Simplicity, Measuring, and Good Engineering

One Way to Build a World Class Automated Deduction System

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 too few good Implementations!

We have too 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 $>$

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

- 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)

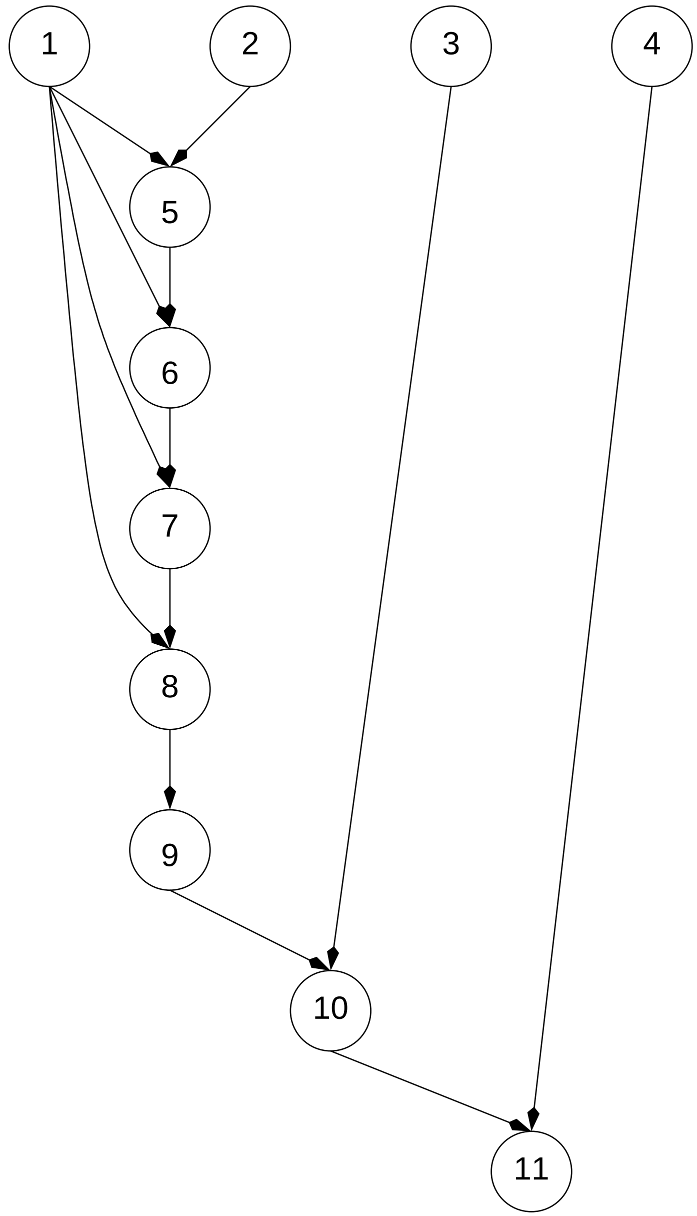
$$\frac{s \simeq t \vee S \quad u \dot{\simeq} v \vee R}{\sigma(u[p \leftarrow t] \dot{\simeq} v \vee S \vee R)}$$

- 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

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)	$f(f(f(a))) \not\simeq a \vee a \simeq b$	paramdodulate 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



