Algorithms and Data Structures for First-Order Equational Deduction

Stephan Schulz

TU München, Germany

schulz@eprover.org

First-Order Theorem Proving

Given: A set of first-order axioms and a hypothesis

$$A = \{A_1, \ldots, A_n\},\ H$$

Question: Do the axioms logically imply the hypothesis?

$$A \overset{?}{\models} H$$

Can this question be answered automatically?

The Two Steps of Refutational Theorem Proving

The hard step

The impossible step

The Two Steps of Refutational Theorem Proving

The hard step: Convert $A \models H$ into S where. . .

- S is a set of first-order clauses
- S is unsatisfiable if and only if $A \models H$ holds

The impossible step

The Two Steps of Refutational Theorem Proving

The hard step: Convert $A \models H$ into S where. . .

- S is a set of first-order clauses
- S is unsatisfiable if and only if $A \models H$ holds

The impossible step: Decide wether S is unsatisfiable

- But we can show unsatisfiability
- . . . given infinite resources!

The Two Steps of Refutational Theorem Proving

The hard step: Convert $A \models H$ into S where...

- S is a set of first-order clauses
- S is unsatisfiable if and only if $A \models H$ holds

The impossible step: Decide wether S is unsatisfiable

- But we can show unsatisfiability
- ... given infinite resources!

> Theorems can be proved
> Non-theorems cannot always be refuted

Hard problems are solved immediately. . .

. . . the impossible may take a bit longer

CNF Conversion

If you want the most advanced converter: Use FLOTTER

- CNF converter of the SPASS project
- Very advanced techniques, usually very good clause normal forms

If you want a readable standard syntax, use E

- `eprover --cnf` converts TPTP-2 or TPTP-3 FOF into CNF
- Reasonably advanced technique (converts all TPTP 3.1.1 problems)
- Typically fast even on large and complex formulae
- Resulting CNF sometimes worse than FLOTTER

Tackling The Impossible Task

Saturation-Based Theorem Proving

- The proof state is a set of clauses
- New clauses are added to the proof state

Generating inference rules:

- Deduce new clauses from several existing clauses
- Most important inference rule: Paramodulation/Superposition/Resolution

Redundancy elimination allows deletion or replacing of clauses

- Rewriting: Apply equations to simplify terms
- Subsumption: Drop more specific clauses in favour of more general ones

Clauses

Clauses are disjunctions of literals

Example:

$$X \not\simeq add(Y,1) \vee odd(X) \vee odd(Y)$$

Alternative views: Implicational

$$X \simeq add(Y,1) \implies (odd(X) \vee odd(Y))$$

or

$$(X \simeq add(Y,1) \wedge \neg odd(X)) \implies odd(Y))$$

or

$$(X \simeq add(Y,1) \wedge \neg odd(Y)) \implies odd(X))$$

or (weirdly)

$$(\neg odd(Y) \wedge \neg odd(X)) \implies X \not\simeq add(Y,1)$$

Literals

$X \not\simeq add(Y, 1) \vee odd(X) \vee odd(Y)$

- $X \not\simeq add(Y, 1)$ is a negative equational literal
- $odd(X)$ and $odd(X)$ are positive non-equational literals

Conventions:

- $s \not\simeq t$ is a more convenient way of writing $\neg s \simeq t$
- We write $s \dot{\simeq} t$ to denote an equational literal that may be either positive or negative
- $s \simeq t$ is a more conventient way of writing $\simeq (s, t)$

Literals

$X \not\simeq add(Y, 1) \vee odd(X) \vee odd(Y)$

- $X \not\simeq add(Y, 1)$ is a negative equational literal
- $odd(X)$ and $odd(X)$ are positive non-equational literals

Convention:

- $s \not\simeq t$ is a more convenient way of writing $\neg s \simeq t$
- We write $s \dot{\simeq} t$ to denote an equational literal that may be either positive or negative
- Heresy: $s \simeq t$ is a more conventient way of writing $\simeq (s, t)$
- Truth: $odd(X)$ is a more convenient way of writing $odd(X) \simeq \top$

