School of informatics

Logic Programming: Theory

Alan Smaill

Nov 9, 2015

Today

School of informatics

- Completeness of the Backchain Inference System
- Therefore there is a complete Inference Strategy
- But there cannot be a decision procedure

Recall: Inference System for Definite Clause Logic

School of informatics

The given definite clauses are taken as axioms. We use a single inference rule, Backchain:

$$\frac{p_1\theta,\ \ p_2\theta,\ \ldots,\ \ p_n\theta,\ \ (p_1 \wedge p_2 \wedge \ldots \wedge p_n \Rightarrow q)}{q'\theta} \text{ where } \theta \text{ is mgu of } q', q$$

Given a query $\exists X\ q(X)$, see if $q(X)$ unifies with the "head" formula of a definite clause, with unifier θ. If so, top-down search will look for justifications of $p_1\theta,\ \ p_2\theta,\ \ldots,\ \ p_n\theta$.
More generally, given goal $\exists X\ q(X), r(X), s(X)$, look for unifier for $q(X)$, and now solve

$$p_1\theta,\ \ p_2\theta,\ \ldots,\ \ p_n\theta, r(X)\theta, s(X)\theta$$

Completeness of Backchain Inference

School of informatics

We write $T \models Q$ when Q is an atomic logical consequence of the definite clauses in T, using the standard semantics for predicate calculus, as described earlier.

We write $T \vdash Q$ when there is a derivation of Q from the statements in T using the Backchain inference system.

Completeness of Backchain Inference (2)

School of informatics

Claim: the backchain inference system is sound and complete for queries Q without variables:

$$T \models Q \text{ if and only if } T \vdash Q$$

The inference system is also sound and complete for queries that contain variables in the following sense:

> *if* $\exists X\ q(X)$ *is logical consequence of* T*, then there is a derivation of* $q(X)$ Sb *from* T*, where* Sb *is a substitution that is computed from the derivation, and gives us a value of* X *for which the query holds.*

Soundness

To check that the backchain inference system is sound, we want to check that any time the assumptions involved are true, then so is the conclusion. So, assume all the statements in T are true, and consider the possibilities:

- Axiom: this is immediate since we supposed that everything in T is true, and axioms are (unit) clauses in T.
- $$\frac{p_1\theta,\ \ p_2\theta,\ \ldots,\ p_n\theta,\ \ (p_1 \wedge p_2 \wedge \ldots \wedge p_n \Rightarrow q)}{q'\theta}$$ where θ is mgu of q', q

  – $p_1 \wedge p_2 \wedge \ldots \wedge p_n \Rightarrow q$ is true, because it's in T
  – So $p_1\theta \wedge p_2\theta \wedge \ldots \wedge p_n\theta \Rightarrow q\theta$ is true
  (substitution gives an instance of $\forall$ quantification)
  – now use propositional reasoning with $\wedge, \Rightarrow$.

Soundness ctd

If each rule application is sound, it follows that if we a complete (finite) derivation tree, if the axioms at the leaves are true, then so is the statement at the root of the tree.
This completes the argument. □

We expect soundness to be easy to show.
Completeness is harder; in general we need to show that whenever every choice of structure and interpretation $\mathcal{I}$ that makes T true also make Q true, then there is a derivation $T \vdash Q$.
Things are a bit easier here; we build a particular structure and interpretation $\mathcal{H}$ such that $T \vdash Q$ iff Q is true in $\mathcal{H}$ –
it's enough to look at a single structure.

Completeness

Completeness then follows: $\mathcal{H}$ is chosen to make everything in T true; if $T \models Q$, then Q is true in every model of T, so it's true in $\mathcal{H}$, and will be derivable if $\mathcal{H}$ has the property claimed above.

We will build a structure out of the syntax of the language at hand; let $\mathcal{L}$ be the choice of constants, function symbols and predicates under consideration. This must include the syntax of the definite clauses (the program) and of possible queries Q. We assume there is at least one constant – if not, add one in.

Completeness ctd

The domain for the interpretation is formed by taking the set $\mathcal{T}$ of all the terms that can be formed from the constants and function symbols (without variables):

$$
\begin{array}{rcl}
\textit{term} & ::= & \textit{constant} \\
 & & \mid \textit{fn\_symbol}\ \mathbf{(}\ \textit{term\_list}\ \mathbf{)} \\
\textit{term\_list} & ::= & \textit{term} \\
 & & \mid \textit{term}\ \mathbf{,}\ \textit{term\_list}
\end{array}
$$

This set is called the Herbrand Universe, which we use as the set underlying the model we describe.

