E 0.8x

User Manual

–preliminary version–

Stephan Schulz

August 1, 2003

Abstract

E is an equational theorem prover for full clausal logic, based on superposition and rewriting. In this *very preliminary* manual we first give a short introduction for impatient new users, and then cover calculus, control, options and input/output of the prover in some more detail.

Contents

1 Introduction

This is a short and very sketchy documentation to the E equational theorem prover. E is an purely equational theorem prover for clausal logic with equality. It is based on paramodulation and rewriting. This means that E reads a set of clauses and saturates it by systematically applying a number of inference rules until either all possible inferences have been performed or until the empty clause has been derived, i.e. the clause set has been found to be unsatisfiable and thus a conjecture has been proved.

E is still a moving target, but most recent releases have been quite stable, and it the prover is being used productively by serveral independent groups of people. This manual should enable you to experiment with the prover and to use some of its more advanced features.

The manual assumes a working knowledge of refutational theorem proving, which can be gained from e.g. [CL73]. For a short description of E including performance data, see [Sch01], a more detailed description will be published as [Sch02]. Most papers on E and much more information is available at or a few hops away from the E homepage, `http://www4.informatik.tu-muenchen.de/~schulz/WORK/eprover.html`.

Some other provers have influenced the design of E and may be referenced in the course of this manual. These include SETHEO [MIL+97], Otter [McC94, MW97], SPASS [WGR96, WAB+99], DISCOUNT [DKS97], Waldmeister [HBF96, HJL99] and Vampire [RV02, RV01].

2 Getting Started

Installation of E should be straightforward. The file `README` in the main directory of the distribution contains the necessary information. After building, you will find the standalone executable `E/PROVER/eprover`.

E is controlled by a very wide range of parameters. However, if you do not want to bother with the details, you can leave configuration for a problem to the prover. To use this feature, use the following command line options:

`-xAuto`	Select a literal selection strategy and a selection heuristic automagically (based on problem features).
`-tAuto`	Select a term ordering automagically.
`--memory-limit=xx`	Tell the prover how much memory (measured in MB) to use at most. In automatic mode E will optimize its behaviour for this amount (20 MB will work, 64 MB is reasonable, 192 MB is what I use. *More is better*[1], but if you go over your physical memory, you will probably experience *very* heavy swapping.).

Example: If you happen to have a workstation with 64 MB RAM[2], the following command is reasonable:

```
eprover -xAuto -tAuto --memory-limit=48 PUZ031-1+rm_eq_rstfp.lop
```

This documentation will probably lag behind the development of the latest version of the prover for quite some time. To find out more about the options available, type `eprover --help` (or consult the source code included with the distribution).

3 Calculus and Proof Procedure

E is a purely equational theorem prover, based on ordered paramodulation and rewriting. As such, it implements an instance of the superposition calculus described in [BG94]. We have extended the calculus with some stronger contraction rules and more general approach to literal selection. The proof procedure is a variant of the *given-clause* algorithm.

3.1 Calculus

$Term(F, V)$ denotes the set of (first order) *terms* over a finite set of function symbols F (with associated arities) and an enumerable set of variables V. We write $t|_p$ to denote the subterm of t at a position p and write $t[p \leftarrow t']$ to denote t with $t|_p$ replaced by t'. An equation $s \simeq t$ is an (implicitly symmetrical) pair of terms. A positive literal is an equation $s \simeq t$, a negative literal is a negated equation $s \not\simeq t$. We write $s \dot{\simeq} t$ to denote an arbitrary literal[3]. Literals

[1] Emphasis added for E 0.7 and up, which globally cache rewrite steps.

[2] Yes, this is outdated. If it still applies to you, get a new computer! It will still work ok, though.

[3] Nonequational literals are encoded as equations or disequations of the form $P(t_1, \ldots, P_n) \dot{\simeq} \top$. In this case, we treat predicate symbols as special function symbols that can only occur at the top-most positions and demand that atoms (terms formed with a top predicate symbol) cannot be unified with a first-order variable from V, i.e. we treat normal terms and predicate terms as two disjoint types.

can be represented as multi-sets of multi-sets of terms, with $s \simeq t$ represented as $\{\{s\}, \{t\}\}$ and $s \not\simeq t$ represented as $\{\{s, t\}\}$. A *ground reduction ordering* $>$ is a Noetherian partial ordering that is stable w.r.t. the term structure and substitutions and total on ground terms. $>$ can be extended to an ordering $>_l$ on literals by comparing the multi-set representation of literals with $\gg\gg$ (the multi-set-multi-set extension of $>$).

Clauses are multi-sets of literals. They are usually represented as disjunctions of literals, $s_1 \dot{\simeq} t_1 \vee s_2 \dot{\simeq} t_2 \ldots \vee s_n \dot{\simeq} t_n$. We write $Clauses(F, P, V)$ to denote the set of all clauses with function symbols F, predicate symbols P and variable V. If $\mathcal{C}$ is a clause, we denote the (multi-)set of positive literals in $\mathcal{C}$ by $\mathcal{C}^+$ and the (multi-)set of negative literals in $\mathcal{C}$ by $\mathcal{C}^-$

The introduction of an extended notion of *literal selection* has improved the performance of E significantly. The necessary concepts are explained in the following.

Definition 3.1 (Selection functions)
$sel : Clauses(F, P, V) \rightarrow Clauses(F, P, V)$ is a *selection function*, if it has the following properties for all clauses $\mathcal{C}$:

- $sel(\mathcal{C}) \subseteq \mathcal{C}$.
- If $sel(\mathcal{C}) \cap \mathcal{C}^- = \emptyset$, then $sel(\mathcal{C}) = \emptyset$.

We say that a literal $\mathcal{L}$ is *selected* (with respect to a given selection function) in a clause $\mathcal{C}$ if $\mathcal{L} \in sel(\mathcal{C})$. ◀

We will use two kinds of restrictions on deducing new clauses: One induced by ordering constraints and the other by selection functions. We combine these in the notion of *eligible literals*.

Definition 3.2 (Eligible literals)
Let $\mathcal{C} = \mathcal{L} \vee \mathcal{R}$ be a clause, let σ be a substitution and let sel be a selection function.

- We say $\sigma(\mathcal{L})$ is *eligible for resolution* if either
  - $sel(C) = \emptyset$ and $\sigma(\mathcal{L})$ is $>_L$-maximal in $\sigma(\mathcal{C})$ or
  - $sel(C) \neq \emptyset$ and $\sigma(\mathcal{L})$ is $>_L$-maximal in $\sigma(sel\mathcal{C}) \cap \mathcal{C}^-)$ or
  - $sel(C) \neq \emptyset$ and $\sigma(\mathcal{L})$ is $>_L$-maximal in $\sigma(sel(\mathcal{C}) \cap \mathcal{C}^+))$.
- $\sigma(\mathcal{L})$ is *eligible for paramodulation* if $\mathcal{L}$ is positive, $sel(C) = \emptyset$ and $\sigma(\mathcal{L})$ is strictly $>_L$-maximal in $\sigma(\mathcal{C})$.

