# First-Order Theorem Proving and Vampire

## The Theory Bit

Giles Reger, Martin Suda,
Laura Kovács, and Andrei Voronkov

University of Manchester and TU Wein

# Outline


Preliminaries

Theorem-Proving Workflow

A Static View: Inferences, Soundness, and Completeness

A Dynamic View: Saturation

Making It Fast in Practice

Let's Try Proving Something – a Mini-Challenge

# Outline

Preliminaries

Theorem-Proving Workflow

A Static View: Inferences, Soundness, and Completeness

A Dynamic View: Saturation

Making It Fast in Practice

Let's Try Proving Something – a Mini-Challenge

# Arbitrary First-Order Formulas

- A first-order signature (vocabulary): function symbols (including constants), predicate symbols. Equality is part of the language.
- A set of variables.
- Terms are built using variables and function symbols. For example, $f(x) + g(x)$.
- Atoms, or atomic formulas are obtained by applying a predicate symbol to a sequence of terms. For example, $p(a, x)$ or $f(x) + g(x) \geq 2$.
- Formulas: built from atoms using logical connectives $\neg$, $\wedge$, $\vee$, $\rightarrow$, $\leftrightarrow$ and quantifiers $\forall$, $\exists$. For example, $(\forall x)x = 0 \vee (\exists y)y > x$.

# Clauses

- Literal: either an atom $A$ or its negation $\neg A$.
- Clause: a disjunction $L_1 \lor \ldots \lor L_n$ of literals, where $n \geq 0$.

# Clauses

- Literal: either an atom $A$ or its negation $\neg A$.
- Clause: a disjunction $L_1 \vee \ldots \vee L_n$ of literals, where $n \geq 0$.
- Empty clause, denoted by $\square$: clause with 0 literals, that is, when $n = 0$.

# Clauses

- Literal: either an atom $A$ or its negation $\neg A$.
- Clause: a disjunction $L_1 \vee \ldots \vee L_n$ of literals, where $n \geq 0$.
- Empty clause, denoted by $\square$: clause with 0 literals, that is, when $n = 0$.
- A formula in Clausal Normal Form (CNF): a conjunction of clauses.

# Clauses

- Literal: either an atom $A$ or its negation $\neg A$.
- Clause: a disjunction $L_1 \vee \ldots \vee L_n$ of literals, where $n \geq 0$.
- Empty clause, denoted by $\square$: clause with 0 literals, that is, when $n = 0$.
- A formula in Clausal Normal Form (CNF): a conjunction of clauses.
- A clause is ground if it contains no variables.
- If a clause contains variables, we assume that it implicitly universally quantified. That is, we treat $p(x) \vee q(x)$ as $\forall x(p(x) \vee q(x))$.

# Outline

Preliminaries

Theorem-Proving Workflow

A Static View: Inferences, Soundness, and Completeness

A Dynamic View: Saturation

Making It Fast in Practice

Let's Try Proving Something – a Mini-Challenge

# What an Automatic Theorem Prover is Expected to Do

Input:

- a set of axioms (first order formulas) or clauses;
- a conjecture (first-order formula or set of clauses).

Output:

- proof (hopefully).

# Proof by Refutation

Given a problem with axioms and assumptions $F_1, \ldots, F_n$ and conjecture $G$,

1. negate the conjecture;
2. establish unsatisfiability of the set of formulas $F_1, \ldots, F_n, \neg G$.

# Proof by Refutation

Given a problem with axioms and assumptions $F_1, \ldots, F_n$ and conjecture $G$,

1. negate the conjecture;
2. establish unsatisfiability of the set of formulas $F_1, \ldots, F_n, \neg G$.

Thus, we reduce the theorem proving problem to the problem of checking unsatisfiability.

# Proof by Refutation

Given a problem with axioms and assumptions $F_1, \ldots, F_n$ and conjecture $G$,

1. negate the conjecture;
2. establish unsatisfiability of the set of formulas $F_1, \ldots, F_n, \neg G$.

Thus, we reduce the theorem proving problem to the problem of checking unsatisfiability.

In this formulation the negation of the conjecture $\neg G$ is treated like any other formula. In fact, Vampire (and other provers) internally treat conjectures differently, to make proof search more goal-oriented.

# General Scheme in One Slide

- Read a problem $P$
- Preprocess the problem: $P \Longrightarrow P'$
- Convert $P'$ into Clause Normal Form $N$
  - replacing connectives, formula naming, distributive laws
  - Skolemisation
- Run a saturation algorithm on it, try to derive $\square$.
  - computes a closure of $N$ with respect to an inference system
  - logical calculus: resolution + superposition
- If $\square$ is derived, report the result, maybe including a refutation.

# General Scheme in One Slide

- Read a problem $P$
- Preprocess the problem: $P \Longrightarrow P'$
- Convert $P'$ into Clause Normal Form $N$
  - replacing connectives, formula naming, distributive laws
  - Skolemisation
- Run a saturation algorithm on it, try to derive $\square$.
  - computes a closure of $N$ with respect to an inference system
  - logical calculus: resolution + superposition
- If $\square$ is derived, report the result, maybe including a refutation.

Trying to derive $\square$ using a saturation algorithm is the hardest part, which in practice may not terminate or run out of memory.

# A Bit More on the CNF Transformation

- replacing unwanted connectives:

$$
\begin{array}{lcl}
A \leftrightarrow B & \Longrightarrow & (A \rightarrow B) \wedge (B \rightarrow A) \\
A \rightarrow B & \Longrightarrow & \neg A \vee B \\
\neg(A \vee B) & \Longrightarrow & \neg A \wedge \neg B \\
 & \dots &
\end{array}
$$

- distributive laws:

$$(A \wedge B) \vee (C \wedge D) \Longrightarrow (A \vee C) \wedge (A \vee D) \wedge (B \vee C) \wedge (B \vee D)$$

- formula naming (recall Tseitin / Pleisted-Greenbaum):

$$(A \wedge B) \vee (C \wedge D) \Longrightarrow (F_{AB} \vee (C \wedge D)) \wedge (F_{AB} \rightarrow A) \wedge (F_{AB} \rightarrow B)$$

- Skolemisation on an example

$$\forall x[x \neq 0 \rightarrow \exists y(x \cdot y = 1)] \quad \Longrightarrow \quad x \neq 0 \rightarrow x \cdot sk_y(x) = 1$$

# A Bit More on the CNF Transformation

- replacing unwanted connectives:

$$\begin{array}{lcl} A \leftrightarrow B & \Longrightarrow & (A \rightarrow B) \wedge (B \rightarrow A) \\ A \rightarrow B & \Longrightarrow & \neg A \vee B \\ \neg(A \vee B) & \Longrightarrow & \neg A \wedge \neg B \\ & \ldots & \end{array}$$

- distributive laws:

$$(A \wedge B) \vee (C \wedge D) \Longrightarrow (A \vee C) \wedge (A \vee D) \wedge (B \vee C) \wedge (B \vee D)$$

- formula naming (recall Tseitin / Pleisted-Greenbaum):

$$(A \wedge B) \vee (C \wedge D) \Longrightarrow (F_{AB} \vee (C \wedge D)) \wedge (F_{AB} \rightarrow A) \wedge (F_{AB} \rightarrow B)$$

- Skolemisation on an example

$$\forall x[x \neq 0 \rightarrow \exists y(x \cdot y = 1)] \quad \Longrightarrow \quad x \neq 0 \rightarrow x \cdot sk_y(x) = 1$$

