Lectures 14 and 15 (Week 9)
Advanced Reasoning with FOL

COMP24412: Symbolic AI

Giles Reger

March 2020

This Week

Going from formulas to clauses

Organisation saturation using the Given Clause Algorithm

A more optimal Ordered Resolution

Soundness and Completeness of (Ordered) Resolution

More optimisations in preprocessing and proof search

How to use Vampire

Aim and Learning Outcomes

The aim of these two lectures is to:

Understand how we can efficiently check entailments for general first-order knowledge bases

Learning Outcomes

By the end you will be able to:

1. Translate an arbitary first-order formulas to a set of clauses
2. Recall and describe key terminology (e.g. clause, saturation, ...)
3. Describe (with examples) the given clause algorithm for saturation
4. Explain and apply <u>ordered</u> resolution
5. Describe why ordered resolution is sound and complete
6. Use Vampire to check entailments

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

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

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

Before applying resolution we need to *name-apart* the premises

We will use resolution for refutational based reasoning.

Recall $\Gamma \models \phi$ if and only if $\Gamma \cup \{\neg\phi\} \models \mathit{false}$. We will saturate $\Gamma \cup \{\neg\phi\}$ until there is nothing left to add or we have derived *false*.
(this doesn't necessarily terminate as FOL is semi-decidable).

Reminder: Naming Apart

Are these two clauses consistent?

$$p(x, a) \qquad\qquad \neg p(b, x)$$

Reminder: Naming Apart

Are these two clauses consistent?

$$p(x, a) \qquad \neg p(b, x)$$

Trying to resolve them requires us to unify $p(x, a)$ with $p(b, x)$, can we?

Reminder: Naming Apart

Are these two clauses consistent?

$$p(x, a) \qquad \neg p(b, x)$$

Trying to resolve them requires us to unify $p(x, a)$ with $p(b, x)$, can we?

But they represent $(\forall x.p(x, a)) \wedge (\forall x.p(b, x))$, which is unsat

Reminder: Naming Apart

Are these two clauses consistent?

$$p(x, a) \qquad \neg p(b, x)$$

Trying to resolve them requires us to unify $p(x, a)$ with $p(b, x)$, can we?

But they represent $(\forall x.p(x, a)) \wedge (\forall x.p(b, x))$, which is unsat

Equivalent to $(\forall x.p(x, a)) \wedge (\forall y.p(b, y))$ i.e. clauses $p(x, a)$ and $\neg p(b, y)$

Hidden rule: Naming Apart.
When unifying literals from two different clauses you should first rename variables so that the literals do not share literals.

Sometimes I do this implicitly e.g. in $p(x)$ and $\neg p(x)$ we should rename to $p(x)$ and $\neg p(y)$ and then apply $\{x \mapsto y\}$ but that's tedious.

An aside about terminology: Theorem Proving

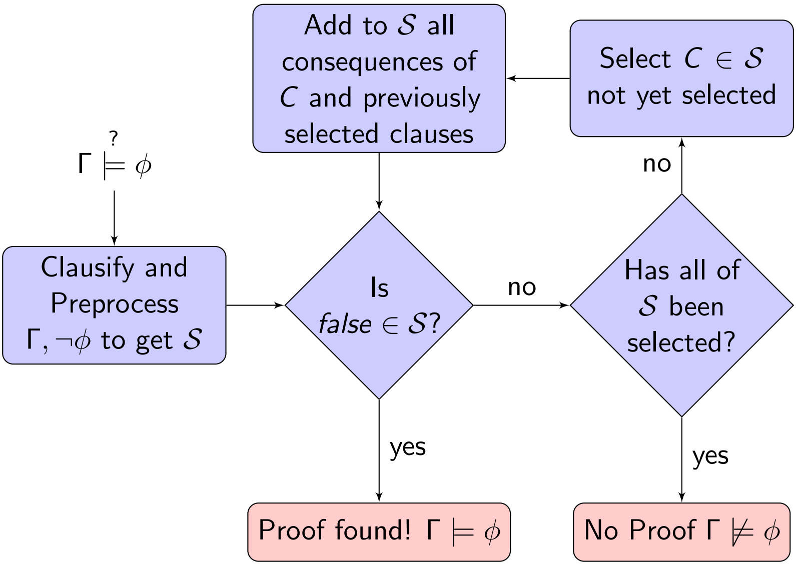
What is this Theorem Proving?

It is a subfield of automated reasoning that focuses on finding proofs of entailment.

The terminology is a bit different due to roots in mathematical logic.

What we are referring to as a knowledge base is referred to as a theory and any statement entailed by a theory is a theorem of that theory, hence theorem proving.

Architecture of a (Saturation-Based) Theorem Prover



Saturation Based Reasoning

To decide $\Gamma \models \varphi$ we are going to follow the below steps

We split $\mathcal{S}$ into two parts to keep track of selected and not selected.

1. Let *NotDone* $= \Gamma \cup \{\neg\varphi\}$ and *Done* be an empty set of clauses
2. Transform *NotDone* into clauses
3. If *NotDone* contains *false* then **return valid**
4. Select a clause C from *NotDone*
5. Perform all inferences (e.g. resolution) between C and all clauses in *Done* putting any (useful) *new* children in *NotDone*
6. Move C to *Done*
7. If *NotDone* is not empty stop go to 3
8. **return not valid**

Saturation Based Reasoning

To decide $\Gamma \models \varphi$ we are going to follow the below steps

We split $\mathcal{S}$ into two parts to keep track of selected and not selected.

1. Let $\mathit{NotDone} = \Gamma \cup \{\neg\varphi\}$ and *Done* be an empty set of clauses
2. Transform *NotDone* into clauses
3. If *NotDone* contains *false* then **return valid**
4. Select a clause C from *NotDone*
5. Perform all inferences (e.g. resolution) between C and all clauses in *Done* putting any (useful) *new* children in *NotDone*
6. Move C to *Done*
7. If *NotDone* is not empty stop go to 3
8. **return not valid**

Transformation to Clausal Form

To go from general formula to set of clauses we're going to go through the following steps

1. Rectify the formula
2. Transform to *Negation Normal Form*
3. Eliminate quantifiers
4. Transform into conjunctive normal form
5. Eliminate equality

At any stage we might simplify using rules about *true* and *false*, e.g. $\mathit{false} \wedge \phi = \mathit{false}$, and remove tautologies, e.g. $p \vee p$.

