Applications of Logic:
Declarative/Logic Programming in PROLOG

Mathias Kegelmann
22 January 2002
Overview

1. Introduction
2. Proof Search—Logic Programming
3. Basic PROLOG
4. Example
1. Introduction

Not all programming is based on commands …

- Imperative programming
  - *Java, C, PASCAL, …*

- Object-oriented programming
  - usually based on imperative programming
  - *SMALLTALK, C++, Java, …*

- Functional programming
  - *LISP, SCHEME, ML, HASKELL, …*

- *Logic* programming
  - based on a fragment of predicate logic
  - *PROLOG, …*
Logic programming

“Algorithm = Logic + Control” (Kowalski ’79)

How to express control, if not imperatively?

- program = set of axioms
- computation = constructive proof of a goal statement from the program
Logic Programming

Advantages

• declarative: program says, *what* it does, *not how* it does it

• abstracts away from the *von Neumann architecture*—similar to functional programming

• thus real high level programming

• correctness straightforward compared to imperative programs
Applications

- mainly in AI
- expert systems
- planning
- natural language processing
- ...
2. Proof Search—Logic Programming

- Given a set of axioms \( \Gamma \), find a proof for

\[
\Gamma \vdash \phi.
\]

- Particularly interesting for formulae

\[
\Gamma \vdash \exists x. \phi(x)
\]

as a constructive proof gives us a concrete \( x \).

- Example: “There is a list \( X \) such that the ordering of [2,3,1] gives \( X \).”
Proof Search

Feasibility

- natural deduction
- related approaches: sequent calculus, tableaux, resolution
- for full predicate logic: too complex
A Horn clause

$$\phi \iff \psi_1 \land \psi_2 \land \cdots \land \psi_n$$

has

- declarative reading: $\phi$ is true if $\psi_1$ and $\psi_2 \ldots \psi_n$ are true
- procedural reading: to solve (execute) $\phi$, solve (execute) $\psi_1 \ldots \psi_n$ first.
Control flow induced by Horn clause

\[ \phi \leftarrow \psi_1 \land \psi_2 \land \cdots \land \psi_n \]

- Similar to
  ```
  static f(...)
  {
      p1(...);
      p2(...);
      ...
      pn(...);
  }
  ```
- plus unification and backtracking
Logic Program

• Facts—axioms/database
  – \( \phi \)
  – short for Horn clause \( \phi \Leftarrow \top \)
  – Examples: male(Isaac), father(Abraham, Isaac), . . .

• rules—“proper” Horn clauses
  – \( \phi \Leftarrow \psi_1 \land \psi_2 \land \cdots \land \psi_n \)
  – Example: son(\( x,y \)) \Leftarrow father(y,x) \land male(x) \)
Variables in Horn clauses

- no quantifiers

- (implicitly) universally quantified:

\[ \phi(x_1, \ldots, x_n) \leftarrow \psi_1(x_1, \ldots, x_n) \land \cdots \land \psi_n(x_1, \ldots, x_n) \]

means

\[ \forall x_1 \ldots \forall x_n. (\phi(x_1, \ldots, x_n) \leftarrow \psi_1(x_1, \ldots, x_n) \land \cdots \land \psi_n(x_1, \ldots, x_n)) \]

- consider:

\[ \text{grandfather}(x) \leftarrow \text{father}(x, y) \land \text{parent}(y, z) \]

- variables that occur only on the right of \( \Leftarrow \) have existential meaning:

\[ \forall x. (\text{grandfather}(x) \Leftarrow \exists y. \exists z. \text{father}(x, y) \land \text{parent}(y, z)) \]
3. Basic PROLOG

- Facts:
  - male(isaac).
  - father(abraham, isaac).

- Queries:
  - ?- male(issac).
  - ?- father(haran, X).

