compile this sum of squares code:

$$sum\ [m * m \mid m \leftarrow [1..n]]$$

Desugars to:

$$\begin{aligned} foldl_s\ (+)\ 0\ (concatMap_s\ (\lambda m.\ return_s\ (m * m))\\ (enumFromTo_s\ 1\ n)) \end{aligned}$$

Using the streams desugaring of comprehensions (see paper).

Inline stream function:

$$
\begin{array}{l}
\textbf{let} \\
\quad \begin{array}{ll} next_{enum}\ i \mid i\ >\ n & =\ Done \\ \qquad\qquad \mid otherwise & =\ Yield\ i\ (i+1) \end{array} \\
\quad next_{cm}\ (i,\ Nothing)\ = \\
\qquad\qquad \textbf{case}\ next_{enum}\ i\ \textbf{of} \\
\qquad\qquad\quad \begin{array}{lll} Done & \rightarrow & Done \\ Yield\ x\ i' & \rightarrow & \textbf{let} \\ & & \quad next_{ret}\ True\ =\ Yield\ (x * x)\ False \\ & & \quad next_{ret}\ False\ =\ Done \\ & & \textbf{in} \\ & & Skip\ (i',\ Just\ (Stream\ next_{ret}\ True)) \end{array} \\
\quad next_{cm}\ (i,\ Just\ (Stream\ next\ s))\ = \\
\qquad\qquad \textbf{case}\ next\ s\ \textbf{of} \\
\qquad\qquad\quad \begin{array}{lll} Done & \rightarrow & Skip\ (i,\ Nothing) \\ Yield\ y\ s' & \rightarrow & Yield\ y\ (i,\ Just\ (Stream\ next\ s')) \end{array} \\
\quad go\ z\ s\ =\ \textbf{case}\ next_{cm}\ s\ \textbf{of} \\
\qquad\qquad\qquad \begin{array}{lll} Done & \rightarrow & z \\ Skip\quad s' & \rightarrow & go\ z\ s' \\ Yield\ x s' & \rightarrow & go\ (z+x)\ s' \end{array} \\
\textbf{in} \\
go\ 0\ (1,\ Nothing)
\end{array}
$$

Apply case-of-case

$$
\begin{array}{l}
go\ z\ (i,\ Nothing)\ |\ i\ >\ n \quad\ \ = z \\
\qquad\qquad\qquad\quad\ |\ otherwise = \\
\quad \mathbf{let} \\
\qquad next_{ret}\ True\ \ = Yield\ (i * i)\ False \\
\qquad next_{ret}\ False\ = Done \\
\quad \mathbf{in} \\
\quad go\ z\ (i + 1,\ Just\ (Stream\ next_{ret}\ True)) \\
go\ z\ (i,\ Just\ (Stream\ next\ s))\ = \\
\quad \mathbf{case}\ next\ s\ \mathbf{of} \\
\qquad Done \qquad\ \ \rightarrow go\ z \qquad\quad (i,\ Nothing) \\
\qquad Skip\ s' \qquad \rightarrow go\ z \qquad\quad (i,\ Just\ (Stream\ next\ s')) \\
\qquad Yield\ x\ s' \rightarrow go\ (z + x)(i,\ Just\ (Stream\ next\ s'))
\end{array}
$$

Apply constructor specialisation

Using:

$$\begin{aligned}
\forall\, z\ i.\qquad & go\ z\ (i,\ Nothing) &&= go_1\ z\ i \\
\forall\, z\ i\ next\ s.\, & go\ z\ (i,\ Just\ (Stream\ next\ s)) &&= go_2\ z\ i\ next\ s
\end{aligned}$$

We get:

$$\begin{aligned}
& go_1\ z\ i \mid i > n \quad = z \\
& \qquad\quad \mid otherwise = \\
& \quad \mathbf{let} \\
& \qquad next_{ret}\ True\ = Yield\ (i * i)\ False \\
& \qquad next_{ret}\ False = Done \\
& \quad \mathbf{in} \\
& \quad go_2\ z\ (i+1)\ next_{ret}\ True \\
& go_2\ z\ \ i\ next\ s\ = \mathbf{case}\ next\ s\ \mathbf{of} \\
& \qquad\qquad Done \quad\ \ \rightarrow go_1\ z\ i \\
& \qquad\qquad Skip\ s' \quad \rightarrow go_2\ z\ \ i\ next\ s' \\
& \qquad\qquad Yield\ x\ s' \rightarrow go_2\ (z+x)\ \ i\ next\ s'
\end{aligned}$$

APPLY STATIC ARGUMENT TRANSFORMATION

$$
\begin{array}{l}
go_1\ z\ i \mid i\ >\ n \qquad =\ z \\
\qquad\quad \mid otherwise = \\
\quad \textbf{let} \\
\qquad go_2'\ z\ True\ =\ go_2'\ (z + i * i)\ False \\
\qquad go_2'\ z\ False =\ go_1\ z\ (i + 1) \\
\quad \textbf{in} \\
\quad go_2'\ z\ True
\end{array}
$$

Getting there...

