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:

\[
\begin{align*}
\text{map}' &= \text{fun } i \rightarrow \text{fun } f \rightarrow \text{fun } l \rightarrow \text{match } l \text{ with } [ ] \rightarrow [ ] \\
&\quad \mid x :: xs \rightarrow f i x : \text{map}' (i + 1) f xs \\
\text{map} &= \text{map}' 0
\end{align*}
\]

We have:

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

Analogously, there is index-dependent accumulation:

\[
\begin{align*}
\text{foldl}' &= \text{fun } i \rightarrow \text{fun } f \rightarrow \text{fun } a \rightarrow \text{fun } l \rightarrow \\
&\quad \text{match } l \text{ with } [ ] \rightarrow a \\
&\quad \mid x :: xs \rightarrow \text{foldl}' (i + 1) f (f i a x) xs \\
\text{foldl} &= \text{foldl}' 0
\end{align*}
\]

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

\texttt{compi}_1 = \texttt{fun} f \rightarrow \texttt{fun} g \rightarrow \texttt{fun} i \rightarrow \texttt{fun} x_1 \rightarrow \texttt{fun} x_2 \rightarrow f i (g i x_1) x_2

\texttt{compi}_2 = \texttt{fun} f \rightarrow \texttt{fun} g \rightarrow \texttt{fun} i \rightarrow \texttt{fun} x_1 \rightarrow \texttt{fun} x_2 \rightarrow f i x_1 (g i x_2)

\texttt{cmp}_1 = \texttt{fun} f \rightarrow \texttt{fun} g \rightarrow \texttt{fun} i \rightarrow \texttt{fun} x_1 \rightarrow \texttt{fun} x_2 \rightarrow f i x_1 (g x_2)

\texttt{cmp}_2 = \texttt{fun} f \rightarrow \texttt{fun} g \rightarrow \texttt{fun} i \rightarrow \texttt{fun} x_1 \rightarrow \texttt{fun} x_2 \rightarrow f x_1 (g i x_2)

\textbf{Discussion:}

- \textit{Warning:} index-dependent transformations may not commute with \texttt{rev} or filters.
- All our rules can only be applied if the functions \texttt{id}, \texttt{map}, \texttt{mapi}, \texttt{foldl}, \texttt{foldli}, \texttt{filter}, \texttt{rev}, \texttt{tabulate}, \texttt{rev\_tabulate}, \texttt{loop}, \texttt{rev\_loop}, \texttt{...} 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 \texttt{tree a}.
- These also provide operations \texttt{map}, \texttt{mapi} and \texttt{foldl}, \texttt{foldli} with corresponding rules.
- Further opportunities are opened up by functions \texttt{to\_list} and \texttt{from\_list} ...
Example

\[
\begin{align*}
\text{type } \text{tree } \alpha & \ = \ \text{Leaf} \mid \text{Node } \alpha \ (\text{tree } \alpha) \ (\text{tree } \alpha) \\
\text{map} & \ = \ \text{fun } f \to \ \text{fun } t \to \ \text{match } t \ \text{with } \text{Leaf} \to \ \text{Leaf} \\
& \quad \mid \ \text{Node } x \ l \ r \to \ \text{let } l' = \ \text{map } f \ l \\
& \quad \quad \ r' = \ \text{map } f \ r \\
& \quad \quad \ \text{in } \ \text{Node } (f \ x) \ l' \ r' \\
\text{foldl} & \ = \ \text{fun } f \to \ \text{fun } a \to \ \text{fun } t \to \ \text{match } t \ \text{with } \text{Leaf} \to \ a \\
& \quad \mid \ \text{Node } x \ l \ r \to \ \text{let } a' = \ \text{foldl } f \ a \ l \\
& \quad \quad \ \text{in } \ \text{foldl } f \ (f \ a' \ x) \ r \\
\end{align*}
\]

Warning:

Not every natural equation is valid:

\[
\begin{align*}
\text{comp } \text{to_list from_list} & \ = \ \text{id} \\
\text{comp from_list to_list} & \ \neq \ \text{id} \\
\text{comp to_list } (\text{map } f) & \ = \ \text{comp } (\text{map } f) \ \text{to_list} \\
\text{comp from_list } (\text{map } f) & \ = \ \text{comp } (\text{map } f) \ \text{from_list} \\
\text{comp } (\text{foldl } f \ a) \ \text{to_list} & \ = \ \text{foldl } f \ a \\
\text{comp } (\text{foldl } f \ a) \ \text{from_list} & \ = \ \text{foldl } f \ a \\
\end{align*}
\]

In this case, there is even a \text{rev}:

\[
\begin{align*}
\text{rev} & \ = \ \text{fun } t \to \\
& \quad \ \text{match } t \ \text{with } \text{Leaf} \to \ \text{Leaf} \\
& \quad \mid \ \text{Node } x \ t_1 \ t_2 \to \ \text{let } s_1 = \ \text{rev } t_1 \\
& \quad \quad \ s_2 = \ \text{rev } t_2 \\
& \quad \quad \ \text{in } \ \text{Node } x \ s_2 \ s_1 \\
\text{comp } \text{to_list rev} & \ = \ \text{comp rev to_list} \\
\text{comp from_list rev} & \ \neq \ \text{comp rev from_list}
\end{align*}
\]
In this case, there is even a `rev`:

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

\[
\text{comp to_list rev} = \text{comp rev to_list} \\
\text{comp from_list rev} \neq \text{comp rev from_list}
\]

### 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 incurs, though, a non-trivial overhead ...

Then CBN yields:

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

- whereas evaluation with CBV does not terminate !!!

**Example**

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

\[
\text{take } = \text{fun } k \rightarrow \text{fun } s \rightarrow \text{if } k \leq 0 \text{ then } [] \\
\quad \text{else } \text{match } s \text{ with } [] \rightarrow [] \\
\quad \quad | x :: xs \rightarrow x :: \text{take } (k - 1) xs
\]
Example

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

\[ \text{take} = \text{fun } k \rightarrow \text{fun } s \rightarrow \begin{cases} \text{if } k \leq 0 \text{ then } [] \\ \text{else match } s \text{ with } [] \rightarrow [] \\ \quad | x :: xs \rightarrow x :: \text{take} (k - 1) xs \end{cases} \]

Then CBN yields:

\[ \text{take} 5 (\text{from } 0) = [0, 1, 2, 3, 4] \]
— whereas evaluation with CBV does not terminate !!!

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

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

\[ \text{fac} = \text{fun } x \rightarrow \text{fac2} x 1 \]

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 ...
Then CBN yields:

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

whereas evaluation with CBV does not terminate !!!

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

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

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 ...

\[
e ::= e \mid x \mid e_1 \boxdot e_2 \mid \text{\#} e \mid f \ e_1 \ldots e_k \mid \text{if } e_0 \text{ then } e_1 \text{ else } e_2 \mid \text{let } r_1 = e_1 \text{ in } e
\]

\[
r ::= x \mid \# x
\]

\[
d ::= f \ x_1 \ldots x_k = e
\]

\[
p ::= \text{letrec and } d_1 \ldots \text{ and } d_n \text{ in } e
\]
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 ...

\[
e ::= e_1 \ | \ x \ | \ e_1 \sqcup e_2 \ | \ \Box e \ | \ f \ e_1 \ldots e_k \ | \ \text{if } e_0 \ \text{then } e_1 \ \text{else } e_2 \ | \ \text{let } r_1 = e_1 \ \text{in } e
\]

\[
r ::= x \ | \ \#x
\]

\[
d ::= f \ x_1 \ldots x_k = e
\]

\[
p ::= \text{letrec and } d_1 \ldots \text{and } d_n \ \text{in } e
\]

Idea:

- Describe a \(k\)-ary function
  \[
f : \text{int} \to \ldots \to \text{int}
\]
  by a function
  \[
  [f]^2 : \mathbb{B} \to \ldots \to \mathbb{B}
  \]
- \(0\) means: evaluation does definitely not terminate.
- \(1\) means: evaluation may terminate.
- \([f]^2 0 = 0\) means: If the function call returns a value, then the evaluation of the argument must have terminated and returned a value.
  \[
  \implies f \text{ is strict.}
  \]

\[
[\text{let } x_1 = e_1 \ \text{in } e]^2 \rho = [e_1]^2 (\rho \oplus \{ x_1 \mapsto [e_1]^2 \rho \})
\]

\[
[\text{let } \#x_1 = e_1 \ \text{in } e]^2 \rho = ([e_1]^2 \rho) \land ([e_1]^2 \rho \oplus \{ x_1 \mapsto 1 \})
\]

