Script **generated by TTT**

Title: Seidl: Programmoptimierung (25.01.2012)

Date: Wed Jan 25 12:31:09 CET 2012

Duration: 88:33 min

Pages: 33

Extension of the Syntax:

We additionally consider expression of the form:

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

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 $\llbracket e \rrbracket^\sharp\, \rho$ with rules for case-distinction ...

858

Extension of the Syntax:

We additionally consider expression of the form:

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

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 $\llbracket e \rrbracket^\sharp\, \rho$ with rules for case-distinction ...

858

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

$$\llbracket \textbf{match } e_0 \textbf{ with } (x_1, x_2) \;\to\; e_1 \rrbracket^\sharp\, \rho \quad = \quad \llbracket e_0 \rrbracket^\sharp\, \rho \wedge \llbracket e_1 \rrbracket^\sharp\, (\rho \oplus \{x_1, x_2 \mapsto 1\})$$

$$\llbracket [\,] \rrbracket^\sharp\, \rho \;=\; \llbracket e_1 :: e_2 \rrbracket^\sharp\, \rho \;=\; \llbracket (e_1, e_2) \rrbracket^\sharp\, \rho \quad = \quad 1$$

- 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` ...

Example:

$$\begin{aligned} \mathsf{app} \;=\; & \textbf{fun } x \to \textbf{fun } y \to \textbf{match } x \textbf{ with } [\,] \to y \\ & \quad \mid\; x :: xs \to x :: \mathsf{app}\; xs\; y \end{aligned}$$

Abstract interpretation yields the system of equations:

$$\begin{aligned} [\![\mathsf{app}]\!]^\sharp\, b_1\, b_2 &= b_1 \wedge \cancel{(b_2 \vee 1)} \\ &= b_1 \end{aligned}$$

We conclude that we may conclude for sure only for the first argument that its top constructor is required :-)

Total Strictness

Assume that the result of the function application is totally required. Which arguments then are also totally required ?

We again refer to Boolean functions ...

$$\begin{aligned} [\![\textbf{match } e_0 \textbf{ with } [\,] \to e_1 \mid x, :: xs \to e_2]\!]^\sharp\, \rho &= \textbf{let } b = [\![e_0]\!]^\sharp\, \rho \textbf{ in} \\ b \wedge [\![e_1]\!]^\sharp\, \rho \vee [\![e_2]\!]^\sharp\, (\rho \oplus \{x \mapsto b, xs \mapsto 1\}) &\vee [\![e_2]\!]^\sharp\, (\rho \oplus \{x \mapsto 1, xs \mapsto b\}) \\ [\![\textbf{match } e_0 \textbf{ with } (x_1, x_2) \to e_1]\!]^\sharp\, \rho &= \textbf{let } b = [\![e_0]\!]^\sharp\, \rho \textbf{ in} \\ [\![e_1]\!]^\sharp\, (\rho \oplus \{x_1 \mapsto 1, x_2 \mapsto b\}) &\vee [\![e_1]\!]^\sharp\, (\rho \oplus \{x_1 \mapsto b, x_2 \mapsto 1\}) \\ [\![[\,]]\!]^\sharp\, \rho &= 1 \\ [\![e_1 :: e_2]\!]^\sharp\, \rho &= [\![e_1]\!]^\sharp\, \rho \wedge [\![e_2]\!]^\sharp\, \rho \\ [\![(e_1, e_2)]\!]^\sharp\, \rho &= [\![e_1]\!]^\sharp\, \rho \wedge [\![e_2]\!]^\sharp\, \rho \end{aligned}$$

○ △ ⊥

1 △ ∈

Example:

$$\begin{aligned} \mathsf{app} \;=\; & \textbf{fun } x \to \textbf{fun } y \to \textbf{match } x \textbf{ with } [\,] \to y \\ & \quad \mid\; x :: xs \to x :: \mathsf{app}\; xs\; y \end{aligned}$$

Abstract interpretation yields the system of equations:

$$\begin{aligned} [\![\mathsf{app}]\!]^\sharp\, b_1\, b_2 &= b_1 \wedge (b_2 \vee 1) \\ &= b_1 \end{aligned}$$

We conclude that we may conclude for sure only for the first argument that its top constructor is required :-)

○ △ 1 :: [ ]

Total Strictness

Assume that the result of the function application is totally required. Which arguments then are also totally required ?

