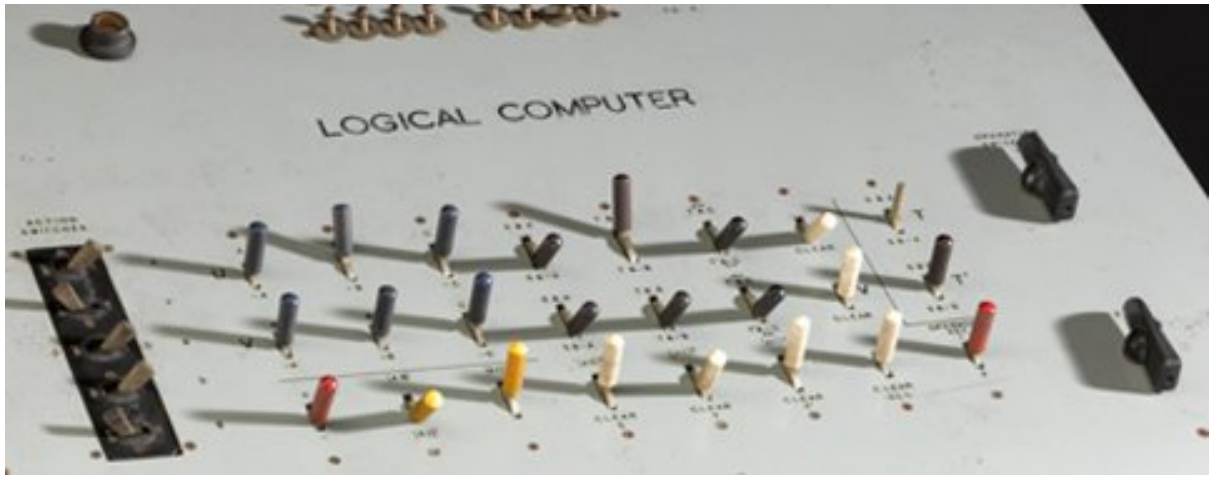
INF551

Computational Logic:

Artificial Intelligence in Mathematical Reasoning



Stéphane Graham-Lengrand

Stephane.Lengrand@Polytechnique.edu

Lecture V

Introduction to logic programming, Resolution

Where we stand

You now know how to perform proof-search in predicate logic

Today we

- see how to program using logic
- see alternative methodology for automated reasoning: resolution

Why treat them in the same lecture?

they involve a common concept, that of clausal forms

Contents

I. Clausal forms

II. Logic programming

III. Resolution

I. Clausal forms

Definitions: from propositional to predicate logic

You know literals and clauses from propositional logic (SAT-solving/DPLL)

We now lift those concepts to predicate logic

- *literal* (denoted $l, l', \ldots$) = atomic formula (positive literal)

  or negation of atomic formula (negative literal)

- *clause* (denoted $C, C', \ldots$) = disjunction of literals

  (implicitly or explicitly) universally quantified

  $\forall x_1 \ldots \forall x_k \; l_1 \vee \ldots \vee l_p$ with $\{x_1, \ldots x_k\} = \mathsf{FV}(l_1 \vee \ldots \vee l_p)$

- Let A be a (closed) formula of predicate logic.

  clausal form of A

  = finite set of clauses $C_1, \ldots, C_n$ such that $A \vdash \bot$ iff $C_1, \ldots, C_n \vdash \bot$

Many automated reasoning techniques are based on handling of clausal forms

Question: Does A always have a clausal form?