◀

The calculus is represented in the form of inference rules. For convenience, we distinguish two types of inference rules. For *generating* inference rules, written with a single line separating preconditions and results, the result is added to the set of all clauses. For *contracting* inference rules, written with a double line, the result clauses are substituted for the clauses in the precondition. In the following, u, v, s and t are terms, σ is a substitution and R, S and T are (partial) clauses. p is a position in a term and λ is the empty or top-position. Different clauses are assumed to not share any common variables.

Definition 3.3 (The inference system SP)
Let $>$ be a total simplification ordering (extended to orderings $>_L$ and $>_C$ on literals and clauses) and let *sel* be a selection function. The inference system **SP** consists of the following inference rules:

- *Equality Resolution*:

$$\text{(ER)}\ \frac{u \not\simeq v \vee R}{\sigma(R)}$$

if $\sigma = mgu(u, v)$ and $\sigma(u \not\simeq v)$ is eligible for resolution.

- *Superposition into negative literals*:

$$\text{(SN)}\ \frac{s \simeq t \vee S \quad u \not\simeq v \vee R}{\sigma(u[p \leftarrow t] \not\simeq v \vee S \vee R)}$$

if $\sigma = mgu(u|_p, s)$, $\sigma(s) \not< \sigma(t)$, $\sigma(u) \not< \sigma(v)$, $\sigma(s \simeq t)$ is eligible for paramodulation, $\sigma(u \not\simeq v)$ is eligible for resolution, and $u|_p \notin V$.

- *Superposition into positive literals*:

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

if $\sigma = mgu(u|_p, s)$, $\sigma(s) \not< \sigma(t)$, $\sigma(u) \not< \sigma(v)$, $\sigma(s \simeq t)$ is eligible for paramodulation, $\sigma(u \not\simeq v)$ is eligible for resolution, and $u|_p \notin V$.

- *Equality factoring*:

$$\text{(EF)}\ \frac{s \simeq t \vee u \simeq v \vee R}{\sigma(t \not\simeq v \vee u \simeq v \vee R)}$$

if $\sigma = mgu(s, u)$, $\sigma(s) \not> \sigma(t)$ and $\sigma(s \simeq t)$ eligible for paramodulation.

- *Rewriting of negative literals*:

$$\text{(RN)}\ \frac{s \simeq t \quad u \not\simeq v \vee R}{s \simeq t \quad u[p \leftarrow \sigma(t)] \not\simeq v \vee R}$$

if $u|_p = \sigma(s)$ and $\sigma(s) > \sigma(t)$.

- *Rewriting of positive literals*[4]:

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

if $u|_p = \sigma(s)$, $\sigma(s) > \sigma(t)$, and if $u\simeq v$ is not eligible for resolution or $u \not> v$ or $p \neq \lambda$.

- *Clause subsumption*:

$$\text{(CS)}\quad \frac{T \quad R \vee S}{T}$$

if $\sigma(S) = T$ for a substitution σ or $\forall s \simeq t \in \sigma(S) : s \simeq t \in T$ for a substitution σ that is not a variable renaming.

- *Equality subsumption*:

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

- *Positive simplify-reflect*[5]:

$$\text{(PS)}\quad \frac{s\simeq t \quad u[p \leftarrow \sigma(s)]\not\simeq u[p \leftarrow \sigma(t)] \vee R}{s\simeq t \qquad R}$$

- *Negative simplify-reflect*

$$\text{(NS)}\quad \frac{s\not\simeq t \quad \sigma(s)\simeq\sigma(t) \vee R}{s\simeq t \qquad R}$$

- *Tautology deletion*:

$$\text{(TD)}\quad \frac{C}{}$$

if C is a tautology[6]. try to detect

[4] A stronger version of (RP) is proven to maintain completeness for Unit and Horn problems and is generally believed to maintain completeness for the general case as well [Bac98]. However, the proof of completeness for the general case seems to be rather involved, as it requires a very different clause ordering than the one introduced [BG94], and we are not aware of any existing proof in the literature. The variant rule allows rewriting of maximal terms of maximal literals under certain circumstances:

$$\text{(RP')}\quad \frac{s\simeq t \quad u\simeq v \vee R}{s\simeq t \quad u[p \leftarrow \sigma(t)]\simeq v \vee R}$$

if $u|_p = \sigma(s)$, $\sigma(s) > \sigma(t)$ and if $u\simeq v$ is not eligible for resolution or $u \not> v$ or $p \neq \lambda$ or σ is not a variable renaming.

This stronger rule is implemented successfully by both E and SPASS [Wei99].

[5] In practice, this rule is only applied if $\sigma(s)$ and $\sigma(t)$ are $>$-incomparable – in all other cases this rule is subsumed by (RN) and the deletion of resolved literals (DR).

tautologies by checking if the ground-completed negative literals imply at least one of the positive literals, as suggested in [NN93].

- *Deletion of duplicate literals*:

$$\text{(DD)} \quad \frac{s \simeq t \vee s \simeq t \vee R}{s \simeq t \vee R}$$

- *Deletion of resolved literals*:

$$\text{(DR)} \quad \frac{s \not\simeq s \vee R}{R}$$

- *Destructive equality resolution*:

$$\text{(DE)} \quad \frac{x \not\simeq y \vee R}{\sigma(R)} \qquad \text{if } x, y \in V, \sigma = mgu(x, y)$$

We write $\mathbf{SP}(N)$ to denote the set of all clauses that can be generated with one generating inference from I on a set of clauses N, $\mathcal{D}_{SP}$ to denote the set of all **SP**-derivations, and $\mathcal{D}_{\overline{SP}}$ to denote the set of all finite **SP**-derivations.

◀

As **SP** only removes clauses that are *composite* with respect to the remaining set of clauses, the calculus is complete. For the case of unit clauses, it degenerates into *unfailing* completion [BDP89] as implemented in DISCOUNT. E can also simulate the positive unit strategy for Horn clauses described in [Der91] using appropriate selection functions.

Contrary to e.g. SPASS, E does not implement special rules for non-equational literals or sort theories, as we expect this part to be taken care of by SETHEO in a later combined system. Instead, non-equation literals are encoded as equations and dealt with accordingly.

