First-Order Theorem Proving in Rigorous Systems Engineering

Laura Kovács and Andrei Voronkov

TU Wien and University of Manchester

Schedule

Today

Session 1 - Getting Started with Vampire

Session 2 - Overview on First-Order Theorem Proving

Vampire in Practice

Session 3 - Cookies

Session 4 - Interpolation

Schedule

Today

Session 1 - Getting Started with Vampire

Session 2 - Overview on First-Order Theorem Proving

Vampire in Practice

Session 3 - Cookies

Session 4 - Interpolation

Outline

Automated Reasoning and Rigorous Systems Engineering

In a vague sense, automated reasoning involves

1. Representing a problem as a mathematical/logical statement
2. Automatically checking this statement's consistency or truth

In rigorous systems engineering there are lots of places where we can apply this reasoning. For example,

- Proving partial correctness properties
- Generating loop invariants
- Synthesis
- Model checking
- Your idea?

Kinds of Reasoning

Given a statement S we can establish different conclusions about it

- Consistency - there is a way of making it true
- Inconsistency - there is no way of making it true
- Validity - it is always true

We can look at these three notions from two different views.

	Semantic view	Syntactic view
S is consistent	A model	No proof of $\bot$ from S
S is inconsistent	No model	A proof of $\bot$ from S
S is valid	True in all models	A proof of $\bot$ from $\neg S$

Notes

1. Here we have focussed only on proofs of inconsistency.
2. Consistency is commonly referred to as satisfiability

Models and Proofs

Models

A model is a structure that can be used to interpret the symbols in a logical statement making the statement semantically true.
S can have 0, 1, n, or ∞ models. A model may be infinite.
New expressions can be evaluated in a model.
If our statement is of the form $\neg S$ then we have a countermodel.

Proofs (in our context)

A proof is a sequence of derived statements that follow logically from the input, ending in a contradiction.
Steps may preserve validity, satisfiability, or models.
A proof may only use part of S and may introduce new symbols.

Kinds of Reasoners

	Input	Example(s)
SAT Solvers	Propositional formulae	MiniSat
Constraint Solvers	Conjunction of theory constraints	
SMT Solvers	(First-order) formulae + theories	Z3,CVC4
Theorem Provers	First-order formulae (+ theories)	Vampire,E
Proof Assistants (interactive)	High-order formulae	Isabelle,Coq

Above the line focus on models and might be decidable. Below the line focus on proofs and are rarely decidable.

More about Logics

Propositional Logic

Propositions and boolean constructors e.g. $p \wedge q$, good $\rightarrow \neg$bad
Common normal forms (CNF, DNF). Models are assignments of true/false to propositions. SAT solvers. QBF.

First-Order or Predicate Logic

Adds predicates, functions, quantifiers, equality e.g. $\exists x : f(x) \neq x$, $\forall x : \text{man}(x) \rightarrow \text{human}(x)$. Skolemisation can remove $\exists$. Models interpret each predicate and function symbol.

Theories

Fix a class of interpretations for a subset of the signature e.g. $+$.

Higher-Order Logic

Allow quantification over functions.

Other Logics (live inside one of the above)

Modal logics (e.g. LTL). Description logics. Seperation Logic.

What is Vampire?

An automated theorem prover for first-order logic and theories

- ▷ It produces detailed proofs but also supports finite model finding
- ▷ It is very fast (44 trophies from CASC over the last 18 years)
- ▷ It competes with SMT solvers on their problems
- ▷ In normal operation it is saturation-based - it saturates a clausal form with respect to an inference system
- ▷ It is portfolio-based - it works best when you allow it to try lots of strategies
- ▷ It has unique proof search features such as LRS and AVATAR
- ▷ It supports lots of extra features helpful for rigorous systems engineering

CASC 2017 results

Higher-order Theorems	Satallax 3.2	Leo-III 1.1	*Satallax 3.0*	LEO-II 1.7.0	Zipperpin 1.1	Isabelle 2016
Solved/500	430/500	382/500	382/500	305/500	179/500	387/500
Solutions	430/500	382/500	375/500	301/500	179/500	0/500

Typed First-order Theorems +*-/	*Vampire 4.1*	Vampire 4.2	CVC4 ARI-1.5.2	Princess 170717	Zipperpin 1.1
Solved/250	194/250	191/250	188/250	130/250	39/250
Solutions	194/250	191/250	188/250	115/250	39/250

First-order Theorems	Vampire 4.2	*Vampire 4.0*	E 2.1	CVC4 NAR-1.5.2	iProver 2.6	Leo-III 1.1	lean-nan 1.0	Zipperpin 1.1	Prover9 1109a	iProverM 2.5-0.1	Scavenger EP-0.2
Solved/500	452/500	444/500	381/500	327/500	283/500	211/500	186/500	154/500	140/500	99/500	71/500
Solutions	452/500	440/500	381/500	327/500	279/500	211/500	186/500	154/500	138/500	99/500	71/500

First-order Non-theorems	*Vampire SAT-4.1*	Vampire SAT-4.2	iProver SAT-2.6	CVC4 SNA-1.5.2	E FNT-2.1	Scavenger EP-0.2
Solved/250	219/250	217/250	175/250	136/250	85/250	12/250
Solutions	217/250	204/250	175/250	136/250	85/250	12/250

Effectively Propositional CNF	iProver 2.6	*iProver 2.5*	Vampire 4.2	E 2.1	Scavenger EP-0.1	Scavenger EP-0.2
Solved/200	174/200	171/200	168/200	53/200	5/200	4/200

SLedgeHammer Theorems	Vampire SLH-4.2	CVC4 SLH-1.5.2	ET 2.0	E SLH-2.1	Leo-III SLH-1.1	iProver SLH-2.6	Zipperpin SLH-1.1	iProverM 2.5-0.1
Solved/2000	1433/2000	1364/2000	1328/2000	1185/2000	652/2000	519/2000	472/2000	320/2000

Large Theory Batch Problems	*Vampire LTB-4.0*	Vampire LTB-4.2	MaLARe 0.6	iProver LTB-2.6	E LTB-2.1
Solved/1500	1156/1500	1144/1486	1131/1500	777/1499	683/1499
Solutions	1156/1500	1144/1486	1131/1500	777/1499	683/1499

The Vampire Team

University of Manchester, UK

Andrei Voronkov

Giles Reger

- Ahmed Bhayat
- Michael Rawson

TU Wien, Austria

Laura Kovacs

Martin Suda

- Evgenii Kotelnikov
- Simon Robillard
- Bernhard Gleiss

Issues to Consider when using Reasoners in Rigorous Systems Engineering

(that we won't necessarily be addressing)

Representing Programs

- How do we encode a program's behaviour in logic
- What abstraction do we want for different types of data
- How do we handle modularity, non-determinism etc

Representing Queries

- Is your query a traditional *does C follow from A*?
- or something else? e.g. which statements among $S_1, \ldots, S_n$ are redundant (implied by other statements)
- How do you turn what you want into a first-order query?

Handling Query Answers

- Is it just a yes/no answer you want?
- Or do you need a model or proof to extract further information from

Outline

Download and Install

Go to

`https://vprover.github.io/download.html`

and pick the route most suitable to you.

Notes:

- For Linux users, a binary is probably the easiest route
- For Mac users, you need to build from source
  - run `make vampire_rel`
- For Windows users, the easiest route for this tutorial is a virtual machine and then use Linux

The TPTP Library

The TPTP library (Thousands of Problems for Theorem Provers), http://www.tptp.org contains a large collection of first-order problems. For representing these problems it uses the TPTP syntax, which is understood by all modern theorem provers, including Vampire.

Hello World in TPTP (`hello-world-1.p`)

In this example:

- `logic(name,type,formula)` syntax
- Predicates (`hello`) and constants (`world`)
- Logical operators: `=>` but also `|`,`~`,`&`

We have three axioms giving some rules and a conjecture that we want to show follows from these axioms.

```
fof(a1,axiom,hello(world) => hello(austria)).
fof(a2,axiom,hello(austria) => hello(vienna)).
fof(a3,axiom,hello(vienna) => hello(riseWS)).
fof(con,conjecture,hello(world) => hello(riseWS)).
```

Hello World in TPTP (`hello-world-1.p`)

In this example:

- `logic(name,type,formula)` syntax
- Predicates (`hello`) and constants (`world`)
- Logical operators: `=>` but also `|`,`~`,`&`

We have three axioms giving some rules and a conjecture that we want to show follows from these axioms.

```
fof(a1,axiom,hello(world) => hello(austria)).
fof(a2,axiom,hello(austria) => hello(vienna)).
fof(a3,axiom,hello(vienna) => hello(riseWS)).
fof(con,conjecture,hello(world) => hello(riseWS)).
```

Hello World in TPTP (`hello-world-1.p`)

In this example:

- `logic(name,type,formula)` syntax
- Predicates (`hello`) and constants (`world`)
- Logical operators: `=>` but also `|`,`~`,`&`

We have three axioms giving some rules and a conjecture that we want to show follows from these axioms.

```
fof(a1,axiom,hello(world) => hello(austria)).
fof(a2,axiom,hello(austria) => hello(vienna)).
fof(a3,axiom,hello(vienna) => hello(riseWS)).
fof(con,conjecture,hello(world) => hello(riseWS)).
```

Hello World in TPTP (`hello-world-1.p`)

In this example:

- `logic(name,type,formula)` syntax
- Predicates (`hello`) and constants (`world`)
- Logical operators: `=>` but also `|`,`~`,`&`

We have three axioms giving some rules and a conjecture that we want to show follows from these axioms.

```
fof(a1,axiom,hello(world) => hello(austria)).
fof(a2,axiom,hello(austria) => hello(vienna)).
fof(a3,axiom,hello(vienna) => hello(riseWS)).
fof(con,conjecture,hello(world) => hello(riseWS)).
```

Hello World in TPTP (`hello-world-1.p`)

In this example:

- `logic(name,type,formula)` syntax
- Predicates (`hello`) and constants (`world`)
- Logical operators: `=>` but also `|`,`~`,`&`

We have three axioms giving some rules and a conjecture that we want to show follows from these axioms.

```
fof(a1,axiom,hello(world) => hello(austria)).
fof(a2,axiom,hello(austria) => hello(vienna)).
fof(a3,axiom,hello(vienna) => hello(riseWS)).
fof(con,conjecture,hello(world) => hello(riseWS)).
```

Hello World in TPTP (`hello-world-1.p`)

In this example:

- `logic(name,type,formula)` syntax
- Predicates (`hello`) and constants (`world`)
- Logical operators: `=>` but also `|`,`~`,`&`

We have three axioms giving some rules and a conjecture that we want to show follows from these axioms.

```
fof(a1,axiom,hello(world) => hello(austria)).
fof(a2,axiom,hello(austria) => hello(vienna)).
fof(a3,axiom,hello(vienna) => hello(riseWS)).
fof(con,conjecture,hello(world) => hello(riseWS)).
```

Hello World in TPTP (`hello-world-1.p`)

In this example:

- `logic(name,type,formula)` syntax
- Predicates (`hello`) and constants (`world`)
- Logical operators: `=>` but also `|`,`~`,`&`

We have three axioms giving some rules and a conjecture that we want to show follows from these axioms.

```
fof(a1,axiom,hello(world) => hello(austria)).
fof(a2,axiom,hello(austria) => hello(vienna)).
fof(a3,axiom,hello(vienna) => hello(riseWS)).
fof(con,conjecture,hello(world) => hello(riseWS)).
```

Hello World (a proof)

```
fof(a1,axiom,hello(world) => hello(austria)).
fof(a2,axiom,hello(austria) => hello(viennna)).
fof(a3,axiom,hello(vienna) => hello(riseWS)).
fof(con,conjecture,hello(world) => hello(riseWS)).
```

```
vampire -av off hello_world-1.p
```

```
10. ~hello(world) | hello(austria) [cnf transformation 6]
11. ~hello(austria) | hello(vienna) [cnf transformation 7]
12. ~hello(vienna) | hello(riseWS) [cnf transformation 8]
13. hello(world) [cnf transformation 9]
14. ~hello(riseWS) [cnf transformation 9]
15. hello(austria) [resolution 10,13]
16. hello(vienna) [resolution 15,11]
17. hello(riseWS) [resolution 16,12]
18. $false [subsumption resolution 17,14]
```

Hello World (a proof)

```
fof(a1,axiom,hello(world) => hello(austria)).
fof(a2,axiom,hello(austria) => hello(viennna)).
fof(a3,axiom,hello(vienna) => hello(riseWS)).
fof(con,conjecture,hello(world) => hello(riseWS)).

vampire -av off hello_world-1.p

10. ~hello(world) | hello(austria) [cnf transformation 6]
11. ~hello(austria) | hello(vienna) [cnf transformation 7]
12. ~hello(vienna) | hello(riseWS) [cnf transformation 8]
13. hello(world) [cnf transformation 9]
14. ~hello(riseWS) [cnf transformation 9]
15. hello(austria) [resolution 10,13]
16. hello(vienna) [resolution 15,11]
17. hello(riseWS) [resolution 16,12]
18. $false [subsumption resolution 17,14]
```

Hello World (a proof)

```
fof(a1,axiom,hello(world) => hello(austria)).
fof(a2,axiom,hello(austria) => hello(viennna)).
fof(a3,axiom,hello(vienna) => hello(riseWS)).
fof(con,conjecture,hello(world) => hello(riseWS)).

vampire -av off hello_world-1.p

10. ~hello(world) | hello(austria) [cnf transformation 6]
11. ~hello(austria) | hello(vienna) [cnf transformation 7]
12. ~hello(vienna) | hello(riseWS) [cnf transformation 8]
13. hello(world) [cnf transformation 9]
14. ~hello(riseWS) [cnf transformation 9]
15. hello(austria) [resolution 10,13]
16. hello(vienna) [resolution 15,11]
17. hello(riseWS) [resolution 16,12]
18. $false [subsumption resolution 17,14]
```

Hello World (a proof)

```
fof(a1,axiom,hello(world) => hello(austria)).
fof(a2,axiom,hello(austria) => hello(viennna)).
fof(a3,axiom,hello(vienna) => hello(riseWS)).
fof(con,conjecture,hello(world) => hello(riseWS)).

vampire -av off hello_world-1.p

10. ~hello(world) | hello(austria) [cnf transformation 6]
11. ~hello(austria) | hello(vienna) [cnf transformation 7]
12. ~hello(vienna) | hello(riseWS) [cnf transformation 8]
13. hello(world) [cnf transformation 9]
14. ~hello(riseWS) [cnf transformation 9]
15. hello(austria) [resolution 10,13]
16. hello(vienna) [resolution 15,11]
17. hello(riseWS) [resolution 16,12]
18. $false [subsumption resolution 17,14]
```

Hello World (a proof)

```
fof(a1,axiom,hello(world) => hello(austria)).
fof(a2,axiom,hello(austria) => hello(viennna)).
fof(a3,axiom,hello(vienna) => hello(riseWS)).
fof(con,conjecture,hello(world) => hello(riseWS)).

vampire -av off hello_world-1.p

10. ~hello(world) | hello(austria) [cnf transformation 6]
11. ~hello(austria) | hello(vienna) [cnf transformation 7]
12. ~hello(vienna) | hello(riseWS) [cnf transformation 8]
13. hello(world) [cnf transformation 9]
14. ~hello(riseWS) [cnf transformation 9]
15. hello(austria) [resolution 10,13]
16. hello(vienna) [resolution 15,11]
17. hello(riseWS) [resolution 16,12]
18. $false [subsumption resolution 17,14]
```

Hello World (a proof)

```
fof(a1,axiom,hello(world) => hello(austria)).
fof(a2,axiom,hello(austria) => hello(viennna)).
fof(a3,axiom,hello(vienna) => hello(riseWS)).
fof(con,conjecture,hello(world) => hello(riseWS)).

vampire -av off hello_world-1.p

10. ~hello(world) | hello(austria) [cnf transformation 6]
11. ~hello(austria) | hello(vienna) [cnf transformation 7]
12. ~hello(vienna) | hello(riseWS) [cnf transformation 8]
13. hello(world) [cnf transformation 9]
14. ~hello(riseWS) [cnf transformation 9]
15. hello(austria) [resolution 10,13]
16. hello(vienna) [resolution 15,11]
17. hello(riseWS) [resolution 16,12]
18. $false [subsumption resolution 17,14]
```

Hello World (a proof)

```
fof(a1,axiom,hello(world) => hello(austria)).
fof(a2,axiom,hello(austria) => hello(viennna)).
fof(a3,axiom,hello(vienna) => hello(riseWS)).
fof(con,conjecture,hello(world) => hello(riseWS)).

vampire -av off hello_world-1.p

10. ~hello(world) | hello(austria) [cnf transformation 6]
11. ~hello(austria) | hello(vienna) [cnf transformation 7]
12. ~hello(vienna) | hello(riseWS) [cnf transformation 8]
13. hello(world) [cnf transformation 9]
14. ~hello(riseWS) [cnf transformation 9]
15. hello(austria) [resolution 10,13]
16. hello(vienna) [resolution 15,11]
17. hello(riseWS) [resolution 16,12]
18. $false [subsumption resolution 17,14]
```

Hello World (a proof)

```
fof(a1,axiom,hello(world) => hello(austria)).
fof(a2,axiom,hello(austria) => hello(viennna)).
fof(a3,axiom,hello(vienna) => hello(riseWS)).
fof(con,conjecture,hello(world) => hello(riseWS)).

vampire -av off hello_world-1.p

10. ~hello(world) | hello(austria) [cnf transformation 6]
11. ~hello(austria) | hello(vienna) [cnf transformation 7]
12. ~hello(vienna) | hello(riseWS) [cnf transformation 8]
13. hello(world) [cnf transformation 9]
14. ~hello(riseWS) [cnf transformation 9]
15. hello(austria) [resolution 10,13]
16. hello(vienna) [resolution 15,11]
17. hello(riseWS) [resolution 16,12]
18. $false [subsumption resolution 17,14]
```

Hello World (a proof)

```
fof(a1,axiom,hello(world) => hello(austria)).
fof(a2,axiom,hello(austria) => hello(viennna)).
fof(a3,axiom,hello(vienna) => hello(riseWS)).
fof(con,conjecture,hello(world) => hello(riseWS)).

vampire -av off hello_world-1.p

10. ~hello(world) | hello(austria) [cnf transformation 6]
11. ~hello(austria) | hello(vienna) [cnf transformation 7]
12. ~hello(vienna) | hello(riseWS) [cnf transformation 8]
13. hello(world) [cnf transformation 9]
14. ~hello(riseWS) [cnf transformation 9]
15. hello(austria) [resolution 10,13]
16. hello(vienna) [resolution 15,11]
17. hello(riseWS) [resolution 16,12]
18. $false [subsumption resolution 17,14]
```

Hello World (a proof)

```
fof(a1,axiom,hello(world) => hello(austria)).
fof(a2,axiom,hello(austria) => hello(viennna)).
fof(a3,axiom,hello(vienna) => hello(riseWS)).
fof(con,conjecture,hello(world) => hello(riseWS)).

vampire -av off hello_world-1.p

10. ~hello(world) | hello(austria) [cnf transformation 6]
11. ~hello(austria) | hello(vienna) [cnf transformation 7]
12. ~hello(vienna) | hello(riseWS) [cnf transformation 8]
13. hello(world) [cnf transformation 9]
14. ~hello(riseWS) [cnf transformation 9]
15. hello(austria) [resolution 10,13]
16. hello(vienna) [resolution 15,11]
17. hello(riseWS) [resolution 16,12]
18. $false [subsumption resolution 17,14]
```

Hello World (a proof)

```
fof(a1,axiom,hello(world) => hello(austria)).
fof(a2,axiom,hello(austria) => hello(viennna)).
fof(a3,axiom,hello(vienna) => hello(riseWS)).
fof(con,conjecture,hello(world) => hello(riseWS)).

vampire -av off hello_world-1.p

10. ~hello(world) | hello(austria) [cnf transformation 6]
11. ~hello(austria) | hello(vienna) [cnf transformation 7]
12. ~hello(vienna) | hello(riseWS) [cnf transformation 8]
13. hello(world) [cnf transformation 9]
14. ~hello(riseWS) [cnf transformation 9]
15. hello(austria) [resolution 10,13]
16. hello(vienna) [resolution 15,11]
17. hello(riseWS) [resolution 16,12]
18. $false [subsumption resolution 17,14]
```

Hello World Again, more on TPTP (`hello-world-2.p`)

In this example

- Variables (upper case)
- Quantification (`!` for $\forall$, `?` for $\exists$)

We can rewrite the previous problem to axiomatise the idea that saying hello to something means saying hello to its parts.

```
fof(a1,axiom, ![X,Y] : (
  (has_part(X,Y) & hello(X)) => hello(Y)
)).
fof(f1,axiom, has_part(world,austria)).
fof(f1,axiom, has_part(austria,vienna)).
fof(f1,axiom, has_part(vienna,riseWS)).
fof(con,conjecture,hello(world) => hello(riseWS)).
```

Hello World Again, more on TPTP (`hello-world-2.p`)

In this example

- Variables (upper case)
- Quantification (`!` for $\forall$, `?` for $\exists$)

We can rewrite the previous problem to axiomatise the idea that saying hello to something means saying hello to its parts.

```
fof(a1,axiom, ![X,Y] : (
  (has_part(X,Y) & hello(X)) => hello(Y)
)).
fof(f1,axiom, has_part(world,austria)).
fof(f1,axiom, has_part(austria,vienna)).
fof(f1,axiom, has_part(vienna,riseWS)).
fof(con,conjecture,hello(world) => hello(riseWS)).
```

Hello World Again, more on TPTP (`hello-world-2.p`)

In this example

- Variables (upper case)
- Quantification (`!` for $\forall$, `?` for $\exists$)

We can rewrite the previous problem to axiomatise the idea that saying hello to something means saying hello to its parts.

```
fof(a1,axiom, ![X,Y] : (
  (has_part(X,Y) & hello(X)) => hello(Y)
)).
fof(f1,axiom, has_part(world,austria)).
fof(f1,axiom, has_part(austria,vienna)).
fof(f1,axiom, has_part(vienna,riseWS)).
fof(con,conjecture,hello(world) => hello(riseWS)).
```

Group Theory (`group.p`)

Group theory theorem: if a group satisfies the identity $x^2 = 1$, then it is commutative.

Group Theory (`group.p`)

Group theory theorem: if a group satisfies the identity $x^2 = 1$, then it is commutative.
More formally: in a group "assuming that $x^2 = 1$ for all x prove that $x \cdot y = y \cdot x$ holds for all x, y."

Group Theory (`group.p`)

Group theory theorem: if a group satisfies the identity $x^2 = 1$, then it is commutative.
More formally: in a group "assuming that $x^2 = 1$ for all x prove that $x \cdot y = y \cdot x$ holds for all x, y."
What is implicit: axioms of the group theory.

$$\forall x(1 \cdot x = x)$$
$$\forall x(x^{-1} \cdot x = 1)$$
$$\forall x \forall y \forall z((x \cdot y) \cdot z = x \cdot (y \cdot z))$$

Group Theory (`group.p`)

Formulation in First-Order Logic:

Axioms (of group theory):	$\forall x(1 \cdot x = x)$
	$\forall x(x^{-1} \cdot x = 1)$
	$\forall x \forall y \forall z((x \cdot y) \cdot z = x \cdot (y \cdot z))$
Assumptions:	$\forall x(x \cdot x = 1)$
Conjecture:	$\forall x \forall y(x \cdot y = y \cdot x)$

Group Theory (`group.p`) – in TPP

```
%---- 1 * x = x
fof(left_identity,axiom,
    ! [X] : mult(e,X) = X).
%---- i(x) * x = 1
fof(left_inverse,axiom,
    ! [X] : mult(inverse(X),X) = e).
%---- (x * y) * z = x * (y * z)
fof(associativity,axiom,
    ! [X,Y,Z] : mult(mult(X,Y),Z) = mult(X,mult(Y,Z))).
%---- x * x = 1
fof(group_of_order_2,hypothesis,
    ! [X] : mult(X,X) = e).
%---- prove x * y = y * x
fof(commutativity,conjecture,
    ! [X] : mult(X,Y) = mult(Y,X)).
```

First-Order Logic and TPTP

FOL	TPTP
$\bot, \top$	`$false, $true`
$\neg a$	`~a`
$a_1 \wedge \ldots \wedge a_n$	`a1 & ... & an`
$a_1 \vee \ldots \vee a_n$	`a1 | ... | an`
$a_1 \rightarrow a_2$	`a1 => a2`
$(\forall x_1) \ldots (\forall x_n) a$	`! [X1,...,Xn] : a`
$(\exists x_1) \ldots (\exists x_n) a$	`? [X1,...,Xn] : a`

More on the TPTP Syntax

```
%---- 1 * x = x
fof(left_identity,axiom,(
  ! [X] : mult(e,X) = X )).
%---- i(x) * x = 1
fof(left_inverse,axiom,(
  ! [X] : mult(inverse(X),X) = e )).
%---- (x * y) * z = x * (y * z)
fof(associativity,axiom,(
  ! [X,Y,Z] :
      mult(mult(X,Y),Z) = mult(X,mult(Y,Z)) )).
%---- x * x = 1
fof(group_of_order_2,hypothesis,
  ! [X] : mult(X,X) = e ).
%---- prove x * y = y * x
fof(commutativity,conjecture,
  ! [X,Y] : mult(X,Y) = mult(Y,X) ).
```

More on the TPTP Syntax

- Comments;

```
%---- 1 * x = x
fof(left_identity,axiom,(
  ! [X] : mult(e,X) = X )).
%---- i(x) * x = 1
fof(left_inverse,axiom,(
  ! [X] : mult(inverse(X),X) = e )).
%---- (x * y) * z = x * (y * z)
fof(associativity,axiom,(
  ! [X,Y,Z] :
      mult(mult(X,Y),Z) = mult(X,mult(Y,Z)) )).
%---- x * x = 1
fof(group_of_order_2,hypothesis,
  ! [X] : mult(X,X) = e ).
%---- prove x * y = y * x
fof(commutativity,conjecture,
  ! [X,Y] : mult(X,Y) = mult(Y,X) ).
```

More on the TPTP Syntax

- Comments;
- Input formula names;

```
%---- 1 * x = x
fof(left_identity,axiom,(
  ! [X] : mult(e,X) = X )).
%---- i(x) * x = 1
fof(left_inverse,axiom,(
  ! [X] : mult(inverse(X),X) = e )).
%---- (x * y) * z = x * (y * z)
fof(associativity,axiom,(
  ! [X,Y,Z] :
      mult(mult(X,Y),Z) = mult(X,mult(Y,Z)) )).
%---- x * x = 1
fof(group_of_order_2,hypothesis,
  ! [X] : mult(X,X) = e ).
%---- prove x * y = y * x
fof(commutativity,conjecture,
  ! [X,Y] : mult(X,Y) = mult(Y,X) ).
```

More on the TPTP Syntax

- Comments;
- Input formula names;
- Input formula roles (very important);

```
%---- 1 * x = x
fof(left_identity,axiom,(
  ! [X] : mult(e,X) = X )).
%---- i(x) * x = 1
fof(left_inverse,axiom,(
  ! [X] : mult(inverse(X),X) = e )).
%---- (x * y) * z = x * (y * z)
fof(associativity,axiom,(
  ! [X,Y,Z] :
      mult(mult(X,Y),Z) = mult(X,mult(Y,Z)) )).
%---- x * x = 1
fof(group_of_order_2,hypothesis,
  ! [X] : mult(X,X) = e ).
%---- prove x * y = y * x
fof(commutativity,conjecture,
  ! [X,Y] : mult(X,Y) = mult(Y,X) ).
```

More on the TPTP Syntax

- Comments;
- Input formula names;
- Input formula roles (very important);
- Equality

```
%---- 1 * x = x
fof(left_identity,axiom,(
  ! [X] : mult(e,X) = X )).
%---- i(x) * x = 1
fof(left_inverse,axiom,(
  ! [X] : mult(inverse(X),X) = e )).
%---- (x * y) * z = x * (y * z)
fof(associativity,axiom,(
  ! [X,Y,Z] :
      mult(mult(X,Y),Z) = mult(X,mult(Y,Z)) )).
%---- x * x = 1
fof(group_of_order_2,hypothesis,
  ! [X] : mult(X,X) = e ).
%---- prove x * y = y * x
fof(commutativity,conjecture,
  ! [X,Y] : mult(X,Y) = mult(Y,X) ).
```

Proof by Vampire (Slightliy Modified)

```
Refutation found.
270. $false [trivial inequality removal 269]
269. mult(sk0,sk1) != mult (sk0,sk1) [superposition 14,125]
125. mult(X2,X3) = mult(X3,X2) [superposition 21,90]
90. mult(X4,mult(X3,X4)) = X3 [forward demodulation 75,27]
75. mult(inverse(X3),e) = mult(X4,mult(X3,X4)) [superposition 22,19]
27. mult(inverse(X2),e) = X2 [superposition 21,11]
22. mult(inverse(X4),mult(X4,X5)) = X5 [forward demodulation 17,10]
21. mult(X0,mult(X0,X1)) = X1 [forward demodulation 15,10]
19. e = mult(X0,mult(X1,mult(X0,X1))) [superposition 12,13]
17. mult(e,X5) = mult(inverse(X4),mult(X4,X5)) [superposition 12,11]
15. mult(e,X1) = mult(X0,mult(X0,X1)) [superposition 12,13]
14. mult(sK0,sK1) != mult(sK1,sK0) [cnf transformation 9]
13. e = mult(X0,X0) [cnf transformation 4]
12. mult(X0,mult(X1,X2)) = mult(mult(X0,X1),X2) [cnf transformation 3]
11. e = mult(inverse(X0),X0) [cnf transformation 2]
10. mult(e,X0) = X0 [cnf transformation 1]
9. mult(sK0,sK1) != mult(sK1,sK0) [skolemisation 7,8]
8. ?[X0,X1]: mult(X0,X1) != mult(X1,X0) <=> mult(sK0,sK1) != mult(sK1,sK0)
                                                    [choice axiom]
7. ?[X0,X1]: mult(X0,X1) != mult(X1,X0) [ennf transformation 6]
6. ~![X0,X1]: mult(X0,X1) = mult(X1,X0) [negated conjecture 5]
5. ![X0,X1]: mult(X0,X1) = mult(X1,X0) [input]
4. ![X0]: e = mult(X0,X0)[input]
3. ![X0,X1,X2]: mult(X0,mult(X1,X2)) = mult(mult(X0,X1),X2) [input]
2. ![X0]: e = mult(inverse(X0),X0) [input]
1. ![X0]: mult(e,X0) = X0 [input]
```

Proof by Vampire (Slightliy Modified)

```
Refutation found.
270. $false [trivial inequality removal 269]
269. mult(sk0,sk1) != mult (sk0,sk1) [superposition 14,125]
125. mult(X2,X3) = mult(X3,X2) [superposition 21,90]
90. mult(X4,mult(X3,X4)) = X3 [forward demodulation 75,27]
75. mult(inverse(X3),e) = mult(X4,mult(X3,X4)) [superposition 22,19]
27. mult(inverse(X2),e) = X2 [superposition 21,11]
22. mult(inverse(X4),mult(X4,X5)) = X5 [forward demodulation 17,10]
21. mult(X0,mult(X0,X1)) = X1 [forward demodulation 15,10]
19. e = mult(X0,mult(X1,mult(X0,X1))) [superposition 12,13]
17. mult(e,X5) = mult(inverse(X4),mult(X4,X5)) [superposition 12,11]
15. mult(e,X1) = mult(X0,mult(X0,X1)) [superposition 12,13]
14. mult(sK0,sK1) != mult(sK1,sK0) [cnf transformation 9]
13. e = mult(X0,X0) [cnf transformation 4]
12. mult(X0,mult(X1,X2)) = mult(mult(X0,X1),X2) [cnf transformation 3]
11. e = mult(inverse(X0),X0) [cnf transformation 2]
10. mult(e,X0) = X0 [cnf transformation 1]
9. mult(sK0,sK1) != mult(sK1,sK0) [skolemisation 7,8]
8. ?[X0,X1]: mult(X0,X1) != mult(X1,X0) <=> mult(sK0,sK1) != mult(sK1,sK0)
                                                        [choice axiom]
7. ?[X0,X1]: mult(X0,X1) != mult(X1,X0) [ennf transformation 6]
6. ~![X0,X1]: mult(X0,X1) = mult(X1,X0) [negated conjecture 5]
5. ![X0,X1]: mult(X0,X1) = mult(X1,X0) [input]
4. ![X0]: e = mult(X0,X0)[input]
3. ![X0,X1,X2]: mult(X0,mult(X1,X2)) = mult(mult(X0,X1),X2) [input]
2. ![X0]: e = mult(inverse(X0),X0) [input]
1. ![X0]: mult(e,X0) = X0 [input]
```

- Each inference derives a formula from zero or more other formulas;

Proof by Vampire (Slightliy Modified)

```
Refutation found.
270. $false [trivial inequality removal 269]
269. mult(sk0,sk1) != mult (sk0,sk1) [superposition 14,125]
125. mult(X2,X3) = mult(X3,X2) [superposition 21,90]
90. mult(X4,mult(X3,X4)) = X3 [forward demodulation 75,27]
75. mult(inverse(X3),e) = mult(X4,mult(X3,X4)) [superposition 22,19]
27. mult(inverse(X2),e) = X2 [superposition 21,11]
22. mult(inverse(X4),mult(X4,X5)) = X5 [forward demodulation 17,10]
21. mult(X0,mult(X0,X1)) = X1 [forward demodulation 15,10]
19. e = mult(X0,mult(X1,mult(X0,X1))) [superposition 12,13]
17. mult(e,X5) = mult(inverse(X4),mult(X4,X5)) [superposition 12,11]
15. mult(e,X1) = mult(X0,mult(X0,X1)) [superposition 12,13]
14. mult(sK0,sK1) != mult(sK1,sK0) [cnf transformation 9]
13. e = mult(X0,X0) [cnf transformation 4]
12. mult(X0,mult(X1,X2)) = mult(mult(X0,X1),X2) [cnf transformation 3]
11. e = mult(inverse(X0),X0) [cnf transformation 2]
10. mult(e,X0) = X0 [cnf transformation 1]
9. mult(sK0,sK1) != mult(sK1,sK0) [skolemisation 7,8]
8. ?[X0,X1]: mult(X0,X1) != mult(X1,X0) <=> mult(sK0,sK1) != mult(sK1,sK0)
                                                        [choice axiom]
7. ?[X0,X1]: mult(X0,X1) != mult(X1,X0) [ennf transformation 6]
6. ~![X0,X1]: mult(X0,X1) = mult(X1,X0) [negated conjecture 5]
5. ![X0,X1]: mult(X0,X1) = mult(X1,X0) [input]
4. ![X0]: e = mult(X0,X0)[input]
3. ![X0,X1,X2]: mult(X0,mult(X1,X2)) = mult(mult(X0,X1),X2) [input]
2. ![X0]: e = mult(inverse(X0),X0) [input]
1. ![X0]: mult(e,X0) = X0 [input]
```

- Each inference derives a formula from zero or more other formulas;
- Input, preprocessing, new symbols introduction, superposition calculus

Proof by Vampire (Slightliy Modified)

```
Refutation found.
270. $false [trivial inequality removal 269]
269. mult(sk0,sk1) != mult (sk0,sk1) [superposition 14,125]
125. mult(X2,X3) = mult(X3,X2) [superposition 21,90]
90. mult(X4,mult(X3,X4)) = X3 [forward demodulation 75,27]
75. mult(inverse(X3),e) = mult(X4,mult(X3,X4)) [superposition 22,19]
27. mult(inverse(X2),e) = X2 [superposition 21,11]
22. mult(inverse(X4),mult(X4,X5)) = X5 [forward demodulation 17,10]
21. mult(X0,mult(X0,X1)) = X1 [forward demodulation 15,10]
19. e = mult(X0,mult(X1,mult(X0,X1))) [superposition 12,13]
17. mult(e,X5) = mult(inverse(X4),mult(X4,X5)) [superposition 12,11]
15. mult(e,X1) = mult(X0,mult(X0,X1)) [superposition 12,13]
14. mult(sK0,sK1) != mult(sK1,sK0) [cnf transformation 9]
13. e = mult(X0,X0) [cnf transformation 4]
12. mult(X0,mult(X1,X2)) = mult(mult(X0,X1),X2) [cnf transformation 3]
11. e = mult(inverse(X0),X0) [cnf transformation 2]
10. mult(e,X0) = X0 [cnf transformation 1]
9. mult(sK0,sK1) != mult(sK1,sK0) [skolemisation 7,8]
8. ?[X0,X1]: mult(X0,X1) != mult(X1,X0) <=> mult(sK0,sK1) != mult(sK1,sK0)
                                                    [choice axiom]
7. ?[X0,X1]: mult(X0,X1) != mult(X1,X0) [ennf transformation 6]
6. ~![X0,X1]: mult(X0,X1) = mult(X1,X0) [negated conjecture 5]
5. ![X0,X1]: mult(X0,X1) = mult(X1,X0) [input]
4. ![X0]: e = mult(X0,X0)[input]
3. ![X0,X1,X2]: mult(X0,mult(X1,X2)) = mult(mult(X0,X1),X2) [input]
2. ![X0]: e = mult(inverse(X0),X0) [input]
1. ![X0]: mult(e,X0) = X0 [input]
```

- Each inference derives a formula from zero or more other formulas;
- Input, preprocessing, new symbols introduction, superposition calculus