We again refer to Boolean functions ...

$$\begin{aligned} [\![\textbf{match } e_0 \textbf{ with } [\,] \to e_1 \mid x, :: xs \to e_2]\!]^\sharp\, \rho &= \textbf{let } b = [\![e_0]\!]^\sharp\, \rho \textbf{ in} \\ b \wedge [\![e_1]\!]^\sharp\, \rho \vee [\![e_2]\!]^\sharp\, (\rho \oplus \{x \mapsto b, xs \mapsto 1\}) &\vee [\![e_2]\!]^\sharp\, (\rho \oplus \{x \mapsto 1, xs \mapsto b\}) \\ [\![\textbf{match } e_0 \textbf{ with } (x_1, x_2) \to e_1]\!]^\sharp\, \rho &= \textbf{let } b = [\![e_0]\!]^\sharp\, \rho \textbf{ in} \\ [\![e_1]\!]^\sharp\, (\rho \oplus \{x_1 \mapsto 1, x_2 \mapsto b\}) &\vee [\![e_1]\!]^\sharp\, (\rho \oplus \{x_1 \mapsto b, x_2 \mapsto 1\}) \\ [\![[\,]]\!]^\sharp\, \rho &= 1 \\ [\![e_1 :: e_2]\!]^\sharp\, \rho &= [\![e_1]\!]^\sharp\, \rho \wedge [\![e_2]\!]^\sharp\, \rho \\ [\![(e_1, e_2)]\!]^\sharp\, \rho &= [\![e_1]\!]^\sharp\, \rho \wedge [\![e_2]\!]^\sharp\, \rho \end{aligned}$$

Total Strictness

Assume that the result of the function application is totally required.
Which arguments then are also totally required ?

We again refer to Boolean functions ...

$$
\begin{array}{lll}
\llbracket \textbf{match } e_0 \textbf{ with } [\,] \;\to\; e_1 \;\mid\; x, :: xs \;\to\; e_2 \rrbracket^\sharp\,\rho & = & \textbf{let } b = \llbracket e_0 \rrbracket^\sharp\,\rho \textbf{ in} \\
\multicolumn{3}{l}{b \wedge \llbracket e_1 \rrbracket^\sharp\,\rho \vee \llbracket e_2 \rrbracket^\sharp\,(\rho \oplus \{x \mapsto b, xs \mapsto 1\}) \vee \llbracket e_2 \rrbracket^\sharp\,(\rho \oplus \{x \mapsto 1, xs \mapsto b\})} \\
\llbracket \textbf{match } e_0 \textbf{ with } (x_1, x_2) \;\to\; e_1 \rrbracket^\sharp\,\rho & = & \textbf{let } b = \llbracket e_0 \rrbracket^\sharp\,\rho \textbf{ in} \\
\multicolumn{3}{r}{\llbracket e_1 \rrbracket^\sharp\,(\rho \oplus \{x_1 \mapsto 1, x_2 \mapsto b\}) \vee \llbracket e_1 \rrbracket^\sharp\,(\rho \oplus \{x_1 \mapsto b, x_2 \mapsto 1\})} \\
\llbracket [\,] \rrbracket^\sharp\,\rho & = & 1 \\
\llbracket e_1 :: e_2 \rrbracket^\sharp\,\rho & = & \llbracket e_1 \rrbracket^\sharp\,\rho \wedge \llbracket e_2 \rrbracket^\sharp\,\rho \\
\llbracket (e_1, e_2) \rrbracket^\sharp\,\rho & = & \llbracket e_1 \rrbracket^\sharp\,\rho \wedge \llbracket e_2 \rrbracket^\sharp\,\rho
\end{array}
$$

Discussion:

- The rules for constructor applications have changed.
- Also the treatment of **match** now involves the components z and x_1, x_2.
- Again, we check the approach for the function `app`.

