First-Order Theorem Proving and VAMPIRE

Baptiste Jacquemot, Yanick Paulo-Amaro and Antoine Brunner

Reference

Laura Kovács and Andrei Voronkov. 2013. First-Order Theorem Proving and Vampire. In *Computer Aided Verification - 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13-19, 2013. Proceedings* (Lecture Notes in Computer Science), Springer, 1–35. DOI:https://doi.org/10.1007/978-3-642-39799-8_1

Outline

- What is theorem proving ?
- What is saturation-based theorem proving ?
- Saturation algorithm implementation

What is theorem proving ?

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: $\forall x \forall y(x \cdot y = y \cdot x)$

What is theorem proving ?

Proof by refutation

Instead of proving that something is correct, try to derive a contradiction from its negation:

1. build the initial set of known formulas: Axioms U Assumptions U (**not** Conjectures)
2. apply inference rules to this set of formulas until a contradiction is found (i.e. *false* appears in the set of propositions)

What is theorem proving ?

$$\frac{F_1 \quad \dots \quad F_n}{F} \,.$$

- Given a set of formulas, check their validity.
- This is done using an inference system.
- An **inference rule** is an relation on formulas that, given n *premises* gives a *conclusion*.
- An **inference system** is a set of inference rules.
- An inference system is **sound** if a formula is unsatisfiable whenever the empty clause is derivable
- An inference system is **complete** if the empty clause is derivable whenever the formula is unsatisfiable

What is theorem proving ?

Sample of proof generated by vampire

```
203. $false [subsumption resolution 202,14]
202. sP1(mult(sK,sK0)) [backward demodulation 188,15]
188. mult(X8,X9) = mult(X9,X8) [superposition 22,87]
87. mult(X2,mult(X1,X2)) = X1 [forward demodulation 71,27]
71. mult(inverse(X1),e) = mult(X2,mult(X1,X2)) [superposition 23,20]
27. mult(inverse(X2),e) = X2 [superposition 22,10]
```

...

```
8. mult(sK,sK0) != mult(sK0,sK) [skolemisation 7]
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(mult(X0,X1),X2) = mult(X0,mult(X1,X2)) [input]
2. ! [X0] : e = mult(inverse(X0),X0) [input]
1. ! [X0] : mult(e,X0) = X0 [input]
```

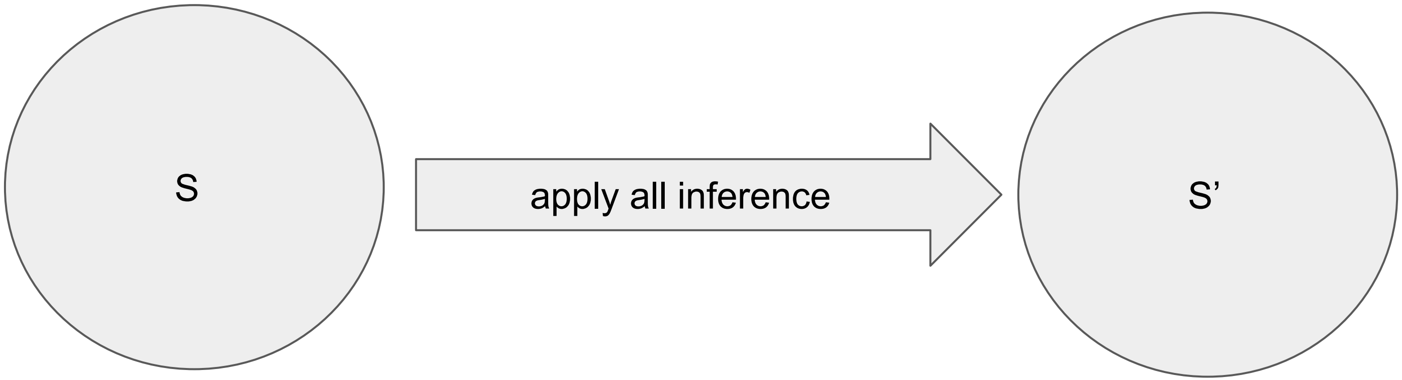
What is theorem proving ?

- Theorem proving in first-order logic is a **hard** problem. It is *complete* (valid formula implies finite proof) but *undecidable* (no algorithm to determine if formula is valid).
- Basic idea: given enough time and a *complete inference system*, if the formula is unsatisfiable, then a refutation will be found
- A theorem prover either:
  - Finds a refutation in a finite amount of time, meaning that what we want to prove is valid
  - Runs forever, meaning that the validity of the formula is unknown

Saturation-based theorem proving

Saturation

A set of clauses *S* is called saturated with respect to an inference system $\mathcal{I}$ if, for every inference in $\mathcal{I}$ with premises in *S*, the conclusion of this inference belongs to *S* too.



