# A Journey Through Deforestation

## Guest Lecture, INFR10065 Compiling Techniques course

---

**Lionel Parreaux***, presenting joint work with Anto (Yijia) Chen*
* HKUST (The Hong Kong University of Science and Technology)

The University of Edinburgh, 22 Jan 2024

# Outline

1. What is Deforestation?
2. The Original Deforestation Proposal
3. Supercompilation
4. Shortcut Deforestation
5. Staged Fusion
6. The Long Way to Deforestation

# 1. What is Deforestation?

# Basic Idea of Deforestation

*Functional programming* languages tend to *allocate* lots of short-lived objects.

This is due to a focus on high-level programming
and the use of composable abstractions.

# Basic Idea of Deforestation

*Functional programming* languages tend to *allocate* lots of short-lived objects.

This is due to a focus on high-level programming
and the use of composable abstractions.

**Example:** list combinators, such as `map`, `filter`, `concatMap`, etc.

Even when implemented *lazily* (as in Haskell), these require allocating intermediate values often *used only once* and then immediately discarded.

# Basic Idea of Deforestation

*Functional programming* languages tend to *allocate* lots of short-lived objects.

This is due to a focus on high-level programming
and the use of composable abstractions.

**Example:** list combinators, such as `map`, `filter`, `concatMap`, etc.

Even when implemented *lazily* (as in Haskell), these require allocating intermediate values often *used only once* and then immediately discarded.

**Deforestation:** the act of removing the unnecessary creation of trees from functional programs without changing their semantics.

## 1.1. Manual Deforestation

# Manual Deforestation Example

Consider the following Haskell program:

```
map f xs = case xs of { []     → [];
                        x : xs → f x : map f xs }
incr x = x + 1
double x = x * 2

main ls = map incr (map double ls)
```

Problem?

# Manual Deforestation Example

Consider the following Haskell program:

```
map f xs = case xs of { []     → [];
                        x : xs → f x : map f xs }
incr x = x + 1
double x = x * 2

main ls = map incr (map double ls)
```

Problem?

The intermediate list `map double ls` is immediately consumed by `map incr`!

## Manual Deforestation Example

```
map f xs = case xs of { []     → [];
                        x : xs → f x : map f xs }
main ls = map incr (map double ls)
```

The intermediate list `map double ls` is immediately consumed by `map incr`!

# Manual Deforestation Example

```
map f xs = case xs of { []       → [];
                        x : xs → f x : map f xs }
main ls = map incr (map double ls)
```

The intermediate list `map double ls` is immediately consumed by `map incr`!

The following code is typically 40% more efficient:

```
map2 f g xs = case xs of { []       → [];
                           x : xs → f (g x) : map2 f g xs }
main ls = map2 incr double ls
```

# Manual Deforestation Example

```
map f xs = case xs of { []       → [];
                        x : xs → f x : map f xs }
main ls = map incr (map double ls)
```

The intermediate list `map double ls` is immediately consumed by `map incr`!

The following code is typically 40% more efficient:

```
map2 f g xs = case xs of { []       → [];
                           x : xs → f (g x) : map2 f g xs }
main ls = map2 incr double ls
```

No more intermediate list created!

# Manual Deforestation

It is possible to rewrite your programs
to avoid the creation of intermediate data structures manually.

# Manual Deforestation

It is possible to rewrite your programs
to avoid the creation of intermediate data structures manually.

This requires heavy refactoring, duplication, and breaks modular abstractions,
as it requires exposing and rewriting implementations.

# Manual Deforestation

It is possible to rewrite your programs
to avoid the creation of intermediate data structures manually.

This requires heavy refactoring, duplication, and breaks modular abstractions,
as it requires exposing and rewriting implementations.

$\Longrightarrow$ **BAD!**

## 1.2. The Pie in The Sky: Automatic Deforestation

# Automatic Deforestation

This lecture is a *high-level introduction* on various *approaches to deforestation* that have been proposed over the years.

# 2. The Original Deforestation Proposal

# The Original Deforestation Idea

Not the first, but the one that coined the name, and one of the simplest

Proposed by Philip Wadler in 1990, then at University of Glasgow (Wadler 1990).

# The Original Deforestation Idea

Not the first, but the one that coined the name, and one of the simplest

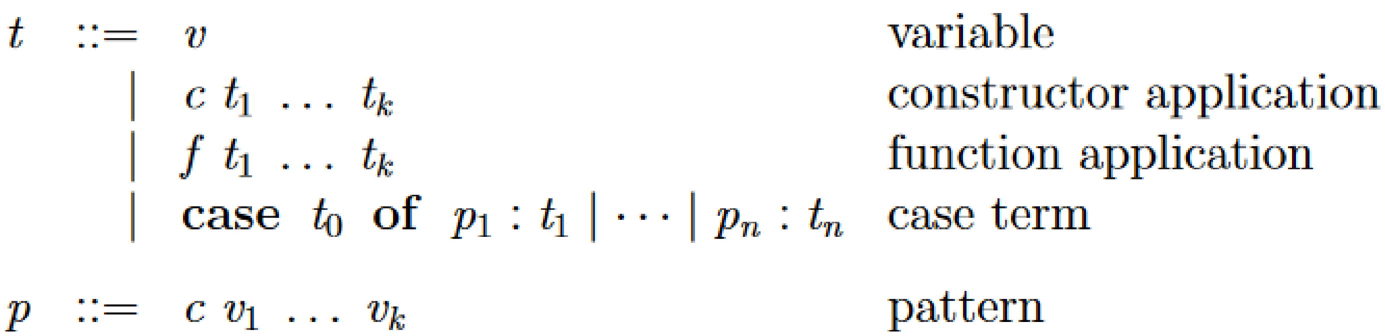
Proposed by Philip Wadler in 1990, then at University of Glasgow (Wadler 1990).

High-level ideas:

- restrict the input language to simplify the problem
- unroll recursive definitions and tie the knot to avoid infinite loops

# The Original Deforestation Idea

**The Language**: $t$ stands for term; $p$ stands for pattern



$$\begin{array}{lcll}
t & ::= & v & \text{variable} \\
 & | & c\ t_1\ \ldots\ t_k & \text{constructor application} \\
 & | & f\ t_1\ \ldots\ t_k & \text{function application} \\
 & | & \textbf{case}\ t_0\ \textbf{of}\ p_1 : t_1 \mid \cdots \mid p_n : t_n & \text{case term} \\
 & & & \\
p & ::= & c\ v_1\ \ldots\ v_k & \text{pattern}
\end{array}$$

- $c$ the name of the constructor, can be arbitrary
- patterns $p$ are not nested for simplicity
- $\overline{a_i}$ denotes $a_1 \ldots a_n$
- $t\{v \to t'\}$ denotes replacing all occurences of variable $v$ inside $t$ with $t'$

# The Original Deforestation Idea

**Key Idea**: simulating the evaluation of the program to bring together the production of data structures to their corresponding consumption sites (**case** terms), then the elimination of intermediate data structures is trivial.