Rectification

(Recall point in Week 7 about renaming variables in quantified formulas)

A formula is rectified if each quantifier binds a different variable and all bound variables are distinct from free ones.

To rectify a formula we identify any name clashes, pick one and rename it consistently. It is important to work out which variables are in the scope of a quantifier e.g.

$$\forall x.(p(x) \vee \exists x.r(x)) \qquad \textit{becomes} \qquad \forall x(p(x) \vee \exists y.r(y))$$

To automate this we typically rename bound variables starting with x_0 etc.

Negation Normal Form

Apply rules in a completely deterministic syntactically-guided way

$$
\begin{array}{rcl}
\neg(F_1 \wedge \ldots \wedge F_n) & \Rightarrow & \neg F_1 \vee \ldots \vee \neg F_n \\
\neg(F_1 \vee \ldots \vee F_n) & \Rightarrow & \neg F_1 \wedge \ldots \wedge \neg F_n \\
F_1 \rightarrow F_2 & \Rightarrow & \neg F_1 \vee F_2 \\
\neg\neg F & \Rightarrow & F \\
\neg \forall x_1, \ldots, x_n F & \Rightarrow & \exists x_1, \ldots, x_n \neg F \\
\neg \exists x_1, \ldots, x_n F & \Rightarrow & \forall x_1, \ldots, x_n \neg F \\
\neg(F_1 \leftrightarrow F_2) & \Rightarrow & F_1 \otimes F_2 \\
\neg(F_1 \otimes F_2) & \Rightarrow & F_1 \leftrightarrow F_2 \\
F_1 \leftrightarrow F_2 & \Rightarrow & (F_1 \rightarrow F_2) \wedge (F_2 \rightarrow F_1); \\
F_1 \otimes F_2 & \Rightarrow & (F_1 \vee F_2) \wedge (\neg F_1 \vee \neg F_2).
\end{array}
$$

Where $\otimes$ is exclusive or. Can get an exponential increase in size.

Examples

Let's do these by hand and then later see what Vampire does.

$$(\forall x.p(x)) \rightarrow (\exists x.p(x))$$

$$(\forall x.(p(x) \vee q(x)) \leftrightarrow (\neg\exists x.(\neg p(x) \wedge \neg q(x)))$$

$$\forall x, y, z.((f(x) = y \wedge f(x) = z) \rightarrow y = z)$$

$$\forall x.((\exists y.p(x, y)) \rightarrow q(x)) \wedge p(a, b) \wedge \neg\exists x.q(x)$$

Dealing with Existential Quantifiers

There is a best kind of pizza
$\exists x.\forall y.((pizza(x) \wedge pizza(y) \wedge x \neq y) \rightarrow better(x, y))$

There are two different people who live in the same house
$\exists x.\exists y.\exists z.(x \neq y \wedge lives\_in(x, z) \wedge lives\_in(y, z))$

Everybody loves somebody
$\forall x.\exists y.loves(x, y)$

Dealing with Existential Quantifiers

There is a best kind of pizza
$\exists x.\forall y.((pizza(x) \wedge pizza(y) \wedge x \neq y) \rightarrow better(x, y))$

There are two different people who live in the same house
$\exists x.\exists y.\exists z.(x \neq y \wedge lives\_in(x, z) \wedge lives\_in(y, z))$

Everybody loves somebody
$\forall x.\exists y.loves(x, y)$

Let $\mathcal{I}$ be some interpretation. If $\mathcal{I}(\exists x.\phi[x])$ is true then there must be some domain constant d such that $\mathcal{I}(\phi[d])$ is true.

Dealing with Existential Quantifiers

There is a best kind of pizza
$\exists x.\forall y.((pizza(x) \wedge pizza(y) \wedge x \neq y) \rightarrow better(x, y))$

There are two different people who live in the same house
$\exists x.\exists y.\exists z.(x \neq y \wedge lives\_in(x, z) \wedge lives\_in(y, z))$

Everybody loves somebody
$\forall x.\exists y.loves(x, y)$

Let $\mathcal{I}$ be some interpretation. If $\mathcal{I}(\exists x.\phi[x])$ is true then there must be some domain constant d such that $\mathcal{I}(\phi[d])$ is true.

Let a be a fresh constant symbol (a new name for the object that has to exist). As it is fresh we can freely let $\mathcal{I}(a) = d$.
(This requires the Axiom of Choice)

Dealing with Existential Quantifiers

There is a best kind of pizza
$\exists x.\forall y.((\mathit{pizza}(x) \wedge \mathit{pizza}(y) \wedge x \neq y) \rightarrow \mathit{better}(x, y))$

There are two different people who live in the same house
$\exists x.\exists y.\exists z.(x \neq y \wedge \mathit{lives\_in}(x, z) \wedge \mathit{lives\_in}(y, z))$

Everybody loves somebody
$\forall x.\exists y.\mathit{loves}(x, y)$

Let $\mathcal{I}$ be some interpretation. If $\mathcal{I}(\exists x.\phi[x])$ is true then there must be some domain constant d such that $\mathcal{I}(\phi[d])$ is true.

Let a be a fresh constant symbol (a new name for the object that has to exist). As it is fresh we can freely let $\mathcal{I}(a) = d$.
(This requires the Axiom of Choice)

Dealing with Existential Quantifiers

There is a best kind of pizza
$\exists x.\forall y.(\neg pizza(x) \vee \neg pizza(y) \vee x = y \vee better(x, y))$

There are two different people who live in the same house
$\exists x.\exists y.\exists z.(x \neq y \wedge lives\_in(x, z) \wedge lives\_in(y, z))$

Everybody loves somebody
$\forall x.\exists y.loves(x, y)$

Let $\mathcal{I}$ be some interpretation. If $\mathcal{I}(\exists x.\phi[x])$ is true then there must be some domain constant d such that $\mathcal{I}(\phi[d])$ is true.

