Script generated by TTT

Title: Seidl: Programmoptimierung (28.01.2016)

Date: Thu Jan 28 08:40:22 CET 2016

Duration: 84:30 min

Pages: 40

Then we have:

$$\text{comp } (\text{map } f)\ (\text{tabulate } g) = \text{tabulate } (\text{comp } f\ g)$$
$$\text{comp } (\text{foldl } f\ a)\ (\text{tabulate } g) = \text{loop } (\text{comp}_2\ f\ g)\ a$$

where

$$\begin{aligned}\text{loop}' &= \textbf{fun } j \to \textbf{fun } f \to \textbf{fun } a \to \textbf{fun } n \to \\ &\qquad \textbf{if } j \geq n \textbf{ then } a \\ &\qquad \textbf{else } \text{loop}'\ (j+1)\ f\ (f\ a\ j))\ n \\ \text{loop} &= \text{loop}'\ 0\end{aligned}$$

Extension (2): List Reversals

foldr (fun a x -> x::a)

Sometimes, the ordering of lists or arguments is reversed:

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

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

foldr f a = function [] -> a | x::xs -> f x (foldr f a xs)

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:

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

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

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

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

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:

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

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

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

We have:

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

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

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

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:

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

Analogously, there is index-dependent accumulation:

$$\begin{array}{lll} \text{foldli}' & = & \textbf{fun}\ i \rightarrow \textbf{fun}\ f \rightarrow \textbf{fun}\ a \rightarrow \textbf{fun}\ l \rightarrow \\ & & \quad \textbf{match}\ l\ \textbf{with}\ [\,] \rightarrow a \\ & & \quad \mid x :: xs \rightarrow \text{foldli}'\ (i+1)\ f\ (f\ i\ a\ x)\ xs \\ \text{foldli} & = & \text{foldli}'\ 0 \end{array}$$

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

$$\begin{array}{lll} \text{compi} & = & \textbf{fun}\ f \rightarrow \textbf{fun}\ g \rightarrow \textbf{fun}\ i \rightarrow \textbf{fun}\ x \rightarrow f\ i\ (g\ i\ x) \\ \\ \text{compi}_1 & = & \textbf{fun}\ f \rightarrow \textbf{fun}\ g \rightarrow \textbf{fun}\ i \rightarrow \textbf{fun}\ x_1 \rightarrow \textbf{fun}\ x_2 \rightarrow \\ & & \quad f\ i\ (g\ i\ x_1)\ x_2 \\ \\ \text{compi}_2 & = & \textbf{fun}\ f \rightarrow \textbf{fun}\ g \rightarrow \textbf{fun}\ i \rightarrow \textbf{fun}\ x_1 \rightarrow \textbf{fun}\ x_2 \rightarrow \\ & & \quad f\ i\ x_1\ (g\ i\ x_2) \\ \\ \text{cmp}_1 & = & \textbf{fun}\ f \rightarrow \textbf{fun}\ g \rightarrow \textbf{fun}\ i \rightarrow \textbf{fun}\ x_1 \rightarrow \textbf{fun}\ x_2 \rightarrow \\ & & \quad f\ i\ x_1\ (g\ x_2) \\ \\ \text{cmp}_2 & = & \textbf{fun}\ f \rightarrow \textbf{fun}\ g \rightarrow \textbf{fun}\ i \rightarrow \textbf{fun}\ x_1 \rightarrow \textbf{fun}\ x_2 \rightarrow \\ & & \quad f\ x_1\ (g\ i\ x_2) \end{array}$$

Then

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

Then

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

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 ...

Then

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

840

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 ...

841

Example

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

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

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

$$\begin{array}{lll}
\text{from_list} & = & \textbf{fun}\ l \to \textbf{match}\ l \\
 & & \textbf{with}\ [\,] \to \text{Leaf} \\
 & & \mid\ x :: xs \to \text{Node}\ x\ \text{Leaf}\ (\text{from_list}\ xs)
\end{array}$$

Caveat

Not every natural equation is valid:

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

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 ...

In this case, there is even a rev:

$$\begin{array}{lll} \text{rev} & = & \textbf{fun}\ t \to \\ & & \textbf{match}\ t\ \textbf{with}\ \text{Leaf} \to \text{Leaf} \\ & & \mid\ \text{Node}\ x\ t_1\ t_2 \to \textbf{let}\ s_1 = \text{rev}\ t_1 \\ & & \qquad\qquad\qquad\qquad s_2 = \text{rev}\ t_2 \\ & & \qquad\qquad\quad \textbf{in}\ \text{Node}\ x\ s_2\ s_1 \end{array}$$

$$\begin{array}{lll} \text{comp to\_list rev} & = & \text{comp rev to\_list} \\ \text{comp from\_list rev} & \neq & \text{comp rev from\_list} \end{array}$$

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 ...

