We have:

$$\begin{array}{lcl}
\textsf{comp rev}\ (\textsf{map}\ f) & = & \textsf{comp}\ (\textsf{map}\ f)\ \textsf{rev} \\
\textsf{comp rev}\ (\textsf{filter}\ p) & = & \textsf{comp}\ (\textsf{filter}\ p)\ \textsf{rev} \\
\textsf{comp rev}\ (\textsf{tabulate}\ f) & = & \textsf{rev_tabulate}\ f
\end{array}$$

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

$$\begin{array}{lcl}
\textsf{comp}\ (\textsf{map}\ f)\ (\textsf{rev_tabulate}\ g) & = & \textsf{rev_tabulate}\ (\textsf{comp}_2\ f\ g) \\
\textsf{comp}\ (\textsf{foldl}\ f\ a)\ (\textsf{rev_tabulate}\ g) & = & \textsf{rev_loop}\ (\textsf{comp}_2\ f\ g)\ a
\end{array}$$

Extension (3): Dependencies on the Index

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

$$
\begin{array}{lcl}
\textsf{mapi}' & = & \textbf{fun}\ i \ \to\ \ \textbf{fun}\ f\ \to\ \textbf{fun}\ l\ \to\ \textbf{match}\ l\ \textbf{with}\ [\,]\ \to\ [\,] \\
 & & \qquad\qquad\qquad \mid\ \ x :: xs\ \to\ f\ i\ x) :: \textsf{mapi}'\ (i+1)\ f\ xs \\
\textsf{mapi} & = & \textsf{mapi}'\ 0
\end{array}
$$

Analogously, there is index-dependent accumulation:

$$\begin{array}{lll} \mathsf{foldli}' & = & \textbf{fun } i \to \textbf{fun } f \to \textbf{fun } a \to \textbf{fun } l \to \\ & & \quad \textbf{match } l \textbf{ with } [\,] \to a \\ & & \quad \mid x :: xs \to \mathsf{foldli}' \; (i+1) \; f \; (f\; i\; a\; x) \; xs \\ \mathsf{foldli} & = & \mathsf{foldli}' \; 0 \end{array}$$

For composition, we must take care that always the same indices are used.
This is achieved by:

$$\textsf{compi} = \textbf{fun}\ f \to \textbf{fun}\ g \to \textbf{fun}\ i \to \textbf{fun}\ x \to f\ i\ (g\ i\ x)$$

$$\begin{aligned}\textsf{compi}_1 = {} & \textbf{fun}\ f \to \textbf{fun}\ g \to \textbf{fun}\ i \to \textbf{fun}\ x_1 \to \textbf{fun}\ x_2 \to \\ & f\ i\ (g\ i\ x_1)\ x_2\end{aligned}$$

$$\begin{aligned}\textsf{compi}_2 = {} & \textbf{fun}\ f \to \textbf{fun}\ g \to \textbf{fun}\ i \to \textbf{fun}\ x_1 \to \textbf{fun}\ x_2 \to \\ & f\ i\ x_1\ (g\ i\ x_2)\end{aligned}$$

$$\begin{aligned}\textsf{cmp}_1 = {} & \textbf{fun}\ f \to \textbf{fun}\ g \to \textbf{fun}\ i \to \textbf{fun}\ x_1 \to \textbf{fun}\ x_2 \to \\ & f\ i\ x_1\ (g\ x_2)\end{aligned}$$

$$\begin{aligned}\textsf{cmp}_2 = {} & \textbf{fun}\ f \to \textbf{fun}\ g \to \textbf{fun}\ i \to \textbf{fun}\ x_1 \to \textbf{fun}\ x_2 \to \\ & f\ x_1\ (g\ i\ x_2)\end{aligned}$$