Let a be a fresh constant symbol (a new name for the object that has to exist). As it is fresh we can freely let $\mathcal{I}(a) = d$.
(This requires the Axiom of Choice)

Dealing with Existential Quantifiers

There is a best kind of pizza
$\forall y.(\neg pizza(a) \vee \neg pizza(y) \vee a = y \vee better(a, y))$

There are two different people who live in the same house
$\exists x.\exists y.\exists z.(x \neq y \wedge lives\_in(x, z) \wedge lives\_in(y, z))$

Everybody loves somebody
$\forall x.\exists y.loves(x, y)$

Let $\mathcal{I}$ be some interpretation. If $\mathcal{I}(\exists x.\phi[x])$ is true then there must be some domain constant d such that $\mathcal{I}(\phi[d])$ is true.

Let a be a fresh constant symbol (a new name for the object that has to exist). As it is fresh we can freely let $\mathcal{I}(a) = d$.
(This requires the Axiom of Choice)

Dealing with Existential Quantifiers

There is a best kind of pizza
$\forall y.(\neg pizza(a) \vee \neg pizza(y) \vee a = y \vee better(a, y))$

There are two different people who live in the same house
$\exists x.\exists y.\exists z.(x \neq y \wedge lives\_in(x, z) \wedge lives\_in(y, z))$

Everybody loves somebody
$\forall x.\exists y.loves(x, y)$

Let $\mathcal{I}$ be some interpretation. If $\mathcal{I}(\exists x.\phi[x])$ is true then there must be some domain constant d such that $\mathcal{I}(\phi[d])$ is true.

Let a be a fresh constant symbol (a new name for the object that has to exist). As it is fresh we can freely let $\mathcal{I}(a) = d$.
(This requires the Axiom of Choice)

Dealing with Existential Quantifiers

There is a best kind of pizza
$\forall y.(\neg pizza(a) \vee \neg pizza(y) \vee a = y \vee better(a, y))$

There are two different people who live in the same house
$a \neq b \wedge lives\_in(a, c) \wedge lives\_in(b, c)$

Everybody loves somebody
$\forall x.\exists y.loves(x, y)$

Let $\mathcal{I}$ be some interpretation. If $\mathcal{I}(\exists x.\phi[x])$ is true then there must be some domain constant d such that $\mathcal{I}(\phi[d])$ is true.

Let a be a fresh constant symbol (a new name for the object that has to exist). As it is fresh we can freely let $\mathcal{I}(a) = d$.
(This requires the Axiom of Choice)

Dealing with Existential Quantifiers

There is a best kind of pizza
$\forall y.(\neg pizza(a) \vee \neg pizza(y) \vee a = y \vee better(a, y))$

There are two different people who live in the same house
$a \neq b \wedge lives\_in(a, c) \wedge lives\_in(b, c)$

Everybody loves somebody
$\forall x.\exists y.loves(x, y)$

Dealing with Existential Quantifiers

There is a best kind of pizza
$\forall y.(\neg pizza(a) \vee \neg pizza(y) \vee a = y \vee better(a, y))$

There are two different people who live in the same house
$a \neq b \wedge lives\_in(a, c) \wedge lives\_in(b, c)$

Everybody loves somebody
$\forall x.\exists y.loves(x, y)$

Let $\mathcal{I}$ be some interpretation. If $\mathcal{I}(\forall x.\exists y.\phi[x, y])$ is true then for every domain constant d_1 there must be some other domain constant d_2 such that $\mathcal{I}(\phi[d_1, d_2])$ is true. But how do we find d_2 given d_1?

Let f be a fresh function symbol whose interpretation can be made to *select* the necessary domain constant.

Dealing with Existential Quantifiers

There is a best kind of pizza
$\forall y.(\neg pizza(a) \vee \neg pizza(y) \vee a = y \vee better(a, y))$

There are two different people who live in the same house
$a \neq b \wedge lives\_in(a, c) \wedge lives\_in(b, c)$

Everybody loves somebody
$\forall x.\exists y.loves(x, y)$

Let $\mathcal{I}$ be some interpretation. If $\mathcal{I}(\forall x.\exists y.\phi[x, y])$ is true then for every domain constant d_1 there must be some other domain constant d_2 such that $\mathcal{I}(\phi[d_1, d_2])$ is true. But how do we find d_2 given d_1?

Let f be a fresh function symbol whose interpretation can be made to *select* the necessary domain constant.

Dealing with Existential Quantifiers

There is a best kind of pizza
$\forall y.(\neg pizza(a) \vee \neg pizza(y) \vee a = y \vee better(a, y))$

There are two different people who live in the same house
$a \neq b \wedge lives\_in(a, c) \wedge lives\_in(b, c)$

Everybody loves somebody
$\forall x.loves(x, f(x))$

Let $\mathcal{I}$ be some interpretation. If $\mathcal{I}(\forall x.\exists y.\phi[x, y])$ is true then for every domain constant d_1 there must be some other domain constant d_2 such that $\mathcal{I}(\phi[d_1, d_2])$ is true. But how do we find d_2 given d_1?

Let f be a fresh function symbol whose interpretation can be made to *select* the necessary domain constant.

Dealing with Universal Quantifiers

Forget about them

Skolemisation

This process is called Skolemisation and the new symbols we introduce are called Skolem constants or Skolem functions.

The rules are simply

$$
\begin{array}{lcl}
\forall x_1, \ldots, x_n F & \Rightarrow & F \\
\exists x_1, \ldots, x_n F & \Rightarrow & F\{x_1 \mapsto f_1(y_1, \ldots, y_m), \ldots, x_n \mapsto f_n(y_1, \ldots, y_m)\},
\end{array}
$$

where f_i are fresh function symbols of the correct arrity and $y_1, \ldots y_m$ are the free variables of F e.g. they are the things universally quantified in the larger scope.

Remember that Skolem constants/functions act as witnesses for something that we know has to exist for the formula to be true.

Validity or Satisfiability Preserving?

Do these two formulas have the same models?

$$\exists x.\forall y.p(x,y) \qquad \forall y.p(a,y)$$

Validity or Satisfiability Preserving?

Do these two formulas have the same models?

$$\exists x.\forall y.p(x, y) \qquad \forall y.p(a, y)$$

No - the first one does not need to interpret a

Validity or Satisfiability Preserving?

