Advanced Programming 2017

Logic Programming, Continued

Andrzej Filinski
andrzej@di.ku.dk

Department of Computer Science
University of Copenhagen

September 28, 2017

Outline

- Two key ingredients of logic programming in Prolog:
  1. Unification-based constraint management.
  2. Backtracking tree search.
- Both are general structuring techniques, useful also when programming in other languages.
- Also, will briefly review key principles for writing correct logic programs.
  - “Algorithm = Logic + Control” [Kowalski]
  - Need to consider both.

Unification

- Formal notion of *two-sided pattern matching*.
- Absolutely central in definition of Prolog.
- Also used elsewhere in programming language design and implementation: notably type inference in ML, Haskell.
- More generally: instance of *constraint-based programming*.
  - Unification: constraints are about equality of subterms only.
  - Other constraint domains are also possible, e.g., linear inequalities.

Terms

- **Recall**: A Prolog *term* is either a *variable* or a *compound* term (*functor* and *arguments*):

  $$t ::= X \mid f(t_1, \ldots, t_n) \quad [n \geq 0]$$

  Note: for simplicity, treating atoms and numbers as nullary constructors, $a \sim a()$.

- Variables cannot be functors: $X(a, b)$ is ill-formed.
  - Possible in *higher-order unification*.

- Binary functors may be written infix: $X + Y$ instead of $+(X, Y)$, etc.
  - Purely a parsing issue.
  - Prolog has elaborate system for declaring prefix, infix, and postfix operators with arbitrary precedence and associativity.

- Additional syntactic sugar for lists built from "." (cons) and "[ ]".

Substitutions

- A *substitution* is a finite mapping of variables to terms:

  $$\sigma = \{X_1 \mapsto t_1, \ldots, X_n \mapsto t_n\}$$

  All the X_i must be different; order doesn't matter.

- *Domain* of a substitution: $\operatorname{dom}\sigma = \{X_1, ..., X_n\}$.

- *Applying* a substitution: $t[\sigma]$, recursively defined by:

$$\begin{array}{rcl} X[\sigma] & = & \begin{cases} t_i & \text{if } X = X_i \text{ for some } i \\ X & \text{if } X \notin \operatorname{dom}\sigma \end{cases} \\ f(t_1, \ldots, t_n)[\sigma] & = & f(t_1[\sigma], \ldots, t_n[\sigma]) \end{array}$$

Composing substitutions

- Assume given substitutions,

$$\begin{array}{rcl} \sigma & = & \{X_1 \mapsto t_1, \ldots, X_n \mapsto t_n\} \\ \sigma' & = & \{X'_1 \mapsto t'_1, \ldots, X'_m \mapsto t'_m\} \end{array}$$

  with $\mathrm{dom}\,\sigma \cap \mathrm{dom}\,\sigma' = \emptyset$, i.e., all variables distinct.

- Can define their *composition*:

$$\sigma \cdot \sigma' = \{X_1 \mapsto t_1[\sigma'], \ldots, X_n \mapsto t_n[\sigma'], X'_1 \mapsto t'_1, \ldots, X'_m \mapsto t'_m\}$$

- **Note:** $\sigma \cdot \sigma' \neq \sigma' \cdot \sigma$ (except in special cases).

- **Fact:** for any t, $(t[\sigma])[\sigma'] = t[\sigma \cdot \sigma']$

Unifiers

- A substitution σ is a *unifier* of t_1 and t_2 if $t_1[\sigma] = t_2[\sigma]$.
- A substitution σ is a *most general unifier (mgu)* of t_1 and t_2 if:
  1. $t_1[\sigma] = t_2[\sigma]$ (i.e., σ is a unifier of t_1 and t_2), and
  2. for any σ' s.t. $t_1[\sigma'] = t_2[\sigma']$, there exists a σ'' such that $\sigma' = \sigma \cdot \sigma''$.
     - σ'' may further instantiate terms in σ, and/or add more bindings.
- **Ex.** Let $t_1 = f(X, a, U)$, $t_2 = f(Z, X, V)$. Then
  - $\sigma_1 = \{X \mapsto a, Z \mapsto a, U \mapsto V\}$ is an mgu of t_1, t_2.