---

```
case (Cons 1 Nil) of Nil: branch1 | Cons h t: branch2
```

can be easily transformed into `branch2`
with `h` replaced by `1` and `t` replaced by `Nil`

## The Original Deforestation Idea

The core transformation algorithm $T$ simulates the evaluation of programs

1. $T[\![v]\!] = v$
2. $T[\![c\ \overline{t_i}]\!] = c\ \overline{T[\![t_i]\!]}$
3. $T[\![f\ \overline{t_i}]\!] = T\left[\!\!\left[t\ \overline{\{v_i \to t_i\}}\right]\!\!\right]$, where $f$ is defined as $f\ \overline{v_i} = t$
4. $T[\![\textbf{case}\ v\ \textbf{of}\ \overline{p_i : t_i}]\!] = \textbf{case}\ v\ \textbf{of}\ \overline{p_i : T[\![t_i]\!]}$
5. $T[\![\textbf{case}\ c_n\ \overline{t_j}\ \textbf{of}\ \overline{p_i : t_i}]\!] = T\left[\!\!\left[t_n \overline{\{v_j \to t_j\}}\right]\!\!\right]$ if $p_n \equiv c_n\ \overline{v_j}$
6. $T[\![\textbf{case}\ f\ \overline{t_j}\ \textbf{of}\ \overline{p_i : t_i}]\!] = T\left[\!\!\left[\textbf{case}\ \left(t\ \overline{\{v_j \to t_j\}}\right)\ \textbf{of}\ \overline{p_i : t_i}\right]\!\!\right]$, where $f$ is defined as $f\ \overline{v_j} = t$

# The Original Deforestation Idea

7. $T[\![\textbf{case}\ (\textbf{case}\ t_0\ \textbf{of}\ \overline{p_i : t_i})\ \textbf{of}\ \overline{p'_j : t'_j}]\!] =$

$$T\left[\!\!\left[\textbf{case}\ t_0\ \textbf{of}\ \overline{p_i : \textbf{case}\ t_i\ \textbf{of}\ \overline{p'_j : t'_j}}\right]\!\!\right]$$

# The Original Deforestation Idea

7. $T[\![\textbf{case}\ (\textbf{case}\ t_0\ \textbf{of}\ \overline{p_i : t_i})\ \textbf{of}\ \overline{p'_j : t'_j}]\!] =$

$$T\left[\!\!\left[\textbf{case}\ t_0\ \textbf{of}\ \overline{p_i : \textbf{case}\ t_i\ \textbf{of}\ \overline{p'_j : t'_j}}\right]\!\!\right]$$

Example:

$$\textbf{case}\ (\textbf{case}\ v\ \textbf{of}\ \mathrm{None} : \mathrm{Just}\ 1 \mid \mathrm{Just}\ a : \mathrm{None})\ \textbf{of}\ \mathrm{None} : 0 \mid \mathrm{Just}\ a : a$$

is transformed in one step to:

$$\begin{aligned} \textbf{case}\ v\ \textbf{of}\quad & \mathrm{None} : \textbf{case}\ \mathrm{Just}\ 1\ \textbf{of}\ \mathrm{None} : 0 \mid \mathrm{Just}\ a : a \\ & \mathrm{Just}\ a : \textbf{case}\ \mathrm{None}\ \textbf{of}\ \mathrm{None} : 0 \mid \mathrm{Just}\ a : a \end{aligned}$$

# The Original Deforestation Idea

7. $T[\![\textsf{\textbf{case}}\ (\textsf{\textbf{case}}\ t_0\ \textsf{\textbf{of}}\ \overline{p_i : t_i})\ \textsf{\textbf{of}}\ \overline{p'_j : t'_j}]\!] =$

$$T\left[\!\!\left[\textsf{\textbf{case}}\ t_0\ \textsf{\textbf{of}}\ \overline{p_i : \textsf{\textbf{case}}\ t_i\ \textsf{\textbf{of}}\ \overline{p'_j : t'_j}}\right]\!\!\right]$$

Example:

$$\textsf{\textbf{case}}\ (\textsf{\textbf{case}}\ v\ \textsf{\textbf{of}}\ \mathrm{None} : \mathrm{Just}\ 1 \mid \mathrm{Just}\ a : \mathrm{None})\ \textsf{\textbf{of}}\ \mathrm{None} : 0 \mid \mathrm{Just}\ a : a$$

is transformed in one step to:

$$\begin{array}{ll} \textsf{\textbf{case}}\ v\ \textsf{\textbf{of}} & \mathrm{None} : \textsf{\textbf{case}}\ \mathrm{Just}\ 1\ \textsf{\textbf{of}}\ \mathrm{None} : 0 \mid \mathrm{Just}\ a : a \\ & \mathrm{Just}\ a : \textsf{\textbf{case}}\ \mathrm{None}\ \textsf{\textbf{of}}\ \mathrm{None} : 0 \mid \mathrm{Just}\ a : a \end{array}$$

into $\quad \textsf{\textbf{case}}\ v\ \textsf{\textbf{of}}\quad \mathrm{None} : 1 \quad\mid\quad \mathrm{Just}\ a : 0$

# The Original Deforestation Idea

The transformation algorithm is designed to proceed as much as possible,
in spite of missing the actual run-time information,
to bring together data constructor applications and **case** terms.

- 3 and 6: unfold function definitions
- 5: eliminate intermediate data structure
- 7: case-of-case commuting

# Example

An more meaningful example:

```
flip (flip t)
where flip tr = case tr of
                  Leaf z: Leaf z
                  Branch l r: Branch (flip r) (flip l)
```

# Example

- `flip (flip t)`
- `case (flip t) of …` by 3
- `case (case t of … ) …` by 6
- 
```
case t of
  Branch l r: case (Branch (flip r) (flip l)) of Branch l r: …
  Leaf z: …
```
by 7
- 
```
case t of
  Branch l r: Branch (flip (flip l)) (flip (flip r))
  Leaf z: Leaf z
```
by 4, 5, 5

We encounter `flip (flip r)` and `flip (flip l)` again!

$T$ Loops forever on unfolding `flip`?

# Tying the Recursive Knot

Keep track of function call terms we have already processed, and later when **similar** terms are encountered again, stop unfolding and tie the knot by introducing new recursive function definitions.

- **Similar** terms: up to renaming of variables

  Examples:

| | | | |
|---|---|---|---|
| $f\ x$ | $g\ x$ | $f\ (g\ x)$ | $f\ x$ |
| $f\ x$ | $g\ y$ | $f\ (g\ y)$ | $f\ (g\ x)$ |

# Tying the Recursive Knot

`flip (flip r)` and `flip (flip l)` are both renamings of the initial term `flip (flip t)`, so the unfolding stops by introducing a new definition `h`, whose body is the current term we get from running $T$ with `flip (flip r)` and `flip (flip l)` replaced by `h r` and `h l`:

```
h t = case t of
          Branch l r: Branch (h l) (h r)
          Leaf z: Leaf z
```

# Treeless Form

Our goal is to eliminate the allocation of intermediate data structures.
There is a syntactic property of programs that approximately describes our goal: **Treeless Form**.