Do these two formulas have the same models?

$$\exists x.\forall y.p(x, y) \qquad \forall y.p(a, y)$$

No - the first one does not need to interpret a

Do the formulas both have models?

Validity or Satisfiability Preserving?

Do these two formulas have the same models?

$$\exists x.\forall y.p(x, y) \qquad \forall y.p(a, y)$$

No - the first one does not need to interpret a

Do the formulas both have models?

Yes - there's no negation, just make everything true

Validity or Satisfiability Preserving?

Do these two formulas have the same models?

$$\exists x.\forall y.p(x,y) \qquad \forall y.p(a,y)$$

No - the first one does not need to interpret a

Do the formulas both have models?

Yes - there's no negation, just make everything true

When we introduce new symbols we preserve satisfiability (the existence of models) not validity (the same models).

However, most transformations are stronger than this and preserve models on the initial signature.

CNF Transformation

CNF stands for Clausal Normal Form (or Conjunctive Normal Form, similar but different)

Now the only connectives left should be $\wedge$ and $\vee$ and we need to push the $\vee$ symbols under the $\wedge$ symbols

The associated rule is

$$(A_1 \wedge \ldots \wedge A_m) \vee B_1 \vee \ldots \vee B_n \quad \Rightarrow \quad \begin{array}{cc} (A_1 \vee B_1 \vee \ldots \vee B_n) & \wedge \\ \cdots & \wedge \\ (A_m \vee B_1 \vee \ldots \vee B_n). & \end{array}$$

This can lead to exponential growth (see optimisation in a bit).

Eliminating Equality

At the moment we're just dealing with the Resolution Calculus. Next week I'll show you how to deal with equality natively but before we do this we need to get rid of equality.

As outlined in Week 7, we add a new equality predicate eq and add clauses capturing reflexivity, symmetry, and transitivity of equality:

$$\mathsf{eq}(x,x) \qquad \neg\mathsf{eq}(x,y) \vee \mathsf{eq}(y,x) \qquad \neg\mathsf{eq}(x,y) \vee \neg\mathsf{eq}(y,z)) \vee \mathsf{eq}(x,z)$$

And clauses representing congruence of functions and predicates:

$$\neg\mathsf{eq}(x_1,y_1) \vee \ldots \vee \neg\mathsf{eq}(x_n,y_n) \vee \mathsf{eq}(f(x_1,\ldots x_n),f(y_1,\ldots,y_n)))$$
$$\neg\mathsf{eq}(x_1,y_1) \vee \ldots \vee \neg\mathsf{eq}(x_n,y_n) \vee \neg p(x_1,\ldots x_n) \vee p(y_1,\ldots,y_n))$$

for all functions f and predicates p of arity n

Vampire

Automated theorem prover for first-order logic

Is a refutational saturation-based theorem prover implementing resolution

Very efficient/powerful (wins lots of competitions)

Used as a back-box solver in lots of other things (in academia and industry)

Available from `https://vprover.github.io` (and I'll provide a direct download link)

You will get to use Vampire in Coursework 3.

TPTP

Language for describing first-order formulas in ASCII format.

See `http://tptp.cs.miami.edu/~tptp/`

For example

```
fof(one,axiom, ![X] : (happy(X) <=> (?[Y] : loves(X,Y)))).
fof(two,axiom, ![X] : (rich(X) => loves(X,money))).
fof(three,axiom, rich(giles)).
fof(goal,conjecture, happy(giles)).
```

Important - `axiom` for knowledge base, `conjecture` for goal and Vampire will do the negation for you.

Looking at Vampire's clausification

Run

```
./vampire --mode clausify --equality_proxy RSTC problem
```

on problem file `problem`

It will sometimes do a lot more than what we've discussed above.

Vampire performs lots of optimisations e.g. naming subformulas, removing pure symbols, or unused definitions.

Examples

Let's do these by hand and then see what Vampire does.

$$(\forall x.p(x)) \rightarrow (\exists x.p(x))$$

$$(\forall x.(p(x) \vee q(x)) \leftrightarrow (\neg\exists x.(\neg p(x) \wedge \neg q(x)))$$

$$\forall x, y, z.((f(x) = y \wedge f(x) = z) \rightarrow y = z)$$

$$\forall x.((\exists y.p(x, y)) \rightarrow q(x)) \wedge p(a, b) \wedge \neg\exists x.q(x)$$

Also, which of the above statements are valid or inconsistent? How do we ask Vampire?

Optimisation (subformula naming)

To go from general formula to set of clauses we're going to go through the following steps

1. Rectify the formula
2a. Transform to *Equivalence Negation Form*
2b. (Optimisation) Apply *naming* of subformulas
2c. Transform to *Negation Normal Form*
3. Eliminate quantifiers
4. Transform into conjunctive normal form
5. Eliminate equality

We first transform into ENF (halfway to NNF) to allow us to perform the naming optimisation

Equivalence Negation Form

Push negations in but preserve equivalences e.g. drop last two rules of NNF transformation.

$$
\begin{array}{rcl}
\neg(F_1 \wedge \ldots \wedge F_n) & \Rightarrow & \neg F_1 \vee \ldots \vee \neg F_n \\
\neg(F_1 \vee \ldots \vee F_n) & \Rightarrow & \neg F_1 \wedge \ldots \wedge \neg F_n \\
F_1 \rightarrow F_2 & \Rightarrow & \neg F_1 \vee F_2 \\
\neg\neg F & \Rightarrow & F \\
\neg\forall x_1, \ldots, x_n F & \Rightarrow & \exists x_1, \ldots, x_n \neg F \\
\neg\exists x_1, \ldots, x_n F & \Rightarrow & \forall x_1, \ldots, x_n \neg F \\
\neg(F_1 \leftrightarrow F_2) & \Rightarrow & F_1 \otimes F_2 \\
\neg(F_1 \otimes F_2) & \Rightarrow & F_1 \leftrightarrow F_2
\end{array}
$$

Only get a linear increase in size.

