

Bird Meertens Formalism

– Fusion and Tupling –

Zhenjiang Hu, Wei Zhang

Department of Computer Science and Technology
Peking University

December 24, 2021

Outline

1 Fusion

2 Tupling

max

Consider the function to compute the maximum of a list (by reusing *sort*):

$$\begin{aligned} \text{max} &: [Int] \rightarrow Int \\ \text{max} &= \text{head} \cdot \text{sort} \end{aligned}$$

max

Consider the function to compute the maximum of a list (by reusing *sort*):

$$\begin{aligned} \text{max} &: [Int] \rightarrow Int \\ \text{max} &= \text{head} \cdot \text{sort} \end{aligned}$$

where *sort* is defined by

$$\begin{aligned} \text{sort} &= \text{foldr insert []} \\ \text{insert } a [] &= [a] \\ \text{insert } a (b : x) &= \text{if } a \geq b \text{ then } a : (b : x) \\ &\quad \text{else } b : \text{insert } a x. \end{aligned}$$

How to eliminate all intermediate results in computing *max*?

reverse

Consider the following function to reverse a list:

$$\begin{aligned} \text{rev } x &= \text{fastrev}' x [] \\ \text{fastrev}' x y &= \text{reverse } x ++ y \end{aligned}$$

where

$$\text{reverse} = \text{foldr } (\lambda a r. r ++ [a]) []$$

How to eliminate the intermediate list in computing *fastrev'*?

reverse

Consider the following function to reverse a list:

$$\begin{aligned}
 \text{rev } x &= \text{fastrev}' x [] \\
 \text{fastrev}' x y &= \text{reverse } x ++ y \\
 &\textbf{where} \\
 &\quad \text{reverse} = \text{foldr } (\lambda a r. r ++ [a]) []
 \end{aligned}$$

How to eliminate the intermediate list in computing *fastrev'*?

Exercise

Show evaluation steps of *rev* [1, 2, 3, 4], and explain that (*rev xs*) is a quadratic program.

Fusion Law for Foldr

Lemma (Foldr Fusion)

$$\frac{f(a \oplus r) = a \otimes f r}{f \cdot \text{foldr} (\oplus) e = \text{foldr} (\otimes) (f e)}$$

Fusion Law for Foldr

Lemma (Foldr Fusion)

$$\frac{f(a \oplus r) = a \otimes f r}{f \cdot \text{foldr } (\oplus) e = \text{foldr } (\otimes) (f e)}$$

Or written as

$$\frac{f(a \oplus r) = a \otimes f r}{f \cdot \oplus \not\leftarrow e = \otimes \not\leftarrow f e}$$

Fusion: max

Consider the fusion for *max*:

$$\mathit{max} = \mathit{head} \cdot \mathit{foldr} \ \mathit{insert} \ []$$

where we assume that $\mathit{max} [] = -\infty$.

Fusion: max

Consider the fusion for *max*:

$$\mathit{max} = \mathit{head} \cdot \mathit{foldr} \ \mathit{insert} \ []$$

where we assume that $\mathit{max} [] = -\infty$.

To apply the foldr fusion lemma, we consider calculation of $\mathit{head} (\mathit{insert} \ a \ r)$.

We calculate as follows.

- For the case of $r = []$, we have:

$$\begin{aligned} & \text{head} (\text{insert } a []) \\ = & \quad \{ \text{def. of } \text{insert} \} \\ & \text{head } [a] \\ = & \quad \{ \text{def. of } \text{head} \} \\ & a \end{aligned}$$

Fusion Example: max

- For the case of $r = b : x$, we have:

$$\begin{aligned}
 & \text{head } (\text{insert } a \ (b : x)) \\
 = & \quad \{ \text{def. of insert} \} \\
 & \text{head } (\text{if } a \geq b \text{ then } a : (b : x) \text{ else } b : \text{insert } a \ x) \\
 = & \quad \{ \text{distribute head over if} \} \\
 & \text{if } a \geq b \text{ then head } (a : (b : x)) \text{ else head } (b : \text{insert } a \ x) \\
 = & \quad \{ \text{def. of head} \} \\
 & \text{if } a \geq b \text{ then } a \text{ else } b \\
 = & \quad \{ \text{assumption: } r = b : x \} \\
 & \text{if } a \geq \text{head } r \text{ then } a \text{ else head } r
 \end{aligned}$$

In summary, we have

$$\begin{aligned} \text{head} (\text{insert } a \ r) &= a \otimes \text{head } r \\ \text{where } a \otimes r &= \text{if } a \geq r \text{ then } a \text{ else } r \end{aligned}$$

It follows from the foldr fusion lemma that we get the following new definition for *max*.

$$\text{max} = \text{foldr} (\otimes) (-\infty)$$

A linear time program!

Fusion Example: Fast Reverse

Consider fusion of the following program:

$$\mathit{fastrev} \ x \ y = \mathit{reverse} \ x \ ++ \ y$$

Fusion Example: Fast Reverse

Consider fusion of the following program:

$$\mathit{fastrev} \ x \ y = \mathit{reverse} \ x \ ++ \ y$$

Exercise

What is the intermediate list in the above computation?

