A Formal Study of Fusion and Lightweight Fusion for Lists

Sanjay Adhith

March 2026

A Formal Study of **Fusion** and Lightweight Fusion for Lists

Sanjay Adhith

March 2026

What is fusion?

A common pattern

- An inner function produces an intermediate data structure.
- An outer function consumes it.

A common pattern

- An inner function produces an intermediate data structure.
- An outer function consumes it.

- We would like to eliminate the intermediate data structure.

A common pattern

- An inner function produces an intermediate data structure.
- An outer function consumes it.

- We would like to eliminate the intermediate data structure.

- This is called *program fusion*.

A quick history

• Burstall and Darlington, 1977	fold/unfold
• Wadler, 1990	deforestation
• Chin, 1992	safe fusion
• Gill, Launchbury, and Peyton Jones, 1993	short-cut fusion
• Launchbury and Sheard, 1995	warm fusion

The point of interest

Lightweight fusion.

The point of interest

Lightweight fusion.

Ohori and Sasano, 2007.

Lightweight fusion

- A mechanical procedure.
- Four steps.
- Built on fold/unfold.

Lightweight fusion

- A mechanical procedure.
- Four steps.
- Built on fold/unfold.

- We illustrate it with a canonical example.

A canonical example

```
Fixpoint map_sq (vs : list nat) : list nat :=
  match vs with
  | nil     => nil
  | v :: vs' => v*v :: map_sq vs'
  end.

Fixpoint sum (vs : list nat) : nat :=
  match vs with
  | nil     => 0
  | v :: vs' => v + sum vs'
  end.
```

On a concrete input

```
map_sq (1 :: 2 :: 3 :: 4 :: nil)  =  1 :: 4 :: 9 :: 16 :: nil

sum    (1 :: 4 :: 9 :: 16 :: nil)  =  30
```

- `map_sq` constructs a list.
- `sum` consumes it.

The composition

```
Definition comp (vs : list nat) : nat :=
  sum (map_sq vs).
```

- We want a fused function that constructs no intermediate list.

Step 1: inline the inner function

```
Definition comp_step_1 (vs : list nat) : nat :=
  sum (match vs with
       | nil      => nil
       | v :: vs' => v*v :: map_sq vs'
       end).
```

Step 2: distribute the outer function

```
Definition comp_step_2 (vs : list nat) : nat :=
  match vs with
  | nil      => sum nil
  | v :: vs' => sum (v*v :: map_sq vs')
  end.
```

The `AppDist` rule. Sound for any strict outer function. (Ohori and Sasano)

Step 3: inline the outer function, simplify

```
Definition comp_step_3 (vs : list nat) : nat :=
  match vs with
  | nil      => 0
  | v :: vs' => v*v + sum (map_sq vs')
  end.
```

Step 4: generate a new recursive binding

```
Fixpoint sum_map_sq (vs : list nat) : nat :=
  match vs with
  | nil      => 0
  | v :: vs' => v*v + sum_map_sq vs'
  end.
```

No intermediate list.

Correct by calculation.

Summary so far

- Four mechanical steps.
- No intermediate list in the result.
- In the report, the same procedure was applied to:
  - big-step machines from small-step machines (Dyck words);
  - a palindrome checker;
  - a tail-recursive powerset function;
  - an example that defeated Pardo and Domínguez's fuser.

Roadmap

- What is fusion?
- Lightweight fusion, step by step. *(done)*

- A second example: `dropWhile` ∘ `filter`. *(next)*
- Future work.

Pardo and Domínguez, 2006

- They give an example where their automatic fuser fails.
- They calculate the fused program by hand, with algebraic laws.

Pardo and Domínguez, 2006

- They give an example where their automatic fuser fails.
- They calculate the fused program by hand, with algebraic laws.

- We reach the same program by ordinary lightweight fusion.

The candidates

```
Fixpoint filter (q : V -> bool) (vs : list V) : list V :=
  match vs with
  | nil      => nil
  | v :: vs' => if q v then v :: filter q vs'
                       else filter q vs'
  end.

Fixpoint dropWhile (p : V -> bool) (vs : list V) : list V :=
  match vs with
  | nil      => nil
  | v :: vs' => if p v then dropWhile p vs'
                       else v :: vs'
  end.

Definition dW_filter p q vs := dropWhile p (filter q vs).
```

Step 1: inline the inner function

```
Definition dW_filter_step_1 (p q : V -> bool) (vs : list V) : list V :=
  dropWhile p (match vs with
               | nil      => nil
               | v :: vs' =>
                   if q v then v :: filter q vs'
                          else filter q vs'
               end).
```

Step 2: distribute the outer function

```
Definition dW_filter_step_2 (p q : V -> bool) (vs : list V) : list V :=
  match vs with
  | nil      => dropWhile p nil
  | v :: vs' =>
      if q v then dropWhile p (v :: filter q vs')
             else dropWhile p (filter q vs')
  end.
```

Step 3: inline the outer function, simplify

```
Definition dW_filter_step_3 (p q : V -> bool) (vs : list V) : list V :=
  match vs with
  | nil      => nil
  | v :: vs' =>
      if q v
      then if p v then dropWhile p (filter q vs')
                  else v :: filter q vs'
      else dropWhile p (filter q vs')
  end.
```

Step 4: generate a new recursive binding

```
Fixpoint dWfil (p q : V -> bool) (vs : list V) : list V :=
  match vs with
  | nil      => nil
  | v :: vs' =>
      if q v
      then if p v then dWfil p q vs'
                  else v :: filter q vs'
      else dWfil p q vs'
  end.
```

Exactly the program Pardo and Domínguez obtained by hand.

Summary

- Lightweight fusion reproduces many ad-hoc fusion results.
- Here, the algebraic derivation required ingenuity.
- The mechanical derivation did not.

Roadmap

- What is fusion?
- Lightweight fusion, step by step. *(done)*
- A second example: `dropWhile` ∘ `filter`. *(done)*

- Future work. *(next)*

What we have shown

- Lightweight fusion is a mechanical four-step procedure.
- We applied it to four examples drawn from the literature.
- It reached a program where an automatic algebraic fuser did not.
- We formalised the equivalence proofs in the Rocq Proof Assistant.

Future work

- Formalise `AppDist` in the Rocq Proof Assistant.
- A comparative study with Launchbury's warm fusion.
- The example where fusion yields a less efficient program: when should we fuse?

A message to Prof

- In the comments on my first report, the feedback was that the formal statements alone made the report inaccessible.

- So in this final report, I have added an informal description of each theorem and an explanation of its intent.

Thank you for your attention.

Any questions?

Pardo and Domínguez's counterexample

```
Fixpoint tails (vs : list nat) : list (list nat) :=
  match vs with
  | nil      => nil
  | v :: vs' => vs' :: tails vs'
  end.

Fixpoint map (vs : list nat) (f : nat -> nat) : list nat :=
  match vs with
  | nil      => nil
  | v :: vs' => f v :: map vs' f
  end.

Definition comp vs f := tails (map vs f).
```

Both `tails` and `map` run in linear time.

The fused program is quadratic

```
Fixpoint comp_step_4 (vs : list nat) (f : nat -> nat) :=
  match vs with
  | nil      => nil
  | v :: vs' => (map vs' f) :: comp_step_4 vs' f
  end.
```

`map` is called at every recursive step, so our program now runs in quadratic time.