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:

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

Analogously, there is index-dependent accumulation:

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

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

\[
\begin{align*}
\text{comp} &= \text{fun } f \rightarrow \text{fun } g \rightarrow \text{fun } i \rightarrow \text{fun } x \rightarrow f \ i \ (g \ i \ x) \\
\text{comp}_1 &= \text{fun } f \rightarrow \text{fun } g \rightarrow \text{fun } i \rightarrow \text{fun } x_1 \rightarrow \text{fun } x_2 \rightarrow f \ i \ (g \ i \ x_1) \ x_2 \\
\text{comp}_2 &= \text{fun } f \rightarrow \text{fun } g \rightarrow \text{fun } i \rightarrow \text{fun } x_1 \rightarrow \text{fun } x_2 \rightarrow f \ i \ x_1 \ (g \ i \ x_2) \\
\text{cmp}_1 &= \text{fun } f \rightarrow \text{fun } g \rightarrow \text{fun } i \rightarrow \text{fun } x_1 \rightarrow \text{fun } x_2 \rightarrow f \ i \ x_1 \ (g \ x_2) \\
\text{cmp}_2 &= \text{fun } f \rightarrow \text{fun } g \rightarrow \text{fun } i \rightarrow \text{fun } x_1 \rightarrow \text{fun } x_2 \rightarrow f \ x_1 \ (g \ i \ x_2)
\end{align*}
\]
Then:

\[
\begin{align*}
\text{comp} \left( \text{map} f \right) \left( \text{map} g \right) &= \text{mapi} \left( \text{comp} \_2 f \ g \right) \\
\text{comp} \left( \text{map} f \right) \left( \text{mapi} g \right) &= \text{mapi} \left( \text{comp} f \ g \right) \\
\text{comp} \left( \text{mapi} f \right) \left( \text{mapi} g \right) &= \text{mapi} \left( \text{comp} f \ g \right) \\
\text{comp} \left( \text{foldli} f \ a \right) \left( \text{mapi} g \right) &= \text{foldli} \left( \text{comp} \_1 f \ g \right) a \\
\text{comp} \left( \text{foldli} f \ a \right) \left( \text{mapi} g \right) &= \text{foldli} \left( \text{comp} \_2 f \ g \right) a \\
\text{comp} \left( \text{foldli} f \ a \right) \left( \text{tabulate} g \right) &= \text{let } h = \text{fun} a \to \text{fun} i \to f \ i \ a \ (g \ i) \\
&\quad \text{in loop } h \ a
\end{align*}
\]

Discussion:

- **Warning:** index-dependent transformations may not commute with \textit{rev} or \textit{filter}.
- All our rules can only be applied if the functions \textit{id}, \textit{map}, \textit{mapi}, \textit{foldi}, \textit{foldli}, \textit{filter}, \textit{rev}, \textit{tabulate}, \textit{rev_tabulate}, \textit{loop}, \textit{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 \textit{tree} \( \alpha \).
- These also provide operations \textit{map}, \textit{mapi} and \textit{foldli}, \textit{foldli} with corresponding rules.
- Further opportunities are opened up by functions \textit{to_list} and \textit{from_list} ...

Then:

\[
\begin{align*}
\text{comp} \left( \text{map} f \right) \left( \text{map} g \right) &= \text{mapi} \left( \text{comp} \_2 f \ g \right) \\
\text{comp} \left( \text{map} f \right) \left( \text{mapi} g \right) &= \text{mapi} \left( \text{comp} f \ g \right) \\
\text{comp} \left( \text{mapi} f \right) \left( \text{mapi} g \right) &= \text{mapi} \left( \text{comp} f \ g \right) \\
\text{comp} \left( \text{foldli} f \ a \right) \left( \text{mapi} g \right) &= \text{foldli} \left( \text{comp} \_1 f \ g \right) a \\
\text{comp} \left( \text{foldli} f \ a \right) \left( \text{mapi} g \right) &= \text{foldli} \left( \text{comp} \_2 f \ g \right) a \\
\text{comp} \left( \text{foldli} f \ a \right) \left( \text{tabulate} g \right) &= \text{let } h = \text{fun} a \to \text{fun} i \to f \ i \ a \ (g \ i) \\
&\quad \text{in loop } h \ a
\end{align*}
\]
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, foldl', 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, foldl' with corresponding rules.
- Further opportunities are opened up by functions to_list and from_list ...

```
type tree α = Leaf | Node(α) (tree α)
map = fun f → fun t → match t with Leaf → Leaf
| Node x l r → let l' = map f l
            r' = map f r
            in Node (f x) l' r'
foldl = fun f → fun a → fun t → match t with Leaf → a
        | Node x l r → let a' = foldl f a l
                         in foldl f (f a' x) r
```

to_list' = fun a → fun t → match t with Leaf → a
| Node x t₁ t₂ → let a' = to_list' a t₂
                     in to_list' (x::a') t₁

to_list = to_list' []

from_list = fun l → match l
            with [] → Leaf
            | x::xs → Node x Leaf (from_list xs)