Fusion Example: Fast Reverse

Consider fusion of the following program:

$$\text{fastrev}'\ x\ y = \text{reverse}\ x\ ++\ y$$

Exercise

What is the intermediate list in the above computation?

We can see where fusion calculation is application if we rewrite the definition.

$$\begin{aligned}\text{fastrev}'\ x &= (++)\ (\text{reverse}\ x) \\ &= \underline{((++))} \cdot \underline{\text{foldr}\ (\lambda a\ r.\ r\ ++\ [a])\ []}\ x\end{aligned}$$

Let us calculate the fusion condition:

$$\begin{aligned}
 & (++) (r ++ [a]) \\
 = & \quad \{ \eta \text{ expansion} \} \\
 & \lambda y. (++) (r ++ [a]) y \\
 = & \quad \{ \text{section notation} \} \\
 & \lambda y. (r ++ [a]) ++ y \\
 = & \quad \{ \text{associativity of } ++ \} \\
 & \lambda y. r ++ ([a] ++ y)
 \end{aligned}$$

Marching it with $a \otimes ((++) r)$ gives

$$a \otimes r' = \lambda y. r' ([a] ++ y)$$

So we get

$$\begin{aligned} \text{fastrev}'\ x &= \text{foldr } (\otimes) ((++) [])\ x \\ \text{where } a \otimes r' &= \lambda y. r' ([a] ++ y) \end{aligned}$$

So we get

$$\begin{aligned} \text{fastrev}'\ x &= \text{foldr } (\otimes) ((++) [])\ x \\ \textbf{where } a \otimes r' &= \lambda y. r' ([a] ++ y) \end{aligned}$$

which is the same as

$$\begin{aligned} \text{fastrev}'\ x &= \text{foldr } (\otimes)\ \text{id}\ x \\ \textbf{where } a \otimes r' &= \lambda y. r' ([a] ++ y) \end{aligned}$$

So we get

$$\begin{aligned} \text{fastrev}'\ x &= \text{foldr } (\otimes) ((+) [])\ x \\ \textbf{where } a \otimes r' &= \lambda y. r' ([a] ++ y) \end{aligned}$$

which is the same as

$$\begin{aligned} \text{fastrev}'\ x &= \text{foldr } (\otimes)\ \text{id}\ x \\ \textbf{where } a \otimes r' &= \lambda y. r' ([a] ++ y) \end{aligned}$$

That is,

$$\begin{aligned} \text{fastrev}'\ []\ y &= y \\ \text{fastrev}'\ (a : r)\ y &= \text{fastrev}'\ r\ (a : y) \end{aligned}$$

So we get

$$\begin{aligned} \text{fastrev}'\ x &= \text{foldr}\ (\otimes)\ ((+)\ [])\ x \\ \textbf{where}\ a \otimes r' &= \lambda y. r' ([a] ++ y) \end{aligned}$$

which is the same as

$$\begin{aligned} \text{fastrev}'\ x &= \text{foldr}\ (\otimes)\ \text{id}\ x \\ \textbf{where}\ a \otimes r' &= \lambda y. r' ([a] ++ y) \end{aligned}$$

That is,

$$\begin{aligned} \text{fastrev}'\ []\ y &= y \\ \text{fastrev}'\ (a : r)\ y &= \text{fastrev}'\ r\ (a : y) \end{aligned}$$

A linear time algorithm!

Homework BMF 3-1

Using the foldr fusion lemma to prove the following two equations.

- ① $\text{foldr } (\oplus) \ e \cdot \text{map } f = \text{foldr } (\lambda a \ r. f \ a \oplus \ r) \ e$
- ② $\text{map } f \cdot \text{map } g = \text{map } (f \cdot g)$

Outline

- 1 Fusion
- 2 Tupling

Exercise

What is the time complexity for the following function that computes the maximum element from a list.

$$\begin{array}{lll} \text{maximum } [a] & = & a \\ \text{maximum } (a : x) \mid a > \text{maximum } x & = & a \\ & \mid \text{otherwise} & = \text{maximum } x \end{array}$$

Enumerating Bigger Elements

Enumerate all bigger elements in a list. An element is bigger if it is greater than the sum of the elements that follow it till the end of the list.

$$\text{biggers } [3, 10, 4, -2, 1, 3] = [10, 4, 3]$$

Enumerating Bigger Elements

Enumerate all bigger elements in a list. An element is bigger if it is greater than the sum of the elements that follow it till the end of the list.

$$\text{biggers } [3, 10, 4, -2, 1, 3] = [10, 4, 3]$$

$$\text{biggers } [] = []$$

$$\text{biggers } (a : x) = \text{if } a > \text{sum } x \text{ then } a : \text{biggers } x \text{ else } \text{biggers } x$$

$$\text{sum } [] = 0$$

$$\text{sum } (a : x) = a + \text{sum } x$$

Enumerating Bigger Elements

Enumerate all bigger elements in a list. An element is bigger if it is greater than the sum of the elements that follow it till the end of the list.

$$\text{biggers } [3, 10, 4, -2, 1, 3] = [10, 4, 3]$$

$$\text{biggers } [] = []$$

$$\text{biggers } (a : x) = \text{if } a > \text{sum } x \text{ then } a : \text{biggers } x \text{ else } \text{biggers } x$$

$$\text{sum } [] = 0$$

$$\text{sum } (a : x) = a + \text{sum } x$$

How can we optimize this program?