    $$f(X, a, U)[\sigma_1] = f(a, a, V) = f(Z, X, V)[\sigma_1].$$
  - Symmetrically, $\sigma_1' = \{X \mapsto a, Z \mapsto a, V \mapsto U\}$ is also an mgu.
    - Can take $\sigma_1'' = [V \mapsto U]$; then $\sigma_1' = \sigma_1 \cdot \sigma_1''$
  - $\sigma_2 = \{X \mapsto a, Z \mapsto a, U \mapsto b, V \mapsto b\}$ is a non-mgu unifier of t_1, t_2.
    - Can take $\sigma_2'' = [V \mapsto b]$; then $\sigma_2 = \sigma_1 \cdot \sigma_2''$.
  - $\sigma_3 = \{X \mapsto a, Z \mapsto X, U \mapsto V]$ is *not* a unifier of t_1, t_2:

    $$f(X, a, U)[\sigma_3] = f(a, a, V) \neq f(X, a, V) = f(Z, X, V)[\sigma_3].$$

    Similarly, $\{X \mapsto f(X)\}$ is *not* a unifier of X and $f(X)$.

Computing mgus: pseudocode

$mgu(t, t') =$ — returns a σ or **fail**

$\quad$ $\textbf{if}_{\exists X}\ t = X \wedge t' = X\ \textbf{then}\ \{\}$

$\quad$ $\textbf{else if}_{\exists X}\ t = X \wedge \neg(X\ \textbf{occurs in}\ t')\ \textbf{then}\ \{X \mapsto t'\}$

$\quad$ $\textbf{else if}_{\exists X'}\ t' = X' \wedge \neg(X'\ \textbf{occurs in}\ t)\ \textbf{then}\ \{X' \mapsto t\}$

$\quad$ $\textbf{else if}_{\exists f,n,\vec{t},\vec{t'}}\ t = f(t_1, ..., t_n) \wedge t' = f(t'_1, ..., t'_n)\ \textbf{then}$

$\quad\quad$ $mgu^*(\langle t_1, ..., t_n\rangle, \langle t'_1, ..., t'_n\rangle)$

$\quad$ **else fail**

$mgu^*(\langle t_1, ..., t_n\rangle, \langle t'_1, ..., t'_n\rangle) =$ — returns a σ or **fail**

$\quad$ $\textbf{if}\ n = 0\ \textbf{then}\ \{\}$

$\quad$ $\textbf{else if}_{\exists \sigma_1}\ mgu(t_1, t'_1) = \sigma_1\ \textbf{then}$

$\quad\quad$ $\textbf{if}_{\exists \sigma_\mathrm{r}}\ mgu^*(\langle t_2[\sigma_1], ..., t_n[\sigma_1]\rangle, \langle t'_2[\sigma_1], ..., t'_n[\sigma_1]\rangle) = \sigma_\mathrm{r}\ \textbf{then}$

$\quad\quad\quad$ $\sigma_1 \cdot \sigma_\mathrm{r}$

$\quad\quad$ **else fail**

$\quad$ **else fail**

Examples

Use the algorithm to compute:

- $mgu(f(X, a, U), f(Z, X, V))$
- $mgu(f(X, Y, Y), f(g(U), U, X))$

Unification: Example 1

$$
\begin{array}{l}
mgu(f(X, a, U)\rangle, f(Z, X, V)\rangle\rangle) = \\
mgu^*(\langle X, a, U\rangle, \langle Z, X, V\rangle) = \\
\quad [\sigma_1 = \{X \mapsto Z\}] \\
\quad [\sigma_r = mgu^*(\langle a, U\rangle, \langle Z, V\rangle) \\
\qquad [\sigma_1' = \{Z \mapsto a\}] \\
\qquad [\sigma_r' = mgu^*(\langle U\rangle, \langle V\rangle) \\
\qquad\quad [\sigma_1'' = \{U \mapsto V\}] \\
\qquad\quad [\sigma_r'' = mgu^*(\langle\rangle, \langle\rangle) = \{\}] \\
\qquad\; = \sigma_1'' \cdot \sigma_r'' = \{U \mapsto V\} \cdot \{\} = \{U \mapsto V\}] \\
\quad\; = \sigma_1' \cdot \sigma_r' = \{Z \mapsto a\} \cdot \{U \mapsto V\} = \{Z \mapsto a, U \mapsto V\}] \\
\; = \sigma_1 \cdot \sigma_r = \{X \mapsto Z\} \cdot \{Z \mapsto a, U \mapsto V\} = \{X \mapsto a, Z \mapsto a, U \mapsto V\}
\end{array}
$$

Unification: Example 2

$$
\begin{array}{l}
mgu(f(X,Y,Y), f(g(U),U,X)) = \\
mgu^*(\langle X,Y,Y\rangle, \langle g(U),U,X\rangle) = \\
\quad [\sigma_1 = mgu(X, g(U)) = \{X \mapsto g(U)\}] \\
\quad [\sigma_r = mgu^*(\langle Y,Y\rangle, \langle U, g(U)\rangle) \\
\qquad [\sigma_1' = mgu(Y,U) = \{Y \mapsto U\}] \\
\qquad [\sigma_r' = mgu^*(\langle U\rangle, \langle g(U)\rangle) \\
\qquad\quad [\sigma_1'' = mgu(U, g(U)) = \mathbf{fail}] \\
\qquad\ = \mathbf{fail}] \\
\quad\ = \mathbf{fail}] \\
\ = \mathbf{fail}
\end{array}
$$

Practical unification

- Works with *unification heap*: incrementally maintained global substitution state.
  - Each variable represented as mutable cell, initialized to “unbound”.
  - When creating a substitution for variable, update binding.
  - Whenever encountering a variable, see if it already has a binding
    - May need to *chase* through chain of bindings: $X \mapsto Y$, $Y \mapsto Z$, ...
    - Can do path compression like in union-find algorithm.

- Sometimes omits *occurs check*:
  - For efficiency: original algorithm is worst-case quadratic-time.
    - And, surprisingly, most of that time is spent on the check.
  - With heap, allows creation of “circular terms”, which may or may not be handled intelligently by rest of system.
    - Can cause later unification attempts to loop, if done naively.
  - Makes raw Prolog unification unsuited for some purposes (e.g., type inference).

Overview of search trees

- Aka *SLD resolution trees* (name comes from a particular variant of resolution-based theorem proving)
- Central in definition of Prolog semantics.
- More generally: instance of *search-based programming*.

Prolog clauses and goals

- A *goal* is of the form:

$$G \; ::= \; ?\!-\; t_1, \ldots, t_n. \quad [n \geq 0]$$

- Each *clause* is of the form:

$$C \; ::= \; t'_0 \; :\!-\; t'_1, \ldots, t'_m. \quad [m \geq 0]$$

  (Facts "$t'_0 \; :\!- \; .$" written as just "t'_0.")

- A *program* is a sequence of clauses:

$$P \; ::= \; C_1 \ldots C_k$$

- Fundamental computation step (resolution):

  Replace first goal term t_1 with body $t'_1, \ldots, t'_m$ of a program clause whose head t'_0 matches t_1 (accounting for variables bound by the match).

  (There may be more than one clause matching!)

Goal solving strategy

- Nondeterministic algorithm (pseudocode):

$$
\begin{array}{l}
\mathit{solve}(P, (?-\ t_1, \ldots, t_n)) = \\
\quad \textbf{if } n = 0 \textbf{ then success} \\
\quad \textbf{else} \\
\quad\quad \textbf{pick a } C \textbf{ from } P \\
\quad\quad \textbf{let } \sigma_0 = \text{(subst. replacing all vars in } C \text{ with fresh ones)} \\
\quad\quad \textbf{let } (t'_0 :-\ t'_1, \ldots, t'_m) = C[\sigma_0] \\
\quad\quad \textbf{if}_{\exists\sigma}\ \mathit{mgu}(t'_0, t_1) = \sigma \textbf{ then} \\
\quad\quad\quad \mathit{solve}(P, (?-\ t'_1[\sigma], \ldots, t'_m[\sigma], t_2[\sigma], \ldots, t_n[\sigma])) \\
\quad\quad \textbf{else fail}
\end{array}
$$

- On failure, undo a previous choice and pick a different clause.
  - typically, but not necessarily, undo only last-made choice first.
- How can we make this precise?