Interpretations

Now we need to say how to interpret terms of the language.

- interpret constant $\mathtt{c}$ as $\mathtt{c}$ (itself!);
- interpret function symbol $\mathtt{f}$ as mapping arguments $\mathtt{t_1}, \ldots, \mathtt{t_n}$ to $\mathtt{f}(\mathtt{t_1}, \ldots, \mathtt{t_n})$.

The latter means that no evaluation is going on —
the interpretation of the function $+$ applied to constants $3, 4$ is $3 + 4$
(and not 7).

Herbrand models

Any model that uses the Herbrand universe as its domain of interpretation is called a Herbrand model. For example, interpret all predicates as being true everywhere; we saw earlier that this makes any set of definite clauses true, so this is a model alright.

For a given language $\mathcal{L}$, we define an ordering relation between models of T:

$M_1 \leq M_2$ is defined as:

for all predicates P and terms $t_1, \ldots, t_n$,

if $M_1 \models P(t_1, \ldots, t_n)$ **then** $M_2 \models P(t_1, \ldots, t_n)$

The model we are interested in is the least Herbrand model in this ordering. At this stage, it's not obvious that there is a least Herbrand model (or that if there is one, it is unique).

Interpretation ctd

School of informatics

For the desired model $\mathcal{H}$, we need to say how to interpret the predicates.
For equality, we just use the equality at the level of syntax.
(So $3 + 4 \neq 7$ according to this notion of equality.)

We use a fixed point construction to define when the predicates are true; for each predicate P with n arguments, we define a set T_P of tuples of terms $(\mathtt{t_1}, \ldots, \mathtt{t_n})$ with the intention that $\mathtt{P}(\mathtt{t_1}, \ldots, \mathtt{t_n})$ is true exactly when $(\mathtt{t_1}, \ldots, \mathtt{t_n}) \in T_P$.

Example

Let's take a simple case where the terms $\mathcal{T}$ are $\mathtt{z}, \mathtt{s}(\mathtt{z}), \mathtt{s}(\mathtt{s}(\mathtt{z})), \ldots$, and there is only one predicate `even`, given by:

```
even(z).
even(s(s(X))) :- even(X).
```

Now define $f : \mathcal{P}(\mathcal{T}) \rightarrow \mathcal{P}(\mathcal{T})$ to build up the set of terms for which `even` is true, if we use the clauses bottom-up (reasoning forward).

Example ctd

- Include base case(s):
  in this case: $\{\mathtt{z}\}$.
- successively add new terms that can be derived by one inference step from existing terms,

  This gives $f(X) = \{\ \mathtt{z}\ \} \cup \{\ s(s(t)) \mid t \in X\ \}$,

Successive applications of f give
$\{\ \mathtt{z}\ \}$,$\{\ \mathtt{z}, \mathtt{s}(\mathtt{s}(\mathtt{z}))\ \}$, $\{\ \mathtt{z}, \mathtt{s}(\mathtt{s}(\mathtt{z})), \mathtt{s}(\mathtt{s}(\mathtt{s}(\mathtt{s}(\mathtt{z}))))\ \}$, ...

The least fixed point then is an infinite set
(of terms where $\mathtt{s}$ is applied an even number of times).

Interpretation: general case

Here we will build up a set of atomic statements (like $\texttt{even(s(s(z)))}$, with the intention of characterising the set of atomic statements (without variables) that follow logically from the definite clauses T. The set of such ground atomic statements from language $\mathcal{L}$ is called $B_{\mathcal{L}}$, the Herbrand base for $\mathcal{L}$.

Interpretation: general case (2)

School of informatics

A *ground instance* of a clause `p(X) :- q(X,Y),r(Y).` is just the result of substituting bound terms for the variables in the clause, eg `p(s(a)) :- q(s(a),b), r(b).` Define $oneStep(P, X)$ for an atomic formula P and set of atomic formulas X to mean that there is a ground instance of a clause with head P, and such that every atomic formula in the body is in X.

