Lectures 16 and 17 (Week 10)
Reasoning beyond FOL

COMP24412: Symbolic AI

Giles Reger

April 2020

This Week

Answering Questions with Resolution+Saturation

Reasoning with Theories in General

Reasoning with the Theory of Equality

Reasoning with the Theory of Arithmetic

Logics 'above' and 'below' FOL in expressiveness

Aim and Learning Outcomes

The aim of these two lectures is to:

Explore how we can reason pragmatically with more than FOL and efficiently with less.

Learning Outcomes

By the end you will be able to:

1. Demonstrate how resolution can be used to answer questions
2. Explain the notion of reasoning with respect to a theory
3. Describe the different theories of ‘Equality’ and ‘Arithmetic
4. Apply equality reasoning rules to solve simple equality problems
5. Recall that adding arithmetic to FOL makes it fully undecidable
6. Describe the high-level ideas used for arithmetic reasoning
7. Recall a number of different formalisms and describe the general way in which they are related to FOL (e.g. more or less expressive)

Quick Recap

A literal is an an atom or its negation. A clause is a disjunction of literals. Clauses are implicitly universally quantified.

Resolution works on clauses

$$\frac{\neg l_1 \vee C \qquad l_2 \vee D}{(C \vee D)\theta} \quad \theta = \mathrm{mgu}(l_1, l_2)$$

Is sound as the conclusion is true in any model of the premises.

We will use resolution for refutational based reasoning e.g. to check entailment $\Gamma \models \phi$ we check consistency of $\Gamma \cup \{\neg\phi\}$.

This works by saturation e.g. we extend $\Gamma \cup \{\neg\phi\}$ by logical consequences (given by resolution) until there is nothing left to add or we have derived *false*. This doesn't necessarily terminate as FOL is semi-decidable.

Questions

Consider the knowledge base

$$
\begin{array}{l}
\forall x.((\text{hasWheels}(x) \land \text{hasEngine}(x)) \rightarrow \text{car}(x) \\
\forall x.(\text{car}(x) \land \text{reallyFast}(x)) \rightarrow \text{supercar}(x) \\
\text{hasWheels}(\text{lada}) \\
\text{hasWheels}(\text{mclarenf1}) \\
\text{hasEngine}(\text{mclarenf1}) \\
\text{reallyFast}(\text{mclarenf1}) \\
\text{reallyFast}(\text{sonic})
\end{array}
$$

I can check whether this knowledge base entails some statements

$$\Gamma \overset{?}{\models} \forall x.(\mathit{wheels}(x) \rightarrow \mathit{supercar}(x)) \qquad \Gamma \overset{?}{\models} \forall x.(\mathit{supercar}(x) \rightarrow \mathit{wheels}(x))$$

$$\Gamma \overset{?}{\models} \text{supercar}(\text{lada}) \qquad \Gamma \overset{?}{\models} \exists x.\text{supercar}(x)$$

We'll get true/false answers but can we get more?

Question Answering (From Week 7)

A question or query can either be viewed as an existential conjecture.

Given knowledge base Γ and a query $\exists X.\phi[X]$ we search for a substitution σ such that

$$\Gamma \models \phi[X]\sigma$$

we do this be transforming the problem into

$$\Gamma \models \exists X.(\mathsf{ans}(X) \wedge \phi[X]) \wedge \exists X.\neg\mathsf{ans}(X)$$

for fresh predicate ans.

We have a single predicate capturing the result of the question.

If we drop the highlighted part then this predicate only appears negatively so cannot be resolved away. Therefore, all instances of the predicate in the saturated set correspond to an answer to the question.

Question Answering: Example

$\forall x.((\text{hasWheels}(x) \wedge \text{hasEngine}(x)) \rightarrow \text{car}(x)$
$\forall x.(\text{car}(x) \wedge \text{reallyFast}(x)) \rightarrow \text{supercar}(x)$
hasWheels(lada)
hasWheels(mclarenf1)
hasEngine(mclarenf1)
reallyFast(mclarenf1)
reallyFast(sonic)

Question Answering: Example

$\neg\text{hasWheels}(x) \vee \neg\text{hasEngine}(x) \vee \text{car}(x)$
$\neg\text{car}(x) \vee \neg\text{reallyFast}(x) \vee \text{supercar}(x)$
$\text{hasWheels}(\text{lada})$
$\text{hasWheels}(\text{mclarenf1})$
$\text{hasEngine}(\text{mclarenf1})$
$\text{reallyFast}(\text{mclarenf1})$
$\text{reallyFast}(\text{sonic})$

Question Answering: Example

$$\left\{\begin{array}{l}\neg\text{hasWheels}(x) \vee \neg\text{hasEngine}(x) \vee \text{car}(x)\\ \neg\text{car}(x) \vee \neg\text{reallyFast}(x) \vee \text{supercar}(x)\\ \text{hasWheels}(\text{lada})\\ \text{hasWheels}(\text{mclarenf1})\\ \text{hasEngine}(\text{mclarenf1})\\ \text{reallyFast}(\text{mclarenf1})\\ \text{reallyFast}(\text{sonic})\end{array}\right\} \overset{?}{\models} \exists x.\text{supercar}(x)$$

