Idea 1:

→ First, we introduce unique variable names.
→ Then, we only substitute functions which are statically within the scope of the same global variables as the application.
→ For every expression, we determine all function definitions with this property.
... in the Example:
\[
\begin{align*}
    f (g(x)) \equiv & \ 1 + x \\
\end{align*}
\]

\[
\begin{align*}
    \text{let } x = 1 \\
    \text{in let } \quad & f = \text{let } x_1 = 2 \\
    \text{in let } \quad & \quad \text{fun } y \rightarrow y + x_1 \\
    \text{fun } x \rightarrow & f \\
\end{align*}
\]

\[
\begin{align*}
    \left(\text{fun } x \rightarrow f\right) \circ \circ \Rightarrow \text{let } x = x_2 \rightarrow 1 + x_2 \\
\end{align*}
\]

the application \( f \ x \) is not in the scope of \( x_1 \)

\[
\begin{align*}
    \Rightarrow \text{we first duplicate the definition of } \ x_1 : \\
\end{align*}
\]

\[
\begin{align*}
    \text{let } x = 1 \\
    \text{in let } \quad x_1 = 2 \\
    \text{in let } \quad f = \text{fun } y \rightarrow y + x_1 \\
    \text{fun } x \rightarrow & f \\
\end{align*}
\]

\[
\begin{align*}
    \Rightarrow \text{now we can apply inlining:} \\
\end{align*}
\]

Removing variable-variable-assignments, we arrive at:
let \( x = 1 \)
in let \( x_1 = 2 \)
in let \( f = \text{fun } y \rightarrow y + x_1 \)
in
\[
\begin{align*}
\text{let } & y = x \\
\text{in } & y + x_1
\end{align*}
\]

Removing variable-variable-assignments, we arrive at:

\[
\begin{align*}
\text{let } & x = 1 \\
\text{in let } & x_1 = 2 \\
\text{in let } & f = \text{fun } y \rightarrow y + x_1 \\
\text{in } & x + x_1
\end{align*}
\]

**Idea 2:**

→ We apply our value analysis.
→ We ignore global variables :-(
→ We only substitute functions **without** free variables :-))

**Example:** The \text{map-Function}

\[
\begin{align*}
\text{let rec } & f = \text{fun } x \rightarrow x \cdot x \\
\text{and } & \text{map} = \text{fun } g \rightarrow \text{fun } x \rightarrow \text{match } x \\
\text{with } & [] \rightarrow [] \\
& | x :: xs \rightarrow g x :: \text{map } g xs \\
\text{in } & \text{map } f \text{ list}
\end{align*}
\]

- The actual parameter \( f \) in the application \( \text{map } g \) is always \( \text{fun } x \rightarrow x \cdot x \) :-)
- Therefore, \( \text{map } g \) can be specialized to a new function \( h \) defined by:

\[
\begin{align*}
\text{let } & g = \text{fun } x \rightarrow x \cdot x \\
\text{in } & \text{fun } x \rightarrow \text{match } x \\
\text{with } & [] \rightarrow [] \\
& | x :: xs \rightarrow g x :: \text{map } g xs
\end{align*}
\]
• The actual parameter \( f \) in the application \( \text{map} \ g \) is always \( \text{fun} \ x \rightarrow x \cdot x \) :-)

• Therefore, \( \text{map} \ g \) can be specialized to a new function \( h \) defined by:

\[
\text{fun} \ g \rightarrow \text{let } \text{map} \ f = \text{map} f \text{ in } \text{map} \ f
\]

\[
\text{map} \ f = \text{let } g = \text{fun} \ x \rightarrow x \cdot x \text{ in } \text{fun} \ x \rightarrow \text{match } x \text{ with } \|
\text{[] } \rightarrow \text{[]}
\text{\| } x::xs \rightarrow g x :: \text{map} g xs
\]

Inlining the function \( g \) yields:

\[
h = \text{let } g = \text{fun} \ x \rightarrow x \cdot x \text{ in } \text{fun} \ x \rightarrow \text{match } x \text{ with } \|
\text{[] } \rightarrow \text{[]}
\text{\| } x::xs \rightarrow (\text{let } x = x \text{ in } x \cdot x) :: h xs
\]

Removing useless definitions and variable-variable assignments yields:

\[
h = \text{fun} \ x \rightarrow \text{match } x \text{ with } \|
\text{[] } \rightarrow \text{[]}
\text{\| } x::xs \rightarrow x \cdot x :: h xs
\]
4.5 Deforestation

- Functional programmers love to collect intermediate results in lists which are processed by higher-order functions.
- Examples of such higher-order functions are:

\[
\text{map } \equiv \text{fun } f \to \text{fun } l \to \text{match } l \text{ with } [ ] \to [ ] \\
| \quad x :: xs \to f x :: \text{map } f \, xs
\]