3.2 Proof Procedure

The basic proof procedure of E is quite straightforward. The set of all clauses is split into two sets, a set `P` of *processed* clauses and a set `U` of *unprocessed* clauses. Initially, all input clauses are in in `U`, and `P` is empty. The algorithm selects a new clause from `U`, simplifies it w.r.t. to `P`, then uses it to simplify the clauses in `P` in turn. It then performs equality factoring, equality resolution and superposition between the selected clause and the set of processed clauses. The generated clauses are added to the set of unprocessed clauses. The process stops when the empty clause is derived or no further inferences are possible. Fig. 1 shows a (slightly simplified) pseudocode sketch of the procedure.

[6] This rule can only be implemented approximately, as the problem of recognizing tautologies is only semi-decidable in equational logic. The latest versions of E

```
# Input:  Axioms in U, P is empty
while U ≠ ∅ begin
    c := select(U)
    U := U \ c
    # Apply (RN), (RP), (SR), (DR1), (DR2)
    simplify(c,P)
    # Apply (SS1), (SS2), (TD1), (TD2)
    if c is trivial or subsumed by P then
        delete(c)
    else if c is the empty clause then
        # Success:  Proof found
        stop
    else
        T := ∅ # Temporary clause set
        foreach p ∈ P do
            if c simplifies a term in a maximal literal of p
            such that the set of maximal terms or
            the set of maximal literals of p potentially changes
            then
                P := P \ p
                T := T ∪ p
            done
            simplify(p, (P \ p) ∪ {c})
        end
        T := T ∪ e-resolvents(c) # (ER)
        T := T ∪ e-factors(c) # (EF)
        T := T ∪ paramodulants(c,P) # (SN), (SP)
        foreach p ∈ T do
            U := U ∪ simplify(p, P)
        end
    fi
end
# Failure:  Initial U is satisfiable, P describes model
```

Figure 1: Main proof procedure of E

The proof search is controlled by three major parameters: The term ordering (described in section 4.2), the literal selection function, and the order in which the `select` operation selects the next clause to process.

E implements two different classes of term orderings, lexicographic term orderings and Knuth-Bendix orderings. A given ordering is determined by instantiating one of the classes with a variety of parameters (described in section 4.2).

Literal selection currently is done according to one of more than 50 predefined functions. Section 4.3 describes this feature.

Clause selection is determined by a heuristic evaluation function, which conceptually sets up a set of priority queues and a weighted round robin scheme that determines from which queue the next clause is to be picked. The order within each queue is determined by a priority function (which partitions the set of unprocessed clauses into one or more subsets) and a heuristic evaluation function, which assigns a numerical rating to each clause. Section 4.1 describes the user interface to this mechanism.

4 Usage

4.1 Search Control Heuristics

Search control heuristics define the order in which the prover considers newly generated clauses. A heuristic is defined by a set of *clause evaluation functions* and a selection scheme wich defines how many clauses are selected according to each evaluation function. A clause evalution function consists of a *priority function* and an instance of a generic *weight function*.

4.1.1 Priority functions

Priority functions define a partition on the set of clauses. A single clause evaluation consists of a priority (which is the first selection criteria) and an evalution. Priorities are usually *not* suitable to encode heuristical control knowledge, but rather are used to express certain elements of a search strategy, or to restrict the effect of heuristic evaluation functions to certain classes of clauses.

Syntactically, the currently available priority functions are described by the following rule:

```
<prio-fun> ::= PreferGroundGoals ||
               PreferUnitGroundGoals ||
               PreferGround ||
               PreferNonGround ||
               PreferProcessed ||
               PreferNew ||
               PreferGoals ||
               PreferNonGoals ||
               PreferUnits ||
               PreferNonUnits ||
               PreferHorn ||
               PreferNonHorn ||
               ConstPrio ||
               ByLiteralNumber ||
               ByDerivationDepth ||
               ByDerivationSize ||
               ByNegLitDist ||
               ByGoalDifficulty ||
```

```
SimulateSOS||
PreferHorn||
PreferNonHorn||
PreferUnitAndNonEq||
DeferNonUnitMaxEq||
ByCreationDate
```

The priority functions are interpreted as follows:

`PreferGroundGoals`: Always prefer ground goals (all negative clauses without variables), do not differentiate between all other clauses.

`PreferUnitGroundGoals`: Prefer unit ground goals.

`PreferGround`: Prefer clauses without variables.

`PreferNonGround`: Prefer clauses with variables.

`PreferProcessed`: Prefer clauses that have already been processed once and have been eleminated from the set of processed clauses due to interreduction (forward contraction).

`PreferNew`: Prefer new clauses, i.e. clauses that are processed for the first time.

`PreferGoals`: Prefer goals (all negative clauses).

`PreferNonGoals`: Prefer non goals, i.e. facts with at least one positive literal.

`PreferUnits`: Prefer unit clauses (clauses with one literal).

`PreferNonUnits`: Prefer non-unit clauses.

`PreferHorn`: Prefer Horn clauses (clauses with no more than one positive literals).

`PreferNonHorn`: Prefer non-Horn clauses.

`ConstPrio`: Assign the same priority to all clauses.

`ByLiteralNumber`: Give a priority according to the number of literals, i.e. always prefer a clause with fewer literals to one with more literals.

`ByDerivationDepth`: Prefer clauses which have a short derivation depth, i.e. give a priority based on the length of the longest path from the clause to an axiom in the derivation tree. Counts generating inferences only.

`ByDerivationSize`: Prefer clauses which have been derived with a small number of (generating) inferences.

`ByNegLitDist`: Prefer goals to non-goals. Among goals, prefer goals with fewer literals and goals with ground literals (more exactly: the priority is increased by 1 for a ground literal and by 3 for a non-ground literal. Clauses with lower values are selected before clauses with higher values).

`ByGoalDifficulty:` Prefer goals to non-goals. Select goals based on a simple estimate of their difficulty: First unit ground goals, then unit goals, then ground goals, then other goals.

`SimulateSOS:` Use the priority system to simulate Set-Of-Support. This prefers all initial clauses and all Set-Of-Support clauses. Some non-SOS-clauses will be generated, but not selected for processing. This is neither well tested nor a particularly good fit with E's calculus, but can be used as one among many heuristics. If you try a pure SOS strategy, you also should set `--restrict-literal-comparisons` and run the prover without literal selection enabled.

`PreferHorn:` Prefer Horn clauses (note: includes units).

`PreferNonHorn:` Prefer non-Horn clauses.

`PreferUnitAndNonEq:` Prefer all unit clauses and all clauses without equational literal. This was an attempt to model some restricted calculi used e.g. in Gandalf [Tam97], but did not quite work out.

`DeferNonUnitMaxEq:` Prefer everything except for non-unit clauses with a maximal equational literal ("Don't paramodulate if its to expensive"). See above, same result.