Question Answering: Example

$$\left\{\begin{array}{l} \neg\mathsf{hasWheels}(x) \vee \neg\mathsf{hasEngine}(x) \vee \mathsf{car}(x) \\ \neg\mathsf{car}(x) \vee \neg\mathsf{reallyFast}(x) \vee \mathsf{supercar}(x) \\ \mathsf{hasWheels}(\mathsf{lada}) \\ \mathsf{hasWheels}(\mathsf{mclarenf1}) \\ \mathsf{hasEngine}(\mathsf{mclarenf1}) \\ \mathsf{reallyFast}(\mathsf{mclarenf1}) \\ \mathsf{reallyFast}(\mathsf{sonic}) \end{array}\right\} \overset{?}{\models} \exists x.(\mathsf{ans}(x) \wedge \mathsf{supercar}(x))$$

Question Answering: Example

$\neg \mathrm{hasWheels}(x) \vee \neg \mathrm{hasEngine}(x) \vee \mathrm{car}(x)$
$\neg \mathrm{car}(x) \vee \neg \mathrm{reallyFast}(x) \vee \mathrm{supercar}(x)$
$\mathrm{hasWheels}(\mathrm{lada})$
$\mathrm{hasWheels}(\mathrm{mclarenf1})$
$\mathrm{hasEngine}(\mathrm{mclarenf1})$
$\mathrm{reallyFast}(\mathrm{mclarenf1})$
$\mathrm{reallyFast}(\mathrm{sonic})$
$\neg \mathrm{ans}(x) \vee \neg \mathrm{supercar}(x)$

Question Answering: Example

$\neg \mathsf{hasWheels}(x) \vee \neg \mathsf{hasEngine}(x) \vee \mathsf{car}(x)$
$\neg \mathsf{car}(x) \vee \neg \mathsf{reallyFast}(x) \vee \mathsf{supercar}(x)$
$\mathsf{hasWheels}(\mathsf{lada})$
$\mathsf{hasWheels}(\mathsf{mclarenf1})$
$\mathsf{hasEngine}(\mathsf{mclarenf1})$
$\mathsf{reallyFast}(\mathsf{mclarenf1})$
$\mathsf{reallyFast}(\mathsf{sonic})$
$\neg \mathsf{ans}(x) \vee \neg \mathsf{supercar}(x)$
$\neg \mathsf{ans}(x) \vee \neg \mathsf{car}(x) \vee \neg \mathsf{reallyFast}(x)$
$\neg \mathsf{ans}(\mathsf{mclarenf1}) \vee \neg \mathsf{car}(\mathsf{mclarenf1})$
$\neg \mathsf{ans}(\mathsf{mclarenf1}) \vee \neg \mathsf{hasWheels}(\mathsf{mclarenf1}) \vee \neg \mathsf{hasEngine}(\mathsf{mclarenf1})$
$\neg \mathsf{ans}(\mathsf{mclarenf1}) \vee \neg \mathsf{hasWheels}(\mathsf{mclarenf1})$
$\neg \mathsf{ans}(\mathsf{mclarenf1})$

Question Answering in Vampire

Just add the option –question_answering answer_literal

However, what if there are infinite answers or even just one answer but we still don't saturate (quickly)?

Currently, Vampire will just detect the first answer and return it.

There is an experimental branch qa that you can checkout and build that returns multiple answers bounded using the option question_count

Another Example

```
fof(l1,axiom, link(a,b)).
fof(l2,axiom, link(b,c)).
fof(l3,axiom, link(b,d)).
fof(l4,axiom, link(d,b)).
fof(l5,axiom, link(d,e)).

fof(r1,axiom, ![X,Y] :
   (link(X,Y) => ispath(X,Y,path(X,Y)))).
fof(r2,axiom, ![X,Y,Z,W] :
   ((link(X,Y) & ispath(Y,W,path(Y,Z)))
      => ispath(X,W,path(X,path(Y,Z))))).

fof(c,conjecture, ?[P] : ispath(a,e,P)).
```

Resolution and Equality

$$\begin{array}{ll} \neg rich(father(x)) \lor happy(x) & \\ rich(david) & \models happy(giles) \\ father(giles) = david & \end{array}$$

Resolution and Equality

$$\neg rich(father(x)) \vee happy(x)$$
$$rich(david)$$
$$father(giles) = david$$
$$\neg happy(giles)$$

Resolution and Equality

$$\neg rich(father(x)) \lor happy(x)$$
$$rich(david)$$
$$father(giles) = david$$
$$\neg happy(giles)$$

Resolution and Equality

$$\neg rich(father(x)) \lor happy(x)$$
$$rich(david)$$
$$father(giles) = david$$
$$\neg happy(giles)$$
$$\neg rich(father(giles))$$

Resolution and Equality

$$\neg rich(father(x)) \vee happy(x)$$
$$rich(david)$$
$$father(giles) = david$$
$$\neg happy(giles)$$
$$\neg rich(father(giles))$$

$father(x)$ and $david$ do not unify.

Equality

Recall that we are working with FOL with equality but cannot reason directly with equality using resolution. The fix is to axiomitise equality but this has a major disadvantage - search space explosion.