Producing clausal forms

Searching for formula B of the form

$$(\forall x_1^1 \ldots \forall x_{k_1}^1\ l_1^1 \vee \ldots \vee l_{p_1}^1) \wedge \ldots \wedge (\forall x_1^q \ldots \forall x_{k_q}^q\ l_1^q \vee \ldots \vee l_{p_q}^q)$$

with $\{x_1^i, \ldots, x_{k_i}^i\} = \mathsf{FV}(l_1^i \vee \ldots \vee l_{p_i}^i)$

and such that $A \vdash \bot$ iff $B \vdash \bot$

4 stages

Producing clausal forms

1. a *prenex form* of A:

a formula logically equivalent to A, of the form

$$Q_1 x_1 \ldots Q_n x_n \, C$$

with all quantifiers $Q_1 \ldots Q_n$ at the head of formula, and C quantifier-free

2. a *skolemised prenex form* of A:

a closed formula B of the form

$$\forall y_1 \ldots \forall y_m \, D$$

where D is quantifier-free, such that $A \vdash \bot$ iff $B \vdash \bot$.

(Every free variable or existentially quantified variable x has been substituted by new function symbols $x(y_1, \ldots, y_i)$)

Producing clausal forms

3. a *skolemised prenex conjunctive normal form* of A:

same thing with D being a conjunction of disjunctions,

i.e. we get a formula of the form

$$\forall y_1 \ldots \forall y_m \, (l_1^1 \vee \ldots \vee l_{p_1}^1) \wedge \ldots \wedge (l_1^q \vee \ldots \vee l_{p_q}^q)$$

4. a *clausal form* of A:

a conjunction of closed clauses, logically equivalent to a skolemised prenex conjunction normal form of A, i.e. we get something of the form:

$$(\forall x_1^1 \ldots \forall x_{k_1}^1 \; l_1^1 \vee \ldots \vee l_{p_1}^1) \wedge \ldots \wedge (\forall x_1^q \ldots \forall x_{k_q}^q \; l_1^q \vee \ldots \vee l_{p_q}^q)$$

with $\{x_1^i, \ldots, x_{k_i}^i\} = \mathsf{FV}(l_1^i \vee \ldots \vee l_{p_i}^i)$

Producing clausal forms

Example: $\neg(r(986) \Rightarrow \exists y, r(y))$ becomes $r(986) \wedge \forall y, \neg r(y)$,

the latter is unsatisfiable iff the former is

(i.e. iff $r(986) \Rightarrow \exists y, r(y)$ is valid)

To save space (but no logical information), we often retain from

$$(\forall x_1^1 \ldots \forall x_{k_1}^1 \; C^1) \wedge \ldots \wedge (\forall x_1^q \ldots \forall x_{k_q}^q \; C^q)$$

only the set of clauses $C_1, \ldots, C_n$ (variables are all implicitly universally quantified), where disjunction is associative & commutative, i.e. clauses = (multi-)sets of literals

II. Logic programming

Remember Church's theorem (1937)

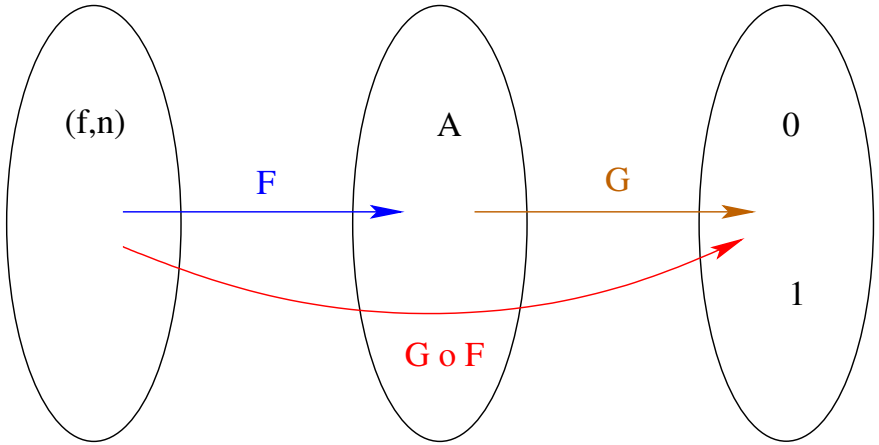
Undecidability: The set of all C such that $\mathcal{PA} \vdash C$ is undecidable

