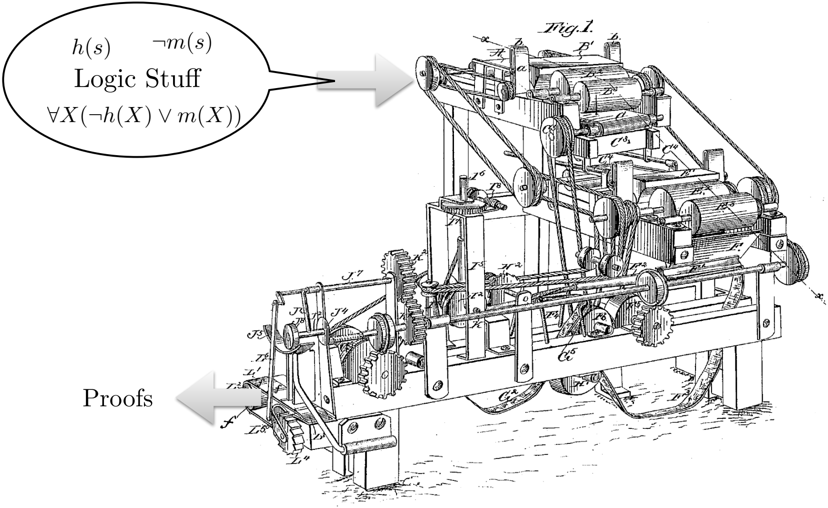
# Implementation of Saturating Theorem Provers

Stephan Schulz

schulz@eprover.org



$h(s)$ $\neg m(s)$

Logic Stuff

$\forall X(\neg h(X) \vee m(X))$

Proofs

*International Summer School on Satisfiability, Satisfiability Modulo Theories, and Automated Reasoning*

# Abstract

We will describe several aspects of the implementation of a saturating, superposition-based theorem prover for first-order logic. We discuss the basic architecture of a prover, and the organization of the actual proof search via the given-clause algorithm. Simplification and redundancy elimination are, in practice, critical, and we describe how these can be integrated into the basic proof procedure.

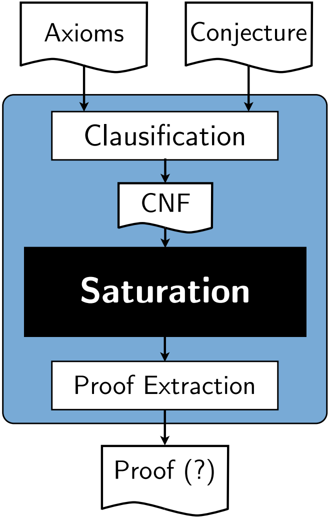
Another topic is that of terms and clauses, the most basic data objects in any prover, and how they can be implemented. We discuss bottlenecks of naive implementations, as well as the principle and some examples of indexing techniques to overcome these bottlenecks. The presentation concludes with a look at the influence of search heuristics.

# Contents

1. Introduction
2. Basic Data Types
3. Saturation Algorithm
4. Term and Clause Indexing
5. Search Control
6. Conclusion

# Introduction

# Black Box View



Axioms

Conjecture

Clausification

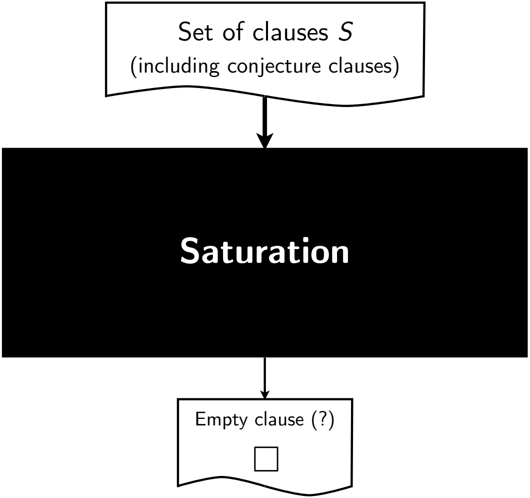
CNF

**Saturation**

Proof Extraction

Proof (?)

# Black Box View (zoom)



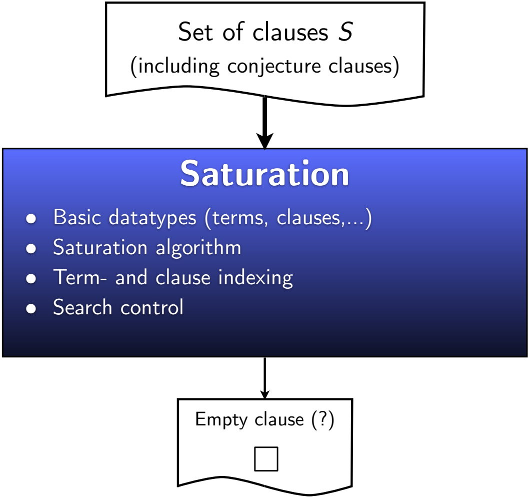
Set of clauses $S$
(including conjecture clauses)

**Saturation**

Empty clause (?)

$\square$

# Blue Box View



Set of clauses $S$

(including conjecture clauses)

**Saturation**

- Basic datatypes (terms, clauses,...)
- Saturation algorithm
- Term- and clause indexing
- Search control

Empty clause (?)

$\square$

# Proving by Saturation

- Goal: Show unsatisfiability of a set of clauses $S$
- Approach:
  - Systematically enrich $S$ with clauses derived via inferences from clauses in $S$ (Saturation)
  - Optionally: Remove/simplify redundant clauses
- Outcome:
  - Derivation of the empty clause $\Box$ (explicit witness of unsatisfiability)
  - Successful saturation (up to redundancy): $S$ is satisfiable
  - …or infinite sequence of derivations
- Properties:
  - Correctness: Only logical consequences are derived
  - Completeness: Every unsatisfiable $S$ will eventually lead to the derivation of $\Box$

(Equality Resolution)

$$\frac{u \not\simeq v \vee R}{\sigma(R)} \qquad \begin{array}{l} \text{if } \sigma \;=\; \mathrm{mgu}(u, v), \\ \ldots \end{array}$$

(Deletion of resolved literals)

$$\begin{array}{c} s \not\simeq s \vee R \\ \hline\hline R \end{array}$$

(Superposition into negative literals)

$$\frac{s \simeq t \vee S \quad u \not\simeq v \vee R}{\sigma(u[p \leftarrow t] \not\simeq v \vee S \vee R)} \qquad \begin{array}{l} \text{if } \sigma = mgu(u|_p, s), \\ \sigma(s) \not< \sigma(t), \; \ldots \end{array}$$

(Rewriting of negative literals)

$$\begin{array}{c} s \simeq t \quad u \not\simeq v \vee R \\ \hline\hline s \simeq t \quad u[p \leftarrow \sigma(t)] \not\simeq v \vee R \end{array} \qquad \begin{array}{l} \text{if } u|_p \;=\; \sigma(s) \text{ and} \\ \sigma(s) > \sigma(t) \end{array}$$

(Equality Resolution)

$$\frac{u \not\simeq v \vee R}{\sigma(R)} \quad \begin{array}{l} \text{if } \sigma \;=\; \mathrm{mgu}(u,v), \\ \ldots \end{array}$$

(Superposition into negative literals)

$$\frac{s \simeq t \vee S \quad u \not\simeq v \vee R}{\sigma(u[p \leftarrow t] \not\simeq v \vee S \vee R)} \quad \begin{array}{l} \text{if } \sigma = mgu(u|_p, s), \\ \sigma(s) \not< \sigma(t), \ \ldots \end{array}$$