Consider

```
fof(a,axiom, f(X) = X).
fof(a,axiom, f(X) = g(X,X)).
fof(a,axiom, p(f(f(f(f(f(a))))))).
fof(a,axiom, b = g(f(f(a)),f(f(a)))).
fof(a,axiom, ~p(g(b,b))).
```

We have to generate 434 clauses to prove this inconsistent whilst it should only take a few rewriting steps.

What's the issue? We want to restrict the calculus to only consider certain models e.g. we would like to reason modulo the theory not with it.

Reasoning Modulo a Theory

Let $\Sigma_{\mathcal{T}}$ be an interpreted signature e.g. the symbols in the theory. For equality this just contains the predicate $=$ but later for arithmetic it will include $+, -, *, \leq$ and an interpreted $=$.

Let a theory $\mathcal{T}$ over $\Sigma_{\mathcal{T}}$ be a class of interpretations that fix some interpretation for $\Sigma_{\mathcal{T}}$. Often this class is singular.

An interpretation is consistent with $\mathcal{T}$ if it is consistent on $\Sigma_{\mathcal{T}}$. A formula is consistent modulo $\mathcal{T}$ if it has a model consistent with $\mathcal{T}$. It is valid if it is true in all interpretations consistent with $\mathcal{T}$.

A finite set of axioms $\mathcal{A}$ defines a theory $\mathcal{T}$ if the set of interpretations in which $\mathcal{A}$ are true coincide with $\mathcal{T}$.

If we saturate then we can build a model, but it is not guaranteed that this model is consistent with $\mathcal{T}$ unless the input contains $\mathcal{A}$ defining $\mathcal{T}$.

Equality as a Theory

It is clear that we can define $\mathcal{T}_=$ over $\Sigma_=$ as the theory that interprets $=$ as equality.

However, the equality axioms we add $\mathcal{A}_=$ don't quite define the exact set of interpretations defined by $\mathcal{T}_=$ as they allow interpretations not allowed in $\mathcal{T}_=$ e.g. by equating domain elements not constrained to be different.

However, it is possible to show that both theories have the same set of provable statements.

We can also introduce some inference rules that allow us to reason directly in $\mathcal{T}_=$ without having to add $\mathcal{A}_=$.

Paramodulation

The paramodulation rule lifts the idea behind resolution to equality

$$\frac{C \vee s = t \qquad\qquad l[u] \vee D}{(l[t] \vee C \vee D)\theta} \quad \theta = \text{mgu}(s, u)$$

where u is not a variable.

Paramodulation

The paramodulation rule lifts the idea behind resolution to equality

$$\frac{C \vee s = t \qquad l[u] \vee D}{(l[t] \vee C \vee D)\theta} \quad \theta = \mathrm{mgu}(s, u)$$

where u is not a variable.

For example

$$\frac{father(giles) = david \qquad \neg rich(father(x)) \vee happy(x)}{\neg rich(david) \vee happy(giles)}$$

where $u = father(x)$ and therefore $\theta = \{x \mapsto giles\}$.

Special case: unit equalities

The demodulation rule works with unit equalities

$$\frac{s = t \qquad l[u] \vee D}{(l[t] \vee D)\theta} \quad \theta = \text{matches}(s, u)$$

Note that u must be an instance of s.

Why is this special? The premise $l[u] \vee D$ becomes redundant and we can remove it.

Still Explosive

Notice that we could apply these two rules in either direction.

For example we can do this

$$\frac{father(giles) = {\color{red}david} \qquad rich({\color{red}david})}{rich(father(giles))}$$

However, $rich(father(giles))$ seems to be going in the wrong direction as it is more complicated than $rich(david)$.

Still Explosive

Notice that we could apply these two rules in either direction.

For example we can do this

$$\frac{\mathit{father}(\mathit{giles}) = \mathit{david} \qquad \mathit{rich}(\mathit{david})}{\mathit{rich}(\mathit{father}(\mathit{giles}))}$$

However, $\mathit{rich}(\mathit{father}(\mathit{giles}))$ seems to be going in the wrong direction as it is more complicated than $\mathit{rich}(\mathit{david})$.

This is similar to how in resolution we could derive the same thing in many redundant ways. In fact, what we can do is order our equalities using the simplification ordering we saw before and have a notion of ordered paramodulation and demodulation.

Ordered paramodulation is usually replaced the nicer superposition rule.

Equality Resolution

We have another special case, is the following ever true?

$$\forall x. f(x) \neq f(a)$$

e.g. for every input to f the result is not equal to applying f to a.

Equality Resolution

We have another special case, is the following ever true?

$$\forall x. f(x) \neq f(a)$$

e.g. for every input to f the result is not equal to applying f to a.

Functions in first-order logic are always total

Equality Resolution

We have another special case, is the following ever true?

$$\forall x. f(x) \neq f(a)$$

e.g. for every input to f the result is not equal to applying f to a.

Functions in first-order logic are always total

So, no. We have a rule called equality resolution

$$\frac{s \neq t \vee C}{C\theta} \quad \theta = \mathrm{mgu}(s, t)$$

e.g. if we can make two terms equal then any disequality is false.

