Logic Programming

Programming Languages CS442

David Toman

School of Computer Science
University of Waterloo

From Type Annotations to Programs

Idea

Observation:

$$\textit{Computation}(\beta) \sim \textit{Cut-elimination}$$

$\Rightarrow$ *make the Cut-elimination the driving force for computation.*

- basis for the Logic Programming paradigm
  1. originated from *resolution-based* theorem provers for FOL
  2. restricted to a *manageable* subset of FOL
  3. fixed *search strategy* (unfortunately, an incomplete one)
- closely resembles cuts in type systems based on sequent calculus

First-order Logic

Idea

Use Proof Techniques for First-order Logic

- Syntax: $\varphi ::= P(t_1, \ldots, t_k) \mid \varphi \wedge \varphi \mid \varphi \vee \varphi \mid \neg\varphi \mid \exists x_i.\varphi \mid \forall x_i.\varphi$
  $\Rightarrow$ P predicate symbols (arity k)
  $\Rightarrow$ t_i terms (variables x_i, constants and function symbols)
- Interpretations and Models (with a domain (universe) U):
  $\Rightarrow$ an interpretation M: $c^M \in U$, $f^M : U^k \to U$, $P^M \subseteq U^k$
  $\Rightarrow$ a model is an interpretation that makes (a closed) φ true
- Logical Implication:

$$\mathcal{T} \models \varphi \quad \textit{iff} \quad \forall M.M \models \mathcal{T} \Rightarrow M \models \varphi$$

$\Rightarrow$ we want a mechanical procedure to do this.

Clauses and Resolution

Idea

How do we derive proofs for FOL implication questions?

$\Rightarrow$ *a (resolution-based) proof system*

- A clause is a formula of the form

$$\forall x_1, \ldots, x_k . L_1 \vee L_2 \vee \ldots \vee L_n$$

  $\Rightarrow$ L_i is a literal: $P(t_1, \ldots, t_l)$ or $\neg P(t_1, \ldots, t_l)$

  $\Rightarrow$ $x_1, \ldots, x_k$ variables free in all L_is.

- A resolution step is an application of a rule

$$\frac{L_1 \vee \ldots \vee L_n \vee A \qquad \neg A \vee L'_1 \vee \ldots \vee L'_m}{L_1 \vee \ldots \vee L_n \vee L'_1 \vee \ldots \vee L'_m}$$

Resolution Proofs

- Proof by contradiction:

$$\mathcal{T} \models \varphi \quad \textit{iff} \quad \mathcal{T} \cup \{\neg\varphi\} \textit{ is unsatisfiable}$$

- Resolution proof:
  1. express $\mathcal{T}$ and $\neg\varphi$ as *clauses*
  2. derive other clauses using resolution steps
  3. successful derivation of empty clause = contradiction

Theorem

Resolution is sound and complete proof system for FOL.

FOL and Skolemization

How do we convert arbitrary formulas to clauses?

- Convert closed formulas to prenex normal form

    $\Rightarrow$ move quantifiers to the top-level

    $\Rightarrow$ result: $Q_1 x_1. \cdots Q_k x_k.\varphi$, $Q_i \in \{\forall, \exists\}$, φ quantifier-free

- Skolemize all $\exists x$ quantifiers

    $\Rightarrow \forall x \exists y.\varphi(x, y) \to \forall x.\varphi(x, f(x))$ $\quad$ f is a Skolem function

- Convert to DNF and push $\forall$s to make clauses

$\Rightarrow$ result is a conjunction (set) of clauses

Note

The result is NOT EQUIVALENT, but it is EQUISATISFIABLE

Resolution and Unification

How do we deal with the function symbols, constants and variables?

Idea

We use the unification algorithm to match literals!

- *improved* resolution rule

$$\frac{L_1 \vee \ldots \vee L_n \vee A \qquad\qquad \neg B \vee L'_1 \vee \ldots \vee L'_m}{\sigma(L_1 \vee \ldots \vee L_n \vee L'_1 \vee \ldots \vee L'_m)} \qquad \sigma = \mathbf{mgu}(A, B)$$

- remember to *rename* variables every time a clause is used

The Horn Fragment

How do we use this to write programs?!?

Idea

Use clauses as procedure specifications and resolution as the computation step. But what is the result of a computation?

$\Rightarrow$ *an answer substitution generated by the resolution steps*

- Restrict to the Horn fragment

  $\Rightarrow$ at most one positive literal in all clauses

$$H \leftarrow G_1, \ldots, G_k$$

- Program = set of *program clauses* (exactly one positive literal)
  and a *goal* (clause with no positive literal)

Example

The program for appending lists:

- Program clauses (PROLOG syntax):

$$
\begin{aligned}
\mathbf{append}([\,], L, L) &\leftarrow \\
\mathbf{append}([X|A], B, [X|C]) &\leftarrow \mathbf{append}(A, B, C)
\end{aligned}
$$

- Goal $\leftarrow \mathbf{append}([a,b,c],[d,e],L)$

  ... answer substitution $L = [a,b,c,d,e]$

- Goal $\leftarrow \mathbf{append}(L1, L2, [a,b,c,d,e])$

  ... answer substitution $L1 = [\,], L2 = [a,b,c,d,e]$
  ... answer substitution $L1 = [a], L2 = [b,c,d,e]$, etc.

- Goal $\leftarrow \mathbf{append}([a],[],[b])$

  ... an answer substitution does not exist

SLD Resolution and DFS

Formal benefits of using Horn clauses:

- if a resolution proof exists, it is a linear input resolution proof
  - $\Rightarrow$ every resolution step: current goal and a program clause
  - $\Rightarrow$ proofs are linear sequences of such steps
- we can use a selection rule:
  - $\Rightarrow$ always resolve against the first literal in the current goal
- all possible linear resolutions can be arranged in a SLD tree
  - $\Rightarrow$ the tree can be searched for answers

Note

WARNING: SLD tree may be infinite!

PROLOG

A programming language based on Horn clauses and SLD resolution.

The good: declarative language

$\Rightarrow$ including industry-strength compilers

$\Rightarrow$ amenable to concurrent execution

The bad: the SLD tree is searched depth-first (backtracking)

$\Rightarrow$ programs may loop even if there is an answer

$\Rightarrow$ execution depends on ordering of *clauses*

. . . but the implementation is fast

The ugly: many things done through side-effects

$\Rightarrow$ cut ("!"): an explicit control of backtracking

$\Rightarrow$ self-modifying programs (database assertions)

Summary

- Resolution and Cut-elimination: two sides of the same coin

  $\Rightarrow$ despite of being developed completely independently

- Many variations and implementations of PROLOG

- Questions:
  1. what happens if we don't restrict ourselves to Horn clauses?
  2. can we have negation in the bodies of clauses?
  3. what if we disallow function symbols?