Proof by Vampire (Slightliy Modified)

```
Refutation found.
270. $false [trivial inequality removal 269]
269. mult(sk0,sk1) != mult (sk0,sk1) [superposition 14,125]
125. mult(X2,X3) = mult(X3,X2) [superposition 21,90]
90. mult(X4,mult(X3,X4)) = X3 [forward demodulation 75,27]
75. mult(inverse(X3),e) = mult(X4,mult(X3,X4)) [superposition 22,19]
27. mult(inverse(X2),e) = X2 [superposition 21,11]
22. mult(inverse(X4),mult(X4,X5)) = X5 [forward demodulation 17,10]
21. mult(X0,mult(X0,X1)) = X1 [forward demodulation 15,10]
19. e = mult(X0,mult(X1,mult(X0,X1))) [superposition 12,13]
17. mult(e,X5) = mult(inverse(X4),mult(X4,X5)) [superposition 12,11]
15. mult(e,X1) = mult(X0,mult(X0,X1)) [superposition 12,13]
14. mult(sK0,sK1) != mult(sK1,sK0) [cnf transformation 9]
13. e = mult(X0,X0) [cnf transformation 4]
12. mult(X0,mult(X1,X2)) = mult(mult(X0,X1),X2) [cnf transformation 3]
11. e = mult(inverse(X0),X0) [cnf transformation 2]
10. mult(e,X0) = X0 [cnf transformation 1]
9. mult(sK0,sK1) != mult(sK1,sK0) [skolemisation 7,8]
8. ?[X0,X1]: mult(X0,X1) != mult(X1,X0) <=> mult(sK0,sK1) != mult(sK1,sK0)
                                                    [choice axiom]
7. ?[X0,X1]: mult(X0,X1) != mult(X1,X0) [ennf transformation 6]
6. ~![X0,X1]: mult(X0,X1) = mult(X1,X0) [negated conjecture 5]
5. ![X0,X1]: mult(X0,X1) = mult(X1,X0) [input]
4. ![X0]: e = mult(X0,X0)[input]
3. ![X0,X1,X2]: mult(X0,mult(X1,X2)) = mult(mult(X0,X1),X2) [input]
2. ![X0]: e = mult(inverse(X0),X0) [input]
1. ![X0]: mult(e,X0) = X0 [input]
```

- Each inference derives a formula from zero or more other formulas;
- Input, preprocessing, new symbols introduction, superposition calculus

Proof by Vampire (Slightliy Modified)

```
Refutation found.
270. $false [trivial inequality removal 269]
269. mult(sk0,sk1) != mult (sk0,sk1) [superposition 14,125]
125. mult(X2,X3) = mult(X3,X2) [superposition 21,90]
90. mult(X4,mult(X3,X4)) = X3 [forward demodulation 75,27]
75. mult(inverse(X3),e) = mult(X4,mult(X3,X4)) [superposition 22,19]
27. mult(inverse(X2),e) = X2 [superposition 21,11]
22. mult(inverse(X4),mult(X4,X5)) = X5 [forward demodulation 17,10]
21. mult(X0,mult(X0,X1)) = X1 [forward demodulation 15,10]
19. e = mult(X0,mult(X1,mult(X0,X1))) [superposition 12,13]
17. mult(e,X5) = mult(inverse(X4),mult(X4,X5)) [superposition 12,11]
15. mult(e,X1) = mult(X0,mult(X0,X1)) [superposition 12,13]
14. mult(sK0,sK1) != mult(sK1,sK0) [cnf transformation 9]
13. e = mult(X0,X0) [cnf transformation 4]
12. mult(X0,mult(X1,X2)) = mult(mult(X0,X1),X2) [cnf transformation 3]
11. e = mult(inverse(X0),X0) [cnf transformation 2]
10. mult(e,X0) = X0 [cnf transformation 1]
9. mult(sK0,sK1) != mult(sK1,sK0) [skolemisation 7,8]
8. ?[X0,X1]: mult(X0,X1) != mult(X1,X0) <=> mult(sK0,sK1) != mult(sK1,sK0)
                                                        [choice axiom]
7. ?[X0,X1]: mult(X0,X1) != mult(X1,X0) [ennf transformation 6]
6. ~![X0,X1]: mult(X0,X1) = mult(X1,X0) [negated conjecture 5]
5. ![X0,X1]: mult(X0,X1) = mult(X1,X0) [input]
4. ![X0]: e = mult(X0,X0)[input]
3. ![X0,X1,X2]: mult(X0,mult(X1,X2)) = mult(mult(X0,X1),X2) [input]
2. ![X0]: e = mult(inverse(X0),X0) [input]
1. ![X0]: mult(e,X0) = X0 [input]
```

- Each inference derives a formula from zero or more other formulas;
- Input, preprocessing, new symbols introduction, superposition calculus

Proof by Vampire (Slightliy Modified)

```
Refutation found.
270. $false [trivial inequality removal 269]
269. mult(sk0,sk1) != mult (sk0,sk1) [superposition 14,125]
125. mult(X2,X3) = mult(X3,X2) [superposition 21,90]
90. mult(X4,mult(X3,X4)) = X3 [forward demodulation 75,27]
75. mult(inverse(X3),e) = mult(X4,mult(X3,X4)) [superposition 22,19]
27. mult(inverse(X2),e) = X2 [superposition 21,11]
22. mult(inverse(X4),mult(X4,X5)) = X5 [forward demodulation 17,10]
21. mult(X0,mult(X0,X1)) = X1 [forward demodulation 15,10]
19. e = mult(X0,mult(X1,mult(X0,X1))) [superposition 12,13]
17. mult(e,X5) = mult(inverse(X4),mult(X4,X5)) [superposition 12,11]
15. mult(e,X1) = mult(X0,mult(X0,X1)) [superposition 12,13]
14. mult(sK0,sK1) != mult(sK1,sK0) [cnf transformation 9]
13. e = mult(X0,X0) [cnf transformation 4]
12. mult(X0,mult(X1,X2)) = mult(mult(X0,X1),X2) [cnf transformation 3]
11. e = mult(inverse(X0),X0) [cnf transformation 2]
10. mult(e,X0) = X0 [cnf transformation 1]
9. mult(sK0,sK1) != mult(sK1,sK0) [skolemisation 7,8]
8. ?[X0,X1]: mult(X0,X1) != mult(X1,X0) <=> mult(sK0,sK1) != mult(sK1,sK0)
                                                    [choice axiom]
7. ?[X0,X1]: mult(X0,X1) != mult(X1,X0) [ennf transformation 6]
6. ~![X0,X1]: mult(X0,X1) = mult(X1,X0) [negated conjecture 5]
5. ![X0,X1]: mult(X0,X1) = mult(X1,X0) [input]
4. ![X0]: e = mult(X0,X0)[input]
3. ![X0,X1,X2]: mult(X0,mult(X1,X2)) = mult(mult(X0,X1),X2) [input]
2. ![X0]: e = mult(inverse(X0),X0) [input]
1. ![X0]: mult(e,X0) = X0 [input]
```

- Each inference derives a formula from zero or more other formulas;
- Input, preprocessing, new symbols introduction, superposition calculus
- Proof by refutation, generating and simplifying inferences, unused formulas . . .

Proof by Vampire (Slightliy Modified)

```
Refutation found.
270. $false [trivial inequality removal 269]
269. mult(sk0,sk1) != mult (sk0,sk1) [superposition 14,125]
125. mult(X2,X3) = mult(X3,X2) [superposition 21,90]
90. mult(X4,mult(X3,X4)) = X3 [forward demodulation 75,27]
75. mult(inverse(X3),e) = mult(X4,mult(X3,X4)) [superposition 22,19]
27. mult(inverse(X2),e) = X2 [superposition 21,11]
22. mult(inverse(X4),mult(X4,X5)) = X5 [forward demodulation 17,10]
21. mult(X0,mult(X0,X1)) = X1 [forward demodulation 15,10]
19. e = mult(X0,mult(X1,mult(X0,X1))) [superposition 12,13]
17. mult(e,X5) = mult(inverse(X4),mult(X4,X5)) [superposition 12,11]
15. mult(e,X1) = mult(X0,mult(X0,X1)) [superposition 12,13]
14. mult(sK0,sK1) != mult(sK1,sK0) [cnf transformation 9]
13. e = mult(X0,X0) [cnf transformation 4]
12. mult(X0,mult(X1,X2)) = mult(mult(X0,X1),X2) [cnf transformation 3]
11. e = mult(inverse(X0),X0) [cnf transformation 2]
10. mult(e,X0) = X0 [cnf transformation 1]
9. mult(sK0,sK1) != mult(sK1,sK0) [skolemisation 7,8]
8. ?[X0,X1]: mult(X0,X1) != mult(X1,X0) <=> mult(sK0,sK1) != mult(sK1,sK0)
                                                    [choice axiom]
7. ?[X0,X1]: mult(X0,X1) != mult(X1,X0) [ennf transformation 6]
6. ~![X0,X1]: mult(X0,X1) = mult(X1,X0) [negated conjecture 5]
5. ![X0,X1]: mult(X0,X1) = mult(X1,X0) [input]
4. ![X0]: e = mult(X0,X0)[input]
3. ![X0,X1,X2]: mult(X0,mult(X1,X2)) = mult(mult(X0,X1),X2) [input]
2. ![X0]: e = mult(inverse(X0),X0) [input]
1. ![X0]: mult(e,X0) = X0 [input]
```

- Each inference derives a formula from zero or more other formulas;
- Input, preprocessing, new symbols introduction, superposition calculus
- Proof by refutation, generating and simplifying inferences, unused formulas . . .

Proof by Vampire (Slightliy Modified)

```
Refutation found.
270. $false [trivial inequality removal 269]
269. mult(sk0,sk1) != mult (sk0,sk1) [superposition 14,125]
125. mult(X2,X3) = mult(X3,X2) [superposition 21,90]
90. mult(X4,mult(X3,X4)) = X3 [forward demodulation 75,27]
75. mult(inverse(X3),e) = mult(X4,mult(X3,X4)) [superposition 22,19]
27. mult(inverse(X2),e) = X2 [superposition 21,11]
22. mult(inverse(X4),mult(X4,X5)) = X5 [forward demodulation 17,10]
21. mult(X0,mult(X0,X1)) = X1 [forward demodulation 15,10]
19. e = mult(X0,mult(X1,mult(X0,X1))) [superposition 12,13]
17. mult(e,X5) = mult(inverse(X4),mult(X4,X5)) [superposition 12,11]
15. mult(e,X1) = mult(X0,mult(X0,X1)) [superposition 12,13]
14. mult(sK0,sK1) != mult(sK1,sK0) [cnf transformation 9]
13. e = mult(X0,X0) [cnf transformation 4]
12. mult(X0,mult(X1,X2)) = mult(mult(X0,X1),X2) [cnf transformation 3]
11. e = mult(inverse(X0),X0) [cnf transformation 2]
10. mult(e,X0) = X0 [cnf transformation 1]
9. mult(sK0,sK1) != mult(sK1,sK0) [skolemisation 7,8]
8. ?[X0,X1]: mult(X0,X1) != mult(X1,X0) <=> mult(sK0,sK1) != mult(sK1,sK0)
                                                    [choice axiom]
7. ?[X0,X1]: mult(X0,X1) != mult(X1,X0) [ennf transformation 6]
6. ~![X0,X1]: mult(X0,X1) = mult(X1,X0) [negated conjecture 5]
5. ![X0,X1]: mult(X0,X1) = mult(X1,X0) [input]
4. ![X0]: e = mult(X0,X0)[input]
3. ![X0,X1,X2]: mult(X0,mult(X1,X2)) = mult(mult(X0,X1),X2) [input]
2. ![X0]: e = mult(inverse(X0),X0) [input]
1. ![X0]: mult(e,X0) = X0 [input]
```

- Each inference derives a formula from zero or more other formulas;
- Input, preprocessing, new symbols introduction, superposition calculus
- Proof by refutation, generating and simplifying inferences, unused formulas ...

Celsius to Fahrenheit (MSC023=2.p)

- tff form and type definition
- Typed quantification
- built-in theory symbols

```
tff(convertt,type,convert: ( $real * $real ) > $o ).

tff(convert,axiom,(
  ! [C: $real,F: $real] :
  ( $sum($product(1.8,C),32.0) = F => convert(C,F) )
)).

tff(fahrenheit_451_to_celsius,conjecture,(
    ? [C: $real] : convert(C,451.0) )).
```

Celsius to Fahrenheit (MSC023=2.p)

- tff form and type definition
- Typed quantification
- built-in theory symbols

```
tff(convertt,type,convert: ( $real * $real ) > $o ).

tff(convert,axiom,(
  ! [C: $real,F: $real] :
  ( $sum($product(1.8,C),32.0) = F => convert(C,F) )
)).

tff(fahrenheit_451_to_celsius,conjecture,(
    ? [C: $real] : convert(C,451.0) )).
```

Celsius to Fahrenheit (MSC023=2.p)

- tff form and type definition
- Typed quantification
- built-in theory symbols

```
tff(convertt,type,convert: ( $real * $real ) > $o ).

tff(convert,axiom,(
  ! [C: $real,F: $real] :
  ( $sum($product(1.8,C),32.0) = F => convert(C,F) )
)).

tff(fahrenheit_451_to_celsius,conjecture,(
    ? [C: $real] : convert(C,451.0) )).
```

Celsius to Fahrenheit (MSC023=2.p)

- tff form and type definition
- Typed quantification
- built-in theory symbols

```
tff(convertt,type,convert: ( $real * $real ) > $o ).

tff(convert,axiom,(
  ! [C: $real,F: $real] :
  ( $sum($product(1.8,C),32.0) = F => convert(C,F) )
)).

tff(fahrenheit_451_to_celsius,conjecture,(
    ? [C: $real] : convert(C,451.0) )).
```

Celsius to Fahrenheit (proof)

- Equality resolution
- Constrained resolution
- Evaluation
- We get a solution

```
./vampire MSC023=2.p -uwa all
```

```
23. convert(X0,X1) | $sum($product(1.8,X0),32.0) != X1
24. ~convert(X0,451.0)
26. convert(X0,$sum($product(1.8,X0),32.0)) [eq res 23]
27. convert(X0,$sum(32.0,$product(1.8,X0))) [demod 26,4]
64. $sum(32.0,$product(1.8,X0)) != 451.0 [con res 27,24]
76. $quotient($sum(451.0,-32.0),1.8) != X0 [evaluation 64]
77. $quotient(419.0,1.8) != X0 [evaluation 76]
78. 232.778 != X0 [evaluation 77]
86. $false [equality resolution 78]
```

Celsius to Fahrenheit (proof)

- Equality resolution
- Constrained resolution
- Evaluation
- We get a solution

```
./vampire MSC023=2.p -uwa all
```

```
23. convert(X0,X1) | $sum($product(1.8,X0),32.0) != X1
24. ~convert(X0,451.0)
26. convert(X0,$sum($product(1.8,X0),32.0)) [eq res 23]
27. convert(X0,$sum(32.0,$product(1.8,X0))) [demod 26,4]
64. $sum(32.0,$product(1.8,X0)) != 451.0 [con res 27,24]
76. $quotient($sum(451.0,-32.0),1.8) != X0 [evaluation 64]
77. $quotient(419.0,1.8) != X0 [evaluation 76]
78. 232.778 != X0 [evaluation 77]
86. $false [equality resolution 78]
```

Celsius to Fahrenheit (proof)

- Equality resolution
- Constrained resolution
- Evaluation
- We get a solution

```
./vampire MSC023=2.p -uwa all
```

```
23. convert(X0,X1) | $sum($product(1.8,X0),32.0) != X1
24. ~convert(X0,451.0)
26. convert(X0,$sum($product(1.8,X0),32.0)) [eq res 23]
27. convert(X0,$sum(32.0,$product(1.8,X0))) [demod 26,4]
64. $sum(32.0,$product(1.8,X0)) != 451.0 [con res 27,24]
76. $quotient($sum(451.0,-32.0),1.8) != X0 [evaluation 64]
77. $quotient(419.0,1.8) != X0 [evaluation 76]
78. 232.778 != X0 [evaluation 77]
86. $false [equality resolution 78]
```

Celsius to Fahrenheit (proof)

- Equality resolution
- Constrained resolution
- Evaluation
- We get a solution

```
./vampire MSC023=2.p -uwa all
```

```
23. convert(X0,X1) | $sum($product(1.8,X0),32.0) != X1
24. ~convert(X0,451.0)
26. convert(X0,$sum($product(1.8,X0),32.0)) [eq res 23]
27. convert(X0,$sum(32.0,$product(1.8,X0))) [demod 26,4]
64. $sum(32.0,$product(1.8,X0)) != 451.0 [con res 27,24]
76. $quotient($sum(451.0,-32.0),1.8) != X0 [evaluation 64]
77. $quotient(419.0,1.8) != X0 [evaluation 76]
78. 232.778 != X0 [evaluation 77]
86. $false [equality resolution 78]
```

Celsius to Fahrenheit (proof)

- Equality resolution
- Constrained resolution
- Evaluation
- We get a solution

```
./vampire MSC023=2.p -uwa all
```

```
23. convert(X0,X1) | $sum($product(1.8,X0),32.0) != X1
24. ~convert(X0,451.0)
26. convert(X0,$sum($product(1.8,X0),32.0)) [eq res 23]
27. convert(X0,$sum(32.0,$product(1.8,X0))) [demod 26,4]
64. $sum(32.0,$product(1.8,X0)) != 451.0 [con res 27,24]
76. $quotient($sum(451.0,-32.0),1.8) != X0 [evaluation 64]
77. $quotient(419.0,1.8) != X0 [evaluation 76]
78. 232.778 != X0 [evaluation 77]
86. $false [equality resolution 78]
```

Celsius to Fahrenheit (proof)

- Equality resolution
- Constrained resolution
- Evaluation
- We get a solution

```
./vampire MSC023=2.p -uwa all
```

```
23. convert(X0,X1) | $sum($product(1.8,X0),32.0) != X1
24. ~convert(X0,451.0)
26. convert(X0,$sum($product(1.8,X0),32.0)) [eq res 23]
27. convert(X0,$sum(32.0,$product(1.8,X0))) [demod 26,4]
64. $sum(32.0,$product(1.8,X0)) != 451.0 [con res 27,24]
76. $quotient($sum(451.0,-32.0),1.8) != X0 [evaluation 64]
77. $quotient(419.0,1.8) != X0 [evaluation 76]
78. 232.778 != X0 [evaluation 77]
86. $false [equality resolution 78]
```

Celsius to Fahrenheit (proof)

- Equality resolution
- Constrained resolution
- Evaluation
- We get a solution

```
./vampire MSC023=2.p -uwa all
```

```
23. convert(X0,X1) | $sum($product(1.8,X0),32.0) != X1
24. ~convert(X0,451.0)
26. convert(X0,$sum($product(1.8,X0),32.0)) [eq res 23]
27. convert(X0,$sum(32.0,$product(1.8,X0))) [demod 26,4]
64. $sum(32.0,$product(1.8,X0)) != 451.0 [con res 27,24]
76. $quotient($sum(451.0,-32.0),1.8) != X0 [evaluation 64]
77. $quotient(419.0,1.8) != X0 [evaluation 76]
78. 232.778 != X0 [evaluation 77]
86. $false [equality resolution 78]
```

The same in SMTLIB (MSC023=2.smt2)

- SMTLIB is a Lisp-like syntax
- We only support declare/assert commands
- Although support for incrementality is being added

```
(declare-fun convert (Real Real) Bool)
(assert (forall ((c Real) (f Real))
          (=> (= (+ (* 1.8 c) 32.0) f) (convert c f))))
(assert (not (exists ((c Real)) (convert c 451.0))))
```

```
./vampire -uwa all --input_syntax smtlib2 MSC023=2.smt2
```

First-Order Theorem Proving in Rigorous Systems Engineering

First-Order Theorem Proving

Laura Kovács and Andrei Voronkov

TU Wien and University of Manchester

Outline

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 $L_1 \vee \ldots \vee L_n$ 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

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:

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

- 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

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

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 PUZ031+1.p
```

Outline

Inference System

‘

- An inference has the form

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

  where $n \geq 0$ and $F_1, \ldots, F_n, G$ are formulas.
- The formula G is called the conclusion of the inference;
- The formulas $F_1, \ldots, F_n$ 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:

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

- 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:

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

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 $\square$ 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

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:

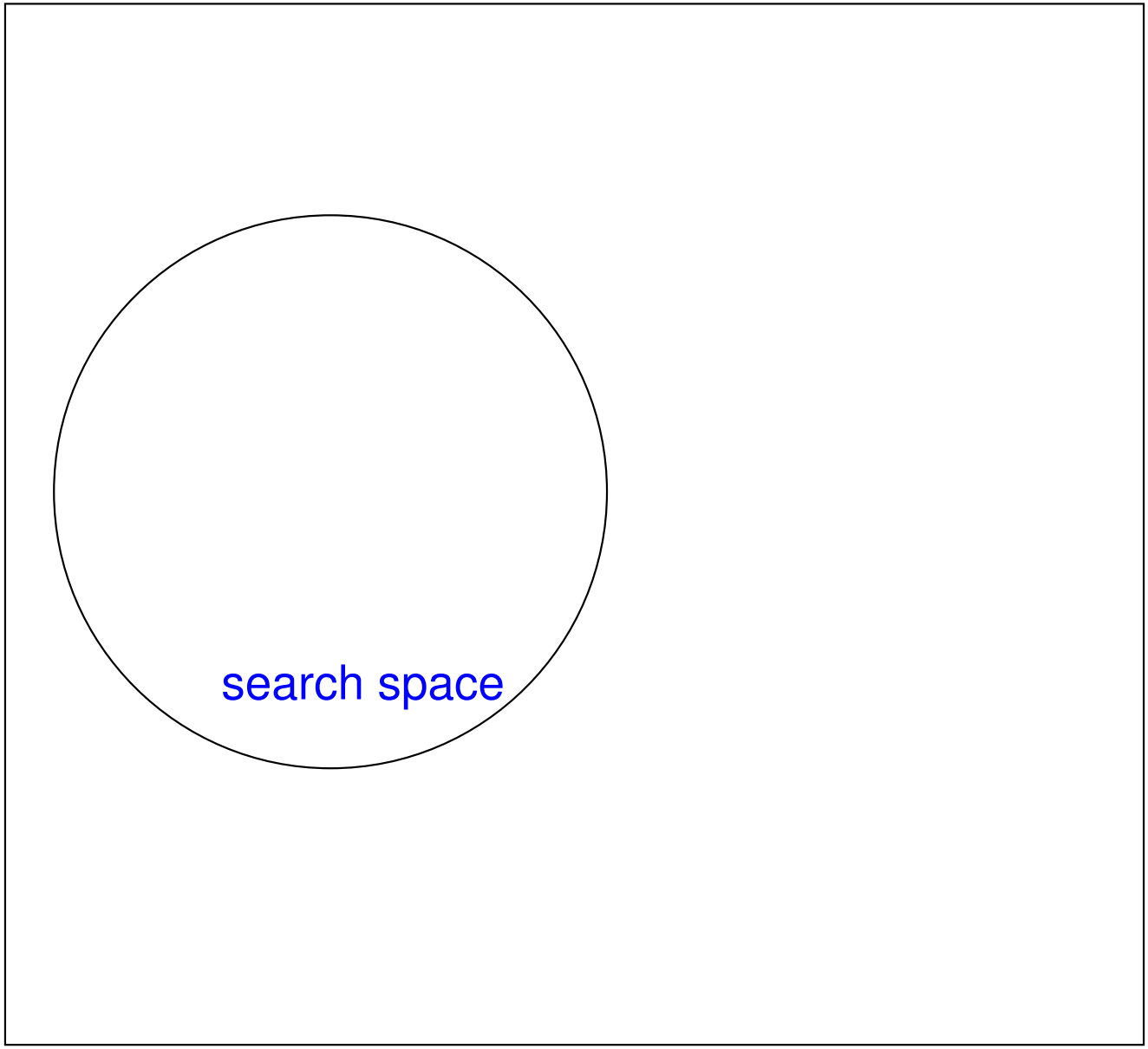
1. At some moment the empty clause $\square$ is generated, in this case the input set of clauses is unsatisfiable.
2. Saturation will terminate without ever generating $\square$, in this case the input set of clauses in satisfiable.
3. Saturation will run forever, but without generating $\square$. In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

Saturation Algorithm in Practice

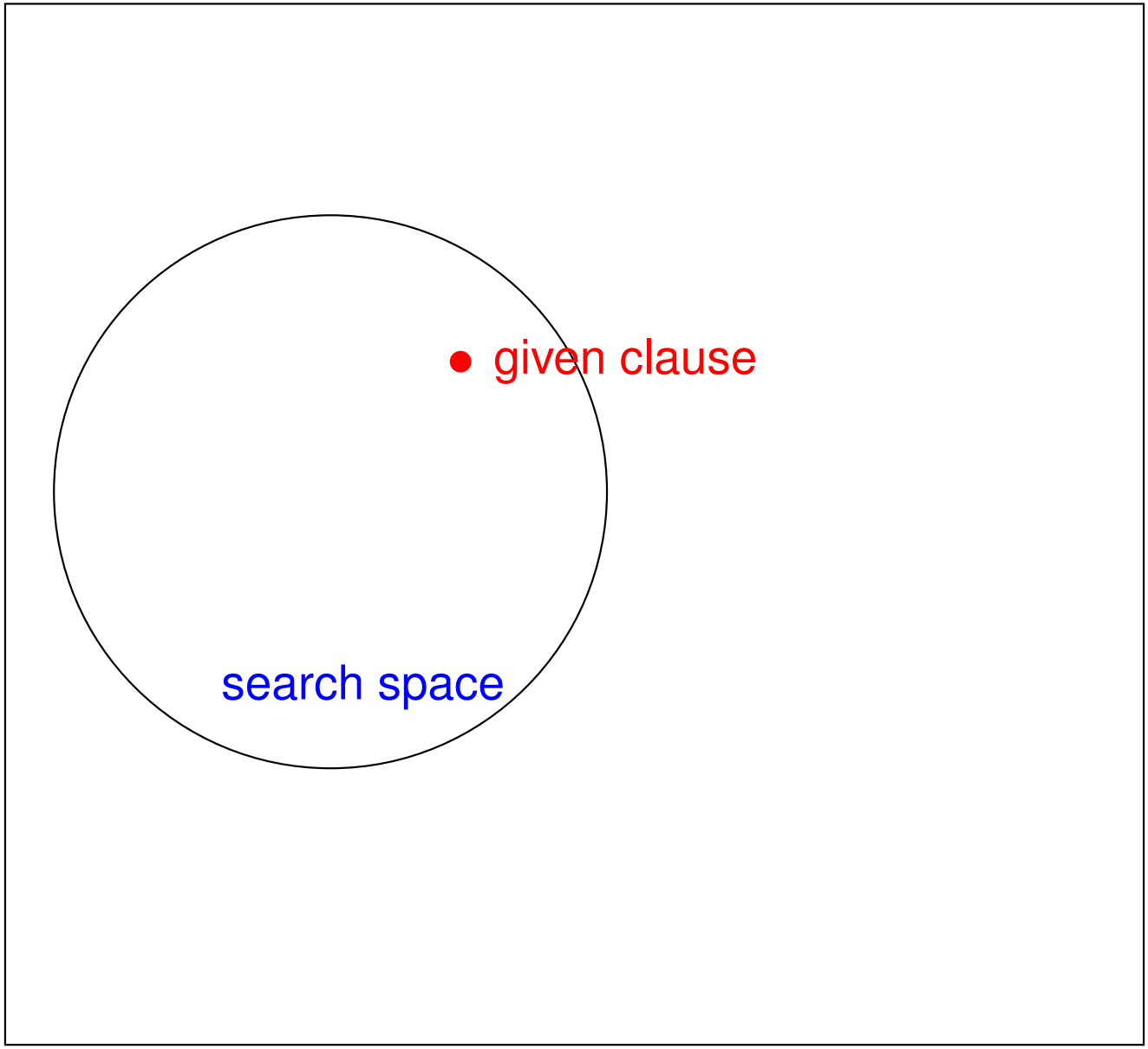
In practice there are three possible scenarios:

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.

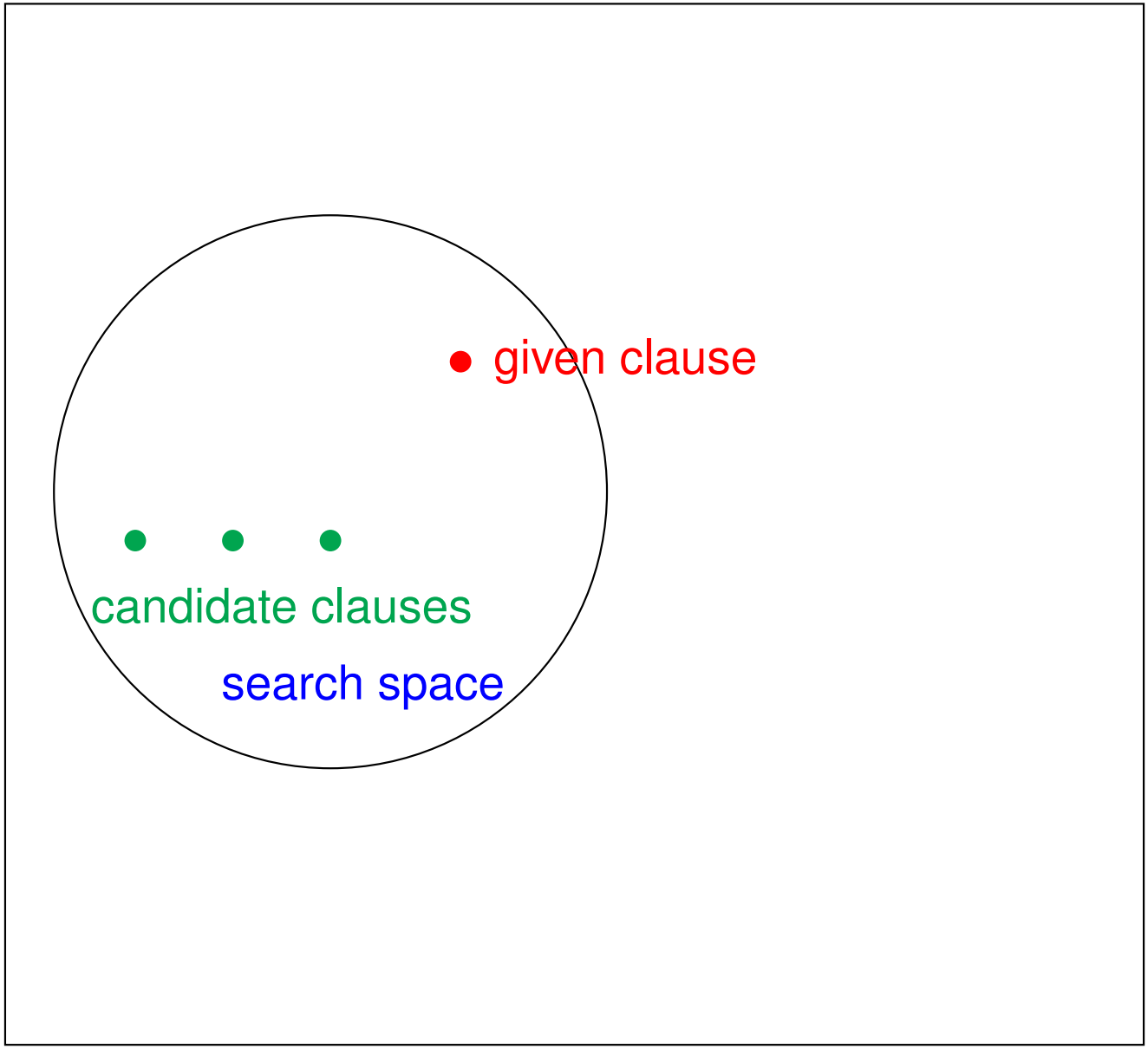
Inference Selection by Clause Selection



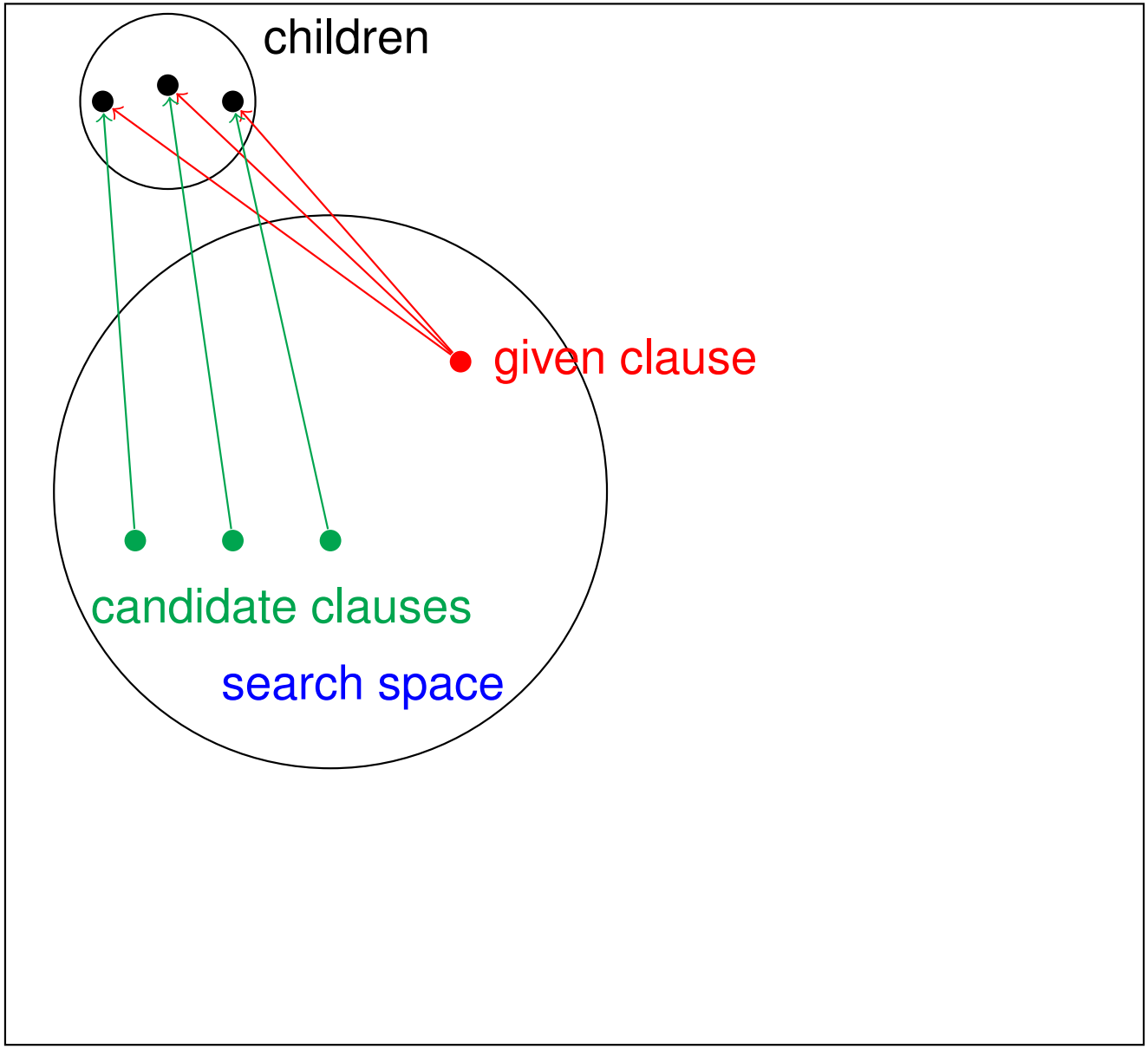
Inference Selection by Clause Selection



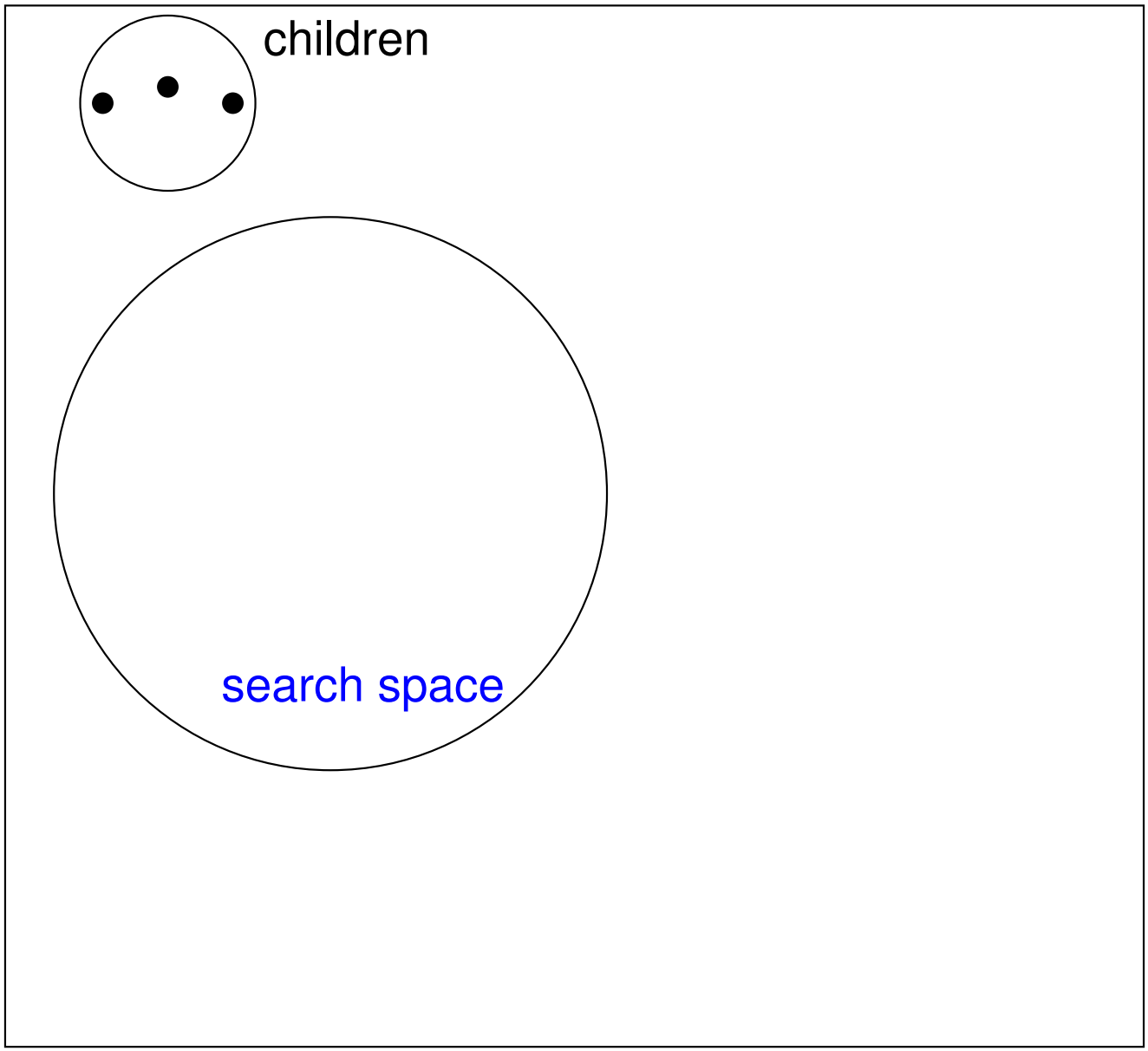
Inference Selection by Clause Selection



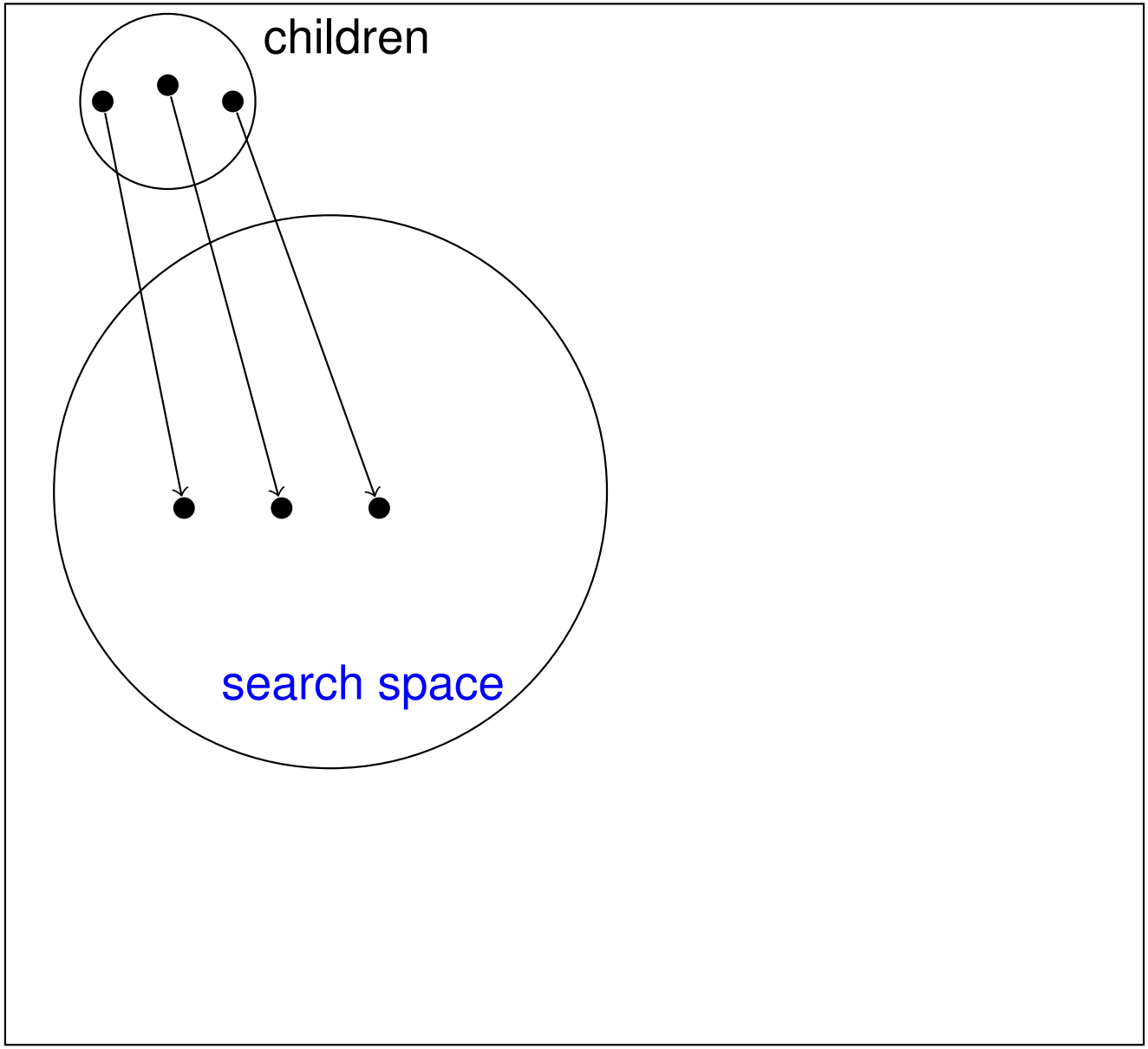
Inference Selection by Clause Selection



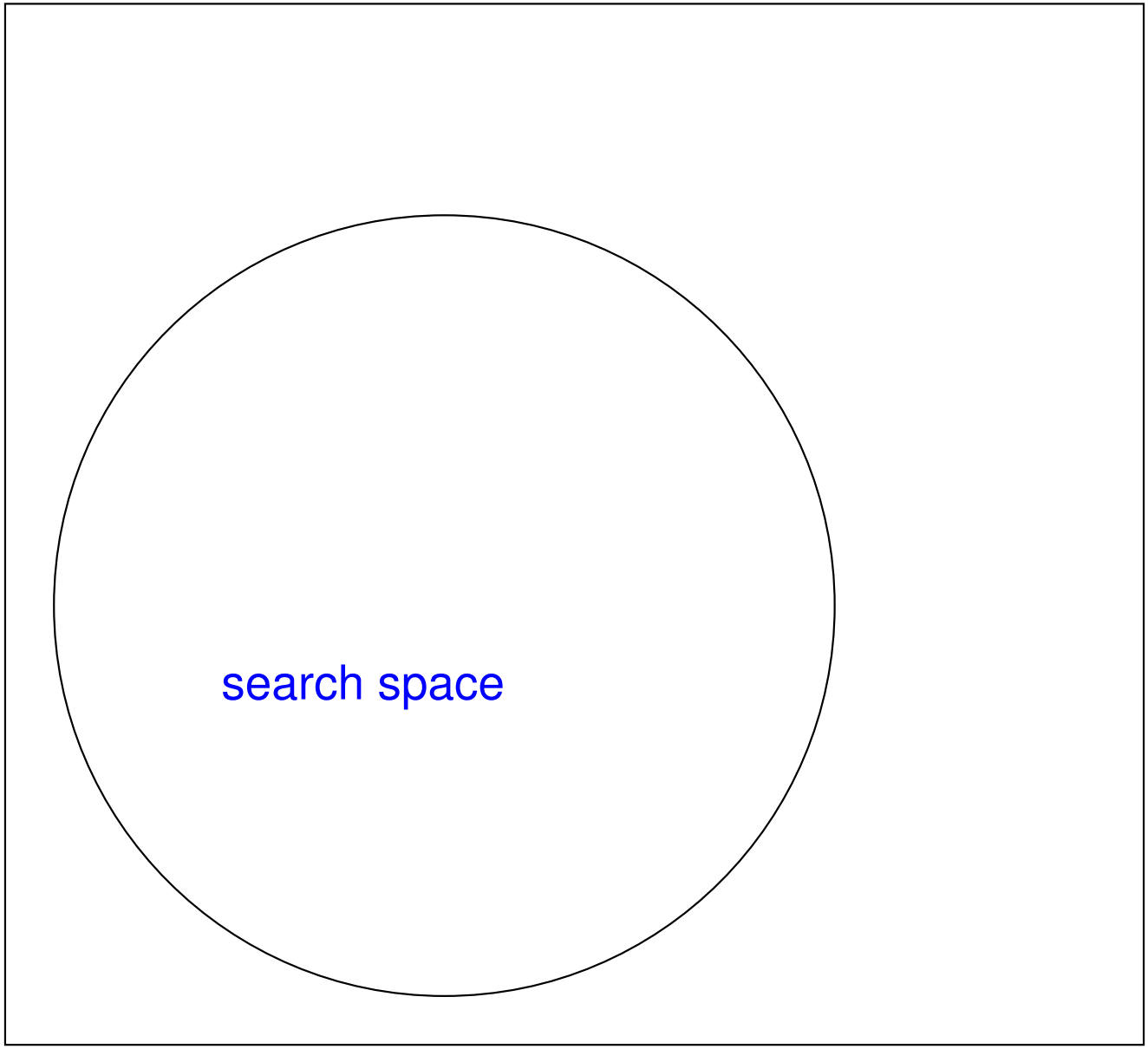
Inference Selection by Clause Selection



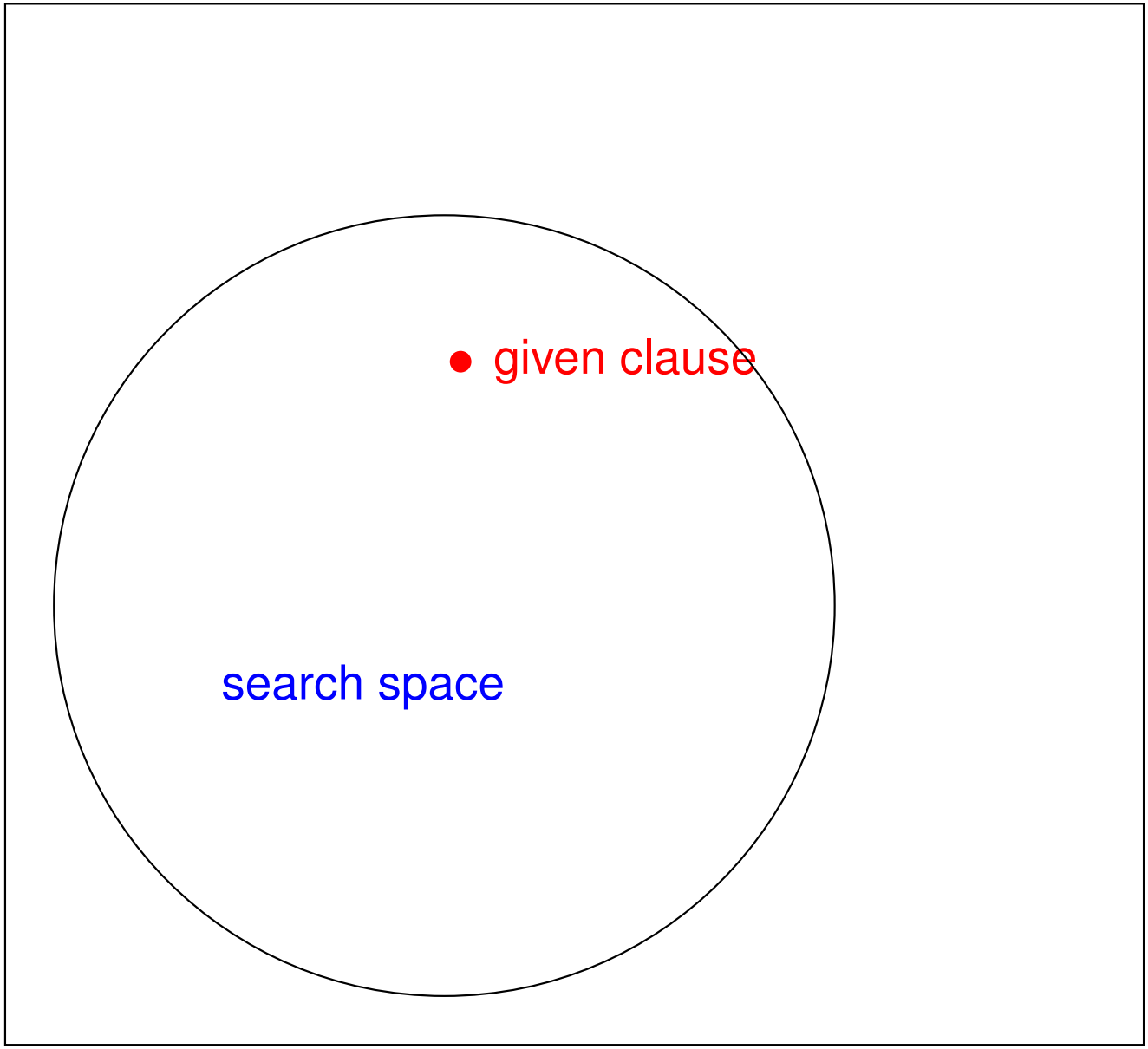
Inference Selection by Clause Selection



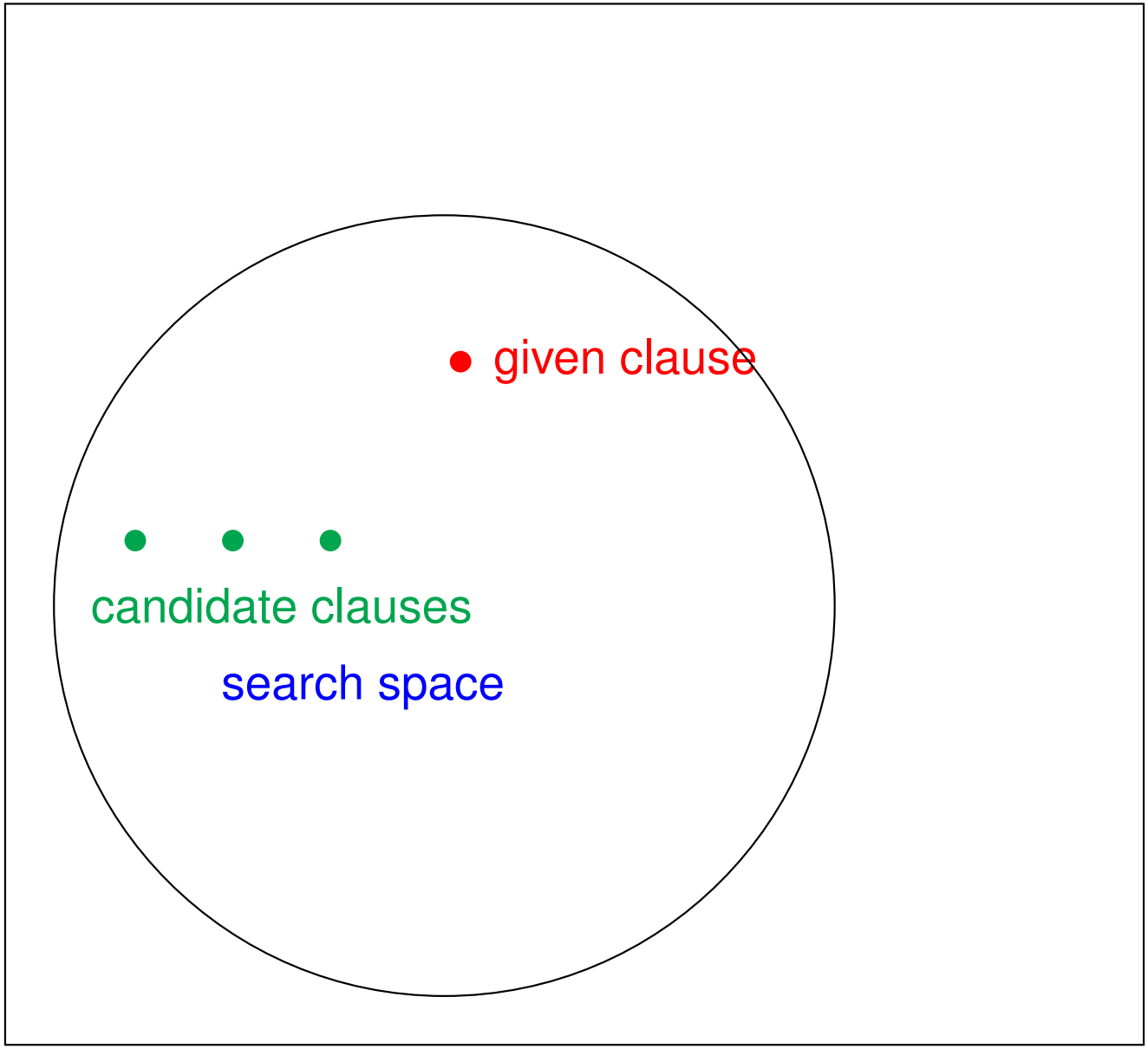
Inference Selection by Clause Selection



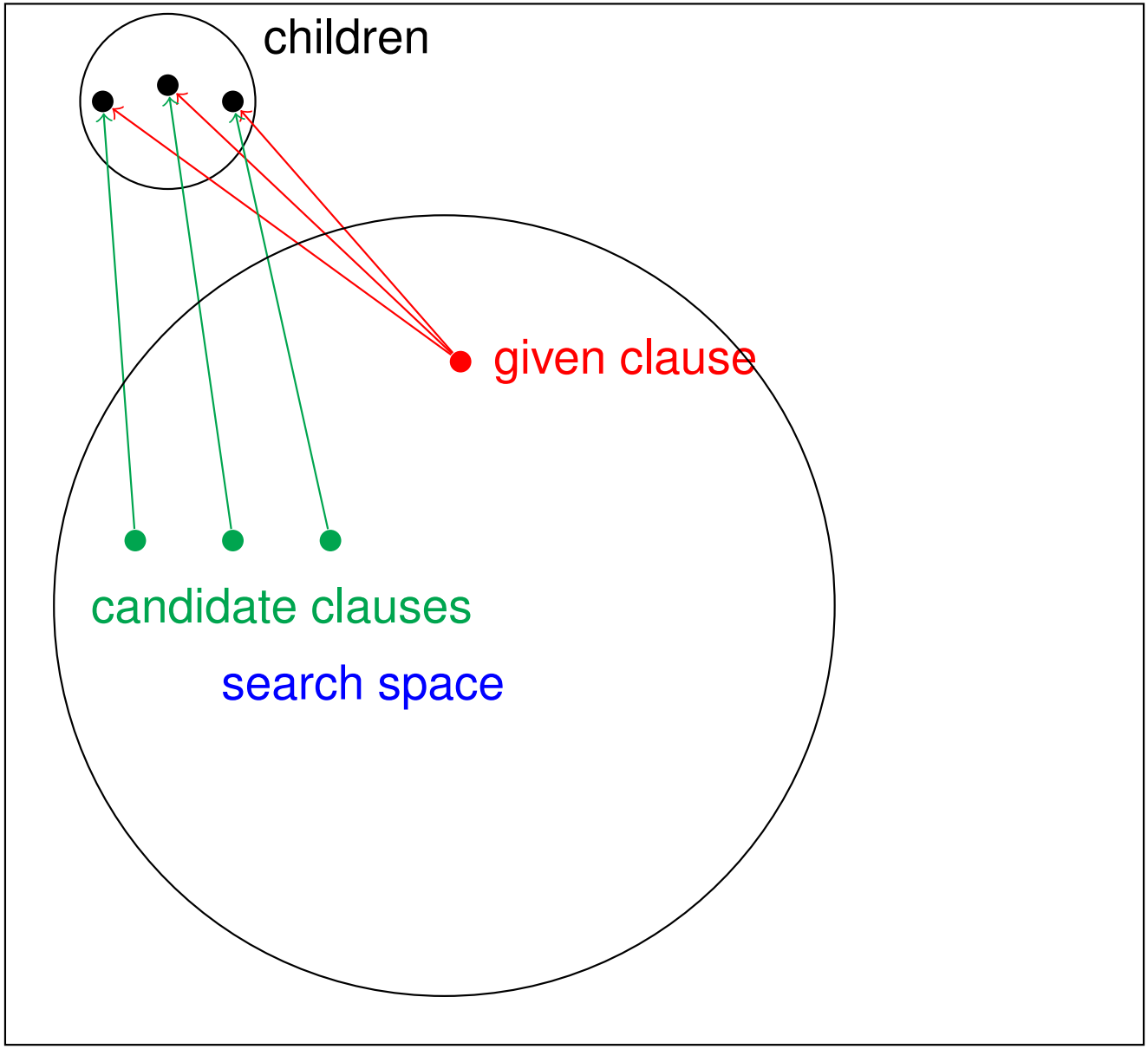
Inference Selection by Clause Selection



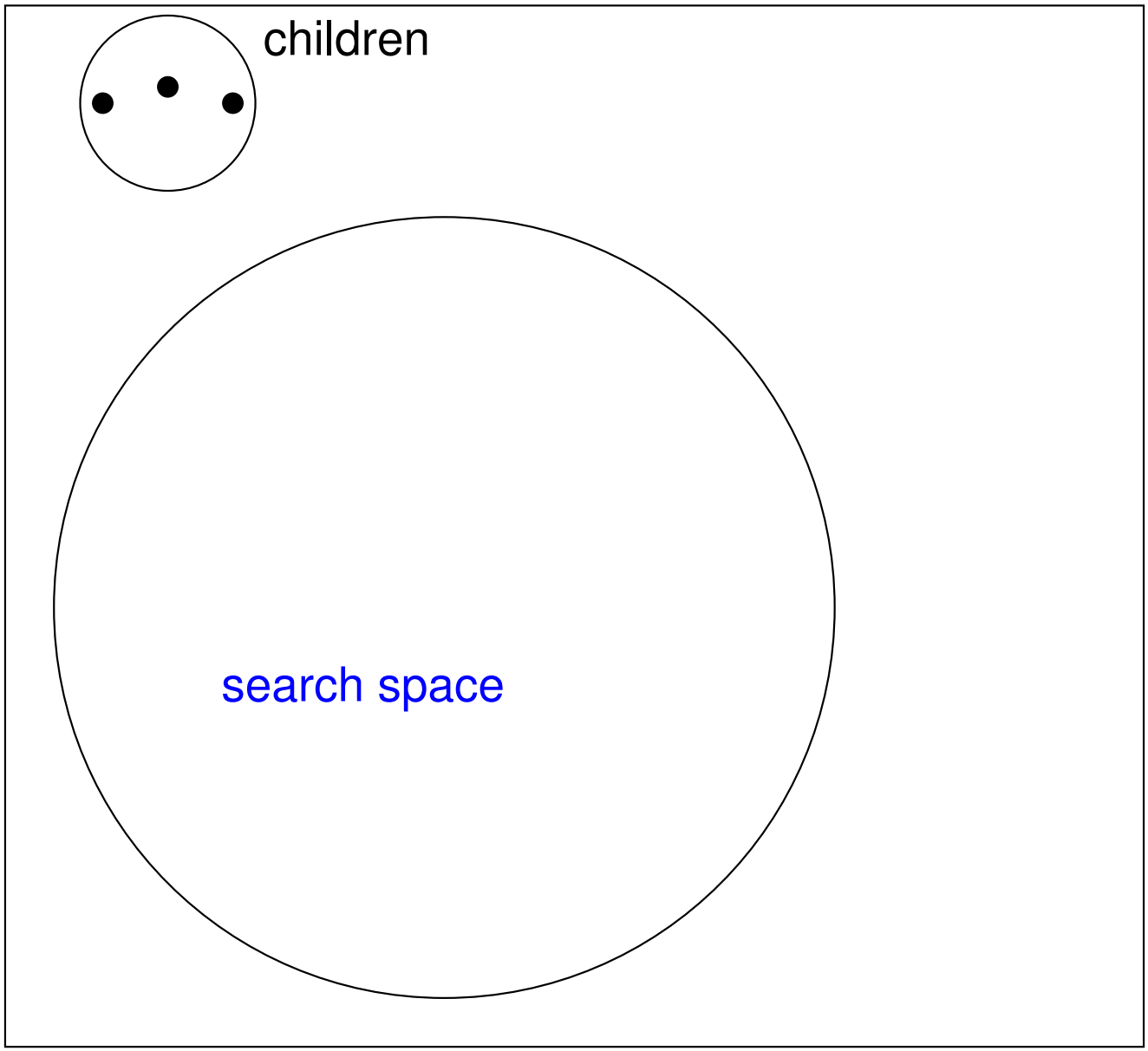
Inference Selection by Clause Selection



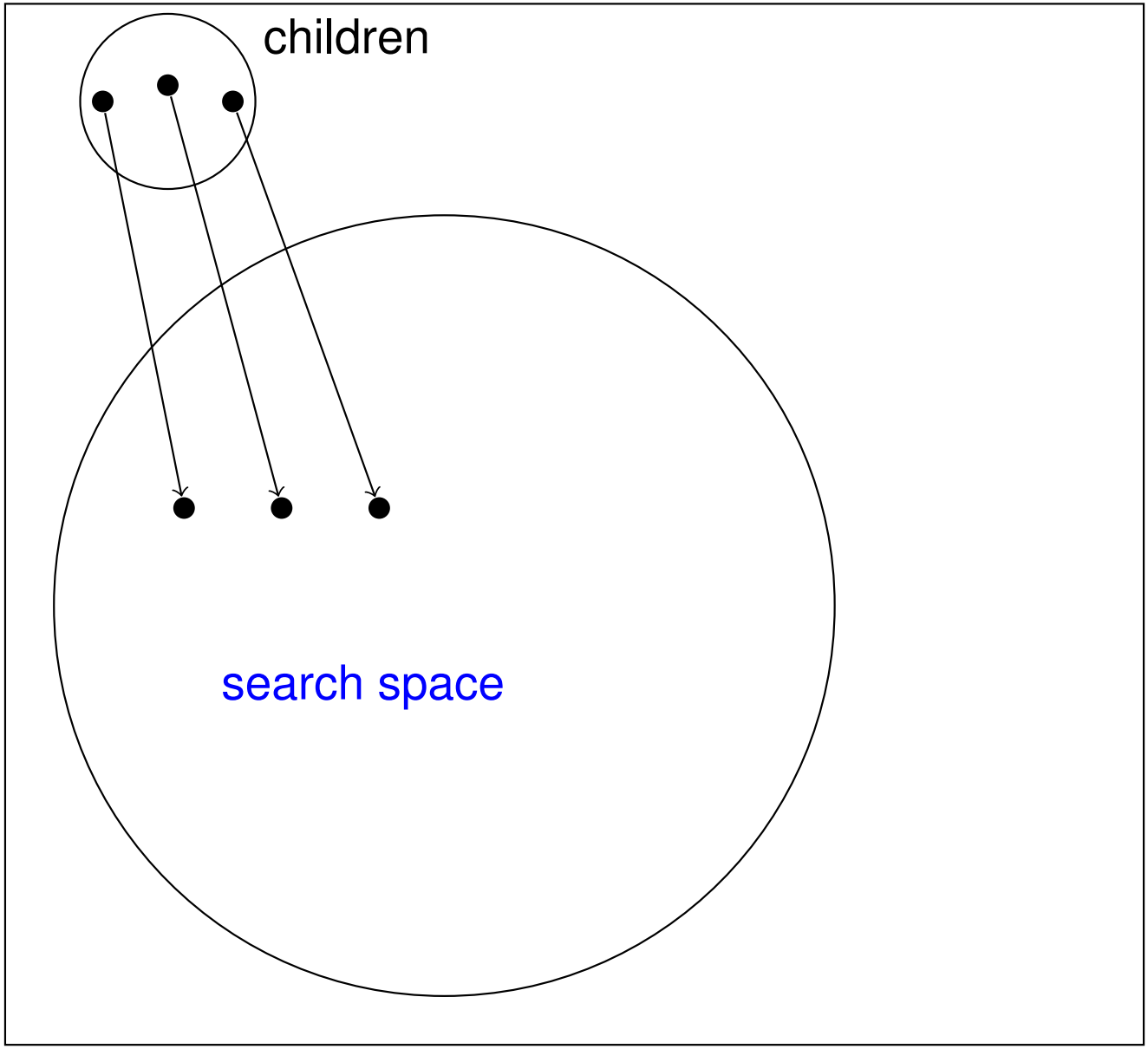
Inference Selection by Clause Selection



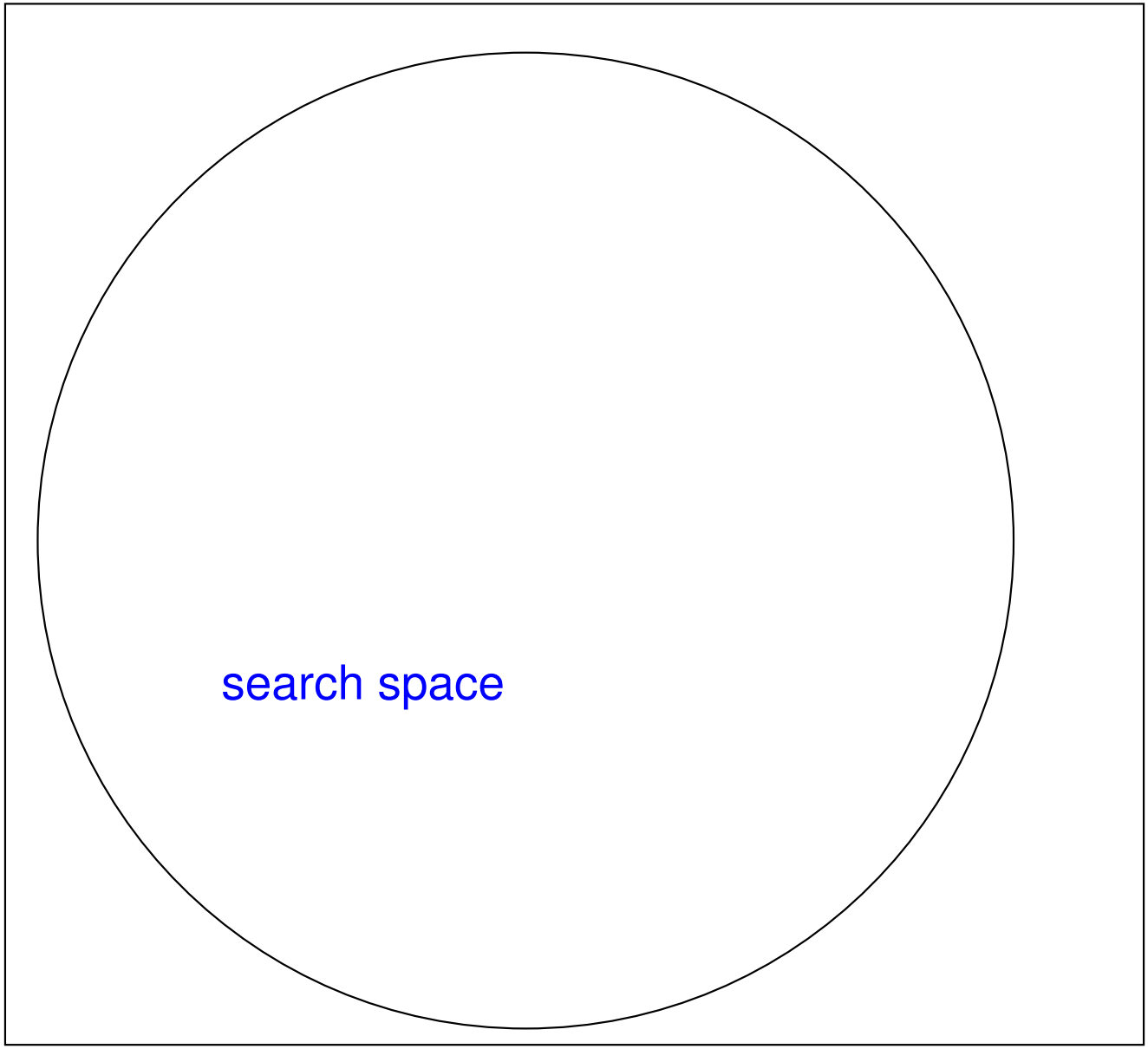
Inference Selection by Clause Selection



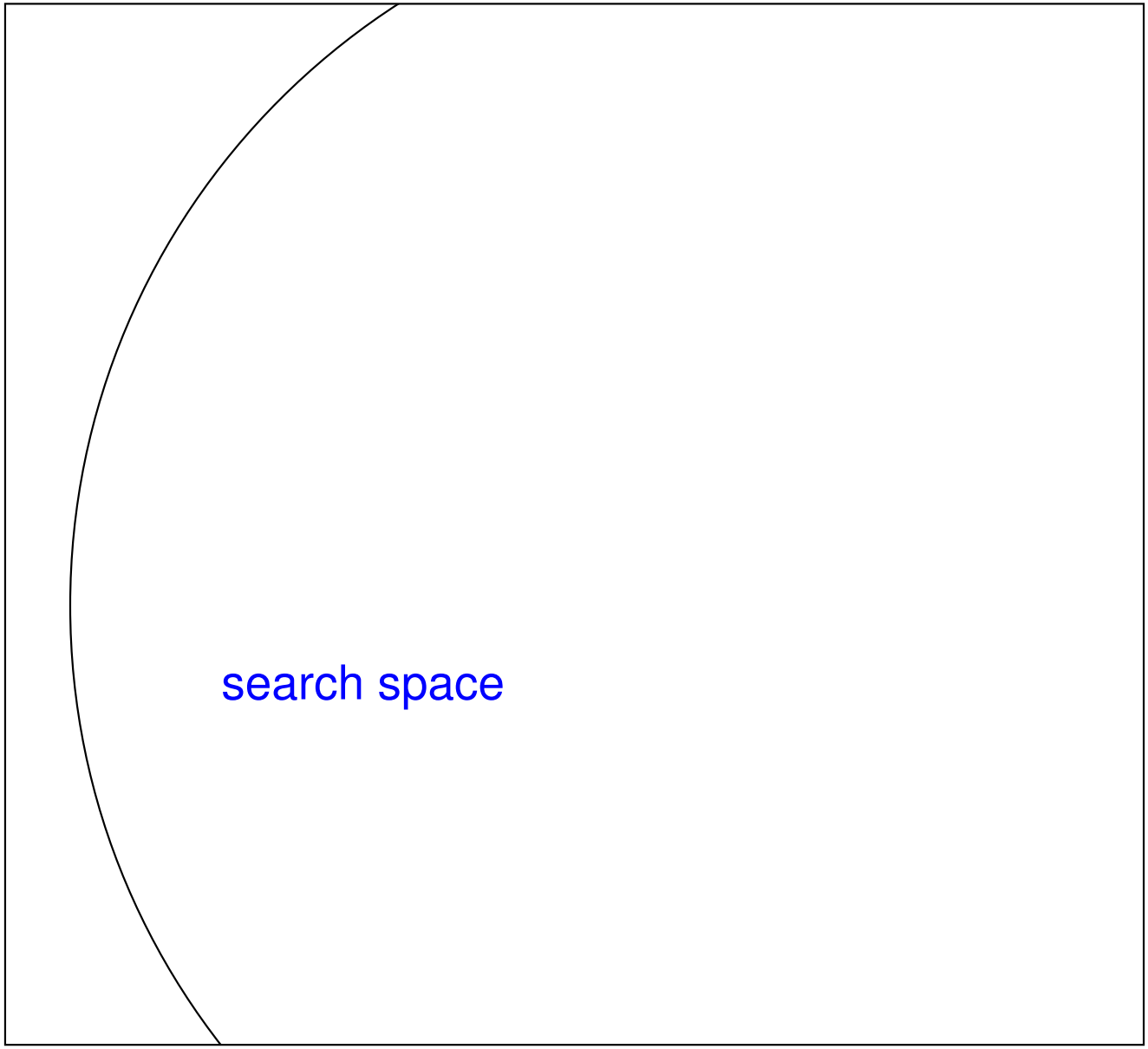
Inference Selection by Clause Selection



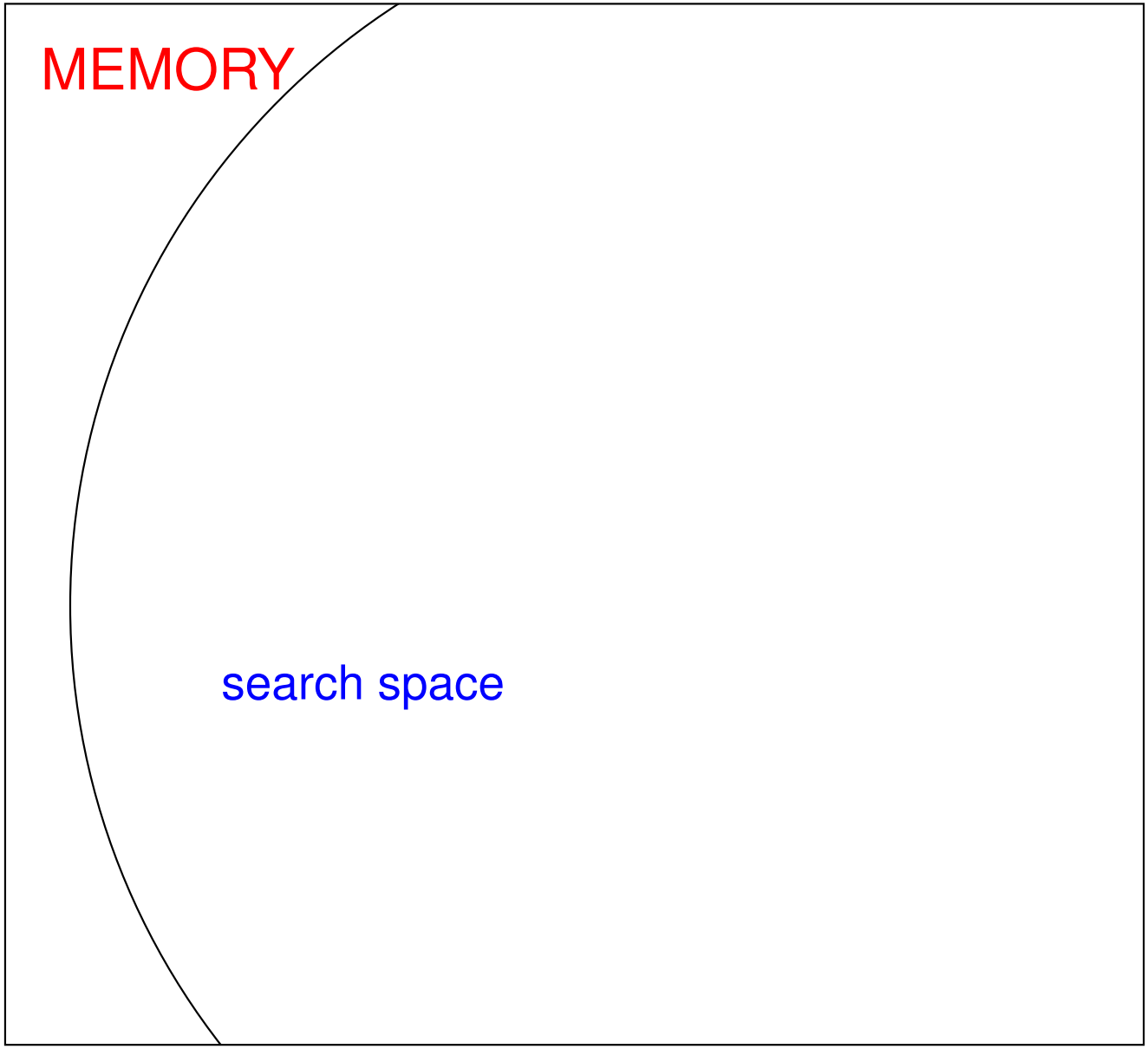
Inference Selection by Clause Selection



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

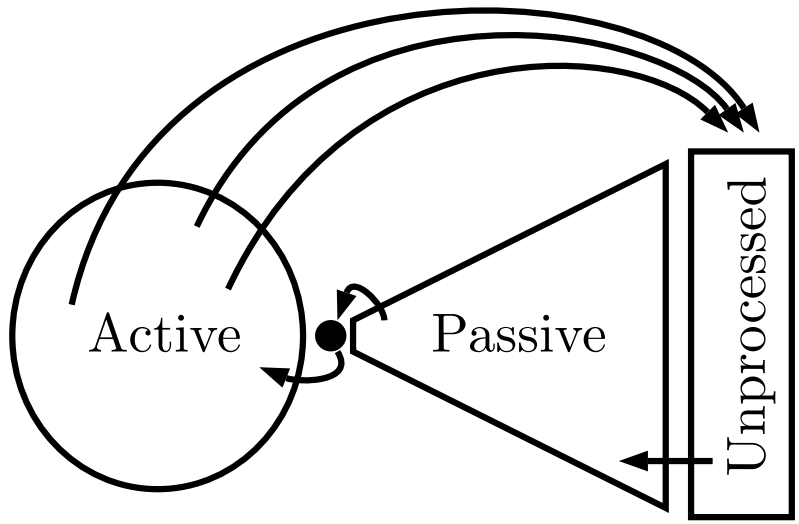
Even when we implement inference selection by clause selection, there are too many inferences, especially when the search space grows.

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

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 underlining them, 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 σ.
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 σ.
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:

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

where

1. θ 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 σ 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 σ 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 σ 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

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

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.

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 θ 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;
apply generating inferences lazily.

checking for simplifying inferences should pay off;
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**
* *current* $:=$ *select*(*passive*)**;** (* clause selection *)
move *current* from *passive* to *active*;
* *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.

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 -fsr off GRP140-1.p`

Vampire – 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

First-Order Theorem Proving in Rigorous Systems Engineering

Vampire in Practice

Laura Kovács and Andrei Voronkov

TU Wien and University of Manchester

Outline

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, ...
- ...

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 σ 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 σ 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 σ 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

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

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.

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 θ 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;
apply generating inferences lazily.

checking for simplifying inferences should pay off;
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**
* *current* $:=$ *select*(*passive*)**;** (* clause selection *)
move *current* from *passive* to *active*;
* *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.

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 -fsr off GRP140-1.p`

Vampire – 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

AVATAR

Laura Kovacs and Andrei Voronkov

AVATAR

Advanced
Vampire
Architecture for
Theories
And
Resolution

AVATAR

Definitions of *Avatar* (from various dictionaries):

Advanced
Vampire
Architecture for
Theories
And
Resolution

AVATAR

Advanced
Vampire
Architecture for
Theories
And
Resolution

Definitions of *Avatar* (from various dictionaries):

Science Fiction: a hybrid creature, composed of human and alien DNA and remotely controlled by the mind of a genetically matched human being

AVATAR

Advanced
Vampire
Architecture for
Theories
And
Resolution

Definitions of *Avatar* (from various dictionaries):

Science Fiction: a hybrid creature, composed of human and alien DNA and remotely controlled by the mind of a genetically matched human being

Hindu Mythology: the descent of a deity to the earth in an incarnate form or some manifest shape; the incarnation of a god

AVATAR

Advanced
Vampire