Remember the proof (from the halting problem)

We define a computable function F mapping each program f and each number n to a formula of arithmetic $A_{f,n}$ s.t.

$$\mathcal{PA} \vdash A_{f,n} \text{ iff } f \text{ terminates on input } n$$

By contradiction: If there were a program (representing G below) to decide provability of formulae in $\mathcal{PA}$, it would decide provability in $\mathcal{PA}$ of formulae of the form $A_{f,n}$.
By composing with F, we would have a program deciding whether "The program f terminate on input n" (the halting problem); contradiction.



Given a program f, construct formula B_f with free variables among $\{x, y\}$ s. t. the **Representation Theorem** holds: The following 3 assertions are equivalent

$$f(p) = q$$
$$\mathcal{PA} \vdash \{{}^{q}\!/_{y}\}\{{}^{p}\!/_{x}\}B_f$$
$$[\![\{{}^{q}\!/_{y}\}\{{}^{p}\!/_{x}\}B_f]\!]^{\mathbb{N}} = 1$$

where $\underline{p} := \overbrace{S(\ldots S}^{p \text{ times}}(0)\ldots)$

All the hard work is in the construction of B_f from f

(see INF412 or INF551 course notes)

What do we learn from this?

Given a computable function f, take the formula B_f

Given an integer input n in the domain of f, the only instance of Y such that

$$\mathcal{PA} \vdash \{Y/y\}\{p/x\}B_f$$

is $\underline{f(n)}$

How do you compute that instance? run proof-search as in Lecture 4!

If such an instance exists, proof-search will find it (otherwise it runs forever)

Proof-search can compute any computable thing

Proof-search is Turing-complete

Formulae of predicate logic form a programming language "LOGIC PROGRAMMING"

A practical programming language

Our “Logic Programming language” described above is currently

- given operational semantics by proof-search in $\mathcal{PA}$
- using arbitrary formulae of predicate logic to represent programs

  (or at least those formulae used in the representation theorem)

This is inherited from our original goal to prove Church’s theorem (and then Goëdel’s)

If only goal is to have a Turing-complete language that is

- given operational semantics by some proof-search process
- using logical formulae to represent programs

. . . then we can build a much simpler “Logic Programming language” that is

- given operational semantics by proof-search in the empty theory
- using a very restricted syntax form of logical formulae to represent programs:

  Horn clauses

This language is called **ProLog**

Other Logic Programming languages

Other choices are possible. Logic Programming languages form a family.

Their semantics is always tied

- to a precise methodology to perform proof-search
- to a class of formulae used for programs
- to a unification algorithm (for ProLog: Robinson's)

Variant: Higher-Order Logic Programming (λ-ProLog)

- proof-search in higher-order logic (i.e. with simply-typed λ-terms, etc)
- bigger class of formulae: Hereditary Harrop formulae
- *pattern unification*, an algorithm more powerful than Robinson's (but still decidable)

Variant: Constraint Logic Programming

uses different proof-search algorithm, with specific treatment of some sub-formulae (*constraints*) in a particular theory (e.g. reals)

All Turing-complete (not more: cf. Church-Turing thesis), very high-level languages

Today, we see basic ProLog

ProLog

Prover implementing goal-directed proof-search, in sequent calculus,

using meta-variables and Robinson's algorithm,

resticted to sequents of the form $C_1, \ldots, C_n \vdash l_1 \wedge \ldots \wedge l_p$ where

$C_1, \ldots, C_n$ are (universally quantified) Horn clauses with one positive literal

$l_1, \ldots, l_p$ are positive literals with meta-variables

Horn clause: clause that has at most one positive literal

Syntax:

`l:-l1,...,ln.` for $(l \vee (\neg l_1) \vee \ldots \vee (\neg l_n))$ with $n \geq 1$

or $(l_1 \wedge \ldots \wedge l_n) \Rightarrow l$

`l.` for $n = 0$

Conjunction of literals $l_1 \wedge \ldots \wedge l_n$ to prove, the *query*, is written

`?:-l1,...,ln.`

Property of this fragment of sequent calculus:

Left-hand side of sequent stays invariant all along proof-search

It's the *programme*, given in a `.pl` file

III. Resolution

A(nother) semi-decision procedure for validity (=provability)

Resolution method is a *refutational method*

(i.e. , tries to determine that negation of formula is unsat):

Let A be a formula of predicate logic.