- Every argument of a function call or selector of a **case** term must be a variable — no possible intermediate data structure allocation
- Every variable must be used only once — no possible duplication of work after unfolding

# Treeless Form

```
app xs ys =
 case xs of
  Nil: ys
  Cons h t:
    Cons h (app t ys)
```

treeless: `ys` appears in two different branches

```
double xs =
 app xs xs
```

not treeless: `xs` used twice

```
appapp xs ys zs =
  app xs (app ys zs)
```

not treeless: `app ys zs` passed as an argument; but can be transformed to a treeless definition

# Termination

Treeless form ensures that the algorithm $T$ can always terminate with a program no less efficient than the original one.

# Termination

Treeless form ensures that the algorithm $T$ can always terminate with a program no less efficient than the original one.

---

***Deforestation Theorem***. Every composition of functions with treeless definitions can be effectively transformed to a single function with a treeless definition, without loss of efficiency.

Pretty strong result!

## Limitations

Though the original deforestation algorithm is simple and elegant,
its applicability is limited,
explaining why it has not been used by practical compilers.

# Limitations

Though the original deforestation algorithm is simple and elegant,
its applicability is limited,
explaining why it has not been used by practical compilers.

- The treeless form is *very* restrictive
    - first-order language (!)
    - linear uses of variables
    - no internal data structures

  (later approaches lifted *some* restrictions but the reasoning was still limited)

# Limitations

Though the original deforestation algorithm is simple and elegant,
its applicability is limited,
explaining why it has not been used by practical compilers.

- The treeless form is *very* restrictive
  - first-order language (!)
  - linear uses of variables
  - no internal data structures

  (later approaches lifted *some* restrictions but the reasoning was still limited)
- Tying the knot on the fly is expensive

# 3. Supercompilation

# Supercompilation

A *powerful* program transformation technique originally due to Turchin (1986), which shares many similarities with Deforestation that it statically simulates the evaluation of a program to expose its internal logic and find optimization opportunities.

Uses:

- Prove theorems about programs
- Specialize function definitions
- **Deforestation**

# Supercompilation

What does a supercompiler do?

- **Driving**: Simulate the evaluation of programs, but with free variables
- **Folding**: Introduce new recursive function definitions

# Supercompilation

- **Driving** propagates more information about free variables instead of simply ignoring them like the original deforestation algorithm
-

# Supercompilation

- **Driving** propagates more information about free variables instead of simply ignoring them like the original deforestation algorithm
- **Folding** together with **Generalization** ensure the termination of Supercompilation on *general programs* (not limited to treeless form)

# 3.1. Positive Supercompilation

# Positive Supercompilation: Driving Rules

Positive supercompilation is a simplified form of full supercompilation: it only propagates **positive** information (to be explained in the next slide).

The driving rules are similar to the algorithm $T[\![t]\!]$ presented in the original deforestation, with one core difference:

$$\mathcal{D}[\![\textbf{case } v \textbf{ of } \overline{p_i \to t_i}]\!] = \textbf{case } v \textbf{ of } \overline{p_i : \mathcal{D}[\![t_i\{v \to p_i\}]\!]}$$

# Positive Supercompilation: Driving Rules

$$\mathcal{D}[\![\textbf{case}\ v\ \textbf{of}\ \overline{p_i \to t_i}]\!] = \textbf{case}\ v\ \textbf{of}\ \overline{p_i : \mathcal{D}[\![t_i\{v \to p_i\}]\!]}$$

- The original deforestation simply does $\textbf{case}\ v\ \textbf{of}\ \overline{p_i : \mathcal{D}[\![t_1]\!]}$, but a positive supercompiler will propagate the information of the exact shape of $v$ in each branch.
- "**Positive**" means the supercompiler will only propagate equality information (i.e. $v \equiv p_i$), but not inequality information.
- A more powerful supercompiler may propagate both kinds of information at a higher cost, and using different approaches other than substitution.

# Positive Supercompilation: Driving Rules

```
firstPlusLast ls
where
  firstPlusLast xs = case xs of
    Nil: Nothing
    Cons h t: Some (h + fromJust (last xs))
  last xs = case xs of Nil: Nothing | Cons h t: Some (last' h t)
  last' a xs = case xs of Nil: a | Cons h t: last' h t
  fromJust m = case m of Just a: a
```

Having the positive information propagated helps to reduce the allocation of `Some (last' h t)` and the call to `fromJust`, by changing `last xs` to `last (Cons h t)`. More examples in paper (Sørensen, Glück, Jones 1996).

# Positive Supercompilation: Folding Strategies

For non-treeless programs, the driving processes of the following programs never terminates without a more powerful folding strategy,

```
nrev xs
where
  nrev [] = []
  nrev (h:t) = app (nrev t) (h:[])
  app [] ys = ys
  app (x:xs) ys = x:(app xs ys)
```

```
arev xs []
where
  arev [] a     = []
  arev (x:xs) a = arev xs (x:a)
```

---

```
nrev xs
case (nrev xs) of ...
case (case (nrev xs) of ...) of
 ...
```

```
arev xs []
arev xs (x:[])
arev xs (x:x':[])
 ...
```

# Positive Supercompilation: Folding Strategies

The following two techniques are used so that recursive knots can be tied during the folding process to ensure termination.

- **Homeomorphic embedding**: detect similar terms
- **Generalization**: handle similar terms and ensure termination

# Positive Supercompilation: Folding Strategies

## Homeomorphic embedding

- $t_1 \lhd t_2$ if $t_1 \lhd_d t_2$ (diving) or $t_1 \lhd_c t_2$ (coupling)
- *Diving*: $t_1 \lhd_d t_2$ if there is a subterm $t_{2_i}$ of $t_2$ such that $t_1 \lhd t_{2_i}$
- *Coupling*: $t_1 \lhd_c t_2$ if $t_1$ and $t_2$ share the same top-level term constructor and all their corresponding subterms $t_{1_i}$ and $t_{2_i}$ satisfy $t_{1_i} \lhd t_{2_i}$
- the homeomorphic embedding relation $t_1 \leq t_2$, if there is a renaming $t_r$ of $t_1$ such that $t_r \lhd_c t_2$

# Positive Supercompilation: Folding Strategies

## Homeomorphic embedding

- Some examples

$\lambda x.x \leq \lambda y.y$ $\quad$ $f\ (g\ x) \leq f\ (g\ y)$ $\quad$ $f\ (h\ x) \leq f\ (g\ (h\ y))$

$\lambda x.x \nleq \lambda y.x$ $\quad$ $f\ z\ z \nleq f\ x\ y$ $\quad$ $f\ (g\ x) \nleq g\ (f\ y)$

# Positive Supercompilation: Folding Strategies

**Generalization** of two similar terms $t_1, t_2$:

A triple $(t, \theta_1, \theta_2)$, where $t$ is a term, and $\theta_1, \theta_2$ are substitutions from variables to terms, such that $t\theta_1 = t_1$ and $t\theta_2 = t_2$