(Deletion of resolved literals)

$$\begin{array}{c} s \not\simeq s \vee R \\ \hline\hline R \end{array}$$

(Rewriting of negative literals)

$$\begin{array}{c} s \simeq t \quad u \not\simeq v \vee R \\ \hline\hline s \simeq t \quad u[p \leftarrow \sigma(t)] \not\simeq v \vee R \end{array} \quad \begin{array}{l} \text{if } u|_p \;=\; \sigma(s) \text{ and} \\ \sigma(s) > \sigma(t) \end{array}$$

(Equality Resolution)

$$\frac{u \not\simeq v \vee R}{\sigma(R)} \quad \begin{array}{l} \text{if } \sigma \ = \ \mathrm{mgu}(u, v), \\ \ldots \end{array}$$

## Generating inference rules

- Necessary for completeness
- Increase size of proof state

(Superposition into negative literals)

$$\frac{s \simeq t \vee S \quad u \not\simeq v \vee R}{\sigma(u[p \leftarrow t] \not\simeq v \vee S \vee R)} \quad \begin{array}{l} \text{if } \sigma = mgu(u|_p, s), \\ \sigma(s) \not< \sigma(t), \ \ldots \end{array}$$

(Deletion of resolved literals)

$$\begin{array}{c} s \not\simeq s \vee R \\ \hline\hline R \end{array}$$

## Simplification rules

- Critical for performance
- Reduce size of proof state

(Rewriting of negative literals)

$$\begin{array}{c} s \simeq t \quad u \not\simeq v \vee R \\ \hline\hline s \simeq t \quad u[p \leftarrow \sigma(t)] \not\simeq v \vee R \end{array} \quad \begin{array}{l} \text{if } u|_p \ = \ \sigma(s) \ \text{ and} \\ \sigma(s) > \sigma(t) \end{array}$$

(Equality Resolution)

$$\frac{u \not\simeq v \vee R}{\sigma(R)} \quad \begin{array}{l}\text{if } \sigma \ = \ \mathrm{mgu}(u,v),\\ \ldots\end{array}$$

(Deletion of resolved literals)

$$\frac{s \not\simeq s \vee R}{\frac{}{R}}$$

(Superposition into negative literals)

$$\frac{s \simeq t \vee S \quad u \not\simeq v \vee R}{\sigma(u[p \leftarrow t] \not\simeq v \vee S \vee R)} \quad \begin{array}{l}\text{if } \sigma = mgu(u|_p, s),\\ \sigma(s) \not< \sigma(t),\ \ldots\end{array}$$

(Rewriting of negative literals)

$$\frac{s \simeq t \quad u \not\simeq v \vee R}{\frac{}{s \simeq t \quad u[p \leftarrow \sigma(t)] \not\simeq v \vee R}} \quad \begin{array}{l}\text{if } u|_p \ = \ \sigma(s) \text{ and}\\ \sigma(s) > \sigma(t)\end{array}$$

### (Equality Resolution)

$$\frac{u \not\simeq v \vee R}{\sigma(R)} \quad \begin{array}{l} \text{if } \sigma \ = \ \mathrm{mgu}(u, v), \\ \ldots \end{array}$$

### (Deletion of resolved literals)

$$\begin{array}{c} s \not\simeq s \vee R \\ \hline\hline R \end{array}$$

---

### (Superposition into negative literals)

$$\frac{s \simeq t \vee S \quad u \not\simeq v \vee R}{\sigma(u[p \leftarrow t] \not\simeq v \vee S \vee R)} \quad \begin{array}{l} \text{if } \sigma = mgu(u|_p, s), \\ \sigma(s) \not< \sigma(t), \ \ldots \end{array}$$

### (Rewriting of negative literals)

$$\begin{array}{c} s \simeq t \quad u \not\simeq v \vee R \\ \hline\hline s \simeq t \quad u[p \leftarrow \sigma(t)] \not\simeq v \vee R \end{array} \quad \begin{array}{l} \text{if } u|_p \ = \ \sigma(s) \ \text{and} \\ \sigma(s) > \sigma(t) \end{array}$$

(Equality Resolution)

$$\frac{u \not\simeq v \vee R}{\sigma(R)} \qquad \begin{array}{l} \text{if } \sigma \;=\; \mathrm{mgu}(u, v), \\ \ldots \end{array}$$

(Deletion of resolved literals)

$$\begin{array}{c} s \not\simeq s \vee R \\ \hline\hline R \end{array}$$

## Local (single premise)

- Easy to keep track of
- Cheap to implement

---

## Non-local (multiple premises)

- Harder to keep track of (pairs of clauses!)
- Expensive to implement (find partners)

(Superposition into negative literals)

$$\frac{s \simeq t \vee S \quad u \not\simeq v \vee R}{\sigma(u[p \leftarrow t] \not\simeq v \vee S \vee R)} \qquad \begin{array}{l} \text{if } \sigma = mgu(u|_p, s), \\ \sigma(s) \not< \sigma(t), \ \ldots \end{array}$$

(Rewriting of negative literals)

$$\begin{array}{c} s \simeq t \quad u \not\simeq v \vee R \\ \hline\hline s \simeq t \quad u[p \leftarrow \sigma(t)] \not\simeq v \vee R \end{array} \qquad \begin{array}{l} \text{if } u|_p \;=\; \sigma(s) \text{ and} \\ \sigma(s) > \sigma(t) \end{array}$$

(Equality Resolution)

$$\frac{u \not\simeq v \vee R}{\sigma(R)} \quad \begin{array}{l} \text{if } \sigma \ = \ \mathrm{mgu}(u, v), \\ \ldots \end{array}$$

(Deletion of resolved literals)

$$\begin{array}{c} s \not\simeq s \vee R \\ \hline\hline R \end{array}$$

**>99% of generated clauses**

(Superposition into negative literals)

$$\frac{s \simeq t \vee S \quad u \not\simeq v \vee R}{\sigma(u[p \leftarrow t] \not\simeq v \vee S \vee R)} \quad \begin{array}{l} \text{if } \sigma = mgu(u|_p, s), \\ \sigma(s) \not< \sigma(t), \ \ldots \end{array}$$

**>90% of cpu time**

(Rewriting of negative literals)

$$\begin{array}{c} s \simeq t \quad u \not\simeq v \vee R \\ \hline\hline s \simeq t \quad u[p \leftarrow \sigma(t)] \not\simeq v \vee R \end{array} \quad \begin{array}{l} \text{if } u|_p \ = \ \sigma(s) \text{ and} \\ \sigma(s) > \sigma(t) \end{array}$$

# Basic Data Types

# Running Example

- Clauses:
  - $C_1 = X \not\simeq 0 \vee add(X, s(Y)) \simeq s(Y)$
  - $C_2 = add(s(X), Y) \simeq s(add(X, Y))$
- Properties:
  - $C_1$ is a non-unit Horn clause (at most one positive
  - $C_2$ is a positive unit-clause (one literal) literal)
  - Both are purely equational

# Running Example

- Clauses:
  - $C_1 = X \not\simeq 0 \vee add(X, s(Y)) \simeq s(Y)$
  - $C_2 = add(s(X), Y) \simeq s(add(X, Y))$
- Properties:
  - $C_1$ is a non-unit Horn clause (at most one positive
  - $C_2$ is a positive unit-clause (one literal) literal)
  - Both are purely equational
- Basic components:
  - *add* is a *function symbol* of arity 2
  - *s* is a function symbol of arity 1
  - 0 is a constant (function symbol of arity 0)
  - $X, Y$ are variables (implicitly universally quantified)
  - $\simeq$ represents the equality relation
  - $\not\simeq$ represents the negated equality relation
  - $\vee$ is the disjunctive operator (logical or)