Architecture for
Theories
And
Resolution

Definitions of *Avatar* (from various dictionaries):

Science Fiction: a hybrid creature, composed of human and alien DNA and remotely controlled by the mind of a genetically matched human being

Hindu Mythology: the descent of a deity to the earth in an incarnate form or some manifest shape; the incarnation of a god

Automated Reasoning: a SAT solver embodied in a first-order theorem prover and in fact controlling its behaviour

Summary

- Original motivation: problems having clauses containing propositional variables and other clauses that can be split into components with disjoint sets of variables.
- Previously: splitting.
- New architecture: a superposition theorem prover tightly integrated with a SAT or an SMT solver.
- Emerging: reasoning with both quantifiers and theories.

Superposition Provers vs. SAT solvers (Saturation Algorithms vs. DPLL)

Resolution Prover:

$A \vee B$
$A \vee \neg B$
$\neg A \vee B$
$\neg A \vee \neg B$

SAT solver:

$A \vee B$
$A \vee \neg B$
$\neg A \vee B$
$\neg A \vee \neg B$

Superposition Provers vs. SAT solvers (Saturation Algorithms vs. DPLL)

Resolution Prover:

$A \vee B$
$A \vee \neg B$
$\neg A \vee B$
$\neg A \vee \neg B$

SAT solver:

$A \vee B$
$A \vee \neg B$
$\neg A \vee B$
$\neg A \vee \neg B$

Superposition Provers vs. SAT solvers (Saturation Algorithms vs. DPLL)

Resolution Prover:

$A \vee B$
$A \vee \neg B$
$\neg A \vee B$
$\neg A \vee \neg B$
A (resolution)

SAT solver:

$A \vee B$
$A \vee \neg B$
$\neg A \vee B$
$\neg A \vee \neg B$

Superposition Provers vs. SAT solvers (Saturation Algorithms vs. DPLL)

Resolution Prover:

$A \vee B$
$A \vee \neg B$
$\neg A \vee B$
$\neg A \vee \neg B$
A (resolution)

SAT solver:

$A \vee B$
$A \vee \neg B$
$\neg A \vee B$
$\neg A \vee \neg B$
A (decision)

Superposition Provers vs. SAT solvers (Saturation Algorithms vs. DPLL)

Resolution Prover:

$A \vee B$
$A \vee \neg B$
$\neg A \vee B$
$\neg A \vee \neg B$
A (resolution)

SAT solver:

$A \vee B$
$A \vee \neg B$
$\neg A \vee B$
$\neg A \vee \neg B$
A (decision)
$\square$ (unit propagation)

Superposition Provers vs. SAT solvers (Saturation Algorithms vs. DPLL)

Resolution Prover:

subsumption

$\neg A \vee B$
$\neg A \vee \neg B$
A (resolution)

SAT solver:

$A \vee B$
$A \vee \neg B$
$\neg A \vee B$
$\neg A \vee \neg B$
A (decision)
$\square$ (unit propagation)

Superposition Provers vs. SAT solvers (Saturation Algorithms vs. DPLL)

Resolution Prover:

$\neg A \vee B$
$\neg A \vee \neg B$
A (resolution)

SAT solver:

$A \vee B$
$A \vee \neg B$
$\neg A \vee B$
$\neg A \vee \neg B$

$\neg A$ (learned clause)

Superposition Provers vs. SAT solvers (Saturation Algorithms vs. DPLL)

Resolution Prover:

$\neg A \vee B$
$\neg A \vee \neg B$
A (resolution)

SAT solver:

$A \vee B$
$A \vee \neg B$
$\neg A \vee B$
$\neg A \vee \neg B$

$\neg A$ (learned clause)

Superposition Provers vs. SAT solvers (Saturation Algorithms vs. DPLL)

Resolution Prover:

$\neg A \vee B$
$\neg A \vee \neg B$
A (resolution)
B (resolution)

SAT solver:

$A \vee B$
$A \vee \neg B$
$\neg A \vee B$
$\neg A \vee \neg B$

$\neg A$ (learned clause)

Superposition Provers vs. SAT solvers (Saturation Algorithms vs. DPLL)

Resolution Prover:

$\neg A \vee B$
$\neg A \vee \neg B$
A (resolution)
B (resolution)

SAT solver:

$A \vee B$
$A \vee \neg B$
$\neg A \vee B$
$\neg A \vee \neg B$

$\neg A$ (learned clause)
$\square$ (unit propagation, UNSAT!)

Superposition Provers vs. SAT solvers (Saturation Algorithms vs. DPLL)

Resolution Prover:

subsumption

$\neg A \vee \neg B$

A (resolution)

B (resolution)

SAT solver:

$A \vee B$

$A \vee \neg B$

$\neg A \vee B$

$\neg A \vee \neg B$

$\neg A$ (learned clause)

Superposition Provers vs. SAT solvers (Saturation Algorithms vs. DPLL)

Resolution Prover:

$\neg A \vee \neg B$
A (resolution)
B (resolution)

SAT solver:

$A \vee B$
$A \vee \neg B$
$\neg A \vee B$
$\neg A \vee \neg B$

$\neg A$ (learned clause)

Superposition Provers vs. SAT solvers (Saturation Algorithms vs. DPLL)

Resolution Prover:

$\neg A \vee \neg B$
A (resolution)
B (resolution)
$\neg B$ (resolution)

SAT solver:

$A \vee B$
$A \vee \neg B$
$\neg A \vee B$
$\neg A \vee \neg B$

$\neg A$ (learned clause)

Superposition Provers vs. SAT solvers (Saturation Algorithms vs. DPLL)

Resolution Prover:

subsumption

A (resolution)

B (resolution)

$\neg B$ (resolution)

SAT solver:

$A \vee B$

$A \vee \neg B$

$\neg A \vee B$

$\neg A \vee \neg B$

$\neg A$ (learned clause)

Superposition Provers vs. SAT solvers (Saturation Algorithms vs. DPLL)

Resolution Prover:

A (resolution)
B (resolution)
$\neg B$ (resolution)

SAT solver:

$A \vee B$
$A \vee \neg B$
$\neg A \vee B$
$\neg A \vee \neg B$

$\neg A$ (learned clause)

Superposition Provers vs. SAT solvers (Saturation Algorithms vs. DPLL)

Resolution Prover:

A (resolution)
B (resolution)
$\neg B$ (resolution)
$\square$ (resolution)

SAT solver:

$A \vee B$
$A \vee \neg B$
$\neg A \vee B$
$\neg A \vee \neg B$

$\neg A$ (learned clause)

Search Space in Saturation Algorithms

Illustrated using bacteria.

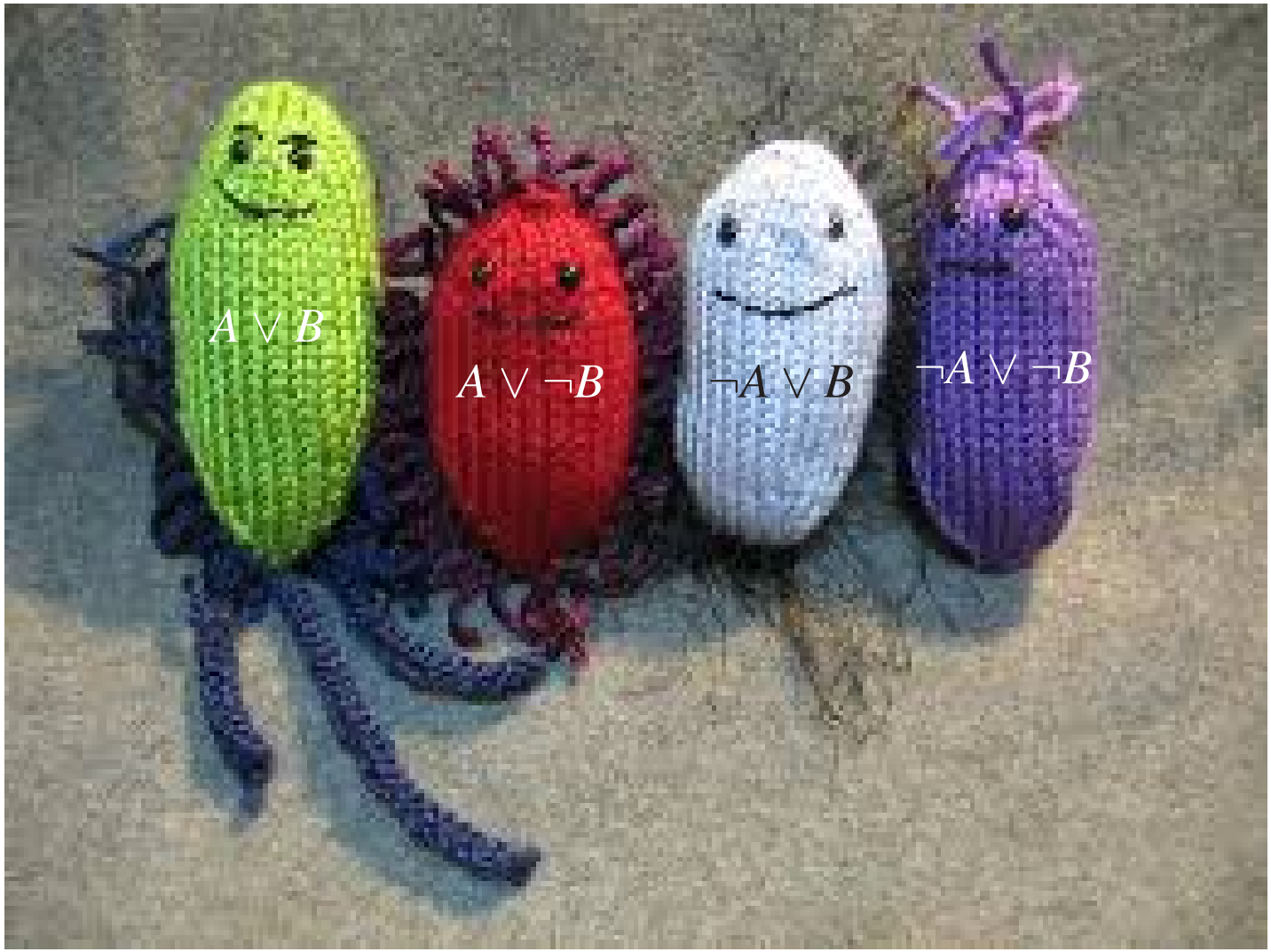
Search Space in Saturation Algorithms

Illustrated using bacteria. In the beginning . . .

precisionnutrition.com

Search Space in Saturation Algorithms

Illustrated using bacteria. In the beginning . . .



precisionnutrition.com

Search Space in Saturation Algorithms

After a few steps . . .

urbanext.illinois.edu

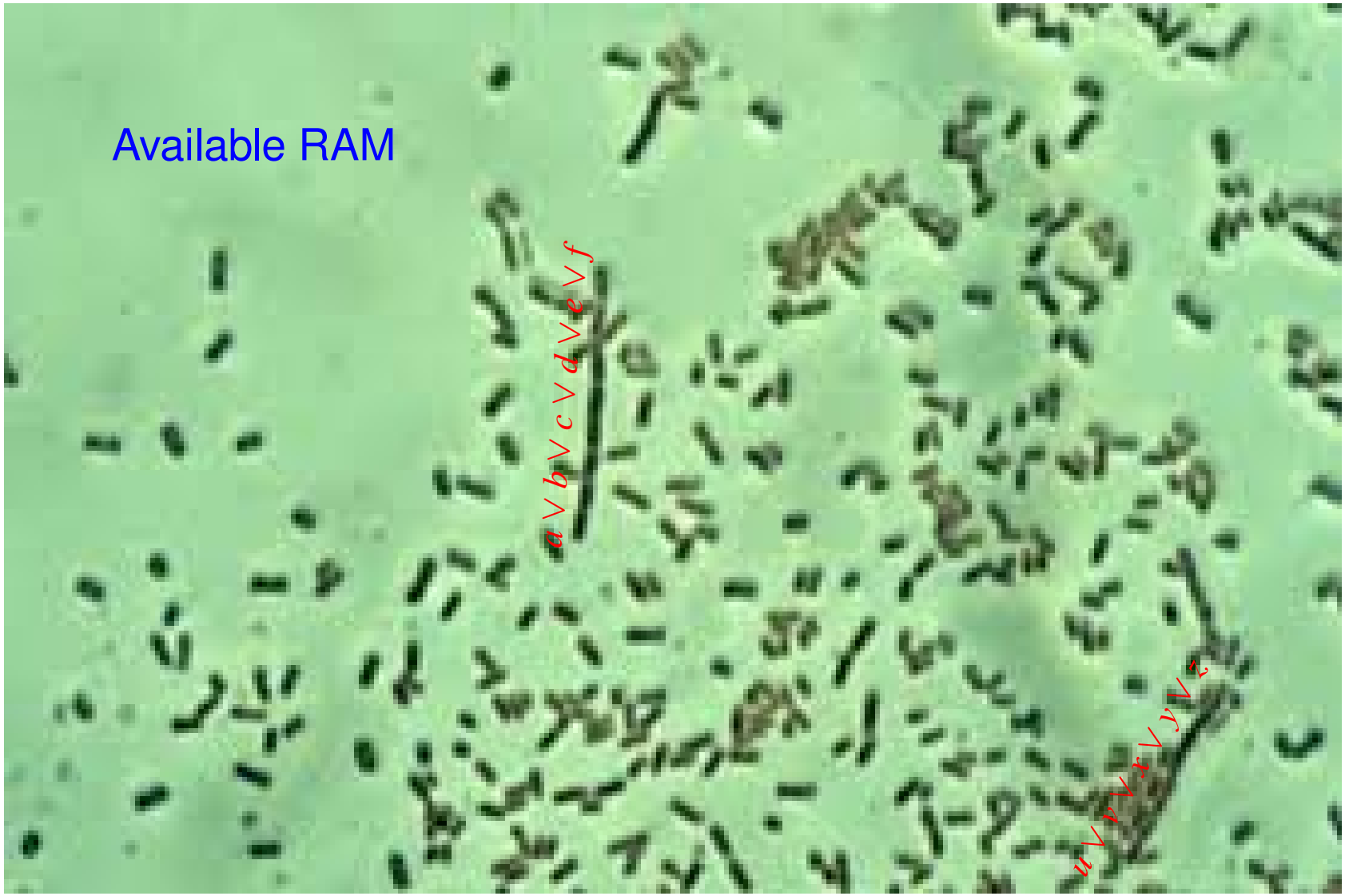
Search Space in Saturation Algorithms

After a few steps . . . and notice long clauses

urbanext.illinois.edu

Search Space in Saturation Algorithms

After a few steps . . . and notice long clauses



urbanext.illinois.edu

Search Space in Saturation Algorithms

After a few more steps . . .

Search Space in Saturation Algorithms

After a few more steps . . . Out of memory and notice the CPU temperature . . .

Reality of Superposition Theorem Proving

- Growing search spaces.
- Repeated applications of algorithms whose complexity depends on clause sizes: resolution, superposition, demodulation, KBO comparison, subsumption.