Example:

Abstract interpretation yields the system of equations:

$$
\begin{array}{lll}
\llbracket \texttt{app} \rrbracket^\sharp\; b_1\; b_2 & = & b_1 \wedge b_2 \vee b_1 \wedge \llbracket \texttt{app} \rrbracket^\sharp\; 1\; b_2 \vee 1 \wedge \llbracket \texttt{app} \rrbracket^\sharp\; b_1\; b_2 \\
& = & b_1 \wedge b_2 \vee b_1 \wedge \llbracket \texttt{app} \rrbracket^\sharp\; 1\; b_2 \vee \llbracket \texttt{app} \rrbracket^\sharp\; b_1\; b_2
\end{array}
$$

This results in the following fixpoint iteration:

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

We deduce that both arguments are definitely totally required if the result is totally required :-)

Warning:

Whether or not the result is totally required, depends on the context of the function call!

In such a context, a specialized function may be called ...

Discussion:

- The rules for constructor applications have changed.
- Also the treatment of **match** now involves the components z and x_1, x_2.
- Again, we check the approach for the function `app`.

Example:

Abstract interpretation yields the system of equations:

$$
\begin{array}{lll}
\llbracket \texttt{app} \rrbracket^\sharp\; b_1\; b_2 & = & b_1 \wedge b_2 \vee b_1 \wedge \llbracket \texttt{app} \rrbracket^\sharp\; 1\; b_2 \vee 1 \wedge \llbracket \texttt{app} \rrbracket^\sharp\; b_1\; b_2 \\
& = & b_1 \wedge b_2 \vee b_1 \wedge \underbrace{\llbracket \texttt{app} \rrbracket^\sharp\; 1\; b_2}_{0} \vee \underbrace{\llbracket \texttt{app} \rrbracket^\sharp\; b_1\; b_2}_{0}
\end{array}
$$

This results in the following fixpoint iteration:

0	**fun** $x \to$ **fun** $y \to 0$
1	**fun** $x \to$ **fun** $y \to x \wedge y$
2	**fun** $x \to$ **fun** $y \to x \wedge y$

We deduce that both arguments are definitely totally required if the result is totally required :-)

Warning:

Whether or not the result is totally required, depends on the context of the function call!

In such a context, a specialized function may be called ...

Discussion:

- The rules for constructor applications have changed.
- Also the treatment of **match** now involves the components z and x_1, x_2.
- Again, we check the approach for the function app.

Example:

Abstract interpretation yields the system of equations:

$$\begin{aligned} [\![\mathsf{app}]\!]^\sharp\, b_1\, b_2 &= b_1 \wedge b_2 \vee b_1 \wedge [\![\mathsf{app}]\!]^\sharp\, 1\, b_2 \vee 1 \wedge [\![\mathsf{app}]\!]^\sharp\, b_1\, b_2 \\ &= b_1 \wedge b_2 \vee b_1 \wedge \cancel{[\![\mathsf{app}]\!]^\sharp}\, 1 \wedge b_2 \vee \cancel{[\![\mathsf{app}]\!]^\sharp}\, b_1 \wedge b_2 \end{aligned}$$

$b_1 \wedge b_2$ $b_1 \wedge b_2$

This results in the following fixpoint iteration:

0	**fun** $x \to$ **fun** $y \to 0$
1	**fun** $x \to$ **fun** $y \to x \wedge y$
2	**fun** $x \to$ **fun** $y \to x \wedge y$

We deduce that both arguments are definitely totally required if the result is totally required :-)

Warning:

Whether or not the result is totally required, depends on the context of the function call!

In such a context, a specialized function may be called ...

$$\begin{aligned} \mathsf{app\#} \;=\; \textbf{fun}\ x \to \textbf{fun}\ y \to\ & \textbf{let}\ \#x' = x\ \textbf{and}\ \#y' = y\ \textbf{in} \\ & \textbf{match}\ 'x\ \textbf{with}\ [\,] \to y' \\ & \quad \mid\ x :: xs \to \textbf{let}\ \#\, r = x :: \mathsf{app\#}\ xs\ y \\ & \qquad\qquad\qquad \textbf{in}\ r \end{aligned}$$

Discussion:

empty Δ 2

- Both strictness analyses employ the same complete lattice.
- Results and application, though, are quite different :-)

2 Δ 1

- Thereby, we use the following description relations:

  Top Strictness : $\perp \,\Delta\, 0$

  $\perp \,\Delta\, 0$

  Total Strictness : $z \,\Delta\, 0$ if $\perp$ occurs in z.
- Both analyses can also be combined to an a joint analysis ...