# Running Example

- Clauses:
  - $C_1 = X \not\simeq 0 \vee add(X, s(Y)) \simeq s(Y)$
  - $C_2 = add(s(X), Y) \simeq s(add(X, Y))$
- Properties:
  - $C_1$ is a non-unit Horn clause (at most one positive
  - $C_2$ is a positive unit-clause (one literal) literal)
  - Both are purely equational
- Terms and literals:
  - $X, Y, 0$ are elementary terms
  - $s(X)$ is a (composite) term
  - So are $add(s(X), Y), s(add(X, Y)), add(X, s(Y)), \ldots$
  - $add(s(X), Y) \simeq s(add(X, Y))$ is a positive literal
  - $add(X, s(Y)) \simeq s(Y)$ is a positive literal
  - $X \not\simeq 0$ is a negative literal

# Signature: Encode symbols and represent properties

| **Enc.** | **Name** | **Arity** | **Remarks** |
|---|---|---|---|
| 0 | - | - | Unused |
| 1 | 0 | 0 | |
| 2 | *add* | 2 | |
| 3 | *s* | 1 | |
| 3 | ... | ... | As needed |

- Signature table
  - Associates function symbol with index (small integer)
  - Can store additional information
  - Implement as array: Fast look-up (O(1)) by index
  - Add e.g. tree for fast mapping *name*→*index*

Function symbols (and predicate symbols, if used) are usually represented by small positive integers!

# Real Code for Real Humans

```
typedef struct funccell
{  /* f_code is implicit by position in the array */
   char*  name;
   int    arity;
   ...
   Type_p type;       /* Simple type of the symbol */
   FunctionProperties properties;
}FuncCell, *Func_p;

typedef struct sigcell
{
   long       size;      /* Size of the array */
   FunCode    f_count;   /* Largest used f_code */
   Func_p     f_info;    /* The array */
   StrTree_p  f_index;   /* Back-assoc: symbol=>index */
   ...
   TypeTable_p type_table;
   ...
}SigCell, *Sig_p;
```

# Variables

Function symbols (and predicate symbols, if used) are usually represented by small positive integers!

# Variables

Function symbols (and predicate symbols, if used) are usually represented by small positive integers!

- Variables have no persistent names
    - Each clause is individually universally quantified
    - Scope of a variable name is one clause
- Frequent encoding: Small negative integers
    - $-1$ is the first variable in a clause
    - $-3$ is the second variable in a clause
    - $-5$ is the third variable in a clause
    - …
    - Temporary association index↔name for parsing

# Variables

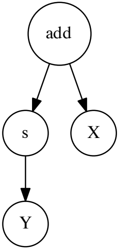
Function symbols (and predicate symbols, if used) are usually represented by small positive integers!

- Variables have no persistent names
  - Each clause is individually universally quantified
  - Scope of a variable name is one clause
- Frequent encoding: Small negative integers
  - $-1$ is the first variable in a clause
  - $-3$ is the second variable in a clause
  - $-5$ is the third variable in a clause
  - ...
  - Temporary association index↔name for parsing
- This leaves even numbers for alternative variables
  - When two clauses interact via unification, the usually need disjoint variable sets!

# Terms are Trees

- Terms are ordered trees
  - Leaves are labeled constants or variables
  - Internal nodes are labeled with non-constant symbols
  - Node with symbol of arity *n* has *n* children

$add(s(Y), X) \Longrightarrow$



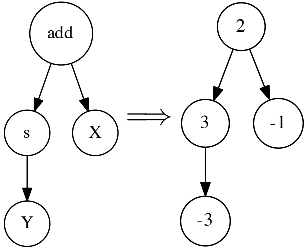
# Terms are Trees

- Terms are ordered trees
  - Leaves are labeled constants or variables
  - Internal nodes are labeled with non-constant symbols
  - Node with symbol of arity $n$ has $n$ children

| Enc. | Name | Arity | Remarks |
|---|---|---|---|
| 0 | - | - | Unused |
| 1 | 0 | 0 | |
| 2 | *add* | 2 | |
| 3 | *s* | 1 | |
| 3 | ... | ... | As needed |

| Enc. | Name |
|---|---|
| -1 | *X* |
| -3 | *Y* |
| -5 | *Z* |

$add(s(Y), X) \Longrightarrow$



# Term Representations

- Cool languages have s-expressions: Terms for free
  - LISP/Scheme: `add(s(Y),X)` $\Longrightarrow$ `(add (s Y) X)`
  - Python: `add(s(Y),X)` $\Longrightarrow$ `[2 [3 -3] -1])`
- Cooler languages have recursive data types
  - ML/Caml/OCaml
  - Haskell

# Term Representations

- Cool languages have s-expressions: Terms for free
  - LISP/Scheme: `add(s(Y),X)` ⟹ `(add (s Y) X)`
  - Python: `add(s(Y),X)` ⟹ `[2 [3 -3] -1])`
- Cooler languages have recursive data types
  - ML/Caml/OCaml
  - Haskell
- C has pointers

```
typedef struct termcell
{
   FunCode           f_code;       /* Top symbol of term */
   TermProperties    properties;   /* Like basic, lhs, top
   int               arity;        /* Redundant, but saves
                                      around the signature
                                      time */
   struct termcell*  *args;        /* Pointer to array of
   ...                                arguments */
}TermCell, *Term_p, **TermRef;
```

# Equations/Inequations/Literals

- Meta-information plus two terms plus `next` pointer
  - This reflects the pure equational paradigm of E
  - Alternative: Atoms are terms (with a predicate symbol as the top symbol)
    - Literals have only one term pointer
    - Equality is just a special predicate symbol

```
typedef struct eqncell
{
   EqnProperties  properties;/* Positive, maximal, eq. */
   Term_p         lterm;
   Term_p         rterm;
   ...
   struct eqncell *next;      /* For lists of equations */
}EqnCell, *Eqn_p, **EqnRef;
```

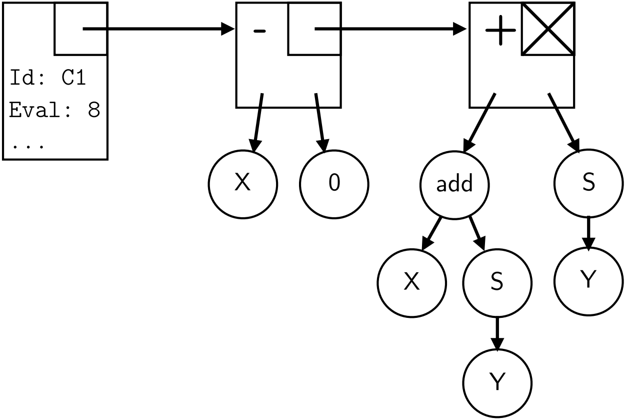
# Clauses

- Meta-information plus list of literals

```
typedef struct clause_cell
{
   long                    ident;        /* Hopefully unique ident
                                            for all clauses created
                                            during a proof run */
   Eqn_p                   literals;     /* List of literals */
   FormulaProperties       properties;
   Eval_p                  evaluations;/* List of evaluations */
   struct clausesetcell*   set;          /* Is the clause in a set?
   struct clause_cell*     pred;         /* For clause sets = doubl
   struct clause_cell*     succ;         /* linked lists */
   ...
}
```