Subformula Naming

We want to get to a conjunction of disjunctions but this process can 'blow up' in general e.g.

$$p(x,y) \leftrightarrow (q(x) \leftrightarrow (p(y,y) \leftrightarrow q(y))),$$

is equivalent to

$$\begin{gathered}
p(x,y) \vee \neg q(y) \vee \neg p(y,y) \vee \neg q(x)) \\
p(x,y) \vee q(y) \vee p(y,y) \vee \neg q(x)) \\
p(x,y) \vee p(y,y) \vee \neg q(y) \vee q(x)) \\
p(x,y) \vee q(y) \vee \neg p(y,y) \vee q(x)) \\
q(x) \vee \neg q(y) \vee \neg p(y,y) \vee \neg p(x,y)) \\
q(x) \vee q(y) \vee p(y,y) \vee \neg p(x,y)) \\
p(y,y) \vee \neg q(y) \vee \neg q(x) \vee \neg p(x,y)) \\
q(y) \vee \neg p(y,y) \vee \neg q(x) \vee \neg p(x,y))
\end{gathered}$$

Subformula Naming

We can replace

$$p(x,y) \leftrightarrow (q(x) \leftrightarrow (p(y,y) \leftrightarrow q(y))),$$

by

$$\begin{array}{l} p(x,y) \leftrightarrow (q(x) \leftrightarrow n(y)); \\ n(y) \leftrightarrow (p(y,y) \leftrightarrow q(y)). \end{array}$$

to get the same number of clauses but each clause is simpler (better for reasoning).

In the case when the subformula $F(x_1,\ldots,x_k)$ has only positive occurrences in G, one can use the axiom $n(x_1,\ldots,x_k) \rightarrow F(x_1,\ldots,x_k)$ instead of $n(x_1,\ldots,x_k) \leftrightarrow F(x_1,\ldots,x_k)$. This will lead to fewer clauses.

Subformula Naming

Assigning a name n to $F_2 \leftrightarrow F_3$ yields two formulas

$$\begin{aligned} &F_1 \leftrightarrow n; \\ &n \leftrightarrow (F_2 \leftrightarrow F_3), \end{aligned}$$

where the second formula has the same structure as the original formula $F_1 \leftrightarrow (F_2 \leftrightarrow F_3)$.

When to Name Subformulas?

Vampire uses a heuristic that that estimates how many clauses a subformula will produce and names that subformula if that number is above a certain threshold.

Naming can not increase the number of clauses introduced but does not always reduce.

For those of you who took Logic and Modelling. The idea is the same as the optimised structural transformation in the propositional case but we don't always apply it as the cost on reasoning is much higher here.

Removing Unnecessary Things

Is this Knowledge Base consistent?

```
fof(a1, axiom, ![X] : (winner(X) <=> ?[Y] : wins(Y) = X)).
fof(a2, axiom, ![X] : (wins(X) = position(X,first))).
fof(a3, axiom, ![X] : (takes_part(X,Y) | ~takes_part(X,Y))).
fof(a4, axiom, ![X,Y] : ((wins(X) = Y) => enters(X,Y))).
```

Removing Unnecessary Things

Is this Knowledge Base consistent?

```
fof(a1, axiom, ![X] : (winner(X) <=> ?[Y] : wins(Y) = X)).
fof(a2, axiom, ![X] : (wins(X) = position(X,first))).
fof(a3, axiom, ![X] : (takes_part(X,Y) | ~takes_part(X,Y))).
fof(a4, axiom, ![X,Y] : ((wins(X) = Y) => enters(X,Y))).
```

We can remove

- Tautologies - formulas that are true in every model
- Definitions of things that are unused
- References to predicates that are used with a single polarity as we know how to make this predicate true.

Try running `vampire -explain updr` and `vampire -explain fde`

Organisation Saturation

Now we have our problem as a set of clauses we need to see if it is consistent or not.

Recall, we do this by saturation.

This is similar to the forward-chaining algorithm but we want to avoid the problems with forward-chaining.

One way of doing this has already been to add the negation of the goal/query into the clause set, thus making the reasoning more goal-directed

The next thing to do is organise things as a best-first rather than breadth-first search.

Given Clause Algorithm

I've renamed things again! I use the term *active* and *passive* as these are the terms used within Vampire (for historic reasons).

input: *Init*: set of clauses;
var *active*, *passive*, *unprocessed*: set of clauses;
var *given*, *new*: clause;
$active := \emptyset$; $unprocessed := Init$;
loop
  while $unprocessed \neq \emptyset$
    $new := pop(unprocessed)$;
    if $new = false$ **then return** *unsatisfiable*;
    add *new* to *passive*
  if $passive = \emptyset$ **then return** *satisfiable* or *unknown*
  $given := select(passive)$; *(* clause selection *)*
  move *given* from *passive* to *active*;
  $unprocessed := infer(given, active)$; *(* generating inferences *)*

Clause Selection and Fairness

What if we infinitely delay performing an inference?

We lose the partial decidability

A saturation process is fair if no clause is delayed infinitely often

Two fair clause selection strategies:

- First-in first-out. This is the same as forward-chaining and whilst fair is not very efficient
- Smallest (in number of symbols) first

  (there are a finite number of terms with at most k symbols)

  The intuition is that smaller clauses are 'closer' to the empty clause

Vampire uses an age-weight ratio, alternately selecting based on age (first-in first-out) and weight (number of symbols).

Complete Example

$$\left\{ \begin{array}{c} \forall x.(happy(x) \leftrightarrow \exists y.(loves(x,y))) \\ \forall x.(rich(x) \rightarrow loves(X, money)) \\ rich(giles) \end{array} \right\} \models happy(giles)$$

Redundancy

The current setup has two kinds of redundancy

Firstly, we are performing more inferences than we need to

Secondly, some clauses may not be needed (or useful)

Redundancy

The current setup has two kinds of redundancy

Firstly, we are performing more inferences than we <u>need</u> to

We will introduce an ordered version of resolution that removes some unnecessary inferences

Secondly, some clauses may not be needed (or useful)

Redundancy

The current setup has two kinds of redundancy

Firstly, we are performing more inferences than we <u>need</u> to

We will introduce an ordered version of resolution that removes some unnecessary inferences

Secondly, some clauses may not be needed (or useful)

We will define a general notion of redundant clause and a specific process that allows us to remove redundant clauses

Demonstrating the need for Ordered Resolution

Consider this knowledge base

1	$rich(giles)$
2	$famous(giles)$
3	$\neg happy(giles)$
4	$\neg rich(X) \vee \neg famous(X) \vee happy(X)$

If we select clauses 1-3 first we don't do any inferences but when we select 4 we can do 3 inferences to produce

5	$\neg famous(giles) \vee happy(giles)$
6	$\neg rich(giles) \vee \vee happy(giles)$
7	$\neg rich(giles) \vee \neg famous(giles)$

Selecting 5-6 in turn and performing inferences gives us 6 new unit clauses. Selecting the first of these allows us to find a contradiction.

