Extension of the Syntax:

We additionally consider expression of the form:

\[ e ::= \ldots \mid [] \mid e_1 :: e_2 \mid \text{match } e_0 \text{ with } [] \rightarrow e_1 \mid x :: xs \rightarrow e_2 \]
\[ \mid (e_1, e_2) \mid \text{match } e_0 \text{ with } (x_1, x_2) \rightarrow e_1 \]

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 \(\triangleright\).
- We extend the abstract evaluation \(\llbracket e \rrbracket^I\rho\) with rules for case-distinction ...

\[
\llbracket \text{match } e_0 \text{ with } [] \rightarrow e_1 \mid x :: xs \rightarrow e_2 \rrbracket^I\rho = \\
\llbracket e_0 \rrbracket^I\rho \land (\llbracket e_1 \rrbracket^I\rho \lor \llbracket e_2 \rrbracket^I(\rho \oplus \{x :: x, x \mapsto 1\}))
\]
\[
\llbracket \text{match } e_0 \text{ with } (x_1, x_2) \rightarrow e_1 \rrbracket^I\rho = \\
\llbracket e_0 \rrbracket^I\rho \land (\llbracket e_1 \rrbracket^I(\rho \oplus \{x_1, x_2 \mapsto 1\}))
\]
\[
\llbracket [] \rrbracket^I\rho = [e_1 :: e_2]^I\rho = [(e_1, e_2)]^I\rho = 1
\]

- The rules for match are analogous to those for if.
- In case of ::, we know nothing about the values beneath the constructor; therefore \(\llbracket x :: x, x \mapsto 1 \rrbracket\).
- We check our analysis on the function app ...
Example:

\[ \text{app} = \text{fun } x \to \text{fun } y \to \text{match } x \text{ with } [] \to y \]

| \[x::xs \to x::\text{app} \, xs \, y\]

Abstract interpretation yields the system of equations:

\[ [\text{app}^2] \, b_1 \, b_2 = b_1 \land (b_2 \lor 1) \]

= \[b_1\]

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

\(\text{:-)}\)

Example:

\[ \text{app} = \text{fun } x \to \text{fun } y \to \text{match } x \text{ with } [] \to y \]

| \[x::xs \to x::\text{app} \, xs \, y\]

Abstract interpretation yields the system of equations:

\[ [\text{app}^2] \, b_1 \, b_2 = b_1 \land (b_2 \lor 1) \]

= \[b_1\]

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

\(\text{:-)}\)

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 ...

\[ [\text{match } e_0 \text{ with } [] \to e_1 \mid x::xs \to e_2]^P \, \rho = \text{let } b = [e_0]^P \, \rho \text{ in} \]

\[ b \land [e_1]^P \, \rho \lor [e_2]^P \, \rho (\rho \oplus \{ x \mapsto b, x_1 \mapsto 1 \}) \lor [e_2]^P \, (\rho \oplus \{ x \mapsto 1, x_1 \mapsto b \}) \]

\[ [\text{match } e_0 \text{ with } (x_1, x_2) \to e_1]^P \, \rho = \text{let } b = [e_0]^P \, \rho \text{ in} \]

\[ [e_1]^P \, (\rho \oplus \{ x_1 \mapsto 1, x_2 \mapsto 1 \}) \lor [e_2]^P \, (\rho \oplus \{ x_1 \mapsto 1, x_2 \mapsto 1 \}) \]

\[ [\text{[[]]}^P \, \rho = 1 \]

\[ [e_1::e_2]^P \, \rho = [e_1]^P \, \rho \land [e_2]^P \, \rho \]

\[ [\{e_1, e_2\}]^P \, \rho = [e_1]^P \, \rho \land [e_2]^P \, \rho \]

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 ...

\[ [\text{match } e_0 \text{ with } [] \to e_1 \mid x::xs \to e_2]^P \, \rho = \text{let } b = [e_0]^P \, \rho \text{ in} \]

\[ b \land [e_1]^P \, \rho \lor [e_2]^P \, \rho (\rho \oplus \{ x \mapsto b, x_1 \mapsto 1 \}) \lor [e_2]^P \, (\rho \oplus \{ x \mapsto 1, x_1 \mapsto b \}) \]

\[ [\text{match } e_0 \text{ with } (x_1, x_2) \to e_1]^P \, \rho = \text{let } b = [e_0]^P \, \rho \text{ in} \]

\[ [e_1]^P \, (\rho \oplus \{ x_1 \mapsto 1, x_2 \mapsto 1 \}) \lor [e_2]^P \, (\rho \oplus \{ x_1 \mapsto 1, x_2 \mapsto 1 \}) \]

\[ [\text{[[]]}^P \, \rho = 1 \]

\[ [e_1::e_2]^P \, \rho = [e_1]^P \, \rho \land [e_2]^P \, \rho \]

\[ [\{e_1, e_2\}]^P \, \rho = [e_1]^P \, \rho \land [e_2]^P \, \rho \]
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...