`ByCreationDate:` Return the creation date of the clause as priority. This imposes a FIFO equivalence class on clauses. Clauses generated from the same given clause are grouped together (and can be ordered with any evaluation function among each other).

Please note that careless use of certain priority functions can make the prover incomplete for the general case.

4.1.2 Generic Weight Functions

Generic weight functions are templates for functions taking a clause and returning a weight (i.e. an estimate of the usefulness) for it, where a lower weight means that the corresponding clause should be processed before a clause with a higher weight. A generic weight function is combined with a priority function and instanciated with a set of parameters to yield a clause evaluation function.

You can specify an instantiated generic weight function as described in this rule[7]:

```
<weight-fun> ::= Clauseweight '(' <prio-fun> ', <int>, <int>,
                                  <float> ')'                     ||
                 Refinedweight '(' <prio-fun> ', <int>, <int>,
                                   <float>, <float>, <float> ')' ||
                 Orientweight '(' <prio-fun>, <int>, <int>,
```

[7] Note that there now are many additional generic weight functions not yet documented.

```
                  <float>, <float>, <float> ')'  ||
Simweight '(' <prio-fun>, <float>, <float>,
                  <float>, <float> ')'            ||
FIFOWeight '(' <prio-fun> ')'                     ||
LIFOWeight '(' <prio-fun> ')'
```

`Clauseweight(prio, fweight, vweight, pos_mult)`: This is the basic symbol counting heuristic. Variables are counted with weight `fweight`, function symbols with weight `vweight`. The weight of positive literals is multiplied by `pos_mult` before being added into the final weight.

`Refinedweight(prio, fweight, vweight, term_pen, lit_pen, pos_mult)`: This weight function is very similar to the first one. It differs only in that it takes the effect of the term ordering into account. In particular, the weight of a term that is maximal in its literal is multiplied by `term_pen`, and the weight of maximal literals is mutiplied by `lit_pen`.

`Orientweight(prio, fweight, vweight, term_pen, lit_pen, pos_mult)`: This weight function is a slight variation of `Refinedweight()`. In this case, the weight of *both* terms of an unorientable literal is multiplied by a penalty `term_pen`.

`Simweight(prio, equal_weight, vv_clash, vt_clash, tt_clash)`: This weight function is intended to return a low weight for literals in which the two terms ar very similar. It does not currently work very well even for unit clauses – RTFS (in <che_simweight.c>) to find out more.

`FIFOWeight(prio)`: This weight function assigns weights that increase in a stricly monotonic manner, i.e. it realises a *first-in/first-out* strategy if used all by itself. This is the most obviously fair strategy.

`LIFOWeight(prio)`: This weight function assigns weights that decrease in a stricly monotonic manner, i.e. it realises a *last-in/first-out* strategy if used all by itself (which, of course, would be unfair and result in an extremely incomplete prover).

4.1.3 Clause Evaluation Functions

A clause evaluation function is constructed by instantiating a generic weight function. It can either be specified directly, or specified and given a name for later reference at once:

```
<eval-fun>          ::= <ident>         ||
                        <weight-fun>    ||
                        <eval-fun-def>
<eval-fun-def>      ::= <ident> = <weight-fun>
<eval-fun-def-list> ::= <eval-fun-def>*
```

Of course a single identifier is only a valid evaluation function if it has been previously defined in a `<eval-fun-def>`. It is possible to define the value of an identifier more than once, in which case later definitions take precedence to former ones.

Clause evaluation functions can be be defined on the command line with the `-D` (--define-weight-function) option, followed by a `<eval-fun-def-list>`.

Example:

```
eprover -D"ex1=Clauseweight(ConstPrio,2,1,1) \
           ex2=FIFOWeight(PreferGoals)" ...
```

sets up the prover to know about two evaluation function `ex1` and `ex2` (wich supposedly will be used later on the command line to define one or more heuristics). The double quotes are necessary because the brackets and the commata are special characters for most shells

There are a variety of clause evaluation functions predefined in the variable `DefaultWeightFunctions`, which can be found in `che_proofcontrol.c`.

4.1.4 Heuristics

A heuristic defines how many selections are to be made according to one of several clause evaluation functions. Syntactically,

```
<heu-element>   ::= <int> '*' <eval-fun>
<heuristic>     ::= '(' <heu-element> (,<heu-element>)* ')' ||
                    <ident>
<heuristic-def> ::= <ident> = <heuristic> ||
                    <heuristic>
```

As above, a single identifier is only a valid heuristic if it has been defined in `<heuristic-def>` previously. A `<heuristic-def>` which degenerates to a simple heuristic defines a heuristic with name `Default` (which the prover will automatically choose if no other heuristic is selected with the `-x` (`--expert-heuristic`).

Example: To continue the above example,

```
eprover -D"ex1=Clauseweight(ConstPrio,2,1,1) \
           ex2=FIFOWeight(PreferGoals)"
        -H"new=(3*ex1,1*ex2)" \
        -x new LUSK3.lop
```

will run the prover on a problem file named LUSK3.lop with a heuristic that chooses 3 out of every 4 clauses according to a simple symbol counting heuristic and the last clause first among goals and then among other clauses, selecting by order of creation in each of these two classes.

4.2 Term Orderings

...exist and are important. Use the default or `-tAuto` until either better documentation turns up or you can pick up the necessary information some other way.

4.3 Literal Selection Strategies

The superposition calculus allows the *selection* of arbitrary negative literals in a clause and only requires generating inferences to be performed on these literals. E supports this feature and implements it via manipulations of the literal ordering. Additionally, E implements strategies that allow inferences into maximal positive literals and selected negative literals. A selection strategy is selected with the option `--literal-selection-strategy`. Currently, at least the following strategies are implemented:

NoSelection: Perform ordinary superposition without selection.

NoGeneration: Do not perform any generating inferences. This strategy is not complete, but applying it to a formula generates a normal form that does not contain any tautologies or redundant clauses.

SelectNegativeLiterals: Select all negative literals. For Horn clauses, this implements the maximal literal positive unit strateg [Der91] previously realized separately in E.

SelectPureVarNegLiterals: Select the first negative literal of the form $X \simeq Y$.

SelectLargestNegLit: Select the largest negative literal (by symbol counting, function symbols count as 2, variables as 1).

SelectSmallestNegLit: As above, but select the smallest literal.

SelectDiffNegLit: Select the negative literal in which both terms have the largest size difference.

SelectGroundNegLit: Select the first negative ground literal for which the size difference between both terms is maximal.

SelectOptimalLit: If there is a ground negative literal, select as in the case of **SelectGroundNegLit**, otherwise as in **SelectDiffNegLit**.