Selecting in a different order would have found contradiction earlier. But we also didn't need all of 5-7 to find a proof.

Introducing Ordered Resolution with Selection

We introduce a refinement of resolution called ordered resolution (with selection) which is defined as

$$\frac{l_1 \vee C \qquad \neg l_1 \vee D}{C \vee D}$$

where l_1 is selected in $l_1 \vee C$ e.g. we only perform an inference using one literal in the given clause.

However, clearly in general this is incomplete e.g. we may select literals such that we stop ourselves from finding a proof. For example, consider

$$p \vee \underline{q} \qquad \underline{p} \vee \neg q \qquad \underline{\neg p} \vee q \qquad \neg p \vee \underline{\neg q}$$

where I have underlined selected literals. The clauses are inconsistent but performing inferences on selected literals produces tautologies.

Literal Selection

How do we pick a good literal selection function?

Firstly, does incompleteness matter?

Secondly, how do we guarantee completeness?

Note: see my paper “Selecting the Selection” in IJCAR 2016 for a full coverage of this topic.

Literal Selection

How do we pick a good literal selection function?

Firstly, does incompleteness matter?

Yes and no. Yes, it's important to know whether we're complete as we if we saturate we should know what this means. But no, if we can find a proof whilst being incomplete then it's still a proof - maybe being incomplete is faster!

Secondly, how do we guarantee completeness?

Note: see my paper "Selecting the Selection" in IJCAR 2016 for a full coverage of this topic.

Literal Selection

How do we pick a good literal selection function?

Firstly, does incompleteness matter?

Yes and no. Yes, it's important to know whether we're complete as we if we saturate we should know what this means. But no, if we can find a proof whilst being incomplete then it's still a proof - maybe being incomplete is faster!

Secondly, how do we guarantee completeness?

The idea is to have a selection strategy such that if we saturate we're guaranteed to have done all inferences that *matter*. We will do this by introducing a notion of ordering on literals and I will give an overview of how this fixes our problem afterwards.

Note: see my paper "Selecting the Selection" in IJCAR 2016 for a full coverage of this topic.

Orderings

A partial ordering (irreflexive, transitive) $\succ$ is well-founded if there exist no infinite chains $a_0 \succ a_1 \succ a_2 \succ \ldots$

An ordering $\succ$ is monotonic if whenever $l \succ r$ then $s[l] \succ s[r]$

An ordering $\succ$ is stable under substitutions if whenever $l \succ r$ then $l\theta \succ r\theta$

An ordering with all of the above is called a reduction ordering

An ordering $\succ$ has the subterm property if r is a subterm of l then $l \succ r$

An ordering with all of the above is called a simplification ordering.

This properties matter for the proof of completeness later. As this isn't a theory-heavy course I'm not going to go into them any more or examine you on them.

Ground Term Ordering

Let the number of (function or variable) symbols in a term be its weight given by a function w e.g. $w(f(a, g(x))) = 4$.

Let $\succ_S$ be an ordering on function symbols. We define a total simplification ordering on ground terms such that $s \succ_G t$ if

1. $w(s) > w(t)$, or
2. $w(s) = w(t)$ and $s = f(s_1, \ldots, s_n)$ and $t = g(t_1, \ldots, t_m)$ and either
   - $f \succ_S g$, or
   - $s_i \succ_G t_i$ for some i and for all $j < i$, $s_j = t_j$

Why is this total? Why is it well-founded?

Ground Term Ordering

Let the number of (function or variable) symbols in a term be its weight given by a function w e.g. $w(f(a,g(x))) = 4$.

Let $\succ_S$ be an ordering on function symbols. We define a total simplification ordering on ground terms such that $s \succ_G t$ if

1. $w(s) > w(t)$, or
2. $w(s) = w(t)$ and $s = f(s_1, \ldots, s_n)$ and $t = g(t_1, \ldots, t_m)$ and either
   - $f \succ_S g$, or
   - $s_i \succ_G t_i$ for some i and for all $j < i$, $s_j = t_j$

Why is this total? Why is it well-founded?

Examples (given $f \succ_S g \succ_S h \succ_G a \succ_G b$):

$$f(a) \succ_G a \qquad f(a) \succ_G h(a) \qquad g(a, f(a)) \succ_G g(a, f(b))$$

Non-Ground Term Ordering

We lift $\succ_G$ to non-ground terms s and t such that $s \succ_N t$ if

1. For each variable x, the number of x in s is $\geq$ that in t, and
2. Either $s \succ_G t$ or $t = x$ and $s = f^n(x)$ for $x > 0$

Why do we need (1)?

Non-Ground Term Ordering

We lift $\succ_G$ to non-ground terms s and t such that $s \succ_N t$ if

1. For each variable x, the number of x in s is $\geq$ that in t, and
2. Either $s \succ_G t$ or $t = x$ and $s = f^n(x)$ for $x > 0$

Why do we need (1)? Consider $f(g(a), x)$ and $f(x, x)$.

Non-Ground Term Ordering

We lift $\succ_G$ to non-ground terms s and t such that $s \succ_N t$ if

1. For each variable x, the number of x in s is $\geq$ that in t, and
2. Either $s \succ_G t$ or $t = x$ and $s = f^n(x)$ for $x > 0$

Why do we need (1)? Consider $f(g(a), x)$ and $f(x, x)$.
Why is this partial?

