Then we have:

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

where

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

Extension (2): List Reversals

Sometimes, the ordering of lists or arguments is reversed:

\[
\begin{align*}
\text{rev}' &= \text{fun} \ a \rightarrow \text{fun} \ l \rightarrow \\
&\quad \text{match} \ l \ \text{with} \ [x] \rightarrow a \\
&\quad \mid \ x :: xs \rightarrow \text{rev}' (x :: a) \ xs \\
\text{rev} &= \text{rev}' \ [] \\
\text{comp} \ \text{rev} \ \text{rev} &= \text{id} \\
\text{swap} &= \text{fun} \ f \rightarrow \text{fun} \ x \rightarrow \text{fun} \ y \rightarrow f \ y \ x \\
\text{comp} \ \text{swap} \ \text{swap} &= \text{id}
\end{align*}
\]

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...
We have:

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

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

\[
\begin{align*}
\text{comp} \; (\text{map} \; f) \; (\text{rev} \; \text{tabulate} \; g) &= \text{rev} \; \text{tabulate} \; (\text{comp}_2 \; f \; g) \\
\text{comp} \; (\text{foldl} \; f \; a) \; (\text{rev} \; \text{tabulate} \; g) &= \text{rev} \; \text{loop} \; (\text{comp}_2 \; f \; g) \; a
\end{align*}
\]

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 ...

We have:

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

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

\[
\begin{align*}
\text{comp} \; (\text{map} \; f) \; (\text{rev} \; \text{tabulate} \; g) &= \text{rev} \; \text{tabulate} \; (\text{comp}_2 \; f \; g) \\
\text{comp} \; (\text{foldl} \; f \; a) \; (\text{rev} \; \text{tabulate} \; g) &= \text{rev} \; \text{loop} \; (\text{comp}_2 \; f \; g) \; a
\end{align*}
\]
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:

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

\[
\text{map} i = \text{map}^{i'} 0
\]

Analogously, there is index-dependent accumulation:

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

\[
\text{foldl} i = \text{foldl}^{i'} 0
\]

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

\[
\text{comp} i = \text{fun } f \rightarrow \text{fun } g \rightarrow \text{fun } i \rightarrow \text{fun } x \rightarrow f i (g i x)
\]

\[
\text{comp}_{i_1} = \text{fun } f \rightarrow \text{fun } g \rightarrow \text{fun } i \rightarrow \text{fun } x_1 \rightarrow \text{fun } x_2 \rightarrow \\
f i (g i x_1) x_2
\]

\[
\text{comp}_{i_2} = \text{fun } f \rightarrow \text{fun } g \rightarrow \text{fun } i \rightarrow \text{fun } x_1 \rightarrow \text{fun } x_2 \rightarrow \\
f i x_1 (g i x_2)
\]

\[
\text{cmp}_{i_1} = \text{fun } f \rightarrow \text{fun } g \rightarrow \text{fun } i \rightarrow \text{fun } x_1 \rightarrow \text{fun } x_2 \rightarrow \\
f i x_1 (g x_2)
\]

\[
\text{cmp}_{i_2} = \text{fun } f \rightarrow \text{fun } g \rightarrow \text{fun } i \rightarrow \text{fun } x_1 \rightarrow \text{fun } x_2 \rightarrow \\
f x_1 (g i x_2)
\]

Then

\[
\text{comp} (\text{map } f) (\text{map } g) = \text{map} (\text{comp}_{i_2} f g) \\
\text{comp} (\text{map } f) (\text{map } g) = \text{map} (\text{comp}_{i_2} f g) \\
\text{comp} (\text{map } f) (\text{map } g) = \text{map} (\text{comp}_{i_2} f g) \\
\text{comp} (\text{foldl } f a) (\text{map } g) = \text{foldl}_{i_2} (\text{comp}_{i_2} f g) a \\
\text{comp} (\text{foldl } f a) (\text{map } g) = \text{foldl}_{i_2} (\text{comp}_{i_2} f g) a \\
\text{comp} (\text{foldl } f a) (\text{map } g) = \text{foldl}_{i_2} (\text{comp}_{i_2} f g) a \\
\text{comp} (\text{foldl } f a) (\text{tabulate } g) = \text{let } h = \text{fun } a \rightarrow \text{fun } i \rightarrow \\
f i a (g i) \text{ in } \text{loop } h a
Then

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

Discussion

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

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

Caveat

Not every natural equation is valid:

\[
\text{comp } \text{to_list } \text{from_list} = \text{id} \\
\text{comp } \text{from_list } \text{to_list} = \neq \text{id} \\
\text{comp } \text{to_list } (\text{map } f) = \text{comp } (\text{map } f) \; \text{to_list} \\
\text{comp } \text{from_list } (\text{map } f) = \text{comp } (\text{map } f) \; \text{from_list} \\
\text{comp } (\text{foldl } f \; a) \; \text{to_list} = \text{foldl } f \; a \\
\text{comp } (\text{foldl } f \; a) \; \text{from_list} = \text{foldl } f \; a
\]

4.6 CBN vs. CBV: Strictness Analysis

Problem

- Programming languages such as Haskell evaluate expressions for \text{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 ...
In this case, there is even a `rev`:

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

\[
\begin{align*}
\text{comp to_list rev} & \quad = \quad \text{comp rev to_list} \\
\text{comp from_list rev} & \quad \neq \quad \text{comp rev from_list}
\end{align*}
\]

### 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**

\[
\begin{align*}
\text{from} & \quad = \quad \text{fun } n \to \text{n::from } (n + 1)
\end{align*}
\]

\[
\begin{align*}
\text{take} & \quad = \quad \text{fun } k \to \text{fun } s \to \text{if } k \leq 0 \text{ then } [] \\
& \quad \text{else } \text{match } s \text{ with } [ ] \to [ ] \\
& \quad \quad \mid \text{x::xs} \to \text{x::take } (k - 1) \text{ xs}
\end{align*}
\]

**Then CBN yields**

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

- whereas evaluation with CBV does not terminate !!!
Then CBN yields

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

- whereas evaluation with CBV does not terminate !!!