Each of the strategies that do actually select negative literals has a corresponding counterpart starting with **P** that additionally allows paramodulation into maximal positive literals[8].

[8] Except for **SelectOptimalLit**, where the resulting strategy, **PSelectOptimalLit** will allow paramodulation into positive literals only if no ground literal has been selected.

Example: Some problems become a lot simpler with the correct strategy. Try e.g.

```
eprover --literal-selection-strategy=NoSelection \
          GRP001-1+rm_eq_rstfp.lop
eprover --literal-selection-strategy=SelectLargestNegLit \
          GRP001-1+rm_eq_rstfp.lop
```

You will find the file `GRP001-1+rm_eq_rstfp.lop` in the `E/PROVER` directory.

As we aim at replacing the vast number of individual literal selection functions with a more abstract mechanism, we refrain from describing all of the currently implemented functions in detail. If you need information about the set of implemented functions, run `eprover -W none`. The indiviual functions are implemented and somewhat described in `E/HEURISTICS/che_litselection.h`.

4.4 Other Options

5 Input Language

E natively uses E-LOP, a dialect of the LOP languange designed for SETHEO.At the moment, your best bet is to retrieve the LOP description from the E web site [Sch99] and/or check out the examples available from it. LOP is very close to Prolog, and E can usually read many fully declarative Prolog files if they do not use arithmetic or rely on predefined symbols. Plain SETHEO files usually also work very well. There are a couple of minor differences, however:

- `equal()` is an interpreted symbol for E. It normally does not carry any meaning for SETHEO (unless equality axioms are added).
- SETHEO allows the same identifier to be used as a constant, a non-constant function symbol and a predicate symbol. E encodes all of these as ordinary function symbols, and hence will complain if a symbol is used inconsistently.
- E allows the use of both `=` and `=>` as infix symbols for equality. `a=b` is equivalent to `equal(a,b)` for E.
- E does not support constraints or SETHEO build-in symbols. This should not usually affect pure theorem proving tasks.
- E normally treats procedural clauses exactly as it treats declarative clauses. Query clauses (clauses with an empty head and starting with `?-`, e.g. `?-~p(X), q(X).` can optionally be used to define the a set of *goal clauses* (by default, all negative clauses are considered to be goals). At the moment, this information is only used for the initial set of support (with

--sos-uses-input-types). Note that you can still specify arbitrary clauses as query clauses, since LOP supports negated literals.

As an alternative, E also supports TPTP syntax [SS97] (if given the option --tptp-in or --tptp-format) without includes and as far as it can be divined from the TPTP manual. In TPTP format, clauses with TPTP type conjecture are considered goal clauses for the --sos-uses-input-types option.

6 Output. . . or how to interpret what you see

E has several different output levels, controlled by the option -l or --output-level. Level 0 prints nearly no output except for the result. Level 1 is intended to give humans a somewhat readable impression of what is going on inside the inference engine. Levels 3 to 6 output increasingly more information about the inside processes in PCL2 format. At level 4 and above, a (large) superset of the proof inferences is printed. You can use the epclextract utility in E/PROVER/ to extract a simple proof object.

In Level 0 and 1, everything E prints is either a clause that is implied by the original axioms, or a comment (or, very often, both).

6.1 The Bare Essentials

In silent mode (--output-level=0, -s or --silent), E will not print any output during saturation. It will print a one-line comment documenting the state of the proof search after termination. The following possibilities exist:

- The prover found a proof. This is denoted by the output string

  ```
  # Proof found!
  ```

- The problem does not have a proof, i.e. the specification is satisfiable (and E can detect this):

  ```
  # No proof found!
  ```

  Ensuring the completeness of a prover is much harder than ensuring correctness. Moreover, proofs can easily be checked by analysing the output of the prover, while such a check for the absence of proofs is rarely possible. I do believe that the current version of E is both correct and complete[9] but my belief in the former is stronger than my beliefe in the later. . . ...

- A (hard) resource limit was hit. For memory this can be either due to a per process limit (set with limit or the prover option --memory-limit), or due to running out of virtual memory. For cpu time, this case is triggered

[9] Unless the prover runs out of memory (see below), the user selects an unfair strategy (in which case the prover may never terminate), or some strange and unexpected things happen.

if the per process cpu time limit is reached and signalled to the prover via a `SIGXCPU` signal. This limit can be set with `limit` or, more reliable, with the option `--cpu-limit`. The output string is one of the following two, depending on the exact reason for termination:

```
# Failure: Resource limit exceeded (memory)
# Failure: Resource limit exceeded (time)
```

- A user-defined limit was reached during saturation, and the saturation process was stopped gracefully. Limits include number of processed clauses, number of total clauses, and cpu time (as set with `--soft-cpu-limit`. The output string is

  ```
  # Faiure: User resource limit exceeded!
  ```

  ... and the user is expected to know which limit he selected.

- Normally, E is complete. However, if the option `--delete-bad-limit` is given or if automatic mode in connection with a memory limit is used, E will periodically delete clauses it deems unlikely to be processed to avoid running out of memory. In this case, completeness cannot be ensured any more. This effect manifests itself extremely rarely. If it does, E will print the following string:

  ```
  # Failure: Out of unprocessed clauses!
  ```

  This is roughtly equivalent to Otter's `SOS empty` message.

- Finally, it is possible to chose restricted calculi when starting E. This is useful if E is used as a normalization tool or as a preprocessor or lemma generator. In this case, E will print a corresponding message:

  ```
  # Clause set closed under restricted calculus!
  ```

6.2 Impressing your Friends

If you run E without selection an output level (or by setting it explicitly to 1), E will print each non-tautological, non-subsumed clause it processes as a comment. It will also print a hash ('`#`') for each clause it tries to process but can prove to be superfluous.

This mode gives some indication of progress, and as the output is fairly restricted, does not slow the prover down too much.

For any output level greater than 0, E will also print statistical information about the proof search and final clause sets. The data should be fairly self-explaining.

6.3 Detailed Reporting

At output levels greater that 1, E prints certain inferences in PCL2 format[10]. At level 2, it only prints generating inferences. At level 4, it prints all generating and modifying inferences, and at level 6 it also prints PCL steps giving a lot of insight into the interal operation of the inference engine. This protocol is fairly readable and, from level 4 on can be used to check the proof with the utility `checkproof` provided with the distribution.

6.4 Requesting Specific Results

There are two additional kinds of information E can provide beyond the normal output during proof search: Statistical information and final clause sets (with additional information).

First, E can give you some technical information about the conditions it runs under.

The option `--print-pid` will make E printing its process id as a comment, in the format `# Pid: XXX`, where `XXX` is an integer number. This is useful if you want to send signals to the prover (in particular, if you want to terminate the prover) to control it from the outside.