- Long clauses are a problem (produce even longer clauses, (subsumption is NP-complete).

Long Clauses

Example: resolving

$p(x,f(y)) \vee p(f(x),y) \vee p(g(x,z),f(f(y))) \vee \neg p(g(z,z),g(y,f(x))) \vee$
$p(f(a,x),g(z,g(y,z))) \vee \neg p(x,y) \vee p(f(y),z)$

against

$\neg p(f(w),v) \vee p(f(v),w) \vee p(g(v,u),f(f(w))) \vee \neg p(g(u,u),g(w,f(v))) \vee$
$p(f(a,v),g(u,g(w,u))) \vee \neg p(v,w) \vee p(f(w),u)$

Long Clauses

Example: resolving

$p(x,f(y)) \vee p(f(x),y) \vee p(g(x,z),f(f(y))) \vee \neg p(g(z,z),g(y,f(x))) \vee$
$p(f(a,x),g(z,g(y,z))) \vee \neg p(x,y) \vee p(f(y),z)$

against

$\neg p(f(w),v) \vee p(f(v),w) \vee p(g(v,u),f(f(w))) \vee \neg p(g(u,u),g(w,f(v))) \vee$
$p(f(a,v),g(u,g(w,u))) \vee \neg p(v,w) \vee p(f(w),u)$

gives

$p(f(f(w)),y) \vee p(g(f(w),z),f(f(y))) \vee \neg p(g(z,z),g(y,f(f(w)))) \vee$
$p(f(a,f(w)),g(z,g(y,z))) \vee \neg p(f(w),y) \vee p(f(y),z) \vee p(f(f(y)),w) \vee$
$p(g(f(y),u),f(f(w))) \vee \neg p(g(u,u),g(w,f(f(y)))) \vee$
$p(f(a,f(y)),g(u,g(w,u))) \vee \neg p(f(y),w) \vee p(f(w),u).$

Long Clauses

Does

$p(f(f(w)),y) \vee p(g(f(w),z),f(f(y))) \vee \neg p(g(z,z),g(y,f(f(w)))) \vee$
$p(f(a,f(w)),g(z,g(y,z))) \vee \neg p(f(w),y) \vee p(f(y),z) \vee p(f(f(y)),w) \vee$
$p(g(f(y),u),f(f(w))) \vee \neg p(g(u,u),g(w,f(f(y)))) \vee$
$p(f(a,f(y)),g(u,g(w,u))) \vee \neg p(f(y),w) \vee p(f(w),u)$

subsume

$p(g(f(y),u),f(f(g(x,y)))) \vee p(f(f(g(x,y))),y) \vee p(f(y),z) \vee$
$p(f(f(y)),w) \vee p(g(f(g(x,y)),z),f(f(y))) \vee p(f(g(x,y)),u) \vee$
$\neg p(g(z,z),g(y,f(f(g(x,y))))) \vee p(f(a,f(g(x,y))),g(z,g(y,z))) \vee$
$\neg p(f(g(x,y)),y) \vee p(f(y),z) \vee p(f(f(y)),g(x,y)) \vee$
$\neg p(g(u,u),g(g(x,y),f(f(y)))) \vee p(f(a,f(y)),g(u,g(g(x,y),u))) \vee$
$\neg p(f(y),g(x,y))$

???

Basis for DPLL

Consider a propositional set of clauses $S \cup \{C_1 \vee \ldots \vee C_n\}$.

$S \cup \{C_1 \vee \ldots \vee C_n\}$ is unsatisfiable if and only if each of the sets

$$S \cup \{C_1\}$$
$$\ldots$$
$$S \cup \{C_n\}$$

is unsatisfiable too.

Basis for DPLL

Consider a propositional set of clauses $S \cup \{C_1 \vee \ldots \vee C_n\}$.

$S \cup \{C_1 \vee \ldots \vee C_n\}$ is unsatisfiable if and only if each of the sets

$$S \cup \{C_1\}$$
$$\ldots$$
$$S \cup \{C_n\}$$

is unsatisfiable too.

Cannot be used in first-order logic:

- $\{p(x) \vee q(x), \neg p(a), \neg q(b)\}$ is satisfiable
- Both $\{p(x), \neg p(a), \neg q(b)\}$ and $\{q(x), \neg p(a), \neg q(b)\}$ are unsatisfiable.

Basis for DPLL

Consider a propositional set of clauses $S \cup \{C_1 \vee \ldots \vee C_n\}$.

$S \cup \{C_1 \vee \ldots \vee C_n\}$ is unsatisfiable if and only if each of the sets

$$S \cup \{C_1\}$$
$$\ldots$$
$$S \cup \{C_n\}$$

is unsatisfiable too.

Cannot be used in first-order logic:

- $\{p(x) \vee q(x), \neg p(a), \neg q(b)\}$ is satisfiable
- Both $\{p(x), \neg p(a), \neg q(b)\}$ and $\{q(x), \neg p(a), \neg q(b)\}$ are unsatisfiable.

Can be used when $C_1, \ldots, C_n$ have pairwise disjoint sets of variables.
Problem: rigid variables.

Attempts to make a First-Order Analogue of DPLL

Some of the recent work on first-order DPLL and instance-based calculi:

- Peter Baumgartner. FDPLL - A First Order Davis-Putnam-Longeman-Loveland Procedure. CADE 2000
- Reinhold Letz, Gernot Stenz. Automated Theorem Proving Proof and Model Generation with Disconnection Tableaux. LPAR 2001
- Peter Baumgartner, Cesare Tinelli. The Model Evolution Calculus. CADE 2003
- Harald Ganzinger, Konstantin Korovin. New Directions in Instantiation-Based Theorem Proving. LICS 2003

Attempts to make a First-Order Analogue of DPLL

Some of the recent work on first-order DPLL and instance-based calculi:

- Peter Baumgartner. FDPLL - A First Order Davis-Putnam-Longeman-Loveland Procedure. CADE 2000
- Reinhold Letz, Gernot Stenz. Automated Theorem Proving Proof and Model Generation with Disconnection Tableaux. LPAR 2001
- Peter Baumgartner, Cesare Tinelli. The Model Evolution Calculus. CADE 2003
- Harald Ganzinger, Konstantin Korovin. New Directions in Instantiation-Based Theorem Proving. LICS 2003

Splitting:

- Christoph Weidenbach. Combining Superposition, Sorts and Splitting. Handbook of Automated Reasoning 2001
- Alexandre Riazanov, Andrei Voronkov. Splitting Without Backtracking. IJCAI 2001
- Krystof Hoder, Andrei Voronkov. The 481 Ways to Split a Clause and Deal with Propositional Variables. CADE 2013

Components, Splitting

Let $C_1, \ldots, C_n$ be clauses with pairwise disjoint sets of variables, $n \geq 2$.

The clause $D \stackrel{\text{def}}{=} C_1 \vee \ldots \vee C_n$ can be split into $C_1, \ldots, C_n$.
A component is a non-empty clause which cannot be split.

Every non-empty clause has a unique splitting into components.

Components, Splitting

Let $C_1, \ldots, C_n$ be clauses with pairwise disjoint sets of variables, $n \geq 2$.

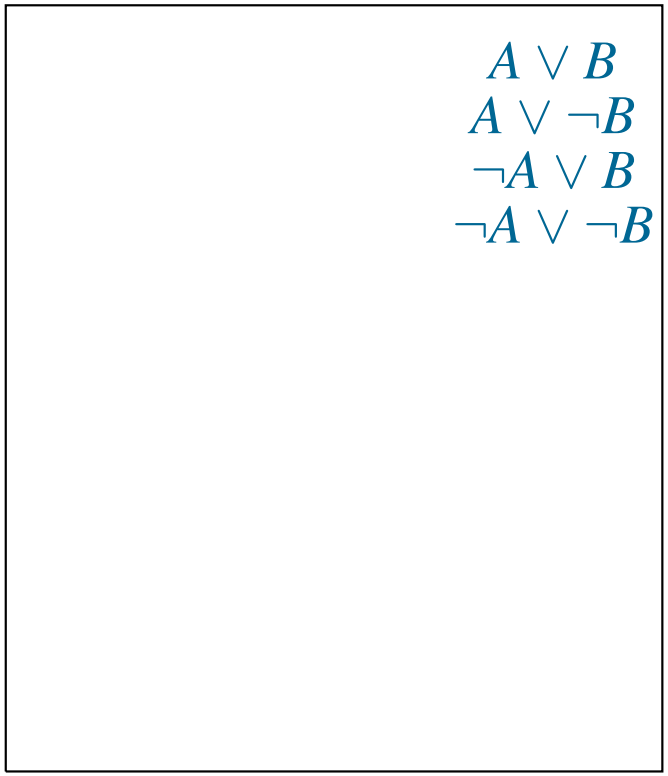
The clause $D \stackrel{\text{def}}{=} C_1 \vee \ldots \vee C_n$ can be split into $C_1, \ldots, C_n$.
A component is a non-empty clause which cannot be split.

Every non-empty clause has a unique splitting into components.

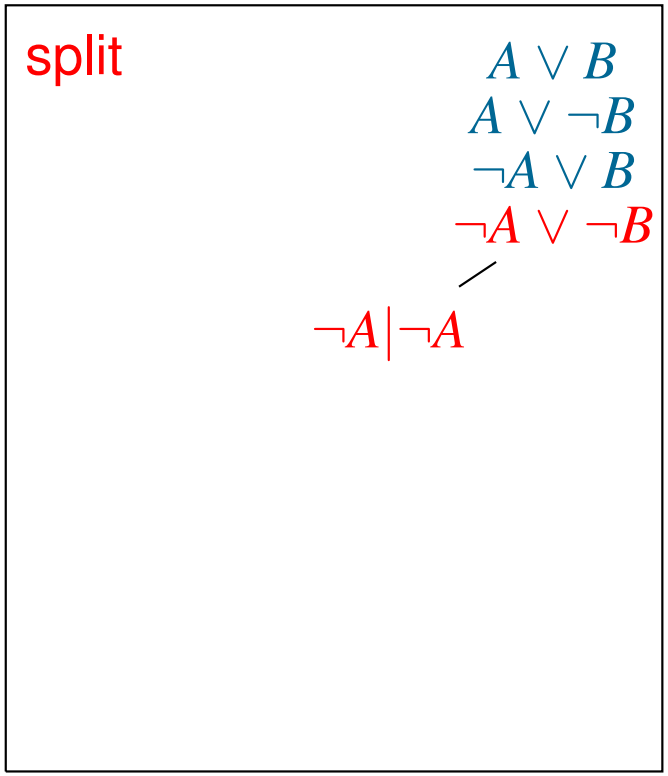
Previous implementations:

- Splitting with backtracking (hard to implement, moderate improvement);
- Splitting without backtracking (rarely improves);

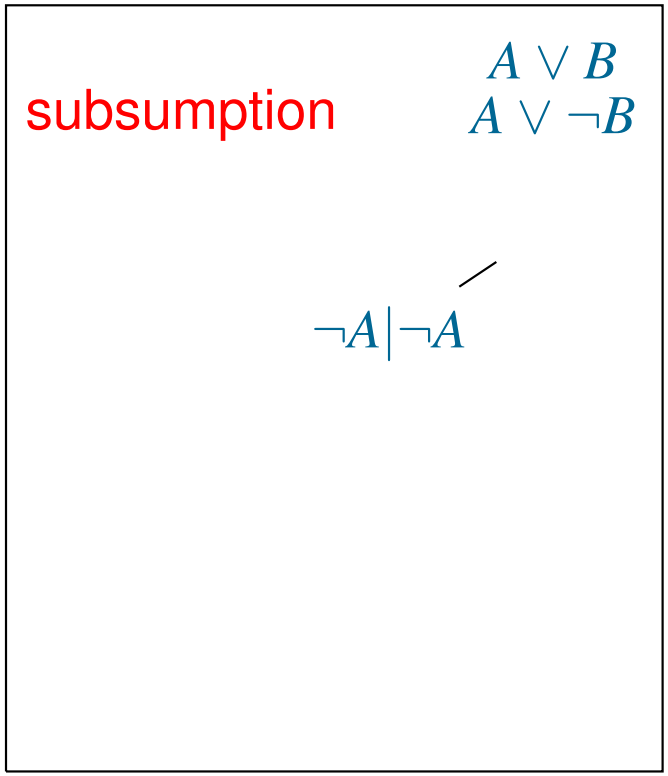
Splitting With Backtracking



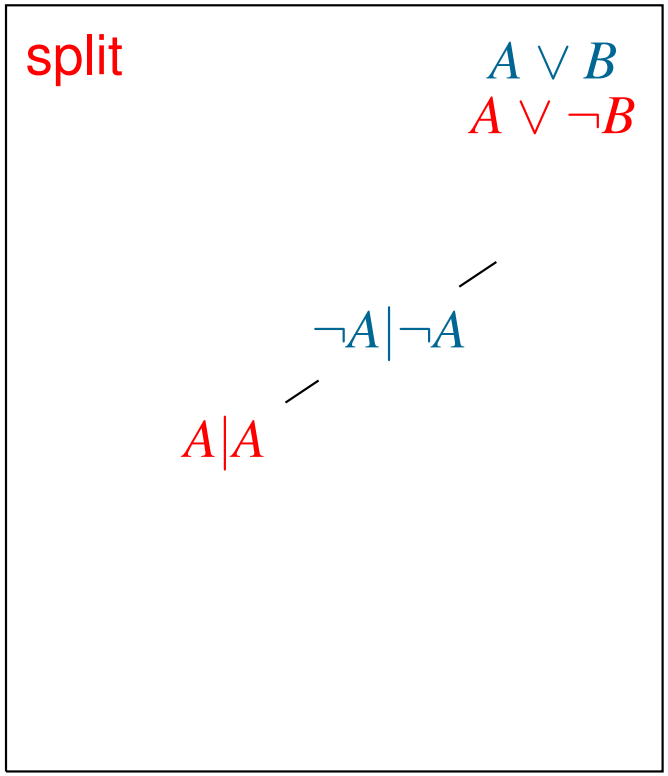
Splitting With Backtracking



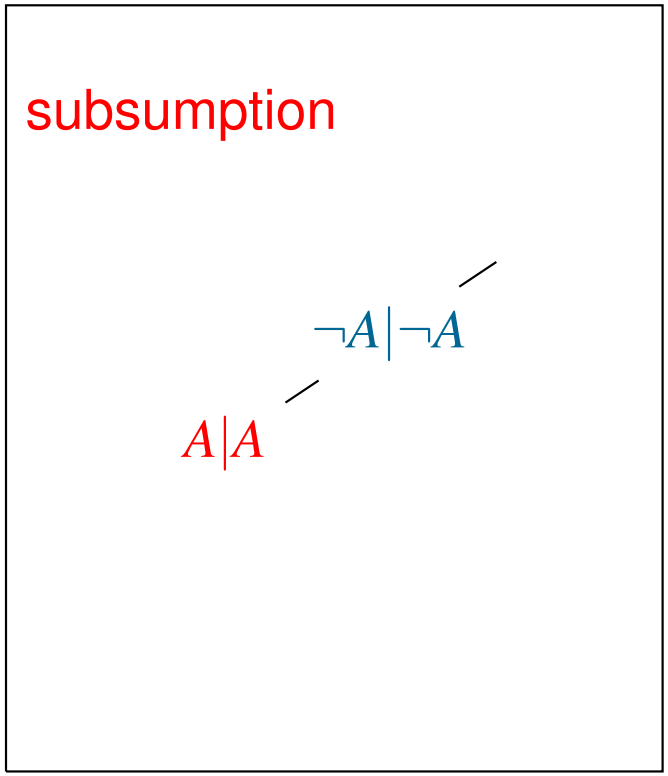
Splitting With Backtracking



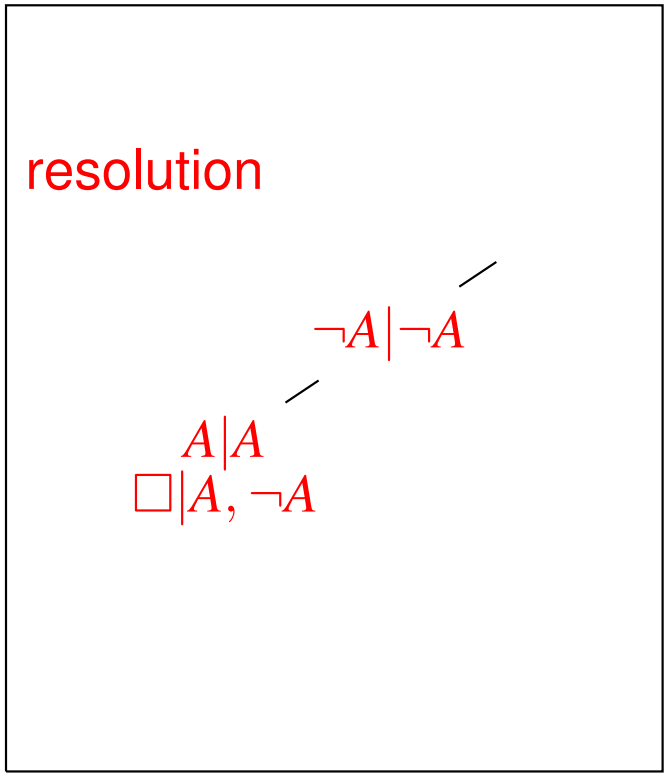
Splitting With Backtracking



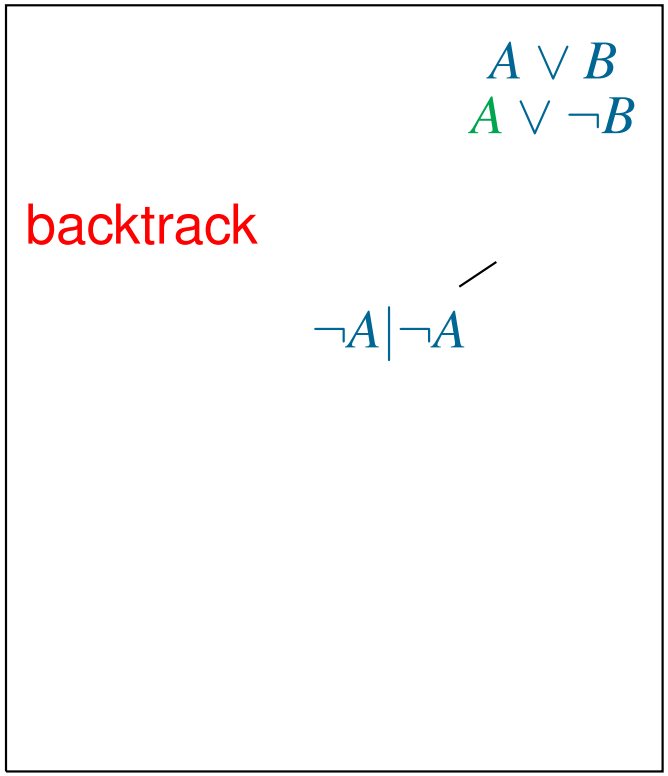
Splitting With Backtracking



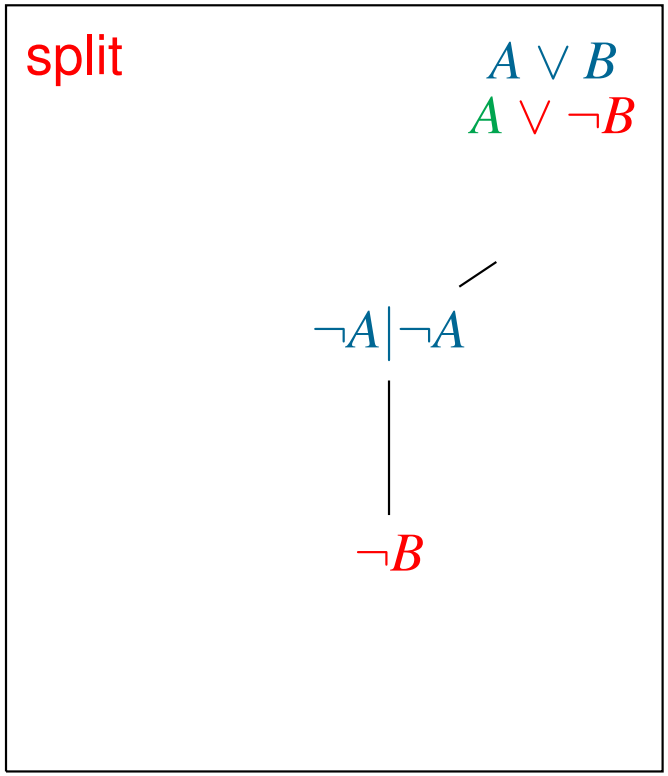
Splitting With Backtracking



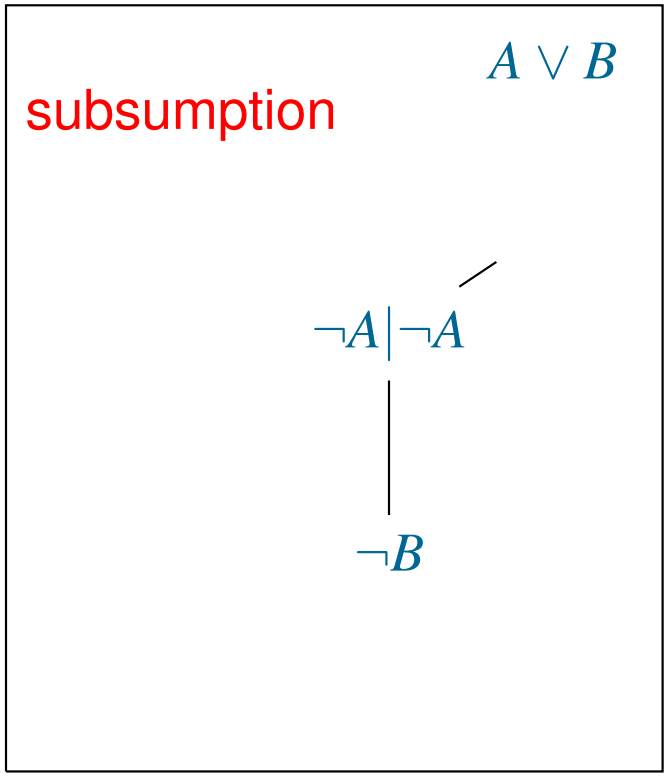
Splitting With Backtracking



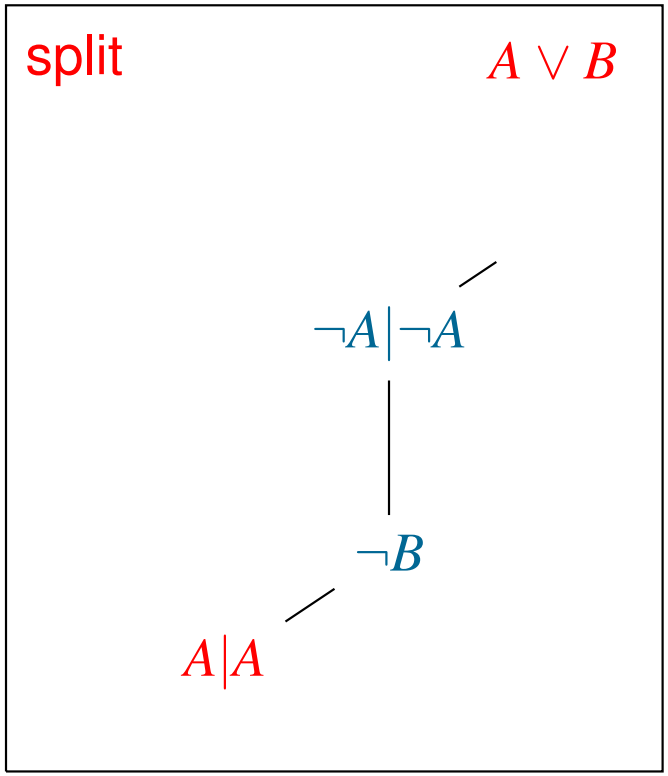
Splitting With Backtracking



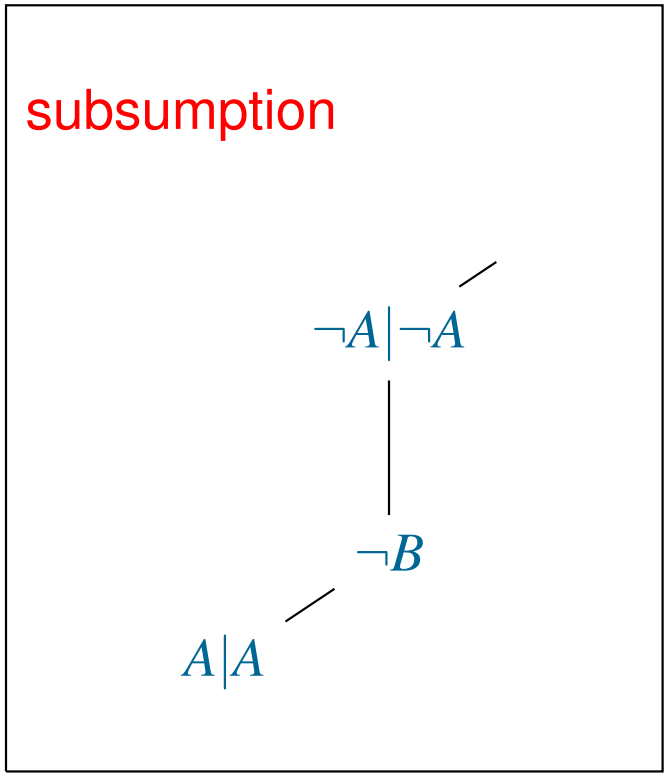
Splitting With Backtracking



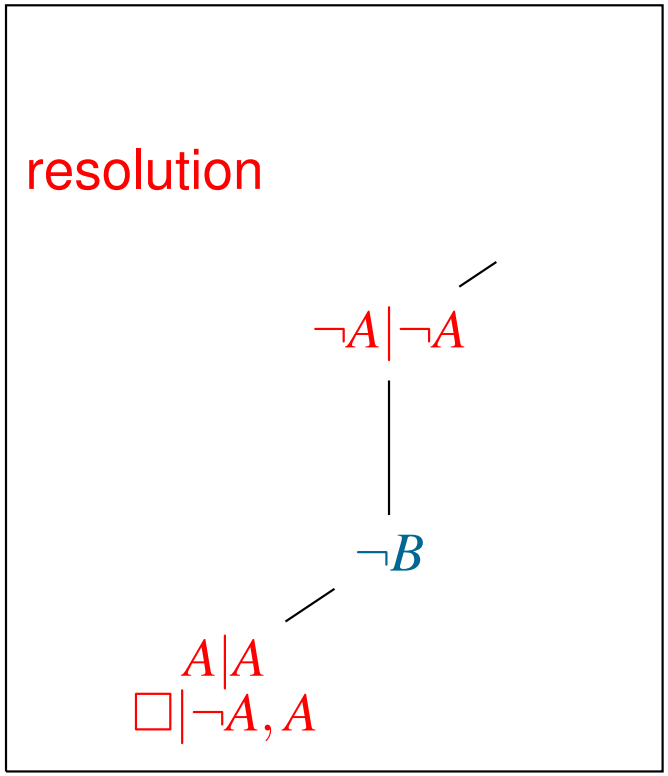
Splitting With Backtracking



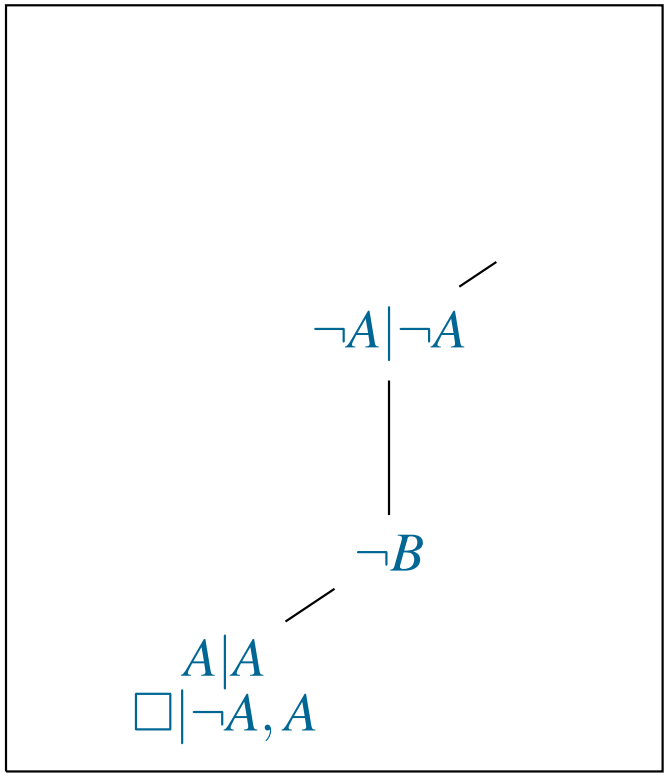
Splitting With Backtracking