Terms

$X \not\simeq add(Y, 1) \vee odd(X) \vee odd(Y)$

- $X, add(Y, 1), 1$, and Y are terms
- X and Y are variables
- 1 is a constant term
- $add(Y, 1)$ is a composite term with proper subterms 1 and Y

Rewriting

Ordered application of equations

- Replace equals with equals. . .
- . . . if this decreases term size with respect to given ordering $>$

$$\frac{s \simeq t \qquad u \dot{\simeq} v \vee R}{s \simeq t \qquad u[p \leftarrow \sigma(t)] \dot{\simeq} v \vee R}$$

Conditions:

- $u|_p = \sigma(s)$
- $\sigma(s) > \sigma(t)$
- Some restrictions on rewriting $>$-maximal terms in a clause apply

Note: If $s > t$, we call $s \simeq t$ a rewrite rule

- Implies $\sigma(s) > \sigma(t)$, no ordering check necessary

Paramodulation/Superposition

Superposition: "Lazy conditional speculative rewriting"

- Conditional: Uses non-unit clauses
  - One positive literal is seen as potential rewrite rule
  - All other literals are seen as (positive and negative) conditions
- Lazy: Conditions are not solved, but appended to result
- Speculative:
  - Replaces potentially larger terms
  - Applies to instances of clauses (generated by unification)
  - Original clauses remain (generating inference)

$$\frac{s \simeq t \vee S \quad u \dot{\simeq} v \vee R}{\sigma(u[p \leftarrow t] \dot{\simeq} v \vee S \vee R)}$$

Conditions:

- $\sigma = mgu(u|_p, s)$ and $u|_p$ is not a variable
- $\sigma(s) \not< \sigma(t)$ and $\sigma(u) \not< \sigma(v)$
- $\sigma(s \simeq t)$ is $>$-maximal in $\sigma(s \simeq t \vee S)$ (and no negative literal is selected)
- $\sigma(u \dot{\simeq} v)$ is maximal (and no negative literal is selected) or selected

Subsumption

Idea: Only keep the most general clauses

- If one clause is subsumed by another, discard it

$$\frac{C \qquad \sigma(C) \vee R}{C}$$

Examples:

- $p(X)$ subsumes $p(a) \vee q(f(X), a)$ ($\sigma = \{X \leftarrow a\}$)
- $p(X) \vee p(Y)$ does not multi-set-subsume $p(a) \vee q(f(X), a)$
- $q(X, Y) \vee q(X, a)$ subsumes $q(a, a) \vee q(a, b)$

Subsumption is hard (NP-complete)

- $n!$ permutations in non-equational clause with n literals
- $n!2^n$ permutations in equational clause with n literals

The Basic Given-Clause Algorithm

Completeness requires consideration of all possible persistent clause combinations for generating inferences

- For superposition: All 2-clause combinations
- Other inferences: Typically a single clause

Given-clause algorithm replaces complex bookkeeping with simple invariant:

- Proofstate $S = P \cup U$, P initially empty
- All inferences between clauses in P have been performed

The algorithm:

while $U \neq \{\}$
  $g =$ delete_best(U)
  if $g == \square$
    SUCCESS, Proof found
  $P = P \cup \{g\}$
  $U = U \cup$generate(g, P)
SUCCESS, original U is satisfiable

DISCOUNT Loop

Aim: Integrate simplification into given clause algorithm

The algorithm (as implemented in E):

while $U \neq \{\}$
  $g =$ delete_best(U)
  $g =$ simplify(g,P)
  if $g == \Box$
    SUCCESS, Proof found
  if g is not redundant w.r.t. P
    $T = \{c \in P | c$ redundant or simplifiable w.r.t. $g\}$
    $P = (P \backslash T) \cup \{g\}$
    $T = T \cup$generate(g, P)
    foreach $c \in T$
      $c =$cheap_simplify(c, P)
      if c is not trivial
        $U = U \cup \{c\}$
SUCCESS, original U is satisfiable

What is so hard about this?

What is so hard about this?

Data from simple TPTP example NUM030-1+rm_eq_rstfp.lop
(solved by E in 30 seconds on ancient Apple Powerbook):

- Initial clauses: 160
- Processed clauses: 16,322
- Generated clauses: 204,436
- Paramodulations: 204,395
- Current number of processed clauses: 1,885
- Current number of unprocessed clauses: 94,442
- Number of terms: 5,628,929

Hard problems run for days!