The option `-R` (`--resources-info`) will make E print a summary of used system resources after graceful termination:

```
# User time                : 0.010 s
# System time              : 0.020 s
# Total time               : 0.030 s
# Maximum resident set size: 0 pages
```

Most operating systems do not provide a valid value for the resident set size and other memory-related resources, so you should probably not depend on the last value to carry any meaningful information. The time information is required by most standards and should be useful for all tested operating systems.

E can be used not only as a prover, but as a normalizer for formulae or as a lemma generator. In this cases, you will not only want to know if E found a proof, but also need some or all of the derived clauses, possibly with statistical information for filtering. This is supported with the `--print-saturated` and `--print-sat-info` options for E.

The option `--print-saturated` takes as it's argument a string of letters, each of which represents a part of the total set of clauses E knows about. The following table contains the meaning of the individual letters:

[10] PCL2 is a proof output protocol language currently being designed by me as a successor to PCL [DS94a, DS94b, DS96].

`e`	Processed positive unit clauses (*Equations*).
`i`	Processed negative unit clauses (*Inequations*).
`g`	Processed non-unit clauses (except for the empty clause, which, if present, is printed separatly). The above three sets are interreduced and all selected inferences between them have been computed.
`E`	Unprocessed positive unit clauses.
`I`	Unprocessed negative unit clauses.
`G`	Unprocessed non-unit clause (this set may contain the empty clause in very rare cases).
`a`	Print equality axioms (if equality is present in the problem). This letter prints axioms for refelxivity, symmetry, and transitivity, and a set of substitutivity axioms, one for each argument position of every function symbol and predicate symbol.
`A`	As `a`, but print a single substitutivity axiom covering all positions for each symbol.

The short form, `-S`, is equivalent to `--print-saturated=eigEIG`. If the option `--print-sat-info` is set, then each of the clauses is followed by a comment of the form `# info(id, pd, pl, sc, cd, nl, no, nv)`. The followig table explains the meaning of these values:

`id`	Clause ident (probably only useful internally)
`pd`	Depth of the derivation graph for this clause
`pl`	Number of nodes in the derivation grap
`sc`	Symbol count (function symbols and variables)
`cd`	Depth of the deepest term in the clause
`nl`	Number of literals in the clause
`no`	Number of variable occurences
`nv`	Number of different variables

A License

The standard distribution of E is free software. You can use, modify and copy it under the terms of the GNU General Public License. You may also have bought a commercial version of E from Safelogic A.B. in Gothenburg, Sweden. In this case, you are bound by whatever license you agreed to. If you are in doubt about which version of E you have, run `eprover -V` or `eprover -h`.

```
                    GNU GENERAL PUBLIC LICENSE
                       Version 2, June 1991

 Copyright (C) 1989, 1991 Free Software Foundation, Inc.
     59 Temple Place, Suite 330, Boston, MA  02111-1307  USA
 Everyone is permitted to copy and distribute verbatim copies
 of this license document, but changing it is not allowed.

                            Preamble

  The licenses for most software are designed to take away your
freedom to share and change it.  By contrast, the GNU General Public
License is intended to guarantee your freedom to share and change free
software--to make sure the software is free for all its users.  This
General Public License applies to most of the Free Software
Foundation's software and to any other program whose authors commit to
using it.  (Some other Free Software Foundation software is covered by
the GNU Library General Public License instead.)  You can apply it to
your programs, too.

  When we speak of free software, we are referring to freedom, not
price.  Our General Public Licenses are designed to make sure that you
have the freedom to distribute copies of free software (and charge for
this service if you wish), that you receive source code or can get it
if you want it, that you can change the software or use pieces of it
in new free programs; and that you know you can do these things.

  To protect your rights, we need to make restrictions that forbid
anyone to deny you these rights or to ask you to surrender the rights.
These restrictions translate to certain responsibilities for you if you
distribute copies of the software, or if you modify it.

  For example, if you distribute copies of such a program, whether
gratis or for a fee, you must give the recipients all the rights that
you have.  You must make sure that they, too, receive or can get the
source code.  And you must show them these terms so they know their
rights.

  We protect your rights with two steps: (1) copyright the software, and
(2) offer you this license which gives you legal permission to copy,
distribute and/or modify the software.
```

Also, for each author's protection and ours, we want to make certain that everyone understands that there is no warranty for this free software. If the software is modified by someone else and passed on, we want its recipients to know that what they have is not the original, so that any problems introduced by others will not reflect on the original authors' reputations.

Finally, any free program is threatened constantly by software patents. We wish to avoid the danger that redistributors of a free program will individually obtain patent licenses, in effect making the program proprietary. To prevent this, we have made it clear that any patent must be licensed for everyone's free use or not licensed at all.

The precise terms and conditions for copying, distribution and modification follow.

GNU GENERAL PUBLIC LICENSE
TERMS AND CONDITIONS FOR COPYING, DISTRIBUTION AND MODIFICATION

0. This License applies to any program or other work which contains a notice placed by the copyright holder saying it may be distributed under the terms of this General Public License. The "Program", below, refers to any such program or work, and a "work based on the Program" means either the Program or any derivative work under copyright law: that is to say, a work containing the Program or a portion of it, either verbatim or with modifications and/or translated into another language. (Hereinafter, translation is included without limitation in the term "modification".) Each licensee is addressed as "you".

Activities other than copying, distribution and modification are not covered by this License; they are outside its scope. The act of running the Program is not restricted, and the output from the Program is covered only if its contents constitute a work based on the Program (independent of having been made by running the Program). Whether that is true depends on what the Program does.

1. You may copy and distribute verbatim copies of the Program's source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice and disclaimer of warranty; keep intact all the notices that refer to this License and to the absence of any warranty; and give any other recipients of the Program a copy of this License along with the Program.

You may charge a fee for the physical act of transferring a copy, and you may at your option offer warranty protection in exchange for a fee.

2. You may modify your copy or copies of the Program or any portion of it, thus forming a work based on the Program, and copy and

distribute such modifications or work under the terms of Section 1 above, provided that you also meet all of these conditions:

a) You must cause the modified files to carry prominent notices stating that you changed the files and the date of any change.

b) You must cause any work that you distribute or publish, that in whole or in part contains or is derived from the Program or any part thereof, to be licensed as a whole at no charge to all third parties under the terms of this License.

c) If the modified program normally reads commands interactively when run, you must cause it, when started running for such interactive use in the most ordinary way, to print or display an announcement including an appropriate copyright notice and a notice that there is no warranty (or else, saying that you provide a warranty) and that users may redistribute the program under these conditions, and telling the user how to view a copy of this License. (Exception: if the Program itself is interactive but does not normally print such an announcement, your work based on the Program is not required to print an announcement.)