# Positive Supercompilation: Folding Strategies

**Generalization** of two similar terms $t_1, t_2$:

A triple $(t, \theta_1, \theta_2)$, where $t$ is a term, and $\theta_1, \theta_2$ are substitutions from variables to terms, such that $t\theta_1 = t_1$ and $t\theta_2 = t_2$

**Some examples**

- for $f\ x\ y$ and $f\ z\ z$ we have $(f\ v_1\ v_2, \{v_1 \to x, v_2 \to y\}, \{v_1 \to z, v_2 \to z\})$
- for $f\ (g\ x)$ and $f\ (g\ y)$ we have $(f\ (g\ v), \{v \to x\}, \{v \to y\})$
- for $f\ x\ x$ and $f\ (g\ y)\ (h\ y)$ we have $(f\ v_1\ v_2, \{v_1 \to x, v_2 \to x\}, \{v_1 \to g\ y, v_2 \to h\ y\})$

# Positive Supercompilation: Folding Strategies

These **folding and generalization strategies** ensure termination:
the homeomorphic embedding generalizes the idea of similarity between terms up to renaming, such that all non-terminating possiblities can be detected during the driving process.

# Positive Supercompilation: Folding Strategies

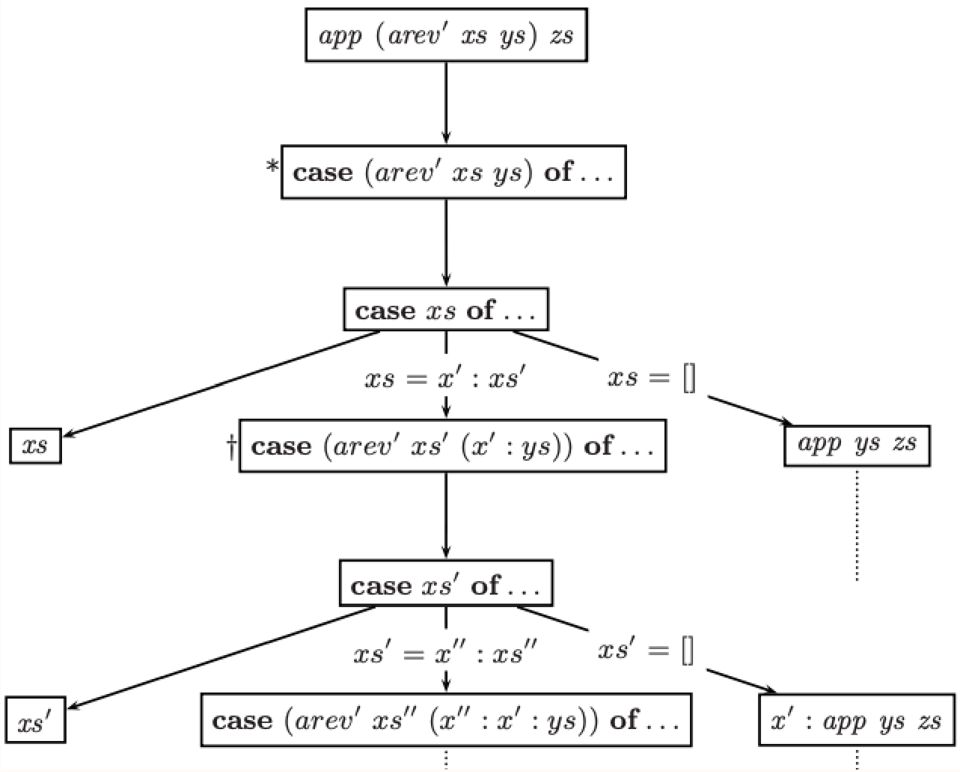
These **folding and generalization strategies** ensure termination:
the homeomorphic embedding generalizes the idea of similarity between terms up to renaming, such that all non-terminating possiblities can be detected during the driving process.

**Downside:** very *complicated* to implement and *expensive* to execute;
to the best of our knowledge, no practical compiler actually does this

# 3.2. Distillation

# Distillation

## **Process Trees**: the trace of the driving process



# Distillation

The homeomorphic embedding and generalization processes are then extended to *process trees* (intuitively, “unrollings” themselves),
giving a more powerful and expensive transformation algorithm.

# Distillation

The homeomorphic embedding and generalization processes are then extended to *process trees* (intuitively, “unrollings” themselves),
giving a more powerful and expensive transformation algorithm.

**Downside:** ***even more*** *complicated* and *expensive*!

# Distillation

The homeomorphic embedding and generalization processes are then extended to *process trees* (intuitively, "unrollings" themselves),
giving a more powerful and expensive transformation algorithm.

**Downside:** ***even more*** *complicated* and *expensive*!

We are not aware of any implementation that's not patently broken
even on basic examples.

# 4. Shortcut Deforestation

# Shortcut Deforestation

“Shortcut Deforestation”, also known as “Shortcut Fusion” or just “Fusion” was introduced by Gill, Launchbury, Peyton Jones (1993) as a practical way of achieving deforestation without the complexities of supercompilation.

# Shortcut Deforestation

“Shortcut Deforestation”, also known as “Shortcut Fusion” or just “Fusion” was introduced by Gill, Launchbury, Peyton Jones (1993) as a practical way of achieving deforestation without the complexities of supercompilation.

**Key ideas:**

- leave recursive definitions alone
- only focus on rewriting the use of *combinators*

# Shortcut Deforestation

“Shortcut Deforestation”, also known as “Shortcut Fusion” or just “Fusion” was introduced by Gill, Launchbury, Peyton Jones (1993) as a practical way of achieving deforestation without the complexities of supercompilation.

**Key ideas:**

- leave recursive definitions alone
- only focus on rewriting the use of *combinators*

**Example:** rewrite `map f (map g xs)` to `map (f . g) xs`

**Problem:** huge set of rules to account for all possible pairs of functions…?

## 4.1. List Fusion

# Essence of Functional Lists

Functional lists can be boiled down to two fundamental operations:

- ***building*** a new list based on some `cons` and `nil` constructors
- ***folding*** a list by replacing all these `cons` and `nil` operations by function calls

# Essence of Functional Lists

Functional lists can be boiled down to two fundamental operations:

- ***building*** a new list based on some `cons` and `nil` constructors
- ***folding*** a list by replacing all these `cons` and `nil` operations by function calls

Example of `build`ing a list:

```
-- syntax sugar for List.cons(1, List.cons(2, List.cons(3, List.nil)))
1 : 2 : 3 : []
```

# Essence of Functional Lists

Functional lists can be boiled down to two fundamental operations:

- ***building*** a new list based on some `cons` and `nil` constructors
- ***folding*** a list by replacing all these `cons` and `nil` operations by function calls

Example of `build`ing a list:

```
-- syntax sugar for List.cons(1, List.cons(2, List.cons(3, List.nil)))
1 : 2 : 3 : []
  == build (\ c n -> c 1 (c 2 (c 3 n)))
```

Definition of `build`:

```
build g = g (:) []
```

# Essence of Functional Lists

Definition of `fold`:

```
foldr k z [] = z
foldr k z (x : xs) = k x (foldr k z xs)
```

Rephrasing classical list functions in terms of `foldr`:

```
sum xs       = foldr (+) 0 xs
elem x xs    = foldr (\ a b -> a == x || b) False xs
map f xs     = foldr (\ a b -> f a : b) [] xs
filter f xs  = foldr (\ a b -> if f a then a : b else b) [] xs
xs ++ ys     = foldr (:) ys xs
concat xs    = foldr (++) [] xs
foldl f z xs = foldr (\ b g a -> g (f a b)) id xs z
```

# Digression: Typing `build` and `foldr`

What type should `foldr` have?