\[
\begin{align*}
\text{match } e_0 \text{ with } [ ] & \rightarrow e_1 \mid x :: xs \rightarrow e_2 \end{align*}
\]

\[
\begin{align*}
\text{let } b = [e_0] \rho \text{ in } \\
& b \land [e_1] \rho \lor [e_2] \rho (\rho \oplus \{x \mapsto b, xs \mapsto 1\}) \lor [e_2] \rho (\rho \oplus \{x \mapsto 1, xs \mapsto b\}) \\
\text{match } e_0 \text{ with } (x_1, x_2) & \rightarrow e_1 \end{align*}
\]

\[
\begin{align*}
[\square] \rho &= 1 \\
[e_1] \rho &= [e_1] \rho \land [e_2] \rho \\
(e_1, e_2) \rho &= [e_1] \rho \land [e_2] \rho
\end{align*}
\]

This results in the following fixpoint iteration:

\[
\begin{array}{c}
0 \quad \text{fun } x \rightarrow \text{fun } y \rightarrow 0 \\
1 \quad \text{fun } x \rightarrow \text{fun } y \rightarrow x \land y \\
2 \quad \text{fun } x \rightarrow \text{fun } y \rightarrow x \land y
\end{array}
\]

We deduce that both arguments are definitely totally required if the result is totally required.

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 \text{app}.

Example:

Abstract interpretation yields the system of equations:

\[
\begin{align*}
[\text{app}] b_1 b_2 &= b_1 \land b_2 \lor b_1 \land [\text{app}] 1 b_2 \lor 1 \land [\text{app}] b_1 b_2 \\
&= b_1 \land b_2 \lor b_1 \land [\text{app}] 1 b_2 \lor [\text{app}] b_1 b_2
\end{align*}
\]

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 \text{app}.

Example:

Abstract interpretation yields the system of equations:

\[
\begin{align*}
[\text{app}] b_1 b_2 &= b_1 \land b_2 \lor b_1 \land [\text{app}] 1 b_2 \lor 1 \land [\text{app}] b_1 b_2 \\
&= b_1 \land b_2 \lor b_1 \land [\text{app}] 1 b_2 \lor [\text{app}] b_1 b_2
\end{align*}
\]
This results in the following fixpoint iteration:

\[
\begin{array}{c|c|c}
0 & \text{fun } x \rightarrow \text{fun } y \rightarrow 0 & \\
1 & \text{fun } x \rightarrow \text{fun } y \rightarrow x \land y & \\
2 & \text{fun } x \rightarrow \text{fun } y \rightarrow x \land y & \\
\end{array}
\]

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 ...

---

This results in the following fixpoint iteration:

\[
\begin{array}{c|c|c}
0 & \text{fun } x \rightarrow \text{fun } y \rightarrow 0 & \\
1 & \text{fun } x \rightarrow \text{fun } y \rightarrow x \land y & \\
2 & \text{fun } x \rightarrow \text{fun } y \rightarrow x \land y & \\
\end{array}
\]

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:

\[
\begin{align*}
\text{[app]}^x b_1 b_2 &= b_1 \land b_2 \lor b_1 \land \text{[app]}^x 1 b_2 \lor 1 \land \text{[app]}^x b_1 b_2 \\
&= b_1 \land b_2 \lor b_1 \land 1 \lor 1 \land b_1 b_2
\end{align*}
\]

\[
\text{app}^# = \text{fun } x \rightarrow \text{fun } y \rightarrow \text{let } \#x' = x \text{ and } \#y' = y \text{ in}
\begin{align*}
\text{match } 'x \text{ with } [1] & \rightarrow y' \\
| x::xs & \rightarrow \text{let } r = x::\text{app}^# xs y \text{ in } r
\end{align*}
\]

**Discussion:**

- Both strictness analyses employ the same complete lattice.
- Results and application, though, are quite different.
- Thereby, we use the following description relations:
  - Top Strictness : \( \bot \Downarrow 0 \)
  - Total Strictness : \( z \Downarrow 0 \) if \( \bot \) occurs in \( z \).
- Both analyses can also be combined to an a joint analysis ...
Combined Strictness Analysis

- We use the complete lattice:
  \[ T = \{ 0 \sqsubset 1 \sqsubset 2 \} \]

- The description relation is given by:
  \[ \bot \sqsubset 0 \quad z \sqsubset 1 \quad (z \text{ contains } \bot) \quad z \sqsubset 2 \quad (z \text{ value}) \]

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

- We require the auxiliary functions:
  \[ (i \sqsubset x) \quad y \begin{cases} 
  y & \text{if } i \sqsubset x \\
  0 & \text{otherwise} 
\end{cases} \]

Example:

For our beloved function \texttt{app}, we obtain:

\[
\begin{align*}
\text{[app]}^2 \, d_1 \, d_2 & = (2 \sqsubset d_1) \sqcup (1 \sqsubset d_1) \sqcup (1 \sqcup \text{[app]}^2 \, d_1 \, d_2 \sqcup \text{[app]}^2 \, 2 \, d_2) \\
& = (2 \sqsubset d_1) \sqcup (1 \sqsubset d_1) \sqcup (1 \sqcup \text{[app]}^2 \, d_1 \, d_2 \sqcup d_1 \sqcup \text{[app]}^2 \, 2 \, d_2)
\end{align*}
\]

d this results in the fixpoint computation:

The Combined Evaluation Function:

\[
\begin{align*}
\text{[match } & c_0 \text{ with } \mid ] \rightarrow e_1 \mid x :: x \rightarrow e_2 \right] \rho & = \text{ let } b = [c_0]^2 \rho \text{ in } \\
& (2 \sqsubset b) ; [e_1]^2 \rho \sqcup \\
& (1 \sqsubset b) ; ([e_2]^2 (\rho \sqcup \{ x \rightarrow 2, x_1 \rightarrow b \}) \\
& \quad \sqcup [e_2]^2 (\rho \sqcup \{ x \rightarrow b, x_2 \rightarrow 2 \}))
\end{align*}
\]

\[
\begin{align*}
\text{[match } & c_0 \text{ with } (x_1, x_2) \rightarrow e_3 \right] \rho & = \text{ let } b = [c_0]^2 \rho \text{ in } \\
& (1 \sqsubset b) ; ([e_1]^2 (\rho \sqcup \{ x_1 \rightarrow 2, x_2 \rightarrow b \}) \\
& \quad \sqcup [e_1]^2 (\rho \sqcup \{ x_1 \rightarrow b, x_2 \rightarrow 2 \}))
\end{align*}
\]

\[
\begin{align*}
\llbracket \text{ if } \rrbracket \rho & = 2 \\
[e_1 :: e_2]^2 \rho & = \\
[(e_1, e_2)]^2 \rho & = 1 \sqcup ([e_1]^2 \rho \sqcap [e_2]^2 \rho)
\end{align*}
\]

0 \quad \text{fun } x \rightarrow \text{fun } y \rightarrow 0
1 \quad \text{fun } x \rightarrow \text{fun } y \rightarrow (2 \sqsubset x) \sqcup (1 \sqsubset x) ; 1
2 \quad \text{fun } x \rightarrow \text{fun } y \rightarrow (2 \sqsubset x) \sqcup (1 \sqsubset 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
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:
  \[
  \text{fun } x_1 \rightarrow \ldots \text{ fun } x_r \rightarrow T
  \]
  :-) 
- 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:

\[
\begin{align*}
\text{bigger}(X, Y) & \leftarrow X = \text{elephant}, Y = \text{horse} \\
\text{bigger}(X, Y) & \leftarrow X = \text{horse}, Y = \text{donkey} \\
\text{bigger}(X, Y) & \leftarrow X = \text{donkey}, Y = \text{jack} \\
\text{is_bigger}(X, Y) & \leftarrow X = \text{donkey}, Y = \text{monkey} \\
\text{is_bigger}(X, Y) & \leftarrow \text{bigger}(X, Z), \text{is_bigger}(Z, Y) \\
\text{is_bigger}(X, Y) & \leftarrow \text{bigger}(\text{elephant}, \text{dog})
\end{align*}
\]

A more realistic Example:

\[
\begin{align*}
\text{app}(X, Y, Z) & \leftarrow X = \text{[]}, Y = Z \\
\text{app}(X, Y, Z) & \leftarrow X = [H\,X'], Z = [H\,Z'], \text{app}(X', Y, Z') \\
& \leftarrow \text{app}(X, [Y, c], [a, b, Z])
\end{align*}
\]

Remark:

\[
\begin{align*}
\text{[]} & \quad \text{the atom empty list} \\
[H\,Z] & \quad \text{binary constructor application} \\
[a, b, Z] & \quad \text{Abbreviation for: } [a][b][Z]\,]]]
\end{align*}
\]
Accordingly, a program $p$ is constructed as follows:

\[
\begin{align*}
t &::= a | X | f(t_1, \ldots, t_n) \\
g &::= p(t_1, \ldots, t_k) | X = t \\
c &::= p(X_1, \ldots, X_k) \leftarrow g_1, \ldots, g_r \\
q &::= g_1, \ldots, g_r \\
p &::= c_1 \ldots c_m \overline{q}
\end{align*}
\]

- 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.

### 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.

### Procedural View of PuP-Programs:

- literal $\quad:\quad procedure$ call
- predicate $\quad:\quad procedure$
- definition $\quad:\quad body$
- term $\quad:\quad value$
- unification $\quad:\quad basic$ computation step
- binding of variables $\quad:\quad side$ effect

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

\[
\text{app}(X, Y, Z) \leftarrow X = [\text{?}], Y = Z \\
\text{app}(X, Y', Z) \leftarrow X = [H|X'], Z = [H|Z], \text{app}(X', Y', Z') \\
\text{app}([a, b], [Y, c], Z)
\]