Script generated by TTT

Title: Seidl: Programmoptimierung (05.02.2014)

Date: Wed Feb 05 08:32:53 CET 2014

Duration: 39:19 min

Pages: 21

Extension (2): List Reversals

Sometimes, the ordering of lists or arguments is reversed:

rev′ = **fun** a → **fun** l →
  match l **with** [] → a
  | $x :: xs$ → rev′ $(x :: a)$ xs

rev = rev′ []

comp rev rev = id

swap = **fun** f → **fun** x → **fun** y → $f\ y\ x$

comp swap swap = id

Extension (2): List Reversals

Sometimes, the ordering of lists or arguments is reversed:

rev′ = **fun** a → **fun** l →
  match l **with** [] → a
  | $x :: xs$ → rev′ $(x :: a)$ xs

rev = rev′ []

comp rev rev = id

swap = **fun** f → **fun** x → **fun** y → $f\ y\ x$

comp swap swap = id

837

let rec foldr f l a = match l
with [] → a
| x :: xs → f x (foldr f xs a)

Extension (2): List Reversals

Sometimes, the ordering of lists or arguments is reversed:

$$
\begin{aligned}
\mathsf{rev'} &= \textbf{fun}\ a \to \textbf{fun}\ l \to \\
&\quad \textbf{match}\ l\ \textbf{with}\ [\,] \to a \\
&\quad \mid\ x :: xs \to \mathsf{rev'}\ (x :: a)\ xs \\
\mathsf{rev} &= \mathsf{rev'}\ [\,] \\
\mathsf{comp\ rev\ rev} &= \mathsf{id} \\
\mathsf{swap} &= \textbf{fun}\ f \to \textbf{fun}\ x \to \textbf{fun}\ y \to f\ y\ x \\
\mathsf{comp\ swap\ swap} &= \mathsf{id}
\end{aligned}
$$

$$\mathsf{foldr}\ f\ a = \mathsf{comp}\ (\mathsf{foldl}\ (\mathsf{swap}\ f)\ a)\ \mathsf{rev}$$

Discussion:

- The standard implementation of foldr is not tail-recursive.
- The last equation decomposes a 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 rev can also be optimized away ...

We have:

$$
\begin{aligned}
\mathsf{comp\ rev}\ (\mathsf{map}\ f) &= \mathsf{comp}\ (\mathsf{map}\ f)\ \mathsf{rev} \\
\mathsf{comp\ rev}\ (\mathsf{filter}\ p) &= \mathsf{comp}\ (\mathsf{filter}\ p)\ \mathsf{rev} \\
\mathsf{comp\ rev}\ (\mathsf{tabulate}\ f) &= \mathsf{rev\_tabulate}\ f
\end{aligned}
$$

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

$$
\begin{aligned}
\mathsf{comp}\ (\mathsf{map}\ f)\ (\mathsf{rev\_tabulate}\ g) &= \mathsf{rev\_tabulate}\ (\mathsf{comp}_2\ f\ g) \\
\mathsf{comp}\ (\mathsf{foldl}\ f\ a)\ (\mathsf{rev\_tabulate}\ g) &= \mathsf{rev\_loop}\ (\mathsf{comp}_2\ f\ g)\ a
\end{aligned}
$$

We have:

$$
\begin{aligned}
\mathsf{comp\ rev}\ (\mathsf{map}\ f) &= \mathsf{comp}\ (\mathsf{map}\ f)\ \mathsf{rev} \\
\mathsf{comp\ rev}\ (\mathsf{filter}\ p) &= \mathsf{comp}\ (\mathsf{filter}\ p)\ \mathsf{rev} \\
\mathsf{comp\ rev}\ (\mathsf{tabulate}\ f) &= \mathsf{rev\_tabulate}\ f
\end{aligned}
$$

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

$$
\begin{aligned}
\mathsf{comp}\ (\mathsf{map}\ f)\ (\mathsf{rev\_tabulate}\ g) &= \mathsf{rev\_tabulate}\ (\mathsf{comp}_2\ f\ g) \\
\mathsf{comp}\ (\mathsf{foldl}\ f\ a)\ (\mathsf{rev\_tabulate}\ g) &= \mathsf{rev\_loop}\ (\mathsf{comp}_2\ f\ g)\ a
\end{aligned}
$$

839

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:

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

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:

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

$$\mathsf{compi}_1 = \textbf{fun } f \to \textbf{fun } g \to \textbf{fun } i \to \textbf{fun } x_1 \to \textbf{fun } x_2 \to f\ i\ (g\ i\ x_1)\ x_2$$

$$\mathsf{compi}_2 = \textbf{fun } f \to \textbf{fun } g \to \textbf{fun } i \to \textbf{fun } x_1 \to \textbf{fun } x_2 \to f\ i\ x_1\ (g\ i\ x_2)$$

$$\mathsf{cmp}_1 = \textbf{fun } f \to \textbf{fun } g \to \textbf{fun } i \to \textbf{fun } x_1 \to \textbf{fun } x_2 \to f\ i\ x_1\ (g\ x_2)$$

$$\mathsf{cmp}_2 = \textbf{fun } f \to \textbf{fun } g \to \textbf{fun } i \to \textbf{fun } x_1 \to \textbf{fun } x_2 \to f\ x_1\ (g\ i\ x_2)$$