We want a (not-too-stupid) semi-decision procedure to answer the question:

Is A provable/valid?

i.e.

Is $\neg A$ unsatisfiable?

i.e.

Is B unsatisfiable ? . . . where B is. . .

a clausal form of $\neg A$

Resolution method

. . . is not goal-directed like sequent calculus

. . . is based instead on a *saturation* mechanism:

from the clausal form of $\neg A$, produce new clauses such that

if the former had a model (were satisfiable), then it would be a model of the latter

We enrich our database of "known" clauses

we saturate it until we hit the empty clause $\bot$

If the clausal form of $\neg A$ were satisfiable, so would $\bot$.

Contradiction. $\neg A$ is unsat., so A is provable

Risk: saturation never terminates, the set of known clauses grows forever

Rules to produce new clauses

There are only 2 of them!

Simplification

$$\frac{C \vee l \vee l'}{\sigma(C \vee l)}\,\sigma = mgu(l = l')$$

Resolution

$$\frac{C \vee l \qquad C' \vee \neg l'}{\sigma(C \vee C')}\,\sigma = mgu(l = l')$$

The above are controlled combinations of the (more easily understandable) rules

$$\frac{C}{\sigma(C)} \qquad \frac{C \vee l \vee l}{C \vee l} \qquad \frac{C \vee l \qquad C' \vee \neg l}{C \vee C'}$$

Example

Want to prove $r(986) \Rightarrow \exists y, r(y)$

Take its clausal form: Clauses $r(986)$ and $\neg r(y)$

Take $\sigma = mgu(986 = y)$, i.e. the substitution that maps y to 986

applying resolution rule to $r(986)$ and $\neg r(y)$ gives the empty clause $\perp$

(resolution rule is to be understood bearing in mind that $C \vee \perp \equiv C$)

...so our set of clauses $r(986)$ and $\neg r(y)$ was unsatisfiable

...so the formula $\neg(r(986) \Rightarrow \exists y, r(y))$ was unsatisfiable

...so the formula $r(986) \Rightarrow \exists y, r(y)$ was valid

More generally...

Soundness and Completeness

A set of clauses $C_1, \ldots, C_n$ is unsatisfiable

if and only if

$\bot$ can be obtained by saturating the set with the simplification and resolution rules

Proof

Soundness easy (bottom implies top): for each of the two rules, just check that a model of the premisses is a model of the conclusion

Completeness: hard

Last example

The drinker's theorem

Formula $\exists x, (p(x) \Rightarrow \forall y, p(y))$, to be proved, becomes:

the set of clauses $p(x)$ and $\neg p(y(z))$, to show unsatisfiable

Apply resolution rule:

with $\sigma = mgu(x = y(z))$, i.e. the substitution mapping x to $y(z)$

obtain the empty clause

cqfd

Conclusion

Resolution used a lot in automated theorem provers for predicate logic

(Carine, Gandalf, Otter, Prover9, SNARK, SPASS, Vampire,. . .)

Currently dominating approach for pure predicate logic

Secret is in how to apply the rules (strategies)

We haven't seen them, but many techniques exist (selection function, orders,. . .)

In presence of theories, user interaction, or more expressive logics,

dominance of resolution not as clear

Goal-directed proof-search offers more natural human interaction (cf. Coq)

Very active research area:

mixing different approaches and try to get the best of all worlds

Questions?