# Digression: Typing `build` and `foldr`

What type should `foldr` have?

```
foldr :: (b → a → a) → a → [b] → a
```

How about `build`?

```
build g = g (:) []
```

# Digression: Typing `build` and `foldr`

What type should `foldr` have?

```
foldr :: (b → a → a) → a → [b] → a
```

How about `build`?

```
build g = g (:) []
```

Tentative:

```
build :: ((a → [a] → [a]) → [a] → [a]) → [a]
```

# Digression: Typing `build` and `foldr`

What type should `foldr` have?

```
foldr :: (b → a → a) → a → [b] → a
```

How about `build`?

```
build g = g (:) []
```

Tentative:

```
build :: ((a → [a] → [a]) → [a] → [a]) → [a]
```

Too specific… More general type?

# Digression: Typing `build` and `foldr`

What type should `foldr` have?

```
foldr :: (b → a → a) → a → [b] → a
```

How about `build`?

```
build g = g (:) []
```

Tentative:

```
build :: ((a → [a] → [a]) → [a] → [a]) → [a]
```

Too specific… More general type?

```
build :: (forall a. (b → a → a) → a → a) → [b]
```

# Crucial Equation of `build` and `foldr`

The following equation is crucial to list fusion:

```
foldr c n (build g)  =  g c n
```

# Crucial Equation of `build` and `foldr`

The following equation is crucial to list fusion:

```
foldr c n (build g)  =  g c n
```

Notice that `g`, which was originally useed to build a list in `build`,
is now used to compute a result which may be something else, such as an `Int`!

> This works thanks to the higher-ranked polymorphic type of `build`,
> meaning that `g` is itself required to be polymorphic

# Back to the Motivating Example

Recall:

```
main ls = map incr (map double ls)
```

# Back to the Motivating Example

Recall:

```
main ls = map incr (map double ls)
```

Desugared into combinators:

```
main ls = build (\ c1 n1 →
  foldr (\ a1 b1 → c1 (incr a1) b1) n1 (map double ls))
```

Desugared further:

```
main ls = build (\ c1 n1 →
  foldr (\ a1 b1 → c1 (incr a1) b1) n1 (build (\ c2 n2 →
    foldr (\ a2 b2 → c2 (double a2) b2) n2 ls)))
```

# List Fusion in Practice

The *Glasgow Haskell Compiler* (**GHC**) allows registering
user-defined *rewrite rules*, which can be used to implement automatic list fusion
(Peyton Jones, Tolmach, Hoare 2001)

# List Fusion in Practice

The *Glasgow Haskell Compiler* (**GHC**) allows registering
user-defined *rewrite rules*, which can be used to implement automatic list fusion (Peyton Jones, Tolmach, Hoare 2001)

Note: several fundamental and practical limitations to this approach (see later)

## 4.2. Other Shortcut Fusion Approaches

# Other Shortcut Fusion Approaches

Many related approaches following the same technique were proposed.

They have different tradeoffs: some programs fuse better than others

# Other Shortcut Fusion Approaches

Many related approaches following the same technique were proposed.

They have different tradeoffs: some programs fuse better than others

For instance, *Stream Fusion* (Coutts, Leshchinskiy, Stewart 2007) supports fusing `zip`, left folds, and nested lists

Streams are like lists but have an additional `Skip` constructor

## 4.3. Limitations of Shortcut Fusion

# Limitations of Staged Fusion

Fundamental limitations:

- Cannot rewrite user-defined functions

  The entire program must be rewritten in terms of combinators
  - not always practical
  - can have performance implications (may make things slower)
- There isn't always a best set of combinators to use

# Limitations of Staged Fusion

Fundamental limitations:

- Cannot rewrite user-defined functions

  The entire program must be rewritten in terms of combinators
  - not always practical
  - can have performance implications (may make things slower)
- There isn't always a best set of combinators to use

Practical limitations:

- Quite unreliable; extremely dependent on heuristic inlining
- User-defined rewrite rules not checked for correctness

# 5. Staged Fusion

# High-level Idea

**Staged Fusion** uses *multi-stage programming*, a metaprogramming technique, to *guarantee* that all constructed programs are *completely fused*.

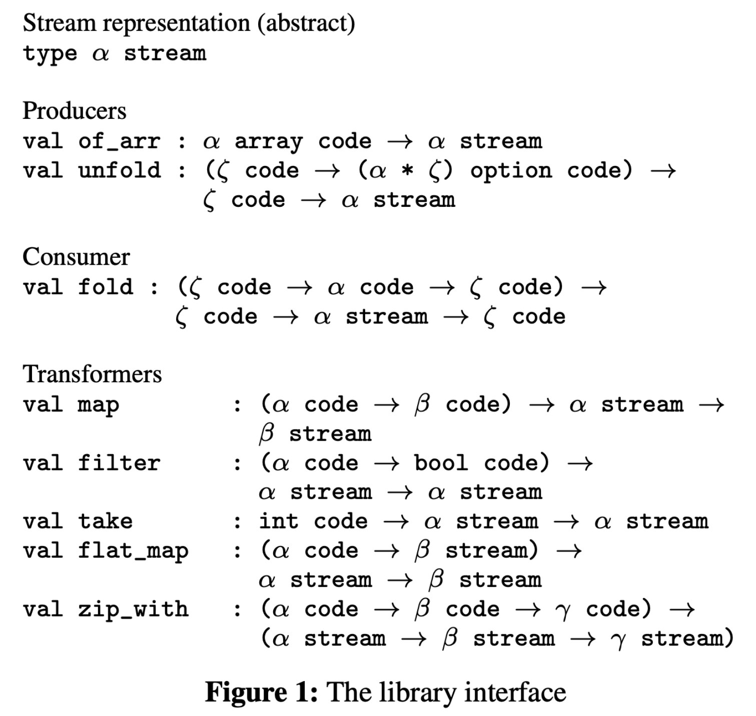
# High-level Idea

**Staged Fusion** uses *multi-stage programming*, a metaprogramming technique, to *guarantee* that all constructed programs are *completely fused*.

Users typically have to rewrite their programs
(the library's interface becomes *staged*)

# Complete Stream Fusion

By Kiselyov, Biboudis, Palladinos, Smaragdakis (2017).