- Millions of clauses generated (and stored)
- Many millions of terms stored and rewritten
- Each rewrite attempt must consider many ($>>$ 10000) rules
- Subsumption must test many ($>>$ 10000) candidates for each subsumption attempt
- Heuristic must find best clause out of millions

First-Order Terms

Terms are words over the alphabet $F \cup V \cup \{'(', ')', ','\}$, where. . .

Variables: $V = \{X, Y, Z, X1, \ldots\}$

Function symbols: $F = \{f/2, g/1, a/0, b/0, \ldots\}$

Definition of terms:

- $X \in V$ is a term
- $f/n \in F, t_1, \ldots, t_n$ are terms $\rightsquigarrow f(t_1, \ldots, t_n)$ is a term
- Nothing else is a term

> Terms are by far the most frequent objects in a typical proof state!
> $\rightsquigarrow$ Term representation is critical!

Representing Function Symbols and Variables

Naive: Representing function symbols as strings: `"f"`, `"g"`, `"add"`

- May be ok for f, g, add
- Users write $unordered\_pair, universal\_class, \ldots$

Solution: Signature table

- Map each function symbol to unique small positive integer
- Represent function symbol by this integer
- Maintain table with meta-information for function symbols indexed by assigned code

Handling variables:

- Rename variables to $\{X_1, X_2, \ldots\}$
- Represent X_i by $-i$
- Disjoint from function symbol codes!

From now on, assume this always done!

Representing Terms

Naive: Represent terms as strings `"f(g(X), f(g(X),a))"`

More compact: `"fgXfgXa"`

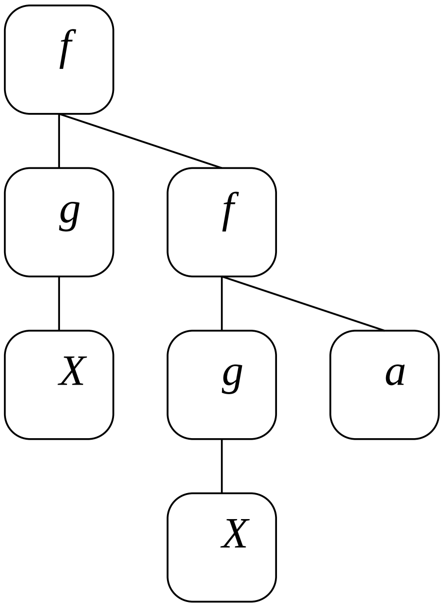
- Seems to be very memory-efficient!
- But: Inconvenient for manipulation!