Real Example

```
# Problem is unsatisfiable, constructing proof object
# TSTP exit status: Unsatisfiable
# Proof object starts here.
     1 : [++equal(j(0,X1), X1)] : initial
     2 : [++equal(j(X1,0), X1)] : initial
     4 : [++equal(j(X1,g(X1)), 0)] : initial
     5 : [++equal(j(j(X1,X2),X3), j(X1,j(X2,X3)))] : initial
     6 : [++equal(j(X1,X2), j(X2,X1))] : initial
     7 : [++equal(f(f(X1,X2),X3), f(X1,f(X2,X3)))] : initial
     8 : [++equal(f(X1,j(X2,X3)), j(f(X1,X2),f(X1,X3)))] : initial
     9 : [++equal(f(j(X1,X2),X3), j(f(X1,X3),f(X2,X3)))] : initial
    10 : [++equal(f(f(X1,X1),X1), X1)] : initial
    11 : [--equal(f(a,b), f(b,a))] : initial
    12 : [++equal(0, g(0))] : pm(1,4)
    26 : [++equal(j(0,X2), j(X1,j(g(X1),X2)))] : pm(5,4)
    28 : [++equal(j(X1,j(X2,X3)), j(X3,j(X1,X2)))] : pm(6,5)
    35 : [++equal(X2, j(X1,j(g(X1),X2)))] : rw(26,1)
    41 : [++equal(f(X1,f(X1,X1)), X1)] : rw(10,7)
    44 : [++equal(j(f(X1,j(X2,X3)),X4), j(f(X1,X2),j(f(X1,X3),X4)))] : pm(5,8)
    56 : [++equal(j(f(X1,f(X2,X3)),f(X4,X3)), f(j(f(X1,X2),X4),X3))] : pm(9,7)
    57 : [++equal(j(f(X1,X2),f(X3,f(X4,X2))), f(j(X1,f(X3,X4)),X2))] : pm(9,7)
    66 : [++equal(f(X1,X2), f(X1,f(f(X1,X1),X2)))] : pm(7,41)
    67 : [++equal(f(X1,X2), f(X1,f(X2,f(f(X1,X2),f(X1,X2)))))] : pm(7,41)
    69 : [++equal(j(X1,f(X1,X2)), f(X1,j(f(X1,X1),X2)))] : pm(8,41)
    73 : [++equal(f(X1,X2), f(X1,f(X1,f(X1,X2))))] : rw(66,7)
    74 : [++equal(f(X1,X2), f(X1,f(X2,f(X1,f(X2,f(X1,X2))))))] : rw(67,7)
    78 : [++equal(j(X1,0), g(g(X1)))] : pm(35,4)
    83 : [++equal(j(X1,j(X2,g(X1))), X2)] : pm(35,6)
    87 : [++equal(X1, g(g(X1)))] : rw(78,2)
    92 : [++equal(j(g(X1),j(X1,X2)), X2)] : pm(35,87)
   103 : [++equal(j(g(X1),j(X2,X1)), X2)] : pm(83,87)
   156 : [++equal(j(g(j(X1,X2)),X2), g(X1))] : pm(103,92)
   157 : [++equal(j(g(j(X1,X2)),X1), g(X2))] : pm(103,103)
   176 : [++equal(j(X2,g(j(X1,X2))), g(X1))] : rw(156,6)
   184 : [++equal(j(f(X1,X2),g(f(X1,j(X3,X2)))), g(f(X1,X3)))] : pm(176,8)
   185 : [++equal(j(f(X1,X2),g(f(j(X3,X1),X2))), g(f(X3,X2)))] : pm(176,9)
   186 : [++equal(j(X1,g(X2)), g(j(X2,g(X1))))] : pm(35,176)
   190 : [++equal(j(g(X1),g(X2)), g(j(X2,X1)))] : pm(92,176)
   340 : [++equal(j(X1,g(j(X1,X2))), g(X2))] : rw(157,6)
   676 : [++equal(f(f(X1,X2),X3), f(X1,f(X2,f(f(X1,X2),f(f(X1,X2),X3)))))] : pm(7,73)
   680 : [++equal(j(f(X1,X2),f(X1,X3)), f(X1,j(X2,f(X1,f(X1,X3)))))] : pm(8,73)
   685 : [++equal(f(X1,f(X2,X3)), f(X1,f(X2,f(f(X1,X2),f(f(X1,X2),X3)))))] : rw(676,7)
