How and Why E Works !

or

Hand-optimized Z80 assembly code

Stephan Schulz

November 11, 2000

Introduction

E: Successful high-performance equational theorem prover for full clausal logic

- Based on paramodulation and rewriting (modified superposition calculus)
- Fully automatic (no user interaction during run time)
- Emphasis on search control heuristics
- Available in source form (compiles on all known current UNIX versions) at

  `http://wwwjessen.informatik.tu-muenchen.de/~schulz/WORK/eprover.html`
- Current stable release: E 0.6 *Kanchanjangha*
- What you all want to know: Releases are named after tea gardens

Overview

- Calculus
- Search organization
- Implementation
- Search control
- Lessons learned

– Calculus –

Superposition (with literal selection)

- Saturating calculus, works on (sets of) *clauses* in equational representation
- Generating inferences:
  - Superposition
  - Equality resolution
  - Equality factoring
- Most important inference ($>$ 95% of new clauses): Superposition (paramodulation constrained by simplification ordering)
- Overlap (only) maximal terms of maximal positive literals into maximal terms of (maximal *xor* selected) literals

Literal selection

- Completeness-maintaining refinement of superposition/resolution
- (Informal) idea: View clause as set of *conditional equations*:
  - $l_1 = r_1 \vee l_2 = r_2 \vee s_1 \neq t_1 \vee s_2 \neq t_2$
    $\hat{=} \{l_1 = r_1 \leftarrow l_2 \neq r_2 \wedge s_1 = t_1 \wedge s_2 = t_2,$
    $\quad l_2 = r_2 \leftarrow l_1 \neq r_1 \wedge s_1 = t_1 \wedge s_2 = t_2\}$
  - We do not know which negative condition (i.e. positive literal) we need to solve
  - But: We need to solve *all* positive conditions (i.e. negative literals)
  - Ergo: We can pick *one* arbitrary negative literal as the first one to solve
- More generally: If there is at least one negative literal in a clause, inferences *can* be restricted to an arbitrary subset of **selected** literals containing at least one negative literal (*Don't-care*-nondeterminism)
- We do not need to paramodulate *from* a clause with selected literals at all

Reducing redundancy

- Redundancy criteria (implemented as *contracting inference rules*) allow us to simplify or eliminate clauses
- Useful mechanisms:
  - Eliminating duplicate/trivial literals: $\{u=v \lor v=u\} \Longrightarrow \{u=v\}$ and $\{u \neq u\} \Longrightarrow \{\Box\}$
  - Rewriting (using oriented unit-clauses): $\{a=b, f(a)=c\} \Longrightarrow \{a=b, f(b)=c\}$ (if $a > b$)
  - Tautology elimination (only partially possible): $\{a=b \lor a \neq b\} \Longrightarrow \{\}$
  - Subsumption (Powerful, but often expensive): $\{f(x)=f(y), f(a)=f(b) \lor s=t\} \Longrightarrow \{f(x)=f(y)\}$
  - *Simplify-reflect* (generalized equational unit-resolution): $\{g(x)=g(y), f(g(a)) \neq f(g(b)) \lor s=t\} \Longrightarrow \{g(x)=g(y), s=t\}$
- Important for efficiency: Apply contracting rules early and often

– Search Organization –

Aims of a Proof Procedure

- Fast (do as little as possible)
- Lazy (delay work as much as possible)
- Minimal (avoid redundant steps)
- Complete (do as much as necessary)
- Intelligent (allow good use of guiding heuristics)

Our Solution (DISCOUNT loop)

- Variant of *given-clause* algorithm
- Strict separation of active and passive clauses
- Invariants of active (or *processed*) clause set **P**:
  - All clauses maximally simplified (with respect to **P**)
  - Contains no clauses that are (provably) redundant within **P**
  - All generating inferences between clauses in **P** performed
- Passive or unprocessed clauses (**U**)
  - Sorted by one or more heuristic evaluations (*best-first* access)
  - Clauses simplified once (at creation) with then current **P**
  - No expensive redundancy checks (e.g. full tautology, non-unit subsumption)
- Initial state: **P** empty, axioms (and goals) in **U**

DISCOUNT loop (continued)

while(**U** $\neq \emptyset$)

  g := remove_best(**U**)

  simplify **g** with **P**

  if(**g** $= \Box$) success; end;endif

  if(**g** is not redundant in **P**)

    simplify **P** with **g** and interreduce **P** $\cup \{$**g**$\}$

    (remove redundant clauses, move critically affected clauses into **U**, kill offspring)

    T := generate(**P**,**g**)