# A Bit More on the CNF Transformation

- replacing unwanted connectives:

$$\begin{array}{lcl} A \leftrightarrow B & \Longrightarrow & (A \rightarrow B) \wedge (B \rightarrow A) \\ A \rightarrow B & \Longrightarrow & \neg A \vee B \\ \neg(A \vee B) & \Longrightarrow & \neg A \wedge \neg B \\ & \ldots & \end{array}$$

- distributive laws:

$$(A \wedge B) \vee (C \wedge D) \Longrightarrow (A \vee C) \wedge (A \vee D) \wedge (B \vee C) \wedge (B \vee D)$$

- formula naming (recall Tseitin / Pleisted-Greenbaum):

$$(A \wedge B) \vee (C \wedge D) \Longrightarrow (F_{AB} \vee (C \wedge D)) \wedge (F_{AB} \rightarrow A) \wedge (F_{AB} \rightarrow B)$$

- Skolemisation on an example

$$\forall x[x \neq 0 \rightarrow \exists y(x \cdot y = 1)] \quad \Longrightarrow \quad x \neq 0 \rightarrow x \cdot sk_y(x) = 1$$

# A Bit More on the CNF Transformation

- replacing unwanted connectives:

$$\begin{array}{lcl} A \leftrightarrow B & \Longrightarrow & (A \rightarrow B) \wedge (B \rightarrow A) \\ A \rightarrow B & \Longrightarrow & \neg A \vee B \\ \neg(A \vee B) & \Longrightarrow & \neg A \wedge \neg B \\ & \ldots & \end{array}$$

- distributive laws:

$$(A \wedge B) \vee (C \wedge D) \Longrightarrow (A \vee C) \wedge (A \vee D) \wedge (B \vee C) \wedge (B \vee D)$$

- formula naming (recall Tseitin / Pleisted-Greenbaum):

$$(A \wedge B) \vee (C \wedge D) \Longrightarrow (F_{AB} \vee (C \wedge D)) \wedge (F_{AB} \rightarrow A) \wedge (F_{AB} \rightarrow B)$$

- Skolemisation on an example

$$\forall x[x \neq 0 \rightarrow \exists y(x \cdot y = 1)] \quad \Longrightarrow \quad x \neq 0 \rightarrow x \cdot sk_y(x) = 1$$

# Lets quickly have a look

```
./vampire --mode clausify Problems/PUZ031+1.p
```

```
./vampire --mode clausify
fof(pel47_14,axiom,
    ( ! [X] :
        ( ( caterpillar(X)
          | snail(X) )
       => ? [Y] :
            ( plant(Y)
            & eats(X,Y) ) ) )).
```

# Lets quickly have a look

```
./vampire --mode clausify Problems/PUZ031+1.p
```

```
./vampire --mode clausify
fof(pel47_14,axiom,
    ( ! [X] :
        ( ( caterpillar(X)
          | snail(X) )
       => ? [Y] :
            ( plant(Y)
            & eats(X,Y) ) ) )).
```

# Outline


Preliminaries

Theorem-Proving Workflow

A Static View: Inferences, Soundness, and Completeness

A Dynamic View: Saturation

Making It Fast in Practice

Let's Try Proving Something – a Mini-Challenge

# Inference System

‘

- An inference has the form

$$\frac{F_1 \quad \dots \quad F_n}{G},$$

where $n \geq 0$ and $F_1, \dots, F_n, G$ are formulas.

- The formula $G$ is called the conclusion of the inference;
- The formulas $F_1, \dots, F_n$ are called its premises.
- An inference rule $R$ is a set of inferences.
- Every inference $I \in R$ is called an instance of $R$.
- An Inference system $\mathbb{I}$ is a set of inference rules.

# Derivation, Proof

- Derivation in an inference system $\mathbb{I}$:
  a DAG built from inferences in $\mathbb{I}$.
- Derivation of $E$ from $E_1, \ldots, E_m$: a finite derivation of $E$ whose every leaf is one of the expressions $E_1, \ldots, E_m$ and the root of which is is $E$.
- A refutation is a derivation of the empty clause $\square$.

# Binary Resolution Inference System

The binary resolution inference system, denoted by $\mathbb{BR}$ is an inference system on propositional clauses (or ground clauses). It consists of two inference rules:

- Binary resolution, denoted by BR:

$$\frac{p \vee C_1 \qquad \neg p \vee C_2}{C_1 \vee C_2} \text{ (BR).}$$

- Factoring, denoted by Fact:

$$\frac{L \vee L \vee C}{L \vee C} \text{ (Fact).}$$

# Soundness

- An inference is sound if the conclusion of this inference is a logical consequence of its premises.
- An inference system is sound if every inference rule in this system is sound.

# Soundness

- An inference is sound if the conclusion of this inference is a logical consequence of its premises.
- An inference system is sound if every inference rule in this system is sound.

$\mathbb{BR}$ is sound.

Consequence of soundness: let $S$ be a set of clauses. If $\square$ can be derived from $S$ in $\mathbb{BR}$, then $S$ is unsatisfiable.

# Example

Consider the following set of clauses

$$\{\neg p \vee \neg q,\ \neg p \vee q,\ p \vee \neg q,\ p \vee q\}.$$

The following derivation derives the empty clause from this set:

$$\dfrac{\dfrac{\dfrac{p \vee q \quad p \vee \neg q}{p \vee p}\,(\mathrm{BR})}{p}\,(\mathrm{Fact}) \qquad \dfrac{\dfrac{\neg p \vee q \quad \neg p \vee \neg q}{\neg p \vee \neg p}\,(\mathrm{BR})}{\neg p}\,(\mathrm{Fact})}{\square}\,(\mathrm{BR})$$

Hence, this set of clauses is unsatisfiable.

# Can this be used for checking (un)satisfiability?

1. What if the empty clause cannot be derived from $S$?
2. How can one systematically search for possible derivations of the empty clause?

# Can this be used for checking (un)satisfiability?

1. What if the empty clause cannot be derived from $S$?
2. How can one systematically search for possible derivations of the empty clause?

Completeness.

*Let $S$ be an unsatisfiable set of clauses. Then there exists a derivation of $\Box$ from $S$ in $\mathbb{BR}$.*

# Can this be used for checking (un)satisfiability?

1. What if the empty clause cannot be derived from $S$?
2. How can one systematically search for possible derivations of the empty clause?

Completeness.

*Let $S$ be an unsatisfiable set of clauses. Then there exists a derivation of $\square$ from $S$ in $\mathbb{BR}$.*

In other words, $\mathbb{BR}$ is complete.

# Outline

Preliminaries

Theorem-Proving Workflow

A Static View: Inferences, Soundness, and Completeness

A Dynamic View: Saturation

Making It Fast in Practice

Let's Try Proving Something – a Mini-Challenge

# Idea of Saturation

Completess is formulated in terms of derivability of the empty clause $\square$ from a set $S_0$ of clauses in an inference system $\mathbb{I}$. However, this formulations gives no hint on how to search for such a derivation.

# Idea of Saturation

Completess is formulated in terms of derivability of the empty clause $\square$ from a set $S_0$ of clauses in an inference system $\mathbb{I}$. However, this formulations gives no hint on how to search for such a derivation.

Idea:

- Take a set of clauses $S$ (the search space), initially $S = S_0$. Repeatedly apply inferences in $\mathbb{I}$ to clauses in $S$ and add their conclusions to $S$, unless these conclusions are already in $S$.
- If, at any stage, we obtain $\square$, we terminate and report unsatisfiability of $S_0$.