Stream representation (abstract)

```
type α stream
```

Producers

```
val of_arr : α array code → α stream
val unfold : (ζ code → (α * ζ) option code) →
             ζ code → α stream
```

Consumer

```
val fold : (ζ code → α code → ζ code) →
           ζ code → α stream → ζ code
```

Transformers

```
val map        : (α code → β code) → α stream →
                 β stream
val filter     : (α code → bool code) →
                 α stream → α stream
val take       : int code → α stream → α stream
val flat_map   : (α code → β stream) →
                 α stream → β stream
val zip_with   : (α code → β code → γ code) →
                 (α stream → β stream → γ stream)
```

**Figure 1:** The library interface

# Limitations of Staged Fusion

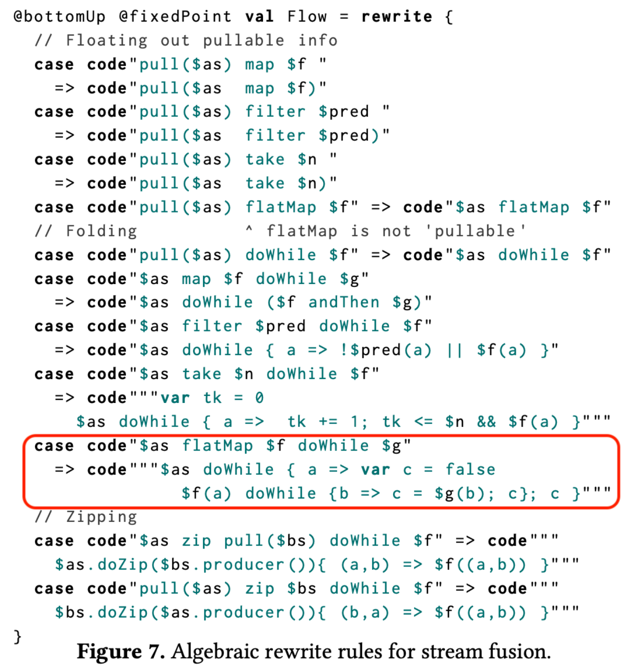
Only *partially* automated: experts need to define the staged libraries

*Intrusive*: users need to rewrite their programs

*Inflexible*: staging boundaries are fixed and can’t easily be changed

# Hybrid Approaches

## Example: “Quoted *staged rewriting*” by Parreaux, Shaikhha, Koch (2017)

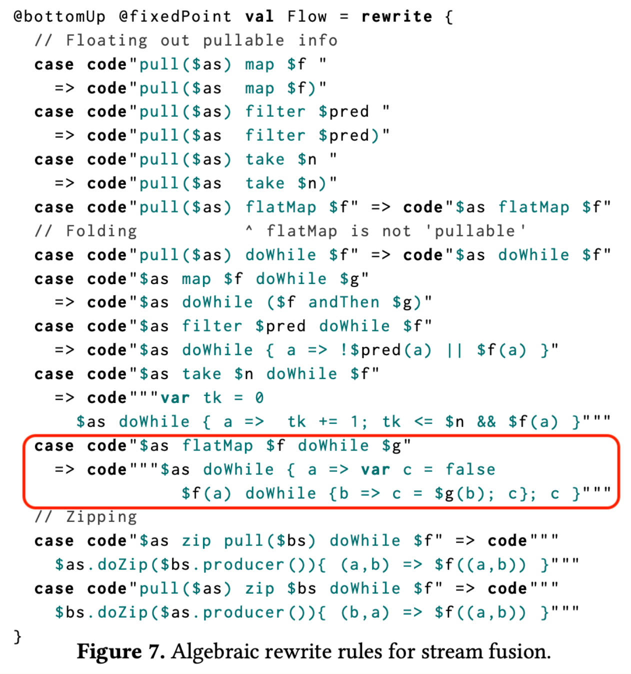


```
@bottomUp @fixedPoint val Flow = rewrite {
  // Floating out pullable info
  case code"pull($as) map $f "
    => code"pull($as  map $f)"
  case code"pull($as) filter $pred "
    => code"pull($as  filter $pred)"
  case code"pull($as) take $n "
    => code"pull($as  take $n)"
  case code"pull($as) flatMap $f" => code"$as flatMap $f"
  // Folding          ^ flatMap is not 'pullable'
  case code"pull($as) doWhile $f" => code"$as doWhile $f"
  case code"$as map $f doWhile $g"
    => code"$as doWhile ($f andThen $g)"
  case code"$as filter $pred doWhile $f"
    => code"$as doWhile { a => !$pred(a) || $f(a) }"
  case code"$as take $n doWhile $f"
    => code"""var tk = 0
      $as doWhile { a =>  tk += 1; tk <= $n && $f(a) }"""
  case code"$as flatMap $f doWhile $g"
    => code"""$as doWhile { a => var c = false
                $f(a) doWhile {b => c = $g(b); c}; c }"""
  // Zipping
  case code"$as zip pull($bs) doWhile $f" => code"""
    $as.doZip($bs.producer()){ (a,b) => $f((a,b)) }"""
  case code"pull($as) zip $bs doWhile $f" => code"""
    $bs.doZip($as.producer()){ (b,a) => $f((a,b)) }"""
}
```

**Figure 7.** Algebraic rewrite rules for stream fusion.

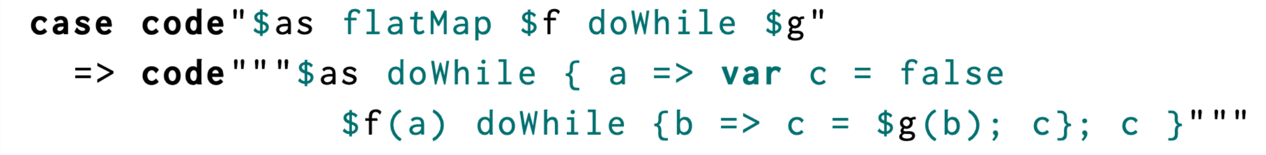
# Hybrid Approaches

Example: “Quoted *staged rewriting*” by Parreaux, Shaikhha, Koch (2017)



```
@bottomUp @fixedPoint val Flow = rewrite {
  // Floating out pullable info
  case code"pull($as) map $f "
    => code"pull($as  map $f)"
  case code"pull($as) filter $pred "
    => code"pull($as  filter $pred)"
  case code"pull($as) take $n "
    => code"pull($as  take $n)"
  case code"pull($as) flatMap $f" => code"$as flatMap $f"
  // Folding             ^ flatMap is not 'pullable'
  case code"pull($as) doWhile $f" => code"$as doWhile $f"
  case code"$as map $f doWhile $g"
    => code"$as doWhile ($f andThen $g)"
  case code"$as filter $pred doWhile $f"
    => code"$as doWhile { a => !$pred(a) || $f(a) }"
  case code"$as take $n doWhile $f"
    => code"""var tk = 0
      $as doWhile { a =>  tk += 1; tk <= $n && $f(a) }"""
  case code"$as flatMap $f doWhile $g"
    => code"""$as doWhile { a => var c = false
                $f(a) doWhile {b => c = $g(b); c}; c }"""
  // Zipping
  case code"$as zip pull($bs) doWhile $f" => code"""
    $as.doZip($bs.producer()){ (a,b) => $f((a,b)) }"""
  case code"pull($as) zip $bs doWhile $f" => code"""
    $bs.doZip($as.producer()){ (b,a) => $f((a,b)) }"""
}
```

