

Simplicity, Measuring, and Good Engineering

One Way to Build a World

Class Automated Deduction System

**4th International Workshop on the Implementation of Logics
– Invited Talk –**

Stephan Schulz

Research Institute for Symbolic Computation
Johannes-Kepler-Universität Linz

and

Automated Reasoning Group
Institut für Informatik
Technische Universität München

`schulz@informatik.tu-muenchen.de`

Supported by the CALCULEMUS Human Potential Programme

We have to few good Implementations!

Just for first order logic there is no good implementation of. . .

- ▶ Superposition with constraints
- ▶ Basic superposition (for non-unit problems)
- ▶ Superposition modulo AC (for non-unit problems)
- ▶ Clause linking
- ▶ Clause trees
- ▶ Model evolution

Success of DCTP shows that there is a lot to be learned from underimplemented calculi!

Overview

- ▶ Equational theorem proving
- ▶ The theorem prover **E**
- ▶ Major changes and major blunders
- ▶ Development tools and techniques
- ▶ Conclusion and future work

Equational Theorem Proving

- ▶ Given: Set of first order **clauses** with equality
- ▶ Task: Show that the clause set is unsatisfiable
- ▶ Predominant method: **Saturation-Based Proving**
 - Systematically enumerate consequences of clause set
 - Simplify or remove redundant clauses
 - Unsatisfiable clause set will eventually yield empty clause (explicit refutation)
- ▶ Predominant calculus: **Superposition**
 - Important generating inferences: Superposition (restricted paramodulation)
 - Important simplifying inferences:
 - * Rewriting
 - * Trivial literal deletion
 - Important deleting inferences:
 - * Subsumption
 - * Tautology deletion

Rewriting

- ▶ Ordered application of equations
 - **Replace** equals with equals. . .
 - . . . if this decreases term size with respect to given $>$

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

- ▶ 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)

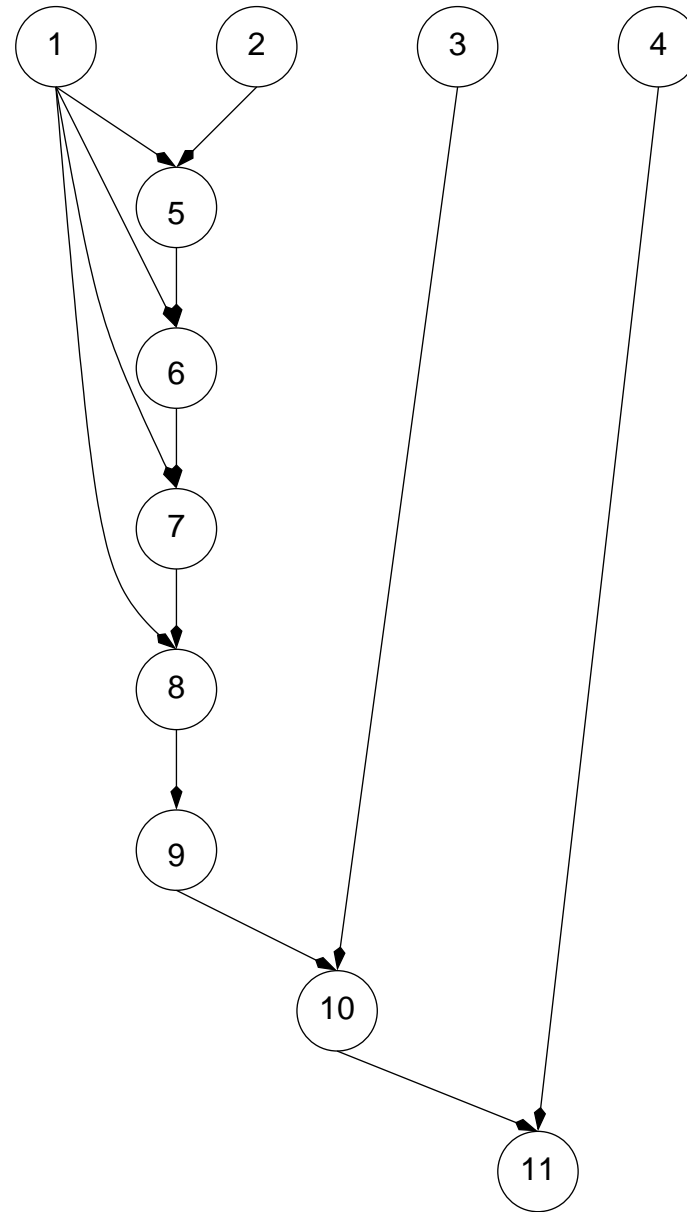
$$\frac{s \simeq t \vee S \quad u \simeq v \vee R}{\sigma(u[p \leftarrow t] \simeq v \vee S \vee R)}$$