Prolog search trees

- Nodes: labeled with goals (term sequences)
  - Root node: initial query goal.

- Edges: labeled with substitutions (mgus)

- There is a σ-labelled edge from G to G' if:
  1. $G = (?-\ t_1, ..., t_n) \quad {\scriptstyle (n \geq 1)}$
  2. There is a renamed clause $(t'_0 :-\ t'_1, ..., t'_m)$ in program
  3. $\sigma = mgu(t'_0, t_1)$
  4. $G' = (?-\ t'_1[\sigma], ..., t'_m[\sigma], t_2[\sigma], ...t_n[\sigma])$

- A leaf node (i.e., with no outgoing edges) G is a
  - *success* if $n = 0$, i.e., nothing more to solve.
  - *failure* if $n > 0$, but t_1 doesn't match any program clauses.

- Note: tree is finitely branching, but may be infinitely deep. May contain finitely or infinitely many success nodes.

Reporting solutions

- Recall: success node is empty goal.
- To report variable bindings from initial goal, must recompute answer substitution from edge labels.
- Simpler: let last term in initial goal be $\mathit{report}(X_1, \ldots, X_n)$ where $X_1, \ldots, X_n$ are the variables whose values we want.
  - Or: $\mathit{report}('X_1' = X_1, \ldots, 'X_n' = X_n)$, to show original variable names as well
- Once all previous goals have been solved, *report*'s arguments will have been instantiated.
- A success node is then a goal with only a $\mathit{report}(\ldots)$ term.

Examples

- Let the program be given by the following clauses:

$$app([\,], L, L). \quad \%\ (1)$$
$$app([H|T], L, [H|R]) :- app(T, L, R). \quad \%\ (2)$$

- Construct the search trees for the goals:
  1. $?-\ app([a, b], [c, d], X), report(X).$
  2. $?-\ app(X, Y, [a, b]), report(X, Y).$

Search trees: Example 1

$$
\begin{array}{l}
?\!-\, app([a,b],[c,d],X), report(X). \\
\overset{(2_1)}{\to} \{H_1 \mapsto a, T_1 \mapsto [b], L_1 \mapsto [c,d], X \mapsto [a|R_1]\} \\
\quad ?\!-\, app([b],[c,d],R_1), report([a|R_1]). \\
\quad \overset{(2_2)}{\to} \{H_2 \mapsto b, T_2 \mapsto [], L_2 \mapsto [c,d], R_1 \mapsto [b|R_2]\} \\
\qquad ?\!-\, app([],[c,d],R_2), report([a|[b|R_2]]). \\
\qquad \overset{(1_3)}{\to} \{L_3 \mapsto [c,d], R_2 \mapsto [c,d]\} \\
\qquad\quad ?\!-\, report([a|[b|[c,d]]]). \quad \%\ success, X = [a,b,c,d]
\end{array}
$$

Notation: $\overset{(c_i)}{\to}$ refers to clause number c, with all its variables renamed by appending i.

