Script generated by TTT

Title: Seidl: Programmoptimierung (28.01.2013)

Date: Mon Jan 28 14:00:31 CET 2013

Duration: 90:52 min

Pages: 41

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:

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

Analogously, there is index-dependent accumulation:

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

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

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

Then:

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

Then:

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

838

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:

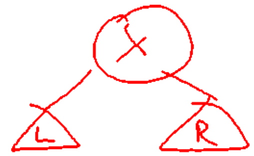
$$\begin{array}{lcl}
\text{comp}\,(\text{mapi}\ f)\ (\text{map}\ g) & = & \text{mapi}\,(\text{comp}_2\ f\ g) \\
\text{comp}\,(\text{map}\ f)\ (\text{mapi}\ g) & = & \text{mapi}\,(\text{comp}\ f\ g) \\
\text{comp}\,(\text{mapi}\ f)\ (\text{mapi}\ g) & = & \text{mapi}\,(\text{compi}\ f\ g) \\
\text{comp}\,(\text{foldli}\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

838

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 ...

Example



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

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

Warning:

Not every natural equation is valid:

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

In this case, there is even a rev:

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

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

In this case, there is even a rev:

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

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

843

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

$$
\mathsf{from} \;=\; \mathbf{fun}\; n \;\to\; \boxed{n ::}\,\boxed{\mathsf{from}\; (n+1)}
$$

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

Then CBN yields:

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

— whereas evaluation with CBV does not terminate !!!

Then CBN yields:

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

— whereas evaluation with CBV does not terminate !!!

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

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

$\boxed{x \cdot a}$

Then CBN yields:

fac = fun x → if x ≤ 0 then 1 else x · f (x - 1)

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

— whereas evaluation with CBV does not terminate !!!

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

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

$x \cdot x$ $\circ\ x - 1$

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:

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

— whereas evaluation with CBV does not terminate !!!

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

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

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 ...

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 $\mathcal{O}(2^k)$:-(

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

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

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:

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

Fixpoint iteration yields:

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

Example:

For fac2, we obtain:

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

$b_1 \wedge b_2$

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:

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

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:

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

fac = fun x -> if x ≤ 0 then 1
else $\boxed{x \cdot f(x-1)}$

$\boxed{\boxed{\boxed{2 \cdot x} \circ (x-1)} \circ x-2}$

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:

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

856

-3 -2 -1 0 1 2 3

Idea (cont.):

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

Auxiliary Function:

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

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

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 $\mathcal{O}(2^k)$:-(

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:

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

$\gamma\, 0 = \{\perp\}$

$\gamma(1) = \mathbb{Z}_\perp$

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:

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

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:

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

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

$$[\![\textbf{match } e_0 \textbf{ with } [\,] \;\to\; e_1 \;\mid\; x :: xs \;\to\; e_2]\!]^\sharp\,\rho \quad =$$
$$[\![e_0]\!]^\sharp\,\rho \wedge ([\![e_1]\!]^\sharp\,\rho \vee [\![e_2]\!]^\sharp\,(\rho \oplus \{x, xs \mapsto 1\}))$$
$$[\![\textbf{match } e_0 \textbf{ with } (x_1, x_2) \;\to\; e_1]\!]^\sharp\,\rho \quad =$$
$$[\![e_0]\!]^\sharp\,\rho \wedge [\![e_1]\!]^\sharp\,(\rho \oplus \{x_1, x_2 \mapsto 1\})$$
$$[\![[\,]]\!]^\sharp\,\rho = \boxed{[\![e_1 :: e_2]\!]^\sharp}\,\rho = \boxed{[\![(e_1, e_2)]\!]^\sharp\,\rho} = \boxed{1}$$

- The rules for **match** are analogous to those for **if**.
- In case of ::, we know nothing about the values beneath the constructor; therefore $\{x, xs \mapsto 1\}$.
- We check our analysis on the function `app` ...