   686 : [++equal(f(X1,f(X2,X3)), f(X1,f(X2,f(X1,f(X2,f(X1,f(X2,X3)))))))] : rw(rw(685,7),7)
   692 : [++equal(f(X1,j(X2,X3)), f(X1,j(X2,f(X1,f(X1,X3)))))] : rw(680,8)
   693 : [++equal(f(X1,f(X1,X1)), j(X1,f(X1,0)))] : pm(69,2)
   694 : [++equal(f(X1,0), j(X1,f(X1,g(f(X1,X1)))))] : pm(69,4)
   695 : [++equal(f(j(X1,f(X1,X2)),X3), f(X1,f(j(f(X1,X1),X2),X3)))] : pm(7,69)
   713 : [++equal(X1, j(X1,f(X1,0)))] : rw(693,41)
   727 : [++equal(j(X1,g(X1)), f(g(X1),0))] : pm(35,713)
   737 : [++equal(0, f(g(X1),0))] : rw(727,4)
   772 : [++equal(f(X1,0), 0)] : pm(737,87)
   788 : [++equal(f(0,X2), f(X1,f(0,X2)))] : pm(7,772)
   818 : [++equal(j(f(0,X2),f(X3,f(0,X2))), f(j(X1,X3),f(0,X2)))] : pm(9,788)
   834 : [++equal(f(0,j(X2,X2)), f(j(X1,X3),f(0,X2)))] : rw(rw(818,788),8)
   835 : [++equal(f(0,j(X2,X2)), f(0,X2))] : rw(834,788)
   858 : [++equal(j(f(0,X1),f(0,X2)), f(0,j(j(X1,X1),X2)))] : pm(8,835)
   873 : [++equal(f(0,j(X1,X2)), f(0,j(j(X1,X1),X2)))] : rw(858,8)
   874 : [++equal(f(0,j(X1,X2)), f(0,j(X1,j(X1,X2))))] : rw(873,5)
  1077 : [++equal(f(0,j(X1,0)), f(0,j(X1,g(X1))))] : pm(874,4)
  1106 : [++equal(f(0,X1), f(0,j(X1,g(X1))))] : rw(1077,2)
  1107 : [++equal(f(0,X1), 0)] : rw(rw(1106,4),772)
  1177 : [++equal(j(X1,f(X1,g(f(X1,X1)))), 0)] : rw(694,772)
  1184 : [++equal(j(f(X1,g(f(X1,X1))),g(0)), g(X1))] : pm(176,1177)
  1200 : [++equal(f(X1,g(f(X1,X1))), g(X1))] : rw(rw(1184,12),2)
  1219 : [++equal(f(X1,f(X1,g(X1))), f(X1,g(f(X1,X1))))] : pm(73,1200)
  1226 : [++equal(f(X1,f(X1,g(X1))), g(X1))] : rw(1219,1200)
  1233 : [++equal(g(f(X1,X2)), f(X1,f(X2,f(f(X1,X2),g(f(X1,X2))))))] : pm(7,1226)
  1235 : [++equal(j(g(X1),f(X1,X2)), f(X1,j(f(X1,g(X1)),X2)))] : pm(8,1226)
  1248 : [++equal(g(f(X1,X2)), f(X1,f(X2,f(X1,f(X2,g(f(X1,X2)))))))] : rw(1233,7)
  2003 : [++equal(j(f(X1,X2),g(X4)), j(f(X1,j(X2,X3)),g(j(X4,f(X1,X3)))))] : pm(44,176)
  2690 : [++equal(j(X1,f(X2,X1)), f(j(f(X1,X1),X2),X1))] : pm(56,41)
  2786 : [++equal(f(j(X2,f(X1,X1)),X1), j(X1,f(X2,X1)))] : pm(2690,6)
  2969 : [++equal(f(j(X2,f(X1,X2)),X3), f(j(X1,f(X2,X2)),f(X2,X3)))] : pm(7,2786)
  3034 : [++equal(j(f(X1,f(X2,X2)),f(X3,X2)), f(j(X1,f(X3,X2)),f(X2,X2)))] : pm(57,41)
  3095 : [++equal(f(j(f(X1,X2),X3),X2), f(j(X1,f(X3,X2)),f(X2,X2)))] : rw(3034,56)
  5592 : [++equal(j(f(X1,g(X2)),g(f(X1,0))), g(f(X1,X2)))] : pm(184,4)
  5694 : [++equal(f(X1,g(X2)), g(f(X1,X2)))] : rw(rw(rw(5592,772),12),2)
```

```
  5757 : [++equal(j(f(X1,g(X2)),g(X3)), g(j(X3,f(X1,X2))))] : pm(190,5694)
  6619 : [++equal(j(f(X1,X2),f(j(X3,X1),g(X2))), g(f(X3,X2)))] : rw(185,5694)