Then:

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

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

$$\mathsf{compi}_1 \; = \; \textbf{fun}\; f \to \textbf{fun}\; g \to \textbf{fun}\; i \to \textbf{fun}\; x_1 \to \textbf{fun}\; x_2 \to f\; i\; (g\; i\; x_1)\; x_2$$

$$\mathsf{compi}_2 \; = \; \textbf{fun}\; f \to \textbf{fun}\; g \to \textbf{fun}\; i \to \textbf{fun}\; x_1 \to \textbf{fun}\; x_2 \to f\; i\; x_1\; (g\; i\; x_2)$$

$$\mathsf{cmp}_1 \; = \; \textbf{fun}\; f \to \textbf{fun}\; g \to \textbf{fun}\; i \to \textbf{fun}\; x_1 \to \textbf{fun}\; x_2 \to f\; i\; x_1\; (g\; x_2)$$

$$\mathsf{cmp}_2 \; = \; \textbf{fun}\; f \to \textbf{fun}\; g \to \textbf{fun}\; i \to \textbf{fun}\; x_1 \to \textbf{fun}\; x_2 \to f\; x_1\; (g\; i\; x_2)$$

Then:

$$\begin{aligned}
\mathsf{comp}\,(\mathsf{mapi}\; f)\;(\mathsf{map}\; g) &= \mathsf{mapi}\,(\mathsf{comp}_2\; f\; g)\\
\mathsf{comp}\,(\mathsf{map}\; f)\;(\mathsf{mapi}\; g) &= \mathsf{mapi}\,(\mathsf{comp}\; f\; g)\\
\mathsf{comp}\,(\mathsf{mapi}\; f)\;(\mathsf{mapi}\; g) &= \mathsf{mapi}\,(\mathsf{compi}\; f\; g)\\
\mathsf{comp}\,(\mathsf{foldli}\; f\; a)\;(\mathsf{map}\; g) &= \mathsf{foldli}\,(\mathsf{cmp}_1\; f\; g)\; a\\
\mathsf{comp}\,(\mathsf{foldl}\; f\; a)\;(\mathsf{mapi}\; g) &= \mathsf{foldli}\,(\mathsf{cmp}_2\; f\; g)\; a\\
\mathsf{comp}\,(\mathsf{foldli}\; f\; a)\;(\mathsf{mapi}\; g) &= \mathsf{foldli}\,(\mathsf{compi}_2\; f\; g)\; a\\
\mathsf{comp}\,(\mathsf{foldli}\; f\; a)\;(\mathsf{tabulate}\; g) &= \textbf{let}\; h = \textbf{fun}\; a \to \textbf{fun}\; i \to \boxed{f\; i\; a\; (g\, i)}\\
&\quad\; \textbf{in}\; \mathsf{loop}\; h\; a
\end{aligned}$$

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 ...

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 ...

844

Example

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

Example

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

$$\begin{array}{lll}
\mathsf{to_list}' & = & \textbf{fun}\ a\ \to\ \textbf{fun}\ t\ \to\ \textbf{match}\ t\ \textbf{with}\ \mathsf{Leaf}\ \to\ a \\
 & & \mid\ \mathsf{Node}\ x\ t_1\ t_2\ \to\ \textbf{let}\ a' = \mathsf{to_list}'\ a\ t_2 \\
 & & \qquad\qquad\qquad\qquad \textbf{in}\ \mathsf{to_list}'\ (x :: a')\ t_1 \\
\mathsf{to_list} & = & \mathsf{to_list}'\ [\,] \\
 & & \\
\mathsf{from_list} & = & \textbf{fun}\ l\ \to\ \textbf{match}\ l \\
 & & \textbf{with}\ [\,]\ \to\ \mathsf{Leaf} \\
 & & \mid\ x :: xs\ \to \mathsf{Node}\ x\ \mathsf{Leaf}\ (\mathsf{from_list}\ xs)
\end{array}$$

Warning:

Not every natural equation is valid:

$$\begin{array}{lll}
\mathsf{comp}\ \mathsf{to_list}\ \mathsf{from_list} & = & \mathsf{id} \\
\mathsf{comp}\ \mathsf{from_list}\ \mathsf{to_list} & \neq & \mathsf{id} \\
\mathsf{comp}\ \mathsf{to_list}\ (\mathsf{map}\ f) & = & \mathsf{comp}\ (\mathsf{map}\ f)\ \mathsf{to_list} \\
\mathsf{comp}\ \mathsf{from_list}\ (\mathsf{map}\ f) & = & \mathsf{comp}\ (\mathsf{map}\ f)\ \mathsf{from_list} \\
\mathsf{comp}\ (\mathsf{foldl}\ f\ a)\ \mathsf{to_list} & = & \mathsf{foldl}\ f\ a \\
\mathsf{comp}\ (\mathsf{foldl}\ f\ a)\ \mathsf{from_list} & = & \mathsf{foldl}\ f\ a
\end{array}$$

In this case, there is even a rev:

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

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

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 ...