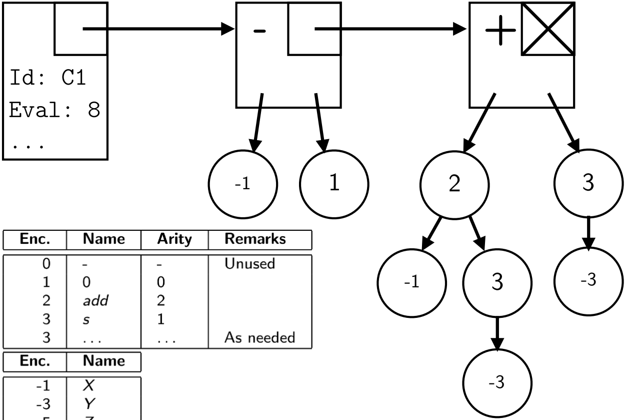
# Clauses

$$C_1 = X \not\simeq 0 \vee add(X, s(Y)) \simeq s(Y)$$



# Clauses

$$C_1 = X \not\simeq 0 \vee add(X, s(Y)) \simeq s(Y)$$

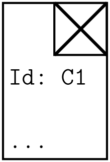


| Enc. | Name | Arity | Remarks |
|---|---|---|---|
| 0 | - | - | Unused |
| 1 | 0 | 0 | |
| 2 | *add* | 2 | |
| 3 | *s* | 1 | |
| 3 | ... | ... | As needed |

| Enc. | Name |
|---|---|
| -1 | *X* |
| -3 | *Y* |
| -5 | *Z* |

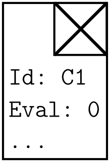
# Clauses

$C_1 = ???$



# Clauses

$C_1 = \square$



```
Id: C1
Eval: 0
...
```

# Saturation Algorithm

(Equality Resolution)

$$\frac{u \not\simeq v \vee R}{\sigma(R)} \quad \begin{array}{l} \text{if } \sigma = \mathrm{mgu}(u, v), \\ \ldots \end{array}$$

# Generating inference rules

- Necessary for completeness
- Increase size of proof state

(Superposition into negative literals)

$$\frac{s \simeq t \vee S \quad u \not\simeq v \vee R}{\sigma(u[p \leftarrow t] \not\simeq v \vee S \vee R)} \quad \begin{array}{l} \text{if } \sigma = mgu(u|_p, s), \\ \sigma(s) \not< \sigma(t), \ \ldots \end{array}$$

(Deletion of resolved literals)

$$\frac{s \not\simeq s \vee R}{R}$$

# Simplification rules

- Critical for performance
- Reduce size of proof state

(Rewriting of negative literals)

$$\frac{s \simeq t \quad u \not\simeq v \vee R}{s \simeq t \quad u[p \leftarrow \sigma(t)] \not\simeq v \vee R} \quad \begin{array}{l} \text{if } u|_p = \sigma(s) \text{ and} \\ \sigma(s) > \sigma(t) \end{array}$$

(Equality Resolution)

$$\frac{u \not\simeq v \lor R}{\sigma(R)} \quad \begin{array}{l} \text{if } \sigma = \mathrm{mgu}(u,v), \\ \ldots \end{array}$$

(Deletion of resolved literals)

$$\begin{array}{c} s \not\simeq s \lor R \\ \hline\hline R \end{array}$$

## Local (single premise)

- Easy to keep track of
- Cheap to implement

---

## Non-local (multiple premises)

- Harder to keep track of (pairs of clauses!)
- Expensive to implement (find partners)

(Superposition into negative literals)

$$\frac{s \simeq t \lor S \quad u \not\simeq v \lor R}{\sigma(u[p \leftarrow t] \not\simeq v \lor S \lor R)} \quad \begin{array}{l} \text{if } \sigma = mgu(u|_p, s), \\ \sigma(s) \not< \sigma(t), \ \ldots \end{array}$$

(Rewriting of negative literals)

$$\begin{array}{c} s \simeq t \quad u \not\simeq v \lor R \\ \hline\hline s \simeq t \quad u[p \leftarrow \sigma(t)] \not\simeq v \lor R \end{array} \quad \begin{array}{l} \text{if } u|_p = \sigma(s) \text{ and} \\ \sigma(s) > \sigma(t) \end{array}$$

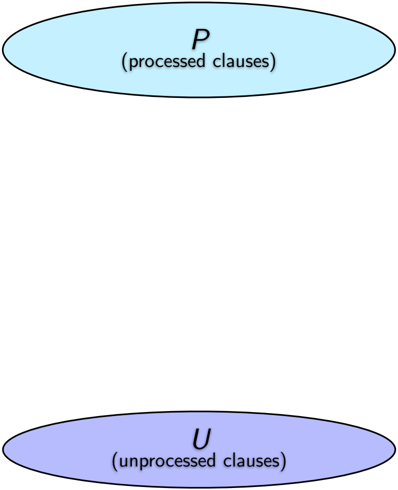
# Requirements for a Saturation Procedure

- Apply all non-redundant generating inferences (in the limit)
  - Necessary for completeness
  - Requires some form of tracking or book-keeping
- Integrate simplification/redundancy elimination
  - Reduces size of proof state and search space
  - Critical for performance
  - Ideal: *No generating inferences involving redundant clauses*
- Support search control
  - No blind saturation
  - Suitable choice point(s) for heuristics
- Low overhead
  - Efficient in time
  - Efficient in memory

# The Given-Clause Algorithm

We start with a blank slate

# The Given-Clause Algorithm



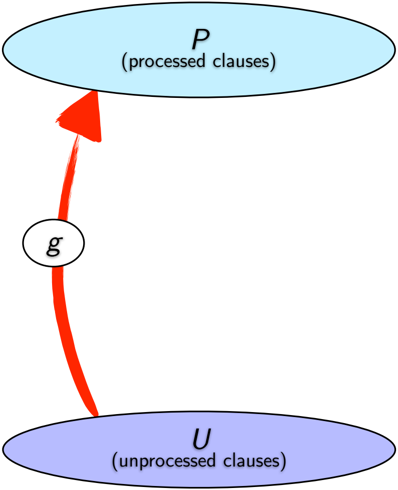
$P$
(processed clauses)

$U$
(unprocessed clauses)

We represent the proof state $S$ by two sets of clauses:

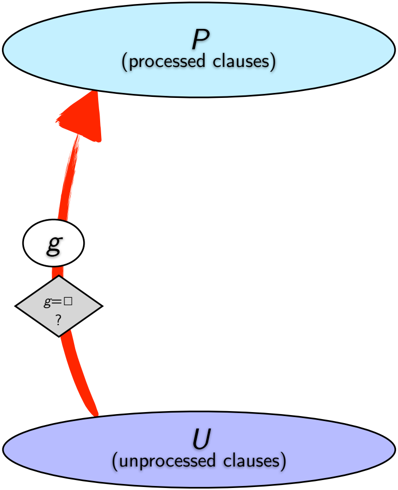
- $P$ holds the processed clauses (originally empty)
- $U$ holds the unprocessed clauses (originally all clauses in $S$)

