Script generated by TTT

Title: Seidl: Programmoptimierung (23.01.2012)

Date: Mon Jan 23 12:35:17 CET 2012

Duration: 84:59 min

Pages: 34

We have:

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

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

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

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:

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

Analogously, there is index-dependent accumulation:

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

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

$$\begin{aligned}
\textsf{compi} &= \textbf{fun}\ f\ \to\ \textbf{fun}\ g\ \to\ \textbf{fun}\ i\ \to\ \textbf{fun}\ x\ \to\ f\ i\ (g\ i\ x) \\
\textsf{compi}_1 &= \textbf{fun}\ f\ \to\ \textbf{fun}\ g\ \to\ \textbf{fun}\ i\ \to\ \textbf{fun}\ x_1\ \to\ \textbf{fun}\ x_2\ \to \\
&\qquad f\ i\ (g\ i\ x_1)\ x_2 \\
\textsf{compi}_2 &= \textbf{fun}\ f\ \to\ \textbf{fun}\ g\ \to\ \textbf{fun}\ i\ \to\ \textbf{fun}\ x_1\ \to\ \textbf{fun}\ x_2\ \to \\
&\qquad f\ i\ x_1\ (g\ i\ x_2) \\
\textsf{cmp}_1 &= \textbf{fun}\ f\ \to\ \textbf{fun}\ g\ \to\ \textbf{fun}\ i\ \to\ \textbf{fun}\ x_1\ \to\ \textbf{fun}\ x_2\ \to \\
&\qquad f\ i\ x_1\ (g\ x_2) \\
\textsf{cmp}_2 &= \textbf{fun}\ f\ \to\ \textbf{fun}\ g\ \to\ \textbf{fun}\ i\ \to\ \textbf{fun}\ x_1\ \to\ \textbf{fun}\ x_2\ \to \\
&\qquad f\ x_1\ (g\ i\ x_2)
\end{aligned}$$

$$\begin{aligned}
\textsf{compi} &= \textbf{fun}\ f\ \to\ \textbf{fun}\ g\ \to\ \textbf{fun}\ i\ \to\ \textbf{fun}\ x\ \to\ f\ i\ (g\ i\ x) \\
\textsf{compi}_1 &= \textbf{fun}\ f\ \to\ \textbf{fun}\ g\ \to\ \textbf{fun}\ i\ \to\ \textbf{fun}\ x_1\ \to\ \textbf{fun}\ x_2\ \to \\
&\qquad f\ i\ (g\ i\ x_1)\ x_2 \\
\textsf{compi}_2 &= \textbf{fun}\ f\ \to\ \textbf{fun}\ g\ \to\ \textbf{fun}\ i\ \to\ \textbf{fun}\ x_1\ \to\ \textbf{fun}\ x_2\ \to \\
&\qquad f\ i\ x_1\ (g\ i\ x_2) \\
\textsf{cmp}_1 &= \textbf{fun}\ f\ \to\ \textbf{fun}\ g\ \to\ \textbf{fun}\ i\ \to\ \textbf{fun}\ x_1\ \to\ \textbf{fun}\ x_2\ \to \\
&\qquad f\ i\ x_1\ (g\ x_2) \\
\textsf{cmp}_2 &= \textbf{fun}\ f\ \to\ \textbf{fun}\ g\ \to\ \textbf{fun}\ i\ \to\ \textbf{fun}\ x_1\ \to\ \textbf{fun}\ x_2\ \to \\
&\qquad f\ x_1\ (g\ i\ x_2)
\end{aligned}$$

838

Then:

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

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 $\textsf{tree}\ \alpha$.
- These also provide operations map, mapi and foldl, foldli with corresponding rules.
- Further opportunities are opened up by functions to_list and from_list ...

Example

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

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

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

$$
\begin{array}{lcl}
\mathsf{from\_list} & = & \textbf{fun}\ l \to \textbf{match}\ l \\
& & \textbf{with}\ [\,] \to \mathsf{Leaf} \\
& & \mid\ x :: xs \to \mathsf{Node}\ x\ \mathsf{Leaf}\ (\mathsf{from\_list}\ xs)
\end{array}
$$

Warning:

Not every natural equation is valid:

$$
\begin{array}{lcl}
\mathsf{comp\ to\_list\ from\_list} & = & \mathsf{id} \\
\mathsf{comp\ from\_list\ to\_list} & \neq & \mathsf{id} \\
\mathsf{comp\ to\_list}\ (\mathsf{map}\ f) & = & \mathsf{comp}\ (\mathsf{map}\ f)\ \mathsf{to\_list} \\
\mathsf{comp\ from\_list}\ (\mathsf{map}\ f) & = & \mathsf{comp}\ (\mathsf{map}\ f)\ \mathsf{from\_list} \\
\mathsf{comp}\ (\mathsf{foldl}\ f\ a)\ \mathsf{to\_list} & = & \mathsf{foldl}\ f\ a \\
\mathsf{comp}\ (\mathsf{foldl}\ f\ a)\ \mathsf{from\_list} & = & \mathsf{foldl}\ f\ a
\end{array}
$$

In this case, there is even a rev:

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

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

In this case, there is even a rev:

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

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

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

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

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

Then CBN yields:

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

— whereas evaluation with CBV does not terminate !!!

from 0

Example

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

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

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 ???

$$\begin{array}{lll} \text{fac2}' & = \;\textbf{fun}\; x \to \textbf{fun}\; a \to & \textbf{if}\; x \leq 0\; \textbf{then}\; a \\ & & \textbf{else}\;\; \text{fac2}\;(x-1)\;(a \cdot x) \end{array}$$

fac = fun x → fac2' x 1

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 ...

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 ???

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

Discussion:

- The multiplications are collected in the accumulating parameter through nested closures.
- Only when the value of a call $\mathsf{fac2}\ x\ 1$ 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 ...

$$\begin{array}{lll} e & ::= & c \mid x \mid e_1 \,\Box_2\, e_2 \mid \Box_1\, e \mid f\ e_1\ \ldots\ e_k \mid \mathbf{if}\ e_0\ \mathbf{then}\ e_1\ \mathbf{else}\ e_2 \\ & & \mid \mathbf{let}\ r_1 = e_1\ \mathbf{in}\ e \\ r & ::= & x \mid \#x \\ d & ::= & f\ x_1\ \ldots\ x_k = e \\ p & ::= & \mathbf{letrec\ and}\ d_1\ \ldots\ \mathbf{and}\ d_n\ \mathbf{in}\ e \end{array}$$

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 ...

$$
\begin{array}{lll}
e & ::= & c \mid x \mid e_1 \,\Box_2\, e_2 \mid \Box_1\, e \mid f\ e_1\ \ldots\ e_k \mid \textbf{if}\ e_0\ \textbf{then}\ e_1\ \textbf{else}\ e_2 \\
 & & \mid \textbf{let}\ r_1 = e_1\ \textbf{in}\ e \\
r & ::= & x \mid \#x \\
d & ::= & f\ x_1\ \ldots\ x_k = e \\
p & ::= & \textbf{letrec and}\ d_1\ \ldots\ \textbf{and}\ d_n\ \textbf{in}\ e
\end{array}
$$

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:

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

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

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:

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

Fixpoint iteration yields:

0	**fun** $x \to$ **fun** $a \to 0$
1	**fun** $x \to$ **fun** $a \to x \wedge a$
2	**fun** $x \to$ **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:

$$
\begin{aligned}
\mathsf{fac2} \;=\; \textbf{fun}\; x \to \textbf{fun}\; a \to \quad & \textbf{if}\; x \le 0\; \textbf{then}\; a \\
& \textbf{else}\;\; \textbf{let}\;\; \#\,x' \;=\; x - 1 \\
& \qquad\qquad\; \#\,a' \;=\; x \cdot a \\
& \textbf{in}\;\; \mathsf{fac2}\; x'\; a'
\end{aligned}
$$

$[\![\mathsf{fac2}]\!] : \mathsf{int} \to \mathsf{int} \to \mathsf{int}$

$\quad\;\Delta \qquad\quad \Delta \qquad\quad \Delta$

$[\![\mathsf{fac2}]\!]^\sharp : \mathsf{bool} \to \mathsf{bool} \to \mathsf{bool}$

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:

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

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 ...

$$\mathsf{hd} \;=\; \mathbf{fun}\; l \;\to\; \mathbf{match}\; l\; \mathbf{with}\; x :: xs \;\to\; x$$

- 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:

$$\begin{array}{lll} e & ::= & \ldots \mid [\,] \mid e_1 :: e_2 \mid \mathbf{match}\; e_0\; \mathbf{with}\; [\,] \;\to\; e_1 \;\mid\; x :: xs \;\to\; e_2 \\ & & \mid (e_1, e_2) \mid \mathbf{match}\; e_0\; \mathbf{with}\; (x_1, x_2) \;\to\; e_1 \end{array}$$

Top Strictness

$\leadsto$ Total Matchings

- 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 the abstract evaluation $[\![e]\!]^\sharp\,\rho$ with rules for case-distinction ...