- ▶ Conditions:
 - $\sigma = mgu(u|_p, s)$ and $u|_p$ is not a variable
 - $\sigma(s) \not\prec \sigma(t)$ and $\sigma(u) \not\prec \sigma(v)$
 - $\sigma(s \simeq t)$ is $>$ -maximal in $\sigma(s \simeq t \vee S)$ (and no negative literal is **selected**)
 - $\sigma(u \simeq v)$ is maximal (and no negative literal is selected) or selected

Example

- | | |
|---|---|
| (1) $f(a) \simeq a$ | initial |
| (2) $f(f(f(X))) \not\simeq a \vee X \simeq b$ | initial |
| (3) $P(a) \quad (P(a) \simeq \top)$ | initial |
| (4) $\neg P(b) \quad (P(b) \not\simeq \top)$ | initial |
| (5) $f(f(f(a))) \not\simeq a \vee a \simeq b$ | paramodulate 1 into 2 |
| (6) $f(f(a)) \not\simeq a \vee a \simeq b$ | rewrite 5 with 1 |
| (7) $f(a) \not\simeq a \vee a \simeq b$ | rewrite 6 with 1 |
| (8) $a \not\simeq a \vee a \simeq b$ | rewrite 7 with 1 |
| (9) $a \simeq b$ | drop trivial literal in 8 |
| (10) $P(b)$ | rewrite 3 with 9 (assumes $a > b$) |
| (11) \square | resolution (can be seen as paramodulation+simplification) of 10 and 4 |

Derivation Tree



Implementation Challenges

- ▶ Provers have to deal with large amounts of data
 - Millions of clauses (for reasonable proof obligations)
 - Many millions of terms
- ▶ Provers have to efficiently perform inferences
 - Millions to many millions of inferences
 - Major cost: Simplification
- ▶ Provers have to implement efficient heuristics
 - Which calculus restrictions should be chosen
 - Which inference(s) should be performed next?

Other Challenges

- ▶ Provers have to be practically usable
 - Critical: **Correctness** and **Reliability**
 - High stability – few (if any) crashes
 - Portable to at least major research platforms
 - Reasonable interface and input syntax
- ▶ Provers typically are research vehicles
 - **“Fluid”** specifications
 - Frequent changes/extensions
 - Limited funding (especially for reimplemention)
 - Irregular funding – few long term developers

A successful competitive implementation must meet technical as well as other challenges!

The Theorem Prover E

- ▶ State-of-the-art theorem prover for clausal logic with equality
 - Won CASC-17 MIX category
 - Has consistently been among the top systems in MIX and UEQ since then
 - 2003: 3rd place in MIX/Proof, MIX/Assurance, UEQ
- ▶ Based on purely equational superposition calculus
 - Restricted paramodulation, equality resolution, equality factoring
 - Lots of redundancy elimination techniques:
 - * Unconditional rewriting
 - * Equational and clause subsumption
 - * AC-simplification
 - * Unit and contextual literal cutting
 - * Syntactic and semantic tautology deletion
- ▶ Prover is fully automatic (at different levels):
 - No user interaction during proof search
 - Automatic mode offers decent parameter selection