# The Given-Clause Algorithm



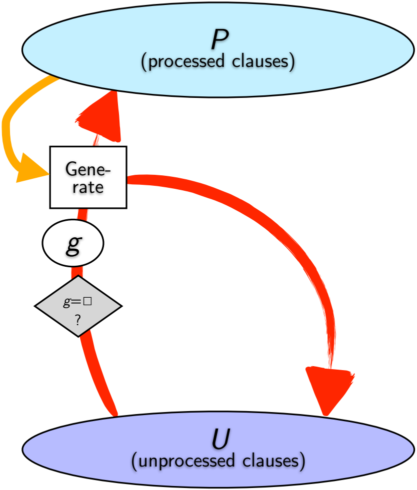
- Aim: Move everything from $U$ to $P$

# The Given-Clause Algorithm



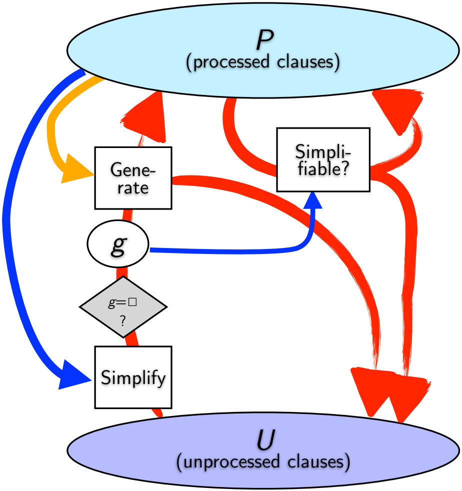
- Aim: Move everything from $U$ to $P$

# The Given-Clause Algorithm



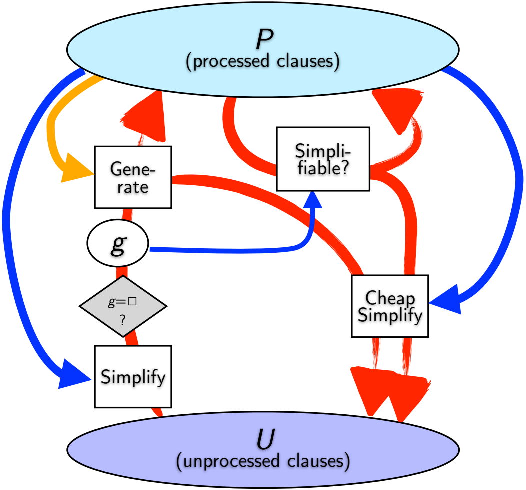
- Aim: Move everything from $U$ to $P$
- Invariant: All generating inferences with premises from $P$ have been performed

# The Given-Clause Algorithm



- Aim: Move everything from $U$ to $P$
- Invariant: All generating inferences with premises from $P$ have been performed
- Invariant: $P$ is interreduced

# The Given-Clause Algorithm

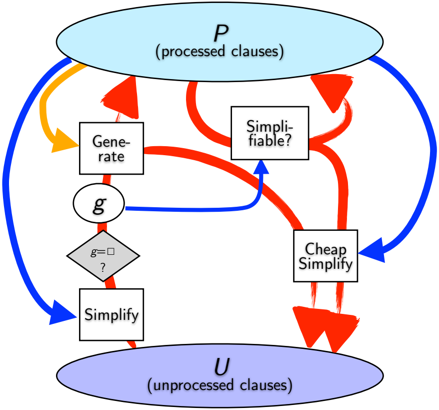


- Aim: Move everything from $U$ to $P$
- Invariant: All generating inferences with premises from $P$ have been performed
- Invariant: $P$ is interreduced
- Clauses added to $U$ are simplified with respect to $P$

# The Given-Clause Loop in Fewer Words

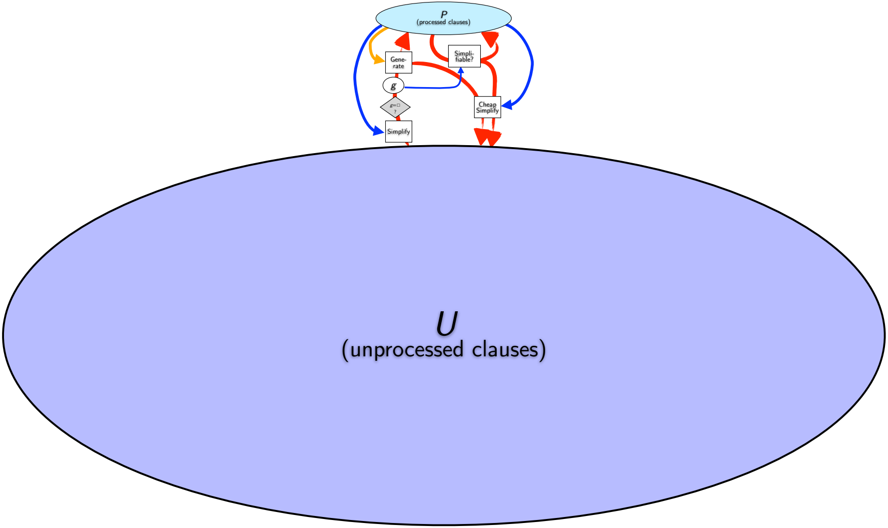
```
while $U \neq \{\}$
  $g = \text{delete\_best}(U)$
  $g = \text{simplify}(g, P)$
  if $g == \Box$
    SUCCESS, Proof found
  if $g$ is not subsumed by any clause in $P$ (or otherwise redundant w.r.t. $P$)
    $P = P \backslash \{c \in P \mid c$ subsumed by (or otherwise redundant w.r.t.) $g\}$
    $T = \{c \in P \mid c$ can be simplified with $g\}$
    $P = (P \backslash T) \cup \{g\}$
    $T = T \cup \text{generate}(g, P)$
    foreach $c \in T$
      $c = \text{cheap\_simplify}(c, P)$
      if $c$ is not trivial
        $U = U \cup \{c\}$
SUCCESS, original $U$ is satisfiable
```

# Compare and Contrast



```
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. $P$
        $P = P \backslash \{c \in P \mid c$ redundant w.r.t. $g\}$
        $T = \{c \in P \mid c$ simplifiable with $g\}$
        $P = (P \backslash T) \cup \{g\}$
        $T = T \cup$ 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
```

# “You can’t handle the truth!”



# Term and Clause Indexing

(Equality Resolution)

$$\frac{u \not\simeq v \vee R}{\sigma(R)} \quad \begin{array}{l} \text{if } \sigma = \mathrm{mgu}(u, v), \\ \ldots \end{array}$$

(Deletion of resolved literals)

$$\begin{array}{c} s \not\simeq s \vee R \\ \hline\hline R \end{array}$$

(Superposition into negative literals)

$$\frac{s \simeq t \vee S \quad u \not\simeq v \vee R}{\sigma(u[p \leftarrow t] \not\simeq v \vee S \vee R)} \quad \begin{array}{l} \text{if } \sigma = mgu(u|_p, s), \\ \sigma(s) \not< \sigma(t), \ \ldots \end{array}$$

**>90% of cpu time**

(Rewriting of negative literals)

$$\begin{array}{c} s \simeq t \quad u \not\simeq v \vee R \\ \hline\hline s \simeq t \quad u[p \leftarrow \sigma(t)] \not\simeq v \vee R \end{array} \quad \begin{array}{l} \text{if } u|_p = \sigma(s) \text{ a} \\ \sigma(s) > \sigma(t) \end{array}$$

# Rewriting

Idea: Replace terms by semantically equal but $>$-smaller terms

(Rewriting of negative literals)