  6620 : [++equal(j(f(X1,X2),f(j(X3,X1),g(X2))), f(X3,g(X2)))] : rw(6619,5694)
  6623 : [++equal(j(f(g(X1),X2),f(0,g(X2))), f(X1,g(X2)))] : pm(6620,4)
  6727 : [++equal(f(g(X1),X2), f(X1,g(X2)))] : rw(rw(6623,1107),2)
  6837 : [++equal(f(j(X2,g(X1)),X3), f(j(X1,g(X2)),g(X3)))] : pm(6727,186)
  7002 : [++equal(f(j(X2,g(f(g(X1),g(X1)))),X1), j(g(X1),f(g(X2),g(X1))))] : pm(2690,6837)
  7053 : [++equal(f(j(X2,f(X1,g(X1))),X1), j(g(X1),f(g(X2),g(X1))))] : rw(rw(rw(7002,5694),87),6727)
  7054 : [++equal(f(j(X2,f(X1,g(X1))),X1), j(g(X1),f(X2,X1)))] : rw(rw(7053,6727),87)
  7250 : [++equal(f(j(g(X1),f(X1,X2)),X3), f(X1,f(j(f(X1,g(X1)),X2),X3)))] : pm(7,1235)
 13084 : [++equal(f(X1,j(g(X2),f(f(X1,X1),X2))), f(j(X1,f(X1,f(X2,g(X2)))),X2))] : pm(695,7054)
 13204 : [++equal(0, f(j(X1,f(X1,f(X2,g(X2)))),X2))] : rw(rw(rw(rw(rw(13084,7),692),6),4),772)
 13442 : [++equal(f(0,X3), f(j(X1,f(X1,f(X2,g(X2)))),f(X2,X3)))] : pm(7,13204)
 13525 : [++equal(0, f(j(X1,f(X1,f(X2,g(X2)))),f(X2,X3)))] : rw(13442,1107)
 18184 : [++equal(f(j(f(X2,X3),X1),f(X3,X3)), f(j(f(X1,X3),X2),X3))] : pm(3095,6)
 20946 : [++equal(f(X1,f(j(X2,f(X2,f(X1,g(X1)))),f(X1,0))), f(X1,j(X2,f(X2,f(X1,g(X1))))))] : pm(74,13525)
 20959 : [++equal(f(j(g(X1),f(X1,g(f(X2,g(X2))))),f(X2,X3)), 0)] : pm(13525,6727)
 21127 : [++equal(0, f(X1,j(X2,f(X2,f(X1,g(X1))))))] : rw(rw(rw(20946,772),772),772)
 21139 : [++equal(f(j(g(X1),f(X1,f(X2,X2))),f(X2,X3)), 0)] : rw(rw(20959,5694),87)
 21237 : [++equal(f(X1,j(f(X2,X3),f(X2,f(X3,f(X1,g(X1)))))), 0)] : pm(21127,7)
 21274 : [++equal(f(X1,j(g(X2),f(X2,g(f(X1,g(X1)))))), 0)] : pm(21127,6727)
 21337 : [++equal(f(X1,f(X2,j(X3,f(X3,f(X1,g(X1)))))), 0)] : rw(21237,8)
 21373 : [++equal(f(X1,j(g(X2),f(X2,f(X1,X1)))), 0)] : rw(rw(21274,5694),87)
 38527 : [++equal(f(X1,f(X2,f(X1,f(X2,f(X1,g(X2)))))), g(f(X1,X2)))] : rw(1248,5694)
 38528 : [++equal(f(X1,f(X2,f(X1,f(X2,f(X1,g(X2)))))), f(X1,g(X2)))] : rw(38527,5694)
 98960 : [++equal(f(X1,f(j(X2,f(f(X1,g(X1)),X2)),X3)), f(j(g(X1),f(X1,f(X2,X2))),f(X2,X3)))] : pm(7250,2969)
 99364 : [++equal(f(X1,f(j(X2,f(X1,f(X1,g(X2)))),X3)), f(j(g(X1),f(X1,f(X2,X2))),f(X2,X3)))] : rw(rw(98960,7),6727)
 99365 : [++equal(f(X1,f(j(X2,f(X1,f(X1,g(X2)))),X3)), 0)] : rw(99364,21139)
 99830 : [++equal(f(j(X1,f(X2,f(X2,g(X1)))),f(X3,j(X2,0))), 0)] : pm(21337,99365)
100176 : [++equal(f(j(X1,f(X2,f(X2,g(X1)))),f(X3,X2)), 0)] : rw(99830,2)
105304 : [++equal(0, j(f(X1,X2),f(f(X2,f(X2,g(f(f(X1,X2),f(X1,X2))))),f(X1,X2))))] : pm(2690,100176)
