Example

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

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

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

$$
\begin{aligned}
\textsf{from\_list} &= \textbf{fun}\, l \to \\
&\quad \textbf{match}\ l\ \textbf{with}\ [\,] \to \textsf{Leaf} \\
&\quad \mid\ :: z \to (\textbf{match}\ z\ \textbf{with}\ (x, xs) \to \\
&\qquad\qquad\quad \textsf{Node}\ x\ \textsf{Leaf}\ (\textsf{from\_list}\ xs))
\end{aligned}
$$

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:

$$
\begin{array}{lcl}
\mathsf{rev} & = & \textbf{fun}\ t\ \rightarrow \\
& & \textbf{match}\ t\ \textbf{with}\ \mathsf{Leaf}\ \rightarrow\ \mathsf{Leaf} \\
& & \mid\ \mathsf{Node}\ x\ t_1\ t_2\ \rightarrow\ \begin{array}[t]{l} \textbf{let}\ \begin{array}[t]{lcl} s_1 & = & \mathsf{rev}\ t_1 \\ s_2 & = & \mathsf{rev}\ t_2 \end{array} \\ \textbf{in}\ \mathsf{Node}\ x\ s_2\ s_1 \end{array}
\end{array}
$$

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

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 ...

Example

$$\mathsf{from} \quad = \quad \mathbf{fun}\ n \ \rightarrow \quad n :: \mathsf{from}\ (n+1)$$

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

Then CBN yields:

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

— whereas evaluation with CBV does not terminate !!!

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 ???

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

Discussion:

- The multiplications are collected in the accumulating parameter through nested closures.
- Only when the value of a call 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 ...

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

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:

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

$$[\![\textbf{let } x_1 = e_1 \textbf{ in } e]\!]^\sharp\,\rho \quad = \quad [\![e]\!]^\sharp\,(\rho \oplus \{x_1 \mapsto [\![e_1]\!]^\sharp\,\rho\})$$

$$[\![\textbf{let } \#x_1 = e_1 \textbf{ in } e]\!]^\sharp\,\rho \quad = \quad ([\![e_1]\!]^\sharp\,\rho) \wedge ([\![e]\!]^\sharp\,(\rho \oplus \{x_1 \mapsto 1\}))$$

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:

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

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:

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

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:

$$\perp \Delta 0 \quad \text{und} \quad z \Delta 1 \quad \text{für } z \in \mathbb{Z}$$

Extension: Data-Structures

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

$$\mathsf{hd} \;=\; \textbf{fun}\ l \;\to\; \textbf{match}\ l\ \textbf{with}\ ::z \;\to\; \textbf{match}\ z\ \textbf{with}\ (x, xs) \;\to\; x$$

- 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 ...