Search trees: Example 2

$$?- app(X, Y, [a, b]), report(X, Y).$$

$$\stackrel{(1_1)}{\rightarrow} \{X \mapsto [], Y \mapsto [a, b], L_1 \mapsto [a, b]\}$$

$$?- report([], [a, b]). \quad \% \; success, X = [], Y = [a, b]$$

$$\stackrel{(2_2)}{\rightarrow} \{X \mapsto [a|T_2], H_2 \mapsto a, L_2 \mapsto Y, R_2 \mapsto [b]\}$$

$$?- app(T_2, Y, [b]), report([a|T_2], Y).$$

$$\stackrel{(1_3)}{\rightarrow} \{T_2 \mapsto [], Y \mapsto [b], L_3 \mapsto [b]\}$$

$$?- report([a|[]], [b]). \quad \% \; success, X = [a], Y = [b]$$

$$\stackrel{(2_4)}{\rightarrow} \{T_2 \mapsto [b|[]], H_4 \mapsto b, L_4 \mapsto Y, R_4 \mapsto []\}$$

$$?- app(T_4, Y, []), report([a|[b|[]]], Y).$$

$$\stackrel{(1_5)}{\rightarrow} \{T_4 \mapsto [], Y \mapsto [], L_5 \mapsto []\}$$

$$?- report([a|[b|[]]], []). \quad \% \; success, X = [a, b], Y = []$$

Traversing the search tree

- Standard Prolog strategy: always try clauses in order, undo last choice first.
- Corresponds to searching for solutions *depth-first*.
- *Incomplete*: may fail to find valid solution nodes.
  - Cf. deep-backtracking parser on a left-recursive grammar...
- **Ex.** May get stuck on useless infinite path:

$$p(X) :- p(X).$$
$$p(a).$$

$$?- p(R).$$

Query loops forever. (Could be “fixed” by different clause order.)

Traversing the search tree (cont'd)

- May fail to consider some combination of subgoal solutions:

  $nat(z).$
  $nat(s(N)) :- nat(N).$

  $?- nat(X).$
  % finds $X = z$; $X = s(z)$; $X = s(s(z)); \ldots$

  $?- nat(X), nat(Y), X = Y.$
  % finds just $X = z, Y = z$; then loops.

  $?- nat(X), X = Y, nat(Y).$
  % works again, as does having $X = Y$ as first subgoal.

- To obtain completeness, need to explore search tree *fairly* (no valid solution ignored forever).
  - Breadth-first traversal (work *queue* instead of *stack*).
  - Iterative deepening (depth-first, but with maximum depth). More space-efficient, but duplicates work.

Search trees in practice

- Committing to depth-first has practical advantages
  - More efficient implementation possible.
  - Works well with heap-based unification. (Bindings must be undone on backtracking; keep *trail* of modifications to undo.)
  - Can be compiled to efficient code (Warren Abstract Machine).
- Strict depth-first search also makes cuts ("!") meaningful: explicit *dynamic* pruning of parts of search tree.
  - May steer search away from infinite paths, "incorrect" solutions.
  - But often cut away too much, or too little.
- When breadth-first search needed, iterative deepening can often be expressed explicitly in Prolog program.
- Numerous extensions to control model exist.
  - Hereditary Harrop formulas: generalization of Prolog's Horn clauses, allow goals to temporarily extend program.
  - Various notions of constraint logic programming, concurrent execution, ...

Checklist for (pure) Prolog program correctness

For every relevant *mode* (division of arguments into inputs and outputs) of predicate:

1. **Soundness**: Are all the facts and rules individually correct?
   - Is every ground instance of every clause a true assertion about the problem domain?
   - **Note:** without cuts, other clauses don't matter here.
2. **Coverage**: Do facts and rules cover all situations in which predicate should hold?
   - Is there a matching clause for every relevant combination of input arguments?
   - Can every correct output be produced by at least one clause?
3. **Termination**: Is all recursion properly guarded?
   - Do all recursive calls have "smaller" main input arguments in callee than in caller?
   - "Smaller than" usually means "proper subterm of".
   - Needed because of Prolog's incomplete search strategy.

What next?

- If you haven't yet, do the recommended readings.
  - Even if they don't seem immediately relevant for the assignment.
- Work on Exercises and Assignment 3, starting at labs this afternoon.
  - May be a TA short, so do dynamic load balancing if necessary.
- Next week (and most of rest of course): Erlang, and concurrent/distributed programming.
  - Some Erlang syntax and concepts evidently inspired by Prolog.
  - In fact, first Erlang implementation was actually done in Prolog.