Auxiliary Function:

\[
[e]^2 : (\text{Vars} \to \mathbb{B}) \to \mathbb{B}
\]

\[
[e]^2 \rho = 1
\]

\[
[x]^2 \rho = \rho x
\]

\[
[\Box c]^2 \rho = [c]^2 \rho
\]

\[
[e_1 \sqcup e_2]^2 \rho = [e_1]^2 \rho \land [e_2]^2 \rho
\]

\[
[\text{if } e_0 \ \text{then } e_1 \ \text{else } e_2]^2 \rho = [e_0]^2 \rho \land ([e_1]^2 \rho \lor [e_2]^2 \rho)
\]

\[
[f \ e_1 \ldots e_k]^2 \rho = [f]^2 ([e_1]^2 \rho) \ldots ([e_k]^2 \rho)
\]

System of Equations:

\[
[f_i]^2 b_1 \ldots b_k = [c_i]^2 \{ x_j \mapsto b_j \mid j = 1, \ldots, k \}, \quad i = 1, \ldots, n, b_1, \ldots, b_k \in \mathbb{B}
\]

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

For \texttt{fac2}, we obtain:

\[
\begin{align*}
\lbrack \texttt{fac2}\rbrack^2 b_1 b_2 &= b_1 \land (b_2 \lor \lbrack \texttt{fac2}\rbrack^2 (b_1 \land b_2)) \\
\text{Fixpoint iteration yields:}
\end{align*}
\]

\[
\begin{array}{c|c}
0 & \text{fun } x \rightarrow \text{fun } a \rightarrow 0 \\
1 & \text{fun } x \rightarrow \text{fun } a \rightarrow x \land a \\
2 & \text{fun } x \rightarrow \text{fun } a \rightarrow x \land a \\
\end{array}
\]

We conclude:

\begin{itemize}
\item The function \texttt{fac2} is strict in both arguments, i.e., if evaluation terminates, then also the evaluation of its arguments.
\item Accordingly, we transform:
\end{itemize}

\[
\begin{align*}
\texttt{fac2} &= \text{fun } x \rightarrow \text{fun } a \rightarrow \text{if } x \leq 0 \text{ then } a \\
& \quad \text{else let } \#x' = x - 1 \\
& \quad \#a' = x \cdot a \\
& \quad \text{in } \texttt{fac2} x' a'
\end{align*}
\]

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 \( \mathbb{Z} \).
- For complete partial orderings, Kleene’s fixpoint theorem is applicable \( \blacksquare\).
- As description relation \( \Delta \) we use:

\[
\bot \Delta 0 \text{ and } z \Delta 1 \text{ for } z \in \mathbb{Z}
\]
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 $\mathbb{Z}_1$.
- For complete partial orderings, Kleene’s fixpoint theorem is applicable $\therefore$
- As description relation $\Delta$ we use:

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

Extension of the Syntax:

We additionally consider expressions of the form:

$$
e \ ::= \ldots \mid [] \mid c_1 \triangleleft c_2 \mid \text{match } c_0 \text{ with } [] \rightarrow e_1 \mid x :: x s \rightarrow e_2 \\
\quad \mid (c_1, c_2) \mid \text{match } c_0 \text{ with } (x_1, x_2) \rightarrow e_1
$$

Top Strictness

- We assume that the program is well-typed.
- We are only interested in top constructors.
- Again, we model this property with (monotonic) Boolean functions.
- For int-values, this coincides with strictness $\therefore$
- We extend the abstract evaluation $\llbracket e \rrbracket^\rho$ with rules for case-distinction ...

Extension: Data Structures

- Functions may vary in the parts which they require from a data structure ... 
  $$
  \text{hd} = \text{fun } l \rightarrow \text{match } l \text{ with } x :: x s \rightarrow x
  $$
- \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 ...