Splitting With Backtracking



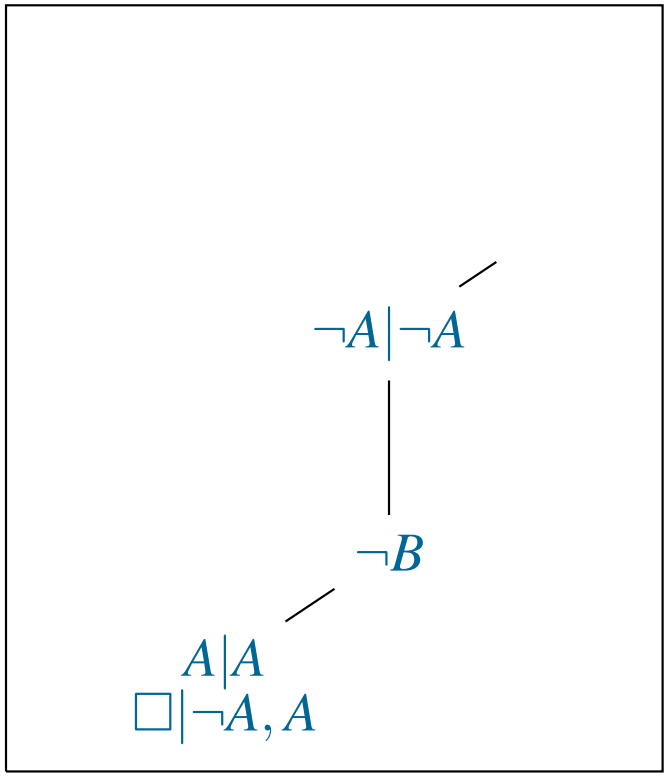
Splitting With Backtracking



And so on . . .

- Too many steps (for this example);
- Backtracking is expensive
- Exploits too many branches . . .

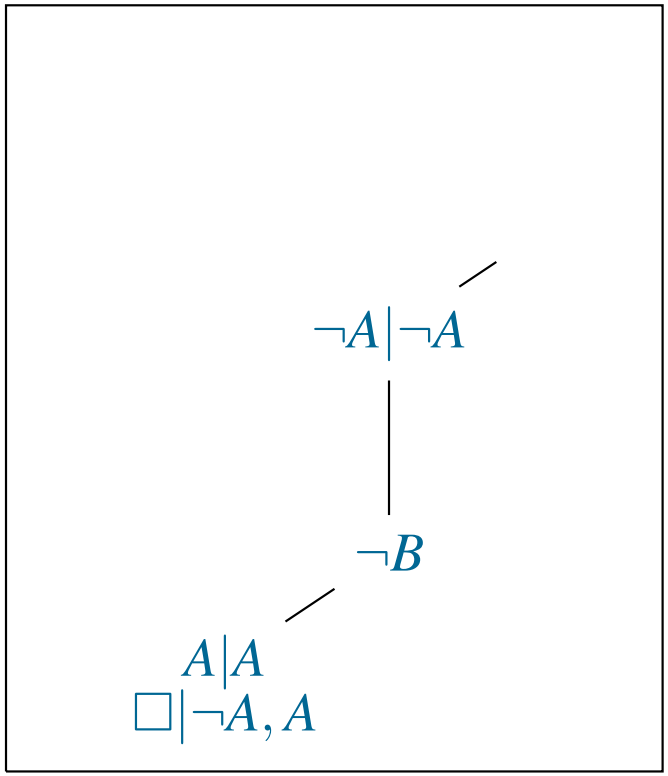
Splitting With Backtracking



And so on . . .

- Too many steps (for this example);
- Backtracking is expensive
- Exploits too many branches . . .
- Similar to the original DPLL . . . yet without unit propagation

Splitting With Backtracking



And so on . . .

- Too many steps (for this example);
- Backtracking is expensive
- Exploits too many branches . . .
- Similar to the original DPLL . . . yet without unit propagation
- Generally behaves well

Superposition for Clauses with Assertions

The standard superposition calculus

$$\frac{\underline{l = r} \vee D_1 \qquad\qquad \underline{s[l'] = t} \vee D_2}{(s[r] = t \vee D_1 \vee D_2)\theta}\ (\text{Sup}).$$

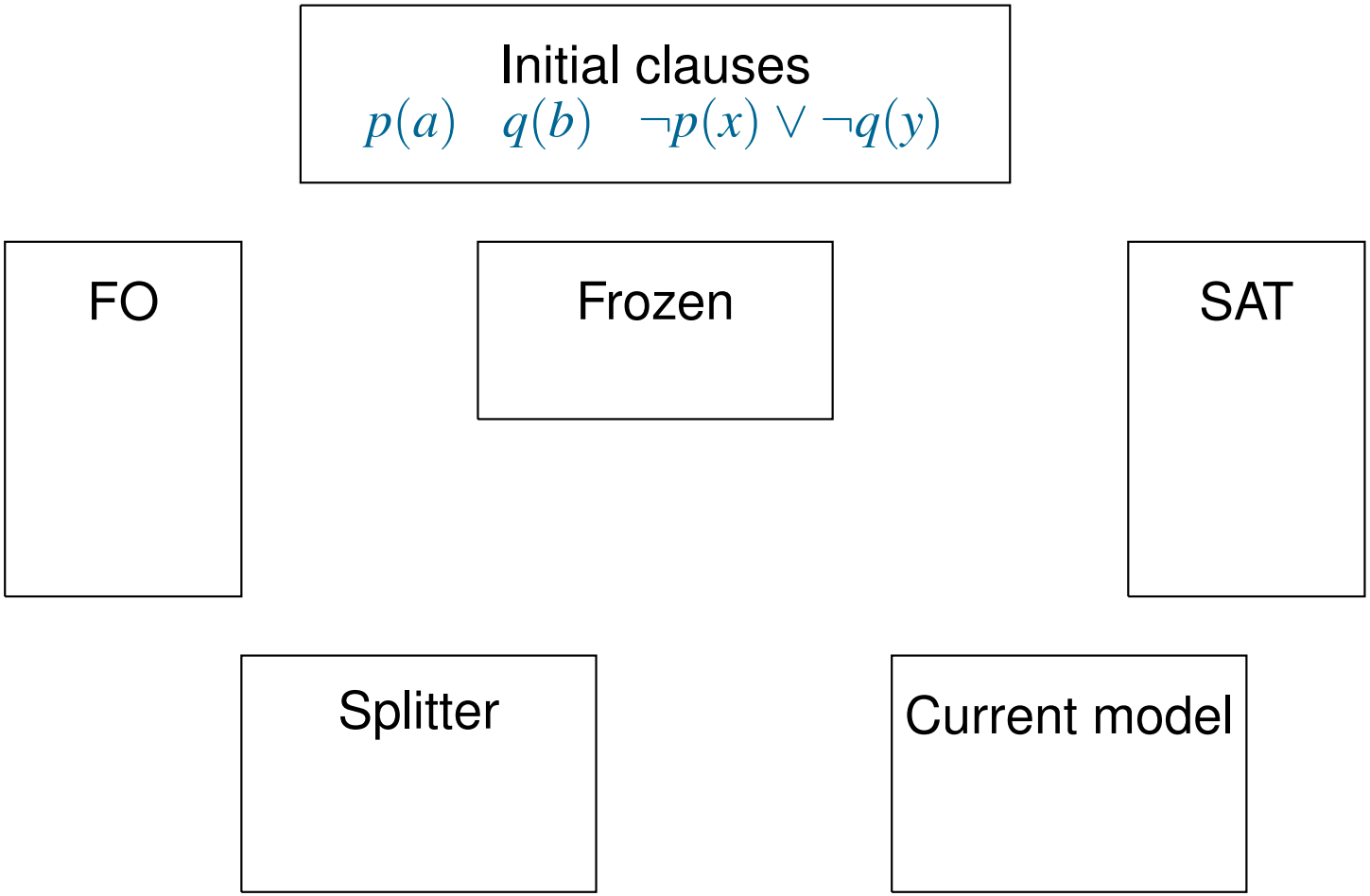
Superposition for Clauses with Assertions

The standard superposition calculus, with assertions inherited from premises.

$$\frac{\underline{l = r} \vee D_1 \mid \{C_1\} \quad s[l'] = t \vee D_2 \mid \{C_2\}}{(s[r] = t \vee D_1 \vee D_2)\theta \mid \{C_1, C_2\}} \text{ (Sup).}$$

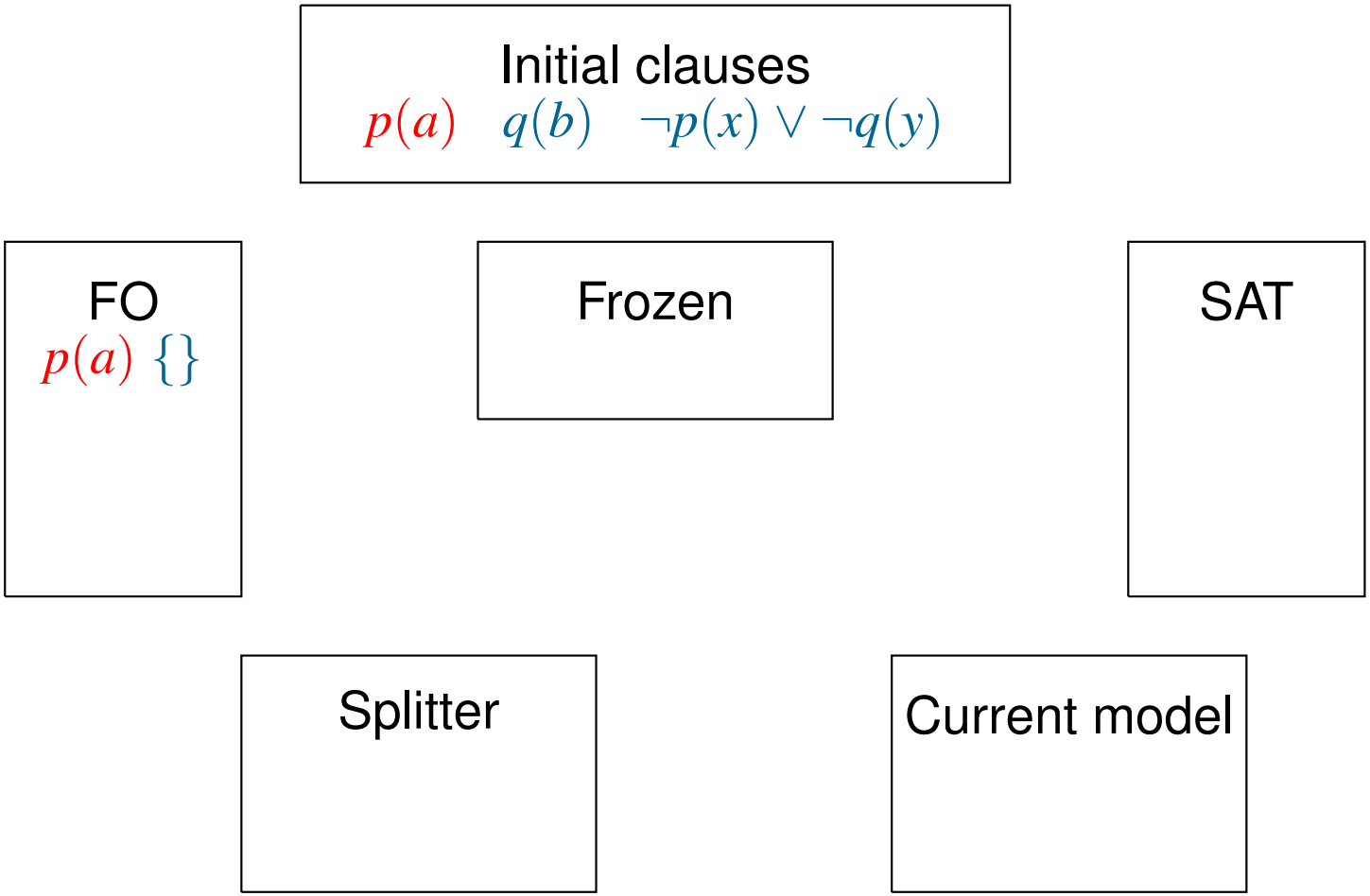
AVATAR

A SAT solver, which treats a component as a propositional variable.



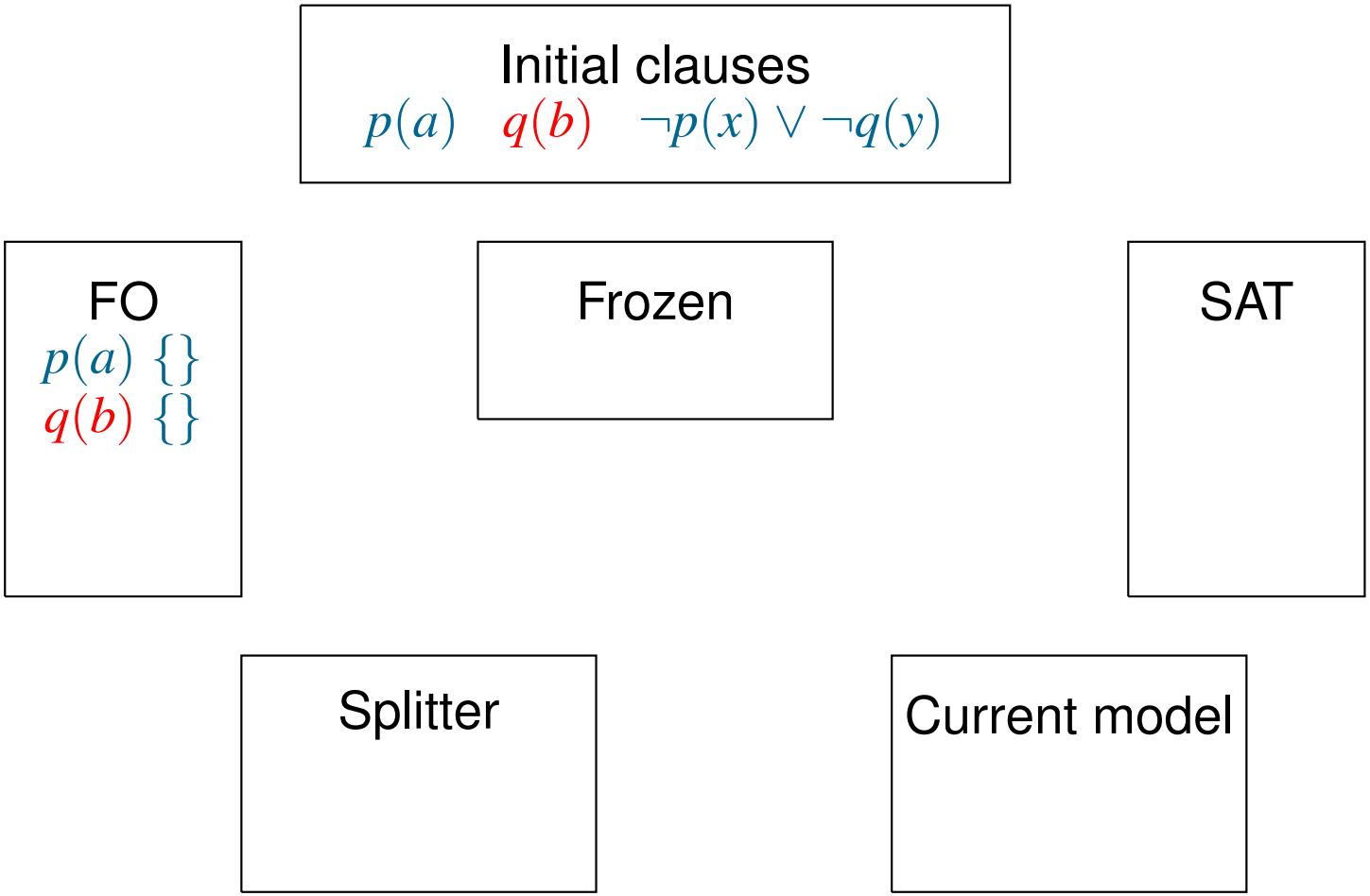
AVATAR

A SAT solver, which treats a component as a propositional variable.



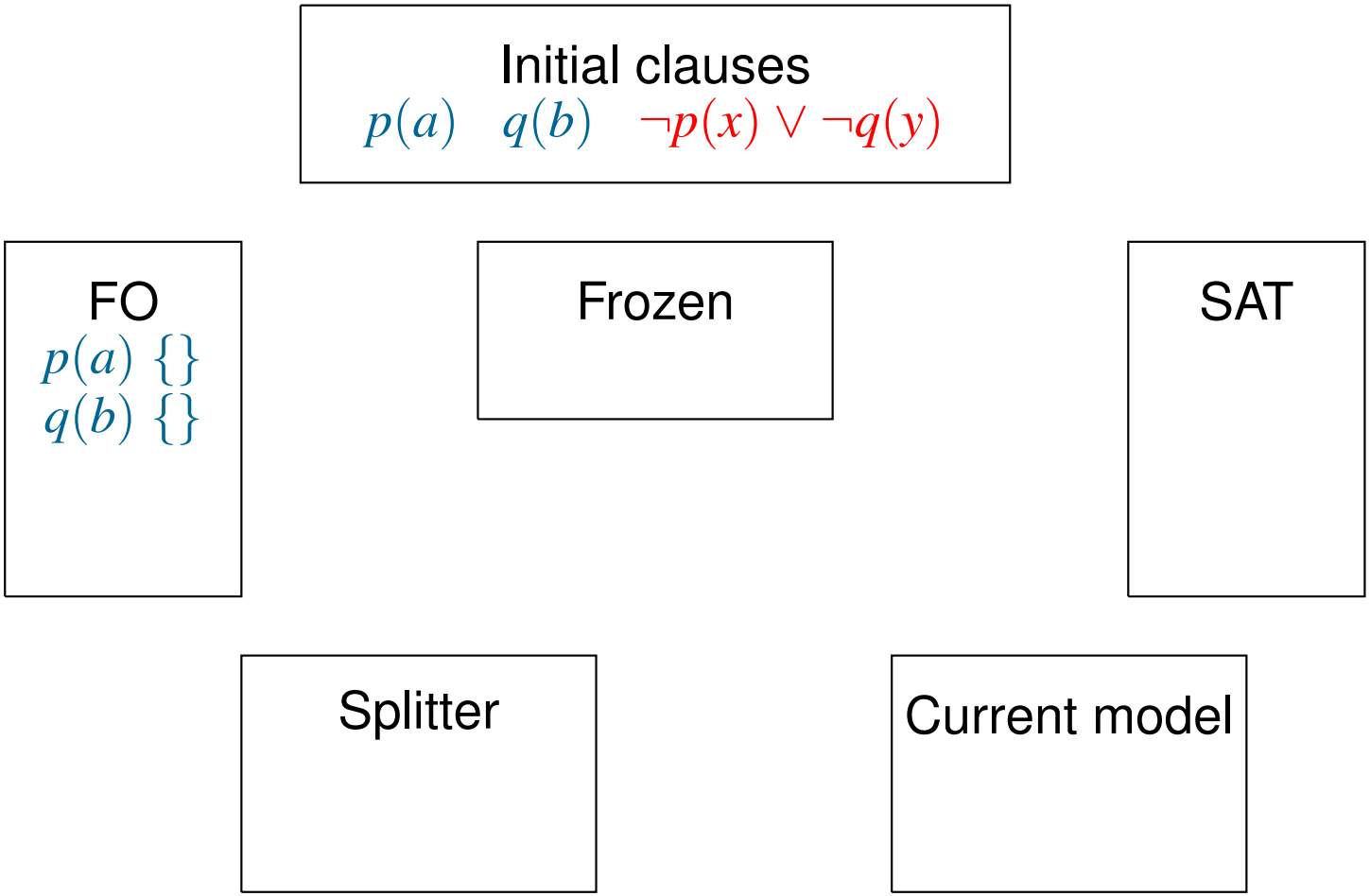
AVATAR

A SAT solver, which treats a component as a propositional variable.



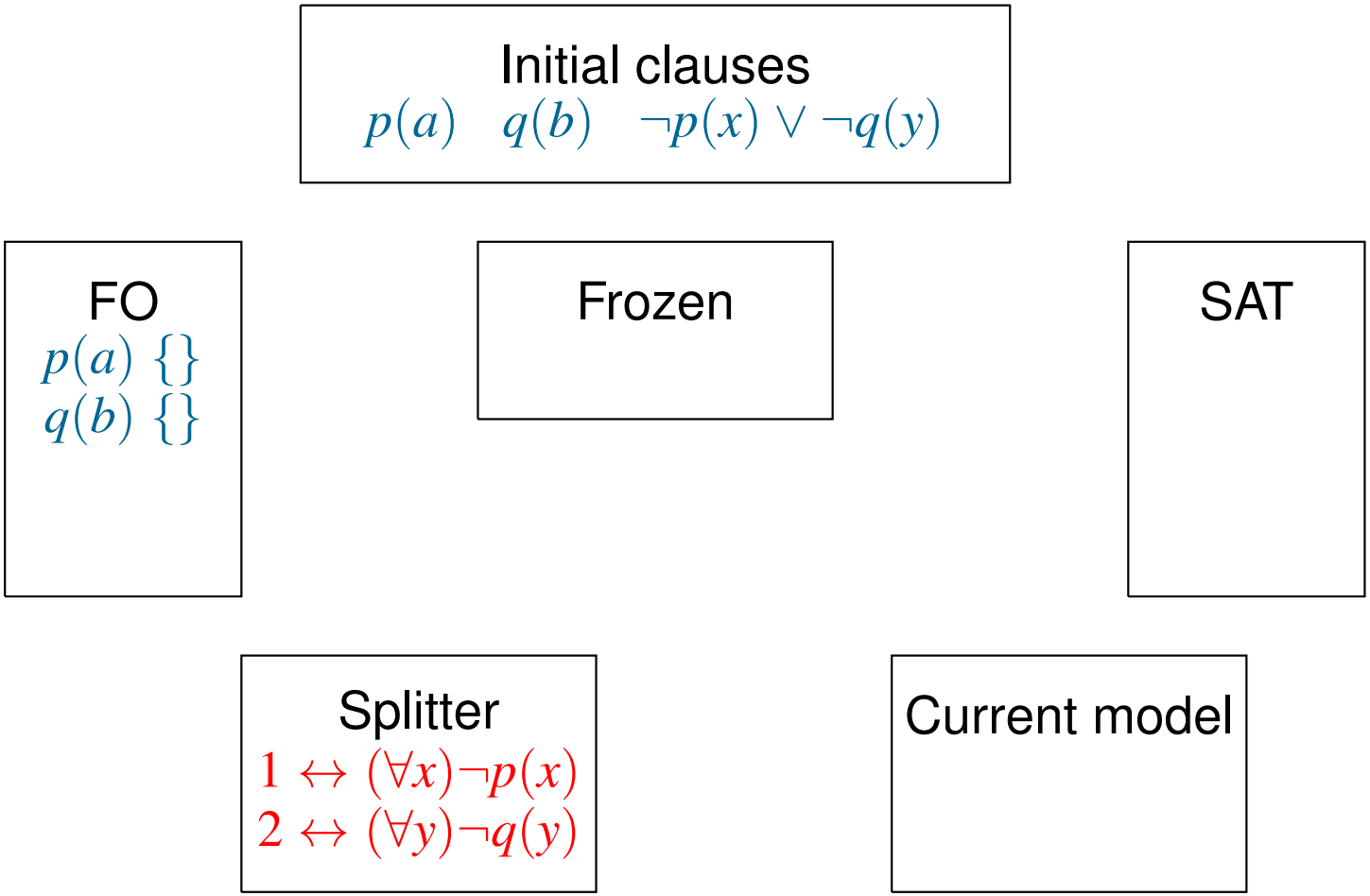
AVATAR

A SAT solver, which treats a component as a propositional variable.



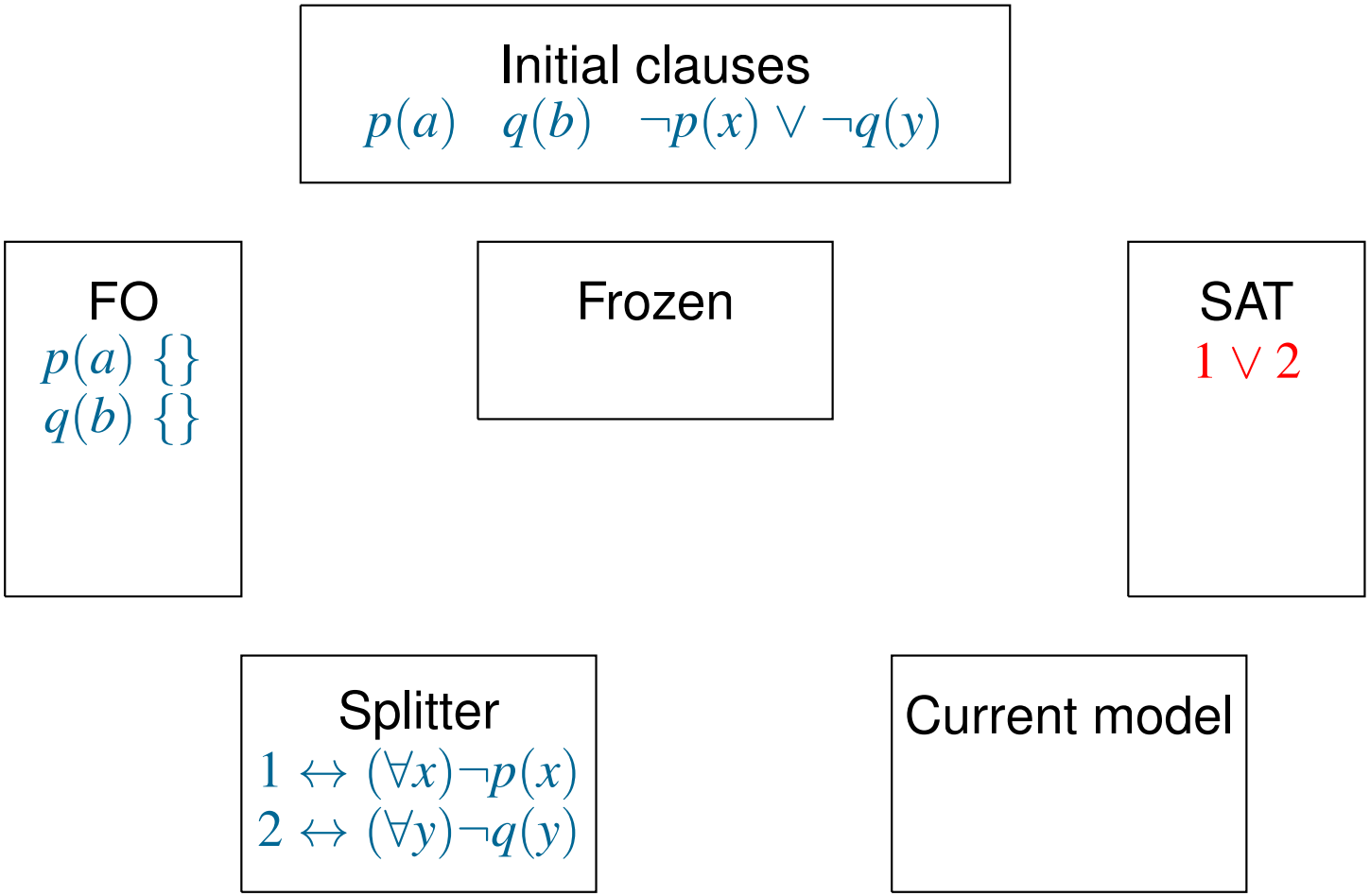
AVATAR

A SAT solver, which treats a component as a propositional variable.



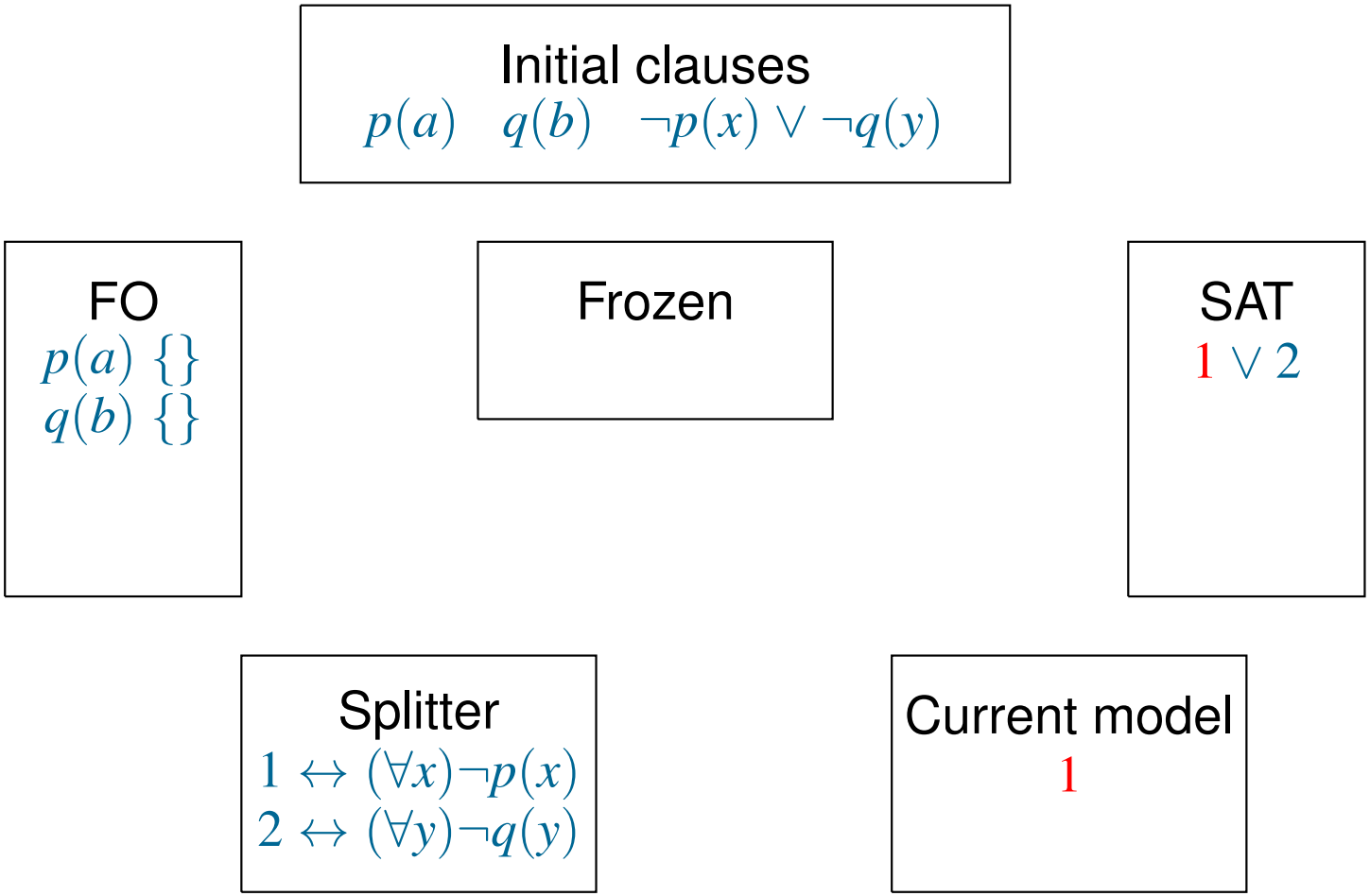
AVATAR

A SAT solver, which treats a component as a propositional variable.



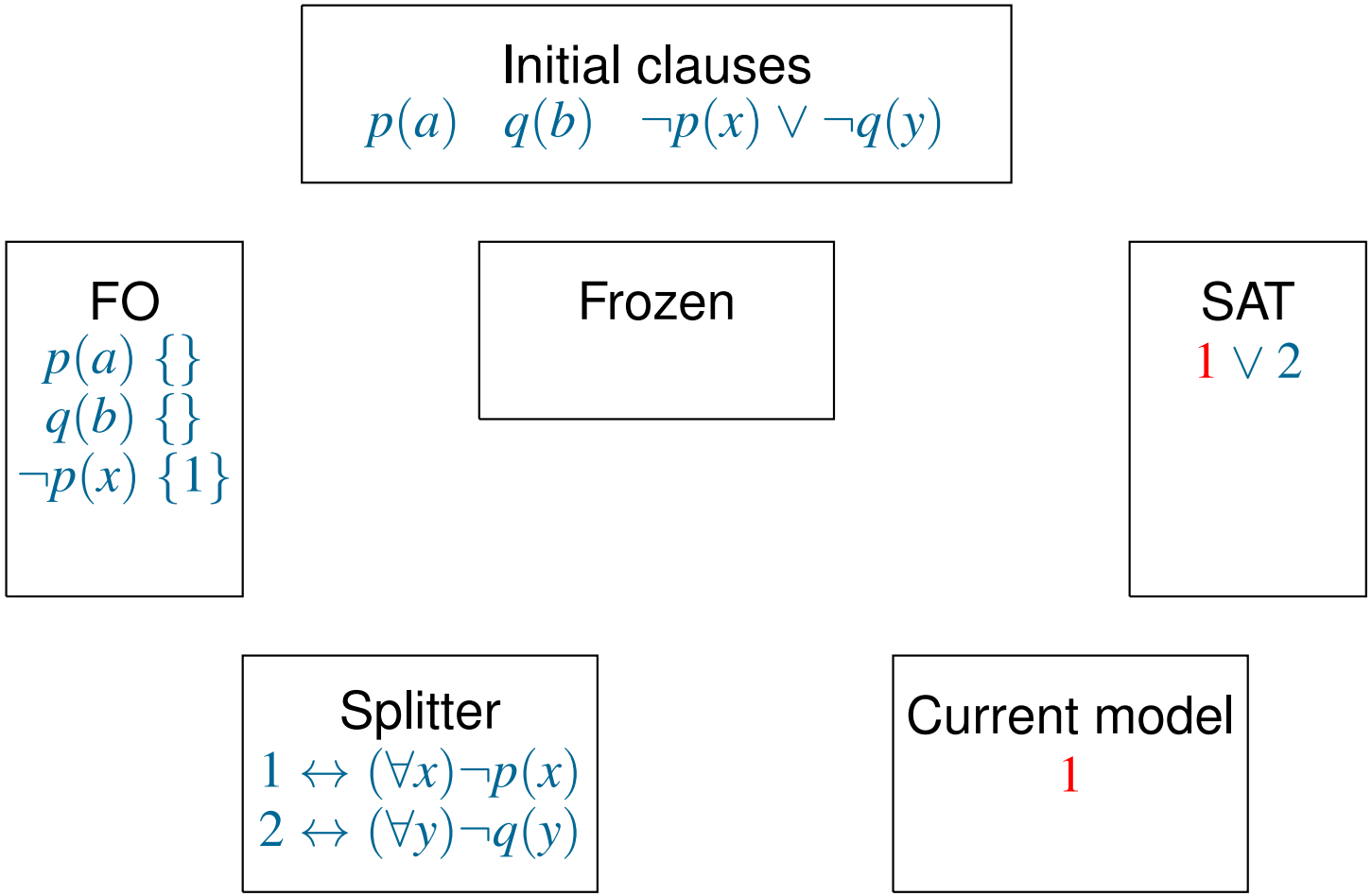
AVATAR

A SAT solver, which treats a component as a propositional variable.



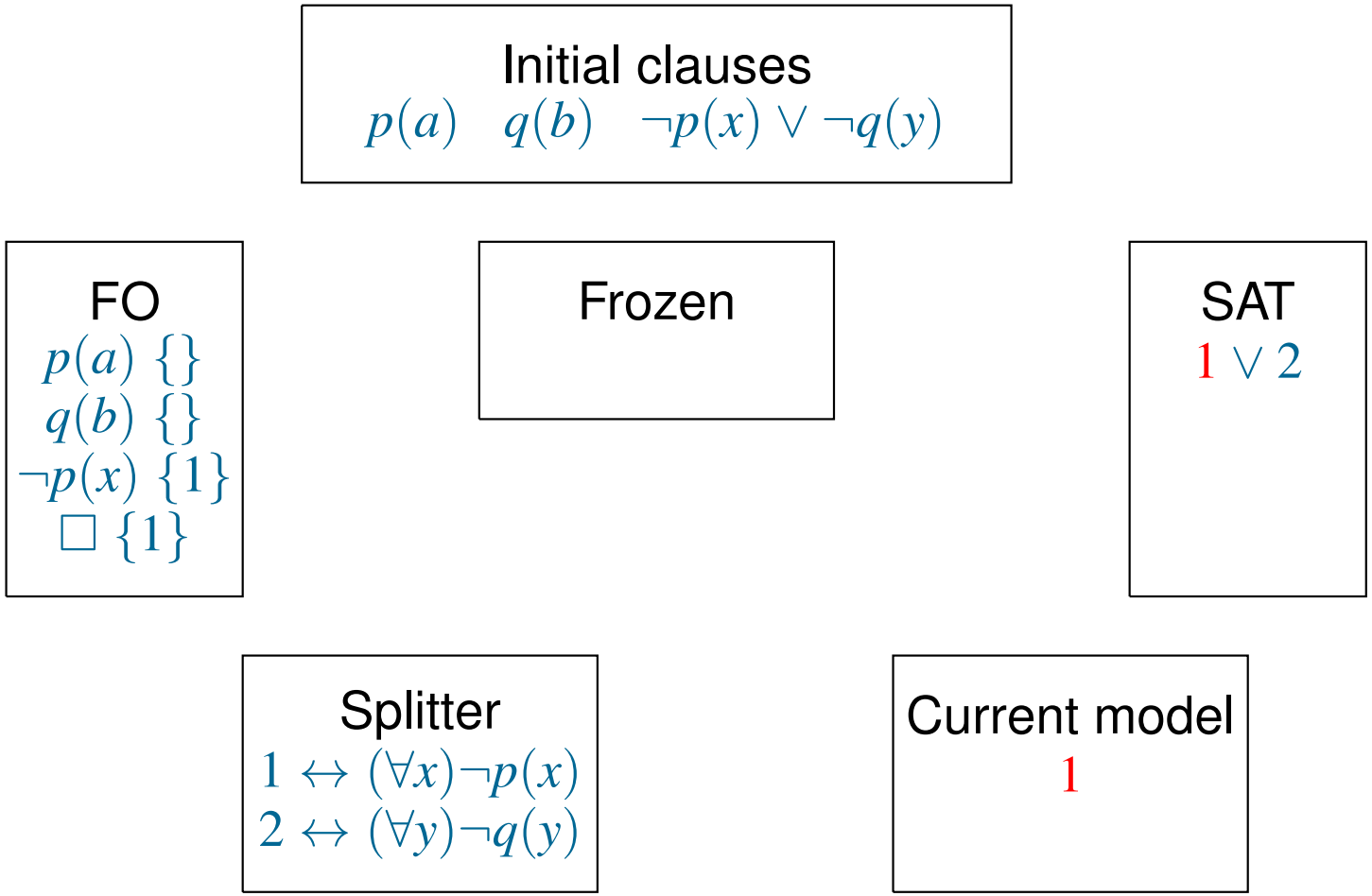
AVATAR

A SAT solver, which treats a component as a propositional variable.



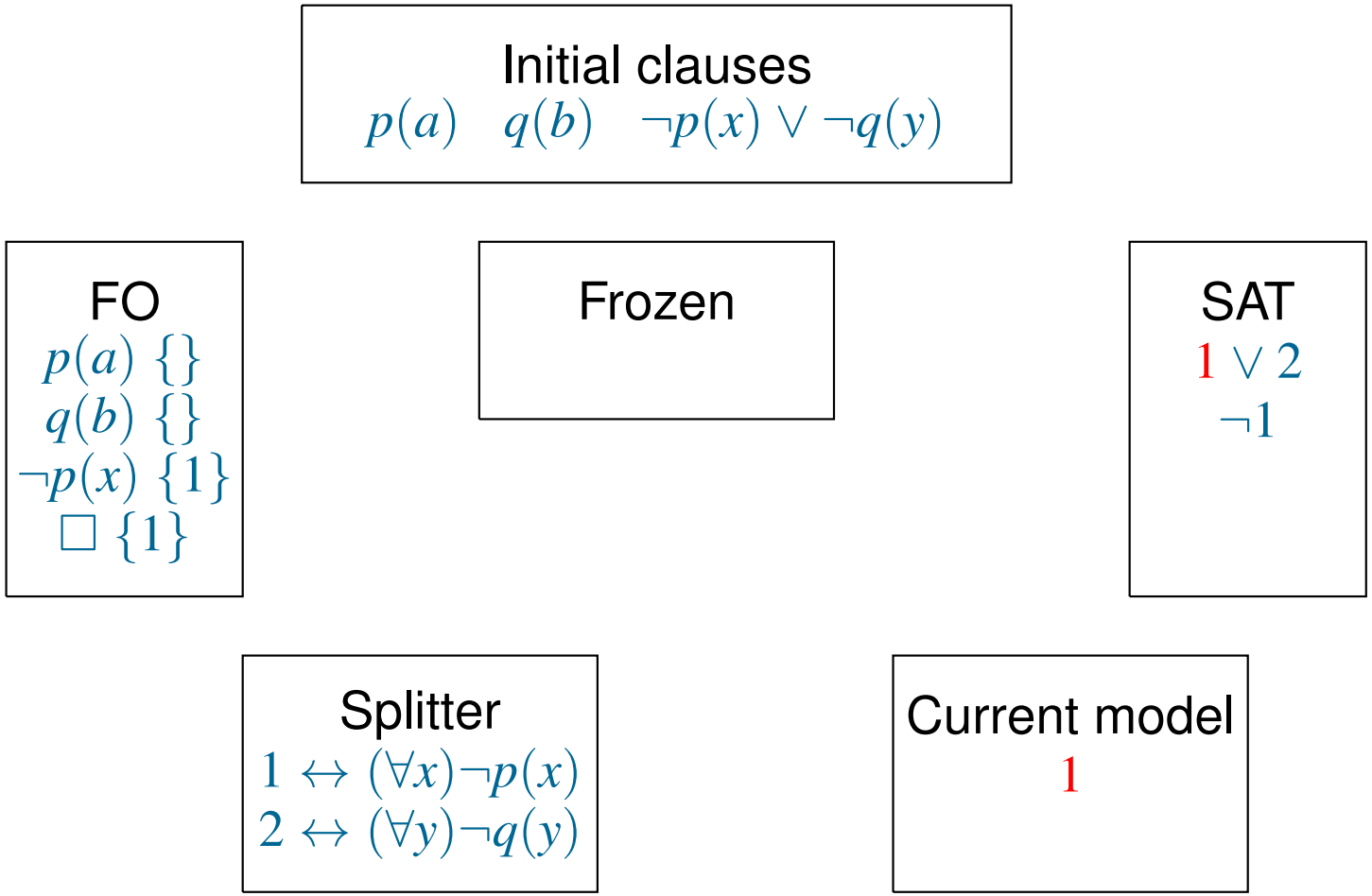
AVATAR

A SAT solver, which treats a component as a propositional variable.



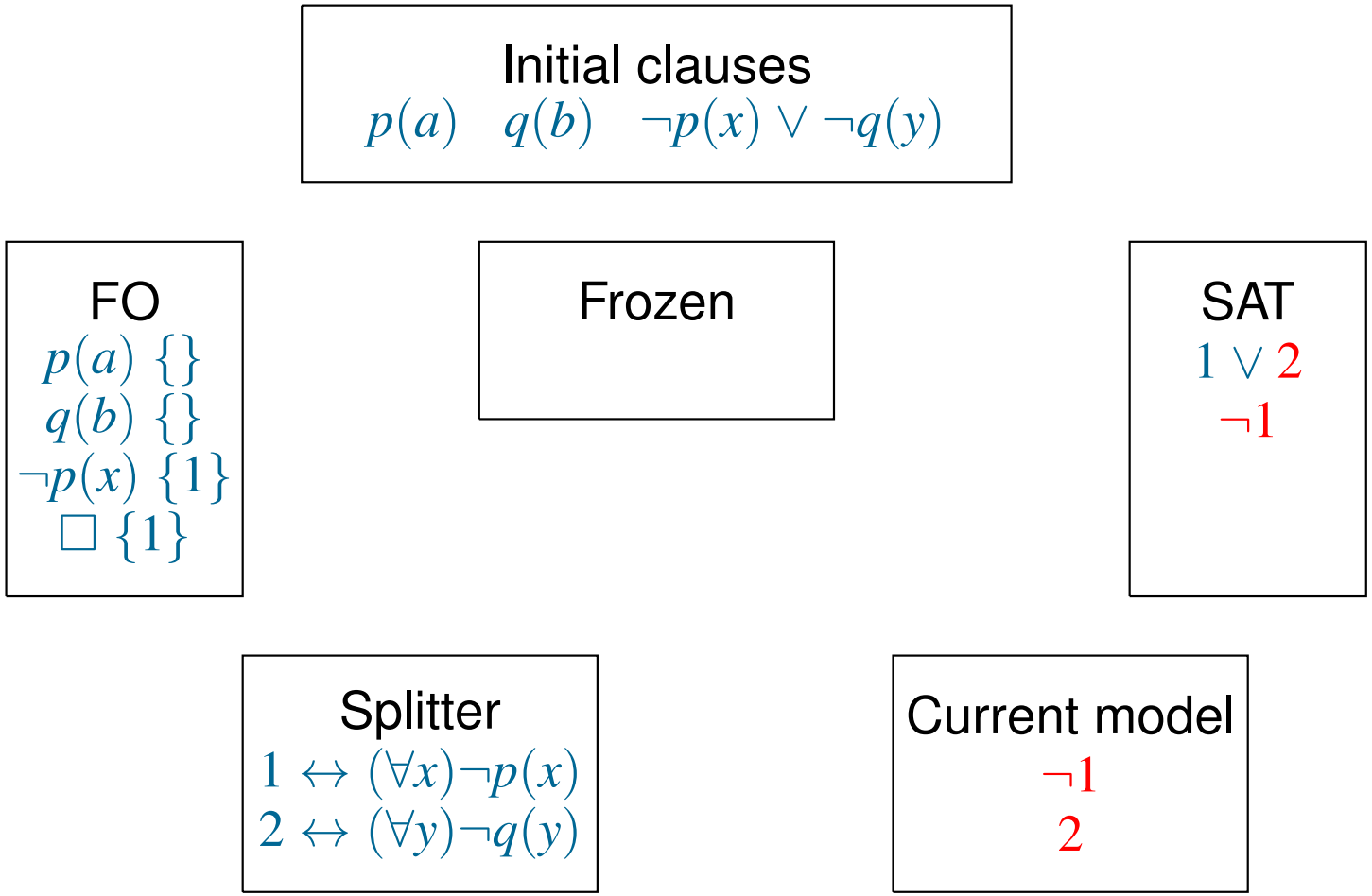
AVATAR

A SAT solver, which treats a component as a propositional variable.



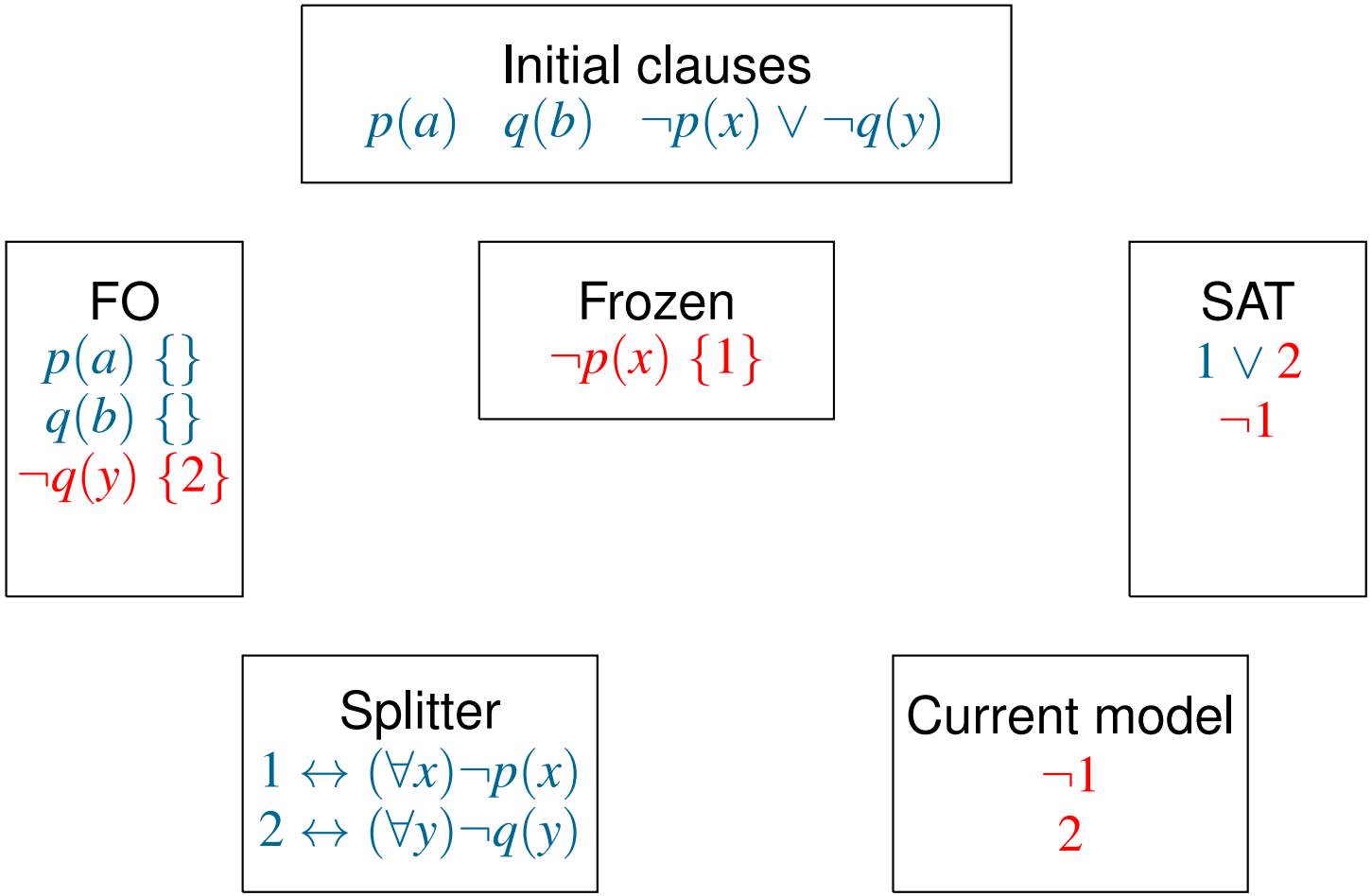
AVATAR

A SAT solver, which treats a component as a propositional variable.



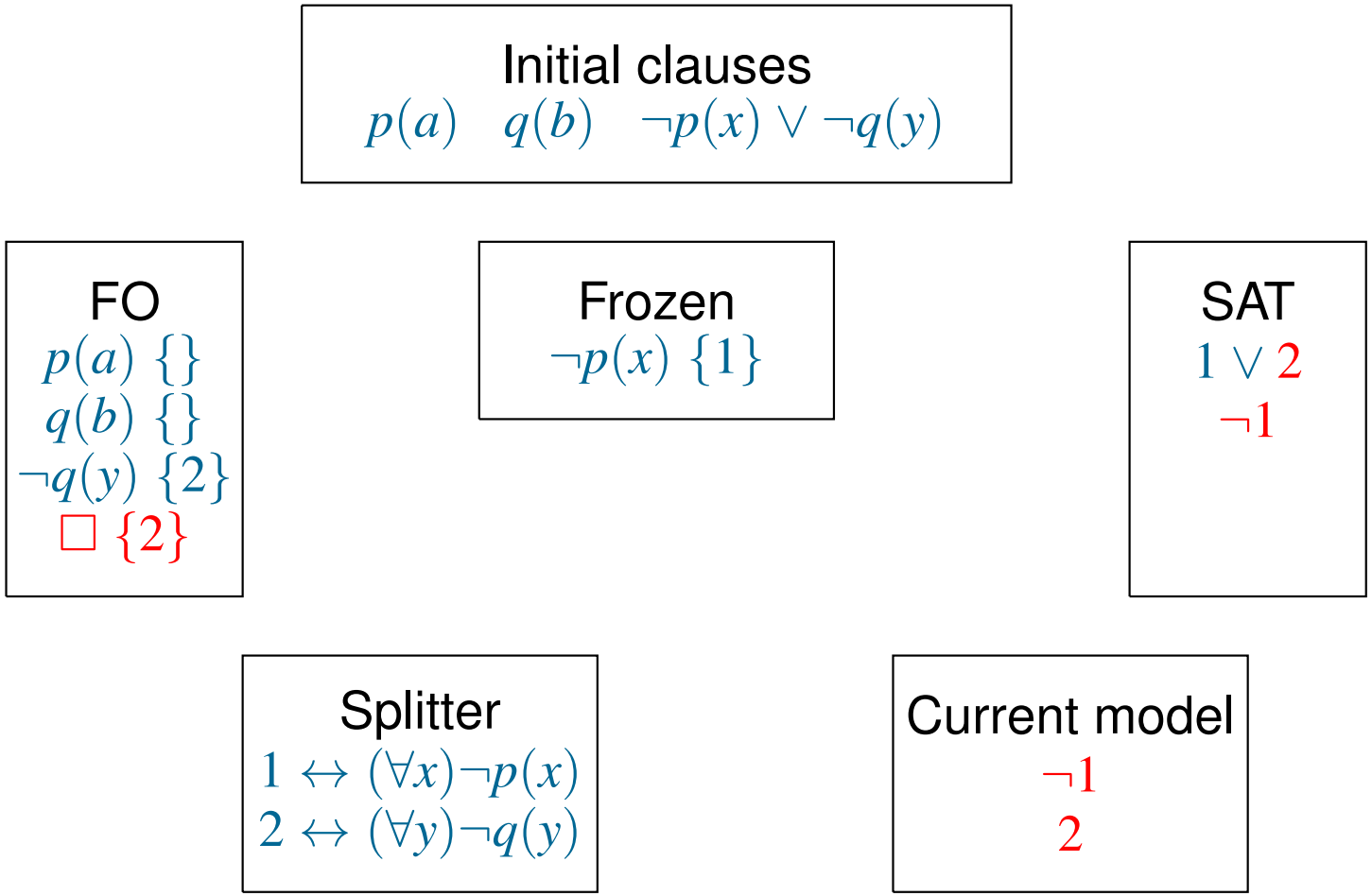
AVATAR

A SAT solver, which treats a component as a propositional variable.



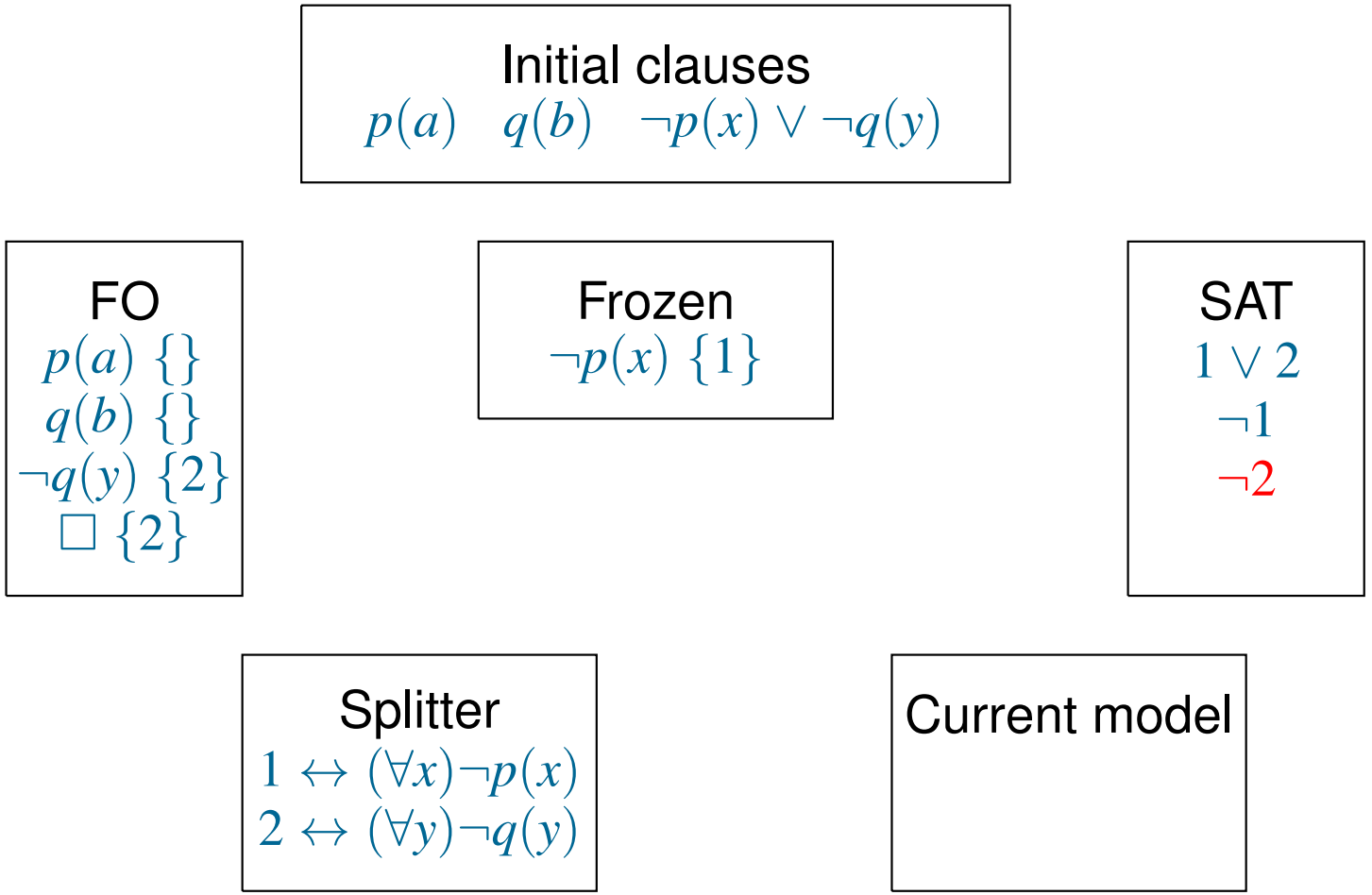
AVATAR

A SAT solver, which treats a component as a propositional variable.



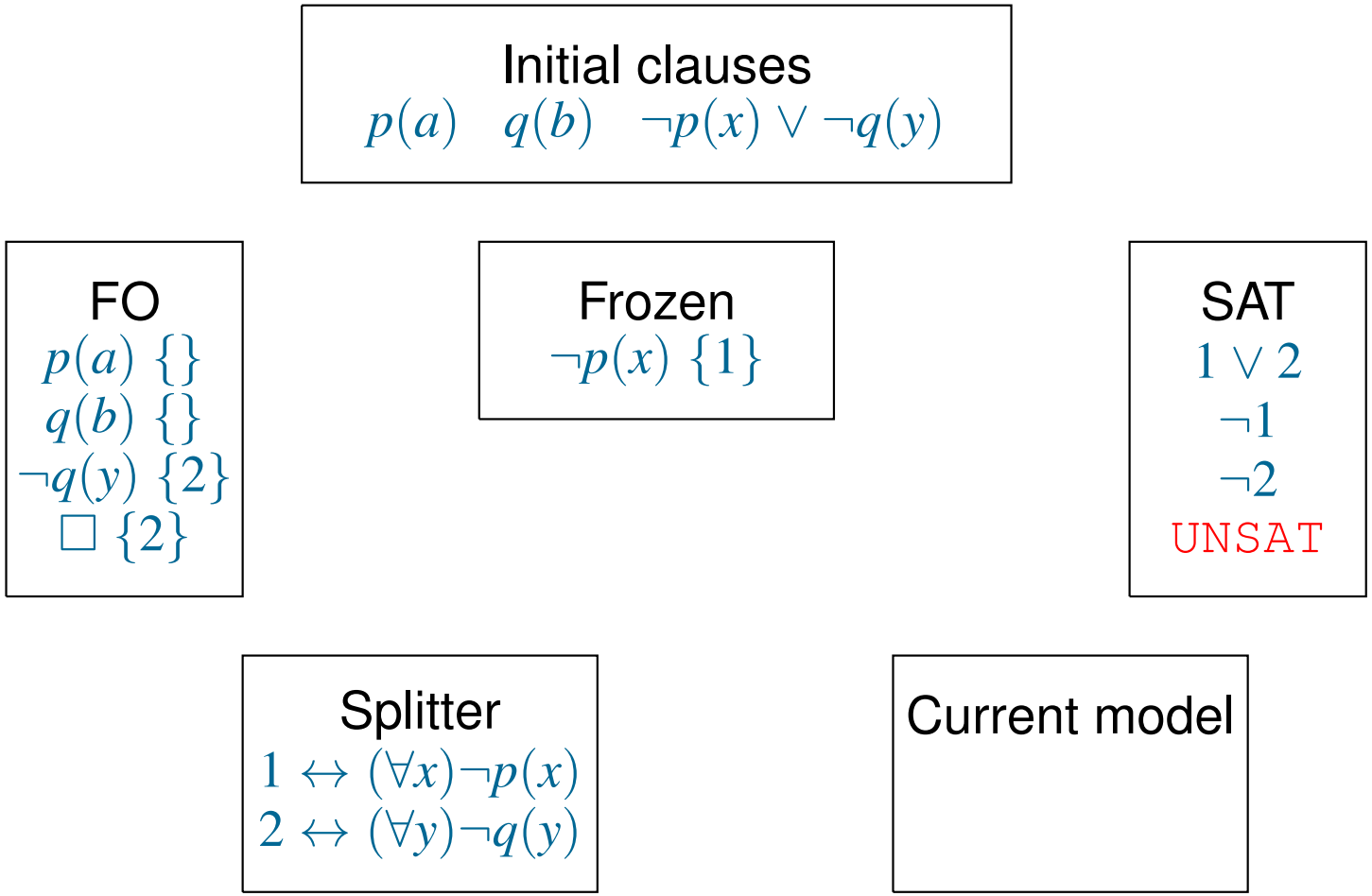
AVATAR

A SAT solver, which treats a component as a propositional variable.



AVATAR

A SAT solver, which treats a component as a propositional variable.



AVATAR and Vampire

AVATAR and Vampire

Notice the Vampire logo . . .

Problems

Implementing AVATAR heavily affects the saturation algorithm, redundancy and indexing.

- Clause deletion and restoration via frozen clauses;
- Redundancy checking;
- Indexing with frozen clauses.

First Results

- Test after the very first implementation (2011?) : we proved over 400 TPTP problems previously unsolved by any prover (including Vampire), probably unmatched since the TPTP appeared.
- About 5-10% increase on the number of problems solved by a single strategy.
- All splitting options and a lot of hard-to-maintain code removed from Vampire.

First Results

- Test after the very first implementation (2011?) : we proved over 400 TPTP problems previously unsolved by any prover (including Vampire), probably unmatched since the TPTP appeared.
- About 5-10% increase on the number of problems solved by a single strategy.
- All splitting options and a lot of hard-to-maintain code removed from Vampire.

First-order Theorems	*Vampire* 2.6	ET 0.1	E 1.9	VanHElsi 1.0	CVC4 1.4-FOF	iProver 1.4	leanCoP 2.2	Prover9 1109a	Zipperpos 0.4-FOF	Muscadet 4.4	Princess 140704
Solved/400	375/400	339/400	321/400	310/400	215/400	216/400	158/400	95/400	73/400	32/400	134/400
Av. CPU Time	13.19	29.31	22.88	17.29	46.03	18.11	55.15	41.45	28.81	19.74	69.31
Solutions	372/400	339/400	321/400	310/400	215/400	214/400	158/400	95/400	73/400	30/400	0/400
µEfficiency	571	361	466	168	228	216	129	119	75	47	17
SOTAC	0.22	0.18	0.17	0.17	0.15	0.16	0.14	0.14	0.13	0.12	0.13
Core Usage	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	0.99	1.22
New Solved	5/6	5/6	0/6	0/6	0/6	6/6	0/6	0/6	0/6	0/6	0/6

Recent Result: a Semigroup Partition Problem

Problem ALG013-1 in TPTP.

Thierry Coquand

I looked back at my notes. The goal was to analyze the proof that a compact semigroup has a fixed point. (In the finite case, there is a direct proof.) As shown by $\{1, 2, 3, \}$ this is not valid in general. I wanted an effective version in the case of a compact totally disconnected semigroup. In this case, if we have $x^2 <> x$ for all x, we get a finite partition $A_1, \ldots, A_n$ where $A_i \cdot A_i$ is disjoint from A_i. Can we find such a partition in general? This is now a first-order statement with no finite model. For $n = 2$, there is a simple direct proof that this is not possible. For $n = 3$, I could not do it, so I asked Koen if he could do it automatically (May 26 2003). He answered at once that Gandalf was able to do it in 24 seconds on his machine. For $n = 4$, one should try the more general statement

$$
\begin{array}{lcl}
A_1(x) \wedge A_1(y) & \rightarrow & A_2(x \cdot y) \vee A_3(x \cdot y) \vee A_4(x \cdot y) \\
A_2(x) \wedge A_2(y) & \rightarrow & A_1(x \cdot y) \vee A_3(x \cdot y) \vee A_4(x \cdot y) \\
A_3(x) \wedge A_3(y) & \rightarrow & A_1(x \cdot y) \vee A_2(x \cdot y) \vee A_4(x \cdot y) \\
A_4(x) \wedge A_4(y) & \rightarrow & A_1(x \cdot y) \vee A_2(x \cdot y) \vee A_3(x \cdot y)
\end{array}
$$

The Semigroup Partition Proof

Inferences just before the very last one. . .

