Logic programming beyond Prolog

Maarten van Emden
joint work with Areski Nait Abdallah
Logic programming is based on Kowalski’s *procedural interpretation of logic*. It consists of two components:

1. interpret the Horn clauses of logic as procedures

2. use Prolog as the language in which to express the procedures

We propose that (1) can be useful when the procedural language is not Prolog.
Contents

1. Logic programming and “Elements of Programming”

2. Relational programming

3. Generalized Herbrand interpretations

4. Model-theoretic semantics of relational programs

5. Fixpoint semantics of relational programs
1. Logic programming and EOP

If verification is loosely defined as connecting a formal specification with executable code, then logic programming is a candidate verification method.

A logic program is

1. a logic formula that defines the relation (often a partial function) to be computed

2. compiled to executable code.

(1) would have to play the role of the code’s specification.
Logic formula as specification of sortedness:

\[
\text{sort}(X, Y) \leftarrow \text{perm}(X, Y), \text{ordered}(Y).
\]

(1)

This is also executable in Prolog.

The following shows that logic programming can do better:

\[
\begin{align*}
n &\leftarrow [\], [] \\
\text{sort}(V, W) &\leftarrow \text{split}(V, V0, V1), \\
&\quad \text{sort}(V0, W0), \text{sort}(V1, W1), \\
&\quad \text{merge}(W0, W1, W).
\end{align*}
\]

But this is not acceptable as a specification: it is a problem-solving recipe expressed as a logical formula.
“Elements of Programming” by Stepanov and McJones (2009) (hereinafter “EOP”) picks up where logic programming left off.

EOP: Specification $\rightarrow$ Theorem $\rightarrow$ C++ code

<table>
<thead>
<tr>
<th>EOP</th>
<th>Logic Programming</th>
</tr>
</thead>
<tbody>
<tr>
<td>Spec. linear order, (1)</td>
<td>(1)</td>
</tr>
<tr>
<td>Thm. Merge sort theorem</td>
<td>Prolog Merge sort program</td>
</tr>
<tr>
<td>Code Derived C++ code</td>
<td>ditto</td>
</tr>
</tbody>
</table>

According to EOP, the specification consists of logic axioms for sortedness (similar to (1)) plus the axioms for linear order. Free of algorithmic considerations.

In EOP, the theorem is also stated in logic, and expresses the algorithmic trick. C++ code relates to the theorem; it relates to the axioms only via the theorem.
Advantages of EOP:

1. The axioms are standard, from your algebra textbook. In addition, computing suggests new concepts to be axiomatized.

2. Exploits the abstractness of logic: the axioms in the specification are true of any structure that satisfies the axioms. I.e. in the case of sorting not just linked lists, as in Prolog, but any structure that satisfies the axioms for linear ordering.

3. Keeps the specification free from algorithmic considerations.

Q: What else is there to do? A: *Relational Programming.*
2. Relational Programming

Goal: to bring Theorem closer to Code while maintaining the advantages of EOP.

Method: Modify logic programming.

1. Instead of Prolog as language for expressing procedures, some other procedural language, for example C++.

2. Instead of clauses, classical logic syntax.

3. Instead of Herbrand interpretations, $F$-interpretations with arbitrary universes of discourse. The model-theoretic and fixpoint semantics of logic programs transfer to relational programs.
Example

Theorem as clausal program:

\[
\text{sort}([], []). \\
\text{sort}(V, W) :- \text{split}(V, V_0, V_1), \\
\quad \text{sort}(V_0, W_0), \text{sort}(V_1, W_1), \\
\quad \text{merge}(W_0, W_1, W).
\]

Theorem as relational program:

\[
\forall v, w. \text{sort}(v, w) \leftarrow \\
(v = \text{nil} \land w = \text{nil}) \lor \\
(\exists v_0, v_1, w_0, w_1. \text{split}(v, v_0, v_1) \land \\
\text{sort}(v_0, w_0) \land \text{sort}(v_1, w_1) \land \\
\text{merge}(w_0, w_1, w))
\]
General form of relational program with set $Q$ of predicate symbols:

$$\bigwedge_{q \in Q} \left[ \forall \left[ A_q \leftarrow \bigvee_{r \in R_q} \exists \bigwedge_{s \in S_{qr}} B_{qrs} \right] \right]$$

Explanation of the parts:

<table>
<thead>
<tr>
<th>$B_q$</th>
<th>$\bigvee_{r \in R_q} B_{qr}$</th>
<th>disjunction of alternative procedure bodies</th>
</tr>
</thead>
<tbody>
<tr>
<td>$B_{qr}$</td>
<td>$\exists \bigwedge_{s \in S_{qr}} B_{qrs}$</td>
<td>procedure body</td>
</tr>
</tbody>
</table>

$A_q$ is header shared by alternative procedure bodies.
3. Generalizing Herbrand interpretations

$F$-interpretation: fix domain of discourse $D$ and fix interpretation $F$ for the function symbols. $F$-interpretations only differ in their interpretation of the predicate symbols and can be viewed as vectors of relations over $D$ indexed by predicate symbols.


Possible $F$-interpretation: $D$ is Herbrand universe, $F$ maps $f$ to the function that maps $(t_0,\ldots,t_{|q|-1})$ to (the symbolic term) $f(t_0,\ldots,t_{|q|-1})$.

A more typical $F$-interpretation might have $D$ consisting of the natural numbers with fixed interpretations (the usual ones) for 0, 1, $+$, and $\times$. This would be an $F$-model of the semiring axioms.
4. Model-theoretic semantics of relational programs

Conventional semantics versus denotational semantics


2. Denotational: What is denoted by a formula $E$ with set $V$ of free variables in interpretation $I$ with domain of discourse $D$?

Answer: $M^I(E) := \{ \alpha \in (V \rightarrow D) \mid E \text{ is true in } I \text{ with } \alpha \}$. That is, $M^I(E)$ is a relation; that is, a set of tuples, and the type of the tuples is $V \rightarrow D$.
Denotational semantics needed because relations need to be assigned as meanings to formulas possibly containing free variables. It subsumes conventional semantics, which determines whether a closed formula is true under a given interpretation.

Relational program $P$ has the form: $\forall A_q \leftarrow B_q$ one for each $q \in Q$, where $A_q$ and $B_q$ have free variables.

For all $q \in Q$:

- $A_q$ is the procedure heading (an atom)
- $B_q$ is the procedure body (a disjunction of existentially quantified conjunctions of atoms (tests or procedure calls))
Given a relational program $P$ with set $Q$ of predicate symbols and an $F$-interpretation $I$ with domain $D$.

For all $q \in Q$ we have that $M^I(A_q)$ and $M^I(B_q)$ are relations of type $V_q \to D$, where $V_q$ is the set of free variables of $A_q$ and $B_q$.

Definition: An $F$-interpretation $I$ is a model of $P$ iff

$$M^I(A_q) \supseteq M^I(B_q)$$

for all $q \in Q$.

Theorem: $P$ has a least model.

Proof: The intersection of any non-empty set of models of $P$ is a model of $P$. 

5. Fixpoint semantics

int f(int n) { if (n == 0) return 1; return n*f(n-1); }

\[
\begin{array}{r|cccccc}
  n  & 0 & 1 & 2 & 3 & 4 & \ldots \\
  \bot & \bot & \bot & \bot & \bot & \bot & \ldots \\
T_f(\bot) & 1 & \bot & \bot & \bot & \bot & \ldots \\
T_f^2(\bot) & 1 & 1 & \bot & \bot & \bot & \ldots \\
T_f^3(\bot) & 1 & 1 & 2 & \bot & \bot & \ldots \\
T_f^4(\bot) & 1 & 1 & 2 & 6 & \bot & \ldots \\
\end{array}
\]

The function \( f \) computes \( \lim_{n \to \infty} T_f^n(\bot) \).

This limit is the least fixpoint of \( T_f \):

\[
T_f(\lim_{n \to \infty} T_f^n(\bot)) = \lim_{n \to \infty} T_f^n(\bot)
\]
In Scott’s fixpoint semantics, the mapping $T_p$ of type

$$(\mathcal{N} \hookrightarrow \mathcal{N}) \rightarrow (\mathcal{N} \hookrightarrow \mathcal{N})$$

is computed by the body of the recursively defined program function and acts on partial functions of type $\mathcal{N} \hookrightarrow \mathcal{N}$.

In logic programming, recursively defined procedures compute relations, of which partial functions are a special case. $T_p$ maps relations to relations. The mapping is defined by the bodies of recursively defined procedures. The $q$-th procedure has the form $A_q \leftarrow B_q$.