Main Proof Procedure (Naive version)

- ▶ Search state: $P \cup U$
 - P : Processed clauses (all necessary consequences computed, interreduced)
 - U : Unprocessed clauses
 - Initial condition: All clauses in U , P empty

```
while  $U \neq \{\}$ 
   $g = \text{delete\_best}(U)$ 
   $g = \text{simplify}(g, P)$ 
  if  $g == \square$ 
    SUCCESS, Proof found
  if  $g$  is not redundant w.r.t.  $P$ 
     $T = \{c \in P \mid c \text{ redundant or simplifiable w.r.t. } g\}$ 
     $P = (P \setminus T) \cup \{g\}$ 
     $T = T \cup \text{generate}(g, P)$ 
    foreach  $c \in T$ 
       $c = \text{simplify}(c, P)$ 
      if  $c$  is not redundant w.r.t.  $P$ 
         $U = U \cup \{c\}$ 
  SUCCESS, original  $U$  is satisfiable
```

Proof Procedure Analysed

- ▶ Typically $|U| \sim |P|^2$, $|T| \sim |P|$
 - Operations on unprocessed clauses use most CPU time!
 - U dominates memory consumption
- ▶ Main implementation considerations:
 - How can we represent U compactly?
 - How can we speed up simplification?
- ▶ First observation: Clauses in U are **passive**
 - Once a clause is in U , it only has an impact on memory consumption
 - Impact on CPU time is negligible

Improved Proof Procedure

```
while  $U \neq \{\}$ 
   $g = \text{delete\_best}(U)$ 
   $g = \text{simplify}(g, P)$ 
  if  $g == \square$ 
    SUCCESS, Proof found
  if  $g$  is not redundant w.r.t.  $P$ 
     $T = \{c \in P \mid c \text{ redundant or simplifiable w.r.t. } g\}$ 
     $P = (P \setminus T) \cup \{g\}$ 
     $T = T \cup \text{generate}(g, P)$ 
    foreach  $c \in T$ 
       $c = \text{cheap\_simplify}(c, P)$ 
      if  $c$  is not trivial
         $U = U \cup \{c\}$ 
SUCCESS, original  $U$  is satisfiable
```

► Safes effort on simplification of T

- Only some modifying inferences applied to newly generated clauses
- Full simplification applied to **given clause**
- Check for syntactic tautologies replaces more expensive redundancy test

Gaining More Performance

- ▶ Implementation structured around **perfectly shared terms**
 - Every term is represented at most once in the system (unless different copies have different roles in the calculus)
 - * Allows efficient representation of U (\approx one term cell per literal!)
 - * Speeds up many operations (usually term identity \equiv pointer identity)
 - Rewriting is cached at the term level
 - * Pointer from rewritten term to new term
 - Non-rewritability is cached at the term level
 - * Each term node carries age of youngest rewrite rule tried at that node
 - * Used in both forward simplification and backward simplification
- ▶ Other techniques to speed up rewriting:
 - Perfect discrimination trees with size and age constraints
 - * Used for forward rewriting
 - * Reused for forward unit subsumption and forward unit cutting
 - Optionally, use only rewrite rules in `cheap_simplify()`
 - * Saves lots of expensive ordering tests
 - * Caching still makes most other rewrite results available cheaply

Early Development History

July 1997: First line of code written

- ▶ Initial goal: Limited saturation for Horn clauses (for METOP calculus)

July 1998: First version (0.1) released

- ▶ Plain superposition (no literal selection), indexing only for unit rewriting
- ▶ Outstanding feature: Destructive shared rewriting

January 1999: E 0.3 *Castleton*

- ▶ First reasonably powerful version
- ▶ Some external users

July 1999: E 0.5 *Mim* 4th in CASC-16 MIX, helps E-SETHEO win. . .