Warning:

Not every natural equation is valid:

- `comp to_list from_list = id`
- `comp from_list to_list ≠ id`
- `comp to_list (map f) = comp (map f) to_list`
- `comp from_list (map f) = comp (map f) from_list`
- `comp (foldl f a) to_list = foldl f a`
- `comp (foldl f a) from_list = foldl f a`
In this case, there is even a `rev`:

```haskell
rev = \t .
  \case { t \in \text{Leaf} } \rightarrow \text{Leaf}
  | \case { t \in \text{Node} x t_1 t_2 } \rightarrow \begin{cases} 
    s_1 = \text{rev } t_1 \\
    s_2 = \text{rev } t_2 \\
  \end{cases}

\text{in Node} x s_2 s_1
```

```haskell
\text{comp to_list rev = comp rev to_list}
\text{comp from_list rev \neq comp rev from_list}
```

In this case, there is even a `rev`:

```haskell
rev = \t .
  \case { t \in \text{Leaf} } \rightarrow \text{Leaf}
  | \case { t \in \text{Node} x t_1 t_2 } \rightarrow \begin{cases} 
    s_1 = \text{rev } t_1 \\
    s_2 = \text{rev } t_2 \\
  \end{cases}

\text{in Node} x s_2 s_1
```

```haskell
\text{comp to_list rev = comp rev to_list}
\text{comp from_list rev \neq 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.

**Example**

```haskell
\text{from = fun } n \rightarrow \begin{cases} 
  \text{from } (n+1) \\
\end{cases}
```