Small Example

$$\begin{array}{ll} \neg rich(father(x)) \vee happy(x) & \\ rich(david) & \models happy(giles) \\ father(giles) = david & \end{array}$$

Small Example

$$\neg rich(father(x)) \vee happy(x)$$
$$rich(david)$$
$$father(giles) = david$$
$$\neg happy(giles)$$

Small Example

$$\neg rich(father(x)) \vee happy(x)$$
$$rich(david)$$
$$father(giles) = david$$
$$\neg happy(giles)$$

Small Example

$$\neg rich(father(x)) \vee happy(x)$$
$$rich(david)$$
$$father(giles) = david$$
$$\neg happy(giles)$$
$$\neg rich(father(giles))$$

Small Example

$$\neg rich(father(x)) \vee happy(x)$$
$$rich(david)$$
$$father(giles) = david$$
$$\neg happy(giles)$$
$$\neg rich(father(giles))$$

Small Example

$$\neg rich(father(x)) \vee happy(x)$$
$$rich(david)$$
$$father(giles) = david$$
$$\neg happy(giles)$$
$$\neg rich(father(giles))$$
$$\neg rich(david)$$

Small Example

$$\neg rich(father(x)) \vee happy(x)$$
$$rich(david)$$
$$father(giles) = david$$
$$\neg happy(giles)$$
$$\neg rich(father(giles))$$
$$\neg rich(david)$$

Small Example

$$
\begin{array}{l}
\neg rich(father(x)) \vee happy(x) \\
rich(david) \\
father(giles) = david \\
\neg happy(giles) \\
\neg rich(father(giles)) \\
\neg rich(david) \\
\color{red}{false}
\end{array}
$$

Equality Reasoning in Vampire

Equality reasoning is on by default in Vampire so we just need to turn off the `equality_proxy` option, either by removing it from `run_vampire` or by adding `-ep off`.

Now with the previous example that took 434 clauses to prove we only need 24 steps.

Arithmetic is Useful

People modelling in first-order logic usually assume that they have access to arithmetic. However, it does not come for free.

There are different ways of encoding arithmetic in first-order logic but the most general are undecidable.

Often full arithmetic is not required and it is better to encode orderings or counting in some other way.

Different Kinds of Arithmetic

Presburger Arithmetic has symbols $0, succ, +, =$ defined by some axioms

$$\begin{array}{l} 0 \neq x + 1 \\ (x + 1 = y + 1) \rightarrow x = y \\ (x + 0) = x \\ x + (y + 1) = (x + y) + 1 \end{array}$$

and *induction* e.g. for every formula $\phi[n]$

$$(\phi[0] \wedge \forall x.(\phi[x] \rightarrow \phi[x+1])) \rightarrow (\forall y.\phi[y])$$

but induction is not finitely axiomitisable - we cannot represent arithmetic in first order logic.

However, by itself Presburg arithmetic is decidable.

Different Kinds of Arithmetic

Peano Arithmetic has symbols $0, succ, +, \times, =, \leq$ defined by some axioms

$$\begin{array}{l}
\forall x.(x \neq succ(x)) \\
\forall x, y.(succ(x) = succ(y) \rightarrow x = y) \\
\forall x.(x + 0 = x) \\
\forall x, y.(x + succ(y) = succ(x + y)) \\
\forall x.(x \times 0 = 0) \\
\forall x, y.(x \times succ(y) = x + (x \times y)) \\
\forall x, y.(x \leq y \leftrightarrow \exists z.(x + z = y))
\end{array}$$

and *induction* e.g. for every formula $\phi[n]$

$$(\phi[0] \wedge \forall x.(\phi[x] \rightarrow \phi[succ(x)])) \rightarrow (\forall y.\phi[y])$$

which, again is not finitely axiomitisable.

Peano arithmetic is incomplete and undecidable.

Reasoning with Arithmetic

Often we just use Integer Arithmetic with constants $0, 1, 2, 3, \ldots$ that interprets $+, -, \times, = \leq$ etc directly on these. When we add division we get an infinite set of models where division by 0 can be interepreted arbitrarily. Clearly, we can do as much as in Peano arithmetic.

Even if we don't have a complete set of axioms we can add some sensible ones e.g.

$$
\begin{aligned}
& x + 0 = x \\
& x + y = y + x \\
& (x + y) + z = x + (y + z) \\
& x + 1 > x
\end{aligned}
$$

and attempt to prove things.

However, if we saturate we don't know whether this corresponds to a model or not. We also know, due to the undecidablity of arithmetic, that there are some provable things we cannot prove.

Basic Support for Arithmetic in Vampire

- Normalization of interpreted operations, e.g.

$$t_1 \geq t_2 \rightsquigarrow \neg(t_1 < t_2) \qquad a - b \rightsquigarrow a + (-b)$$

- Evaluation of ground interpreted terms, e.g.

$$f(1+2) \rightsquigarrow f(3) \qquad f(x+0) \rightsquigarrow f(x) \qquad 1+2 < 4 \rightsquigarrow \mathit{true}$$

- Balancing interpreted literals, e.g.

$$4 = 2 \times (x+1) \rightsquigarrow (4 \text{ div } 2) - 1 = x \rightsquigarrow x = 1$$

- Interpreted operations treated specially by ordering
  (make interpreted things small, do uninterpreted things first)

Satisfiability Modulo Theories

This is a powerful technique for reasoning with arithmetic without quantifiers.