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

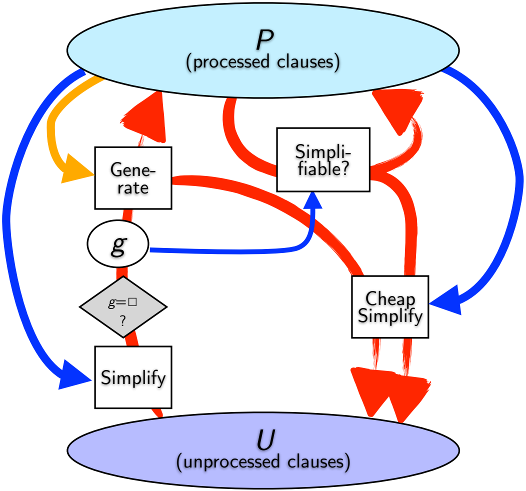
if $u|_p = \sigma(s)$ and $\sigma(s) > \sigma(t)$

(Rewriting of positive literals)

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

if $u|_p = \sigma(s)$ and $\sigma(s) > \sigma(t)$ and $u \simeq v$ is not maximal in $u \simeq v \vee R$ or $u < v$ or $p \neq \lambda$

# Reminder: Rewriting in Action



- Maximally simplify $g$ with respect to all unit equations in $P$
- Maximally simplify all clauses inserted into $U$ with respect to all unit equations in $P$
- Check for all clauses in $P$ if they can be simplified with $g$

# Simplification

Compute normal form of clause $C$ with respect to $P$:

```
while C is not in normal form:
    for all literals l in C:
        for all terms t in l:
            for all subterms s of t:
                for all unit clauses l=r in P:
                     sigma = match(l,s)
                     if sigma and other conditions:
                         replace s by sigma(r)
                     else:
                         sigma = match(r,s)
                         if sigma and other conditions:
                             replace s by sigma(l)
```

# Simplification

Compute normal form of clause *C* with respect to *P*:

```
while C is not in normal form:
    for all literals l in C:
        for all terms t in l:
            for all subterms s of t:
                for all unit clauses l=r in P:
                     sigma = match(l,s)
                     if sigma and other conditions:
                         replace s by sigma(r)
                     else:
                         sigma = match(r,s)
                         if sigma and other conditions:
                             replace s by sigma(l)
```

…and $|P|$ is growing to $\approx 10^5$ in 300 seconds!

# Term Indexing

A *term index* is a data structure supporting one or more of the following operations:

- Given a term $t$, find all terms $s$ from some set $S$ such that
  - $s$ matches $t$ (our current use case)
    - $t$ is the subterm to be rewritten
    - Every $s$ is a potential left hand side of a unit clause
  - $t$ matches $s$
  - $s$ and $t$ can be unified
- Indexing can be...
  - Perfect - all retrieved terms $s$ are in the retrieval relation with the query term $t$
  - Non-perfect - index returns a (hopefully small) superset of candidates

# Top Symbol Hashing

- Assume $t \equiv \mathbf{f}(t_1, \ldots, t_n)$
- Observation:
  - $\sigma(t) = s$ implies $s \equiv \mathbf{f}(s_1, \ldots, s_n)$
  - $t = \sigma(s)$ implies $s \equiv \mathbf{f}(s_1, \ldots, s_n)$ or $s \equiv x \in V$
  - $\sigma(t) = \sigma(s)$ implies $s \equiv \mathbf{f}(s_1, \ldots, s_n)$ or $s \equiv x \in V$
- … because $\sigma$ never affects $f$

# Top Symbol Hashing

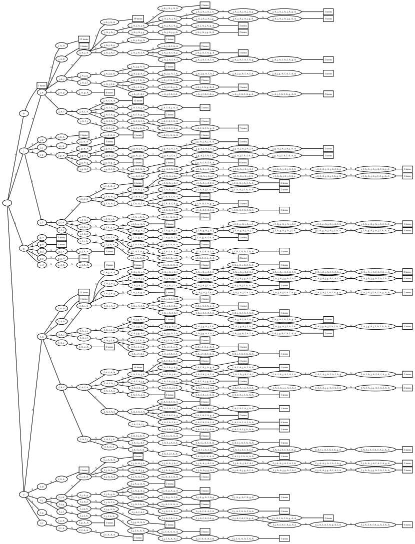
- Assume $t \equiv \mathbf{f}(t_1, \ldots, t_n)$
- Observation:
  - $\sigma(t) = s$ implies $s \equiv \mathbf{f}(s_1, \ldots, s_n)$
  - $t = \sigma(s)$ implies $s \equiv \mathbf{f}(s_1, \ldots, s_n)$ or $s \equiv x \in V$
  - $\sigma(t) = \sigma(s)$ implies $s \equiv \mathbf{f}(s_1, \ldots, s_n)$ or $s \equiv x \in V$
- . . . because $\sigma$ never affects $f$

**If we organize $P$ by top symbol of potentially maximal sides of unit clauses, we can avoid most matching attempts!**

# Fingerprint/Discrimination Tree Indexing

- Core idea: Iterated top symbol hashing
  - Traverse tern left-to-right
  - Each function symbol potentially restricts candidates for matching/unification
- Index data structure: Trie
  - Convert terms to symbol strings: $add(s(X), Y) \Longrightarrow$ *'add s X Y'*
  - Organize strings in a trie
  - Leaf nodes carry original terms and associated date (e.g. originating clause)
- Retrieval:
  - Convert query term into symbols string
  - Follow all compatible branches
  - If a leaf is reached, try candidates stored there
    - For *perfect discrimination trees*, only terms compatible with the retrieval relation will be found
    - For *fingerprint indexing* and *non-perfect discrimination tree indexing*: Candidates must be checked

# Impressionist Art



- Non-perfect discrimination tree
- Clause set $P$ from final state of 6th Lusk/Overbeck problem
  - Unit-equational example
  - A ring with $x * x * x = x$ is commutative
  - Historically considered hard
  - Now: 0.2 seconds on this computer

# Indexing - Summary

- Term indexing can convert the time needed to find inference partners from roughly $O(|P|)$ to $O(\log(|P|))$
- Unification indices speed up paramodulation
  - E.g. Fingerprint Indexing (Choice for E)
  - E.g. Discrimination Tree Indexing (but ugly unification)
- Find-Matching indices speed up forward rewriting
  - E.g. Discrimination Tree Indexing (Choice for E)
  - E.h. Fingerprint Indexing
- Find-Matched indices speed up backwards-rewriting
  - E.g. Fingerprint Indexing (Choice for E)
- Subsumption indices speed up subsumption
  - Index clauses, not terms
  - E.g. Feature Vector Indexing

# Search Control

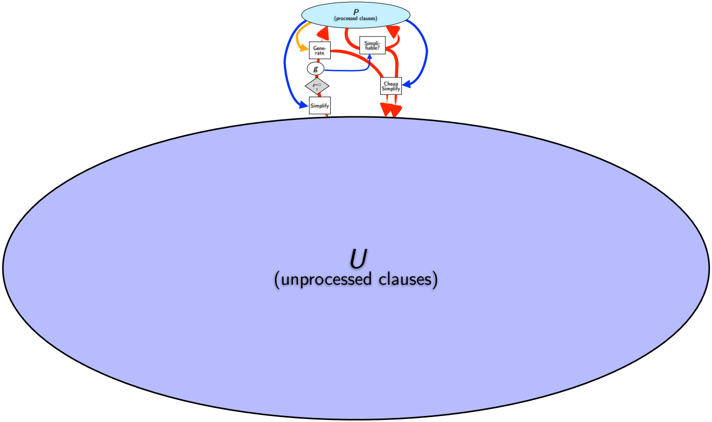
# Search Heuristics

- Heuristics are crucial for first-order theorem provers
  - Practical experience is clear
  - Proof search happens in an infinite search space
  - Proofs are rare
- Three major choice points
  - Choice of the term ordering
  - Choice of the literal selection strategy
  - Choice of the next given clause

