Programming language design and analysis

Constraint Logic Programming

Marius Minea

7 December 2015

Declarative programming

specify *what* the program should do, now *how*

in particular, *avoid state* (exposes internal implementation details)
or *side effects* (expose/observe computation flow)

Main exponents:

functional programming
still directly expresses formulas by which computations are done

logic programming
problem domain expressed as logic rules/implications

constraint programming
properties of solutions expressed as constraints over a given *theory*

Foundations of Prolog

developed ca. 1970 by Alain Colmerauer et al. in Marseille

A (pure) Prolog program is a list of *Horn clauses*.

- a *rule*: *Head* `:-` *Body* `.`
  - where *Body* is a conjunction *Predicate* , ... , *Predicate*
- a *fact*: *Predicate* `.`
  - equivalent to *Predicate* `:-` *true* `.`

`:-` means implication $\leftarrow$

- the *head* of a rule is the *conclusion*
- the predicates in the *body* are *hypotheses* (premises)

Executing a program means trying to satisfy a *query* (*goal*)

- i.e., determining if the goal follows as conclusion from the rules.

Prolog programs essentially encode *predicate logic*

Syntax of predicate logic: terms and formulas

Terms

variables v

$f(t_1, \cdots, t_n)$ where f is an n-ary function and $t_1, \cdots, t_n$ are *terms*.

constants can be viewed as 0-ary functions (no arguments)

Formulas (well-formed formulas)

$P(t_1, \cdots, t_n)$	with P an n-ary predicate, $t_1, \cdots, t_n$ terms
$\neg\alpha$	where α is a formula
$\alpha \to \beta$	where α, β are formulas
$\forall v\, \alpha$	with v variable, α formula: *universal quantification*

Other usual connectors:

$\alpha \wedge \beta \stackrel{def}{=} \neg(\alpha \to \neg\beta)$ (AND) $\alpha \vee \beta \stackrel{def}{=} \neg\alpha \to \beta$ (OR)

existential quantifier: $\exists x \varphi \stackrel{def}{=} \neg\forall x(\neg\varphi)$

Compared to propositional logic: instead of propositions, *predicates* over *terms*

Prolog examples and logic meaning

```
desc(X, Y) :- child(X, Y).
desc(X, Z) :- child(X, Y), desc(Y, Z).
child(anna, jon).
child(jon, peter).
child(eve, jon).
child(peter, mary).
```

Variables in clause head are *universally* quantified.

Rest of variables in clause body are *existentially* quantified.

$\forall X \forall Y \ child(X, Y)$

$\forall X \forall Z . \exists Y (child(X, Y) \land desc(Y, Z)) \rightarrow desc(X, Z)$

Resolution (in propositional logic)

Resolution is an *inference rule* that produces a new clause from two clauses with complementary literals (p and $\neg p$).

$$\frac{p \vee \alpha \qquad \neg p \vee \beta}{\alpha \vee \beta} \qquad \textit{resolution}$$

The new clause = *resolvent* of the two clauses w.r.t. p

Example: $rez_p(p \vee q \vee \neg r, \neg p \vee s) = q \vee \neg r \vee s$

Modus ponens may be seen as a *special case of resolution*:

$$\frac{p \vee \mathit{false} \qquad \neg p \vee q}{\mathit{false} \vee q}$$

Resolution is a *valid* inference rule:

$$\{p \vee \alpha, \neg p \vee \beta\} \models \alpha \vee \beta$$

(for any truth assignment where premises are true, conclusion is true)

Corollary: if $\alpha \vee \beta$ is a contradition, so is $(p \vee \alpha) \wedge (\neg p \vee \beta)$.

We use resolution to show that a formula is a *contradiction*.

resolution is a method for proof by *refutation*

Why substitution and term unification ?

We have two formuas where a predicate may appear positive and negated:

$\forall x.\forall y.P(x, g(y))$ and $\forall z.\neg P(z, a)$.

or

$\forall x.\forall y.P(x, g(y))$ and $\forall z.\neg P(a, z)$

Are these contradictory ?

We may *substitute* a universally quantified variable with *any* term

$\Rightarrow$ in the second case, we may substitute $x \mapsto a$, $z \mapsto g(y)$

$\Rightarrow$ we obtain $P(a, g(y))$ and $\neg P(a, g(y))$, *contradiction*

In the first case, we may not substitute y and obtain a from $g(y)$

interpretation: we may not assume that the arbitrary function g *must* also take the constant value a.