105706 : [++equal(0, j(f(X1,X2),f(X2,f(X2,f(X1,g(X2))))))] : rw(rw(rw(rw(rw(rw(rw(rw(rw(rw(rw(rw(105304,7),5694),5694),5694),7),7),7),7),7),6727),5694),38528)
123143 : [++equal(j(f(X1,X2),g(0)), g(f(X2,f(X2,f(X1,g(X2))))))] : pm(340,105706)
123573 : [++equal(f(X1,X2), g(f(X2,f(X2,f(X1,g(X2))))))] : rw(rw(123143,12),2)
123574 : [++equal(f(X1,X2), f(X2,f(X2,f(X1,X2))))] : rw(rw(rw(rw(123573,5694),5694),5694),87)
124051 : [++equal(f(f(X2,X1),X3), f(X1,f(f(X1,f(X2,X1)),X3)))] : pm(7,123574)
124056 : [++equal(j(f(X1,X2),f(X3,X1)), f(X1,j(X2,f(X1,f(X3,X1)))))] : pm(8,123574)
124364 : [++equal(f(X2,f(X1,X3)), f(X1,f(f(X1,f(X2,X1)),X3)))] : rw(124051,7)
124365 : [++equal(f(X2,f(X1,X3)), f(X1,f(X1,f(X2,f(X1,X3)))))] : rw(rw(124364,7),7)
126375 : [++equal(f(X1,f(X1,X2)), f(X2,f(X1,f(X2,f(X1,X2)))))] : pm(124365,74)
132278 : [++equal(f(X1,f(X1,j(X2,f(X3,X1)))), j(f(X1,f(X1,X2)),f(X3,X1)))] : pm(124056,8)
196237 : [++equal(j(0,g(j(X3,f(X1,f(X2,f(X1,X1)))))), j(f(X1,g(X2)),g(X3)))] : pm(2003,21373)
196902 : [++equal(g(j(X3,f(X1,f(X2,f(X1,X1))))), j(f(X1,g(X2)),g(X3)))] : rw(196237,1)
196903 : [++equal(g(j(X3,f(X1,f(X2,f(X1,X1))))), g(j(X3,f(X1,X2))))] : rw(196902,5757)
279981 : [++equal(g(f(X1,f(X2,f(X1,X1)))), g(j(0,f(X1,X2))))] : pm(196903,1)
280420 : [++equal(f(X1,f(X2,f(X1,g(X1)))), g(j(0,f(X1,X2))))] : rw(rw(rw(279981,5694),5694),5694)
280421 : [++equal(f(X1,f(X2,f(X1,g(X1)))), f(X1,g(X2)))] : rw(rw(280420,1),5694)
281014 : [++equal(f(g(X1),f(X2,f(g(X1),X1))), f(g(X1),g(X2)))] : pm(280421,87)
281405 : [++equal(f(X1,f(X2,f(X1,X1))), f(g(X1),g(X2)))] : rw(rw(rw(rw(rw(281014,6727),6727),5694),5694),87)
281406 : [++equal(f(X1,f(X2,f(X1,X1))), f(X1,X2))] : rw(rw(281405,6727),87)
281924 : [++equal(f(f(X1,X2),X3), f(X1,f(f(X2,f(X1,X1)),X3)))] : pm(7,281406)
281928 : [++equal(j(f(X1,X2),f(X1,X3)), f(X1,j(f(X2,f(X1,X1)),X3)))] : pm(8,281406)
282022 : [++equal(f(X1,f(X2,f(X1,f(X2,X1)))), f(X1,f(X2,X2)))] : pm(686,281406)
282291 : [++equal(f(X1,f(X2,X3)), f(X1,f(f(X2,f(X1,X1)),X3)))] : rw(281924,7)
282292 : [++equal(f(X1,f(X2,X3)), f(X1,f(X2,f(X1,f(X1,X3)))))] : rw(rw(282291,7),7)
282297 : [++equal(f(X1,j(X2,X3)), f(X1,j(f(X2,f(X1,X1)),X3)))] : rw(281928,8)
282412 : [++equal(f(X2,f(X2,X1)), f(X1,f(X2,X2)))] : rw(282022,126375)
294341 : [++equal(f(X1,f(j(f(X2,X1),X3),X1)), f(X1,j(X2,f(X3,X1))))] : pm(282297,56)
311251 : [++equal(j(f(X2,f(X2,X1)),f(X3,X2)), f(j(f(X1,X2),X3),X2))] : rw(rw(3095,282412),132278)
334509 : [++equal(f(f(j(f(X1,X2),X3),X2),f(f(j(f(X1,X2),X3),X2),f(X2,j(X1,f(X3,X2))))), f(X2,f(j(f(X1,X2),X3),X2)))] : pm(123574,294341)