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

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

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 ...
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 ...

\[
e \ ::= \ c \mid x \mid c_1 \Box c_2 \mid \Box c \mid f \ c_1 \ldots c_k \mid \text{let } r_1 = c_1 \text{ in } c
\]

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

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

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

852

Idea (cont.)

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

Auxiliary Function

\[
\begin{align*}
[c]^1 & : (\text{Vars} \to \mathbb{B}) \to \mathbb{B} \\
[c]^2 \rho & = 1 \\
[x]^1 \rho & = \rho x \\
[\Box c]^1 \rho & = [c]^1 \rho \\
[c_1 \Box c_2]^1 \rho & = [c_1]^1 \rho \land [c_2]^1 \rho \\
[\text{if } c_0 \text{ then } c_1 \text{ else } c_2]^1 \rho & = [c_0]^1 \rho \land ([c_1]^1 \rho \lor [c_2]^1 \rho) \\
[f \ c_1 \ldots c_k]^1 \rho & = [f]^1 ([c_1]^1 \rho) \ldots ([c_k]^1 \rho)
\end{align*}
\]

854

Idea

- Describe a \( k \)-ary function

\[
f : \text{int} \to \ldots \to \text{int}
\]

by a function

\[
[f]^1 : \mathbb{B} \to \ldots \to \mathbb{B}
\]

- 0 means: evaluation does definitely not terminate.
- 1 means: evaluation may terminate.

\[
[f]^1 0 = 0 \quad \text{means: If the function call returns a value, then the evaluation of the argument must have terminated and returned a value.}
\]

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

853
Idea (cont.)

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

**Auxiliary Function**

\[
\begin{align*}
[v]_\rho & : (\text{Vars} \to \text{B}) \to \text{B} \\
[e]_\rho & = 1 \\
[\varnothing]_\rho & = \rho x \\
[e_1 \cdot\cdot\cdot e_2]_\rho & = [e_1]_\rho \wedge [e_2]_\rho \\
\text{if } e_0 \text{ then } e_1 \text{ else } e_2]_\rho & = [e_0]_\rho \wedge ([e_1]_\rho \vee [e_2]_\rho) \\
[f_1 \cdot\cdot\cdot f_k]_\rho & = [f_1]_\rho ([e_1]_\rho \cdot\cdot\cdot ([e_k]_\rho) \\
& \ldots
\end{align*}
\]

**System of Equations**

\[
\begin{align*}
[f_i]_b & = [e_i]_\rho (x_j \mapsto b_j | j = 1, \ldots, k), & i = 1, \ldots, n, b_1, \ldots, b_k & \in \text{B}
\end{align*}
\]

- The unknowns of the system of equations are the functions \([f_i]_b\) and the individual entries \([e_i]_\rho\) in the value table.
- All right-hand sides are **monotonic**.
- Consequently, there is a least solution.
- The complete lattice \(\text{B} \to \ldots \to \text{B}\) has height \(O(2^k)\).

**Example**

For \text{fac2}, we obtain:

\[
\begin{align*}
[\text{fac2}]_b & = b_1 \wedge (b_2 \vee \\
& [\text{fac2}]_b (b_1 \wedge b_2))
\end{align*}
\]

Fixpoint iteration yields:

| 0 | \text{fun} \ x \to \ \text{fun} \ a \to \ 0 |
| 1 | \text{fun} \ x \to \ \text{fun} \ a \to \ x \wedge a |
| 2 | \text{fun} \ x \to \ \text{fun} \ a \to \ x \wedge a |

**Then CBN yields**

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

- whereas evaluation with \text{CBV} does not terminate !!!

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

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

\[
\begin{align*}
[\text{fac2}]^1 b_1 b_2 &= b_1 \land (b_2 \lor b_1) \\
[\text{fac2}]^2 b_1 (b_1 \land b_2)
\end{align*}
\]

Fixpoint iteration yields:

<table>
<thead>
<tr>
<th>0</th>
<th>fun ( x ) \to fun ( a ) \to 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>fun ( x ) \to fun ( a ) \to ( x \land a )</td>
</tr>
<tr>
<td>2</td>
<td>fun ( x ) \to 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:

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

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:

\[
\bot \Delta 0 \quad \text{and} \quad x \Delta 1 \quad \text{for } x \in \mathbb{Z}_1
\]

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.
- 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:

\[
\begin{align*}
e & ::= \ldots | [ ] | e_1 :: e_2 | \text{match } e_0 \text{ with } [ ] \to e_1 | x :: x_0 \to e_2 \\
& | (e_1, e_2) | \text{match } e_0 \text{ with } (x_1, x_2) \to e_1
\end{align*}
\]

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.
- We extend $\llbracket e \rrbracket^p$ with rules for case-distinction ...