```
727706.  c2(f(a3,f(a4,a4))) | c2(f(a3,f(a4,a4))) {5688} (26:13) [forward demodulation 561922,1]
727707.  c2(f(a3,f(a4,a4))) {5688} (26:7) [duplicate literal removal 727706]
727720. d2(f(a3,f(a4,f(a4,f(a3,a4))))) | c2(f(a3,f(a4,a4))) {218, 5401} (25:18) [subsumption resolution
220373,6400]
728355.  c2(f(a4,f(f(a3,a4),X8))) | c2(f(a3,f(a4,a4))) | d1(X8) {218, 5689} (26:18) [subsumption resolution
659376,6400]
728356.  c2(f(a4,f(a3,f(a4,X8)))) | c2(f(a3,f(a4,a4))) | d1(X8) {218, 5689} (26:18) [forward demodulation
728355,1]
728408.  c2(f(a3,f(a4,a3))) | c2(f(a3,f(a4,a4))) {5400} (25:13) [forward demodulation 220226,1]
728409. $false {53, 1350, 3434} (16:3) [subsumption resolution 233789,37927]
729294.  d1(X3) | c1(f(a4,f(a4,X3))) | c1(a1) {622} (24:11) [resolution 18631,2325]
729349.  d1(X3) | c1(f(a4,f(a4,X3))) {622} (24:9) [subsumption resolution 729294,9]
731025.  c2(f(a4,f(f(a4,f(a1,a4)),a4))) {452, 2452} (18:12) [resolution 91317,12597]
731082.  c2(f(a4,f(a4,f(f(a1,a4),a4)))) {452, 2452} (18:12) [forward demodulation 731025,1]
731083.  c2(f(a4,f(a4,f(a1,f(a4,a4))))) {452, 2452} (18:12) [forward demodulation 731082,1]
731199.  d2(f(a1,f(a1,a4))) | d2(f(a1,a3)) {1785, 7903} (26:12) [subsumption resolution 566737,52555]
731200.  d2(f(a1,f(a1,a4))) {124, 1785, 7903} (26:9) [subsumption resolution 731199,5387]
731903.  c2(f(a1,f(f(a1,a4),a4))) {452, 890} (26:10) [resolution 26604,12597]
731953.  c2(f(a1,f(a1,f(a4,a4)))) {452, 890} (26:10) [forward demodulation 731903,1]
737309.  c1(f(a1,f(a1,f(a4,f(a4,a3))))) {602, 1908} (22:12) [resolution 54000,18255]
740261.  c2(f(a3,f(a4,a4))) | c2(f(a4,f(a1,a3))) {54, 4050} (26:14) [resolution 142021,118608]
745106. c2(f(a4,a3)) | c1(f(a4,a3)) | c2(f(a4,f(a4,a3))) {225, 7196} (20:16) [resolution 271392,6524]
753227.  d1(X16) | c2(f(a1,f(a4,f(a4,a1)))) | c2(f(X16,a4)) {4692} (23:15) [resolution 183284,751]
753228.  c2(f(a1,f(a4,f(a4,a1)))) | c2(f(a3,a4)) {4692} (23:13) [resolution 183284,752]
753294.  d1(X16) | c2(f(X16,a4)) {760, 4692} (23:8) [subsumption resolution 753227,22552]
753295.  c2(f(a3,a4)) {760, 4692} (23:6) [subsumption resolution 753228,22552]
753296. $false {218, 760, 4692} (23:3) [subsumption resolution 753295,6400]
765660.  c2(f(a4,f(a3,f(a4,a3)))) {11638} (26:9) [resolution 579126,11]
767337.  d2(X8) | c2(f(a1,f(a1,a4))) | c2(f(X8,f(a3,a4))) {11816} (27:15) [resolution 584412,704]
767451.  d2(X8) | c2(f(X8,f(a3,a4))) {788, 11816} (27:10) [subsumption resolution 767337,23570]
768354. c2(f(a4,a3)) | c2(f(a4,f(a4,a3))) {225, 407, 7196} (20:13) [subsumption resolution 745106,11562]
768440.  d2(f(a3,a4)) | d2(f(a4,f(a1,a1))) {4001, 4841} (19:12) [subsumption resolution 542877,195650]
768697. d1(f(a3,f(a4,f(a4,a3)))) | c1(f(a3,f(a4,f(a4,a3)))) {641, 5931} (29:18) [subsumption resolution
538829,19301]
770228.  c2(f(a3,X0)) | c1(X0) | c2(X0) {1532, 1584} (25:10) [subsumption resolution 514531,41129]
```

The Semigroup Partition Proof: The Last Inference

772037. **$false** (0:0) [sat splitting refutation 322381,31135,511244,367817,65868,63149,61353,504668,123636,
23654,12834,23764,23766,118106,368552,99546,57496,65344,112646,133363,348639,153716,159265,98761,148710,
117176,519910,23731,584403,222823,593104,602542,270943,89534,155137,271783,133365,217863,177563,275097,368343,
425800,56830,64804,48880,577969,51824,133731,31734,281734,215549,210668,86895,219130,272445,471989,13224,
13137,14021,130752,321528,25940,230433,222045,222047,333490,323646,93106,9121,8372,121024,41038,351210,121706
,14032,48137,321427,272414,252891,367310,376394,38296,242860,125476,8106,54274,46574,26508,26509,378812,
231931,145002,51767,155658,71410,61253,65550,38287,68966,186165,119297,275530,47130,13256,99061,377570,103157,
133512,145397,335155,71145,320662,13131,18548,43984,18553,18552,141544,422592,227742,228357,142003,128063,
261294,262929,205815,349868,297981,451367,368371,191531,117314,63467,8605,452105,212948,278524,278532,41115,
450612,91346,54231,271154,54703,89897,87716,263017,6985,4280,58180,93680,62053,348469,119325,272181,107664,
23884,276808,320405,12298,12978,307233,277194,295329,65473,120055,102179,251668,104651,98133,61666,9386,
155151,225024,41501,5892,12641,53354,12977,90193,227337,40651,40872,58984,4946,61706,61893,42129,144604,155675,
12629,4475,118222,81240,91342,35421,12299,268689,31312,153612,71488,61738,24073,24142,247185,274573,137070,
57326,97119,12297,121972,78670,91178,137033,151266,20739,28612,37348,66071,94792,56734,134945,131024,233382,
71360,97126,268759,27961,40257,65847,40337,9309,61704,61680,61762,102095,126350,54,63544,226841,8920,168486,
76928,108583,238002,230182,90198,23334,124219,98293,14968,127202,71399,188782,221358,71940,186731,230066,56755,
93838,55145,127233,123609,71276,125637,45264,87418,90053,47033,275375,252142,18007,135833,168893,272078,275285,
71252,168347,270488,103021,87495,191677,191907,13090,167008,153719,221431,227326,38402,131360,190932,230465,
229496,61233,229797,68263,177040,271352,92865,131026,140144,47409,34984,31501,15002,221767,272280,40840,273184,
55560,279060,117316,63024,5269,91981,250636,226354,41731,234676,211896,164050,63549,97392,61609,122835,25373,
12699,12623,159674,223735,61932,165981,122469,9684,9681,9797,197830,276359,46972,48077,61726,164068,92834,
15958,233995,57977,104285,122918,274604,273904,31890,42262,46325,159989,73495,62342,46880,122944,6733,159153,
144882,8339,76159,164731,137845,113144,143938,161625,18423,177528,98289,10088,217489,122906,15567,10107,42886,
160185,20223,12521,90906,273895,142208,185942,125124,10961,44614,35897,59194,91588,43850,276786,223037,123211,
272222,147003,34269,104286,132413,20257,125518,19506,228225,230424,242308,241709,217398,20008,55748,134028,
31830,16861,8319,159151,131038,89781,159832,107207,130530,6990,130165,69100,177530,270866,122539,68597,255716,
262757,39263,186769,161159,161093,205902,50575,224565,66561,63022,225639,22752,230821,166254,91669,22693,17790,
43215,25091,180284,158830,127021,192963,275708,70772,41362,42845,197339,264347,138632,51069,37059,119536,
125128,5835,11554,7852,5876,166738,65487,124890,61669,158191,122544,155480,38871,159081,160489,17710,814,
225182,118374,98579,73576,143628,47539,93472,87224,94234,13216,48078,133198,50041,223813,187244,42211,15505,
20232,6794,16127,6833,52818,129493,117506,90661,90270,92228,244173,206502,206376,224913,91594,223035,79242,
79094,83689,133283,274259,269413,31445,266431,214517,37613,269919,251936,43181,139405,65022,91236,188654,
187243,71769,186748,9794,248064,58864,142787,55712,145004,119535,86598,90268,86634,118300,7572,7552,...

The Semigroup Partition Proof: The Last Inference

772037. **$false** (0:0) [sat splitting refutation 322381,31135,511244,367817,65868,63149,61353,504668,123636,
23654,12834,23764,23766,118106,368552,99546,57496,65344,112646,133363,348639,153716,159265,98761,148710,
117176,519910,23731,584403,222823,593104,602542,270943,89534,155137,271783,133365,217863,177563,275097,368343,
425800,56830,64804,48880,577969,51824,133731,31734,281734,215549,210668,86895,219130,272445,471989,13224,
13137,14021,130752,321528,25940,230433,222045,222047,333490,323646,93106,9121,8372,121024,41038,351210,121706
,14032,48137,321427,272414,252891,367310,376394,38296,242860,125476,8106,54274,46574,26508,26509,378812,
231931,145002,51767,155658,71410,61253,65550,38287,68966,186165,119297,275530,47130,13256,99061,377570,103157,
133512,145397,335155,71145,320662,13131,18548,43984,18553,18552,141544,422592,227742,228357,142003,128063,
261294,262929,205815,349868,297981,451367,368371,191531,117314,63467,8605,452105,212948,278524,278532,41115,
450612,91346,54231,271154,54703,89897,87716,263017,6985,4280,58180,93680,62053,348469,119325,272181,107664,
23884,276808,320405,12298,12978,307233,277194,295329,65473,120055,102179,251668,104651,98133,61666,9386,
155151,225024,41501,5892,12641,53354,12977,90193,227337,40651,40872,58984,4946,61706,61893,42129,144604,155675,
12629,4475,118222,81240,91342,35421,12299,268689,31312,153612,71488,61738,24073,24142,247185,274573,137070,
57326,97119,12297,121972,78670,91178,137033,151266,20739,28612,37348,66071,94792,56734,134945,131024,233382,
71360,97126,268759,27961,40257,65847,40337,9309,61704,61680,61762,102095,126350,54,63544,226841,8920,168486,
76928,108583,238002,230182,90198,23334,124219,98293,14968,127202,71399,188782,221358,71940,186731,230066,56755,
93838,55145,127233,123609,71276,125637,45264,87418,90053,47033,275375,252142,18007,135833,168893,272078,275285,
71252,168347,270488,103021,87495,191677,191907,13090,167008,153719,221431,227326,38402,131360,190932,230465,
229496,61233,229797,68263,177040,271352,92865,131026,140144,47409,34984,31501,15002,221767,272280,40840,273184,
55560,279060,117316,63024,5269,91981,250636,226354,41731,234676,211896,164050,63549,97392,61609,122835,25373,
12699,12623,159674,223735,61932,165981,122469,9684,9681,9797,197830,276359,46972,48077,61726,164068,92834,
15958,233995,57977,104285,122918,274604,273904,31890,42262,46325,159989,73495,62342,46880,122944,6733,159153,
144882,8339,76159,164731,137845,113144,143938,161625,18423,177528,98289,10088,217489,122906,15567,10107,42886,
160185,20223,12521,90906,273895,142208,185942,125124,10961,44614,35897,59194,91588,43850,276786,223037,123211,
272222,147003,34269,104286,132413,20257,125518,19506,228225,230424,242308,241709,217398,20008,55748,134028,
31830,16861,8319,159151,131038,89781,159832,107207,130530,6990,130165,69100,177530,270866,122539,68597,255716,
262757,39263,186769,161159,161093,205902,50575,224565,66561,63022,225639,22752,230821,166254,91669,22693,17790,
43215,25091,180284,158830,127021,192963,275708,70772,41362,42845,197339,264347,138632,51069,37059,119536,
125128,5835,11554,7852,5876,166738,65487,124890,61669,158191,122544,155480,38871,159081,160489,17710,814,
225182,118374,98579,73576,143628,47539,93472,87224,94234,13216,48078,133198,50041,223813,187244,42211,15505,
20232,6794,16127,6833,52818,129493,117506,90661,90270,92228,244173,206502,206376,224913,91594,223035,79242,
79094,83689,133283,274259,269413,31445,266431,214517,37613,269919,251936,43181,139405,65022,91236,188654,
187243,71769,186748,9794,248064,58864,142787,55712,145004,119535,86598,90268,86634,118300,7572,7552,...

This is less than 1/20th of the inference

Future

There is more to AVATAR:

- Giles Reger, Martin Suda, Andrei Voronkov. Playing with AVATAR. CADE 2015
- Giles Reger, Dmitry Tishkovsky, Andrei Voronkov. Cooperating Proof Attempts. CADE 2015
- SMT solver instead of SAT solver (VampireZ3)
- Arbitrary theory reasoning with quantifiers

First-Order Theorem Proving in Rigorous Systems Engineering

Interpolation

Laura Kovács and Andrei Voronkov

TU Wien and University of Manchester

Outline

Interpolation

Theorem

Let R, B be closed formulas and let $R \vdash B$.

Then there exists a formula I such that

1. $R \vdash I$ *and* $I \vdash B$*;*
2. *every symbol of I occurs both in R and B;*

Interpolation

Theorem

Let R, B be closed formulas and let $R \vdash B$.

Then there exists a formula I such that

1. *$R \vdash I$ and $I \vdash B$;*
2. *every symbol of I occurs both in R and B;*

Any formula I with this property is called an interpolant of R and B.

Essentially, an interpolant is a formula that is

1. intermediate in power between R and B;
2. Uses only common symbols of R and B.

Interpolation has many uses in verification.

Interpolation

Theorem

Let R, B be closed formulas and let $R \vdash B$.

Then there exists a formula I such that

1. *$R \vdash I$ and $I \vdash B$;*
2. *every symbol of I occurs both in R and B;*

Any formula I with this property is called an interpolant of R and B.

Essentially, an interpolant is a formula that is

1. intermediate in power between R and B;
2. Uses only common symbols of R and B.

Interpolation has many uses in verification.

When we deal with refutations rather than proofs and have an unsatisfiable set $\{R, B\}$, it is convenient to use reverse interpolants of R and B, that is, a formula I such that

1. $R \vdash I$ and $\{I, B\}$ is unsatisfiable;
2. every symbol of I occurs both in R and B;

Interpolation Through Colors

- There are three colors: red, blue and grey.

Interpolation Through Colors

- There are three colors: red, blue and grey.
- Each symbol (function or predicate) is colored in exactly one of these colors.

Interpolation Through Colors

- There are three colors: red, blue and grey.
- Each symbol (function or predicate) is colored in exactly one of these colors.
- We have two formulas: R and B.
- Each symbol in R is either red or grey.
- Each symbol in B is either blue or grey.

Interpolation Through Colors

- There are three colors: red, blue and grey.
- Each symbol (function or predicate) is colored in exactly one of these colors.
- We have two formulas: R and B.
- Each symbol in R is either red or grey.
- Each symbol in B is either blue or grey.
- We know that $\vdash R \to B$.
- Our goal is to find a grey formula I such that:
  1. $\vdash R \to I$;
  2. $\vdash I \to B$.

Interpolation with Theories

- Theory T: any set of closed green formulas.
- $C_1, \ldots, C_n \vdash_T C$ denotes that the formula $C_1 \wedge \ldots \wedge C_1 \rightarrow C$ holds in all models of T.
- Interpreted symbols: symbols occurring in T.
- Uninterpreted symbols: all other symbols.

Interpolation with Theories

- Theory T: any set of closed green formulas.
- $C_1, \ldots, C_n \vdash_T C$ denotes that the formula $C_1 \wedge \ldots \wedge C_1 \rightarrow C$ holds in all models of T.
- Interpreted symbols: symbols occurring in T.
- Uninterpreted symbols: all other symbols.

Theorem

Let R, B be formulas and let $R \vdash_T B$.

Then there exists a formula I such that

1. *$R \vdash_T I$ and $I \vdash B$;*
2. *every uninterpreted symbol of I occurs both in R and B;*
3. *every interpreted symbol of I occurs in B.*

Likewise, there exists a formula I such that

1. *$R \vdash I$ and $I \vdash_T B$;*
2. *every uninterpreted symbol of I occurs both in R and B;*
3. *every interpreted symbol of I occurs in R.*

Local Derivations

A derivation is called local (well-colored) if each inference in it

$$\frac{C_1 \quad \cdots \quad C_n}{C}$$

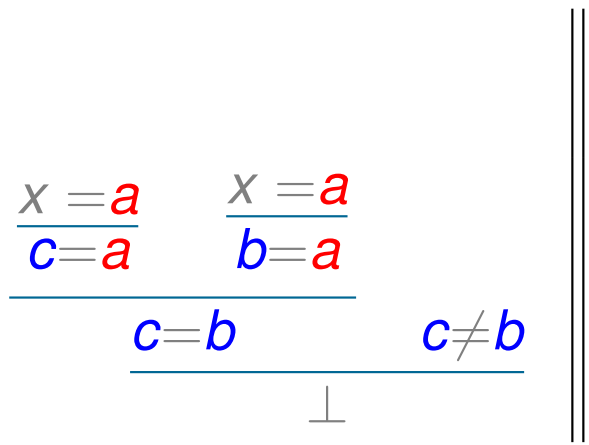
either has no blue symbols or has no red symbols.
That is, one cannot mix blue and red in the same inference.

Local Derivations: Example

- $R := \forall x(x = a)$
- $B := c = b$
- Interpolant: $\forall x \forall y(x = y)$ (note: universally quantified!)

Local Derivations: Example

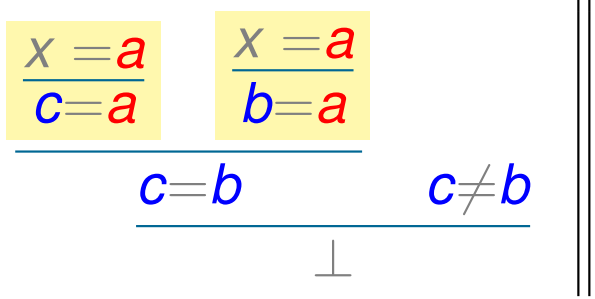
- $R := \forall x(x = a)$
- $B := c = b$
- Interpolant: $\forall x \forall y(x = y)$ (note: universally quantified!)



Local Derivations: Example

- $R := \forall x(x = a)$
- $B := c = b$
- Interpolant: $\forall x \forall y(x = y)$ (note: universally quantified!)

Non-local proof



Local Derivations: Example

- $R := \forall x(x = a)$
- $B := c = b$
- Interpolant: $\forall x \forall y(x = y)$ (note: universally quantified!)

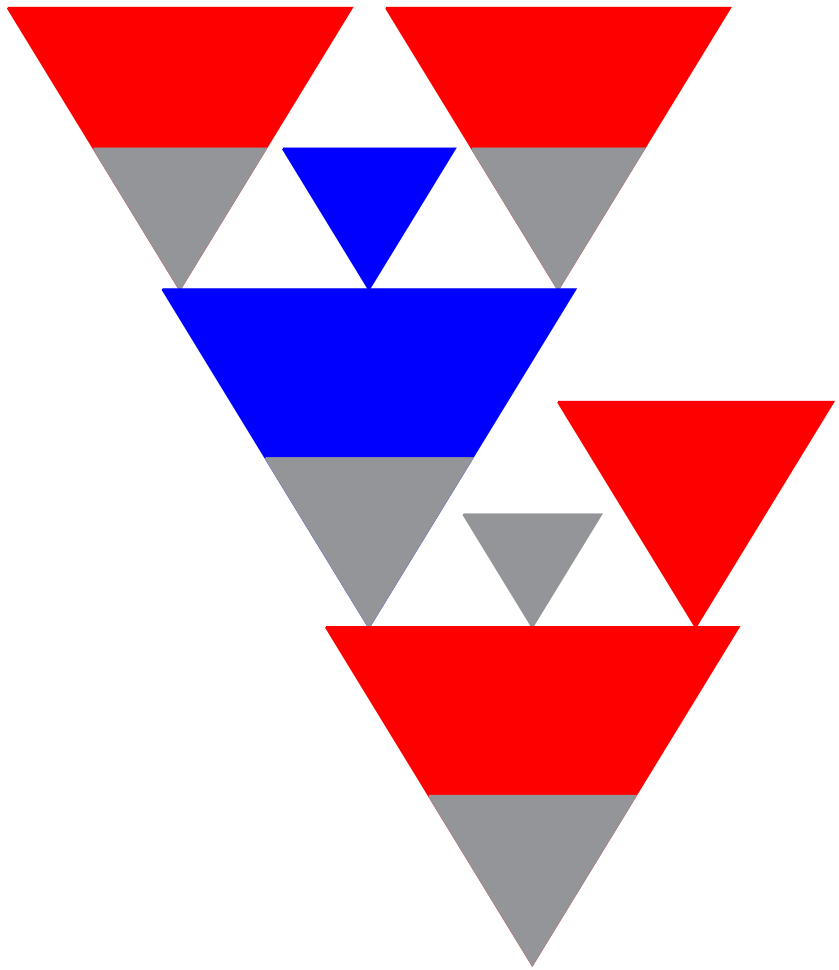
Non-local proof

$$\dfrac{\dfrac{\dfrac{x=a}{c=a} \quad \dfrac{x=a}{b=a}}{c=b} \qquad c \neq b}{\bot}$$

Local Proof

$$\dfrac{\dfrac{\dfrac{x=a \quad y=a}{x=y} \qquad c \neq b}{y \neq b}}{\bot}$$

Shape of a local derivation

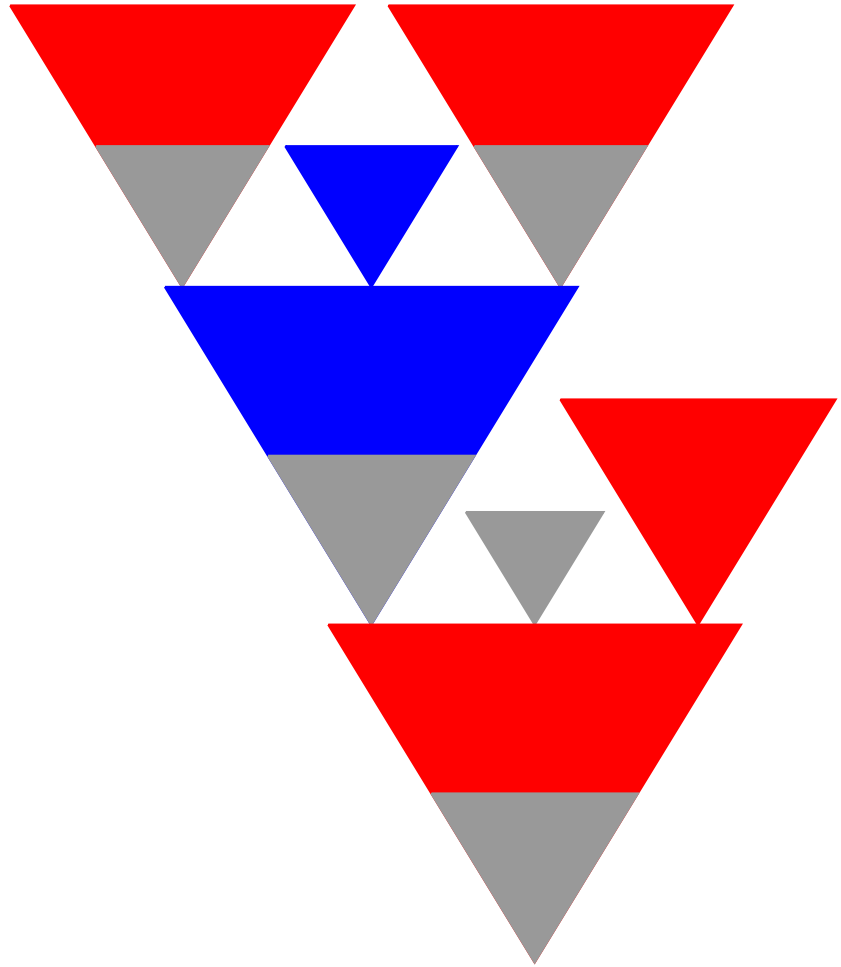


Symbol Eliminating Inference

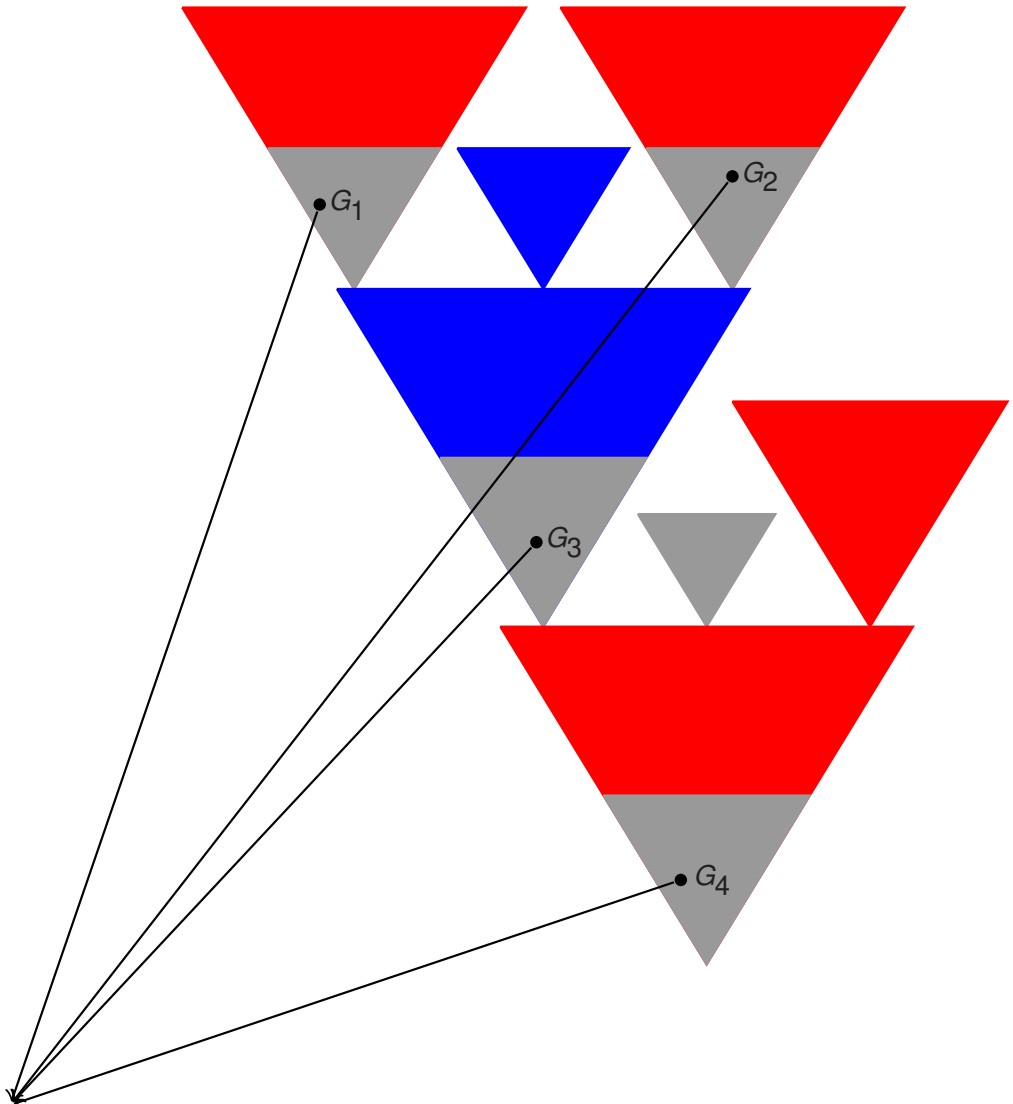
- At least one of the premises is not grey.
- The conclusion is grey.

$$\dfrac{\dfrac{\boxed{\dfrac{x = a \quad y = a}{x = y}} \qquad c \neq b}{\boxed{\dfrac{y \neq b}{\bot}}}}{}$$

Extracting Interpolants from Local Proofs



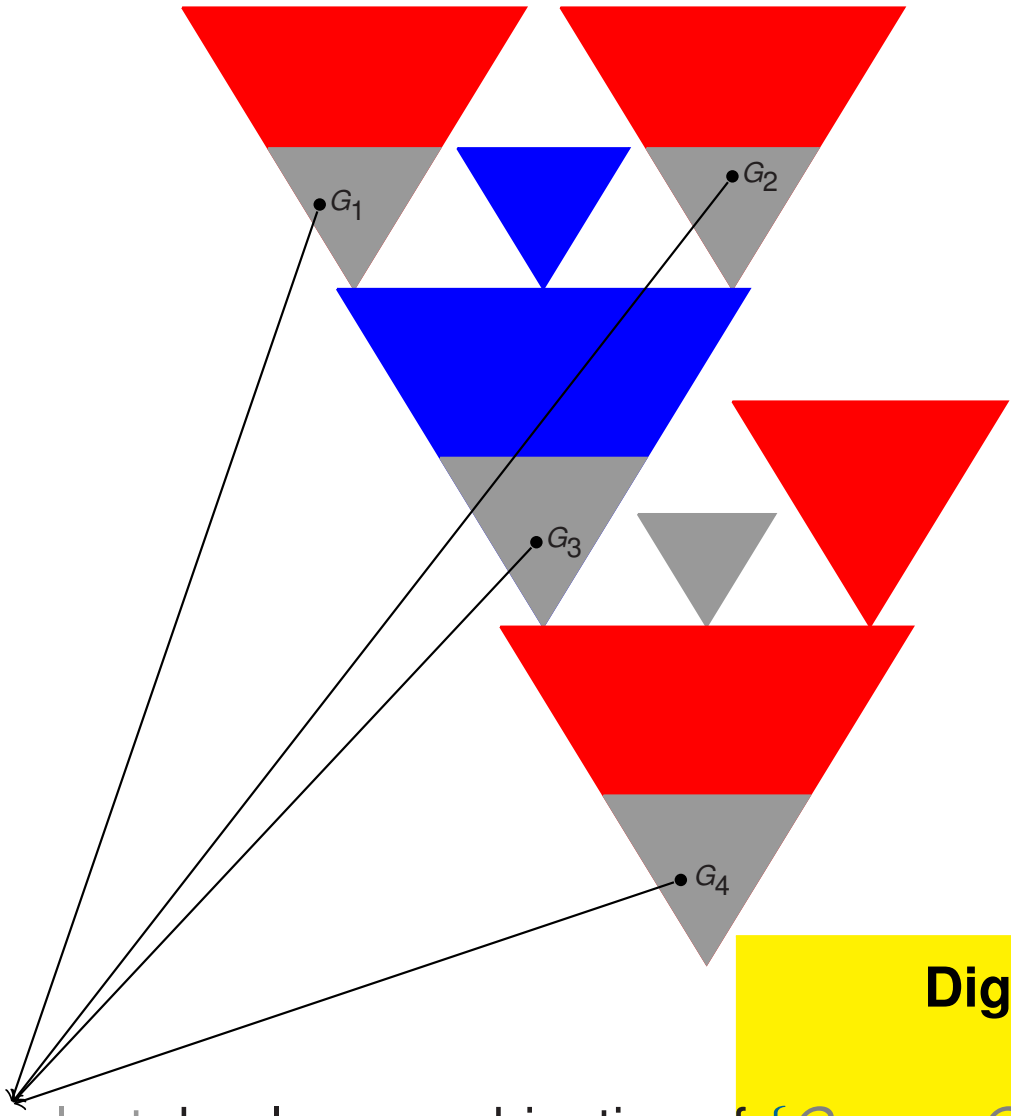
Extracting Interpolants from Local Proofs



Interpolant: boolean combination of $\{G_1, \ldots, G_4\}$

[McMillan05, KV09]

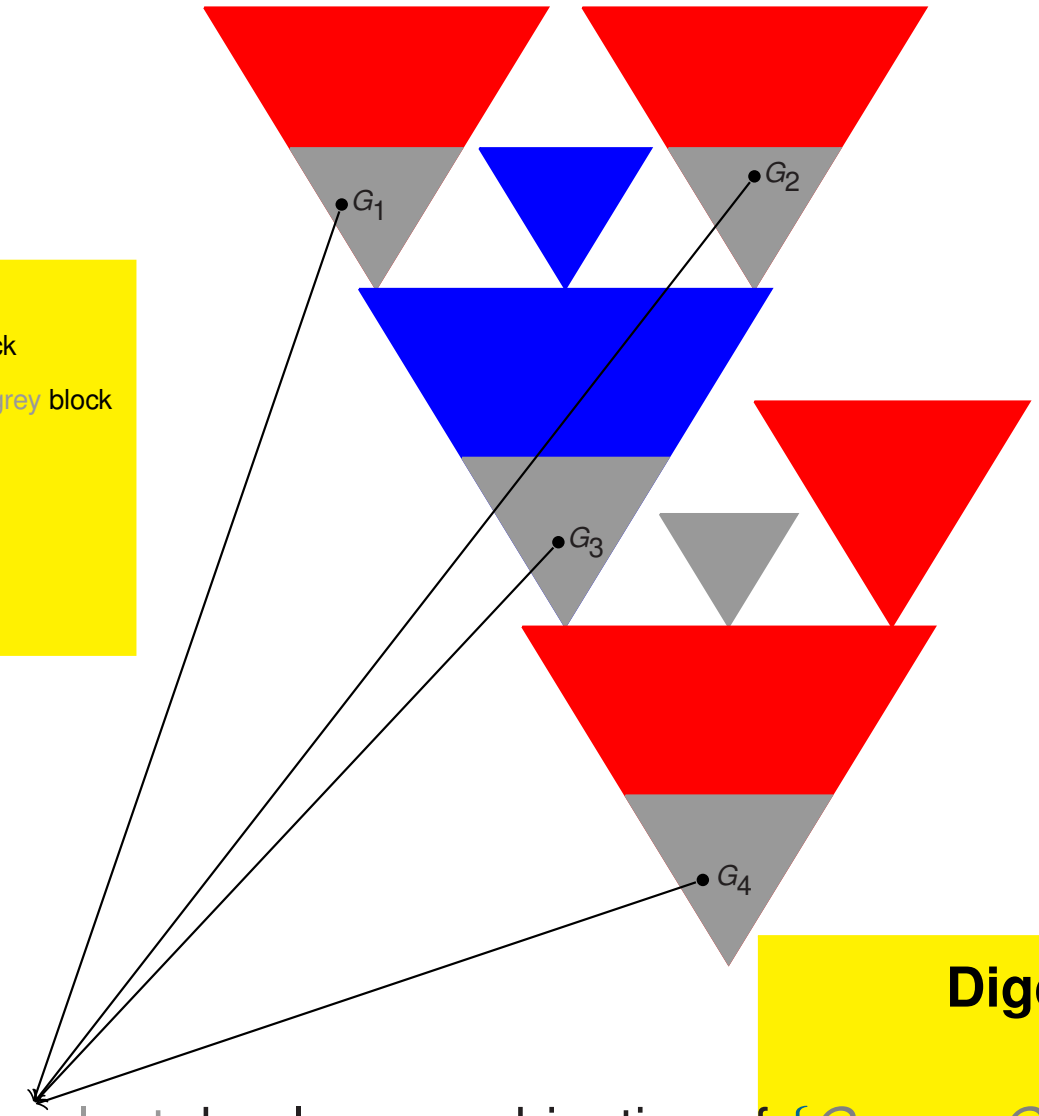
Extracting Interpolants from Local Proofs



Digest

Interpolant: boolean combination of $\{G_1, \ldots, G_4\}$

Extracting Interpolants from Local Proofs



G is in the digest:

- comes from a red block
- followed by a blue or grey block

Digest

Interpolant: boolean combination of $\{G_1, \ldots, G_4\}$

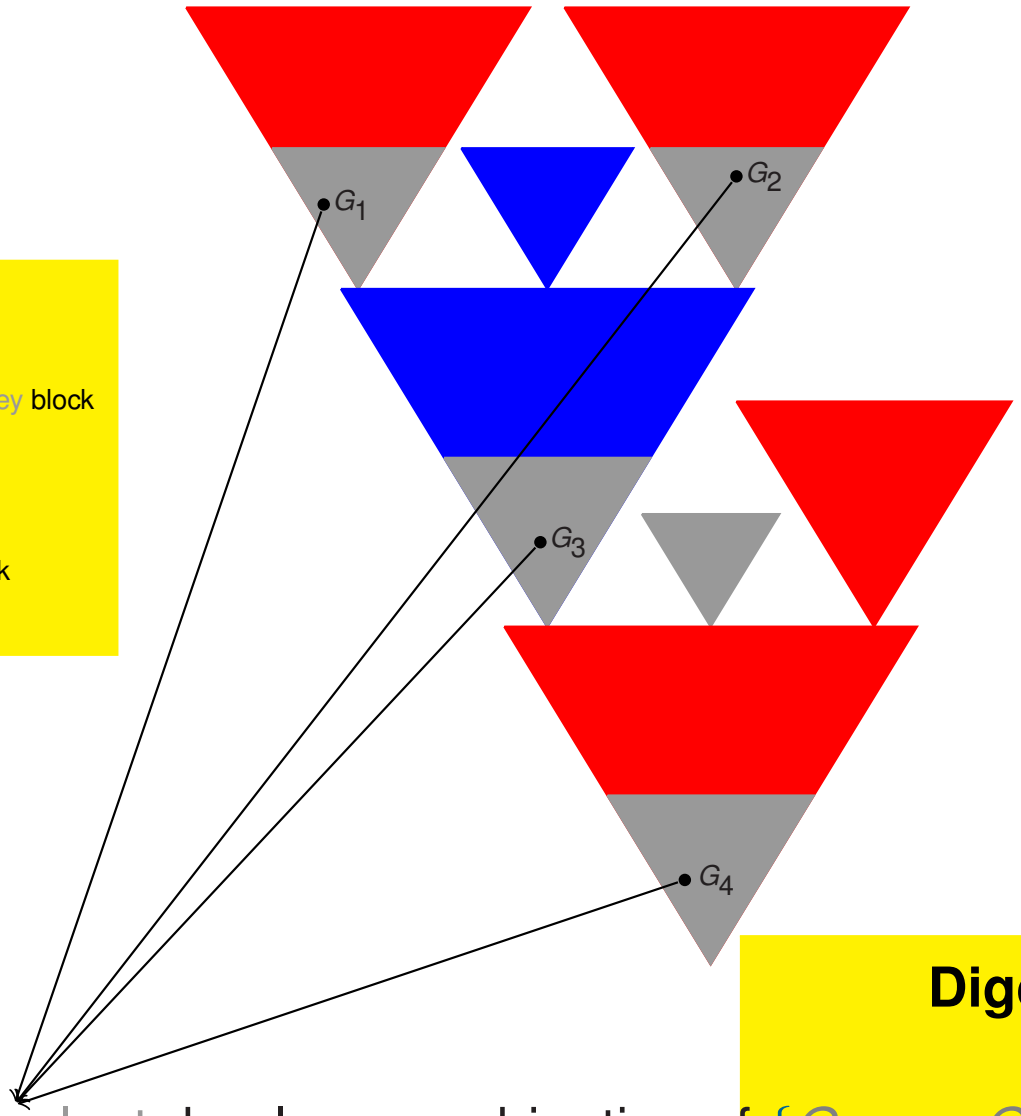
Extracting Interpolants from Local Proofs

G is in the digest:

- comes from a red block
- followed by a blue or grey block

or

- comes from a blue block
- followed by a red



Digest

Interpolant: boolean combination of $\{G_1, \ldots, G_4\}$

Extracting Interpolants from Local Proofs

Theorem

Let Π be a local refutation. Then one can extract from Π in linear time a reverse interpolant I of R and B. This interpolant is ground if all formulas in Π are ground.

Extracting Interpolants from Local Proofs

Theorem

Let Π be a local refutation. Then one can extract from Π in linear time a reverse interpolant I of R and B. This interpolant is ground if all formulas in Π are ground. This reverse interpolant is a boolean combination of conclusions of symbol-eliminating inferences of Π.

Extracting Interpolants from Local Proofs

Theorem

Let Π be a local refutation. Then one can extract from Π in linear time a reverse interpolant I of R and B. This interpolant is ground if all formulas in Π are ground. This reverse interpolant is a boolean combination of conclusions of symbol-eliminating inferences of Π.

What is remarkable in this theorem:

- No restriction on the calculus (only soundness required) – can be used with theories.
- Can generate interpolants in theories where no good interpolation algorithms exist.

Interpolation: Examples in Vampire

Our running example:

Local proof and interpolant: `vampire interpol1.p`

Non-local proof: `vampire interpol2.p`

Interpolation: Examples in Vampire