Example

$$\text{from} = \textbf{fun}\ n \to n :: \text{from}\ (n+1)$$

$$\begin{array}{l} \text{take} = \textbf{fun}\ k \to \textbf{fun}\ s \to \textbf{if}\ k \leq 0\ \textbf{then}\ [\,] \\ \qquad\qquad\qquad\qquad\qquad \textbf{else}\ \textbf{match}\ s\ \textbf{with}\ [\,] \to [\,] \\ \qquad\qquad\qquad\qquad\qquad\qquad \mid\ x :: xs \to x :: \text{take}\ (k-1)\ xs \end{array}$$

Then CBN yields

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

— whereas evaluation with CBV does not terminate !!!

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 ???

$$\begin{array}{llllll} \text{fac2} & = & \mathbf{fun}\ x & \rightarrow & \mathbf{fun}\ a\ \rightarrow & \mathbf{if}\ x \leq 0\ \mathbf{then}\ a \\ & & & & & \mathbf{else}\ \ \text{fac2}\ (x-1)\ (a \cdot x) \end{array}$$

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 ...

851

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 ...

$$
\begin{array}{lll}
e & ::= & c \mid x \mid e_1 \,\square_2\, e_2 \mid \square_1\, e \mid f\; e_1\; \ldots\; e_k \mid \textbf{if}\; e_0\; \textbf{then}\; e_1\; \textbf{else}\; e_2 \\
  &   & \mid \textbf{let}\; r_1 = e_1\; \textbf{in}\; e \\
r & ::= & x \mid \#x \\
d & ::= & f\; x_1\; \ldots\; x_k = e \\
p & ::= & \textbf{letrec and}\; d_1\; \ldots\; \textbf{and}\; d_n\; \textbf{in}\; e
\end{array}
$$

Idea

- Describe a k-ary function

$$f : \textbf{int} \to \ldots \to \textbf{int}$$

by a function

$$[\![f]\!]^\sharp : \mathbb{B} \to \ldots \to \mathbb{B}$$

- 0 means: evaluation does definitely not terminate.
- 1 means: evaluation may terminate.
- $[\![f]\!]^\sharp\; 0 = 0$ means: If the function call returns a value, then the evaluation of the argument must have terminated and returned a value.

$\Longrightarrow$ f is strict.

Idea (cont.)

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

Auxiliary Function

$$
\begin{array}{lll}
[\![e]\!]^\sharp & : & (\mathit{Vars} \to \mathbb{B}) \to \mathbb{B} \\
[\![c]\!]^\sharp\; \rho & = & 1 \\
[\![x]\!]^\sharp\; \rho & = & \rho\, x \\
[\![\square_1\; e]\!]^\sharp\; \rho & = & [\![e]\!]^\sharp\; \rho \\
[\![e_1\; \square_2\; e_2]\!]^\sharp\; \rho & = & [\![e_1]\!]^\sharp\; \rho \wedge [\![e_2]\!]^\sharp\; \rho \\
[\![\textbf{if}\; e_0\; \textbf{then}\; e_1\; \textbf{else}\; e_2]\!]^\sharp\; \rho & = & [\![e_0]\!]^\sharp\; \rho \wedge ([\![e_1]\!]^\sharp\; \rho \vee [\![e_2]\!]^\sharp\; \rho) \\
[\![f\; e_1\; \ldots\; e_k]\!]^\sharp\; \rho & = & [\![f]\!]^\sharp\; ([\![e_1]\!]^\sharp\; \rho)\; \ldots\; ([\![e_k]\!]^\sharp\; \rho) \\
\ldots & &
\end{array}
$$

Idea

- Describe a k-ary function

$$f : \textbf{int} \to \ldots \to \textbf{int}$$

by a function

$$[\![f]\!]^\sharp : \mathbb{B} \to \ldots \to \mathbb{B}$$

- 0 means: evaluation does definitely not terminate.
- 1 means: evaluation may terminate.
- $[\![f]\!]^\sharp\; 0 = 0$ means: If the function call returns a value, then the evaluation of the argument must have terminated and returned a value.

$\Longrightarrow$ f is strict.

853

Idea (cont.)

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

Auxiliary Function

$$
\begin{array}{lcl}
[\![e]\!]^\sharp & : & (\mathit{Vars} \to \mathbb{B}) \to \mathbb{B} \\
[\![c]\!]^\sharp\ \rho & = & 1 \\
[\![x]\!]^\sharp\ \rho & = & \rho\, x \\
[\![\Box_1\ e]\!]^\sharp\ \rho & = & [\![e]\!]^\sharp\ \rho \\
[\![e_1\ \Box_2\ e_2]\!]^\sharp\ \rho & = & [\![e_1]\!]^\sharp\ \rho \wedge [\![e_2]\!]^\sharp\ \rho \\
[\![\mathbf{if}\ e_0\ \mathbf{then}\ e_1\ \mathbf{else}\ e_2]\!]^\sharp\ \rho & = & [\![e_0]\!]^\sharp\ \rho \wedge ([\![e_1]\!]^\sharp\ \rho \vee [\![e_2]\!]^\sharp\ \rho) \\
[\![f\ e_1\ \ldots\ e_k]\!]^\sharp\ \rho & = & [\![f]\!]^\sharp\ ([\![e_1]\!]^\sharp\ \rho)\ \ldots\ ([\![e_k]\!]^\sharp\ \rho) \\
\ldots & &
\end{array}
$$