The idea:

1. Use a SAT solver to explore the boolean structure of the problem
2. Whenever it produces a model, check whether it is theory-consistent
3. If it is stop with model, otherwise block and carry on

Little example

$$a > 0 \vee b > 0 \qquad a * 5 = 0 \qquad -b > 1$$

In reality, it is much more clever than that. Modern SMT solvers:

1. Learn from inconsistencies to prune the search space
2. Combine arbitrary theory solvers

They also include quantifier instantiation heuristics to for quantifiers.

Satisfiability Modulo Theories

This is a powerful technique for reasoning with arithmetic without quantifiers.

The idea:

1. Use a SAT solver to explore the boolean structure of the problem
2. Whenever it produces a model, check whether it is theory-consistent
3. If it is stop with model, otherwise block and carry on

Little example

$$a > 0 \lor b > 0 \qquad a * 5 = 0 \qquad -b > 1$$

In reality, it is much more clever than that. Modern SMT solvers:

1. Learn from inconsistencies to prune the search space
2. Combine arbitrary theory solvers

They also include quantifier instantiation heuristics to for quantifiers.

Satisfiability Modulo Theories

This is a powerful technique for reasoning with arithmetic without quantifiers.

The idea:

1. Use a SAT solver to explore the boolean structure of the problem
2. Whenever it produces a model, check whether it is theory-consistent
3. If it is stop with model, otherwise block and carry on

Little example

$$a > 0 \vee b > 0 \qquad a * 5 = 0 \qquad -b > 1$$

In reality, it is much more clever than that. Modern SMT solvers:

1. Learn from inconsistencies to prune the search space
2. Combine arbitrary theory solvers

They also include quantifier instantiation heuristics to for quantifiers.

Theory Reasoning is a Research Topic

In the following I describe a few research-level ideas we have implemented in Vampire for theory reasoning.

This are for your interest only and for you to use in the Coursework if you decide to use theories.

They are non-examinable and I don't describe them in enough detail to understand them fully. But I am happy to discuss them further with any of you.

AVATAR modulo Theories (since 2015)

The AVATAR architecture [Voronkov 2014]

- modern architecture of first-order theorem provers
- combines saturation with SAT-solving
- efficient realization of the *clause splitting rule*

$$\forall x, z, w.\ \underbrace{s(x) \vee \neg r(x, z)}_{\textit{share } x \textit{ and } z} \vee \underbrace{\neg q(w)}_{\textit{is disjoint}}$$

- "propositional essence" of the problem delegated to SAT solver

AVATAR modulo Theories (since 2015)

The AVATAR architecture [Voronkov 2014]

- modern architecture of first-order theorem provers
- combines saturation with SAT-solving
- efficient realization of the *clause splitting rule*

$$\forall x, z, w.\ \underbrace{s(x) \vee \neg r(x, z)}_{share\ \ x\ \ and\ \ z} \vee \underbrace{\neg q(w)}_{is\ \ disjoint}$$

- "propositional essence" of the problem delegated to SAT solver

AVATAR modulo Theories [Reger et al. 2016]

- use an SMT solver instead of the SAT solver
- sub-problems considered are **ground-theory-consistent**
- implemented in Vampire using Z3

Does Vampire Need Instantiation?

Example

Consider the conjecture $(\exists x)(x + x \simeq 2)$ negated and clausified to

$$x + x \not\simeq 2.$$

It takes Vampire 15 s to solve using theory axioms deriving lemmas such as

$$x + 1 \simeq y + 1 \vee y + 1 \leq x \vee x + 1 \leq y.$$

Does Vampire Need Instantiation?

Example

Consider the conjecture $(\exists x)(x + x \simeq 2)$ negated and clausified to

$$x + x \not\simeq 2.$$

It takes Vampire 15 s to solve using theory axioms deriving lemmas such as

$$x + 1 \simeq y + 1 \vee y + 1 \leq x \vee x + 1 \leq y.$$

Heuristic instantiation would help, but normally any instance of a clause is immediately subsumed by the original!

Another Example

Example

Consider a problem containing

$$14x \neq x^2 + 49 \vee p(x)$$

It takes a long time to derive $p(7)$ whereas if we had guessed $x = 7$ we immediately get

$$14 \cdot 7 \neq 7^2 + 49 \vee p(7)$$

Another Example

Example

Consider a problem containing

$$14x \neq x^2 + 49 \vee p(x)$$

It takes a long time to derive $p(7)$ whereas if we had guessed $x = 7$ we immediately get

$$\begin{array}{ll} 14 \cdot 7 \neq 7^2 + 49 \vee p(7) & \\ \rightsquigarrow & \text{evaluate} \\ 98 \neq 98 \vee p(7) & \end{array}$$

Another Example

Example

Consider a problem containing

$$14x \neq x^2 + 49 \vee p(x)$$

It takes a long time to derive $p(7)$ whereas if we had guessed $x = 7$ we immediately get

$$\begin{array}{cl} 14 \cdot 7 \neq 7^2 + 49 \vee p(7) & \\ \rightsquigarrow & \text{evaluate} \\ 98 \neq 98 \vee p(7) & \\ \rightsquigarrow & \text{remove trivial inequality} \\ p(7) & \end{array}$$

How do we guess $x = 7$?