This is precisely defined by *substitution* and *unification*

Term substitutions

A *substitution* is a *function* that associates *terms* to *variables*:

$$\{x_1 \mapsto t_1,\ \ldots, x_n \mapsto t_n\}$$

For example, $f(x, g(y, z), a, t)\{x \mapsto g(y), y \mapsto f(b), t \mapsto u\}$
$= f(g(y), g(f(b), z), a, u)$

Obs: other encountered notations: x_i/t_i, or t_i/x_i

Usually postfix notation $T\sigma$ is used for substitutions σ applied to term T

The composition of two substitutions is a substitution

Term unification

Two terms t_1 and t_2 may be *unified* if there is a substitution σ that makes them equal: $t_1\sigma = t_2\sigma$.
Such a substitution is called *unifier*.

Example: $f(x, g(y))\{x \mapsto a\} = f(a, g(y)) = f(a, z)\{z \mapsto g(y)\}$
i.e., the substitution $\{x \mapsto a, z \mapsto g(y)\}$ is a *unifier* .

More generally: applied to a *set of* pairs of terms.

The *most general unifier* is that from which any other unifier may be obtained by using another substitution.

In *resolution*: having the clauses $P(l_1, l_2, \ldots l_n)$ and $\neg P(r_1, r_2, \ldots r_n)$ if we find a unifier for (l_1, r_1), ... we have a *contradiction*.

Unification rules

A variable x may be unified with any term t
if x *does not occur* in t not: x with $f(g(y), h(x, z))$
(substitution would lead to an infinite term)

Two functional terms may be unified only if they have identical functions, and the term arguments may be pairwise unified.
in particular: only identical constants may be unified

Prolog and resolution

Prolog execution can be seen in two ways:

Match goal with head of rule or fact, until no more subgoals.

Apply resolution with negation of goal, until empty clause.

Prolog and resolution

Consider as goal: `desc(X, peter).`

A *solution* = a value for X that makes the predicate true

A formula is *satisfiable* if its *negation* is a *contradiction*.

We derive a contradiction using *resolution*.

Prolog and resolution

Consider as goal: `desc(X, peter).`

A *solution* = a value for X that makes the predicate true

A formula is *satisfiable* if its *negation* is a *contradiction*.

We derive a contradiction using *resolution*.

Write the negated goal: $\neg$ `desc(X, peter).`

i.e., `desc(X, peter)` is *false* for any `X`.

Prolog and resolution

Consider as goal: `desc(X, peter).`
A *solution* = a value for X that makes the predicate true
A formula is *satisfiable* if its *negation* is a *contradiction*.
We derive a contradiction using *resolution*.

Write the negated goal: $\neg$ `desc(X, peter).`
i.e., `desc(X, peter)` is *false* for any `X`.

Choose the first rule for unification (use fresh variables):
`desc(X1, Y1)` $\vee$ $\neg$ `child(X1, Y1).`
We get as resolvent $\neg$ `child(X, peter).` `X1=X, Y1=peter`

Prolog and resolution

Consider as goal: `desc(X, peter).`
A *solution* = a value for X that makes the predicate true
A formula is *satisfiable* if its *negation* is a *contradiction*.
We derive a contradiction using *resolution*.

Write the negated goal: ¬ `desc(X, peter).`
i.e., `desc(X, peter)` is *false* for any `X`.

Choose the first rule for unification (use fresh variables):
`desc(X1, Y1)` ∨ ¬ `child(X1, Y1).`
We get as resolvent ¬ `child(X, peter).` `X1=X, Y1=peter`

Choose for unification the fact `child(jon, peter)` (nr. 3).
We get as resolvent the empty clause (contradiction) `X=jon`

Prolog and resolution

Consider as goal: `desc(X, peter).`

A *solution* = a value for X that makes the predicate true

A formula is *satisfiable* if its *negation* is a *contradiction*.

We derive a contradiction using *resolution*.

Write the negated goal: ¬ `desc(X, peter).`

i.e., `desc(X, peter)` is *false* for any `X`.

Choose the first rule for unification (use fresh variables):

`desc(X1, Y1)` ∨ ¬ `child(X1, Y1).`

We get as resolvent ¬ `child(X, peter).` `X1=X, Y1=peter`

Choose for unification the fact `child(jon, peter)` (nr. 3).

We get as resolvent the empty clause (contradiction) `X=jon`

Thus `desc(X, peter)` *is NOT false* for any `X`.

`desc(jon, peter)` is *true*. `X=jon` is a solution

Prolog and resolution

Consider as goal: `desc(X, peter).`
A *solution* = a value for X that makes the predicate true
A formula is *satisfiable* if its *negation* is a *contradiction*.
We derive a contradiction using *resolution*.