```haskell
\text{take = fun } k \rightarrow \text{fun } s \rightarrow \begin{cases} 
  \text{match } s \text{ with } \begin{cases} 
    [] \rightarrow [] \\
    x :: xs \rightarrow x :: \text{take } (k-1) xs \\
  \end{cases}
\end{cases}
```
Then CBN yields:

take 5 [from 0] = [0, 1, 2, 3, 4]
— whereas evaluation with CBV does not terminate !!!

Then CBN yields:

take 5 (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 ???

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

\[
\text{fac} = \text{fun } x \rightarrow \begin{cases} 
0 & \text{if } x \leq 0 \\
1 & \text{else}
\end{cases}
\]

According to the evaluation with CBV, does not terminate!!!

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

\[
fac2 = \text{fun } x \rightarrow \text{fun } a \rightarrow \begin{cases} 
0 & \text{if } x \leq 0 \\
x \cdot (x - 1) \cdot (x - 2) & \text{else}
\end{cases}
\]

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...

\[
e := e \mid \mathbf{x} \mid e_1 \mathbf{\oplus} e_2 \mid e_1 e \mid e \mathbf{\text{ if } e_1 \mathbf{\text{ then } e_2 \text{ else } e_2}}
\mid \text{let } \mathbf{r}_1 = e_1 \text{ in } e
\]

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

\[
d := f \mathbf{x}_1 \ldots \mathbf{x}_k = e
\]

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

Idea:

- Describe a \( k \)-ary function

\[
f : \mathbf{\text{int}} \rightarrow \ldots \rightarrow \mathbf{\text{int}}
\]

by a function

\[
[f]^k : \mathbf{B} \rightarrow \ldots \rightarrow \mathbf{B}
\]

- \( 0 \): evaluation does definitely not terminate.
- \( 1 \): evaluation may terminate.
- \( \forall f \exists f^0 = 0 \): If the function call returns a value, then the evaluation of the argument must have terminated and returned a value.

\[
\implies f \text{ is strict.}
\]

Idea (cont.):

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

Auxiliary Function:

\[
[f]^k \cdot b_1 \ldots b_k = [e]^k \cdot \{ x_j \mapsto b_j \mid j = 1, \ldots, k \}, \quad i = 1, \ldots, n, b_1, \ldots, b_k \in \mathbf{B}
\]

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

\[ [f_1]^k b_1 \ldots b_k = [e_1]^k \{ x_j \mapsto b_j \mid j = 1, \ldots, k \}, \quad i = 1, \ldots, n, b_1, \ldots, b_k \in \mathbb{B} \]

- The unknowns of the system of equations are the functions \([f_i]^k\) or the individual entries \([f_i]^k b_1 \ldots b_k\) 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 \(O(2^k)\) :-(

Example:

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

\[ [\text{fac2}]^k b_1 b_2 = b_1 \land (b_2 \lor [\text{fac2}]^k b_1 (b_2 \land b_3)) \]

Fixpoint iteration yields:

<table>
<thead>
<tr>
<th></th>
<th>\text{fun } x \to \text{fun } a \to 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>\text{fun } x \to \text{fun } a \to x \land a</td>
</tr>
<tr>
<td>2</td>
<td>\text{fun } x \to \text{fun } a \to x \land a</td>
</tr>
</tbody>
</table>

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:

\[
\text{fac2} = \text{fun } x \to \text{fun } a \to \text{if } x \leq 0 \text{ then } a \\
\text{else let } \# x' = x - 1 \\
\# a' = x \cdot a \\
\text{in } \text{fac2 } x' a'\
\]
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 $\mathbb{Z}_1$.
- For complete partial orderings, Kleene’s fixpoint theorem is applicable.
- As description relation $\Delta$ we use:

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

Idea (cont.):

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

Auxiliary Function:

\[
\begin{align*}
\uparrow & : (\text{Vars} \to \mathbb{B}) \to \mathbb{B} \\
\uparrow c & = \begin{cases} 
    \text{true} & \text{false} \\
    \text{false} & \text{true}
\end{cases} \\
\uparrow x & = \rho x \\
\uparrow [\text{true} \sqcup \text{false}] & = \uparrow [\text{true}] \sqcup \uparrow [\text{false}] \\
\uparrow \text{if } c_0 \text{ then } c_1 \text{ else } c_2 & = \uparrow [c_0] \sqcap \uparrow [c_1] \sqcup \uparrow [c_2] \\
\uparrow [f_1 \ldots f_n] & = \uparrow [f] \sqcap \uparrow [f_1] \sqcup \ldots \sqcup \uparrow [f_n]
\end{align*}
\]
System of Equations:

\[
[f_i]_{h_1} \ldots b_k = [e_i]_{h} \{ x_j \mapsto b_j \mid j = 1, \ldots, k \}, \quad i = 1, \ldots, n, b_1, \ldots, b_k \in \mathbb{B}
\]

- The unknowns of the system of equations are the functions \([f_i]_{h}\) or the individual entries \([f_i]_{h_1} \ldots b_k\) in the value table.
- All right-hand sides are monotonic!
- Consequently, there is a least solution \(\bot\)
- The complete lattice \(\mathbb{B} \rightarrow \ldots \rightarrow \mathbb{B}\) has height \(O(2^k)\) \(\because\)

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 \(\because\)
- As description relation \(\Delta\) we use:

\[
\bot \triangleq 0 \quad \text{and} \quad z \triangleq 1 \quad \text{for} \quad z \in \mathbb{Z}
\]

Extension: Data Structures

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

\[
\text{hd} = \text{fun } l \to \text{match } l \text{ with } x :: x# \to x
\]

- \(\text{hd}\) 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 ...
Extension of the Syntax:

We additionally consider expression of the form:

\[ e ::= \ldots \mid [\_] \mid e_1 :: e_2 \mid \text{match } e_0 \text{ with } [\_] \rightarrow e_1 \mid x :: xs \rightarrow e_2 \]

\| (e_1, e_2) \mid \text{match } e_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 \texttt{int}-values, this coincides with strictness \(\triangleright\).
- We extend the abstract evaluation \([e]^{\mathcal{I}}_{\rho}\) with rules for case-distinction \(\ldots\)

\[
[\text{match } e_0 \text{ with } [\_] \rightarrow e_1 \mid x :: xs \rightarrow e_2]^{\mathcal{I}}_{\rho} = \]
\[
[\text{match } e_0 \text{ with } (x_1, x_2) \rightarrow e_1]^{\mathcal{I}}_{\rho} = \]

\[
[\text{match } e_0 \text{ with } (x_1, x_2) \rightarrow e_1]^{\mathcal{I}}_{\rho} = \]

- The rules for \texttt{match} are analogous to those for \texttt{if}.
- In case of \texttt{::}, we know nothing about the values beneath the constructor; therefore \(\{x, xs \mapsto 1\}\).
- We check our analysis on the function \texttt{app} \(\ldots\).