Combined Strictness Analysis

- We use the complete lattice:

$$\mathbb{T} = \{0 \sqsubset 1 \sqsubset 2\}$$

- The description relation is given by:

$$\bot \,\Delta\, 0 \qquad z \,\Delta\, 1 \; (z \text{ contains } \bot) \qquad z \,\Delta\, 2 \; (z \text{ value})$$

- The lattice is more informative, the functions, though, are no longer as efficiently representable, e.g., through Boolean expressions :-(
- We require the auxiliary functions:

$$(i \sqsubseteq x);\; y = \begin{cases} y & \text{if } i \sqsubseteq x \\ 0 & \text{otherwise} \end{cases}$$

The Combined Evaluation Function:

$$\begin{array}{lcl}
\llbracket \mathbf{match}\; e_0 \;\mathbf{with}\; [\,] \;\to\; e_1 \;\mid\; x :: xs \;\to\; e_2 \rrbracket^\sharp\, \rho & = & \mathbf{let}\; b = \llbracket e_0 \rrbracket^\sharp\, \rho \;\mathbf{in} \\
 & & \qquad (2 \sqsubseteq b)\,;\, \llbracket e_1 \rrbracket^\sharp\, \rho \sqcup \\
 & & \qquad (1 \sqsubseteq b)\,;\, (\llbracket e_2 \rrbracket^\sharp\, (\rho \oplus \{x \mapsto 2, xs \mapsto b\}) \\
 & & \qquad \sqcup\, \llbracket e_2 \rrbracket^\sharp\, (\rho \oplus \{x \mapsto b, xs \mapsto 2\})) \\
\llbracket \mathbf{match}\; e_0 \;\mathbf{with}\; (x_1, x_2) \;\to\; e_1 \rrbracket^\sharp\, \rho & = & \mathbf{let}\; b = \llbracket e_0 \rrbracket^\sharp\, \rho \;\mathbf{in} \\
 & & \qquad (1 \sqsubseteq b)\,;\, (\llbracket e_1 \rrbracket^\sharp\, (\rho \oplus \{x_1 \mapsto 2, x_2 \mapsto b\}) \\
 & & \qquad \sqcup\, \llbracket e_1 \rrbracket^\sharp\, (\rho \oplus \{x_1 \mapsto b, x_2 \mapsto 2\})) \\
\llbracket [\,] \rrbracket^\sharp\, \rho & = & 2 \\
\llbracket e_1 :: e_2 \rrbracket^\sharp\, \rho & = & \\
\llbracket (e_1, e_2) \rrbracket^\sharp\, \rho & = & 1 \sqcup (\llbracket e_1 \rrbracket^\sharp\, \rho \sqcap \llbracket e_2 \rrbracket^\sharp\, \rho)
\end{array}$$

Example:

For our beloved function app, we obtain:

$$\begin{array}{lcl}
\llbracket \mathsf{app} \rrbracket^\sharp\, d_1\; d_2 & = & (2 \sqsubseteq d_1)\,;\, d_2 \sqcup \\
 & & (1 \sqsubseteq d_1)\,;\, (1 \sqcup \llbracket \mathsf{app} \rrbracket^\sharp\, d_1\; d_2 \sqcup d_1 \sqcap \llbracket \mathsf{app} \rrbracket^\sharp\, 2\; d_2) \\
 & = & (2 \sqsubseteq d_1)\,;\, d_2 \sqcup \\
 & & (1 \sqsubseteq d_1)\,;\, 1 \sqcup \\
 & & (1 \sqsubseteq d_1)\,;\, \llbracket \mathsf{app} \rrbracket^\sharp\, d_1\; d_2 \sqcup \\
 & & d_1 \sqcap \llbracket \mathsf{app} \rrbracket^\sharp\, 2\; d_2
\end{array}$$

this results in the fixpoint computation:

0	$\mathbf{fun}\; x \to \mathbf{fun}\; y \to\; 0$
1	$\mathbf{fun}\; x \to \mathbf{fun}\; y \to\; (2 \sqsubseteq x);\; y \sqcup (1 \sqsubseteq x);\; 1$
2	$\mathbf{fun}\; x \to \mathbf{fun}\; y \to\; (2 \sqsubseteq x);\; y \sqcup (1 \sqsubseteq x);\; 1$

We conclude

- that both arguments are totally required if the result is totally required; and
- that the root of the first argument is required if the root of the result is required :-)

Remark:

The analysis can be easily generalized such that it guarantees evaluation up to a depth d ;-)

Further Directions:

- Our Approach is also applicable to other data structures.
- In principle, also higher-order (monomorphic) functions can be analyzed in this way :-)
- Then, however, we require higher-order abstract functions — of which there are many :-(
- Such functions therefore are approximated by:

$$\textbf{fun } x_1 \to \ldots \textbf{ fun } x_r \to \top$$

:-)

- For some known higher-order functions such as map, foldl, loop, ... this approach then should be improved :-))

$$\text{map} : (\text{int} \to \text{int}) \to \text{int list}$$
$$\to \text{int list}$$

$$\text{pmap}^\#: \underbrace{\underbrace{\underbrace{(\mathbb{B}^n \to \mathbb{B})}_{\leq 2^2 \times 2} \to \mathbb{B}}_{\leq 8} \to \mathbb{B}}_{\leq 2^8 = 256}$$

Further Directions:

- Our Approach is also applicable to other data structures.
- In principle, also higher-order (monomorphic) functions can be analyzed in this way :-)
- Then, however, we require higher-order abstract functions — of which there are many :-(
- Such functions therefore are approximated by:

$$\textbf{fun } x_1 \to \ldots \textbf{ fun } x_r \to \top$$

:-)

- For some known higher-order functions such as map, foldl, loop, ... this approach then should be improved :-))

5 Optimization of Logic Programs

We only consider the mini language PuP ("Pure Prolog"). In particular, we do not consider:

- arithmetic;
- the cut-operator.
- Self-modification by means of assert and retract.

Example:

$\mathsf{bigger}(X,Y) \leftarrow X = \mathit{elephant}, Y = \mathit{horse}$

$\mathsf{bigger}(X,Y) \leftarrow X = \mathit{horse}, Y = \mathit{donkey}$

$\mathsf{bigger}(X,Y) \leftarrow X = \mathit{donkey}, Y = \mathit{dog}$

$\mathsf{bigger}(X,Y) \leftarrow X = \mathit{donkey}, Y = \mathit{monkey}$

$\mathsf{is\_bigger}(X,Y) \leftarrow \mathsf{bigger}(X,Y)$

$\mathsf{is\_bigger}(X,Y) \leftarrow \mathsf{bigger}(X,Z), \mathsf{is\_bigger}(Z,Y)$

$\leftarrow \mathsf{is\_bigger}(\mathit{elephant}, \mathit{dog})$

?-

A more realistic Example:

$\mathsf{app}(X,Y,Z) \leftarrow X = [\,],\; Y = Z$

$\mathsf{app}(X,Y,Z) \leftarrow X = [H|X'],\; Z = [H|Z'],\; \mathsf{app}(X',Y,Z')$

$\leftarrow \mathsf{app}(X,[Y,c],[a,b,Z])$

A more realistic Example:

$\mathsf{app}(X,Y,Z) \leftarrow X = [\,],\; Y = Z$

$\mathsf{app}(X,Y,Z) \leftarrow X = [H|X'],\; Z = [H|Z'],\; \mathsf{app}(X',Y,Z')$

$\leftarrow \mathsf{app}(X,[Y,c],[a,b,Z])$

Remark:

$[\,]$	$==$	the atom empty list
$[H|Z]$	$==$	binary constructor application
$[a,b,Z]$	$==$	Abbreviation for: $[a|[b|[Z|[\,]]]]$

A more realistic Example:

$\mathsf{app}(X,Y,Z) \leftarrow X = [\,],\; Y = Z$

$\mathsf{app}(X,Y,Z) \leftarrow X = [H|X'],\; Z = [H|Z'],\; \mathsf{app}(X',Y,Z')$

$\leftarrow \mathsf{app}(X,[Y,c],[a,b,Z])$

Remark:

$[\,]$ $==$ the atom empty list

$[H|Z]$ $==$ binary constructor application

$[a,b,Z]$ $==$ Abbreviation for: $[a|[b|[Z|[\,]]]]$

Accordingly, a program p is constructed as follows:

$$\begin{array}{lcl} t & ::= & a \mid X \mid \_ \mid f(t_1, \ldots, t_n) \\ g & ::= & p(t_1, \ldots, t_k) \mid X = t \\ c & ::= & p(X_1, \ldots, X_k) \leftarrow g_1, \ldots, g_r \\ q & ::= & \leftarrow g_1, \ldots, g_r \\ p & ::= & c_1 \ldots c_m q \end{array}$$

- A term t either is an atom, a (possibly anonymous) variable or a constructor application.
- A goal g either is a literal, i.e., a predicate call, or a unification.
- A clause c consists of a head $p(X_1, \ldots, X_k)$ together with body consisting of a sequence of goals.
- A program consists of a sequence of clauses together with a sequence of goals as query.

Procedural View of PuP-Programs:

literal	$==$	procedure call
predicate	$==$	procedure
definition	$==$	body
term	$==$	value
unification	$==$	basic computation step
binding of variables	$==$	side effect

Warning: Predicate calls ...

- do not return results!
- modify the caller solely through side effects :-)
- may fail. Then, the following definition is tried $\Longrightarrow$ backtracking

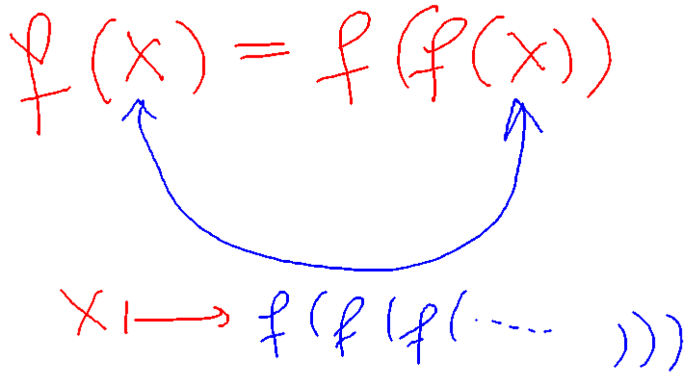
---

Inefficiencies:

Backtracking:
- The matching alternative must be searched for $\Longrightarrow$ Indexing
- Since a successful call may still fail later, the stack can only be cleared if there are no pending alternatives.

Unification:
- The translation possibly must switch between build and check several times.
- In case of unification with a variable, an Occur Check must be performed.

Type Checking:
- Since Prolog is untyped, it must be checked at run-time whether or not a term is of the desired form.
- Otherwise, ugly errors could show up.



$f(X) = f(f(X))$

$X \longmapsto f(f(f(\cdots)))$

Inefficiencies:

Backtracking:
- The matching alternative must be searched for
  ⟹ Indexing
- Since a successful call may still fail later, the stack can only be cleared if there are no pending alternatives.

Unification:
- The translation possibly must switch between build and check several times.
- In case of unification with a variable, an Occur Check must be performed.

Type Checking:
- Since Prolog is untyped, it must be checked at run-time whether or not a term is of the desired form.
- Otherwise, ugly errors could show up.

Some Optimizations:

- Replacing last calls with jumps;
- Compile-time type inference;
- Identification of deterministic predicates ...

Example:

$$\begin{array}{lcl} \mathsf{app}(X, Y, Z) & \leftarrow & X = [\,],\; Y = Z \\ \mathsf{app}(X, Y, Z) & \leftarrow & X = [H|X'],\; Z = [H|Z'],\; \boxed{\mathsf{app}(X', Y, Z')} \\ & \leftarrow & \mathsf{app}([a, b], [Y, c], Z) \end{array}$$