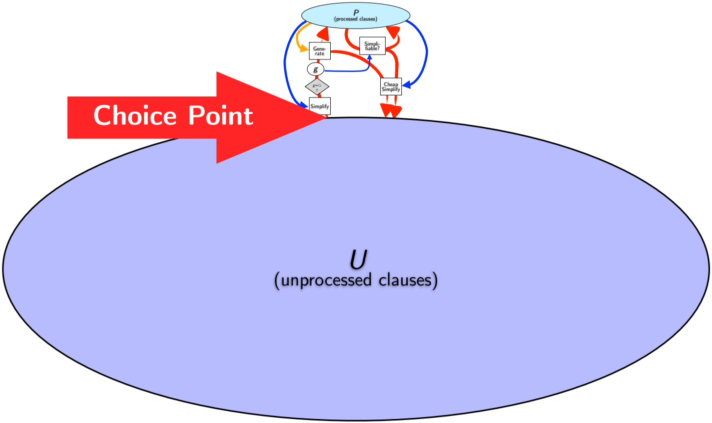
# Search Heuristics

- Heuristics are crucial for first-order theorem provers
  - Practical experience is clear
  - Proof search happens in an infinite search space
  - Proofs are rare
- Three major choice points
  - Choice of the term ordering
  - Choice of the literal selection strategy
  - **Choice of the next given clause**

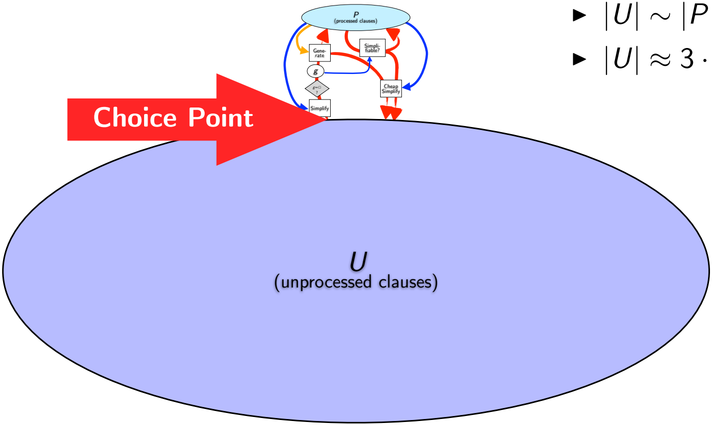
# The Size of the Problem



# The Size of the Problem

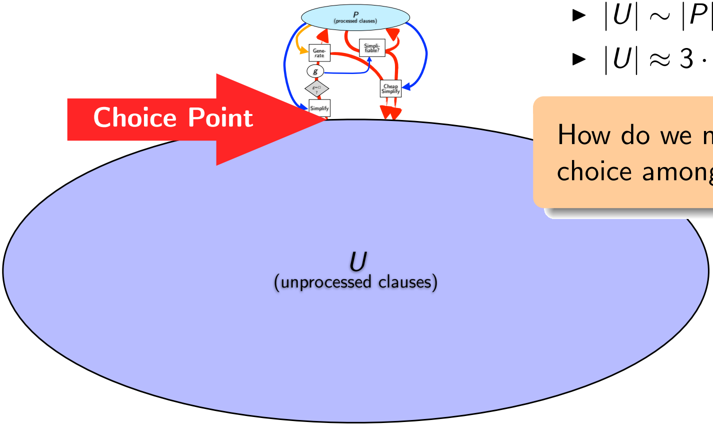


# The Size of the Problem



- $|U| \sim |P|^2$
- $|U| \approx 3 \cdot 10^7$ after 300s

# The Size of the Problem



- $|U| \sim |P|^2$
- $|U| \approx 3 \cdot 10^7$ after 300s

How do we make the best choice among millions?

# Basic Clause Selection Heuristics

- Basic idea: Clauses ordered by heuristic evaluation
  - Heuristic assigns a numerical value to a clause
  - Clauses with smaller (better) evaluations are processed first
- Example: Evaluation by symbol counting
  - $|\{f(X) \neq a, P(a) \neq \$true, g(Y) = f(a)\}| = 10$
  - Motivation: Small clauses are general, $\square$ has 0 symbols
  - *Best-first* search
- Example: FIFO evaluation
  - Clause evaluation based on generation time (always prefer older clauses)
  - Motivation: Simulate *breadth-first* search, find shortest proofs
- Combine best-first/breadth-first search
  - E.g. pick 4 out of every 5 clauses according to size, the last according to age

# Clause Selection Heuristics in E

- Many symbol-counting variants
  - E.g. Assign different weights to symbol classes (predicates, functions, variables)
  - E.g. Goal directed: lower weight for symbols occurring in original conjecture
  - E.g. ordering-aware/calculus-aware: higher weight for symbols in inference terms
- Arbitrary combinations of base evaluation functions
  - E.g. 5 priority queues ordered by different evaluation functions, weighted round-robin selection

# Clause Selection Heuristics in E

- Many symbol-counting variants
    - E.g. Assign different weights to symbol classes (predicates, functions, variables)
    - E.g. Goal directed: lower weight for symbols occurring in original conjecture
    - E.g. ordering-aware/calculus-aware: higher weight for symbols in inference terms
- Arbitrary combinations of base evaluation functions
    - E.g. 5 priority queues ordered by different evaluation functions, weighted round-robin selection

E can simulate nearly all other approaches to clause selection!

# Folklore on Clause Selection/Evaluation

- FIFO is obviously fair, but awful – *Everybody*
- Preferring small clauses is good – *Everybody*
- Interleaving best-first (small) and breadth-first (FIFO) is better
  - *"The optimal pick-given ratio is 5" – Otter*
- Processing all initial clauses early is good – *Waldmeister*
- Preferring clauses with orientable equation is good – *DISCOUNT*
- Goal-direction is good – *E*

# Folklore on Clause Selection/Evaluation

- FIFO is obviously fair, but awful – *Everybody*
- Preferring small clauses is good – *Everybody*
- Interleaving best-first (small) and breadth-first (FIFO) is better
  - *"The optimal pick-given ratio is 5" – Otter*
- Processing all initial clauses early is good – *Waldmeister*
- Preferring clauses with orientable equation is good – *DISCOUNT*
- Goal-direction is good – *E*

Can we confirm or refute these claims?

# Experimental setup

- Prover: E 1.9.1-pre
- 14 different heuristics
  - 13 selected to test folklore claims (interleave 1 or 2 evaluations)
  - Plus modern *evolved* heuristic (interleaves 5 evaluations)
- TPTP release 6.3.0
  - Only (assumed) provable first-order problems
  - 13774 problems: 7082 FOF and 6692 CNF
- Compute environment
  - StarExec cluster: single threaded run on Xeon E5-2609 (2.4 GHz)
  - 300 second time limit, no memory limit ($\geq$64 GB/core physical)

