We have:

\[
\begin{align*}
\text{comp rev (map } f\text{)} &= \text{comp (map } f\text{) rev} \\
\text{comp rev (filter } p\text{)} &= \text{comp (filter } p\text{) rev} \\
\text{comp rev (tabulate } f\text{)} &= \text{rev_tabulate } f
\end{align*}
\]

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

\[
\begin{align*}
\text{comp (map } f\text{) (rev_tabulate } g\text{)} &= \text{rev_tabulate (comp}_2 f g) \\
\text{comp (foldl } f a\text{) (rev_tabulate } g\text{)} &= \text{rev_loop (comp}_2 f g) a
\end{align*}
\]
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:

\[
\text{mapi'} = \text{fun } i \rightarrow \text{fun } f \rightarrow \text{fun } l \rightarrow \text{match } l \text{ with } [\ ] \rightarrow [\ ] \\
| x :: xs \rightarrow f i x) :: \text{mapi'} (i + 1) f xs
\]

\[
\text{mapi} = \text{mapi'} 0
\]
Analogously, there is index-dependent accumulation:

\[
\text{foldli'} = \text{fun } i \to \text{fun } f \to \text{fun } a \to \text{fun } l \to \\
\quad \text{match } l \text{ with } [] \to a \\
\quad | \ x :: xs \to \text{foldli'} (i + 1) f (f i a x) xs
\]

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

For composition, we must take care that always the same indices are used. This is achieved by:
\[
\text{compi} = \text{fun } f \rightarrow \text{fun } g \rightarrow \text{fun } i \rightarrow \text{fun } x \rightarrow f \ i \ (g \ i \ x)
\]

\[
\text{compi}_1 = \text{fun } f \rightarrow \text{fun } g \rightarrow \text{fun } i \rightarrow \text{fun } x_1 \rightarrow \text{fun } x_2 \rightarrow f \ i \ (g \ i \ x_1) \ x_2
\]

\[
\text{compi}_2 = \text{fun } f \rightarrow \text{fun } g \rightarrow \text{fun } i \rightarrow \text{fun } x_1 \rightarrow \text{fun } x_2 \rightarrow f \ i \ x_1 \ (g \ i \ x_2)
\]

\[
\text{cmp}_1 = \text{fun } f \rightarrow \text{fun } g \rightarrow \text{fun } i \rightarrow \text{fun } x_1 \rightarrow \text{fun } x_2 \rightarrow f \ i \ x_1 \ (g \ x_2)
\]

\[
\text{cmp}_2 = \text{fun } f \rightarrow \text{fun } g \rightarrow \text{fun } i \rightarrow \text{fun } x_1 \rightarrow \text{fun } x_2 \rightarrow f \ x_1 \ (g \ i \ x_2)
\]
Then:

\[
\begin{align*}
\text{comp} (\text{mapi } f) (\text{map } g) &= \text{mapi} (\text{comp}_2 f g) \\
\text{comp} (\text{map } f) (\text{mapi } g) &= \text{mapi} (\text{comp } f g) \\
\text{comp} (\text{mapi } f) (\text{mapi } g) &= \text{mapi} (\text{compi } f g) \\
\text{comp} (\text{foldli } f \ a) (\text{map } g) &= \text{foldli} (\text{cmp}_1 f g) a \\
\text{comp} (\text{foldl } f \ a) (\text{mapi } g) &= \text{foldli} (\text{cmp}_2 f g) a \\
\text{comp} (\text{foldli } f \ a) (\text{mapi } g) &= \text{foldli} (\text{compi}_2 f g) a \\
\text{comp} (\text{foldli } f \ a) (\text{tabulate } g) &= \text{let } h = \text{ fun } a \rightarrow \text{ fun } i \rightarrow f \ i \ a \ (g \ i) \text{ in loop } h \ a
\end{align*}
\]
Discussion:

- **Warning:** index-dependent transformations may not commute with `rev` or `filter`.

- All our rules can only be applied if the functions `id`, `map`, `mapi`, `foldl`, `foldli`, `filter`, `rev`, `tabulate`, `rev_tabulate`, `loop`, `rev_loop`, ... are provided by a standard library: Only then the algebraic properties can be guaranteed !!!

- Similar simplification rules can be derived for any kind of tree-like data-structure `tree α`.

- These also provide operations `map`, `mapi` and `foldl`, `foldli` with corresponding rules.

