Extension (2): List Reversals

Sometimes, the ordering of lists or arguments is reversed:

\[
\begin{align*}
rev' &= \text{fun } a \rightarrow \text{fun } l \rightarrow \\
&\quad \text{match } l \text{ with } [] \rightarrow a \\
&\quad \mid x :: xs \rightarrow rev' (x :: a) xs \\
rev &= rev' [] \\
\text{comp rev rev} &= id \\
\text{swap} &= \text{fun } f \rightarrow \text{fun } x \rightarrow \text{fun } y \rightarrow f y x \\
\text{comp swap swap} &= id
\end{align*}
\]
Extension (2): List Reversals

Sometimes, the ordering of lists or arguments is reversed:

\[
\text{let } \text{rev } \text{ = comp (foldl (swap f) a) rev}
\]

\[
\text{let } \text{rc } \text{ = comp (map f) rev}
\]

\[
\text{let } \text{tabulate } \text{ = comp (tabulate f) rev}
\]

Let \( \text{rc} \) be a list of\( f \) applied to \( a \) and \( \text{rev} \) be the reverse of \( \text{rc} \).

\[
\text{comp rev (map f) } = \text{comp (map f) rev}
\]

\[
\text{comp rev (filter p) } = \text{comp (filter p) rev}
\]

\[
\text{comp rev (tabulate f) } = \text{rev_tabulate f}
\]

\[
\text{comp (map f) (rev_tabulate g) } = \text{rev_tabulate (comp \_ f g)}
\]

\[
\text{comp (foldl f a) (rev_tabulate g) } = \text{rev_loop (comp \_ f g) a}
\]

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:

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

\[
\text{comp (map f) (rev_tabulate g) } = \text{rev_tabulate (comp \_ f g)}
\]

\[
\text{comp (foldl f a) (rev_tabulate g) } = \text{rev_loop (comp \_ f g) a}
\]
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:
  \[
  \text{mapi}' = \begin{array}{l}
  \text{fun } i \to \text{fun } f \to \text{fun } l \to \\
  \text{match } l \text{ with } [] \to [] \\
  | x :: xs \to f x :: \text{mapi}' (i + 1) f xs
  \end{array}
  \]
  \[
  \text{mapi} = \text{mapi}' 0
  \]

Analogously, there is index-dependent accumulation:

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

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

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

\[
\text{compi} = \text{fun } f \to \text{fun } g \to \text{fun } i \to \text{fun } x \to f i (g i x)
\]

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

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

\[
\text{cmpi} = \text{fun } f \to \text{fun } g \to \text{fun } i \to \text{fun } x_1 \to \text{fun } x_2 \to \\
f i x_1 (g i x_2)
\]

Then:

\[
\text{comp} (\text{mapi } f) (\text{mapi } g) = \text{mapi} (\text{comp}_2 f g)
\]

\[
\text{comp} (\text{mapi } f) (\text{mapi } g) = \text{mapi} (\text{comp}_2 f g)
\]

\[
\text{comp} (\text{mapi } f) (\text{mapi } g) = \text{mapi} (\text{comp}_2 f g)
\]

\[
\text{comp} (\text{foldli } f a) (\text{mapi } g) = \text{foldli} (\text{cmpi}_2 f g) a
\]

\[
\text{comp} (\text{foldli } f a) (\text{mapi } g) = \text{foldli} (\text{cmpi}_2 f g) a
\]

\[
\text{comp} (\text{foldli } f a) (\text{mapi } g) = \text{foldli} (\text{cmpi}_2 f g) a
\]

\[
\text{comp} (\text{foldli } f a) (\text{tabulate } g) = \text{let } h = \text{fun } a \to \text{fun } i \to \\
f i a (g i) \in \text{loop } h a
\]
\[ \text{comp} \quad = \quad \text{fun } f \rightarrow \text{fun } g \rightarrow \text{fun } i \rightarrow \text{fun } x \rightarrow f \ i \ (g \ i \ x) \]
\[ \text{comp}_1 \quad = \quad \text{fun } f \rightarrow \text{fun } g \rightarrow \text{fun } i \rightarrow \text{fun } x_1 \rightarrow \text{fun } x_2 \rightarrow f \ i \ (g \ i \ x_1) \ x_2 \]
\[ \text{comp}_2 \quad = \quad \text{fun } f \rightarrow \text{fun } g \rightarrow \text{fun } i \rightarrow \text{fun } x_1 \rightarrow \text{fun } x_2 \rightarrow f \ i \ x_1 \ (g \ i \ x_2) \]
\[ \text{cmp}_1 \quad = \quad \text{fun } f \rightarrow \text{fun } g \rightarrow \text{fun } i \rightarrow \text{fun } x_1 \rightarrow \text{fun } x_2 \rightarrow f \ i \ x_1 \ (g \ x_2) \]
\[ \text{cmp}_2 \quad = \quad \text{fun } f \rightarrow \text{fun } g \rightarrow \text{fun } i \rightarrow \text{fun } x_1 \rightarrow \text{fun } x_2 \rightarrow f \ x_1 \ (g \ i \ x_2) \]

**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`, `foldll`, `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`, `foldll` with corresponding rules.
- Further opportunities are opened up by functions `to_list` and `from_list` ...

**Then:**
\[ \text{comp} \ (\text{mapi } f) \ (\text{map } g) \quad = \quad \text{mapi} \ (\text{comp}_2 f g) \]
\[ \text{comp} \ (\text{map } f) \ (\text{mapi } g) \quad = \quad \text{mapi} \ (\text{comp } f g) \]
\[ \text{comp} \ (\text{mapi } f) \ (\text{mapi } g) \quad = \quad \text{mapi} \ (\text{comp } f g) \]
\[ \text{comp} \ (\text{foldl } f \ a) \ (\text{map } g) \quad = \quad \text{foldl} \ (\text{comp } f g) \ a \]
\[ \text{comp} \ (\text{foldl } f \ a) \ (\text{mapi } g) \quad = \quad \text{foldl} \ (\text{comp}_2 f g) \ a \]
\[ \text{comp} \ (\text{foldl } f \ a) \ (\text{mapi } g) \quad = \quad \text{foldl} \ (\text{comp}_2 f g) \ a \]
\[ \text{comp} \ (\text{foldl } f \ a) \ (\text{tabulate } g) \quad = \quad \text{let } h = \text{fun } a \rightarrow \text{fun } i \rightarrow \]
\[ f \ i \ a \ (g \ i) \]
\[ \text{in } \text{loop } h \ a \]

**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`, `foldll`, `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`, `foldll` with corresponding rules.
- Further opportunities are opened up by functions `to_list` and `from_list` ...
Example

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

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

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

Example

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

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

\[
\text{foldl} = \text{fun } f \to \text{fun } a \to \text{fun } t \to \text{match } t \text{ with } \text{Leaf} \to a
\mid \text{Node } x \ l \ r \to \text{let } a' = \boxed{\text{foldl } f \, a \, l} \\
\quad \text{in } \text{foldl } f (f \, a' \, x) \, r
\]

Warning:

Not every natural equation is valid:

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

```haskell
rev  =  fun t ->
  match t with Leaf -> Leaf
  | Node x t1 t2 -> let s1 = rev t1
                    s2 = rev t2
                    in Node x s2 s1
```

```haskell
comp to_list rev  =  comp rev to_list
comp from_list rev ≠ 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...