    P := **P** $\cup \{$**g**$\}$

    simplify **T** with **P**

    U := **U** $\cup$ **T**

  endif

endwhile

Failure, initial clause set is satisfiable

– Implementation –

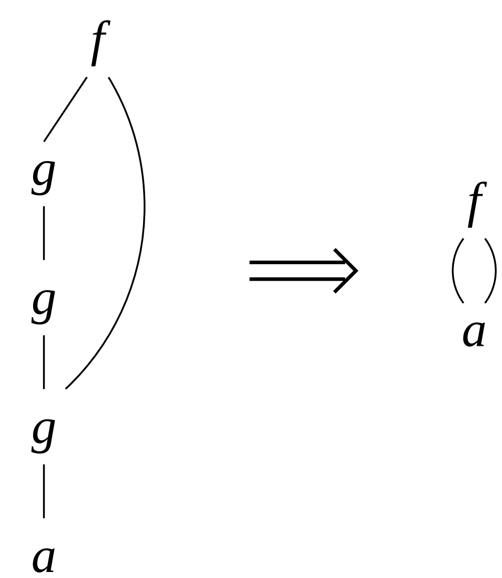
Some Important Aspects

- Term representation
- Indexing
- Subsumption
- Memory management

Term Representation

- Terms in E are perfectly shared – no term ever is represented twice unless the two copies are distinguished by the calculus (saves 90%–99.9% of all term nodes)
- *Heavy* terms: Precomputed weights, normal form time stamps, reducibility status, . . .
- Rewriting is performed directly on the shared representation:

$$f(g(g(g(a),g(a)))) \Longrightarrow_{g(x)\to x} f(a,a) \qquad \hat{=}$$



Example: Shared Terms & Forward Rewriting

- Situation (Simplifying newly generated clauses):
  - Given: System **R** of rewrite rules (and equations)
  - Given: (Large) set **T** of new clause (generated by superposition) and (larger) set **U** of old unprocessed clauses
  - Aim: Find **R**-normal forms of all terms in **T**
- Benefits of shared rewriting:
  - Typical: Many common terms in **T**

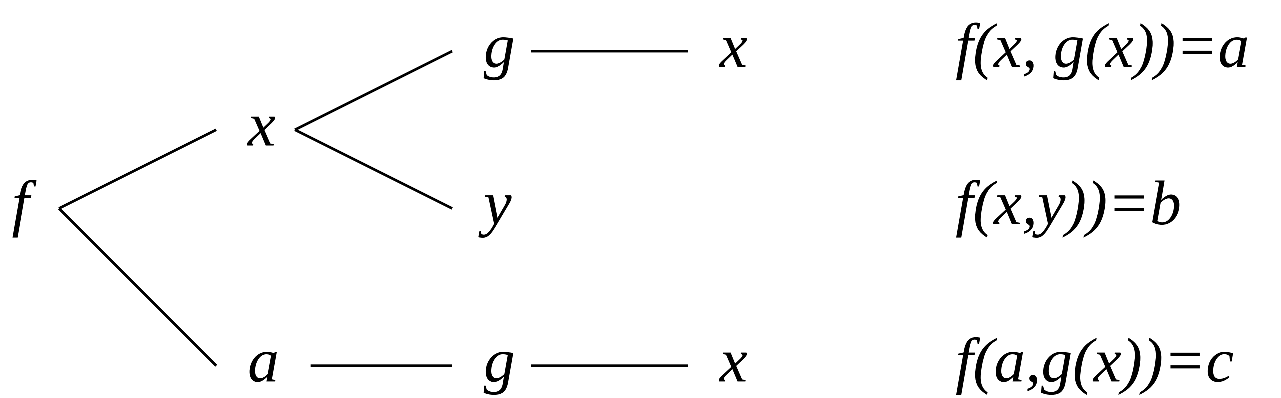
    $\Rightarrow$ Each term only needs to be considered once
  - Typical: Many common terms in **T** and **U**

    $\Rightarrow$ Many terms are already in normal form and are recognized as such
  - Many remaining common terms are in normal form with respect to earlier **R**

    $\Rightarrow$ Age-constraints speed up matching (see below)
  - Instances of common terms in **U** are rewritten for free

Indexing in E

- Indexing: Speed up clause-set/clause or clause-set/clause-set operations
- E currently indexes only processed unit clauses
- Index used for rewriting, unit-subsumption, simplify-reflect
- Index type: Perfect discrimination tree
  - Consider term as linear word
  - Branch on differences in words



Constrained Perfect Discrimination Trees

- Aim: Speed up matching (in particular for rewriting)
- Size constraints:
  - A term can only match terms that are of at least equal size
- Age constraints:
  - Reducibility relation becomes monotonically stronger over time
  - Terms in normal form with respect to clauses at time t can at most be rewritten by younger clauses
- How to use this:
  - Annotate branches in discrimination tree with age of youngest indexed clause and size of smallest indexed left hand side
  - Consider only branches with satisfiable constraints during matching