```
fof(fA,axiom, q(f(a)) & ~q(f(b)) ).
fof(fB,conjecture, ?[V]: V != c).
```

Non-local proof: `vampire interpol4.p`

Interpolation: Examples in Vampire

```
% request to generate an interpolant
vampire(option,show_interpolant,on).
% symbol coloring
vampire(symbol,predicate,q,1,left).
vampire(symbol,function,f,1,left).
vampire(symbol,function,a,0,left).
vampire(symbol,function,b,0,left).
vampire(symbol,function,c,0,right).
% formula R
vampire(left_formula).
  fof(fA,axiom, q(f(a)) & ~q(f(b)) ).
vampire(end_formula).
% formula B
vampire(right_formula).
  fof(fB,conjecture, ?[V]: V != c).
vampire(end_formula).
```

Local proof and interpolant: `vampire interpol3.p`

Outline

Localizing Proofs

Task: eliminate non-local inferences

Localizing Proofs

Task: eliminate non-local inferences

Idea: quantify away colored symbols

colored symbols replaced by logical variables.

Localizing Proofs

Task: eliminate non-local inferences

Idea: quantify away colored symbols

$\downarrow$

colored symbols replaced by logical variables.

Given $R(a) \vdash B$ where a is an uninterpreted constant not occurring in B.

Then, $R(a) \vdash (\exists x) R(x)$ and $(\exists x) R(x) \vdash B$.

Localizing Proofs

Task: eliminate non-local inferences

Idea: quantify away colored symbols

$\downarrow$

colored symbols replaced by logical variables.

Given $R(a) \vdash B$ where a is an uninterpreted constant not occurring in B.

Then, $R(a) \vdash (\exists x) R(x)$ and $(\exists x) R(x) \vdash B$.

$$\dfrac{\dfrac{R_1(a)}{R_2(a)} \quad B}{A} \qquad \left\| \qquad \dfrac{\dfrac{R_1(a)}{(\exists x) R_2(x)} \quad B}{A} \right.$$

Localizing Proofs

Task: eliminate non-local inferences

Idea: quantify away colored symbols

$\downarrow$

colored symbols replaced by logical variables.

Cons: Comes at the cost of using extra quantifiers.

Given $R(a) \vdash B$ where a is an uninterpreted constant not occurring in B.

Then, $R(a) \vdash (\exists x) R(x)$ and $(\exists x) R(x) \vdash B$.

$$\dfrac{\dfrac{R_1(a)}{R_2(a)} \quad B}{A} \qquad \left\| \qquad \dfrac{\dfrac{R_1(a)}{(\exists x) R_2(x)} \quad B}{A} \right.$$

Localizing Proofs

Task: eliminate non-local inferences

Idea: quantify away colored symbols

$\downarrow$

colored symbols replaced by logical variables.

Cons: Comes at the cost of using extra quantifiers.

But we can **minimise** the number of quantifiers in the interpolant.

Given $R(a) \vdash B$ where a is an uninterpreted constant not occurring in B.

Then, $R(a) \vdash (\exists x) R(x)$ and $(\exists x) R(x) \vdash B$.

$$\dfrac{\dfrac{R_1(a)}{R_2(a)} \quad B}{A} \qquad \left\| \qquad \dfrac{\dfrac{R_1(a)}{(\exists x) R_2(x)} \quad B}{A} \right.$$

Outline

Minimizing Interpolants

Our Interest: Small Interpolants

- in size;
- in weight;
- in the number of quantifiers;
- ...

Minimizing Interpolants

Our Interest: Small Interpolants

- in size;
- in weight;
- in the number of quantifiers;
- ...

Given $\vdash R \to B$, find a grey formula I:

. $\vdash R \to I$;

. $\vdash I \to B$;

. I is small.

Minimizing Interpolant

Task: minimise interpolants = minimise digest

Minimizing Interpolant

Task: minimise interpolants = minimise digest

Minimizing Interpolant

Task: minimise interpolants = minimise digest

Hercule Poirot:

It is the little GREY CELLS*, mon ami, on which one must rely.*

Mon Dieu, mon ami, but use your little GREY CELLS*!*

Minimizing Interpolant

Task: minimise interpolants = minimise digest

Minimizing Interpolant

Task: minimise interpolants = minimise digest

Idea: Change the grey areas of the local proof

Minimizing Interpolant

Task: minimise interpolants = minimise digest

Idea: Change the grey areas of the local proof

Slicing off formulas

$$\dfrac{A_1 \quad \cdots \quad A_n \qquad \dfrac{A_{n+1} \quad \cdots \quad A_m}{A}}{A_0} \xrightarrow{\text{slicing off } A} \dfrac{A_1 \quad \cdots \quad A_n \quad A_{n+1} \quad \cdots \quad A_m}{A_0}$$

Minimizing Interpolant

Task: minimise interpolants = minimise digest

Idea: Change the grey areas of the local proof

Slicing off formulas

$$\dfrac{A_1 \quad \cdots \quad A_n \qquad \dfrac{A_{n+1} \quad \cdots \quad A_m}{A}}{A_0} \xrightarrow{\text{slicing off } A} \dfrac{A_1 \quad \cdots \quad A_n \quad A_{n+1} \quad \cdots \quad A_m}{A_0}$$

If A is grey: Grey slicing

Minimizing Interpolant

Task: minimise interpolants = minimise digest

Idea: Change the grey areas of the local proof

Slicing off formulas

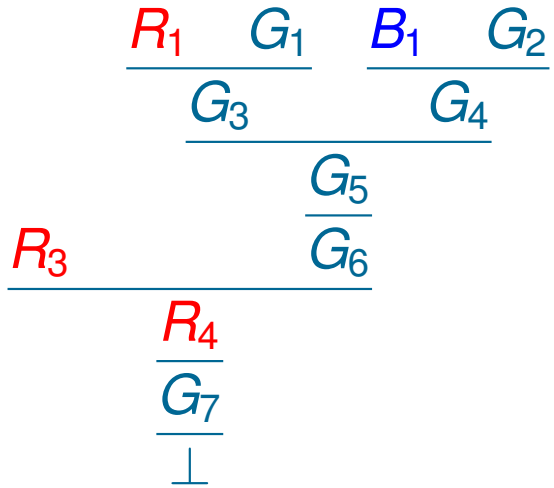
Minimizing Interpolant

Task: minimise interpolants = minimise digest

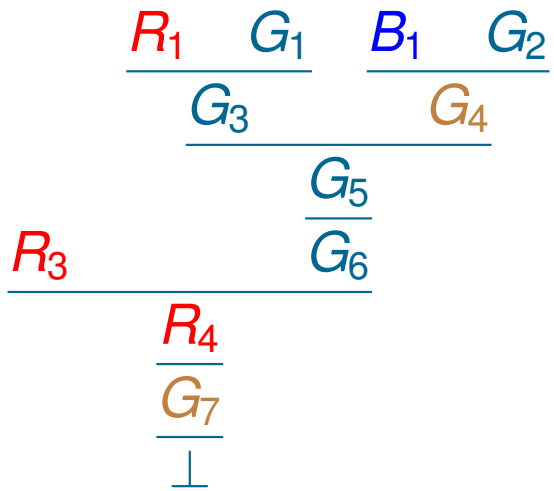
Idea: Change the grey areas of the local proof, but preserve locality!

Slicing off formulas

Minimizing Interpolant



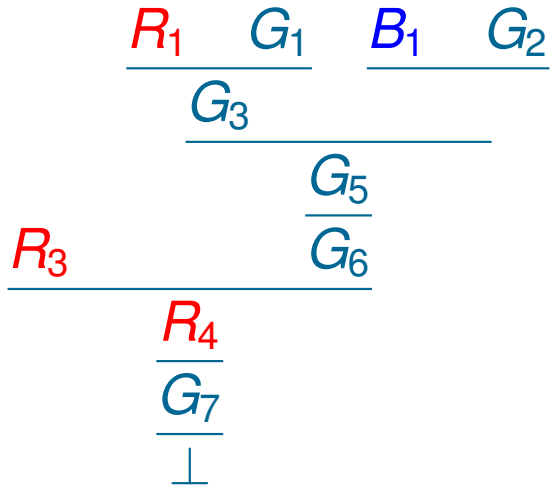
Minimizing Interpolant



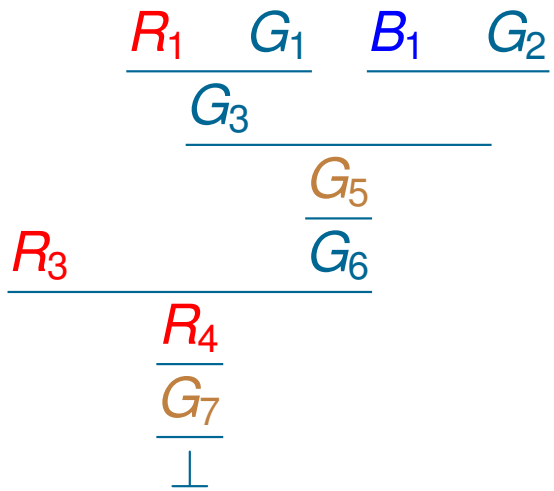
Digest: $\{G_4, G_7\}$

Reverse interpolant: $G_4 \rightarrow G_7$

Minimizing Interpolant



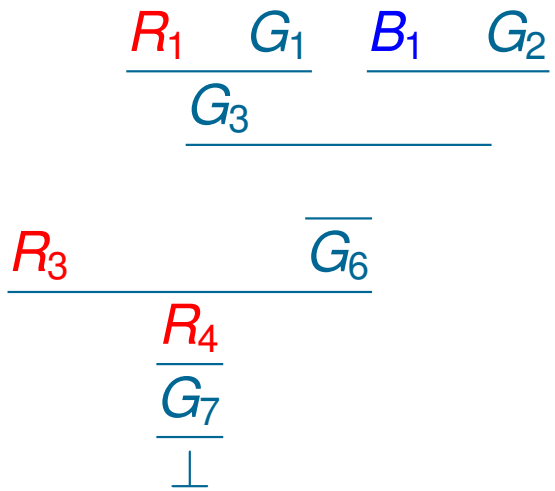
Minimizing Interpolant



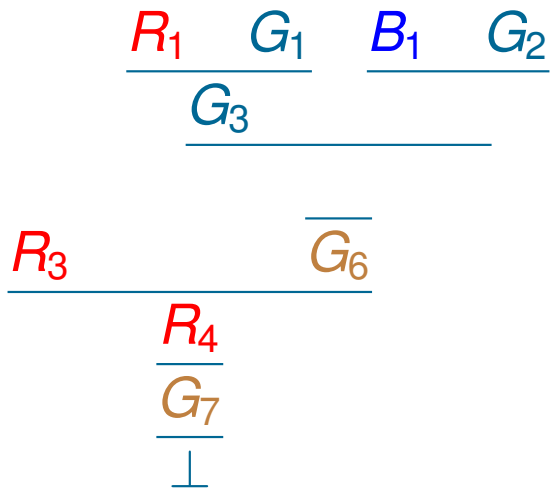
Digest: $\{G_5, G_7\}$

Reverse interpolant: $G_5 \rightarrow G_7$

Minimizing Interpolant



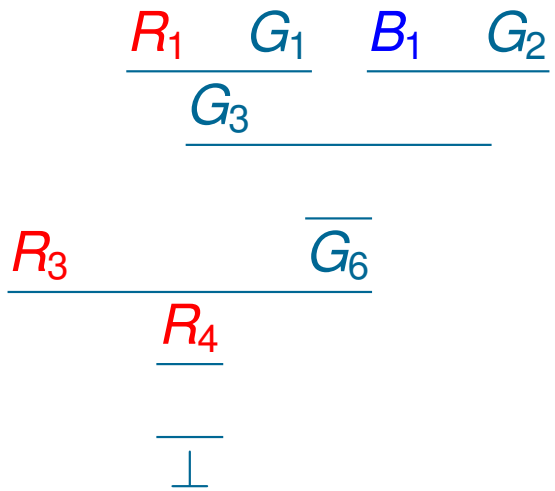
Minimizing Interpolant



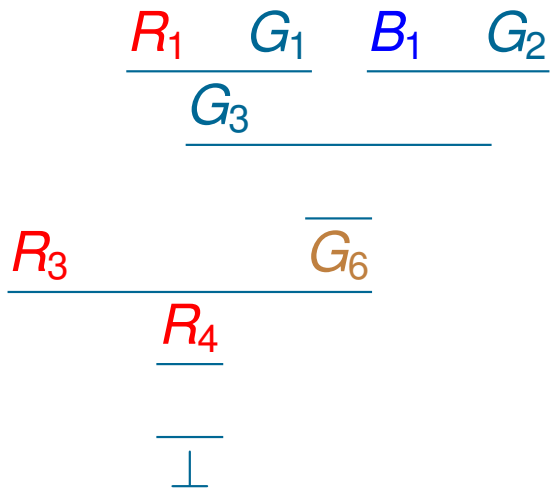
Digest: $\{G_6, G_7\}$

Reverse interpolant: $G_6 \rightarrow G_7$

Minimizing Interpolant



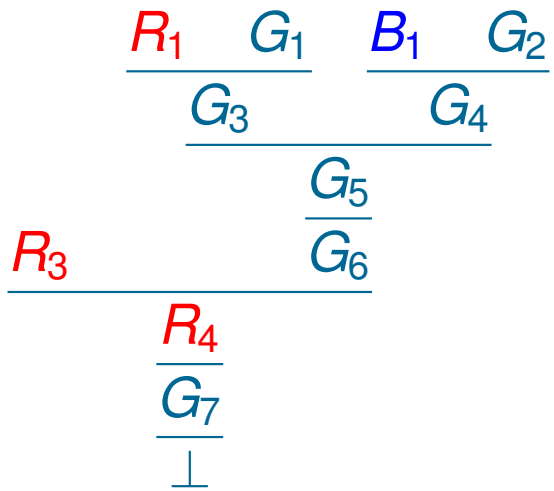
Minimizing Interpolant



Digest: $\{G_6\}$

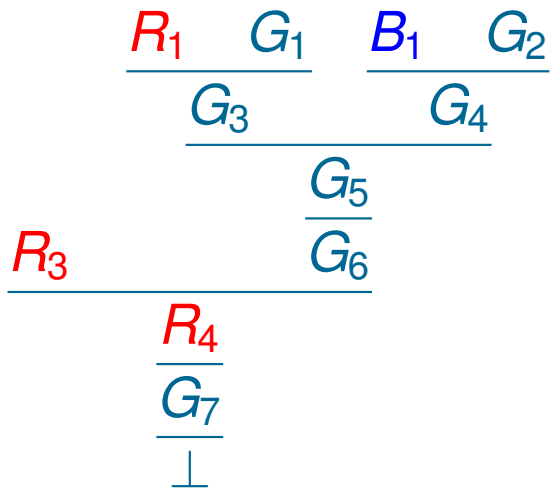
Reverse interpolant: $\neg G_6$

Minimizing Interpolant



Note that the interpolant has changed from $G_4 \rightarrow G_7$ to $\neg G_6$.

Minimizing Interpolant



Note that the interpolant has changed from $G_4 \rightarrow G_7$ to $\neg G_6$.

- There is no obvious logical relation between $G_4 \rightarrow G_7$ and $\neg G_6$, for example none of these formulas implies the other one;
- These formulas may even have no common atoms or no common symbols.

Minimizing Interpolant

If grey slicing gives us very different interpolants, we can use it for finding small interpolants.

Problem: if the proof contains n grey formulas, the number of possible different slicing off transformations is 2^n.

Minimizing Interpolant

If grey slicing gives us very different interpolants, we can use it for finding small interpolants.

Problem: if the proof contains n grey formulas, the number of possible different slicing off transformations is 2^n.

Minimizing Interpolant

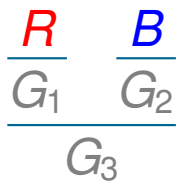
Solution:

- encode all sequences of transformations as an instance of SAT
- solutions encode all slicing off transformations

Minimizing Interpolant

Solution:

- encode all sequences of transformations as an instance of SAT
- solutions encode all slicing off transformations



Minimizing Interpolant

Solution:

- encode all sequences of transformations as an instance of SAT
- solutions encode all slicing off transformations

G_3, and at most one of G_1, G_2 can be sliced off.

$$\dfrac{\dfrac{R}{G_1}\quad\dfrac{B}{G_2}}{G_3}$$

Minimizing Interpolant

Solution:

- encode all sequences of transformations as an instance of SAT
- solutions encode all slicing off transformations

$$\dfrac{\dfrac{R}{G_1}\quad\dfrac{B}{G_2}}{G_3}$$

Some predicates on grey formulas:

- sliced(G): G was sliced off;
- red(G): the trace of G contains a red formula;
- blue(G): the trace of G contains a blue formula;
- grey(G): the trace of G contains only grey formulas;
- digest(G): G belongs to the digest.

Minimizing Interpolant

Solution:

- encode all sequences of transformations as an instance of SAT
- solutions encode all slicing off transformations

$$\dfrac{\dfrac{R}{G_1} \quad \dfrac{B}{G_2}}{G_3}$$

$$\neg \text{sliced}(G_1) \rightarrow \text{grey}(G_1)$$
$$\text{sliced}(G_1) \rightarrow \text{red}(G_1)$$

Some predicates on grey formulas:

- sliced(G): G was sliced off;
- red(G): the trace of G contains a red formula;
- blue(G): the trace of G contains a blue formula;
- grey(G): the trace of G contains only grey formulas;
- digest(G): G belongs to the digest.

Minimizing Interpolant

Solution:

- encode all sequences of transformations as an instance of SAT
- solutions encode all slicing off transformations

$$\dfrac{\dfrac{R}{G_1} \quad \dfrac{B}{G_2}}{G_3}$$

Some predicates on grey formulas:

- sliced(G): G was sliced off;
- red(G): the trace of G contains a red formula;
- blue(G): the trace of G contains a blue formula;
- grey(G): the trace of G contains only grey formulas;
- digest(G): G belongs to the digest.

$$\neg \text{sliced}(G_3) \rightarrow \text{grey}(G_3)$$
$$\text{sliced}(G_3) \rightarrow (\text{grey}(G_3) \leftrightarrow \text{grey}(G_1) \wedge \text{grey}(G_2))$$
$$\text{sliced}(G_3) \rightarrow (\text{red}(G_3) \leftrightarrow \text{red}(G_1) \vee \text{red}(G_2))$$
$$\text{sliced}(G_3) \rightarrow (\text{blue}(G_3) \leftrightarrow \text{blue}(G_1) \vee \text{blue}(G_2))$$

Minimizing Interpolant

Solution:

- encode all sequences of transformations as an instance of SAT
- solutions encode all slicing off transformations

$$\dfrac{\dfrac{R}{G_1} \quad \dfrac{B}{G_2}}{G_3}$$

Some predicates on grey formulas:

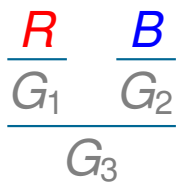
- sliced(G): G was sliced off;
- red(G): the trace of G contains a red formula;
- blue(G): the trace of G contains a blue formula;
- grey(G): the trace of G contains only grey formulas;
- digest(G): G belongs to the digest.

$$\text{digest}(G_1) \rightarrow \neg\text{sliced}(G_1)$$

Minimizing Interpolant

Solution:

- encode all sequences of transformations as an instance of SAT
- solutions encode all slicing off transformations



Some predicates on grey formulas:

- sliced(G): G was sliced off;
- red(G): the trace of G contains a red formula;
- blue(G): the trace of G contains a blue formula;
- grey(G): the trace of G contains only grey formulas;
- digest(G): G belongs to the digest.

$$\neg \text{sliced}(G_1) \rightarrow \text{grey}(G_1)$$
$$\text{sliced}(G_1) \rightarrow \text{red}(G_1)$$
$$\neg \text{sliced}(G_3) \rightarrow \text{grey}(G_3)$$
$$\text{sliced}(G_3) \rightarrow (\text{grey}(G_3) \leftrightarrow \text{grey}(G_1) \wedge \text{grey}(G_2))$$
$$\text{sliced}(G_3) \rightarrow (\text{red}(G_3) \leftrightarrow \text{red}(G_1) \vee \text{red}(G_2))$$
$$\text{sliced}(G_3) \rightarrow (\text{blue}(G_3) \leftrightarrow \text{blue}(G_1) \vee \text{blue}(G_2))$$
$$\text{digest}(G_1) \rightarrow \neg \text{sliced}(G_1)$$
$$\ldots$$

Minimizing Interpolant

Solution:

- encode all sequences of transformations as an instance of SAT
- solutions encode all slicing off transformations

$$\dfrac{\dfrac{R}{G_1} \quad \dfrac{B}{G_2}}{G_3}$$

Express digest(G)

Minimizing Interpolant

Solution:

- encode all sequences of transformations as an instance of SAT
- solutions encode all slicing off transformations

$$\dfrac{\dfrac{R}{G_1} \quad \dfrac{B}{G_2}}{G_3}$$

Express digest(G)

by considering the possibilities:

- G comes from a red/ blue/ grey formula
- G is followed by a red/ blue/ grey formula

Minimizing Interpolant

Solution:

- encode all sequences of transformations as an instance of SAT
- solutions encode all slicing off transformations

$$\dfrac{\dfrac{R}{G_1}\quad\dfrac{B}{G_2}}{G_3}$$

Express digest(G)

by considering the possibilities:

- G comes from a red/ blue/ grey formula

  rc(G)/bc(G)

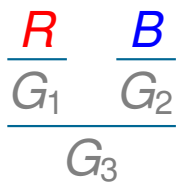
- G is followed by a red/ blue/ grey formula

  bf(G)/rf(G)

Minimizing Interpolant

Solution:

- encode all sequences of transformations as an instance of SAT
- solutions encode all slicing off transformations



Express digest(G)

by considering the possibilities:

- G comes from a red/ blue/ grey formula

  rc(G)/bc(G)

- G is followed by a red/ blue/ grey formula

  bf(G)/rf(G)

$$\text{digest}(G_3) \leftrightarrow (\text{rc}(G_3) \wedge \text{rf}(G_3)) \vee (\text{bc}(G_3) \wedge \text{bf}(G_3))$$
$$\text{rc}(G_3) \leftrightarrow (\neg\text{sliced}(G_3) \wedge (\text{red}(G_1) \vee \text{red}(G_2))$$

Minimizing Interpolant

Solution:

- encode all sequences of transformations as an instance of SAT
- solutions encode all slicing off transformations

$$\dfrac{\dfrac{R}{G_1} \quad \dfrac{B}{G_2}}{G_3}$$

Express digest(G)

by considering the possibilities:

- G comes from a red/ blue/ grey formula

  rc(G)/bc(G)

- G is followed by a red/ blue/ grey formula

  bf(G)/rf(G)

$$\neg\text{sliced}(G_1) \rightarrow \text{grey}(G_1)$$
$$\text{sliced}(G_1) \rightarrow \text{red}(G_1)$$
$$\neg\text{sliced}(G_3) \rightarrow \text{grey}(G_3)$$
$$\text{sliced}(G_3) \rightarrow (\text{grey}(G_3) \leftrightarrow \text{grey}(G_1) \wedge \text{grey}(G_2))$$
$$\text{sliced}(G_3) \rightarrow (\text{red}(G_3) \leftrightarrow \text{red}(G_1) \vee \text{red}(G_2))$$
$$\text{sliced}(G_3) \rightarrow (\text{blue}(G_3) \leftrightarrow \text{blue}(G_1) \vee \text{blue}(G_2))$$
$$\text{digest}(G_1) \rightarrow \neg\text{sliced}(G_1)$$

$$\text{digest}(G_3) \leftrightarrow (\text{rc}(G_3) \wedge \text{rf}(G_3)) \vee (\text{bc}(G_3) \wedge \text{bf}(G_3))$$
$$\text{rc}(G_3) \leftrightarrow (\neg\text{sliced}(G_3) \wedge (\text{red}(G_1) \vee \text{red}(G_2))$$
$$\ldots$$

Minimizing Interpolant

Solution:

- encode all sequences of transformations as an instance of SAT
- solutions encode all slicing off transformations

$$\frac{\dfrac{R}{G_1} \quad \dfrac{B}{G_2}}{G_3}$$

Express digest(G)

by considering the possibilities:

- G comes from a red/ blue/ grey formula

  rc(G)/bc(G)

- G is followed by a red/ blue/ grey formula

  bf(G)/rf(G)

$$\neg \text{sliced}(G_1) \rightarrow \text{grey}(G_1)$$
$$\text{sliced}(G_1) \rightarrow \text{red}(G_1)$$
$$\neg \text{sliced}(G_3) \rightarrow \text{grey}(G_3)$$
$$\text{sliced}(G_3) \rightarrow (\text{grey}(G_3) \leftrightarrow \text{grey}(G_1) \wedge \text{grey}(G_2))$$
$$\text{sliced}(G_3) \rightarrow (\text{red}(G_3) \leftrightarrow \text{red}(G_1) \vee \text{red}(G_2))$$
$$\text{sliced}(G_3) \rightarrow (\text{blue}(G_3) \leftrightarrow \text{blue}(G_1) \vee \text{blue}(G_2))$$
$$\text{digest}(G_1) \rightarrow \neg \text{sliced}(G_1)$$

$$\text{digest}(G_3) \leftrightarrow (\text{rc}(G_3) \wedge \text{rf}(G_3)) \vee (\text{bc}(G_3) \wedge \text{bf}(G_3))$$
$$\text{rc}(G_3) \leftrightarrow (\neg \text{sliced}(G_3) \wedge (\text{red}(G_1) \vee \text{red}(G_2))$$
$$\ldots$$

Minimizing Interpolant

Solution:

- encode all sequences of transformations as an instance of SAT;
- solutions encode all slicing off transformations;
- compute small interpolants: smallest digest of grey formulas;

$$\min_{\{G_{i_1},\ldots,G_{i_n}\}} \left(\sum_{G_i} \text{digest}(G_i) \right)$$

- use a pseudo-boolean optimisation tool or an SMT solver to minimise interpolants;
- minimising interpolants is an NP-complete problem.

Minimizing Interpolant

Solution:

- encode all sequences of transformations as an instance of SAT;
- solutions encode all slicing off transformations;
- compute small interpolants: smallest digest of grey formulas;

$$\texttt{min}_{\{G_{i_1},\ldots,G_{i_n}\}} \left(\sum_{G_i} \text{digest}(G_i) \right)$$

- use a pseudo-boolean optimisation tool or an SMT solver to minimise interpolants;
- minimising interpolants is an NP-complete problem.

Minimizing Interpolant

Solution:

- encode all sequences of transformations as an instance of SAT;
- solutions encode all slicing off transformations;
- compute small interpolants: smallest digest of grey formulas;

$$\texttt{min}_{\{G_{i_1},\ldots,G_{i_n}\}} \left(\sum_{G_i} \text{digest}(G_i) \right)$$

$$\texttt{min}_{\{G_{i_1},\ldots,G_{i_n}\}} \left(\sum_{G_i} \text{quantifier\_number}(G_i)\ \text{digest}(G_i) \right)$$

- use a pseudo-boolean optimisation tool or an SMT solver to minimise interpolants;
- minimising interpolants is an NP-complete problem.

Minimizing Interpolant

Solution:

- encode all sequences of transformations as an instance of SAT;
- solutions encode all slicing off transformations;
- compute small interpolants: smallest digest of grey formulas;

$$\texttt{min}_{\{G_{i_1},\ldots,G_{i_n}\}} \left(\sum_{G_i} \text{digest}(G_i) \right)$$

$$\texttt{min}_{\{G_{i_1},\ldots,G_{i_n}\}} \left(\sum_{G_i} \text{quantifier\_number}(G_i)\ \text{digest}(G_i) \right)$$

- use a pseudo-boolean optimisation tool or an SMT solver to minimise interpolants;
- minimising interpolants is an NP-complete problem.

Minimizing Interpolant

Solution:

- encode all sequences of transformations as an instance of SAT;
- solutions encode all slicing off transformations;
- compute small interpolants: smallest digest of grey formulas;

$$\texttt{min}_{\{G_{i_1},\ldots,G_{i_n}\}} \left(\sum_{G_i} \text{digest}(G_i) \right)$$

$$\texttt{min}_{\{G_{i_1},\ldots,G_{i_n}\}} \left(\sum_{G_i} \text{quantifier\_number}(G_i)\ \text{digest}(G_i) \right)$$

- use a pseudo-boolean optimisation tool or an SMT solver to minimise interpolants;
- minimising interpolants is an NP-complete problem.

Experiments with Small Interpolants

- Implemented in Vampire;
- We used Yices for solving pseudo-boolean constraints;
- Experimental results:
  - 9632 first-order examples from the TPTP library:

    for example, for 2000 problems the size of the interpolants became 20-49 times smaller;
  - 4347 SMT examples:
    - we used Z3 for proving SMT examples;
    - Z3 proofs were localised in Vampire;
    - small interpolants were generated for 2123 SMT examples.

Experiments with Small Interpolants

- Implemented in Vampire;
- We used Yices for solving pseudo-boolean constraints;
- Experimental results:
  - 9632 first-order examples from the TPTP library:

    for example, for 2000 problems the size of the interpolants became 20-49 times smaller;
  - 4347 SMT examples:
    - we used Z3 for proving SMT examples;
    - Z3 proofs were localised in Vampire;
    - small interpolants were generated for 2123 SMT examples.

Experiments with Small Interpolants

- Implemented in Vampire;
- We used Yices for solving pseudo-boolean constraints;
- Experimental results:
  - 9632 first-order examples from the TPTP library:

    for example, for 2000 problems the size of the interpolants became 20-49 times smaller;
  - 4347 SMT examples:
    - we used Z3 for proving SMT examples;
    - Z3 proofs were localised in Vampire;
    - small interpolants were generated for 2123 SMT examples.

Outline

Quantifier Complexity of Interpolants

Local Proofs Do Not Always Exist

- R: $(\forall x)p(r, x)$
- B: $(\forall y)\neg p(y, b)$
- Reverse interpolant I of R and B: $(\exists y)(\forall x)p(y, x)$.
- Note: R and B contain no quantifier alternations, yet I contains quantifier alternations. One can prove that every interpolant of this formula must have at least one quantifier alternation.
- There is no local refutation of R, B in the resolution/superposition calculus.
- There is a non-local one:

$$\frac{p(r, x) \quad \neg p(y, b)}{\bot}$$

Quantifier Complexity of Interpolants

Local Proofs Do Not Always Exist

- R: $(\forall x)p(r, x)$
- B: $(\forall y)\neg p(y, b)$
- Reverse interpolant I of R and B: $(\exists y)(\forall x)p(y, x)$.
- Note: R and B contain no quantifier alternations, yet I contains quantifier alternations. One can prove that every interpolant of this formula must have at least one quantifier alternation.
- There is no local refutation of R, B in the resolution/superposition calculus.
- There is a non-local one:

$$\frac{p(r, x) \quad \neg p(y, b)}{\bot}$$

Quantifier Complexity of Interpolants

Local Proofs Do Not Always Exist

- R: $(\forall x)p(r, x)$
- B: $(\forall y)\neg p(y, b)$
- Reverse interpolant I of R and B: $(\exists y)(\forall x)p(y, x)$.
- Note: R and B contain no quantifier alternations, yet I contains quantifier alternations. One can prove that every interpolant of this formula must have at least one quantifier alternation.
- There is no local refutation of R, B in the resolution/superposition calculus.
- There is a non-local one:

$$\frac{p(r, x) \quad \neg p(y, b)}{\bot}$$

Quantifier Complexity of Interpolants

Theorem There is no lower bound on the number of quantifier alternations in interpolants of universal sentences.

That is, for every positive integer n there exist universal sentences R, B such that $\{R, B\}$ is unsatisfiable and every reverse interpolant of R and B has at least n quantifier alternations.

Quantifier Complexity of Interpolants

Example

Take the formula A: $\forall x_1 \exists y_1 \forall x_1 \exists y_2 \ldots p(x_1, y_1, x_2, y_2, \ldots)$ and let R be obtained by skolemizing A and B be obtained by skolemizing $\neg A$:

$$
\begin{aligned}
R &= \forall x_1 \forall x_2 \ldots p(x_1, r_1(x_1), x_2, r_2(x_1, x_2), \ldots) \\
B &= \forall y_1 \forall y_2 \ldots \neg p(b_1, y_1, b_2(y_1), y_2, \ldots) \\
I &= \forall x_1 \exists y_1 \forall x_2 \exists y_2 \ldots p(x_1, y_1, x_2, y_2, \ldots)
\end{aligned}
$$

There is no reverse interpolant with a smaller number of quantifier alternations.

The resolution refutation consists of a single step deriving the empty clause from R and B.

Quantifier Complexity of Interpolants

Example

Take the formula A: $\forall x_1 \exists y_1 \forall x_1 \exists y_2 \ldots p(x_1, y_1, x_2, y_2, \ldots)$ and let R be obtained by skolemizing A and B be obtained by skolemizing $\neg A$:

$$
\begin{aligned}
R &= \forall x_1 \forall x_2 \ldots p(x_1, r_1(x_1), x_2, r_2(x_1, x_2), \ldots) \\
B &= \forall y_1 \forall y_2 \ldots \neg p(b_1, y_1, b_2(y_1), y_2, \ldots) \\
I &= \forall x_1 \exists y_1 \forall x_2 \exists y_2 \ldots p(x_1, y_1, x_2, y_2, \ldots)
\end{aligned}
$$

There is no reverse interpolant with a smaller number of quantifier alternations.

The resolution refutation consists of a single step deriving the empty clause from R and B.

Quantifier Complexity of Interpolants

Example

Take the formula A: $\forall x_1 \exists y_1 \forall x_1 \exists y_2 \ldots p(x_1, y_1, x_2, y_2, \ldots)$ and let R be obtained by skolemizing A and B be obtained by skolemizing $\neg A$:

$$
\begin{aligned}
R &= \forall x_1 \forall x_2 \ldots p(x_1, r_1(x_1), x_2, r_2(x_1, x_2), \ldots) \\
B &= \forall y_1 \forall y_2 \ldots \neg p(b_1, y_1, b_2(y_1), y_2, \ldots) \\
I &= \forall x_1 \exists y_1 \forall x_2 \exists y_2 \ldots p(x_1, y_1, x_2, y_2, \ldots)
\end{aligned}
$$

There is no reverse interpolant with a smaller number of quantifier alternations.

The resolution refutation consists of a single step deriving the empty clause from R and B.

Quantifier Complexity of Interpolants

Bad News for Local Proof Systems

Let S be an inference system with the following property: for every red formula R and blue formula B, if $\{R, B\}$ is unsatisfiable, then there is a local refutation of R, B in S.

Then the number of quantifier alternations in refutations of universal formulas of S is not bound by any positive integer.

Quantifier Complexity of Interpolants

- There is no bound on the number of quantifier alternations in reverse interpolants of universal formulas.
- Any complete local proof system for first-order predicate logic must have inferences dealing with formulas of an arbitrary quantifier complexity, even if the input formulas have no quantifier alternations.
- There is no simple modification of the superposition calculus for logic with/without equality in which every unsatisfiable formula has a local refutation.

Quantifier Complexity of Interpolants

- There is no bound on the number of quantifier alternations in reverse interpolants of universal formulas.
- Any complete local proof system for first-order predicate logic must have inferences dealing with formulas of an arbitrary quantifier complexity, even if the input formulas have no quantifier alternations.
- There is no simple modification of the superposition calculus for logic with/without equality in which every unsatisfiable formula has a local refutation.

Quantifier Complexity of Interpolants

- There is no bound on the number of quantifier alternations in reverse interpolants of universal formulas.
- Any complete local proof system for first-order predicate logic must have inferences dealing with formulas of an arbitrary quantifier complexity, even if the input formulas have no quantifier alternations.
- There is no simple modification of the superposition calculus for logic with/without equality in which every unsatisfiable formula has a local refutation.