# Saturation Algorithm

A saturation algorithm tries to saturate a set of clauses with respect to a given inference system.
In theory there are three possible scenarios:

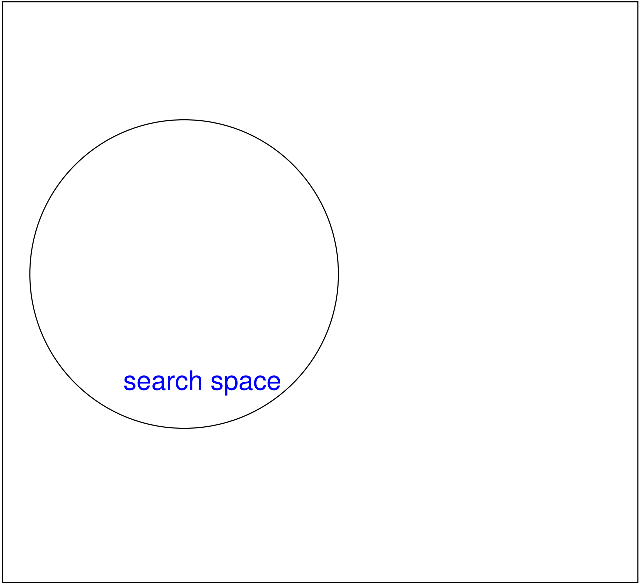
1. At some moment the empty clause $\square$ is generated, in this case the input set of clauses is unsatisfiable.
2. Saturation will terminate without ever generating $\square$, in this case the input set of clauses in satisfiable.
3. Saturation will run forever, but without generating $\square$. In this case the input set of clauses is satisfiable.

# Saturation Algorithm in Practice

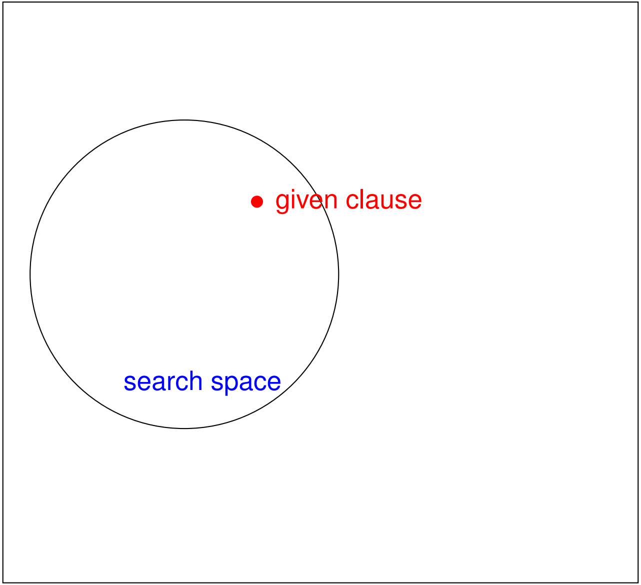
In practice there are three possible scenarios:

1. At some moment the empty clause $\square$ is generated, in this case the input set of clauses is unsatisfiable.
2. Saturation will terminate without ever generating $\square$, in this case the input set of clauses in satisfiable.
3. Saturation will run until we run out of resources, but without generating $\square$. In this case it is unknown whether the input set is unsatisfiable.

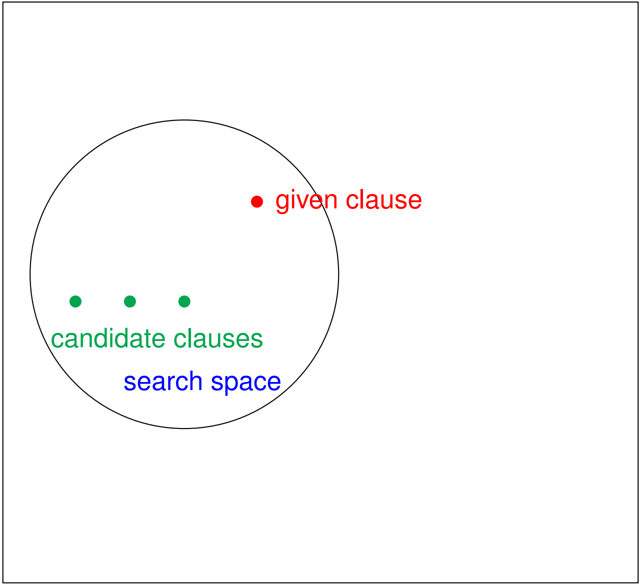
# Inference Selection by Clause Selection



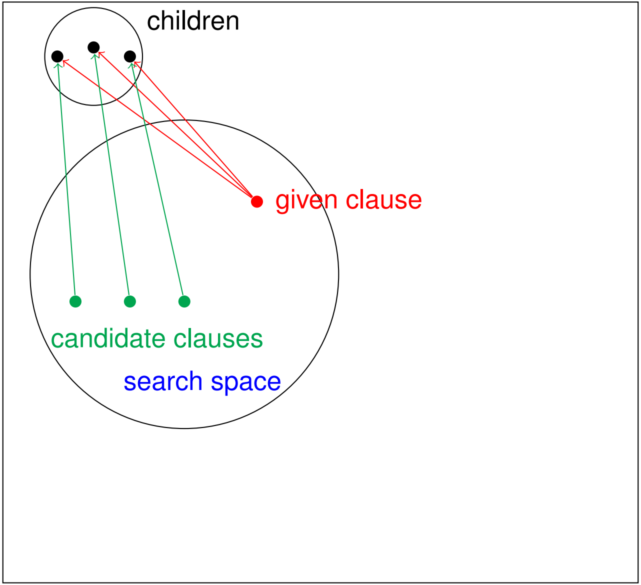
# Inference Selection by Clause Selection



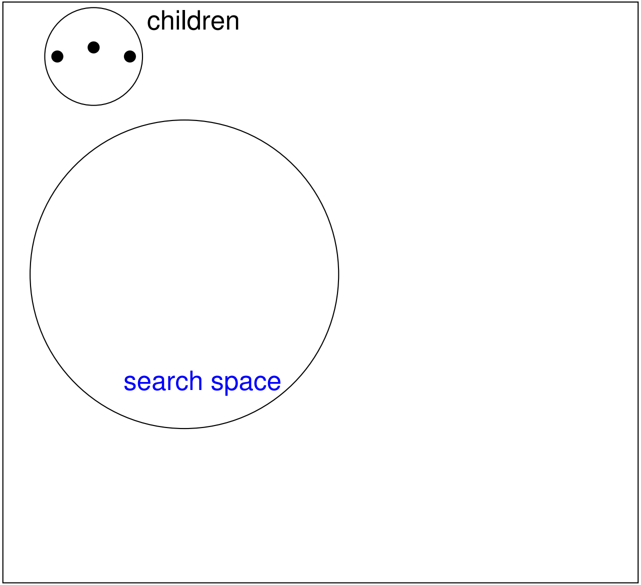
# Inference Selection by Clause Selection