- The actual parameter \( f \) in the application \( \text{map } g \) is always \( \text{fun } x \to x \cdot x \):)
- Therefore, \( \text{map } g \) can be specialized to a new function \( h \) defined by:

\[
\begin{align*}
\text{h } & \equiv \text{let } g = \text{fun } x \to x \cdot x \\
& \quad \text{in } \text{fun } x \to \text{match } x \\
& \quad \text{with } [ ] \to [ ] \\
& \quad | \quad x :: xs \to g x :: \text{map } g \, xs
\end{align*}
\]

4.5 Deforestation

- Functional programmers love to collect intermediate results in lists which are processed by higher-order functions.
- Examples of such higher-order functions are:

\[
\text{map } f \equiv \text{fun } l \to \text{match } l \text{ with } [ ] \to [ ] \\
| \quad x :: xs \to f x :: \text{map } f \, xs
\]

\[
\begin{align*}
\text{filter} : (\alpha \to \text{bool}) \to \alpha \text{ list } & \to \alpha \text{ list } \\
\text{filter } & \equiv \text{fun } p \to \text{fun } l \to \text{match } l \text{ with } [ ] \to [ ] \\
& \quad | \quad x :: xs \to \text{if } p x \text{ then } x :: \text{filter } p \, xs \\
& \quad \text{else } \text{filter } p \, xs
\end{align*}
\]

\[
\begin{align*}
\text{foldl} : (\alpha \times \text{list}) \to \alpha & \to \text{list } \\
\text{foldl} & \equiv \text{fun } a \to \text{fun } l \to \text{match } l \text{ with } [ ] \to a \\
& \quad | \quad x :: xs \to \text{foldl } f (f a \, x) \, xs
\end{align*}
\]

\[
\begin{align*}
\text{foldl } a \, (x_1, x_2, \ldots) & = \left( a \odot x_1 \odot x_2 \odot \ldots \right)_{\omega}
\end{align*}
\]
filter = \( \text{fun } p \rightarrow \text{fun } l \rightarrow \text{match } l \text{ with } [\ ] \rightarrow [\ ] \)
\[ | x::xs \rightarrow \text{if } p x \text{ then } x :: \text{filter } p xs \text{ else filter } p xs) \]

foldl = \( \text{fun } f \rightarrow \text{fun } a \rightarrow \text{fun } l \rightarrow \text{match } l \text{ with } [\ ] \rightarrow a \)
\[ | x::xs \rightarrow \text{foldl } f (f a x) xs) \]

Example:

\[ [x_1, x_2, x_3] \rightarrow [a, a, a] \]

\[ \text{sum} = \text{foldl } (+) 0 \]

\[ \text{length} = \text{let } f = \text{map } (\text{fun } x \rightarrow 1) \]
\[ \text{in} \ \text{comp } \text{sum } f \]

\[ \text{dev} = \text{fun } l \rightarrow \text{let } s_1 = \text{sum } l \]
\[ n = \text{length } l \]
\[ \text{mean } = s_1 / n \]
\[ l_1 = \text{map } (\text{fun } x \rightarrow x - \text{mean}) l \]
\[ l_2 = \text{map } (\text{fun } x \rightarrow x \cdot x) l_1 \]
\[ s_2 = \text{sum } l_2 \]
\[ \text{in } s_2 / n \]
Example:

```
sum = foldl (+) 0
length = let f = map (fun x -> 1)
          in comp sum f
dev = fun l -> let s1 = sum l
        n = length l
        mean = s1/n
        l1 = map (fun x -> x - mean) l
        l2 = map (fun x -> x * x) l1
        s2 = sum l2
        in s2/n
```

Observations:

- Explicit recursion does no longer occur!
- The implementation creates unnecessary intermediate data-structures!

  length could also be implemented as:

  ```
  length = let f = fun a -> fun x -> a + 1
           in foldl f 0
  ```

- This implementation avoids to create intermediate lists !!!

Observations:

- Explicit recursion does no longer occur!
- The implementation creates unnecessary intermediate data-structures!

  length could also be implemented as:

  ```
  length = let f = fun a -> fun x -> a + 1
           in foldl f 0
  ```

- This implementation avoids to create intermediate lists !!!
id = \text{fun } x \rightarrow x \\
\text{comp} = \text{fun } f \rightarrow \text{fun } g \rightarrow \text{fun } x \rightarrow f \,(g \,x) \\
\text{comp}_1 = \text{fun } f \rightarrow \text{fun } g \rightarrow \text{fun } x_1 \rightarrow \text{fun } x_2 \rightarrow f \, (g \,x_1) \,x_2 \\
\text{comp}_2 = \text{fun } f \rightarrow \text{fun } g \rightarrow \text{fun } x_1 \rightarrow \text{fun } x_2 \rightarrow f \,x_1 \, (g \,x_2) \\