Enumerating Bigger Elements

Enumerate all bigger elements in a list. An element is bigger if it is greater than the sum of the elements that follow it till the end of the list.

$$\text{biggers } [3, 10, 4, -2, 1, 3] = [10, 4, 3]$$

$$\text{biggers } [] = []$$

$$\text{biggers } (a : x) = \text{if } a > \text{sum } x \text{ then } a : \text{biggers } x \text{ else } \text{biggers } x$$

$$\text{sum } [] = 0$$

$$\text{sum } (a : x) = a + \text{sum } x$$

How can we optimize this program?

Exercise: Is *biggers* a foldr?

Definition (Mutumorphism)

Functions f_1, \dots, f_n are said to form a mutumorphism if each f_i ($i = 1, 2, \dots, n$) is defined in the following form:

$$\begin{aligned} f_i [] &= e_i \\ f_i (a : x) &= a \oplus_i (f_1 x, f_2 x, \dots, f_n x) \end{aligned}$$

where e_i ($i = 1, 2, \dots, n$) are given constants and \oplus_i ($i = 1, 2, \dots, n$) are given binary functions. We represent the function $f x = (f_1 x, \dots, f_n x)$ as follows.

$$f = \llbracket (e_1, \dots, e_n), (\oplus_1, \dots, \oplus_n) \rrbracket.$$

Expressive Power of Mutumorphism

- *foldr* is a special case:

$$\text{foldr } (\oplus) e = \llbracket (e), (\text{oplus}) \rrbracket$$

- It covers all primitive recursive functions on lists.

$$\begin{aligned} \text{prim } [] &= e \\ \text{prim } (a : x) &= F(a, x, \text{prim } x) \end{aligned}$$

This is because we can *prim* is mutually defined with the identity function on lists.

biggers as a Mutumorphism

$biggers = fst \circ \llbracket ([], 0), (\oplus_1, \oplus_2) \rrbracket$
where $a \oplus_1 (r, s) = \text{if } a > s \text{ then } a : r \text{ else } r$
 $a \oplus_2 (r, s) = a + s$

Lemma (Mutu-Tupling)

$$\begin{aligned} & \llbracket (e_1, e_2, \dots, e_n), (\oplus_1, \oplus_2, \dots, \oplus_n) \rrbracket \\ &= \text{foldr } (\oplus) (e_1, e_2, \dots, e_n) \\ & \quad \text{where } a \oplus r = (a \oplus_1 r, a \oplus_2 r, \dots, a \oplus_n r) \end{aligned}$$

Consider, as an example, to apply the mutu-tupling lemma to *biggers*.

$$\begin{aligned}
 & \textit{biggers} \\
 = & \quad \{ \text{mutumorphism for } \textit{biggers} \} \\
 & \textit{fst} \circ \llbracket ([], 0), (\oplus_1, \oplus_2) \rrbracket \\
 = & \quad \{ \text{mutu-tupling lemma} \} \\
 & \textit{fst} \circ \textit{foldr} (\oplus) ([], 0) \\
 & \quad \textbf{where } a \oplus (r, s) = (\textbf{if } a > s \textbf{ then } a : r \textbf{ else } r, a + s)
 \end{aligned}$$

Inlining *foldr* in the derived program gives the following readable recursive program:

```
biggers x = let (r, s) = tup x in r  
           where tup [] = ([], 0)  
                tup (a : x) = let (r, s) = tup x  
                               in (if a > s then a : r else r, a + s)
```

Lemma (Foldr-Tupling)

$$(foldr (\oplus_1) e_1 x, foldr (\oplus_2) e_2 x) = foldr (\oplus) (e_1, e_2) x$$

where $a \oplus (r_1, r_2) = (a \oplus_1 r_1, a \oplus_2 r_2)$

For example, the following program for computing the average of a list:

$$\text{average } x = \text{sum } x / \text{length } x$$

can be transformed into the following with the foldr-tupling lemma.

$$\begin{aligned} \text{average } x = & \mathbf{let} \ (s, l) = \text{tup } x \mathbf{in} \ s / l \\ & \mathbf{where} \ \text{tup} = \text{foldr} \ (\lambda a \ (s, l). \ (a + s, 1 + l)) \ (0, 0) \end{aligned}$$

Homework BMF 3-2

(1) Using tupling transformation to derive an efficient program for computing *tailsums*.

$$\begin{aligned} \text{tailsums } [] &= [0] \\ \text{tailsums } (a : x) &= \text{tailsums } x ++ [a + \text{sum } x] \end{aligned}$$

(2) Code the efficient program in Haskell.