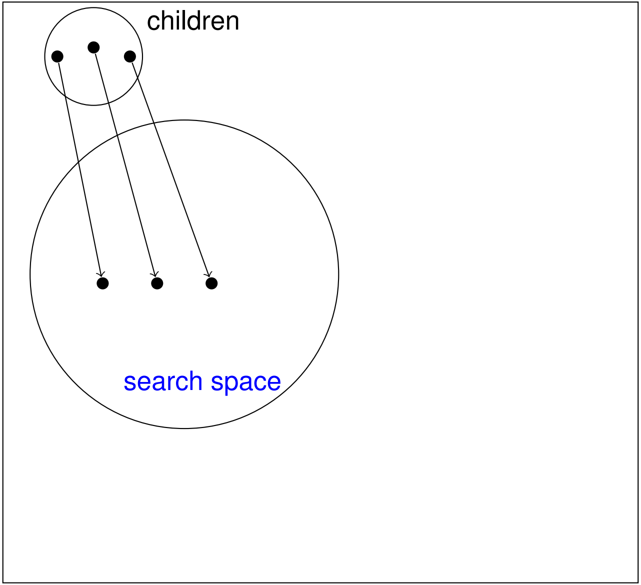
# Inference Selection by Clause Selection



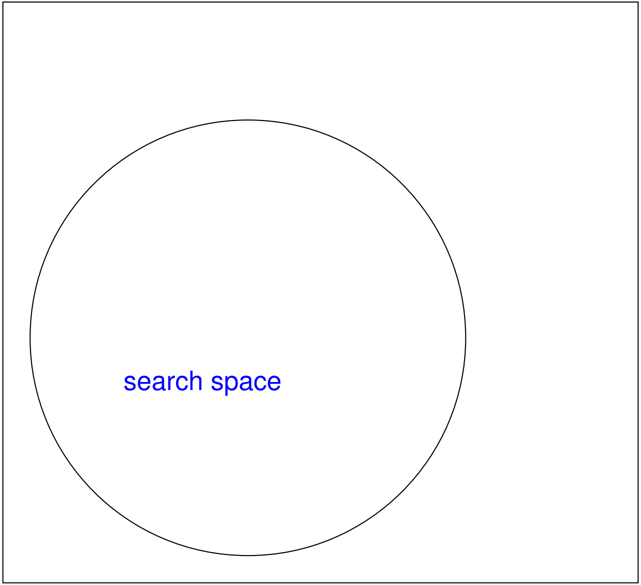
# Inference Selection by Clause Selection



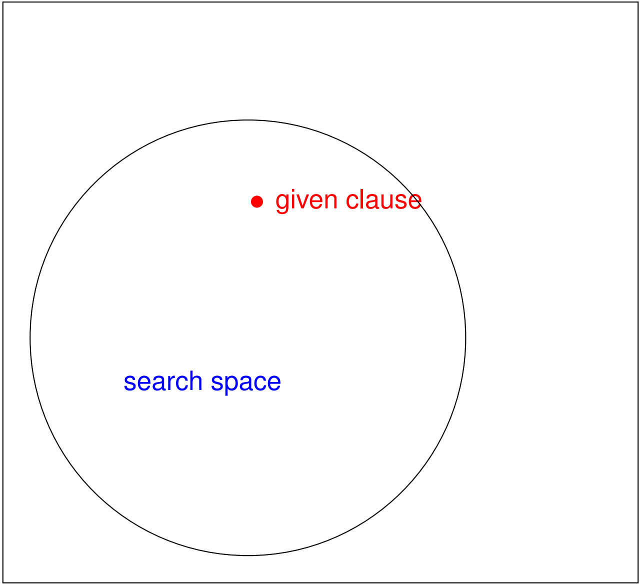
# Inference Selection by Clause Selection



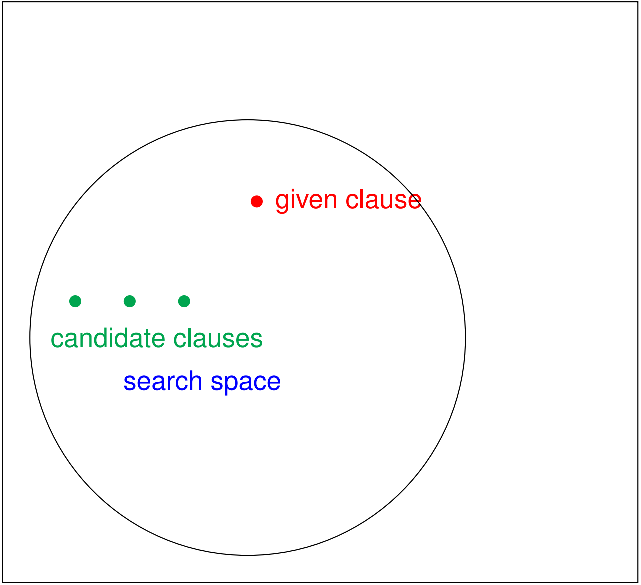
# Inference Selection by Clause Selection



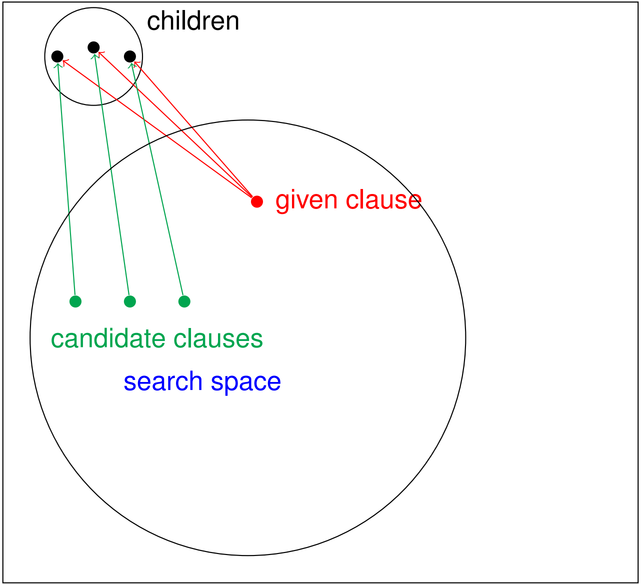
# Inference Selection by Clause Selection



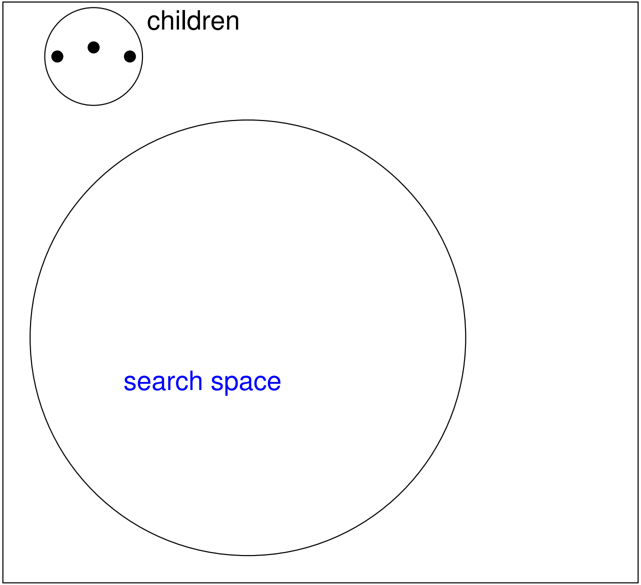
# Inference Selection by Clause Selection