$$
\begin{array}{lcl}
[\![\mathbf{let}\ x_1 = e_1\ \mathbf{in}\ e]\!]^\sharp\ \rho & = & [\![e]\!]^\sharp\ (\rho \oplus \{x_1 \mapsto [\![e_1]\!]^\sharp\ \rho\}) \\
[\![\mathbf{let}\ \#x_1 = e_1\ \mathbf{in}\ e]\!]^\sharp\ \rho & = & ([\![e_1]\!]^\sharp\ \rho) \wedge ([\![e]\!]^\sharp\ (\rho \oplus \{x_1 \mapsto 1\}))
\end{array}
$$

System of Equations

$$
[\![f_i]\!]^\sharp b_1\ \ldots\ b_k = [\![e_i]\!]^\sharp\ \{x_j \mapsto b_j \mid j = 1, \ldots, k\}, \qquad i = 1, \ldots, n, b_1, \ldots, b_k \in \mathbb{B}
$$

- The unkowns of the system of equations are the functions $[\![f_i]\!]^\sharp$ or the individual entries $[\![f_i]\!]^\sharp b_1\ \ldots\ b_k$ 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 $\mathcal{O}(2^k)$.

Example

For fac2, we obtain:

$$
\begin{array}{lcl}
[\![\mathsf{fac2}]\!]^\sharp\ b_1\ b_2 & = & b_1 \wedge (b_2 \vee \\
& & [\![\mathsf{fac2}]\!]^\sharp\ b_1\ (b_1 \wedge b_2))
\end{array}
$$

Fixpoint iteration yields:

0	$\mathbf{fun}\ x \to \mathbf{fun}\ a \to 0$
1	$\mathbf{fun}\ x \to \mathbf{fun}\ a \to x \wedge a$
2	$\mathbf{fun}\ x \to \mathbf{fun}\ a \to x \wedge a$

Then CBN yields

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

— whereas evaluation with CBV does not terminate !!!

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

$$
\begin{array}{lcl}
\mathsf{fac2} & = & \mathbf{fun}\ x\ \to\ \mathbf{fun}\ a\ \to\ \ \mathbf{if}\ x \leq 0\ \mathbf{then}\ a \\
& & \qquad\qquad\qquad\qquad\ \mathbf{else}\ \mathsf{fac2}\ (x-1)\ (a \cdot x)
\end{array}
$$

Example

For fac2, we obtain:

$$[\![\text{fac2}]\!]^\sharp\ b_1\ b_2 = b_1 \wedge (b_2 \vee [\![\text{fac2}]\!]^\sharp\ b_1\ (b_1 \wedge b_2))$$

Fixpoint iteration yields:

0	$\textbf{fun}\ x \to \textbf{fun}\ a \to 0$
1	$\textbf{fun}\ x \to \textbf{fun}\ a \to x \wedge a$
2	$\textbf{fun}\ x \to \textbf{fun}\ a \to x \wedge a$

We conclude:

- The function fac2 is strict in both arguments, i.e., if evaluation terminates, then also the evaluation of its arguments.
- Accordingly, we transform:

$$\begin{array}{lllll} \text{fac2} & = & \textbf{fun}\ x \to \textbf{fun}\ a \to & \textbf{if}\ x \leq 0\ \textbf{then}\ a \\ & & & \textbf{else}\ \ \textbf{let}\ \ \#\,x' = x - 1 \\ & & & \qquad\qquad \#\,a' = x \cdot a \\ & & & \textbf{in}\ \ \text{fac2}\ x'\ a' \end{array}$$

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 $\mathbb{Z}_\perp$.
- For complete partial orderings, Kleene's fixpoint theorem is applicable.
- As description relation Δ we use:

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

Extension: Data Structures

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

$$\text{hd} = \textbf{fun}\ l \to \textbf{match}\ l\ \textbf{with}\ x :: xs \to x$$

- hd only accesses the first element of a list.
- length only accesses the backbone of its argument.
- 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:

$$e \quad ::= \quad \ldots \quad | \quad [\,] \mid e_1 :: e_2 \mid \textbf{match}\ e_0\ \textbf{with}\ [\,]\ \to\ e_1 \mid x :: xs\ \to\ e_2$$
$$\quad | \quad (e_1, e_2) \mid \textbf{match}\ e_0\ \textbf{with}\ (x_1, x_2)\ \to\ e_1$$

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.
- We extend $[\![e]\!]^\sharp\ \rho$ with rules for case-distinction ...