Then:

$$
\begin{array}{rcl}
\mathsf{comp}\,(\mathsf{mapi}\ f)\ (\mathsf{map}\ g) & = & \mathsf{mapi}\,(\mathsf{comp}_2\ f\ g) \\
\mathsf{comp}\,(\mathsf{map}\ f)\ (\mathsf{mapi}\ g) & = & \mathsf{mapi}\,(\mathsf{comp}\ f\ g) \\
\mathsf{comp}\,(\mathsf{mapi}\ f)\ (\mathsf{mapi}\ g) & = & \mathsf{mapi}\,(\mathsf{compi}\ f\ g) \\
\mathsf{comp}\,(\mathsf{foldli}\ f\ a)\ (\mathsf{map}\ g) & = & \mathsf{foldli}\,(\mathsf{cmp}_1\ f\ g)\ a \\
\mathsf{comp}\,(\mathsf{foldl}\ f\ a)\ (\mathsf{mapi}\ g) & = & \mathsf{foldli}\,(\mathsf{cmp}_2\ f\ g)\ a \\
\mathsf{comp}\,(\mathsf{foldli}\ f\ a)\ (\mathsf{mapi}\ g) & = & \mathsf{foldli}\,(\mathsf{compi}_2\ f\ g)\ a \\
\mathsf{comp}\,(\mathsf{foldli}\ f\ a)\ (\mathsf{tabulate}\ g) & = & \mathbf{let}\ h = \ \mathbf{fun}\ a\ \to\ \mathbf{fun}\ i\ \to \\
 & & \qquad\qquad f\ i\ a\ (g\, i) \\
 & & \mathbf{in}\ \ \mathsf{loop}\ h\ a
\end{array}
$$

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

$$
\begin{array}{lcl}
\textbf{type}\ \mathsf{tree}\ \alpha & = & \mathsf{Leaf} \mid \mathsf{Node}\ \alpha\ (\mathsf{tree}\ \alpha)\ (\mathsf{tree}\ \alpha) \\
\mathsf{map} & = & \textbf{fun}\ f \ \rightarrow\ \textbf{fun}\ t\ \rightarrow\ \textbf{match}\ t\ \textbf{with}\ \mathsf{Leaf}\ \rightarrow\ \mathsf{Leaf} \\
& & \qquad\qquad \mid\ \mathsf{Node}\ x\ l\ r\ \rightarrow\ \textbf{let}\ \ l' = \mathsf{map}\ f\ l \\
& & \qquad\qquad\qquad\qquad\qquad\qquad\ \ r' = \mathsf{map}\ f\ r \\
& & \qquad\qquad\qquad\qquad\qquad \textbf{in}\ \ \mathsf{Node}\ (f\ x)\ l'\ r'
\end{array}
$$

$$
\begin{array}{lcl}
\mathsf{foldl} & = & \textbf{fun}\ f\ \rightarrow\ \textbf{fun}\ a\ \rightarrow\ \textbf{fun}\ t\ \rightarrow\ \textbf{match}\ t\ \textbf{with}\ \mathsf{Leaf}\ \rightarrow\ a \\
& & \qquad\qquad \mid \mathsf{Node}\ x\ l\ r\ \rightarrow\ \textbf{let}\ a' = \mathsf{foldl}\ f\ a\ l \\
& & \qquad\qquad\qquad\qquad\qquad \textbf{in}\ \mathsf{foldl}\ f\ (f\ a'\ x)\ r
\end{array}
$$

$$
\begin{array}{lcl}
\text{to\_list}' & = & \textbf{fun}\, a \;\rightarrow\; \textbf{fun}\, t \;\rightarrow\; \textbf{match}\, t \;\textbf{with}\; \mathsf{Leaf} \;\rightarrow\; a \\
 & & \mid\; \mathsf{Node}\; x\; t_1\; t_2 \;\rightarrow\; \textbf{let}\;\; a' \;=\; \text{to\_list}'\; a\; t_2 \\
 & & \qquad\qquad\qquad\qquad\quad \textbf{in}\;\; \text{to\_list}'\; (x :: a')\; t_1 \\
\text{to\_list} & = & \text{to\_list}'\; [\,]
\end{array}
$$

$$
\begin{array}{lcl}
\text{from\_list} & = & \textbf{fun}\, l \;\rightarrow\; \textbf{match}\; l \\
 & & \textbf{with}\; [\,] \;\rightarrow\; \mathsf{Leaf} \\
 & & \mid\;\; x :: xs \;\rightarrow\; \mathsf{Node}\; x\; \mathsf{Leaf}\; (\text{from\_list}\; xs)
\end{array}
$$

Warning:

Not every natural equation is valid:

$$
\begin{array}{rcl}
\text{comp to\_list from\_list} & = & \text{id} \\
\text{comp from\_list to\_list} & \neq & \text{id} \\
\text{comp to\_list } (\text{map } f) & = & \text{comp } (\text{map } f) \text{ to\_list} \\
\text{comp from\_list } (\text{map } f) & = & \text{comp } (\text{map } f) \text{ from\_list} \\
\text{comp } (\text{foldl } f\ a) \text{ to\_list} & = & \text{foldl } f\ a \\
\text{comp } (\text{foldl } f\ a) \text{ from\_list} & = & \text{foldl } f\ a
\end{array}
$$

In this case, there is even a rev:

$$
\begin{array}{lcl}
\mathsf{rev} & = & \mathbf{fun}\, t \;\to \\
& & \mathbf{match}\; t\; \mathbf{with}\; \mathsf{Leaf} \;\to\; \mathsf{Leaf} \\
& & \mid\; \mathsf{Node}\; x\; t_1\; t_2 \;\to\; \mathbf{let}\;\; s_1 \;=\; \mathsf{rev}\; t_1 \\
& & \qquad\qquad\qquad\qquad\qquad\;\; s_2 \;=\; \mathsf{rev}\; t_2 \\
& & \qquad\qquad\qquad\qquad\; \mathbf{in}\;\; \mathsf{Node}\; x\; s_2\; s_1
\end{array}
$$

$$
\begin{array}{lcl}
\mathsf{comp\ to\_list\ rev} & = & \mathsf{comp\ rev\ to\_list} \\
\mathsf{comp\ from\_list\ rev} & \neq & \mathsf{comp\ rev\ from\_list}
\end{array}
$$

4.6 CBN vs. CBV: Strictness Analysis

Problem:

- Programming languages such as Haskell evaluate expressions for **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 :-)
- Delaying evaluation by default incures, though, a non-trivial overhead ...

Example

$$\mathsf{from} \;=\; \mathbf{fun}\ n \;\rightarrow\; n :: \mathsf{from}\ (n+1)$$

$$\begin{array}{lcl} \mathsf{take} & = & \mathbf{fun}\ k \;\rightarrow\; \mathbf{fun}\ s \;\rightarrow\; \mathbf{if}\ k \leq 0\ \mathbf{then}\ [\,] \\ & & \qquad\qquad\qquad\qquad\quad \mathbf{else}\ \ \mathbf{match}\ s\ \mathbf{with}\ [\,] \;\rightarrow\; [\,] \\ & & \qquad\qquad\qquad\qquad\qquad\qquad \mid\ x :: xs \;\rightarrow\; x :: \mathsf{take}\ (k-1)\ xs \end{array}$$

Then CBN yields:

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

— whereas evaluation with CBV does not terminate !!!

Then CBN yields:

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

— whereas evaluation with CBV does not terminate !!!

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

$$\begin{array}{lllll} \mathsf{fac2} & = & \mathbf{fun}\ x \ \rightarrow\ \mathbf{fun}\ a\ \rightarrow & \mathbf{if}\ x \leq 0\ \mathbf{then}\ a \\ & & & \mathbf{else}\ \ \mathsf{fac2}\ (x-1)\ (a \cdot x) \end{array}$$

Discussion:

- The multiplications are collected in the accumulating parameter through nested closures.
- Only when the value of a call 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 ...

$$
\begin{array}{lll}
e & ::= & c \mid x \mid e_1 \,\Box_2\, e_2 \mid \Box_1\, e \mid f\ e_1\ \ldots\ e_k \mid \textbf{if}\ e_0\ \textbf{then}\ e_1\ \textbf{else}\ e_2 \\
 & & \mid \textbf{let}\ r_1 = e_1\ \textbf{in}\ e \\
r & ::= & x \mid \#x \\
d & ::= & f\ x_1\ \ldots\ x_k = e \\
p & ::= & \textbf{letrec and}\ d_1\ \ldots\ \textbf{and}\ d_n\ \textbf{in}\ e
\end{array}
$$

Idea:

- Describe a k-ary function

$$f : \mathbf{int} \to \ldots \to \mathbf{int}$$

by a function

$$[\![f]\!]^\sharp : \mathbb{B} \to \ldots \to \mathbb{B}$$

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

$\Longrightarrow$ f is strict.