Non-Ground Term Ordering

We lift $\succ_G$ to non-ground terms s and t such that $s \succ_N t$ if

1. For each variable x, the number of x in s is $\geq$ that in t, and
2. Either $s \succ_G t$ or $t = x$ and $s = f^n(x)$ for $x > 0$

Why do we need (1)? Consider $f(g(a), x)$ and $f(x, x)$.
Why is this partial? Consider $f(x)$ and $f(b)$ where $a \succ_S b \succ_S c$

Non-Ground Term Ordering

We lift $\succ_G$ to non-ground terms s and t such that $s \succ_N t$ if

1. For each variable x, the number of x in s is $\geq$ that in t, and
2. Either $s \succ_G t$ or $t = x$ and $s = f^n(x)$ for $x > 0$

Why do we need (1)? Consider $f(g(a), x)$ and $f(x, x)$.
Why is this partial? Consider $f(x)$ and $f(b)$ where $a \succ_S b \succ_S c$

Examples (given $f \succ_S g \succ_S a$):

$$f(x) \succ_N g(x) \qquad f(x) \succ_N x \qquad g(x, f(y)) \succ_N g(x, a)$$

What about $g(x, y)$ and $g(y, x)$?

This ordering is $\succ_{KBO}$ the Knuth-Bendix Ordering (KBO) used in Vampire

Ordered Resolution

$\succ_{KBO}$ lifts to predicates directly (extend $\succ_S$ to predicate symbols).

Lift $\succ_{KBO}$ to literals: $\neg l \succ l$ for every atom l, treat $=$ as biggest predicate.

A literal is maximal in a clause if there are no other literals greater than it. Due to partiality of the ordering they can be multiple maximal literals.

A selection function is well-behaved if it either selects (i) at least one negative literal, or (ii) all maximal literals. This ensures completeness.

Reminder: Ordered Resolution (with selection) is then

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

where l_1 and $\neg l_2$ are selected.

Revisiting our Motivating Example

Consider this knowledge base

$$\begin{array}{ll} 1 & rich(giles) \\ 2 & famous(giles) \\ 3 & \neg happy(giles) \\ 4 & \neg rich(X) \vee \neg famous(X) \vee happy(X) \end{array}$$

If we select clauses 1-3 first we don't do any inferences but when we select 4 we should select one literal and produce one conclusion e.g.

$$5 \qquad \neg rich(giles) \vee happy(giles)$$

then we select one literal again and produce one conclusion

$$6 \qquad \neg rich(giles)$$

and on the 7th step we find a contradiction without producing anything not needed for the proof.

Missing Rule: Factoring

Usually we also have the (positive) factoring rule

$$\frac{C \vee l_1 \vee l_2}{(C \vee l_1)\theta} \ \theta = \text{mgu}(l_1, l_2)$$

Missing Rule: Factoring

Usually we also have the (positive) factoring rule

$$\frac{C \vee l_1 \vee l_2}{(C \vee l_1)\theta} \; \theta = \mathrm{mgu}(l_1, l_2)$$

which is required in some cases

$$\begin{array}{ll} 1 & p(u) \vee p(f(u)) \\ 2 & \neg p(v) \vee p(f(w)) \\ 3 & \neg p(x) \vee \neg p(f(x)) \end{array}$$

Missing Rule: Factoring

Usually we also have the (positive) factoring rule

$$\frac{C \vee l_1 \vee l_2}{(C \vee l_1)\theta} \; \theta = \mathrm{mgu}(l_1, l_2)$$

which is required in some cases

$$\begin{array}{ll} 1 & p(u) \vee p(f(u)) \\ 2 & \neg p(v) \vee p(f(w)) \\ 3 & \neg p(x) \vee \neg p(f(x)) \end{array}$$

Resolvents

Missing Rule: Factoring

Usually we also have the (positive) factoring rule

$$\frac{C \vee l_1 \vee l_2}{(C \vee l_1)\theta} \; \theta = \mathrm{mgu}(l_1, l_2)$$

which is required in some cases

$$\begin{array}{lr} 1 & p(u) \vee p(f(u)) \\ 2 & \neg p(v) \vee p(f(w)) \\ 3 & \neg p(x) \vee \neg p(f(x)) \end{array}$$

Resolvents

$$\begin{array}{lll} 4 & p(u) \vee p(f(w)) & (1,2) \end{array}$$

Missing Rule: Factoring

Usually we also have the (positive) factoring rule

$$\frac{C \vee l_1 \vee l_2}{(C \vee l_1)\theta} \ \theta = \text{mgu}(l_1, l_2)$$

which is required in some cases

$$\begin{array}{ll} 1 & p(u) \vee p(f(u)) \\ 2 & \neg p(v) \vee p(f(w)) \\ 3 & \neg p(x) \vee \neg p(f(x)) \end{array}$$

Resolvents

$$\begin{array}{lll} 4 & p(u) \vee p(f(w)) & (1,2) \\ 5 & p(u) \vee \neg p(f(f(u))) & (1,3) \end{array}$$

Missing Rule: Factoring

Usually we also have the (positive) factoring rule

$$\frac{C \vee l_1 \vee l_2}{(C \vee l_1)\theta} \; \theta = \text{mgu}(l_1, l_2)$$

which is required in some cases

$$\begin{array}{ll} 1 & p(u) \vee p(f(u)) \\ 2 & \neg p(v) \vee p(f(w)) \\ 3 & \neg p(x) \vee \neg p(f(x)) \end{array}$$

Resolvents

$$\begin{array}{lll} 4 & p(u) \vee p(f(w)) & (1,2) \\ 5 & p(u) \vee \neg p(f(f(u))) & (1,3) \end{array}$$

We're only going to get clauses with 2 literals.
However, we can factor (4) to $p(f(w))$
Resolving with (3) gives $\neg p(f(f(z)))$ then with $p(f(w))$ gives *false*

Soundness and Completeness

Recall

- A system is sound if it answers questions correctly
- A system is complete if it answers all question
- A system is refutationaly complete if it answers all questions where the answer is no

If we consider our system as The Given Clause Algorithm with Resolution and Factoring then

Claim 1: Our System is Sound for consistency checking

Claim 2: Our System is Refutationaly Complete for consistency checking

Soundness

This means that we only derive *false* if the input set is inconsistent.

We can simply show that all conclusions of inference rules are true in the same models as their premises and we get soundness.