- ▶ . . . but later disqualified for undetected bug in literal cutting
- ▶ Still, prover is now state-of-the-art
- ▶ Important change: Automatic auto-mode generation from test data
- ▶ Also first CASC-version to support literal selection (since 0.32).

July 2000: E 0.6 *Kanchanjangha* wins CASC-17 MIX for good

Major Changes

- ▶ Changed from pre-saturation for METOP to full prover
- ▶ Added perfect discrimination tree indexing
- ▶ Added literal selection
- ▶ Replaced reference counting with mark-and-sweep garbage collection for term cells
- ▶ Replaced destructive shared rewriting with cached rewriting
- ▶ Changed proof output from complex, reproduction-based to plain PCL2
- ▶ Simplified main loop
- ▶ Added frequency vector indexing for subsumption/contextual literal cutting

Code base has survived a lot of abuse!

E Source Code

- ▶ Implemented in ANSI C89
 - Uses mostly C standard library
 - A few POSIX extensions
 - Very few UNIX-95 extensions
- ▶ Result: Very portable code
 - Runs on SunOS, Solaris, Linux, FreeBSD, MacOS-X, HP-UX
 - Third-Party reports on Sinix, Digital UNIX, Windows. . .
- ▶ 117198 “Lines of Code”
 - Approximately 21000 statements (up from ≈ 20000 in 1999)
 - * Lots of comments
 - * Lots of vertical white space for formatting
 - Approximately 2500 exported functions (<10 statements per function!)
 - 10% totally generic basic functionality
 - 5% proof analysis (accounts for most of growth since 1999)
 - 20% heuristics (2% of that generated automatically)

Bad Decision 1: Destructive Shared Rewriting

- ▶ Original implementation: Rewriting would destructively change all occurrences of original term to rewritten term
 - Idea: Rewriting on shared structure
- ▶ Complex implementation:
 - Term cells had to carry pointers to superterms
 - Term cells had to carry pointers to literals in which they occurred
 - Serious complications with collapsing rules ($f(a) \simeq a$) will **not** directly eliminate $f(a)$ if applied innermost to $f(f(f(f(a))))$
- ▶ Constant source of problems:
 - Explicit side effects: Clauses change “magically”
 - Indexed terms cannot be rewritten
 - Complexity lead to poor trust (actual bugs found after burn-in: 2)
- ▶ My hair started falling out!

Much Pain, No Gain!

- ▶ Measurement for IWIL-2001 in Cuba:
 - Expectation: Good performance on large search state with many shared terms
 - Reality: Performance equivalent to unshared rewriting for nearly all cases, independent from search state size
 - Explanations:
 - * Superterm and literal pointer management expensive
 - * Most terms occur in T , $|T| \ll |U|!$
 - * No sharing of rewriting over time
- ▶ Solution: Cached rewriting
 - Faster
 - Less memory consumption (no superterm management)
 - Much much much MUCH simpler (my hair grows back)

Bad Decision 2: Overoptimized Main Loop

- ▶ Original main loop (up to E 0.7):

```
while  $U \neq \{\}$ 
   $g = \text{delete\_best}(U)$ 
   $g = \text{simplify}(g, P)$ 
  if  $g == \square$ 
    SUCCESS, Proof found
  if  $g$  is not redundant w.r.t.  $P$ 
     $T = \{c \in P \mid c \text{ redundant or critically simplifiable w.r.t. } g\}$ 
     $P = (P \setminus T) \cup \{g\}$ 
     $P = \text{interreduce}(P)$ 
     $T = T \cup \text{generate}(g, P)$ 
    foreach  $c \in T$ 
       $c = \text{cheap\_simplify}(c, P)$ 
      if  $c$  is not trivial
         $U = U \cup \{c\}$ 
    SUCCESS, original  $U$  is satisfiable
```

- ▶ Inherited from DISCOUNT (UEQ prover)

- Rewriting of non-maximal terms does not necessarily require reprocessing

Some Pain, No Gain!