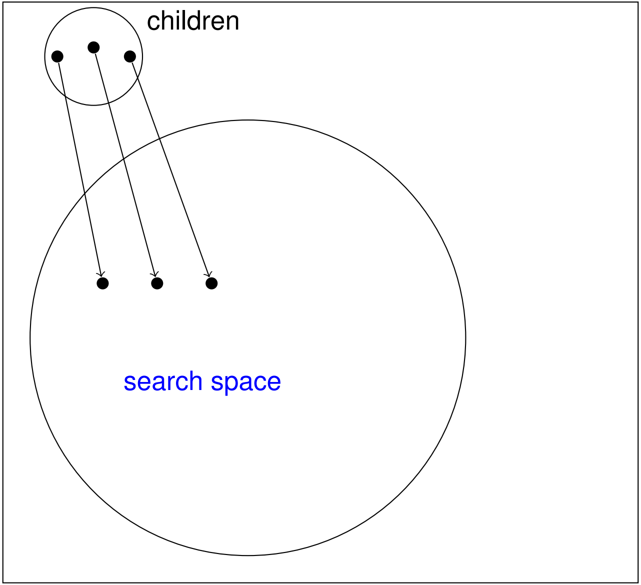
# Inference Selection by Clause Selection



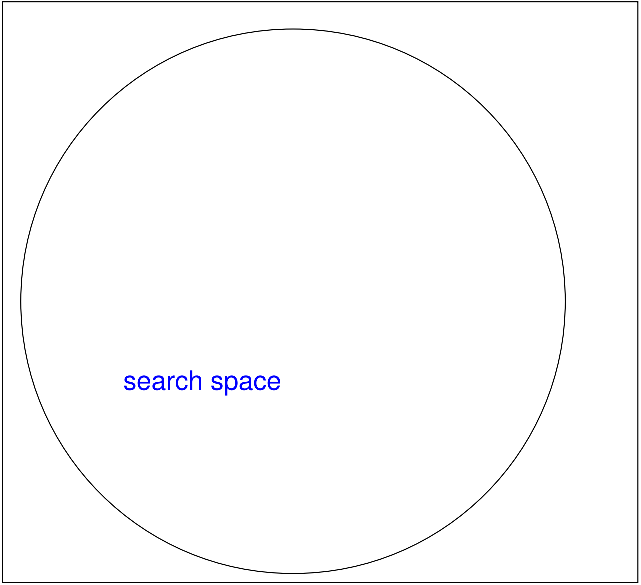
# Inference Selection by Clause Selection



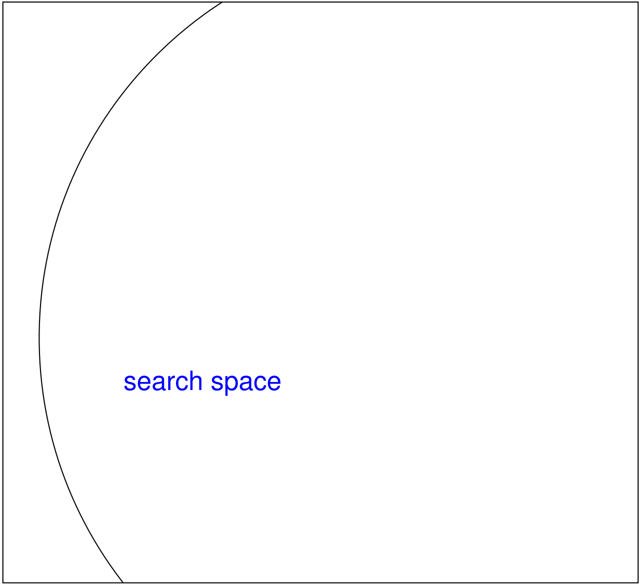
# Inference Selection by Clause Selection



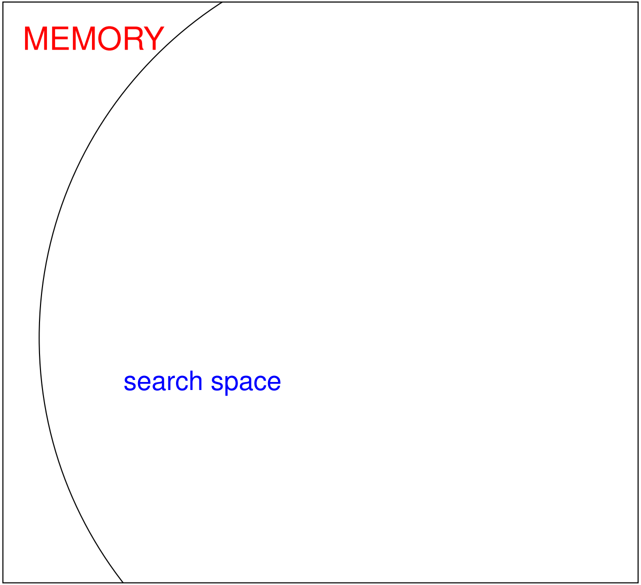
# Inference Selection by Clause Selection



# Inference Selection by Clause Selection



# Inference Selection by Clause Selection



# Saturation with the Given-Clause Algorithm

Even when we implement inference selection by clause selection, there are too many inferences, especially when the search space grows.

# Saturation with the Given-Clause Algorithm

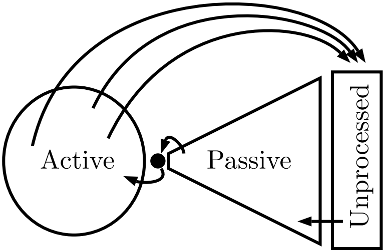
Even when we implement inference selection by clause selection, there are too many inferences, especially when the search space grows.

Solution: only apply inferences to the selected clause and the previously selected clauses.

# Saturation with the Given-Clause Algorithm

Even when we implement inference selection by clause selection, there are too many inferences, especially when the search space grows.

Solution: only apply inferences to the selected clause and the previously selected clauses.




Thus, the search space is divided in two parts:

- active clauses, that participate in inferences;
- passive clauses, that do not participate in inferences.

Observation: the set of passive clauses is usually considerably larger than the set of active clauses, often by 2-4 orders of magnitude (depending on the saturation algorithm and the problem).

# Outline


Preliminaries

Theorem-Proving Workflow

A Static View: Inferences, Soundness, and Completeness

A Dynamic View: Saturation

Making It Fast in Practice

Let's Try Proving Something – a Mini-Challenge

# Making It Fast in Practice

- Literal selection and ordering constraints
- Redundancy elimination and simplifications
- Saturation loop variants
- Clause selection heuristics
- The AVATAR architecture
- Portfolio mode
- Efficient data structures: term sharing, indexing, ...
- ...

# Selection Function

A literal selection function selects literals in a clause.

- If $C$ is non-empty, then at least one literal is selected in $C$.

# Selection Function

A literal selection function selects literals in a clause.

- If $C$ is non-empty, then at least one literal is selected in $C$.

We denote selected literals by underlining them, e.g.,

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

# Selection Function

A literal selection function selects literals in a clause.

- If $C$ is non-empty, then at least one literal is selected in $C$.

We denote selected literals by <u>underlining them</u>, e.g.,

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

Note: selection function does not have to be a function. It can be any oracle that selects literals.

# Binary Resolution with Selection

We introduce a family of inference systems, parametrised by a literal selection function $\sigma$.
The binary resolution inference system, denoted by $\mathbb{BR}_\sigma$, consists of two inference rules:

- Binary resolution, denoted by BR

$$\frac{\underline{p} \vee C_1 \quad \underline{\neg p} \vee C_2}{C_1 \vee C_2} \text{ (BR).}$$

- Positive factoring, denoted by Fact:

$$\frac{\underline{p} \vee \underline{p} \vee C}{p \vee C} \text{ (Fact).}$$

# Binary Resolution with Selection

We introduce a family of inference systems, parametrised by a literal selection function $\sigma$.
The binary resolution inference system, denoted by $\mathbb{BR}_\sigma$, consists of two inference rules:

- Binary resolution, denoted by BR

$$\frac{\underline{p} \vee C_1 \quad \underline{\neg p} \vee C_2}{C_1 \vee C_2} \text{ (BR).}$$

- Positive factoring, denoted by Fact:

$$\frac{\underline{p} \vee \underline{p} \vee C}{p \vee C} \text{ (Fact).}$$

Completeness considerations!