Another Example

Example

Consider a problem containing

$$14x \neq x^2 + 49 \vee p(x)$$

It takes a long time to derive $p(7)$ whereas if we had guessed $x = 7$ we immediately get

$$\begin{array}{cc}
14 \cdot 7 \neq 7^2 + 49 \vee p(7) & \\
\rightsquigarrow & \text{evaluate} \\
98 \neq 98 \vee p(7) & \\
\rightsquigarrow & \text{remove trivial inequality} \\
p(7) &
\end{array}$$

How do we guess $x = 7$?

Theory Instantiation

Instantiation which makes some theory literals immediately false

Theory Instantiation

Instantiation which makes some theory literals immediately false

As an inference rule

$$\frac{P \vee D}{D\theta}\ \mathit{TheoryInst}$$

where P contains only pure theory literals and $\neg P\theta$ is valid in $\mathcal{T}$

Theory Instantiation

Instantiation which makes some theory literals immediately false

As an inference rule

$$\frac{P \vee D}{D\theta}\ \mathit{TheoryInst}$$

where P contains only pure theory literals and $\neg P\theta$ is valid in $\mathcal{T}$

Implementation:

- Collect relevant pure theory literals $L_1, \ldots, L_n$
- Run an SMT solver on the ground $\neg P[\mathbf{x}] = \neg L_1 \wedge \ldots \wedge \neg L_n$
- If the SMT solver returns a model, transform it into a substitution θ and produce an instance

Instantiation which makes some theory literals immediately false

As an inference rule

$$\frac{P \vee D}{D\theta}\ \mathit{TheoryInst}$$

where P contains only pure theory literals and $\neg P\theta$ is valid in $\mathcal{T}$

Implementation:

- Collect relevant pure theory literals $L_1, \ldots, L_n$
- Run an SMT solver on the ground $\neg P[\mathbf{x}] = \neg L_1 \wedge \ldots \wedge \neg L_n$
- If the SMT solver returns a model, transform it into a substitution θ and produce an instance

Problems with Abstraction

- Suppose we want to resolve

$$r(14y)$$

$$\neg r(x^2 + 49) \lor p(x)$$

$\Rightarrow$ No pure literals

Problems with Abstraction

- Suppose we want to resolve

$$r(14y)$$
$$\neg r(x^2 + 49) \vee p(x)$$

$\Rightarrow$ No pure literals

- Abstract to

$$z \neq 14y \vee r(z)$$
$$u \neq x^2 + 49 \vee \neg r(u) \vee p(x)$$

- (We discuss abstraction more later)

- Instantiation undoes abstraction:

$$p(1, 5)$$
$$\rightsquigarrow \quad \text{abstract}$$
$$x \neq 1 \vee y \neq 5 \vee p(x, y)$$
$$\rightsquigarrow \quad \text{instantiate}$$

Updated Rule

$$\frac{P \vee D}{D\theta} \text{ theory instance}$$

- $P\theta$ unsatisfiable in the theory
- P pure
- P does not contain trivial literals

A literal is trivial if

- Form: $x \neq t$ (x not in t)
- Pure (only theory symbols)
- x only occurs in other trivial literals or other non-pure literals

Theory Reasoning Conclusion

Reasoning with quantifiers and theories is a very difficult problem that remains an active research area.

Quantifying over Functions/Predicates/Sets

There are some things we cannot say in first-order logic, for example the induction schema we saw earlier where we want to quantify over all predicates in the language.

Something we cannot express in first-order logic is reachability as we cannot capture the transitive closure of a relation in FOL. But we can in higher-order logic

$$\forall P(\forall x, y, z \left(\begin{array}{l} P(x,x) \wedge \\ (P(x,y) \wedge P(z,y) \rightarrow P(x,z)) \wedge \\ (R(x,y) \rightarrow P(x,y)) \end{array} \right) \rightarrow P(u,v))$$

In program specification/verification we often want to reason over objects representing programs and requirements (predicates on states). Again, something we cannot do directly in FOL.

Higher-order Logic

In first-order logic we have variables representing individuals and we can quantify over them.

In second-order logic we have variables representing sets of individuals (or functions on individuals) and we can quantify over them.

In third-order logic we have variables representing sets of sets of individuals...

We call second-order logic and above higher-order logic

λ-calculus

This is a calculus of functions. The standard building-block is the anonymous function $\lambda x.E[x]$ which can be read as a function that takes a value for x and evaluates E with x replaced by that value (similar to functions we're familiar with).

We can then have functions that take other functions as arguments and can return functions.

We can do 'computation' with λ-terms by applying β-reduction

$$((\lambda x.M)E) \rightarrow_\beta (M[x := E])$$

We can also do α-conversion and η-reduction.

Functions can then be applied to each other e.g.

$$(\lambda x.\lambda y.xy)(\lambda x.x) \rightarrow_\beta (\lambda y.(\lambda x.x)y) \rightarrow_\eta \lambda x.x$$

Other Examples in λ-calculus

The λ-calculus is Turing-complete e.g. we can use it to emulate Turing machines (and therefore any other model of computation).