Terms as ordered trees

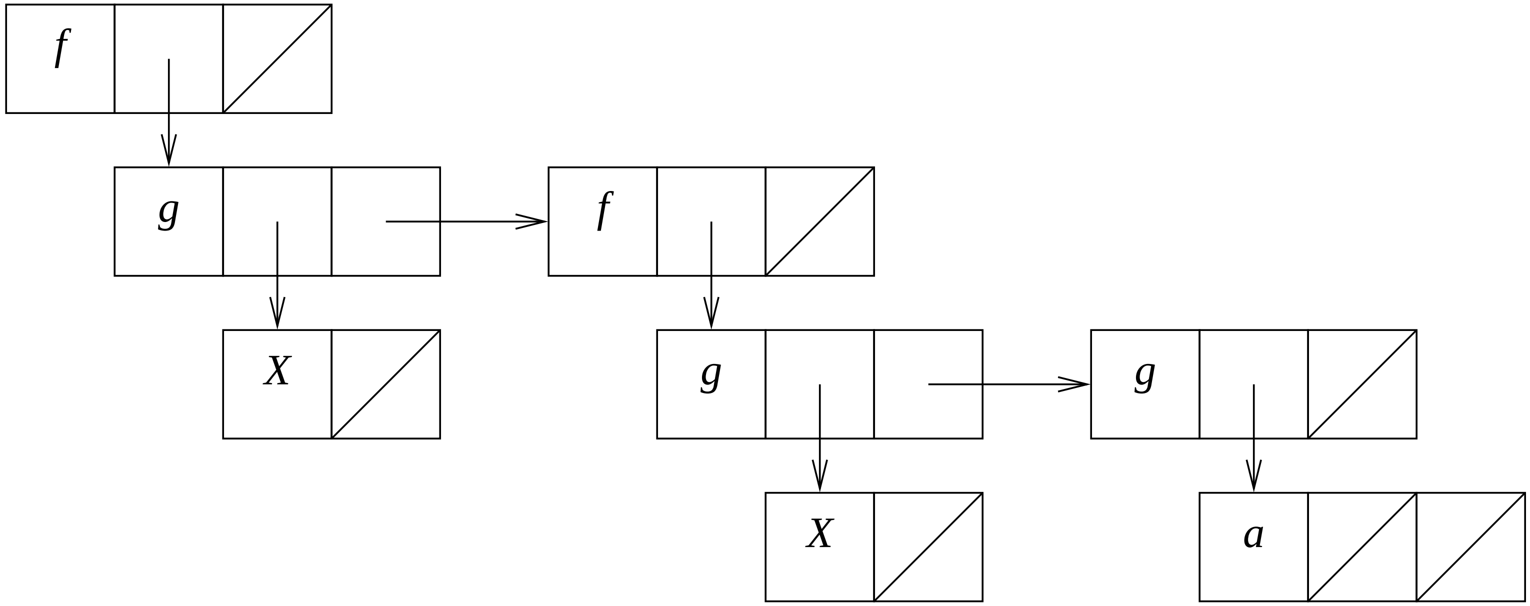
- Nodes are labeled with function symbols or variables
- Successor nodes are subterms
- Leaf nodes correspond to variables or constants
- Obvious approach, used in many systems!