# The Main Rule for Dealing with Equality

Superposition:

$$\frac{\underline{l = r} \vee C \quad \underline{s[l'] = t} \vee D}{(s[r] = t \vee C \vee D)\theta}\ (\mathrm{Sup}), \qquad \frac{\underline{l = r} \vee C \quad \underline{s[l'] \neq t} \vee D}{(s[r] \neq t \vee C \vee D)\theta}\ (\mathrm{Sup}),$$

where

1. $\theta$ is an mgu of $l$ and $l'$;
2. $l'$ is not a variable;
3. $r\theta \not\succeq l\theta$;
4. $t\theta \not\succeq s[l']\theta$.
5. ...

# Subsumption and Tautology Deletion

A clause is a propositional tautology if it is of the form $p \vee \neg p \vee C$, that is, it contains a pair of complementary literals.

There are also equational tautologies, for example $a \neq b \vee b \neq c \vee f(c, c) \simeq f(a, a)$.

# Subsumption and Tautology Deletion

A clause is a propositional tautology if it is of the form $p \vee \neg p \vee C$, that is, it contains a pair of complementary literals.

There are also equational tautologies, for example $a \neq b \vee b \neq c \vee f(c, c) \simeq f(a, a)$.

A clause $C$ subsumes clause $D$
if there is a substitution $\sigma$ such that $C\sigma \subset D$

# Subsumption and Tautology Deletion

A clause is a propositional tautology if it is of the form $p \vee \neg p \vee C$, that is, it contains a pair of complementary literals.

There are also equational tautologies, for example $a \neq b \vee b \neq c \vee f(c,c) \simeq f(a,a)$.

A clause $C$ subsumes clause $D$
if there is a substitution $\sigma$ such that $C\sigma \subset D$

It was known since 1965 that

> *Subsumed clauses and tautologies can be removed from the search space.*

# Subsumption and Tautology Deletion

A clause is a propositional tautology if it is of the form $p \vee \neg p \vee C$, that is, it contains a pair of complementary literals.

There are also equational tautologies, for example
$a \neq b \vee b \neq c \vee f(c, c) \simeq f(a, a)$.

A clause $C$ subsumes clause $D$
if there is a substitution $\sigma$ such that $C\sigma \subset D$

It was known since 1965 that

> *Subsumed clauses and tautologies can be removed from the search space.*

State of the art:

- they fall under the general notion of redundancy
- redundant clauses can be removed without compromising completeness
- substantial part of prover's work spent on redundancy elimination

# Generating and Simplifying Inferences

An inference

$$\frac{C_1 \quad \ldots \quad C_n}{C} \ .$$

is called simplifying if at least one premise $C_i$ becomes redundant after the addition of the conclusion $C$ to the search space. We then say that $C_i$ is simplified into $C$.

A non-simplifying inference is called generating.

# Generating and Simplifying Inferences

An inference

$$\frac{C_1 \quad \dots \quad C_n}{C} \; .$$

is called simplifying if at least one premise $C_i$ becomes redundant after the addition of the conclusion $C$ to the search space. We then say that $C_i$ is simplified into $C$.

A non-simplifying inference is called generating.

Note. The property of being simplifying is undecidable. So is the property of being redundant. So in practice we employ sufficient conditions for simplifying inferences and for redundancy.

# Generating and Simplifying Inferences

An inference

$$\frac{C_1 \quad \ldots \quad C_n}{C} \ .$$

is called simplifying if at least one premise $C_i$ becomes redundant after the addition of the conclusion $C$ to the search space. We then say that $C_i$ is simplified into $C$.
A non-simplifying inference is called generating.

Note. The property of being simplifying is undecidable. So is the property of being redundant. So in practice we employ sufficient conditions for simplifying inferences and for redundancy.

Idea: try to search eagerly for simplifying inferences bypassing the strategy for inference selection.

# Generating and Simplifying Inferences

Two main implementation principles:

apply simplifying inferences eagerly;
apply generating inferences lazily.

checking for simplifying inferences should pay off;
so it must be cheap.

# Redundancy Checking

Redundancy-checking occurs upon addition of a new child $C$. It works as follows

- Retention test: check if $C$ is redundant.
- Forward simplification: check if $C$ can be simplified using a simplifying inference.
- Backward simplification: check if $C$ simplifies or makes redundant an old clause.

# Examples

Retention test:

- tautology-check;
- subsumption.

(A clause $C$ subsumes a clause $D$ if there exists a substitution $\theta$ such that $C\theta$ is a submultiset of $D$.)

Simplification:

- demodulation (forward and backward);
- subsumption resolution (forward and backward).

# Some redundancy criteria are expensive

- Tautology-checking is based on congruence closure.
- Subsumption and subsumption resolution are NP-complete.

# Observations

- There may be chains (repeated applications) of forward simplifications.
- After a chain of forward simplifications another retention test can (should) be done.

# Observations

- There may be chains (repeated applications) of forward simplifications.
- After a chain of forward simplifications another retention test can (should) be done.
- Backward simplification is often expensive.

# Observations

- There may be chains (repeated applications) of forward simplifications.
- After a chain of forward simplifications another retention test can (should) be done.
- Backward simplification is often expensive.
- In practice, the retention test may include other checks, resulting in the loss of completeness, for example, we may decide to discard too heavy clauses.

# How to Design a Good Saturation Algorithm?

A saturation algorithm must be fair: every possible generating inference must eventually be selected.

Two main implementation principles:

| | |
|---|---|
| apply simplifying inferences eagerly;<br>apply generating inferences lazily. | checking for simplifying inferences should pay off;<br>so it must be cheap. |

# Given Clause Algorithm (no Simplification)

**input**: *init*: set of clauses;
**var** *active*, *passive*, *queue*: sets of clauses;
**var** *current*: clauses ;
*active* $:= \emptyset$;
*passive* $:=$ *init*;
**while** *passive* $\neq \emptyset$ **do**
$\star$ *current* $:=$ *select*(*passive*); (* clause selection *)
move *current* from *passive* to *active*;
$\star$ *queue* $:=$ *infer*(*current*, *active*); (* generating inferences *)
**if** $\square \in$ *queue* **then return** *unsatisfiable*;
*passive* $:=$ *passive* $\cup$ *queue*
**od**;
**return** *satisfiable*

# Given Clause Algorithm (with Simplification)

In fact, there is more than one . . .

# Given Clause Algorithm (with Simplification)

In fact, there is more than one . . .

unprocessed clauses and kept (active and passive) clauses

`--saturation_algorithm {lrs,otter,discount}`

# Otter vs. Discount Saturation

Otter saturation algorithm:

- active clauses participate in generating and simplifying inferences;
- passive clauses participate in simplifying inferences.

Discount saturation algorithm:

- active clauses participate in generating and simplifying inferences;
- passive clauses do not participate in inferences.