We can encode numbers as *Church Numerals*

$$\text{one} = \lambda f.\lambda x.x \quad \text{two} = \lambda f.\lambda x.f\ x \quad \text{three} = \lambda f.\lambda x.f\ (f\ x)$$

and then operations on numbers

$$\begin{aligned} \text{succ} &= \lambda n.\lambda f.\lambda x.f\ (n\ f\ x) \\ \text{plus} &= \lambda m.\lambda n.\lambda f.\lambda x.m\ f\ (n\ f\ x) \\ \text{mult} &= \lambda m.\lambda n.\lambda f.m\ (n\ f) \end{aligned}$$

For example

$$\begin{array}{ll} \text{plus one two} & \\ (\lambda m.\lambda n.\lambda f.\lambda x.m\ f\ (n\ f\ x))(\lambda f.\lambda x.x)(\lambda f.\lambda x.f\ x) & \rightarrow_\beta \\ \lambda f.\lambda x.(\lambda f.\lambda x.x)\ f\ ((\lambda f.\lambda x.f\ x)\ f\ x)) & \rightarrow_\beta \\ \lambda f.\lambda x.f\ ((\lambda f.\lambda x.f\ x)\ f\ x)) & \rightarrow_\beta \\ \lambda f.\lambda x.f\ (f\ x)) & \end{array}$$

Simply-Typed λ-calculus

However, because it's Turing-complete we can also write some odd things in it e.g.

$$\Omega = (\lambda x.x\ x)\ (\lambda x.x\ x)$$

This is odd as when we try and reduce it we get

$$(\lambda x.x\ x)\ (\lambda x.x\ x) \to_\beta (\lambda x.x\ x)\ (\lambda x.x\ x)$$

e.g. non-termination.

We don't want to allow x to apply to itself, we fix this with types (see Russel's Paradox).

A function $\lambda x.x$ has type $\alpha \to \alpha$ e.g. takes things of one type and returns something of the same type. Similarly, $\lambda f.\lambda x.f\ (f\ x))$ takes an f of $\alpha \to \alpha$ and an x of α and returns an α so its type is $(\alpha \to \alpha) \to \alpha \to \alpha$.

We cannot give a type to Ω.

Higher-Order Logic with λ-terms

We can define the syntax of higher-order logic with λ-terms as

Types	$A ::= K \mid A \to A$	(K is an atomic type)
Terms	$t ::= x \mid c \mid t_1 @ t_2 \mid \lambda x_A.t$	(c is a nullary function)
Formulas	$f ::= t_1 = t_2 \mid \forall x : A.f \mid \neg f \mid f_1 \wedge f_2$	

e.g. we use an extra syntax for application and add types to λs and $\forall$s.

We can now write statements such as

$$\forall f_1 : a \to b.\forall f_2 : a \to b.((\forall x : a.f_1 @ a = f_2 @ a) \to f_1 = f_2)$$

e.g. functional extensionality.

An extension to this adds quantifiers over types - this is Polymorphic HOL and uses polymorphic λ-calculus.

Combinatory Logic

An important result from Haskell Curry was the introduction of the SKI combinator calculus.

This introduces three combinators

$$\begin{array}{lcl} \mathbf{I} & \equiv & \lambda x.x \\ \mathbf{K} & \equiv & \lambda x.\lambda y.x \\ \mathbf{S} & \equiv & \lambda x.\lambda y.\lambda z.(x\ z\ (y\ z)) \end{array}$$

such that **all** λ-expressions can be represented as a sequence of applied combinators.

Notice that one is **I** but two is (**S** (**S** (**K** **S**) **K**)) **I** - combinators are not a succinct representation!

Reasoning in Higher-order Logic

Standard methods for reasoning in it are either...

Interactive e.g. using a proof assistant to allow a human to make reasoning steps (and suggesting possibly good reasoning steps)

or

Approximate by a translation to first-order logic that preserves inconsistency (this is what we do in Vampire)

The translation:

- Translates to applicative form e.g. $t_1 @ t_2$ becomes $\mathrm{app}(t_1, t_2)$
- Replaces λ-terms with combinators
- Replaces logical symbols inside terms with equivalent functions

To get a problem in first-order logic.

Less Expressive/More Efficient

Find things that are decidable but still usefully expressive:

- Propositional logic (QBF, PLFD)
- Well-behaved First-order fragments
- (Some) Description Logics
- (Some) Modal Logics

QBF and PLFD

Quantified Boolean Formulas

We quantify over booleans e.g. $\forall x.\exists y.(x \rightarrow y)$. If there are no free-variables then the formula is either true or false.

Propositional Logic with Finite Domains

Variables have domains. Atoms are now of the form $x = v$ where v is in the domain of x e.g. $\mathit{food} = \text{none} \rightarrow \neg \mathit{emotion} = \text{happy}$.

Both logics are more succinct than propositional logic but logically equivalent. However, in practice, problems in QBF or PLFD can be easier to solve than equivalent ones in PL.

Quantification in QBF is different from that in FOL as in FOL we always quantifier over all inhabitants of the universe (or the type if we have types).

Some Decidable Fragments of FOL

Monadic Fragment
Every predicate has arity at most 1 and there are no function symbols.

Two-variable Fragment
The formula can be written using at most 2 variables.

Guarded Fragment
If the formula is built using $\neg$ and $\wedge$, or is of the form $\exists \overline{x}.(G[\overline{y}] \wedge \phi[\overline{z}])$ such that G is an atom and $\overline{z} \subseteq \overline{y}$. Intuitively all usage of variables are *guarded* by a something positive.