AND CLEANUP

$$
\begin{aligned}
go_1\ z\ i \mid i\ >\ n \quad &= \ z \\
\mid otherwise &= \ go_1\ (z + i * i)\ (i + 1)
\end{aligned}
$$

Phew! The original nested loop becomes a fast, flat loop.

Needs those four key optimisations, and in particular, SpecConstr.

AUTOMATED TESTING

STRICTNESS TESTING WITH QUICKCHECK

Needed to test for equivalence to a number of models:

- ➜ Data.Stream == H'98
- ➜ Data.Stream == Data.List
- ➜ Data.Stream.List == H'98
- ➜ Data.Stream.List == Data.List

Perfect use case for QuickCheck!

913 QC properties later, feeling more confident that the code is sane.

BUT WE NEED TO BE CAREFUL ABOUT $\perp$

A port of SmallCheck, to insert $\perp$ into lists.

- ➜ Breadth first search of the test case space
- ➜ Test for correctness in the presence of $\perp$ for all lists up to depth n
- ➜ Inserting and catching $\perp$ in random lists

Caught a lot of strictness differences wrt. to the models, none found by usual QuickCheck!

And bugs (?) in Data.List...

FOLDL' NOT STRICT ENOUGH?

foldl' from Data.List

$$
\begin{aligned}
&foldl' && :: (a \to b \to a) \to a \to [b] \to a \\
&foldl'\ f\ a\ [\,] && = a \\
&foldl'\ f\ a\ (x : xs) && = let\ a' = f\ a\ x\ in\ a'\ `seq`\ foldl'\ f\ a'\ xs
\end{aligned}
$$

And our version:

$$
\begin{aligned}
&foldl'\ f\ z0\ xs0 = go\ z0\ xs0 \\
&\quad \mathbf{where} \\
&\qquad go\ !z\ [\,] \qquad\quad = z \\
&\qquad go\ !z\ (x : xs) = go\ (f\ z\ x)\ xs
\end{aligned}
$$

QuickCheck says they're the same...

But the strictness checker finds:

```
** test 2 of Reducing lists (folds) failed:
**   <function /= _|_>
**   _|_
**   [_|_]
```

That is:

$$Data.List.foldl' \; (\lambda \_ \_ \; \rightarrow \; 0) \; \perp \; [1]$$
$$\Rightarrow \; 0$$

While:

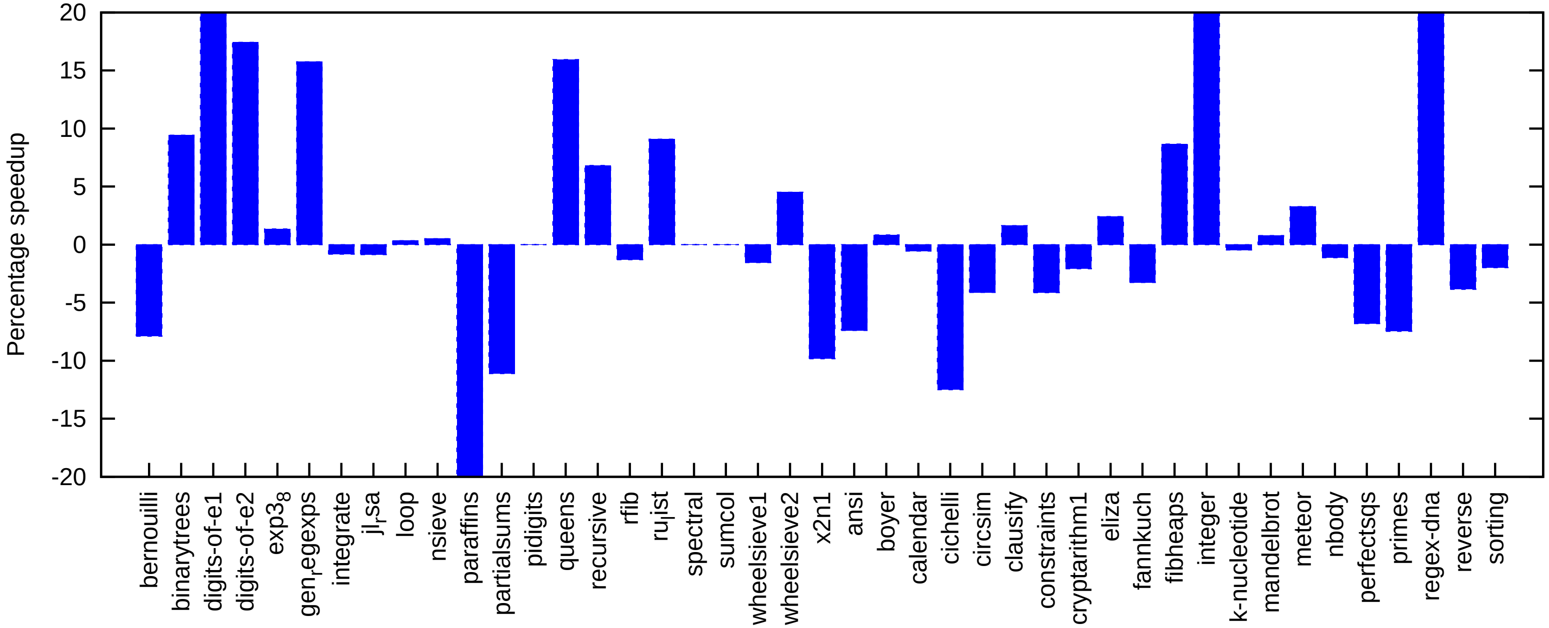
$$Data.List.Stream.foldl' \; (\lambda \_ \_ \; \rightarrow \; 0) \; \perp \; [1]$$
$$\Rightarrow \perp$$

The standard foldl’ is not as strict as it could be!

Strictness properties are hairy, and rarely specified.