- Further opportunities are opened up by functions `to_list` and `from_list` ...
Example

\[
\text{type } \text{tree } \alpha = \text{Leaf} \mid \text{Node } \alpha (\text{tree } \alpha) (\text{tree } \alpha)
\]

\[
\text{map} = \text{fun } f \to \text{fun } t \to \text{match } t \text{ with } \text{Leaf} \to \text{Leaf} \\
| \text{Node } x \ l \ r \to \text{let } l' = \text{map } f \ l \ \ r' = \text{map } f \ r \\
\text{in } \text{Node } (f \ x) \ l' \ r'
\]

\[
\text{foldl} = \text{fun } f \to \text{fun } a \to \text{fun } t \to \text{match } t \text{ with } \text{Leaf} \to a \\
| \text{Node } x \ l \ r \to \text{let } a' = \text{foldl } f \ a \ l \\
\text{in } \text{foldl } f \ (f \ a' \ x) \ r
\]
\[
\begin{align*}
\text{to\_list}' &= \text{fun } a \to \text{fun } t \to \text{match } t \text{ with } \text{Leaf } \to a \\
&\quad \mid \text{Node } x \ t_1 \ t_2 \to \text{let } a' = \text{to\_list}' \ a \ t_2 \\
&\quad \quad \quad \text{in } \text{to\_list}' (x :: a') \ t_1 \\
\text{to\_list} &= \text{to\_list}' \ [] \\
\text{from\_list} &= \text{fun } l \to \text{match } l \\
&\quad \text{with } [] \to \text{Leaf} \\
&\quad \mid x :: xs \to \text{Node } x \ \text{Leaf} \ (\text{from\_list} \ xs)
\end{align*}
\]
Warning:

Not every natural equation is valid:

\[
\begin{align*}
\text{comp to_list from_list} & = \text{id} \\
\text{comp from_list to_list} & \neq \text{id} \\
\text{comp to_list (map } f) & = \text{comp (map } f) \text{ to_list} \\
\text{comp from_list (map } f) & = \text{comp (map } f) \text{ from_list} \\
\text{comp (foldl } f a) \text{ to_list} & = \text{foldl } f a \\
\text{comp (foldl } f a) \text{ from_list} & = \text{foldl } f a
\end{align*}
\]
In this case, there is even a \( \text{rev} \):

\[
\text{rev} \quad = \quad \text{fun} \; t \to \\
\quad \text{match} \; t \; \text{with} \; \text{Leaf} \to \; \text{Leaf} \\
\quad \mid \; \text{Node} \; x \; t_1 \; t_2 \to \; \text{let} \; s_1 = \; \text{rev} \; t_1 \\
\quad \; s_2 = \; \text{rev} \; t_2 \\
\quad \; \text{in} \; \text{Node} \; x \; s_2 \; s_1
\]

\[
\text{comp to_list rev} \quad = \quad \text{comp rev to_list} \\
\text{comp from_list rev} \not= \quad \text{comp rev from_list}
\]
4.6 CBN vs. CBV: Strictness Analysis

Problem:

- Programming languages such as Haskell evaluate expressions for \texttt{let}-defined variables and actual parameters not before their values are accessed.
- This allows for an elegant treatment of (possibly) infinite lists of which only small initial segments are required for computing the result \(\text{-)}\)
- Delaying evaluation by default incures, though, a non-trivial overhead \(...\)
Example

\[
\text{from} = \text{fun } n \rightarrow n :: \text{from} (n + 1)
\]

\[
\text{take} = \text{fun } k \rightarrow \text{fun } s \rightarrow \begin{align*}
\text{if } k \leq 0 \text{ then } & \left[ \right] \\
\text{else } \text{match } s \text{ with } & \left[ \right] \rightarrow \left[ \right] \\
& \text{| } x :: xs \rightarrow x :: \text{take} (k - 1) \: xs
\end{align*}
\]
Then CBN yields:

\[ \text{take } 5 \ (\text{from } 0) = [0, 1, 2, 3, 4] \]

— whereas evaluation with CBV does not terminate !!!
Then CBN yields:

\[
\text{take } 5 \text{ (from } 0 \text{)} = [0, 1, 2, 3, 4]
\]

— whereas evaluation with CBV does not terminate !!!

On the other hand, for CBN, tail-recursive functions may require non-constant space ????

\[
fac2 = \text{fun } x \rightarrow \text{fun } a \rightarrow \begin{cases} 
    a & \text{if } x \leq 0 \\
    \text{fac2} (x - 1) \ (a \cdot x) & \text{else}
\end{cases}
\]
Discussion:

- The multiplications are collected in the accumulating parameter through nested closures.
- Only when the value of a call $\text{fac2} \ x \ 1$ is accessed, this dynamic data structure is evaluated.
- Instead, the accumulating parameter should have been passed directly by-value !!!
- This is the goal of the following optimization ...
Simplification:

- At first, we rule out data structures, higher-order functions, and local function definitions.
- We introduce an unary operator `#` which forces the evaluation of a variable.
- Goal of the transformation is to place `#` at as many places as possible ...
Simplification:

- At first, we rule out data structures, higher-order functions, and local function definitions.
- We introduce an unary operator `#` which forces the evaluation of a variable.
- Goal of the transformation is to place `#` at as many places as possible ...