- ▶ Problems:
 - Interreduction necessary – code duplication
 - Processed clauses can be rewritten → Indexing of processed clauses more difficult
- ▶ Measuring tells us that non-critical simplification is rare
 - Backward simplification is rare to begin with
 - Usually $>$ -maximal terms are syntactically large, hence more likely to be rewritten
- ▶ Fix: Move to new, simpler loop
- ▶ Effect:
 - Very few proof problems affected, no significant overall difference
 - Very few problems become much harder (unstable 2nd order effect)
 - Very few problems become much easier (ditto)

Bad Decision 3: Cheap Literal Selection Implementation

- ▶ Superposition with literal selection:
 - Arbitrary negative literal in a clause can be **selected**
 - Generating inferences only allowed with this literal
 - Can significantly reduce branching factor
 - Good heuristics here one of the major strength of E
- ▶ What's wrong?
 - Literal selection only added as an afterthought (Roberto Nieuwenhuis)
 - Real significance only detected much later
 - Quick-and-dirty implementation:
 - * Fudged literal comparison function: Selected literals are always maximal
 - * Mixed very different concepts
 - * Both are useful for heuristic control
 - * Literal comparisons become invalid after selection → hard to cache
- ▶ Fix: None yet, still part of E
 - Easy to fix in principle, but much code depends on it

Top-Down Design? No, Thanks!

- ▶ Top-Down design is often advocated as opposed to “hacking”
- ▶ My experience:
 - Works fine for medium-complexity problems
 - Requires static and detailed specification
 - **Results in brittle code!**
- ▶ For complex research projects I prefer a more bottom-up approach
 - Start with a rough sketch of the design
 - Build general libraries
 - Keep interfaces small and general
 - Test individual components

Optimization

*We should forget about the small efficiencies, say about 97% of the time:
Premature optimization is the root of all evil.*

Donald Knuth (who ascribes it to Tony Hoare)

- ▶ Data types should support Big-O efficient operations
 - Use trees instead of lists
 - Consider hashes
- ▶ Don't count cycles until your profiler has
 - Often bottlenecks are non-obvious
 - Don't waste time to optimize irrelevant code
- ▶ **Never** destroy the structure of your code for speed
 - If there is a better variant, reimplement cleanly!

Generic Tools

▶ Useful:

- CVS
- Make
- gawk/Python (for support and test programs)
- Profiler (gprof)
- Good text editor (GNU Emacs for me)

▶ Useless: Debugger

- Errors often occur after millions of statements
- Data structures too complex to follow

▶ Better:

- Use C `assert()` or similar for checking simple pre/postconditions and invariants
- Include optional code for internal consistency checks
- Provide pretty-print functions for all data types

Specialized Tools

- ▶ Memory tracer
 - Keeps count of allocated and deallocated memory
 - Extended version warns against most allocation errors
 - Helps tracking down memory leaks
- ▶ Automated test environment
 - Configuration files specify parameters and test problems
 - Test runs can be local or distributed over NFS clusters
 - Normalizes computer performance based on E benchmark
 - Typically, I'm the biggest user of CPU time at TU Munich!
- ▶ Evaluation and analysis tools
 - Check consistency of test results
 - Generate automatic mode
 - Generate various statistics
- ▶ Implementation: Python, gawk, UNIX shell

Conclusion

- ▶ High-performance implementation of deduction systems is a challenging and fascinating field
 - Can be highly rewarding
 - But: Needs a lot of effort and **time**
- ▶ Keep your code base small and well-commented
 - If possible always pick a cleaner implementation
 - Build generic libraries
 - Build robust code with reasonable safeguards
- ▶ Measure before you optimize
 - Don't optimize at all costs
 - Reengineer bad code (better: throw away, reimplement)
- ▶ More work on E will be revealed after CASC-J2 (or maybe at IJCAR-2004)