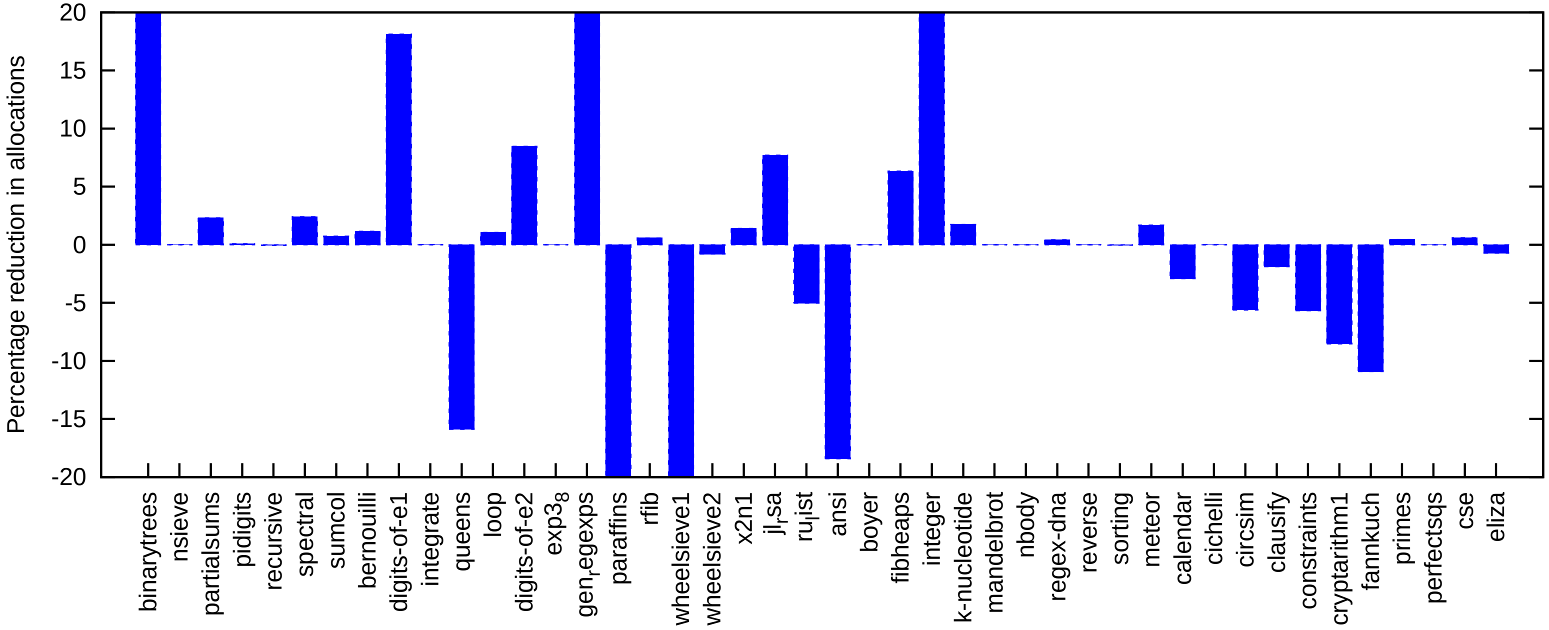
RESULTS

TIME



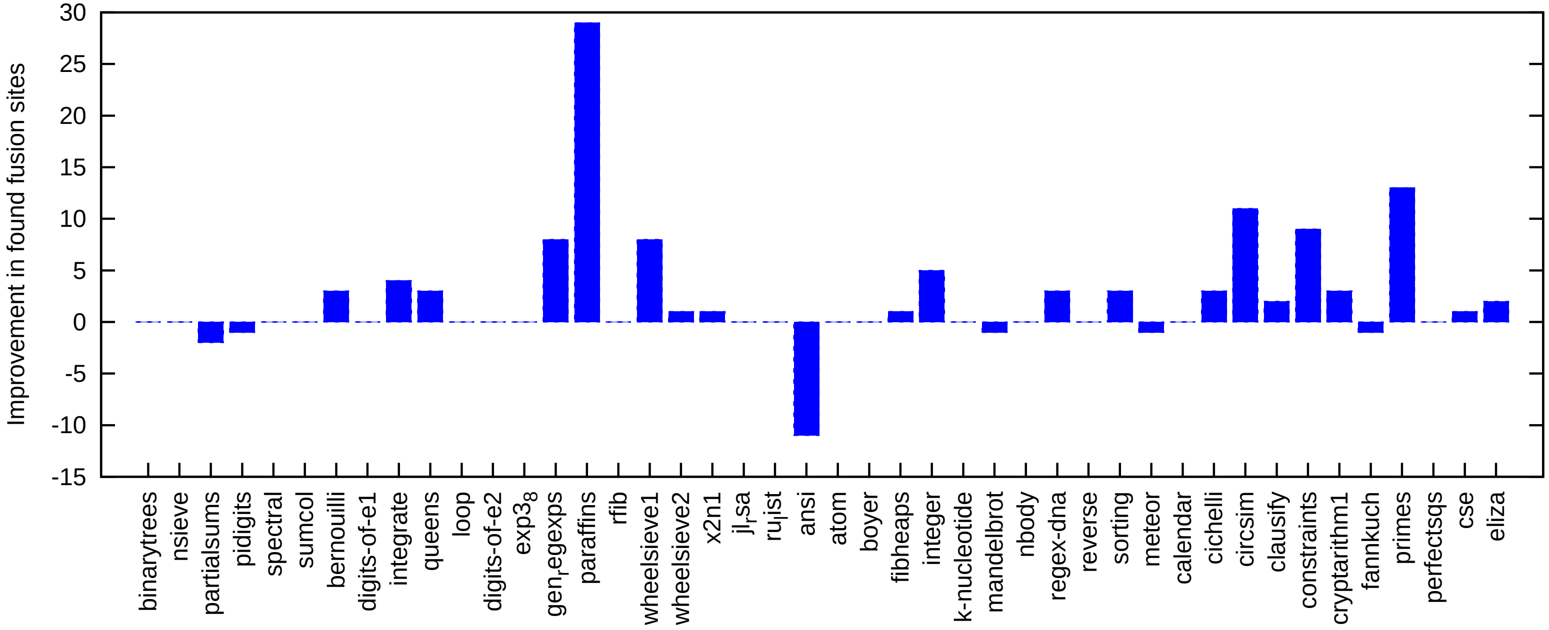
Percentage improvement in running time compared to *build/foldr*

Space



Percent reduction in allocations compared to *build/foldr*

Fusion opportunities



New fusion opportunities found when compared to *build/foldr*

FUTURE WORK

- Improved optimisations: need all $Step$ constructors removed statically (the slow downs indicate which programs aren't tidied up properly)
- Fusing general recursive definitions via a translation to streams
- Fusing other algebraic data types, and back port full system to Data.ByteString

QUESTIONS!

➜ Home page :
`http://www.cse.unsw.edu.au/~dons/streams.html`