These requirements apply to the modified work as a whole. If identifiable sections of that work are not derived from the Program, and can be reasonably considered independent and separate works in themselves, then this License, and its terms, do not apply to those sections when you distribute them as separate works. But when you distribute the same sections as part of a whole which is a work based on the Program, the distribution of the whole must be on the terms of this License, whose permissions for other licensees extend to the entire whole, and thus to each and every part regardless of who wrote it.

Thus, it is not the intent of this section to claim rights or contest your rights to work written entirely by you; rather, the intent is to exercise the right to control the distribution of derivative or collective works based on the Program.

In addition, mere aggregation of another work not based on the Program with the Program (or with a work based on the Program) on a volume of a storage or distribution medium does not bring the other work under the scope of this License.

3. You may copy and distribute the Program (or a work based on it, under Section 2) in object code or executable form under the terms of Sections 1 and 2 above provided that you also do one of the following:

a) Accompany it with the complete corresponding machine-readable source code, which must be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,

b) Accompany it with a written offer, valid for at least three years, to give any third party, for a charge no more than your cost of physically performing source distribution, a complete machine-readable copy of the corresponding source code, to be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,

c) Accompany it with the information you received as to the offer to distribute corresponding source code. (This alternative is allowed only for noncommercial distribution and only if you received the program in object code or executable form with such an offer, in accord with Subsection b above.)

The source code for a work means the preferred form of the work for making modifications to it. For an executable work, complete source code means all the source code for all modules it contains, plus any associated interface definition files, plus the scripts used to control compilation and installation of the executable. However, as a special exception, the source code distributed need not include anything that is normally distributed (in either source or binary form) with the major components (compiler, kernel, and so on) of the operating system on which the executable runs, unless that component itself accompanies the executable.

If distribution of executable or object code is made by offering access to copy from a designated place, then offering equivalent access to copy the source code from the same place counts as distribution of the source code, even though third parties are not compelled to copy the source along with the object code.

4. You may not copy, modify, sublicense, or distribute the Program except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense or distribute the Program is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

5. You are not required to accept this License, since you have not signed it. However, nothing else grants you permission to modify or distribute the Program or its derivative works. These actions are prohibited by law if you do not accept this License. Therefore, by modifying or distributing the Program (or any work based on the Program), you indicate your acceptance of this License to do so, and all its terms and conditions for copying, distributing or modifying the Program or works based on it.

6. Each time you redistribute the Program (or any work based on the Program), the recipient automatically receives a license from the original licensor to copy, distribute or modify the Program subject to

these terms and conditions. You may not impose any further restrictions on the recipients' exercise of the rights granted herein. You are not responsible for enforcing compliance by third parties to this License.

7. If, as a consequence of a court judgment or allegation of patent infringement or for any other reason (not limited to patent issues), conditions are imposed on you (whether by court order, agreement or otherwise) that contradict the conditions of this License, they do not excuse you from the conditions of this License. If you cannot distribute so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not distribute the Program at all. For example, if a patent license would not permit royalty-free redistribution of the Program by all those who receive copies directly or indirectly through you, then the only way you could satisfy both it and this License would be to refrain entirely from distribution of the Program.

If any portion of this section is held invalid or unenforceable under any particular circumstance, the balance of the section is intended to apply and the section as a whole is intended to apply in other circumstances.

It is not the purpose of this section to induce you to infringe any patents or other property right claims or to contest validity of any such claims; this section has the sole purpose of protecting the integrity of the free software distribution system, which is implemented by public license practices. Many people have made generous contributions to the wide range of software distributed through that system in reliance on consistent application of that system; it is up to the author/donor to decide if he or she is willing to distribute software through any other system and a licensee cannot impose that choice.

This section is intended to make thoroughly clear what is believed to be a consequence of the rest of this License.

8. If the distribution and/or use of the Program is restricted in certain countries either by patents or by copyrighted interfaces, the original copyright holder who places the Program under this License may add an explicit geographical distribution limitation excluding those countries, so that distribution is permitted only in or among countries not thus excluded. In such case, this License incorporates the limitation as if written in the body of this License.

9. The Free Software Foundation may publish revised and/or new versions of the General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

Each version is given a distinguishing version number. If the Program specifies a version number of this License which applies to it and "any later version", you have the option of following the terms and conditions either of that version or of any later version published by the Free Software Foundation. If the Program does not specify a version number of this License, you may choose any version ever published by the Free Software Foundation.

10. If you wish to incorporate parts of the Program into other free programs whose distribution conditions are different, write to the author to ask for permission. For software which is copyrighted by the Free Software Foundation, write to the Free Software Foundation; we sometimes make exceptions for this. Our decision will be guided by the two goals of preserving the free status of all derivatives of our free software and of promoting the sharing and reuse of software generally.

NO WARRANTY

11. BECAUSE THE PROGRAM IS LICENSED FREE OF CHARGE, THERE IS NO WARRANTY FOR THE PROGRAM, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE PROGRAM "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE PROGRAM IS WITH YOU. SHOULD THE PROGRAM PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.

12. IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MAY MODIFY AND/OR REDISTRIBUTE THE PROGRAM AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE PROGRAM (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A FAILURE OF THE PROGRAM TO OPERATE WITH ANY OTHER PROGRAMS), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

END OF TERMS AND CONDITIONS

How to Apply These Terms to Your New Programs

If you develop a new program, and you want it to be of the greatest possible use to the public, the best way to achieve this is to make it free software which everyone can redistribute and change under these terms.

To do so, attach the following notices to the program. It is safest to attach them to the start of each source file to most effectively

```
convey the exclusion of warranty; and each file should have at least
the "copyright" line and a pointer to where the full notice is found.

    <one line to give the program's name and a brief idea of what it does.>
    Copyright (C) 19yy  <name of author>

    This program is free software; you can redistribute it and/or modify
    it under the terms of the GNU General Public License as published by
    the Free Software Foundation; either version 2 of the License, or
    (at your option) any later version.

    This program is distributed in the hope that it will be useful,
    but WITHOUT ANY WARRANTY; without even the implied warranty of
    MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
    GNU General Public License for more details.

    You should have received a copy of the GNU General Public License
    along with this program; if not, write to the Free Software
    Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA  02111-1307  USA

Also add information on how to contact you by electronic and paper mail.

If the program is interactive, make it output a short notice like this
when it starts in an interactive mode:

    Gnomovision version 69, Copyright (C) 19yy name of author
    Gnomovision comes with ABSOLUTELY NO WARRANTY; for details type `show w'.
    This is free software, and you are welcome to redistribute it
    under certain conditions; type `show c' for details.

The hypothetical commands `show w' and `show c' should show the appropriate
parts of the General Public License.  Of course, the commands you use may
be called something other than `show w' and `show c'; they could even be
mouse-clicks or menu items--whatever suits your program.

You should also get your employer (if you work as a programmer) or your
school, if any, to sign a "copyright disclaimer" for the program, if
necessary.  Here is a sample; alter the names:

  Yoyodyne, Inc., hereby disclaims all copyright interest in the program
  `Gnomovision' (which makes passes at compilers) written by James Hacker.

  <signature of Ty Coon>, 1 April 1989
  Ty Coon, President of Vice

This General Public License does not permit incorporating your program into
proprietary programs.  If your program is a subroutine library, you may
consider it more useful to permit linking proprietary applications with the
library.  If this is what you want to do, use the GNU Library General
Public License instead of this License.
```

References

[Bac98] L. Bachmair. Personal communication at CADE-15, Lindau. Unpublished, 1998.