# Otter vs. Discount Saturation, Newly Generated Clauses

Otter saturation algorithm:

- active clauses participate in generating and simplifying inferences;
- new clauses participate in simplifying inferences;
- passive clauses participate in simplifying inferences.

Discount saturation algorithm:

- active clauses participate in generating and simplifying inferences;
- new clauses participate in simplifying inferences;
- passive clauses do not participate in inferences.

# Otter vs. Discount Saturation, Newly Generated Clauses

Otter saturation algorithm:

- active clauses participate in generating inferences with the selected clause and simplifying inferences with new clauses;
- new clauses participate in simplifying inferences with all clauses;
- passive clauses participate in simplifying inferences with new clauses.

Discount saturation algorithm:

- active clauses participate in generating inferences and simplifying inferences with the selected clause and simplifying inferences with the new clauses;
- new clauses participate in simplifying inferences with the selected and active clauses;
- passive clauses do not participate in inferences.

# Otter Saturation Algorithm

```
  input: init: set of clauses;
  var active, passive, unprocessed: set of clauses;
  var given, new: clause;
  active := ∅;
  unprocessed := init;
  loop
    while unprocessed ≠ ∅
      new :=pop(unprocessed);
      if new = □ then return unsatisfiable;
*     if retained(new) then                          (* retention test *)
*       simplify new by clauses in active ∪ passive ;(* forward simplification *)
        if new = □ then return unsatisfiable;
*       if retained(new) then                        (* another retention test *)
*         delete and simplify clauses in active and (* backward simplification *)
                                     passive using new;
          move the simplified clauses to unprocessed;
          add new to passive
    if passive = ∅ then return satisfiable or unknown
*   given := select(passive);                        (* clause selection *)
    move given from passive to active;
*   unprocessed :=infer(given, active);              (* generating inferences *)
```

# Discount Saturation Algorithm

```
  input: init: set of clauses;
  var active, passive, unprocessed: set of clauses;
  var given, new: clause;
  active := ∅;
  unprocessed := init;
  loop
    while unprocessed ≠ ∅
      new:=pop(unprocessed);
      if new = □ then return unsatisfiable;
*     if retained(new) then                                  (* retention test *)
*       simplify new by clauses in active ;              (* forward simplification *)
        if new = □ then return unsatisfiable;
*       if retained(new) then                                (* retention test *)
*         delete and simplify clauses in active using new;(* backward simplification *)
          move the simplified clauses to unprocessed;
          add new to passive
    if passive = ∅ then return satisfiable or unknown
*   given := select(passive);                             (* clause selection *)
*   simplify given by clauses in active;                 (* forward simplification *)
*   if given = □ then return unsatisfiable;
    if retained(given) then                                  (* retention test *)
*     delete and simplify clauses in active using given;  (* backward simplification *)
      move the simplified clauses to unprocessed;
      add given to active;
      unprocessed:=infer(given, active);                (* generating inferences *)
```

# Age-Weight Ratio

How to select nice clauses?

- Small clauses are nice.
- Selecting only small clauses can postpone the selection of an old clause (e.g., input clause) for too long, in practice resulting in incompleteness.

# Age-Weight Ratio

How to select nice clauses?

- Small clauses are nice.
- Selecting only small clauses can postpone the selection of an old clause (e.g., input clause) for too long, in practice resulting in incompleteness.

Solution:

- A fixed percentage of clauses is selected by weight, the rest are selected by age.
- So we use an age-weight ratio $a : w$: of each $a + w$ clauses select $a$ oldest and $w$ smallest clauses.

# Limited Resource Strategy

Limited Resource Strategy: try to approximate which clauses are unreachable by the end of the time limit and remove them from the search space.

# Limited Resource Strategy

Limited Resource Strategy: try to approximate which clauses are unreachable by the end of the time limit and remove them from the search space.

Try: `./vampire -awr 5:1 -fsr off Problems/GRP140-1.p`

# What is AVATAR?

# What is AVATAR?

## AVATAR [Voronkov'14]

- modern architecture of first order theorem provers
- integrates *saturation* with a *SAT solver*
- efficient realization of the *clause splitting rule*
- *instead of one* monolithic proof search
  *a sequence* of proof searches on (much) smaller sub-problems

- implemented in theorem prover Vampire
- shown highly successful in practice

# Clause splitting

## Central idea

Let $C_1$ and $C_2$ are variable disjoint. Then the clause set

$$S \cup \{C_1 \vee C_2\} \text{ is unsatisfiable}$$

if and only if

$$\text{both } S \cup \{C_1\} \text{ and } S \cup \{C_2\} \text{ are unsatisfiable.}$$

# Clause splitting

## Central idea

Let $C_1$ and $C_2$ are variable disjoint. Then the clause set

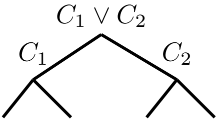
$$S \cup \{C_1 \vee C_2\} \text{ is unsatisfiable}$$

if and only if

$$\text{both } S \cup \{C_1\} \text{ and } S \cup \{C_2\} \text{ are unsatisfiable.}$$

## Previous approaches to splitting

- splitting with backtracking [Wei01]




- splitting without backtracking [RV01]

$$p_1 \vee p_2 \quad \neg p_1 \vee C_1 \quad \neg p_2 \vee C_2$$

# Splitting as much as possible

## Components of a clause

- (non-empty) sub-clauses which do not share a variable
- finest decomposition with this property

# Splitting as much as possible

## Components of a clause

- (non-empty) sub-clauses which do not share a variable
- finest decomposition with this property

## Example (a clause splittable into two components)

$$\forall X, Y, Z \quad p(X, f(Y)) \vee \neg q(Y) \vee c \simeq Z$$

$$\equiv$$

$$\forall X, Y\, [p(X, f(Y)) \vee \neg q(Y)] \quad \vee \quad \forall Z\, c \simeq Z$$

# Building blocks of AVATAR

## Naming of components

A splittable first-order clause abstracted to a SAT clause

$$C_1 \lor \ldots \lor C_n \quad \rightsquigarrow \quad [C_1] \lor \ldots \lor [C_n]$$

SAT solver makes the splitting decisions

- the "propositional essence" of the given problem delegated to the efficient dedicated solver

# Building blocks of AVATAR

## Naming of components

A splittable first-order clause abstracted to a SAT clause

$$C_1 \vee \ldots \vee C_n \quad \rightsquigarrow \quad [C_1] \vee \ldots \vee [C_n]$$

SAT solver makes the splitting decisions

- the "propositional essence" of the given problem delegated to the efficient dedicated solver

## Proving under assumptions

Components selected by the SAT solver are exposed as

$$C \leftarrow [C],$$

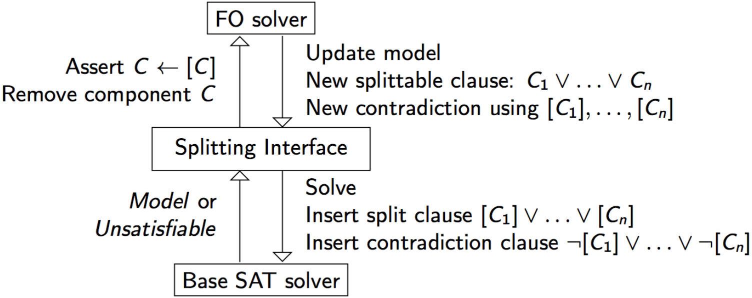
inferences keep track of dependencies

$$\frac{(l \vee C_1) \leftarrow A_1 \qquad (\neg l \vee C_2) \leftarrow A_2}{(C_1 \vee C_2) \leftarrow A_1 \wedge A_2},$$