\text{Simplification Rules:} \\
\text{comp id } f = \text{comp f id } = f \\
\text{comp}_1 f \, id = \text{comp}_2 f \, id = f \\
\text{map id } = id \\
\text{comp (map f) (map g)} = \text{map (comp f g)} \\
\text{comp (foldl f a) (map g)} = \text{foldl (comp}_2 f \,g) \,a \\

\text{foldl} : (\alpha \rightarrow \beta \rightarrow \alpha) \rightarrow \alpha \rightarrow \beta \rightarrow \text{id } \rightarrow \alpha \\

\text{Simplification Rules:} \\
\text{comp id } f = \text{comp f id } = f \\
\text{comp}_1 f \, id = \text{comp}_2 f \, id = f \\
\text{map id } = id \\
\text{comp (map f) (map g)} = \text{map (comp f g)} \\
\text{comp (foldl f a) (map g)} = \text{foldl (comp}_2 f \,g) \,a
id = fun x → x
comp = fun f → fun g → fun x → f (g x)

comp₁ = fun f → fun g → fun x₁ → fun x₂ →
        f₁ (g x₁) x₂

comp₂ = fun f → fun g → fun x₁ → fun x₂ →
        f₁ (g x₂)

Simplification Rules:
comp id f = comp f id = f
comp₁ f id = comp₂ f id = f
map id = id
comp (map f) (map g) = map (comp f g)
comp (foldl f a) (map g) = foldl (comp₂ f g) a
comp (filter p₁) (filter p₂) = filter (fun x → if p₂ x then p₁ x else false)
comp (foldl f a) (filter p) = let h = fun a → fun x → if p x then f a x else a
          in foldl h a

Warning:
Function compositions also could occur as nested function calls ...

id x = x
map id l = l
map f (map g l) = map (comp f g) l
foldl f a (map g l) = foldl (comp₂ f g) a l
filter p₁ (filter p₂ l) = filter (fun x → p₁ x ∧ p₂ x) l
foldl f a (filter p₁ l) = let h = fun a → fun x → if p₁ x then f a x else a
          in foldl h a l

Warning:
Function compositions also could occur as nested function calls ...

id x = x
map id l = l
map f (map g l) = map (comp f g) l
foldl f a (map g l) = foldl (comp₂ f g) a l
filter p₁ (filter p₂ l) = filter (fun x → p₁ x ∧ p₂ x) l
foldl f a (filter p₁ l) = let h = fun a → fun x → if p₁ x then f a x else a
          in foldl h a l
Example, optimized:

\[
\begin{align*}
\text{sum} &= \text{foldl} \ (+) \ 0 \\
\text{length} &= \text{let } f = \text{comp}_3 \ (+) \ (\text{fun} \ x \to 1) \\
&\quad \text{in } \text{foldl} \ f \ 0 \\
\text{dev} &= \text{fun} \ l \to \text{let } s_1 = \text{sum} \ l \\
&\quad \quad n = \text{length} \ l \\
&\quad \quad \text{mean} = s_1 / n \\
&\quad \quad f = \text{comp} \ (\text{fun} \ x \to x \cdot x) \\
&\quad \quad \quad (\text{fun} \ x \to x - \text{mean}) \\
&\quad \quad g = \text{comp}_3 \ (+) \ f \\
&\quad \quad \quad \text{in } s_2 / n \\
&\quad \quad \quad \quad \left(0 + \left(f \times x_1\right) + \cdots + \left(f \times x_2\right)\right) + \cdots
\end{align*}
\]

Remarks:

- All intermediate lists have disappeared  
- Only \text{foldl} remain — i.e., loops  
- Compositions of functions can be further simplified in the next step by \textbf{Inlining}. 
- Inside \text{dev}, we then obtain:

\[
\begin{align*}
g &= \text{fun} \ a \to \text{fun} \ x \to \text{let } x_1 = x - \text{mean} \\
&\quad x_2 = x_1 \cdot x_1 \\
&\quad \text{in } a + x_2
\end{align*}
\]

- The result is a sequence of \textbf{let}-definitions !!!
Remarks:

- All intermediate lists have disappeared  
  :-(
- Only fold remain — i.e., loops  
  :-(
- Compositions of functions can be further simplified in the next step 
  by Inlining.
- Inside dev, we then obtain:

  \[ g = \text{fun } a \rightarrow \text{fun } x \rightarrow \text{let } x_1 = x - \text{mean} \]
  \[ x_2 = x_1 \cdot x_1 \]
  \[ \text{in } a + x_2 \]

- The result is a sequence of let definitions !!!