[BDP89] L. Bachmair, N. Dershowitz, and D.A. Plaisted. Completion Without Failure. In H. Ait-Kaci and M. Nivat, editors, *Resolution of Equations in Algebraic Structures*, volume 2, pages 1–30. Academic Press, 1989.

[BG94] L. Bachmair and H. Ganzinger. Rewrite-Based Equational Theorem Proving with Selection and Simplification. *Journal of Logic and Computation*, 3(4):217–247, 1994.

[CL73] C. Chang and R.C. Lee. *Symbolic Logic and Mechanical Theorem Proving*. Computer Science and Applied Mathematics. Academic Press, 1973.

[Der91] N. Dershowitz. Ordering-Based Strategies for Horn Clauses. In J. Mylopoulos, editor, *Proc. of the 12th IJCAI, Sydney*, volume 1, pages 118–124. Morgan Kaufmann, 1991.

[DKS97] J. Denzinger, M. Kronenburg, and S. Schulz. DISCOUNT: A Distributed and Learning Equational Prover. *Journal of Automated Reasoning*, 18(2):189–198, 1997. Special Issue on the CADE 13 ATP System Competition.

[DS94a] J. Denzinger and S. Schulz. Analysis and Representation of Equational Proofs Generated by a Distributed Completion Based Proof System. Seki-Report SR-94-05, Universität Kaiserslautern, 1994.

[DS94b] J. Denzinger and S. Schulz. Recording, Analyzing and Presenting Distributed Deduction Processes. In H. Hong, editor, *Proc. 1st PASCO, Hagenberg/Linz*, volume 5 of *Lecture Notes Series in Computing*, pages 114–123, Singapore, 1994. World Scientific Publishing.

[DS96] J. Denzinger and S. Schulz. Recording and Analysing Knowledge-Based Distributed Deduction Processes. *Journal of Symbolic Computation*, 21(4/5):523–541, 1996.

[HBF96] T. Hillenbrand, A. Buch, and R. Fettig. On Gaining Efficiency in Completion-Based Theorem Proving. In H. Ganzinger, editor, *Proc. of the 7th RTA, New Brunswick*, volume 1103 of *LNCS*, pages 432–435. Springer, 1996.

[HJL99] T. Hillenbrand, A. Jaeger, and B. Löchner. System Abstract: Waldmeister – Improvements in Performance and Ease of Use. In H. Ganzinger, editor, *Proc. of the 16th CADE, Trento*, volume 1632 of *LNAI*, pages 232–236. Springer, 1999.

[McC94] W.W. McCune. Otter 3.0 Reference Manual and Guide. Technical Report ANL-94/6, Argonne National Laboratory, 1994.

[MIL+97] M. Moser, O. Ibens, R. Letz, J. Steinbach, C. Goller, J. Schumann, and K. Mayr. SETHEO and E-SETHEO – The CADE-13 Systems. *Journal of Automated Reasoning*, 18(2):237–246, 1997. Special Issue on the CADE 13 ATP System Competition.

[MW97] W.W. McCune and L. Wos. Otter: The CADE-13 Competition Incarnations. *Journal of Automated Reasoning*, 18(2):211–220, 1997. Special Issue on the CADE 13 ATP System Competition.

[NN93] P. Nivela and R. Nieuwenhuis. Saturation of First-Order (Constrained) Clauses with the Saturate System. In C. Kirchner, editor, *Proc. of the 5th RTA, Montreal*, volume 690 of *LNCS*, pages 436–440. Springer, 1993.

[RV01] A. Riazanov and A. Voronkov. Vampire 1.1 (System Description). In R. Goré, A. Leitsch, and T. Nipkow, editors, *Proc. of the 1st IJCAR, Siena*, volume 2083 of *LNAI*, pages 376–380. Springer, 2001.

[RV02] A. Riazanov and A. Voronkov. The Design and Implementation of VAMPIRE. *Journal of AI Communications*, 15(2/3):91–110, 2002.

[Sch99] S. Schulz. The E Web Site. `http://www4.informatik.tu-muenchen.de/\-$\sim$schulz/\-WORK/\-eprover.h%tml`, 1999.

[Sch01] S. Schulz. System Abstract: E 0.61. In R. Goré, A. Leitsch, and T. Nipkow, editors, *Proc. of the 1st IJCAR, Siena*, volume 2083 of *LNAI*, pages 370–375. Springer, 2001.

[Sch02] S. Schulz. E – A Brainiac Theorem Prover. *Journal of AI Communications*, 15(2/3):111–126, 2002.

[SS97] C.B. Suttner and G. Sutcliffe. The TPTP Problem Library (TPTP v2.1.0). Technical Report AR-97-01 (TUM), 97/04 (JCU), Institut für Informatik, Technische Universität München, Munich, Germany/Department of Computer Science, James Cook University, Townsville, Australia, 1997. Jointly published.

[Tam97] T. Tammet. Gandalf. *Journal of Automated Reasoning*, 18(2):199–204, 1997. Special Issue on the CADE 13 ATP System Competition.

[WAB+99] C. Weidenbach, B. Afshordel, U. Brahm, C. Cohrs, T. Engel, G. Jung, E. Keen, C. Theobalt, and D. Topic. System Abstract: SPASS Version 1.0.0. In H. Ganzinger, editor, *Proc. of the 16th CADE, Trento*, volume 1632 of *LNAI*, pages 378–382. Springer, 1999.

[Wei99] C. Weidenbach. Personal communication at CADE-16, Trento. Unpublished, 1999.

[WGR96] C. Weidenbach, B. Gaede, and G. Rock. SPASS & FLOTTER Version 0.42. In M.A. McRobbie and J.K. Slaney, editors, *Proc. of the 13th CADE, New Brunswick*, volume 1104 of *LNAI*, pages 141–145. Springer, 1996.