Abstract Term Trees

Example term: $f(g(X), f(g(X), a))$



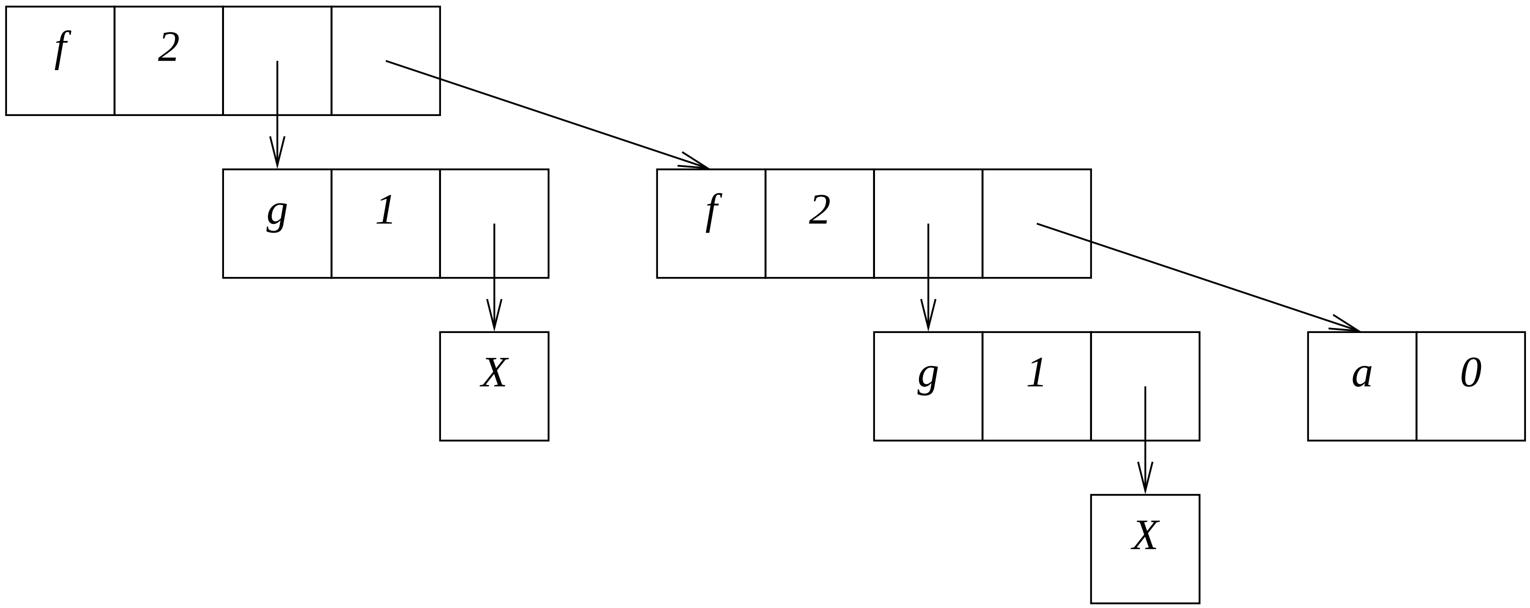
LISP-Style Term Trees



Argument lists are represented as linked lists

Implemented e.g. in PCL tools for DISCOUNT and Waldmeister

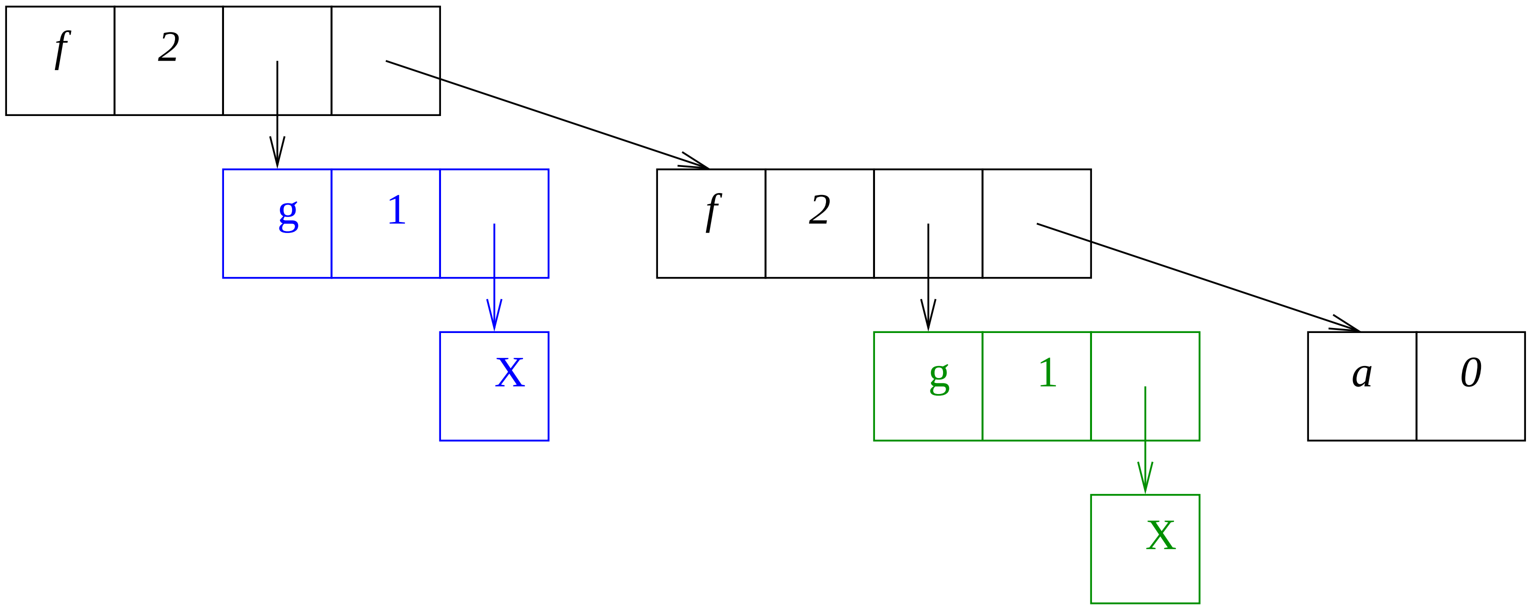
C/ASM Style Term Trees



Argument lists are represented by arrays with length

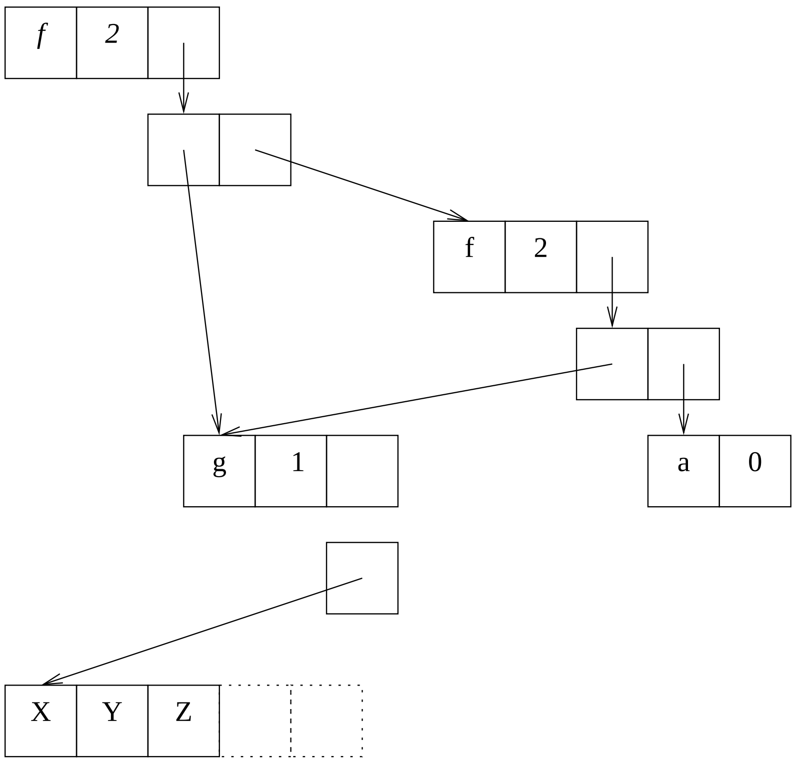
Implemented e.g. in DISCOUNT (as an evil hack)

C/ASM Style Term Trees



In this version: Isomorphic subterms have isomorphic representation!

Shared Terms (E)



Idea: Consider terms not as trees, but as DAGs

- Reuse identical parts
- Shared variable banks (trivial)
- Shared term banks maintained bottom-up

Shared Terms

Disadvantages:

- More complex
- Overhead for maintaining term bank
- Destructive changes must be avoided

Direct Benefits:

- Saves between 80% and 99.99% of term nodes
- Consequence: We can afford to store precomputed values
  - Term weight
  - Rewrite status (see below)
  - Groundness flag
  - ...
- Term identity: One pointer comparison!

Efficient Rewriting

Problem:

- Given term t, equations $E = \{l_1 \simeq r_1 \ldots l_n \simeq r_n\}$
- Find normal form of t w.r.t. E

Bottlenecks:

- Find applicable equations
- Check ordering constraint ($\sigma(l) > \sigma(r)$)

Solutions in E:

- Cached rewriting (normal form date, pointer)
- Perfect discrimination tree indexing with age/size constraints