The body defines the relation $M^I(B_q)$. $T_p$ is defined as the mapping from $I$ to an interpretation that has $M^I(B_q)$ as its $q$-th component.
$T_P$ for relational programs:

Definition: For all $q \in Q$,

$$[T_P(I)]_q := M^I(B_q).$$

Theorem: $T_P(I) \subseteq I$ iff $I$ is a model of $P$.

That is, fixpoint semantics is equivalent to model-theoretic semantics.
Definition: For all $q \in Q$,

$$[T_P(I)]_q := \{ \vec{d} \in D^{\|q\|} \mid B_q \text{ is true in } I \text{ with } \vec{d} \circ (\vec{x})^{-1} \}$$

where $\vec{d}$ is $(d_0, \ldots, d_{|q|-1})$, $\vec{x}$ is $(x_0, \ldots, x_{|q|-1})$, $\circ$ is function composition, and $\vec{x}$ has an inverse because no repeated occurrence of any of the variables.

Think of it as $[T_P(I)]_q := \lambda(x_0, \ldots, x_{|q|-1}).M^I(B_q)$.

Theorem: $T_P(I) \subseteq I$ iff $I$ is a model of $P$.

That is, fixpoint semantics is equivalent to model-theoretic semantics.
7. Future work

1. Transcription of relational programs to C++ is easy enough. However, those who are oppressed by the size and complexity of C++ might be interested in the language resulting from eliminating everything not needed for the transcription of relational programs.

2. Conversely, formal logic was formed a century ago and has, with few exceptions, only been used for theoretical purposes. Even textbooks on abstract algebra give the axioms informally. Logic lacks facilities for writing large formulas in a structured fashion. It may benefit from structuring facilities invented for conventional programming languages.
8. Conclusions

1. Fixpoint and model-theoretic semantics of logic programs with respect to Herbrand interpretations generalize to these semantics for relational programs with respect to $F$-interpretations.

2. Kowalski’s Procedural Interpretation of Logic, has not only procedurally interpreted Horn clauses, but also limited the language for expressing procedures to pure Prolog. This work does interpret Horn clauses as procedures, but leaves open the choice of procedural language.

3. We do not propose to replace Prolog, but to expand the scope of logic programming.
Appendix: Hierarchical relational programs

The set of clauses of a relational program can be decomposed into mutually disjoint recursion clusters. Each of these may import predicates from other clusters. Clusters are partially ordered according to whether one imports from another.

The results obtained above hold for each cluster as a separate relational program where there is a distinction between imported predicates, which do not occur in a left-hand side and those that do.

Henceforth consider only relational programs that do not decompose into recursion clusters. For these it does not matter whether the imported predicates are defined by another relational program: they could be defined by a textbook structure.
Constraint Logic Programming

:- \langle 115.0 \mid R \mid 120.0 \rangle; \text{netw}(A,N,B,R,PL).\text{ with } \\
N = \text{par}(\text{at}(R150), \text{ser}(\text{at}(R500), \text{par}(\text{at}(R100), \text{at}(R250)))) \\
and \\
\langle 117.1 \mid R \mid 119.3 \rangle, \langle 149.9 \mid R150 \mid 150.1 \rangle, \ldots.

Procedural language: Prolog
Signature:
(sorts) reals, networks, integers
(functors) \cdot(\text{item, list}), \text{at}(\text{real}), \text{ser}(\text{network, network}), \text{par}(\text{network, network}), \text{:}(\text{network, integer})
(imported predicates) \text{sum}(\text{real, real, real}), \text{inv}(\text{real}), \langle \text{real} | \text{real} | \text{real} \rangle
(indigenous predicates) \text{netw}(\text{terminal, netw, terminal, real, list}), \text{merge}(\text{list, list, list})
netw(A,at(R),B,R,(r150:1).nil) :- <149.9|R|150.1>;;.
% Similarly for 100, 250, and 500 ohms.
netw(A,ser(N1,N2),C,R,PL)
   :- sum(R1,R2,R);
      netw(A,N1,B,R1,PL1), netw(B,N2,C,R2,PL2),
      merge(PL1,PL2,PL).
netw(A,par(N1,N2),B,R,PL)
   :- inv(R,RR),inv(R1,RR1),inv(R2,RR2), sum(RR1,RR2,RR);
      netw(A,N1,B,R1,PL1), netw(A,N2,B,R2,PL2),
      merge(PL1,PL2,PL).