Factoring

$$\frac{C \vee l_1 \vee l_2}{(C \vee l_1)\theta} \; \theta = \text{mgu}(l_1, l_2)$$

If $\mathcal{M} \models C$ then $\mathcal{M} \models C\theta$. If $\mathcal{M} \models l_1 \vee l_2$ and $\theta = \text{mgu}(l_2, l_2)$ then $\mathcal{M} \models l_1\theta$

Resolution

$$\frac{\underline{l_1} \vee C \qquad \underline{\neg l_2} \vee D}{(C \vee D)\theta} \; \theta = \text{mgu}(l_1, l_2)$$

If $\mathcal{M} \models l_1$ and $\mathcal{M} \models \neg l_1 \vee D$ then $\mathcal{M} \models D$. If $\mathcal{M} \models \neg l_1$ and $\mathcal{M} \models l_1 \vee C$ then $\mathcal{M} \models C$. Always $\mathcal{M} \models l_1 \vee l_2$ therefore $\mathcal{M} \models D \vee C$.

Soundness

This means that we only derive *false* if the input set is inconsistent.

We can simply show that all conclusions of inference rules are true in the same models as their premises and we get soundness.

Factoring

$$\frac{C \vee l_1 \vee l_2}{(C \vee l_1)\theta} \ \theta = \text{mgu}(l_1, l_2)$$

If $\mathcal{M} \models C$ then $\mathcal{M} \models C\theta$. If $\mathcal{M} \models l_1 \vee l_2$ and $\theta = \text{mgu}(l_2, l_2)$ then $\mathcal{M} \models l_1\theta$

Resolution

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

If $\mathcal{M} \models l_1$ and $\mathcal{M} \models \neg l_1 \vee D$ then $\mathcal{M} \models D$. If $\mathcal{M} \models \neg l_1$ and $\mathcal{M} \models l_1 \vee C$ then $\mathcal{M} \models C$. Always $\mathcal{M} \models l_1 \vee l_2$ therefore $\mathcal{M} \models D \vee C$.

Our Ordering Stratifies Ground Clauses

Lift $\succ$ to clauses: $C \succ D$ if for every l in D/C there is a $l' \succ l$ in C/D

Example, given $p \succ q \succ r$

$$p \vee q \succ p \succ \neg q \vee r \succ q \vee r$$

$\succ$ on ground clauses is total and well-founded

Let $\max(C)$ be the maximal literal in C, this exists and is unique

If $\max(C) \succ \max(D)$ then $C \succ D$

If $\max(C) = \max(D)$ but $\max(C)$ is neg and $\max(D)$ pos then $C \succ D$

This gives a stratification of clause sets by maximal literal

Model Construction

Idea:

- Build up an interpretation $\mathcal{M}$ incrementally
- Look at clauses from smallest to largest
- If C is already true in I then carry on
- Otherwise, make the maximal literal in C true in I

We use $\mathcal{M}_C$ for the interpretation after processing C and Δ_C for the new clauses produced by C. Then

$$\begin{array}{rcl} \mathcal{M}_C & = & \bigcup_{C \succ D} \Delta_D \\ \Delta_C & = & \begin{cases} \{l\} & \textit{if } \mathcal{M}_C \not\models C \textit{ and } l = \max(C) \textit{ and } l \textit{ is pos} \\ \emptyset & \textit{otherwise} \end{cases} \end{array}$$

If $\Delta_C = \{l\}$ we say that C is produces l and C is productive

Example

Let $p_5 \succ p_4 \succ p_3 \succ p_2 \succ p_1 \succ p_0$

clauses	$\mathcal{M}_C$	Δ_C	Remark
$\neg p_0$	$\emptyset$	$\emptyset$	true in $\mathcal{M}_C$
$p_0 \vee p_1$	$\emptyset$	$\{p_1\}$	
$p_1 \vee p_2$	$\{p_1\}$	$\emptyset$	true in $\mathcal{M}_C$
$\neg p_1 \vee p_2$	$\{p_1\}$	$\{p_2\}$	
$\neg p_1 \vee p_3 \vee p_0$	$\{p_1, p_2\}$	$\{p_3\}$	
$\neg p_1 \vee p_4 \vee p_3 \vee p_0$	$\{p_1, p_2, p_3\}$	$\emptyset$	true in $\mathcal{M}_C$
$\neg p_1 \vee \neg p_4 \vee p_3$	$\{p_1, p_2, p_3\}$	$\emptyset$	true in $\mathcal{M}_C$
$\neg p_4 \vee p_5$	$\{p_1, p_2, p_3\}$	$\{p_5\}$	

So $\mathcal{M} = \{p_1, p_2, p_3, p_5\}$

Completeness

If a set of clauses is saturated we can take its grounding and construct a model that makes all clauses true.

Therefore, if we saturate the set of clauses must be consistent

Due to fairness of clause selection, we will visit all consequences eventually.

Redundancy

A clause c is redundant with respect to a set of clauses S if there is a set $S' \subset S$ such that

$$S' \models c \qquad \textit{and for all } c' \textit{ in } S' \textit{ we have } c \succ c'$$

The first part is clear, if c can be derived from S' then we do not need it.

The second part comes from our model construction argument as at any point in the construction we can only rely on smaller clauses.

The key redundancy check is subsumption. C subsumes B if $B = (C \vee D)\theta$ e.g. C is a generalisation of a subclause of B.

Saturation Up to Redundancy

Given a set of clauses S and a set of inferences $\mathcal{I}$ we say that S is saturated up to redundancy if there are no clauses c not in S such that c can be derived from S using $\mathcal{I}$ and c is not redundant with respect to S.

Example 1

$$\left\{\begin{array}{c} \forall x.(happy(x) \leftrightarrow \exists y.(loves(x,y))) \\ \forall x.(rich(x) \rightarrow loves(x, money)) \\ rich(giles) \end{array}\right\} \models happy(giles)$$

Example 2

$$\left\{ \begin{array}{c} \forall x.(require(x) \rightarrow require(depend(x))) \\ depend(a) = b \\ depend(b) = c \\ require(a) \end{array} \right\} \models require(c)$$

Summary

This week we have seen

- The theorem proving architecture
- Clausification
- Preprocessing optimisations
- The Given Clause Algorithm
- Ordered Resolution
- Soundness and Completeness arguments

Next week:

- Reasoning with FOL with Equality
- Reasoning with FOL with Arithmetic
- Other kinds of Automated Reasoning