Extension: Tabulation

If the list has been created by tabulation of a function, the creation of the 
list sometimes can be avoided ...

\[
\text{tabulate'} = \text{fun } j \rightarrow \text{fun } f \rightarrow \text{fun } n \rightarrow \\
\quad \quad \quad \quad \text{if } j \geq n \text{ then } [] \\
\quad \quad \quad \quad \text{else } (f j) :: \text{tabulate'} (j + 1) f n \\
\text{tabulate} = \text{tabulate'} 0
\]

\[
\text{tabulate } f n = [f 0; f 1; \ldots; f (n - 1)]
\]

Then we have:

\[
\text{comp } (\text{map } f) (\text{tabulate } g) = \text{tabulate } (\text{comp } f g)
\]
\[
\text{comp } (\text{foldl } f a) (\text{tabulate } g) = \text{loop } (\text{comp } f g) a
\]

where:

\[
\text{loop'} = \text{fun } j \rightarrow \text{fun } f \rightarrow \text{fun } a \rightarrow \text{fun } n \rightarrow \\
\quad \quad \quad \quad \text{if } j \geq n \text{ then } a \\
\quad \quad \quad \quad \text{else } \text{loop'} (j + 1) f (f a) (f n)
\]
\[
\text{loop} = \text{loop'} 0
\]

Extension (2): List Reversals

Sometimes, the ordering of lists or arguments is reversed:

\[
\text{rev'} = \text{fun } a \rightarrow \text{fun } l \rightarrow \\
\quad \quad \quad \quad \text{match } l \text{ with } [] \rightarrow a \\
\quad \quad \quad \quad \quad \quad | x :: xs \rightarrow \text{rev'} (x :: a) xs
\]
\[
\text{rev} = \text{rev'} []
\]
\[
\text{comp rev rev} = \text{id}
\]
\[
\text{swap} = \text{fun } f \rightarrow \text{fun } x \rightarrow \text{fun } y \rightarrow f y x
\]
\[
\text{comp swap swap} = \text{id}
\]
Extension (2): List Reversals

Sometimes, the ordering of lists or arguments is reversed:

\[
\text{rev'} = \text{fun } a \rightarrow \text{fun } l \rightarrow \\
\quad \text{match } l \text{ with } [] \rightarrow a \\
\quad | \ x::xs \rightarrow \text{rev'}(x::a)xs
\]

\[
\text{rev} = \text{rev'}[]
\]

\[
\text{comp rev rev} = \text{id}
\]

\[
\text{swap} = \text{fun } f \rightarrow \text{fun } x \rightarrow \text{fun } y \rightarrow fyx
\]

\[
\text{comp swap swap} = \text{id}
\]

Discussion:

- The standard implementation of \text{foldr} is not tail-recursive.
- The last equation decomposes a \text{foldr} into two tail-recursive functions — at the price that an intermediate list is created.
- Therefore, the standard implementation is probably faster :)  
- Sometimes, the operation \text{rev} can also be optimized away ...

Extension: Tabulation

If the list has been created by tabulation of a function, the creation of the list sometimes can be avoided ...

\[
\text{tabulate'} = \text{fun } j \rightarrow \text{fun } f \rightarrow \text{fun } n \rightarrow \\
\quad \text{if } j \geq n \text{ then } [] \\
\quad \text{else } (fj) :: \text{tabulate'}(j+1)f n
\]

\[
\text{tabulate} = \text{tabulate'}0
\]

\[
\text{foldl} \quad \beta \rightarrow \alpha \rightarrow \alpha
\]

\[
\text{foldr} \quad \beta \rightarrow \alpha \rightarrow \alpha
\]

\[
\text{swap} \quad \beta \rightarrow \alpha \rightarrow \alpha
\]

\[
\text{swap} \quad \beta \rightarrow \alpha \rightarrow \alpha
\]
id = fun x → x

comp = fun f → fun g → fun x → f (g x)

comp₁ = fun f → fun g → fun x₁ → fun x₂ →
        f (g x₁) x₂

comp₂ = fun f → fun g → fun x₁ → fun x₂ →
        f x₁ (g x₂)

We have:

comp rev (map f) = comp (map f) rev
comp rev (filter p) = comp (filter p) rev
comp rev (tabulate f) = rev_tabulate f

Here, rev_tabulate tabulates in reverse ordering. This function has
properties quite analogous to tabulate:

comp (map f) (rev_tabulate g) = rev_tabulate (comp₂ f g)
comp (foldl f a) (rev_tabulate g) = rev_loop (comp₂ f g) a

Extension (3): Dependencies on the Index

- Correctness is proven by induction on the lengths of occurring lists.
- Similar composition results also hold for transformations which take
the current indices into account:

mapᵢ = fun i → fun f → fun l → match l with [] → []
|   | x :: xs → f i x :: mapᵢ (i + 1) f xs
map₀ = mapᵢ 0