S is saturated
If S = S’

Saturation-based theorem proving

- Saturation generates an exponential number of propositions
- Redundancy can help mitigate that problem

Redundancy

For a clause C in a set of clauses S, C is called **redundant** iff there exists $U \subset S$ such that $\{U_1, \ldots U_n\} \Rightarrow C$ and $U_i < C$ forall i

- "<" is a relation called a term ordering that gives a notion of a term being smaller than another
- It is used to ensure that we only replace redundant clauses by smaller clauses

Saturation-based theorem proving

Two types of inferences:

- **Simplifying inferences**: an inference of the form $C_1, \ldots, C_n \rightarrow C$ is called simplifying inference iff one of the C_i becomes redundant after its C is added to the set. C_i can therefore be removed from the set.

- **Generating inferences**: all inferences that are not simplifying inferences

To achieve efficiency:

- apply simplifying inferences eagerly
- apply generating inferences lazily

To achieve completeness:

- apply inferences in a *fair* manner

Example of simplifying inferences

- Subsumption resolution

$$A\theta \vee C\theta \subseteq B \vee D \qquad \frac{A \vee C \quad \cancel{\neg B \vee D}}{D}$$

- Demodulation

$$l\theta \succ r\theta \text{ and } (l = r)\theta \succ C[l\theta] \qquad \frac{l = r \quad \cancel{C[l\theta]}}{C[r\theta]}$$

Fig. 5 A Simple Saturation Algorithm

```
var kept, unprocessed: sets of clauses;
var new: clause;
unprocessed := the initial sets of clauses;
kept := ∅;
loop
  while unprocessed ≠ ∅
    new :=select(unprocessed);
    if new = □ then return unsatisfiable;
✓   if retained(new) then                                        (* retention test *)
✓     simplify new by clauses in kept ;                     (* forward simplification *)
      if new = □ then return unsatisfiable;
✓     if retained(new) then                             (* another retention test *)
✓       delete and simplify clauses in kept using new;  (* backward simplification *)
        move the simplified clauses from kept to unprocessed;
        add new to kept
  if there exists an inference with premises in kept not selected previously then
✓   select such an inference;                               (* inference selection *)
✓   add to unprocessed the conclusion of this inference    (* generating inference *)
  else return satisfiable or unknown
```

E Theorem prover

```
while U ≠ ∅ begin
  c :=select_best(U)
  U := U\{c}
  simplify(c, P)
  if not redundant(c, P) then
    if c is the empty clause then
      success; clause set is unsatisfiable
    else
      T := ∅
      foreach p ∈ P do
      if c simplifies a maximal literal of
        p such that the set of maximal
        terms, the set of maximal literals or
        the number of literals in p potentially
        changes
      then
        P := P\{p}
        T := T ∪ {p}
        U := U\{d|d is direct descendant of p}
      fi
      simplify(p, (P\{p}) ∪ {c})
    done
    T := T∪ generate(c, P)
    foreach p ∈ T do
      p :=cheap_simplify(p, P)
      if not trivial(p, P) then
        U := U ∪ {p}
      fi
    done
    fi
  fi
end
Failure: Initial U is satisfiable, P describes model
```

Reference

Stephan Schulz. 2002. E - a brainiac theorem prover.
AI Commun. 15, 2–3 (2002), 111–126.

Given clause algorithm

Select a “given clause” and apply **all** inferences on it

Differentiate between active and passive clauses

- Active clause: selected clauses that can’t be simplified
- Passive clauses: all other clauses

Types of “Given clause algorithms”

- Otter algorithms: Include passive clauses during simplification
- Discount algorithms: Only use active clauses for simplification
- LRS (only in Vampire): Otter but drop unreachable clauses

Selection strategy

Select among 2 priority queues:

- Age: Sort by decreasing age
- Weight (clauses size): Sort by increasing weight

An age-weight ratio (a, w) determines the queue to use

- *a* clauses are selected from the age-queue
- *w* clauses from the weight-queue

Users can choose the (a, w) ratio in Vampire

What we have seen

- The basics of theorem proving and inference systems
- The concept of saturation theorem proving
- The different types of saturation algorithms used by modern theorem provers
- The unique LRS strategy that makes Vampire so powerful