**Figure 7.** Algebraic rewrite rules for stream fusion.



```
case code"$as flatMap $f doWhile $g"
  => code"""$as doWhile { a => var c = false
              $f(a) doWhile {b => c = $g(b); c}; c }"""
```

# 6. The Long Way to Deforestation

## 6.1. Type Inference

# Type Inference

- **Type Inference**: assign a type to each term $(t : \tau)$ in the program such that the types describe the behavior of the value represented by terms (how the value is produced / consumed)
- **Type Check**: make sure that values are *consumed* as intended when they are *produced* so that we will not end up in weird errors, such as using lists as booleans $(\mathrm{List} \neq \mathrm{Bool})$

## 6.2. Subtype Inference

# Subtype Inference

In languages like Haskell, **type inference** propagates *equalities* between types: $\tau_1 = \tau_2$, which discards the direction of the flow of data.

Subtype information is more flexible because it can encode data flow information of programs: $\tau_1 <: \tau_2$ ("$\tau_1$ is a subtype of $\tau_2$") means that the data of the term with type $\tau_1$ flows into another term with type $\tau_2$.

```
fromMaybe p 0
where p = Just 1
      fromMaybe x d = case x of Just a: a | Nothing: d
```

The *data flow* from the data structure allocation `Just 1` to its consuming `case` expression: `Just 1` → `p` → `x` → `case x of Just … | Nothing …`

# Subtype Inference

```
fromMaybe p 0
where p = Just 1
      fromMaybe x d = case x of Just a: a | Nothing: d
```

- *subtyping information* collected during subtype inference:

  $\text{Just } 1 <: \tau_p \quad \tau_p <: \tau_x \quad \tau_x <: \{\text{Just } \tau_a \mid \text{Nothing}\}$

  after resolving the above iequalities (chaining them together), we get

  $\text{Just } 1 <: \{\text{Just } \tau_a \mid \text{Nothing}\}$

  which naturally brings together the producer and consumer of the data structure `Just 1`, indicating an opportunity to eliminate it.

## 6.3. Elaboration

## Elaboration

By efficiently *resolving* the subtyping inequalities collected when doing subtyping inference using Simple-sub (Parreaux 2020) and *keeping track of* types with their corresponding terms, deforestation can be done in a novel way.

```
fromMaybe p 0
where p = Just 1
      fromMaybe x d = case x of Just a: a | Nothing: d
```

can be eventually transformed to

```
fromMaybe' p 0
where p' = let a = 1 in a
      fromMaybe' x d = x
```

# Elaboration

The transformation is done through *elaboration*, which rewrites a term according to the type information attached to it. Fusible **producers** will have types of data constructors, with the information that they are subtypes of types of their consumers; similarly for fusible **consumers**.

Rewriting is done by *importing* the body of consumer into the site where the data constructor is called, binding arguments using `let`, and leaving the body of new “consumer” empty.

- `p = Just 1` → `p' = let a = 1 in a`
- `fromMaybe x d = case x of Just a: a | Nothing: d` → `fromMaybe' x d = x`

## 6.4. A Recursive Example

# A Recursive Example

```
sum (enumerate x)
where
  enumerate n = if n ⩾ 0 then n : enumerate (n - 1) else []
  sum xs = case xs of { [] → 0; x : xs → x + sum xs }
```

---

- `n:enumerate (n - 1)` → `(enumerate x)` → `xs`(parameter of `sum`) → `case xs of { … }` , so this constructor call is transformed into

  `let x = n; xs = numerate' (n - 1) in x + sum' xs`

- `[]` → `(enumerate x)` → `xs`(parameter of `sum`) → `case xs of { … }`, so this constructor call is transformed into `0`

# A Recursive Example

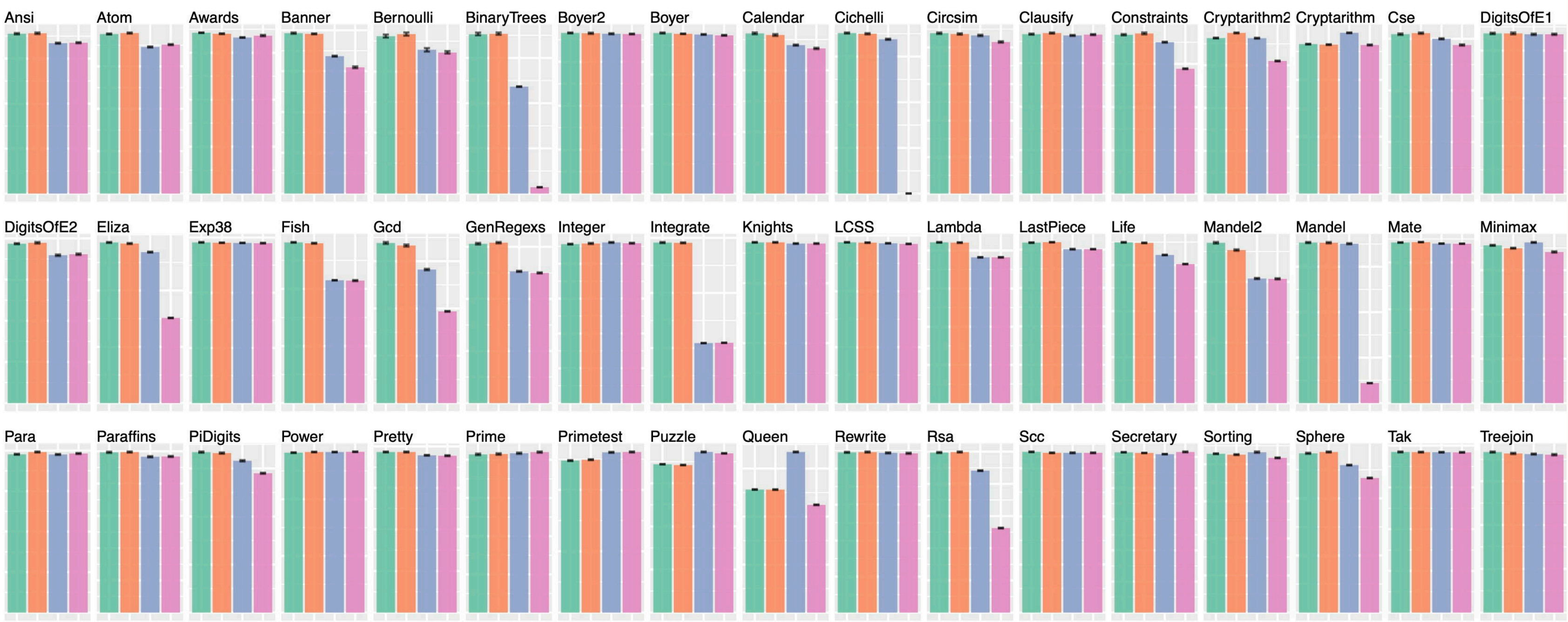
```
sum (enumerate x)
where
  enumerate n = if n ⩾ 0 then n : enumerate (n - 1) else []
  sum xs = case xs of { [] → 0; x : xs → x + sum xs }
```

