Abductive Logic Programming with Tabled Abduction

Luís Moniz Pereira, Ari Saptawijaya

Centro de Inteligência Artificial (CENTRIA)
Departamento de Informática
Faculdade de Ciências e Tecnologia
Universidade Nova de Lisboa

ICSEA 2012 – Lisbon
20 November 2012
Abduction (1)

- From observed evidence to its best explanation
- Example
  - Beliefs:
    - The shoes are wet if the grass is wet.
    - The grass is wet if the sprinkler was running.
    - The grass is wet if it rained.
  - Observation
    - The shoes are wet.
  - Abducibles:
    - “The sprinkler was running”,
    - “It rained”.
  - Minimal explanations:
    - “The sprinkler was running”, or
    - “It rained”.
Abduction (2)

- Consistent explanations, not necessarily minimal.
- Example
  - Previous beliefs:
    - The shoes are wet if the grass is wet.
    - The grass is wet if the sprinkler was running.
    - The grass is wet if it rained.
  - Plus, new beliefs:
    - The clothes outside are wet if it rained.
    - The clothes are dry.
    - Integrity Constraint (IC):
      No clothes are both dry and wet.
  - Same abducibles: “The sprinkler was running”, “It rained”
  - Satisfying IC + Observation “The shoes are wet”
  - Single Explanation: The sprinkler was running.
Abductive Logic Programming

- Abduction in Logic Programs
- Example (cont’d)
  - Rules:
    - shoes_wet ← grass_wet.
    - grass_wet ← sprinkler_running.
    - grass_wet ← rained.
    - clothes_wet ← rained.
    - clothes_dry.
    - IC: false ← clothes_wet, clothes_dry.
  - Abducibles: sprinkler_running, rained.
  - Query: ?- shoes_wet, not false.
  - Abductive solutions: sprinkler_running
- Applications: diagnosis, decision making, …
Tabled Abduction: Motivation

\[ P_1 : \quad q \leftarrow a. \quad r \leftarrow b, q. \quad p \leftarrow r, q. \]

- Abducibles: \{a, b\}
- Query: \?-q. \quad ?-r. \quad ?-p.
  - Explaining \(q\): \([a]\).
  - Explaining \(r\): recompute \(q\)?
  - Explaining \(p\): recompute \(r\) and \(q\)?
- Adopt \textit{tabling} in LP, for abductive solution reuse
  - Table \([a]\) as solution to \?-q.
- Solutions reuse in distinct context!
  - \?-r: reuse solution \(q\) with context \([b]\), but
  - \?-p: reuse solution \(q\) with \(r\)'s solution \([a, b]\) as its context.

<table>
<thead>
<tr>
<th>Goal</th>
<th>Solutions</th>
</tr>
</thead>
<tbody>
<tr>
<td>(q)</td>
<td>([a])</td>
</tr>
<tr>
<td>(r)</td>
<td>([a, b])</td>
</tr>
<tr>
<td>(p)</td>
<td>([a, b])</td>
</tr>
</tbody>
</table>
Program Transformation: Tabling Solutions

- Table abductive solution entry
  - XSB-Prolog tabling
  - $P_1 : q \leftarrow a. \quad r \leftarrow b, q. \quad p \leftarrow r, q.$
  - Table $q^{ab}/1$, $r^{ab}/1$, and $p^{ab}/1$

  \[ q^{ab}([a]). \]
  \[ r^{ab}(E) \leftarrow q([b], E). \]
  \[ p^{ab}(E) \leftarrow r([ ], T), q(T, E). \]

- Re-uptake context-independent solutions from “ab” tables into different contexts

  \[ q(I, O) \leftarrow q^{ab}(E), prod(O, I, E). \]
  \[ r(I, O) \leftarrow r^{ab}(E), prod(O, I, E). \]
  \[ p(I, O) \leftarrow p^{ab}(E), prod(O, I, E). \]

- $prod/3$: produces consistent output abduction result
Program Transformation: Dealing with “not”

- $P_2 : \quad p \leftarrow a, \neg q. \quad q \leftarrow a, b. \quad q \leftarrow c.$
  - Abductive solutions of $\neg q$: compute first all abductive solutions for $q$, before negate them?
  - Finding solutions incrementally.

- Dual rules via dual transformation
  - Replace default literal $\neg q$ to $\neg_q$
    \[ p^{ab}(E) \leftarrow \neg_q([a], E). \]
  - Provide dual rules: $\neg_q$
    \[ \begin{align*}
        \neg_q(I, O) & \leftarrow \neg_q_1(I, T), \neg_q_2(T, O). \\
        \neg_q_1(I, O) & \leftarrow \neg_a(I, O). \\
        \neg_q_1(I, O) & \leftarrow \neg_b(I, O). \\
        \neg_q_2(I, O) & \leftarrow \neg_c(I, O).
    \end{align*} \]
Program Transformation: Loops

- Mostly employ XSB-Prolog’s tabling to deal with loops.

  \[ P_3 : \quad p \leftarrow q. \quad \neg q \leftarrow p. \]

  - Direct positive loop: \(?- p.\) is correctly answered: ‘no’.
    - Detected via loop between tabled predicates \(p^{ab}\) and \(q^{ab}\).
  - What about query: \(?- \neg p.\)
    - It loops, instead of ‘yes’.

\[
\begin{align*}
\neg p(I,O) & \leftarrow \neg p_1(I,O). & \neg p_1(I,O) & \leftarrow \neg q(I,O). \\
\neg q(I,O) & \leftarrow \neg q_1(I,O). & \neg q_1(I,O) & \leftarrow \neg p(I,O).
\end{align*}
\]

- Detect such loops by maintaining an ancestor list (with just negative “\(\neg\)” literals)

\[
\text{ancestor: } [\quad] \quad [\neg p] \quad [\neg p, \neg q] \quad \text{loop!}
\]

- When a positive literal is called, reset ancestor list to [ ].

- Additionally, negative loops over negation are also handled by the transformation, e.g., programs like

\[ P_4 : \quad p \leftarrow q. \quad q \leftarrow \neg p. \]
Query Transformation

- Add (input and output) abductive contexts
- Conjoin with not false, to meet ICs
  - ICs in program are translated as any other rules.
  - In case no ICs, add not_false(l, l) in the program.
- In case of negative query, made it “positive”.
- Example: ?- not p.
- This query is called as a top goal:

  ?- not_p([], T), not_false(T, O).
Comparison with ABDUAL and NegABDUAL

<table>
<thead>
<tr>
<th>Feature</th>
<th>ABDUAL</th>
<th>NegABDUAL</th>
<th>TABDUAL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tabled solutions reuse</td>
<td>✗</td>
<td>✗</td>
<td>✓</td>
</tr>
<tr>
<td>Dual transformation</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Meta-interpreter</td>
<td>✓</td>
<td>✓</td>
<td>✗</td>
</tr>
<tr>
<td>Programs with variables</td>
<td>✗</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Constructive negation</td>
<td>✗</td>
<td>✓</td>
<td>✗</td>
</tr>
</tbody>
</table>
Conclusions and Future Work

- Addressed the issue of tabling abductive solutions
- Achieved via program transformation
  - Table abductive solutions
  - Deal with negative literals
  - Deal with loops
  - Deal with programs and queries containing variables
- Future work:
  - Perfecting implementation
  - Evaluation TABDUAL
  - Application of TABDUAL
  - Migrating core features into an engine-level
    - Tabling abduction entries
    - Hiding data structures, e.g. the ancestor list
Thank you!