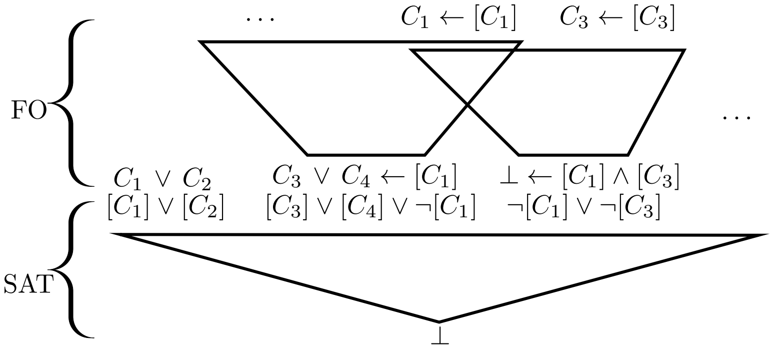
"conditional empty clauses"; sent back to the SAT solver:

$$\bot \leftarrow [C'_1] \wedge \ldots \wedge [C'_k] \quad \rightsquigarrow \quad \neg[C'_1] \vee \ldots \vee \neg[C'_k]$$

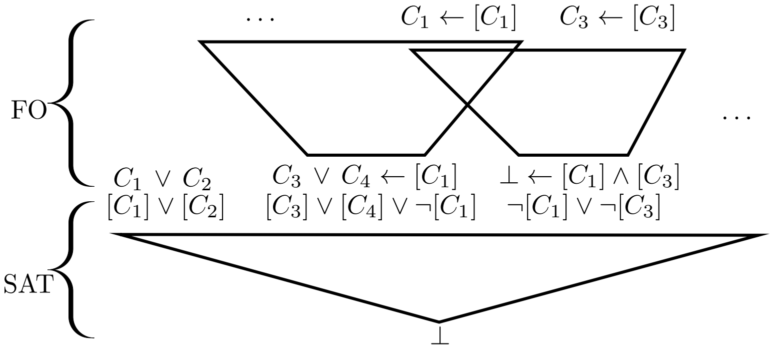
# The AVATAR architecture



# The shape of AVATAR refutation



# The shape of AVATAR refutation




```
./vampire --proof on PUZ001+1.p
```

How can one efficiently apply complex operations to hundreds of thousands of terms and clauses?

# Term Indexing

How can one efficiently apply complex operations to hundreds of thousands of terms and clauses?

Given a set $\mathcal{L}$ (the set of indexed terms), a binary relation $R$ over terms (the retrieval condition) and a term $t$ (called the query term), identify the subset $\mathcal{M}$ of $\mathcal{L}$ consisting of all of the terms $l$ such that $R(l, t)$ holds.

# Term Indexing

How can one efficiently apply complex operations to hundreds of thousands of terms and clauses?
Given a set $\mathcal{L}$ (the set of indexed terms), a binary relation $R$ over terms (the retrieval condition) and a term $t$ (called the query term), identify the subset $\mathcal{M}$ of $\mathcal{L}$ consisting of all of the terms $l$ such that $R(l, t)$ holds.
The problem (and solution) is similar to database query answering, but data are much more complex than relational data (a clause is a finite set of trees, so the search space is a (large) set of finite sets of trees).

# Term Indexing

How can one efficiently apply complex operations to hundreds of thousands of terms and clauses?

Given a set $\mathcal{L}$ (the set of indexed terms), a binary relation $R$ over terms (the retrieval condition) and a term $t$ (called the query term), identify the subset $\mathcal{M}$ of $\mathcal{L}$ consisting of all of the terms $l$ such that $R(l, t)$ holds.

The problem (and solution) is similar to database query answering, but data are much more complex than relational data (a clause is a finite set of trees, so the search space is a (large) set of finite sets of trees).

One puts the clauses in $\mathcal{L}$ in a data structure, called the index. The data structure is designed with the only purpose to make the retrieval fast.

# Term Indexing

- Different indexes are needed to support different operations;
- The set of clauses is dynamically (and often) changes, so that index maintenance must be efficient.
- Memory is an issue (badly designed indexes may take much more space than clauses).
- The inverse retrieval conditions (the same algorithm on clauses) may require very different indexing techniques (e.g., forward and backward subsumption).
- Sensitive to the signature of the problem: techniques good for small signatures are too slow and too memory consuming for large signatures.

# Term Indexing in Vampire

- Various hash tables.
- Flatterms in constant memory for storing temporary clauses.
- Code trees for forward subsumption;
- Code trees with precompiled ordering constraints;
- Discrimination trees;
- Substitution trees;
- Variables banks;
- Shared terms with renaming lists;
- Path index with compiled database joins;
- …

# Outline


Preliminaries

Theorem-Proving Workflow

A Static View: Inferences, Soundness, and Completeness

A Dynamic View: Saturation

Making It Fast in Practice

Let's Try Proving Something – a Mini-Challenge

# Dark art mini-CHALLENGE

```
./vampire --show_options on
```

## Some options to play with

1. Set of support (`-sos on`)
2. AVATAR turned off (`-spl off`)
   default: 9552; sploff: 8700, also 234 new
3. Discount saturation loop and the age-weight ratio (`-sa discount -awr 10`)
   discount only: 9421; with awr10: 9577
4. default: 9552; lookahead: 8937 but 839 new
5. Backward subsumption (`-bs on`)

# Dark art mini-CHALLENGE

```
./vampire --show_options on
```

## Some options to play with

1. Set of support (`-sos on`)

2. AVATAR turned off (`-spl off`)
   default: 9552; sploff: 8700, also 234 new

3. Discount saturation loop and the age-weight ratio
   (`-sa discount -awr 10`)
   discount only: 9421; with awr10: 9577

4. default: 9552; lookahead: 8937 but 839 new

5. Backward subsumption (`-bs on`)

# Ready made solution from the Vizzard

## Portfolio mode (a.k.a. CASC mode)

- a conditional portfolio mode
- a cocktail of a strategies optimized for good general performance
- incomplete strategies in the mix; complementarity for coverage
- `--mode casc` (there is also `--mode casc_sat`)
- The schedule is 5+ minutes long (use with `-t 5m`)
- `--cores <number>` for executing in parallel

## A small experiment (5 minutes time limit)

| | | |
|---|---|---|
| TPTP 7.0.0 total: | 21851 | |
| Discarded (hol + poly): | 4323 | |
| Eligible (cnf, fof, tff): | 17528 | |
| casc: | 13460 | 76.8 % |
| casc_sat: | 10434 | 59.5 % |
| union: | 14125 | 80.6 % |

# Ready made solution from the Vizzard

## Portfolio mode (a.k.a. CASC mode)

- a conditional portfolio mode
- a cocktail of a strategies optimized for good general performance
- incomplete strategies in the mix; complementarity for coverage
- `--mode casc` (there is also `--mode casc_sat`)
- The schedule is 5+ minutes long (use with `-t 5m`)
- `--cores <number>` for executing in parallel

## A small experiment (5 minutes time limit)

| | | |
|---|---|---|
| TPTP 7.0.0 total: | 21851 | |
| Discarded (hol + poly): | 4323 | |
| Eligible (cnf, fof, tff): | 17528 | |
| casc: | 13460 | 76.8 % |
| casc_sat: | 10434 | 59.5 % |
| union: | 14125 | 80.6 % |