---

```
sum' (enumerate' x)
where
  enumerate' n = if n ⩾ 0
    then let x = n; xs = numerate' (n - 1) in x + sum' xs
    else 0
  sum' xs = xs
```

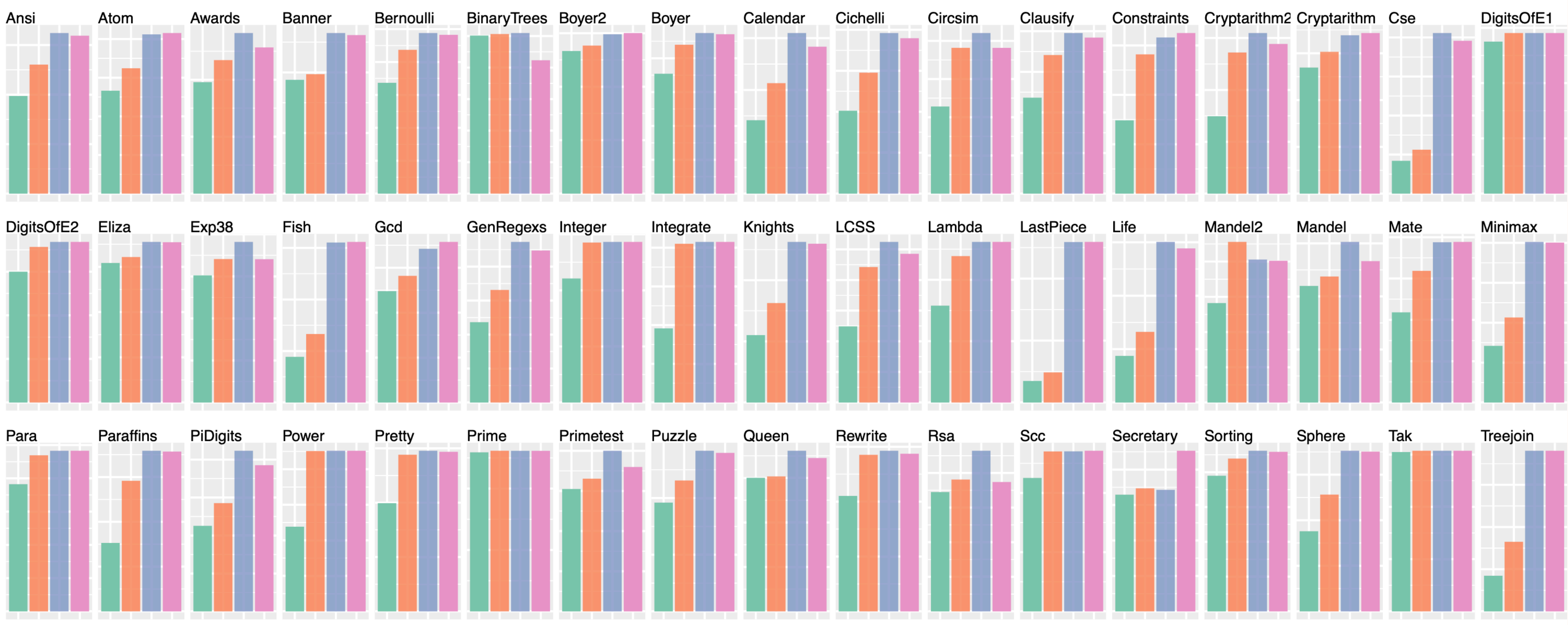
# Benchmark Results (51 tests in the *nofib* benchmark suite)

- average speedup: 14%
- leftmost: original program; rightmost: after all the steps of our transformation



# Benchmark Results (51 tests in the *nofib* benchmark suite)

- average code size increases by 1.8x



# Bibliography

COUTTS, Duncan, LESHCHINSKIY, Roman and STEWART, Don, 2007. Stream Fusion: From Lists to Streams to Nothing at All. In: *Proceedings of the 12th ACM SIGPLAN International Conference on Functional Programming*. Online. Freiburg, Germany: Association for Computing Machinery. 2007. p. 315. ICFP '07. ISBN 9781595938152. DOI 10.1145/1291151.1291199.

GILL, Andrew, LAUNCHBURY, John and PEYTON JONES, Simon L., 1993. A Short Cut to Deforestation. In: *Proceedings of the Conference on Functional Programming Languages and Computer Architecture*. Online. Copenhagen, Denmark: Association for Computing Machinery. 1993. p. 223. FPCA '93. ISBN 089791595X. DOI 10.1145/165180.165214.

KISELYOV, Oleg, BIBOUDIS, Aggelos, PALLADINOS, Nick and SMARAGDAKIS, Yannis, 2017. Stream fusion, to completeness. In: *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*. Online. Paris, France: Association for Computing Machinery. 2017. p. 285. POPL '17. ISBN 9781450346603. DOI 10.1145/3009837.3009880.

PARREAUX, Lionel, 2020. The Simple Essence of Algebraic Subtyping: Principal Type Inference with Subtyping Made Easy (Functional Pearl). *Proc. ACM Program. Lang*. Online. August 2020. Vol. 4, no. ICFP. DOI 10.1145/3409006.

PARREAUX, Lionel, SHAIKHHA, Amir and KOCH, Christoph E., 2017. Quoted staged rewriting: a practical approach to library-defined optimizations. In: *Proceedings of the 16th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences.* Online. Vancouver, BC, Canada: Association for Computing Machinery. 2017. p. 131. GPCE 2017. ISBN 9781450355247. DOI 10.1145/3136040.3136043.

PEYTON JONES, Simon, TOLMACH, Andrew and HOARE, Tony, 2001. Playing by the rules: rewriting as a practical optimisation technique in GHC. In: *2001 Haskell Workshop.* Online. September 2001. Available from: https://www.microsoft.com/en-us/research/publication/playing-by-the-rules-rewriting-as-a-practical-optimisation-technique-in-ghc/

SØRENSEN, M. H., GLÜCK, R. and JONES, N. D., 1996. A positive supercompiler. *Journal of Functional Programming*. 1996. Vol. 6, no. 6p. 811. DOI 10.1017/S0956796800002008.

TURCHIN, Valentin F., 1986. The Concept of a Supercompiler. *ACM Trans. Program. Lang. Syst.* Online. June 1986. Vol. 8, no. 3p. 292. DOI 10.1145/5956.5957.

WADLER, Philip, 1990. Deforestation: transforming programs to eliminate trees. *Theoretical Computer Science.* Online. 1990. Vol. 73, no. 2p. 231–248. DOI https://doi.org/10.1016/0304-3975(90)90147-A.