- queries are existentially quantified
father(terach, abraham).  male(terach).
father(terach, nachor).  male(abraham).
father(terach, haran).  male(nachor).
father(abraham, isaac).  male(haran).
father(haran, lot).  male(isaac).
father(haran, milcah).  male(lot).
father(haran, yiscah).

mother(sarah, isaac).  female(sarah).
female(milcah).
female(yiscah).
• Rules:

  `son(X, Y) :- father(Y, X), male(X).`
  `son(X, Y) :- mother(Y, X), male(X).`

  `daughter(X, Y) :- father(Y, X), female(X).`
  `daughter(X, Y) :- mother(Y, X), female(X).`

  `grandfather(X) :- father(X, Y), parent(Y, Z).`

  `parent(X, Y) :- father(X, Y).`
  `parent(X, Y) :- mother(X, Y).`

  `grandparent(X, Y) :- parent(X, Z), parent(Z, Y).`
Recursion

ancestor(X, Y) :- parent(X, Y).
ancestor(X, Y) :- parent(X, Z), ancestor(Z, Y).
• Examples
  - \([1, 2, 3]\)
  - []

• Head and tail

\[
cons(x, l) \approx [x \mid l]
\]
**Recursion**

```prolog
member(X, [X|Xs]).
member(X, [Y|Xs]) :- member(X, Xs).

append([], L, L).
append([X|Xs], L, [X|L2]) :- append(Xs, L, L2).
```
4. Example

PROLOG is well suited for solving planning problems.

Wolf-Goat-Cabbage Problem

- How to transport a wolf, a goat and a cabbage from one bank of a river to the other using a tiny boat?
- Only one of them fits in the boat at a time.
- The wolf and the goat, and the goat and the cabbage must never be on the same bank alone.
Wolf-Goat-Cabbage Problem

Representation

- possible bank positions are empty, w, wg, wc, wgc, g, gc and c
- predicate cross(with, from, to, new from bank, new to bank)
- PROLOG code:

  cross(wolf, wg, c, g, wc).
  cross(wolf, wc, g, c, wg).
  cross(goat, wg, c, w, gc).
  cross(goat, wgc, empty, wc, g).
  cross(goat, g, wc, empty, wgc).
  cross(goat, gc, w, c, wg).
  cross(cabbage, wc, g, w, gc).
  cross(cabbage, gc, w, g, wc).
  cross(alone, empty, wgc, empty, wgc).
  cross(alone, wc, g, wc, g).
  cross(alone, g, wc, g, wc).
Representing moves

- predicate pos(boat position, left bank, right bank)
- predicate move(position, passenger, new position)
- PROLOG code:

  move(pos(left, Left, Right), Passenger, pos(right, L1, R1)) :-
    cross(Passenger, Left, Right, L1, R1).
  move(pos(right, Left, Right), Passenger, pos(left, L1, R1)) :-
    cross(Passenger, Right, Left, R1, L1).
Wolf-Goat-Cabbage Problem

Solution—first try

- predicate solve(position, moves), where moves is simply a list of passengers.

- in PROLOG:

\[
\begin{align*}
solve & (pos(right, empty, wgc), []). \\
solve(Position, [Move | SubSolution]) & :- \\
& \text{move}(Position, Move, Pos2), \\
& \text{solve}(Pos2, SubSolution).
\end{align*}
\]
Solution with history

- there are solutions without cycles
- predicate solve(position, cycle-free moves)
- predicate solve(position, forbidden positions, moves)
- in PROLOG:
  
  \[
  \text{solve}(\text{Start}, \text{Solution}) :
  \begin{align*}
  & \text{solve}(\text{Start}, [\text{Start}], \text{Solution}). \\
  \text{solve}(\text{pos(right, empty, wgc)}, \text{History}, []).
  \end{align*}
  \]

  \[
  \text{solve}(\text{Position}, \text{History}, [\text{Move} \mid \text{SubSolution}]) :
  \begin{align*}
  & \text{move}(\text{Position}, \text{Move}, \text{Pos2}), \\
  & \text{\+ member(}\text{Pos2, History}), \\
  & \text{solve}(\text{Pos2, [Pos2 \mid History]}, \text{SubSolution}). \\
  \end{align*}
  \]
References