Write the negated goal: ¬ `desc(X, peter).`
i.e., `desc(X, peter)` is *false* for any `X`.

Choose the first rule for unification (use fresh variables):
`desc(X1, Y1)` ∨ ¬ `child(X1, Y1).`
We get as resolvent ¬ `child(X, peter).` `X1=X, Y1=peter`

Choose for unification the fact `child(jon, peter)` (nr. 3).
We get as resolvent the empty clause (contradiction) `X=jon`

Thus `desc(X, peter)` *is NOT false* for any `X`.
`desc(jon, peter)` is *true*. `X=jon` is a solution

Continue for other solutions....

Prolog example (cont.)

We restart with the negated goal: $\neg$`desc(X, peter)`.

Prolog example (cont.)

We restart with the negated goal: $\neg$`desc(X, peter)`.

We unify with rule 2 (renaming variables again):

`desc(X2, Z2)` $\vee \neg$ `child(X2, Y2)` $\vee \neg$ `desc(Y2, Z2)`

We get: $\neg$ `child(X, Y2)` $\vee\neg$ `desc(Y2, peter)`   `X2=X`, `Z2=peter`

Prolog example (cont.)

We restart with the negated goal: $\neg$`desc(X, peter)`.

We unify with rule 2 (renaming variables again):

`desc(X2, Z2)` $\vee \neg$ `child(X2, Y2)` $\vee \neg$ `desc(Y2, Z2)`

We get: $\neg$ `child(X, Y2)` $\vee\neg$ `desc(Y2, peter)` `X2=X`, `Z2=peter`

We unify with `child(anna, jon)` (nr. 3) `X=anna`, `Y2=jon`

We get as resolvent $\neg$ `desc(jon, peter)`.

Prolog example (cont.)

We restart with the negated goal: ¬`desc(X, peter)`.

We unify with rule 2 (renaming variables again):

`desc(X2, Z2)` ∨ ¬ `child(X2, Y2)` ∨ ¬ `desc(Y2, Z2)`

We get: ¬ `child(X, Y2)` ∨¬ `desc(Y2, peter)`   `X2=X`, `Z2=peter`

We unify with `child(anna, jon)` (nr. 3)   `X=anna`, `Y2=jon`

We get as resolvent ¬ `desc(jon, peter)`.

We've already seen `desc(petre, vasile)` ⇒ leads to empty clause.

⇒ `X=anna` is another solution for initial question

Prolog example (cont.)

We restart with the negated goal: ¬`desc(X, peter)`.

We unify with rule 2 (renaming variables again):
`desc(X2, Z2)` ∨ ¬ `child(X2, Y2)` ∨ ¬ `desc(Y2, Z2)`
We get: ¬ `child(X, Y2)` ∨¬ `desc(Y2, peter)`   `X2=X`, `Z2=peter`

We unify with `child(anna, jon)` (nr. 3)   `X=anna`, `Y2=jon`
We get as resolvent ¬ `desc(jon, peter)`.

We've already seen `desc(petre, vasile)` ⇒ leads to empty clause.
⇒ `X=anna` is another solution for initial question

If goal has variables, Prolog searches for all unifications/substitutions.

With no variables, determines if predicate is true.

Example with terms: list reversal

Use constant `nil` and binary function c (`cons`) to model lists.

Model n-ary *function* with $n+1$-ary *relation* (between args and result)

Model tail-recursive call using same variable in the result position.

```
rev3(nil, R, R).
rev3(c(H, T), Ac, R) :- rev3(T, c(H, Ac), R).
rev(L, R) :- rev3(L, nil, R)
```

With goal $rev(c(1, c(2, c(3, nil)))), X)$ we get $X = c(3, c(2, c(1, nil)))$.
Derivation: $rev(c(1, c(2, c(3, nil))), X)$ `L1=c(1,c(2,c(3,nil)))`, `R1=X`
$\leftarrow rev3(c(1, c(2, c(3, nil))), nil, X)$ `H1=1`, `T1=c(2,c(3,nil))`, `Ac1=nil`
$\leftarrow rev3(c(2, c(3, nil)), c(1, nil), X)$ `H2=2`, `T2=c(3,nil)`, `Ac2=c(1,nil)`
$\leftarrow rev3(c(3, nil), c(2, c(1, nil)), X)$ `H3=3`, `T3=nil`, `Ac3=c(2,c(1,nil))`
$\leftarrow rev3(nil, c(3, c(2, c(1, nil))), X)$ `X=c(3,c(2,c(1,nil)))`