335356 : [++equal(f(j(f(X3,X2),X1),X2), f(X2,f(j(f(X1,X2),X3),X2)))] : rw(rw(rw(rw(rw(rw(334509,7),132278),311251),7),282292),18184)
335357 : [++equal(f(j(f(X3,X2),X1),X2), f(X2,j(X1,f(X3,X2))))] : rw(335356,294341)
335871 : [++equal(f(X3,X2), f(X2,j(j(g(f(X1,X2)),X3),f(X1,X2))))] : pm(335357,35)
336649 : [++equal(f(X3,X2), f(X2,X3))] : rw(rw(rw(rw(rw(rw(rw(335871,5694),5),28),44),4),772),1)
338431 : [] : ar(11,6,5,7,336649)
338432 : [] : 338431 : 'proof'
# Proof object ends here.
```

- Small example:
  - ≈150 inferences in proof
  - ≈500000 inferences in proof search
  - Run time 30 seconds on 1GHz G4 Powerbook
  - Today, 40 times bigger proof runs routine!

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 elemination 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 $U \neq \{\}$
  $g =$ 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 $c \in T$
      $c =$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 $U \neq \{\}$
  $g =$ 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

- 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 $\approx$20000 in 1999)
    - Lots of comments
    - Lots of vertical white space for formatting
  - Approximately 2500 exported functions ($<$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 occurences 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 occured
  - Serious complications with collapsing rules ($f(a) \simeq a$) will not directly eleminate $f(a)$ 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
  - Explantions:
    * Superterm and literal pointer management expensive
    * Most terms occur in T, $|T| << |U|$!
    * 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 $U \neq \{\}$
    $g =$ 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 critically simplifiable w.r.t. $g\}$
      $P = (P \backslash T) \cup \{g\}$
      $P =$interreduce(P)
      $T = T \cup$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

- 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 overal 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

- Superpositition 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 afterthough (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

Premature optimization is the root of all evil.

Donald Knuth

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)

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 to complex to follow
- Better:
  - Use C `assert()` or similar for checking simpe 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)