Idea (cont.):

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

Auxiliary Function:

$$
\begin{array}{lcl}
[\![e]\!]^\sharp & : & (\mathit{Vars} \to \mathbb{B}) \to \mathbb{B} \\
[\![c]\!]^\sharp\ \rho & = & 1 \\
[\![x]\!]^\sharp\ \rho & = & \rho\, x \\
[\![\Box_1\ e]\!]^\sharp\ \rho & = & [\![e]\!]^\sharp\ \rho \\
[\![e_1\ \Box_2\ e_2]\!]^\sharp\ \rho & = & [\![e_1]\!]^\sharp\ \rho \wedge [\![e_2]\!]^\sharp\ \rho \\
[\![\mathbf{if}\ e_0\ \mathbf{then}\ e_1\ \mathbf{else}\ e_2]\!]^\sharp\ \rho & = & [\![e_0]\!]^\sharp\ \rho \wedge ([\![e_1]\!]^\sharp\ \rho \vee [\![e_2]\!]^\sharp\ \rho) \\
[\![f\ e_1\ \ldots\ e_k]\!]^\sharp\ \rho & = & [\![f]\!]^\sharp\ ([\![e_1]\!]^\sharp\ \rho)\ \ldots\ ([\![e_k]\!]^\sharp\ \rho) \\
\ldots & &
\end{array}
$$

$$\begin{array}{lll} [\![\mathbf{let}\ x_1 = e_1\ \mathbf{in}\ e]\!]^\sharp\ \rho & = & [\![e]\!]^\sharp\ (\rho \oplus \{x_1 \mapsto [\![e_1]\!]^\sharp\ \rho\}) \\ [\![\mathbf{let}\ \#x_1 = e_1\ \mathbf{in}\ e]\!]^\sharp\ \rho & = & ([\![e_1]\!]^\sharp\ \rho) \wedge ([\![e]\!]^\sharp\ (\rho \oplus \{x_1 \mapsto 1\})) \end{array}$$

System of Equations:

$$[\![f_i]\!]^\sharp b_1\ \ldots\ b_k = [\![e_i]\!]^\sharp\ \{x_j \mapsto b_j \mid j = 1, \ldots, k\}, \qquad i = 1, \ldots, n, b_1, \ldots, b_k \in \mathbb{B}$$

- The unkowns of the system of equations are the functions $[\![f_i]\!]^\sharp$ or the individual entries $[\![f_i]\!]^\sharp b_1\ \ldots\ b_k$ 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 fac2, we obtain:

$$\begin{array}{lll} [\![\mathsf{fac2}]\!]^\sharp\ b_1\ b_2 & = & b_1 \wedge (b_2 \vee \\ & & [\![\mathsf{fac2}]\!]^\sharp\ b_1\ (b_1 \wedge b_2)) \end{array}$$

Fixpoint iteration yields:

0	$\textbf{fun}\, x \to \textbf{fun}\, a \to 0$
1	$\textbf{fun}\, x \to \textbf{fun}\, a \to x \wedge a$
2	$\textbf{fun}\, x \to \textbf{fun}\, a \to x \wedge a$

We conclude:

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

$$
\begin{array}{lllllll}
\mathsf{fac2} & = & \textbf{fun}\ x \to \textbf{fun}\ a \to & \textbf{if}\ x \leq 0\ \textbf{then}\ a & & & \\
 & & & \textbf{else}\ \ \textbf{let} & \#\, x' & = & x - 1 \\
 & & & & \#\, a' & = & x \cdot a \\
 & & & \textbf{in}\ \ \mathsf{fac2}\ x'\ a' & & &
\end{array}
$$

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 $\mathbb{Z}_\perp$.
- For complete partial orderings, Kleene's fixpoint theorem is applicable :-)
- As description relation Δ we use:

$$\perp \,\Delta\, 0 \quad \text{and} \quad z \,\Delta\, 1 \quad \text{for } z \in \mathbb{Z}$$

Extension: Data Structures

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

$$\mathsf{hd} \quad = \quad \mathbf{fun}\ l \ \rightarrow \ \mathbf{match}\ l\ \mathbf{with}\ x :: xs \ \rightarrow \ x$$

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