Shared Terms and Cached Rewriting

Shared terms can be long-term persistent!

Shared terms can afford to store more information per term node!

Hence: Store rewrite information

- Pointer to resulting term
- Age of youngest equation with respect to which term is in normal form

Terms are at most rewritten once!

Search for matching rewrite rule can exclude old equations!

Subsumption Indexing

Problem:

- Given clause C, clause set $S = \{C_1, \ldots, C_n\}$
- Find σ, C_i with $\sigma(C_i) \subseteq C$
- Find all C_i with $\sigma(C) \subseteq C_i$

Bottlenecks:

- Checking one pair C, C_i for subsumption is NP-hard!
- S is large!

Solutions in E:

- Use feature vector indexing to find subsumption candidates (reduces number of tests by 97%)

. . . it's still NP-Complete

Speeding up clause-clause subsumption:

- Test simple required conditions
  - Weight
  - Lenght
  - Number of positive/negative literals
  - Single literal matches
- Use stable orderings to preorder literals
  - Not always possible
  - But extremely effective in practice
- Cheating:
  - Terminate subsumption after predetermined amount of time
  - Never try very large clauses against each other

Conlusion

Building a good implementation is a non-trivial undertaking

Major algorithmic problems

Many good approaches exist. . .

. . . but are too little known!

> The IWIL workshop series helps!