Extension of the Syntax:

We additionally consider expression of the form:

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

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 \texttt{int}-values, this coincides with strictness :-)
- We extend the abstract evaluation \([e]# \rho\) with rules for case-distinction ...
\[
\text{match } e_0 \text{ with } [] \rightarrow e_1 \mid x :: xs \rightarrow e_2 \rceil \rho = \\
[e_0] \rho \land ([e_1] \rho \lor [e_2] \rho (\rho \oplus \{x, xs \mapsto 1\}))
\]

\[
\text{match } e_0 \text{ with } (x_1, x_2) \rightarrow e_1 \rceil \rho = \\
[e_0] \rho \land [e_1] \rho (\rho \oplus \{x_1, x_2 \mapsto 1\})
\]

\[
[[ ]] \rho = [e_1 :: e_2] \rceil \rho = [(e_1, e_2)] \rho = 1
\]

- The rules for \textbf{match} are analogous to those for \textbf{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 \textbf{app} ...
Example:

\[
\text{app} = \text{fun } x \rightarrow \text{fun } y \rightarrow \text{match } x \text{ with } [ ] \rightarrow y \\
| x :: xs \rightarrow x :: \text{app } xs \ y
\]

Abstract interpretation yields the system of equations:

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

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

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

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

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

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

|   | fun x → fun y → 0
|---|------------------|
| 0 | fun x → fun y → x ∧ y
| 1 | fun x → fun y → x ∧ y
| 2 |

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 ...
app# = fun x → fun y → let #x' = x and #y' = y in
    match 'x with [] → y'
    | x :: xs → let #r = x :: app# xs y
        in r

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 : ⊥ ∆ 0
  Total Strictness : z ∆ 0 if ⊥ occurs in z.

• Both analyses can also be combined to an a joint analysis ...
Combined Strictness Analysis

- We use the complete lattice:

\[ \mathbb{T} = \{ 0 \sqsubseteq 1 \sqsubseteq 2 \} \]

- The description relation is given by:

\[ \perp \Delta 0 \quad z \Delta 1 \ (z \text{ contains } \perp) \quad z \Delta 2 \ (z \text{ value}) \]

- 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:

\[
\begin{align*}
\text{[match } e_0 \text{ with } [ ] & \rightarrow e_1 \mid x::xs \rightarrow e_2] \# \rho = \quad \text{let } b = [e_0] \# \rho \text{ in} \\
&(2 \sqsubseteq b); [e_1] \# \rho \sqcup \\
&(1 \sqsubseteq b); (\{e_2\} \ (\rho \oplus \{x \mapsto 2, xs \mapsto b\})) \\
&\sqcup [e_2] \ (\rho \oplus \{x \mapsto b, xs \mapsto 2\}) \\
\text{[match } e_0 \text{ with } (x_1, x_2) & \rightarrow e_1] \# \rho = \text{let } b = [e_0] \# \rho \text{ in} \\
&(1 \sqsubseteq b); (\{e_1\} \ (\rho \oplus \{x_1 \mapsto 2, x_2 \mapsto b\})) \\
&\sqcup [e_1] \ (\rho \oplus \{x_1 \mapsto b, x_2 \mapsto 2\}) \\
\text{[[ ]]} \ # \rho &= 2 \\
[e_1:: e_2] \ # \rho &= \\
[(e_1, e_2)] \ # \rho &= 1 \sqcup ([e_1] \ # \rho \sqcap [e_2] \ # \rho)
\end{align*}
\]
Example:

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

$$
\begin{align*}
[\text{app}]^\# \ d_1 \ d_2 &= (2 \sqsubseteq d_1) \; ; \; d_2 \sqcup \\
&\quad (1 \sqsubseteq d_1) \; ; \; (1 \sqcup [\text{app}]^\# \ d_1 \ d_2 \sqcup d_1 \sqcap [\text{app}]^\# \ 2 \ d_2) \\
&= (2 \sqsubseteq d_1) \; ; \; d_2 \sqcup \\
&\quad (1 \sqsubseteq d_1) \; ; \; 1 \sqcup \\
&\quad (1 \sqsubseteq d_1) \; ; \; [\text{app}]^\# \ d_1 \ d_2 \sqcup \\
&\quad d_1 \sqcap [\text{app}]^\# \ 2 \ d_2
\end{align*}
$$

this results in the fixpoint computation:
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 $d$.

868
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:
  \[
  \text{fun } x_1 \rightarrow \ldots \text{fun } x_r \rightarrow \top 
  \]
  
  :)  
- 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:

\[
\begin{align*}
bigger(X, Y) & \leftarrow X = \text{elephant}, Y = \text{horse} \\
bigger(X, Y) & \leftarrow X = \text{horse}, Y = \text{donkey} \\
bigger(X, Y) & \leftarrow X = \text{donkey}, Y = \text{dog} \\
bigger(X, Y) & \leftarrow X = \text{donkey}, Y = \text{monkey} \\
is\_bigger(X, Y) & \leftarrow bigger(X, Y) \\
is\_bigger(X, Y) & \leftarrow bigger(X, Z), is\_bigger(Z, Y) \\
& \leftarrow is\_bigger(\text{elephant}, \text{dog})
\end{align*}
\]
A more realistic Example:

\[
\text{app}(X, Y, Z) \leftarrow X = [], \ Y = Z
\]

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

\[
\leftarrow \ \text{app}(X, [Y, c], [a, b, Z])
\]
A more realistic Example:

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

Remark:

[ ] \quad \quad \quad \text{the atom empty list}

[H|Z] \quad \quad \quad \text{binary constructor application}

[a, b, Z] \quad \quad \quad \text{Abbreviation for: } [a|[b|[Z|[ ]][]]]
Accordingly, a program \( p \) is constructed as follows:

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

- 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.
Procedural View of PuP-Programs:

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

Warning: Predicate calls ...

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

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