Outline

From Theory to Practice

From theory to practice

- Preprocessing and CNF transformation;
- Superposition system;
- Orderings;
- Selection functions;
- Fairness (saturation algorithms);
- Redundancy.

Vampire's preprocessing (incomplete list)

1. (Optional) Select a relevant subset of formulas.
2. (Optional) Add theory axioms;
3. Rectify the formula.
4. If the formula contains any occurrence of $\top$ or $\bot$, simplify the formula.
5. Remove if-then-else and let-in connectives.
6. Flatten the formula.
7. Apply pure predicate elimination.
8. (Optional) Remove unused predicate definitions.
9. Convert the formula into equivalence negation normal form.
10. Use a naming technique to replace some subformulas by their names.
11. Convert the formula into negation normal form.
12. Skolemize the formula.
13. (Optional) Replace equality axioms.
14. Determine a literal ordering to be used.
15. Transform the formula into its conjunctive normal form.
16. (Optional) Function definition elimination.
17. (Optional) Inequality splitting.
18. Remove tautologies.
19. Pure literal elimination.
20. Remove clausal definitions.

Checking Redundancy

Suppose that the current search space S contains no redundant clauses. How can a redundant clause appear in the inference process?

Checking Redundancy

Suppose that the current search space S contains no redundant clauses. How can a redundant clause appear in the inference process?
Only when a new clause (a child of the selected clause and possibly other clauses) is added.
Classification of redundancy checks:

- The child is redundant;
- The child makes one of the clauses in the search space redundant.

Checking Redundancy

Suppose that the current search space S contains no redundant clauses. How can a redundant clause appear in the inference process?
Only when a new clause (a child of the selected clause and possibly other clauses) is added.
Classification of redundancy checks:

- The child is redundant;
- The child makes one of the clauses in the search space redundant.

We use some fair strategy and perform these checks after every inference that generates a new clause.
In fact, one can do better.

Demodulation, Non-Ground Case

$$\frac{l \simeq r \quad L[l'] \vee D}{L[r\theta] \vee D} \text{ (Dem)},$$

where $l\theta = l'$, $l\theta \succ r\theta$, and $(L[l'] \vee D)\theta \succ (l\theta \succ r\theta)$.

Demodulation, Non-Ground Case

$$\frac{l \simeq r \quad L[l'] \vee D}{L[r\theta] \vee D} \text{ (Dem)},$$

where $l\theta = l'$, $l\theta \succ r\theta$, and $(L[l'] \vee D)\theta \succ (l\theta \succ r\theta)$.

Easier to understand:

$$\frac{l \simeq r \quad L[l\theta] \vee D}{L[r\theta] \vee D} \text{ (Dem)},$$

where $l\theta \succ r\theta$, and $(L[l'] \vee D)\theta \succ (l\theta \succ r\theta)$.

Generating and Simplifying Inferences

An inference

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

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

A non-simplifying inference is called generating.

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.

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

Term Indexing

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

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

Term Indexing

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

Term Indexing

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

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

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

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

Term Indexing

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

Term Indexing in Vampire

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

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 := ∅;
  passive := init;
  while passive ≠ ∅ do
*   current := select(passive);                  (* clause selection *)
    move current from passive to active;
*   queue:=infer(current, active);              (* generating inferences *)
    if □ ∈ queue then return unsatisfiable;
    passive := passive ∪ queue
  od;
  return satisfiable
```

Given Clause Algorithm (with Simplification)

In fact, there is more than one . . .

Otter vs. Discount Saturation

Otter saturation algorithm:

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

Discount saturation algorithm:

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

Otter vs. Discount Saturation, Newly Generated Clauses

Otter saturation algorithm:

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

Discount saturation algorithm:

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

Otter vs. Discount Saturation, Newly Generated Clauses

Otter saturation algorithm:

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

Discount saturation algorithm:

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

Otter Saturation Algorithm

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

Discount Saturation Algorithm

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

Age-Weight Ratio

How to select nice clauses?

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

Age-Weight Ratio

How to select nice clauses?

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

Solution:

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

Limited Resource Strategy

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

Limited Resource Strategy

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

Try:

```
vampire --age_weight_ratio 10:1
  --backward_subsumption off
  --time_limit 60000
      GRP140-1.p
```