Prenex Fragments
If a function-free formula is in prenex normal form and can be written as $\exists^*\forall^*.F$ it is in the BernaysSchönfinkel fragment.

The following logics often target these fragments to ensure decidability.

Description Logic

A family of logics that are usually decidable. They are used for describing ontologies. The terminology is different from what we're used to.

In description logic we separate facts in the $\mathcal{A}$-Box and rules in the $\mathcal{T}$-Box

Individuals belong to Concepts and may be related by Roles.

Concepts are sets of elements (unary predicates)

Roles relate two individuals (binary relations/predicates)

Complex concepts are logical combinations of concepts/roles

Facts assert individuals belong to concepts or roles

Rules capture relationships between complex concepts

Description Logic: Concepts

The most basic description logic is $\mathcal{ALC}$, which stands for *Attributive Concept Language with Complements*.

In $\mathcal{ALC}$ we have the following complex concepts:

- $A \sqcap B$: things that are A and B
- $A \sqcup B$: things that are A or B
- $\neg A$: things that are not A
- $\exists r.C$ things that are related by r to things that are C
- $\forall r.C$ things where all r related things are C

Examples:

Somebody that has a human child	$\exists$hasChild.Human
Somebody that only drinks beer	$\forall$drinks.Beer
Somebody that is either French or knows somebody who is	French $\sqcup$ $\exists$knows.French

Rules and Reasoning

Rules are of the form $C \sqsubseteq D$ e.g. everything that is a C is also a D

$C \equiv D$ is the same as $C \sqsubseteq D$ and $D \sqsubseteq C$

e.g. $\mathsf{Father} \equiv \mathsf{Man} \sqcap \exists \mathsf{hasChild}.\mathsf{Human}$

An ontology is a set of facts and rules ($\mathcal{A}$-box and $\mathcal{T}$-box)

The semantics are defined in terms of interpretations (should be familiar)

Standard reasoning problems include

- Is an ontology consistent
- Is an individual in a concept (entailment)
- Is one concept subsumed by another (entailment)

Embedding in FOL

Introduce translation function t_x that maps into FOL formula with free x

Concepts map directly to FOL predicts, $t_x(A) = A(x)$

Complex Concepts map to logical combinations e.g.

$$t_x(\exists r.C) = \exists y.r(x, y) \wedge t_y(C)$$

The resulting FOL formulas are in the two-variable fragment and the guarded fragment.

This is for the simplest description logic $\mathcal{ALC}$. There are lots of more complicated description logics with extra features where the translation is less straightforward. Some DLs are more expressive than FOL.

Propositional Modal Logic

It would be nice to be able to not only talk about what is true but when it is true (when in a general sense)

Modal logic allows us to do this. In English a *modal* qualifies a statement.

In modal logic we typically have two modal operators $\Diamond$ and $\Box$

Traditionally $\Diamond P$ means *Possibly P* whereas $\Box P$ means *Necessarily P*

For example,

$$\neg\Diamond win \rightarrow \Box\neg win$$
$$\Box(rain \wedge wind) \rightarrow \Box rain$$
$$(\Diamond rain \wedge \Diamond wind) \rightarrow \Diamond(rain \vee wind)$$

If you took Logic and Modelling you met LTL, which is a modal logic.

Semantics and Flavours

The semantics of modal logic is given by something called a Kripke structure, which is really just a graph. We have a relation R between worlds where different propositions are true in each world. $\Box$ then means *in all worlds adjacent by* R and $\Diamond$ means *in some worlds adjacent by* r.

We get different kinds of modal logic depending on how we control r, or equivalently which axioms about $\Box$ and $\Diamond$ we assume.

For example, reflexivity of R or $\Box A \rightarrow A$, and transitivity of R or $\Box A \rightarrow \Box\Box A$ gives us temporal logic where $\Box$ means all futures and $\Diamond$ means some

Other popular flavours in AI (particularly agent-based reasoning) are epistemic logic where modalities correspond to knowledge and doxastic logic where they correspond to belief.

Embedding in FOL

We use the adjacency relation R and a predicate $holds(x, y)$ that is true if x is true in world y

We can then encoding the meaning of modal formulas e.g.

$$holds(\Box p, u) \leftrightarrow \forall v.(R(u, v) \rightarrow holds(p, v))$$

The satisfiability/validity of a modal formula is the existential/universal closure of the resulting translation

We also need to add the axioms e.g. reflexivity and transitivity of R

We actually have to do quite a bit of extra work to get things into a decidable fragment, but we can.

Decidability does not necessarily mean more efficient

A decision procedure is good because it will terminate in finite time, this does not mean that time is short.

Many fragments/logics also have strong complexity bounds on specific reasoning problems, which can help. But these are upper bounds.

In practice, it might be that a less efficient method, or an incomplete one, solves particular instances of problems faster.

This is generally our experience with Vampire.

Summary

This week we have seen

- Higher-Order Logic
- Fragments of First-Order Logic
- Description Logic
- Propositional Modal logic

and translations of all of the above into FOL (if not already in FOL)

Next week: Andre will talk to you about Inductive Logic Programming