Subsumption

- Unit-subsumption:
  - Index on maximal terms of candiate clauses
  - Sequential search for matching right hand side
  - Only unorientable clauses are candidates (otherwise: Rewriting possible)
- Non-unit-subsumption:
  - Sequential search through candidate clauses
  - Pre-tests (literal numbers, symbol count, individual literal matches with literal symbol count) for filtering
  - Standard backtracking algorithm (modulo symmetry of equations)
  - Rather slow, but application kept to a minimum by proof procedure
    $\Rightarrow$ Overall cost is acceptable

Memory management

- Keeping memory manageable:
  - Prover keeps track of approximate memory use (term cells, literals, clauses)
  - If memory reaches predefined limit, 50% of clauses are discarded
  - Discarding: Simulate future clause selection strategy, delete the clauses that would be selected last
- Managing memory:
  - Intermediate abstract layer above C standard `malloc`
  - Keeps free lists for memory blocks of relevant sizes, reuse freed blocks
  - Complex operations (reorganization of free memory) left to C standard library
  - Optionally: Full logging of memory operations and/or counting of allocated/freed blocks

– Search Control –

Don't-care-Nondeterminism $\equiv$ Chances for Heuristics

- Important choice points for E:
  - Simplification ordering
  - Clause selection
  - Literal selection
- Other choice points:
  - Choice of rewrite relation (usually strongest, don't care *which* normal form)
  - Application of rewrite relation to terms (leftmost-innermost, strongly suggested by shared terms)

Simplification Orderings

- Implemented: Knuth-Bendix-Orderings, Lexicographic Path Orderings
- Precedence: Fully user defined or simple algorithms
  - **Sorted by arity** (higher arity $\rightarrow$ larger)
  - **Sorted by arity, but unary first**
  - Sorted by inverse arity
  - Sorted by frequency of appearance in axioms
  - ...
- Weights for KBO: Similar simple algorithms (constant weights (optionally weight 0 for maximal symbol), arity, position in precedence ...)
- No good automatic selection of orderings yet – auto mode switches between two simple KBO schemes

Clause selection

- Most important choice point
- Probably also hardest chocice (find best clause among millions)
- Implementation in E: Multiple priority queues sorted by heuristic evaluation and strategy-defined priority
- Selection in weighted round-robin-scheme (generalizes *pick-given* ratio)
- Example: 8*Refinedweight(PreferGoals,1,2,2,3,0.8),
  8*Refinedweight(PreferNonGoals,2,1,2,3,0.8),
  1*Clauseweight(ConstPrio,1,1,0.7),
  1*FIFOWeight(ByNegLitDist)
- Auto-mode: Select between strategies based on problem features (Unit/Horn/General, Pure/Some/no equality, number of clauses, . . .)
- Auto-mode automatically generated from test results

Literal Selection

- Problem: Which literals should be selected for inferences in a clause?
- Ideas:
  - Select hard literals first (if we cannot solve this, the clause is useless)
  - Select small literals (fewer possible overlaps)
  - Select ground literals (no instantiation, most unit-overlaps eleminated by rewriting)
  - Propagate inference literals to children clauses (inheritance)
- Problem: Should we always select literals if possible?
  - Only select if no unique maximal literal exists
  - Do not select in conditional rewrite rules
- Surprisingly successful: Additional selection of maximal positive literals
- See E source code for large number of things we have tried...

– Lessons Learned –

Development, Bondage, and Discipline

- Develop around a small number of strong central ideas
  - Shared terms and shared rewriting
  - DISCOUNT loop
  - Round-robin clause selection
- Later add-ons can destroy your code base !
  - Only add features that fit well into your framework
  - Minimize ad-hoc hacks
  - Note: One often cannot judge the value of a technique from a prototypical implementation anyways
- **Don’t hesitate to throw code away**

Optimization

- Use profiling tools to find hot spots (guessing is bad!)
  - DISCOUNT: Unification $<$1%, list management up to 50%
- Don't over-optimize (it may backfire anyways)
- Optimize overall performance, not (perhaps) useless operations
  - (Perhaps avoid) Repeated simplification of unprocessed clauses
  - (Perhaps avoid) Full redundancy tests on unprocessed clauses
- $O()$ is important, k is not (as long as it is small enough)
- Memory is more of a bottleneck than CPU

Search Control

- If inference speed is sufficient, search control is crucial for performance
- Intuition is often wrong – experimental evaluation is *very* important
- Allow for flexible use of heuristics at important choice points
- Experiences from other provers do not always apply
- Distrust authority and legends
  - Especially older folklore does not always hold for current hardware/software
  - 5 is not the optimal *pick-given ratio*

Assorted Topics – Conclusion

- Perfectly shared terms with shared rewriting are a success
  - Useful in forward and backward rewriting
  - Gives many of the benefits of more involved indexing techniques
  - Saves memory (no. of terms cells $\approx$ no. of literals)
  - But: Using 50% of the potential gain made me loose 50% of my hair. . .
- Don't hesitate to ~~steal~~ accept good ideas from other people or provers
- Future work . . .

  . . . will we revealed after CASC-18 **;-)**