\[
\begin{align*}
e & ::= c \mid x \mid e_1 \square_2 e_2 \mid \square_1 e \mid f \ e_1 \ldots \ e_k \mid \text{if } e_0 \text{ then } e_1 \text{ else } e_2 \\
& \quad \mid \text{let } r_1 = e_1 \text{ in } e \\
\end{align*}
\]

\[
\begin{align*}
r & ::= x \mid \#x \\
d & ::= f \ x_1 \ldots \ x_k = e \\
p & ::= \text{letrec and } d_1 \ldots \text{ and } d_n \text{ in } e
\end{align*}
\]
Idea:

- Describe a $k$-ary function

$$f : \text{int} \rightarrow \ldots \rightarrow \text{int}$$

by a function

$$[f]^\# : \mathbb{B} \rightarrow \ldots \rightarrow \mathbb{B}$$

- 0 means: evaluation does definitely not terminate.
- 1 means: evaluation may terminate.
- $[f]^\# \ 0 = 0$ means: If the function call returns a value, then the evaluation of the argument must have terminated and returned a value.

$\implies f$ is strict.
Idea (cont.):

- We determine the abstract semantics of all functions  
- For that, we put up a system of equations ...

Auxiliary Function:

\[
\begin{align*}
[e]^\# & : (\text{Vars} \to \mathbb{B}) \to \mathbb{B} \\
[c]^\# \rho & = 1 \\
[x]^\# \rho & = \rho x \\
[e_1 \quad e_2]^\# \rho & = [e_1]^\# \rho \wedge [e_2]^\# \rho \\
[\text{if } e_0 \text{ then } e_1 \text{ else } e_2]^\# \rho & = [e_0]^\# \rho \wedge ([e_1]^\# \rho \vee [e_2]^\# \rho) \\
[f \ e_1 \ldots \ e_k]^\# \rho & = [f]^\# ([e_1]^\# \rho) \ldots ([e_k]^\# \rho) \\
\ldots
\end{align*}
\]
\[
\begin{align*}
\text{let } x_1 = e_1 \text{ in } e \# \rho &= [e] \# (\rho \oplus \{ x_1 \mapsto [e_1] \# \rho \}) \\
\text{let } #x_1 = e_1 \text{ in } e \# \rho &= ([e_1] \# \rho) \land ([e] \# (\rho \oplus \{ x_1 \mapsto 1 \}))
\end{align*}
\]

**System of Equations:**

\[
[f_i] \# b_1 \ldots b_k = [e_i] \# \{ x_j \mapsto b_j \mid j = 1, \ldots, k \}, \quad i = 1, \ldots, n, \ b_1, \ldots, b_k \in \mathbb{B}
\]

- The unknowns of the system of equations are the functions \([f_i] \#\) or the individual entries \([f_i] \# b_1 \ldots b_k\) in the value table.
- All right-hand sides are monotonic!
- Consequently, there is a least solution \(:-)\)
- The complete lattice \(\mathbb{B} \to \ldots \to \mathbb{B}\) has height \(\mathcal{O}(2^k)\) \(:-(
\)
Example:

For \( \text{fac2} \), we obtain:

\[
\begin{align*}
\llbracket \text{fac2} \rrbracket \# b_1 b_2 &= b_1 \land (b_2 \lor \\
& \quad \llbracket \text{fac2} \rrbracket \# b_1 (b_1 \land b_2))
\end{align*}
\]

Fixpoint iteration yields:

\[
\begin{array}{|c|c|}
\hline
0 & \textbf{fun} x \to \textbf{fun} a \to 0 \\
1 & \textbf{fun} x \to \textbf{fun} a \to x \land a \\
2 & \textbf{fun} x \to \textbf{fun} a \to x \land a \\
\hline
\end{array}
\]
We conclude:

- The function \( \text{fac2} \) is strict in both arguments, i.e., if evaluation terminates, then also the evaluation of its arguments.
- Accordingly, we transform:

\[
\text{fac2} = \quad \text{fun} \ x \ \rightarrow \ \text{fun} \ a \ \rightarrow \quad \text{if} \ x \leq 0 \ \text{then} \ a \\
\text{else let} \quad \# \ x' = x - 1 \\
\quad \# \ a' = x \cdot a \\
\text{in} \quad \text{fac2} \ x' \ a'
\]
Correctness of the Analysis:

- The system of equations is an abstract denotational semantics.
- The denotational semantics characterizes the meaning of functions as least solution of the corresponding equations for the concrete semantics.
- For values, the denotational semantics relies on the complete partial ordering $\mathbb{Z}_\bot$.
- For complete partial orderings, Kleene’s fixpoint theorem is applicable $:-)$
- As description relation $\Delta$ we use:

\[
\bot \Delta 0 \quad \text{and} \quad z \Delta 1 \quad \text{for} \quad z \in \mathbb{Z}
\]
Extension: Data Structures

- Functions may vary in the parts which they require from a data structure ...

\[
\text{hd} = \text{fun } l \rightarrow \text{match } l \text{ with } x :: xs \rightarrow x
\]

- \text{hd} only accesses the first element of a list.
- \text{length} only accesses the backbone of its argument.
- \text{rev} forces the evaluation of the complete argument — given that the result is required completely ...