For a given set of clauses T in language $\mathcal{L}$, define $f : \mathcal{P}(B_\mathcal{L}) \to \mathcal{P}(B_\mathcal{L})$ by

$$f(X) = \{\ P \in B_\mathcal{L} \mid oneStep(P, X)\ \}$$

Least Herbrand Model

We can check that f is monotone, and so has a least fixed point.
We can now define the least Herbrand model by saying that the true atomic statements are exactly those in the least fixed point of the function f.
It can be shown that the fixed point in this case is just the union of all the sets

$$f(\{\ \}), f(f(\{\ \})), f(f(f(\{\ \}))), \ldots$$

where f is applied finitely often.
This is not a computationally effective way to find out what is true in $\mathcal{H}$ however, since the definition of f uses (possibly infinitely many) instances of the program clauses.

Completeness ctd

Notice that if K is another Herbrand model for T, it must also correspond to a fixed point for f:
if ground instances of the body of a clause are true, and the clause is true, then the head instance must be true in K. So $H \leq K$.

Claim: For ground Q, $T \vdash Q$ if and only if $H \models Q$.

- left to right is easy, since the inference system is sound.
- For the harder direction, the full argument is only sketched here.
  Since Q appears in $\mathcal{H}$, it is obtained after a finite number n of applications of f. Show by induction on n that each formula that appears has a derivation using backchain. (This depends on properties of unification we have mentioned but not used so far.)

Complete inference procedure

School of informatics

We have already seen that Prolog's inference procedure is not complete – it may fail to find a solution, even when there is a solution.

However, the completeness result for the inference system using backchain gives us a complete algorithm for finding whether a query follows logically from a set of definite clauses.

Consider the search space generated by applying the inference rule backwards. The branching factor is finite (why?). If the query follows, then there will be a successful derivation at some depth.

So use a search strategy that is known to be complete in these circumstances (eg iterative deepening, breadth first).

No decision procedure

School of informatics

What we would like ideally would be a decision procedure that could tell us for a given query whether it follows logically from the given clauses. This is impossible.

We believe that no algorithm can be a decision procedure for this problem.

One argument for this uses the standard result from the theory of computability, which says that there is no algorithm that can determine whether a Turing machine computation on a given input terminates or not.

Halting problem

School of informatics

We can take the *Turing machine* as a general model of computation. We work with a finite set of symbols and states (with start and end state), an unbounded tape, and a head that can read, write symbols and move to right or left on tape.
A program is given by a finite set of instructions, of the form:

> *in state q looking at symbol S, write symbol S_2, move 1 step right, or left, or stay, and go into state q_2.*

It is known that there is no decision procedure for whether a given Turing machine will terminate. (see Hopcraft and Ullman, Introduction to Automata Theory . . .).

Turing machines in definite clause logic (Eriksson)

School of informatics

```
% tm( State, Left, Right, FinalLeft, FinalRight )
% args:  current state, rev tape to left of head,
%        (current symbol & tape to right), tape to left,
%          & to right at end state.

% for each program statement:
% if no movement:

tm(state, Left, [symbol | Right], FinalL, FinalR) :-
  tm(newstate, Left, [newsymbol | Right], FinalL, FinalR).

% If movement is r:
tm(state, Left, [symbol | Right], Final, Final) :-
  tm(newstate, [newsymbol | Left], Right, Final, FinalR).
```

TM in definite clause logic

School of informatics

```
% if movement is l, we define two clauses:

tm(state, [X | Left], [symbol | Right], FinalL, FinalR) :-
   tm(newstate, Left, [X, newsymbol | Right], FinalL, FinalR).
tm(state, [], [symbol | Right], FinalL, FinalR) :-
   tm(newstate, [], [empty, newsymbol | Right], FinalL, FinalR).

% and also
tm(State, Left, [], FinalL, FinalR) :-
  tm(State, Left, [empty], FinalL, FinalR).
tm(stop, Left, Right, Left, Right).

% does it terminate?
?-  tm(startstate, Left, Right, FinalL, FinalR).
```

And so . . .

If we use the standard Prolog interpreter (noting that there are no choice points, given a deterministic TM):

- If the Turing machine halts, the query succeeds (given enough time and memory).
- If the Turing machine halts in a non-accepting state, the query returns `no.`
- If the Turing machine does not terminate, then the Prolog query will not return an answer (given unlimited time and memory).

And so . . .

School of informatics

Is it possible that a different algorithm can be a decision procedure for queries to definite clause programs?

If we had a decision procedure for definite clause logic, we could *decide* whether the halting statement is true, for an arbitrary Turing machine program.

But this is impossible: we cannot decide the halting problem. Therefore there is no algorithm that will decide queries in definite clause logic.

Summary

School of informatics

- Completeness of the Backchain Inference System
- Therefore there is a complete Inference Strategy
- But there cannot be a decision procedure