# Meet the Heuristics

| Heuristic | Rank | Successes | | Successes within 1s | |
|---|---|---|---|---|---|
| | | total | unique | absolute | of column 3 |
| FIFO | 14 | 4930 (35.8%) | 17 | 3941 | 79.9% |
| SC12 | 13 | 4972 (36.1%) | 5 | 4155 | 83.6% |
| SC11 | 9 | 5340 (38.8%) | 0 | 4285 | 80.2% |
| SC21 | 10 | 5326 (38.7%) | 17 | 4194 | 78.7% |
| RW212 | 11 | 5254 (38.1%) | 13 | 5764 | 79.8% |
| 2SC11/FIFO | 7 | 7220 (52.4%) | 24 | 5846 | 79.7% |
| 5SC11/FIFO | 5 | 7331 (53.2%) | 3 | 5781 | 78.3% |
| 10SC11/FIFO | 3 | 7385 (53.6%) | 1 | 5656 | 77.6% |
| 15SC11/FIFO | 6 | 7287 (52.9%) | 6 | 5006 | 82.5% |
| GD | 12 | 4998 (36.3%) | 12 | 5856 | 78.4% |
| 5GD/FIFO | 4 | 7379 (53.6%) | 62 | 4213 | 80.2% |
| SC11-PI | 8 | 6071 (44.1%) | 13 | 4313 | 86.3% |
| 10SC11/FIFO-PI | 2 | 7467 (54.2%) | 31 | 5934 | 80.4% |
| Evolved | 1 | 8423 (61.2%) | 593 | 6406 | 76.1% |

# Folklore put to the Test

- FIFO is awful, preferring small clauses is good – mostly confirmed
  - In general, only modest advantage for symbol counting (36% FIFO vs. 39% for best SC)
  - Exception: UEQ (32% vs. 63%)

# Folklore put to the Test

- FIFO is awful, preferring small clauses is good – mostly confirmed
  - In general, only modest advantage for symbol counting (36% FIFO vs. 39% for best SC)
  - Exception: UEQ (32% vs. 63%)
- Interleaving best-first/breadth-first is better – confirmed
  - 54% for interleaving vs. 39% for best SC
  - Influence of different pick-given ratios is surprisingly small
  - UEQ is again an outlier (60% for 2:1 vs. 70% for 15:1)
  - *The optimal pick-given ratio is 10* (for E)

# Folklore put to the Test

- FIFO is awful, preferring small clauses is good – mostly confirmed
  - In general, only modest advantage for symbol counting (36% FIFO vs. 39% for best SC)
  - Exception: UEQ (32% vs. 63%)
- Interleaving best-first/breadth-first is better – confirmed
  - 54% for interleaving vs. 39% for best SC
  - Influence of different pick-given ratios is surprisingly small
  - UEQ is again an outlier (60% for 2:1 vs. 70% for 15:1)
  - *The optimal pick-given ratio is 10* (for E)
- Processing all initial clauses early is good – confirmed
  - Effect is less pronounced for interleaved heuristics

# Folklore put to the Test

- FIFO is awful, preferring small clauses is good – mostly confirmed
  - In general, only modest advantage for symbol counting (36% FIFO vs. 39% for best SC)
  - Exception: UEQ (32% vs. 63%)
- Interleaving best-first/breadth-first is better – confirmed
  - 54% for interleaving vs. 39% for best SC
  - Influence of different pick-given ratios is surprisingly small
  - UEQ is again an outlier (60% for 2:1 vs. 70% for 15:1)
  - *The optimal pick-given ratio is 10* (for E)
- Processing all initial clauses early is good – confirmed
  - Effect is less pronounced for interleaved heuristics
- Preferring clauses with orientable equation is good – not confirmed
  - There is no evidence in our data, not even for UEQ

# Folklore put to the Test

- FIFO is awful, preferring small clauses is good – mostly confirmed
  - In general, only modest advantage for symbol counting (36% FIFO vs. 39% for best SC)
  - Exception: UEQ (32% vs. 63%)
- Interleaving best-first/breadth-first is better – confirmed
  - 54% for interleaving vs. 39% for best SC
  - Influence of different pick-given ratios is surprisingly small
  - UEQ is again an outlier (60% for 2:1 vs. 70% for 15:1)
  - *The optimal pick-given ratio is 10* (for E)
- Processing all initial clauses early is good – confirmed
  - Effect is less pronounced for interleaved heuristics
- Preferring clauses with orientable equation is good – not confirmed
  - There is no evidence in our data, not even for UEQ
- Goal-direction is good – partially confirmed
  - GD on its own performs similar to SC
  - GD shines in combination with FIFO

# Selected Results

- Good heuristics do make a difference
    - 71% more solutions with Evolved vs. FIFO
    - 58% more solutions with Evolved vs. best SC

# Selected Results

- Good heuristics do make a difference
  - 71% more solutions with Evolved vs. FIFO
  - 58% more solutions with Evolved vs. best SC
- Success comes early
  - $\approx$ 80% of all proofs found in less than 1s
  - . . . with little variation between strategies (spread: 76%–84%)

# Selected Results

- Good heuristics do make a difference
    - 71% more solutions with Evolved vs. FIFO
    - 58% more solutions with Evolved vs. best SC
- Success comes early
    - $\approx$ 80% of all proofs found in less than 1s
    - . . . with little variation between strategies (spread: 76%–84%)
- Cooperation beats portfolio/strategy scheduling
    - SC11 solves 5340 problems
    - FIFO solves 4930 problems
    - Union of the previous two contains 6329 problems
    - . . . but 10SC11/FIFO solves 7385

# Selected Results

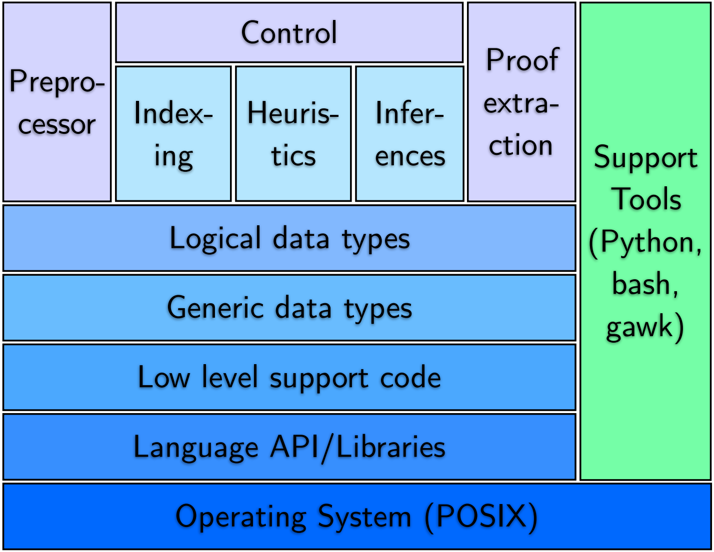
- Good heuristics do make a difference
  - 71% more solutions with Evolved vs. FIFO
  - 58% more solutions with Evolved vs. best SC
- Success comes early
  - $\approx$ 80% of all proofs found in less than 1s
  - …with little variation between strategies (spread: 76%–84%)
- Cooperation beats portfolio/strategy scheduling
  - SC11 solves 5340 problems
  - FIFO solves 4930 problems
  - Union of the previous two contains 6329 problems
  - …but 10SC11/FIFO solves 7385
- Evolving *Evolved* paid off
  - Significantly better than best "naive" heuristic
  - 10× more unique solutions than second-best

# Bonus Material

# Small things…

- Version control (`git` and beyond)
  - Value of version control should be self-evident
  - But: Have the system *print* a version number
    - Automatically record the version number with experiments
    - Loudly complain about bug reports without version number (and command line and input file)
- Automatic input format
  - E supports TPTP-3 (TFA/FOF/CNF), TPTP-2 and LOP
    - Originally only selectable by option (e.g. `--tptp3-in`)
    - Now automatic default based on first token
  - Automatic format is a big win for convenience

# 90% of the Iceberg is Under Water



Preprocessor

Control

Indexing

Heuristics

Inferences

Proof extraction

Support Tools (Python, bash, gawk)

Logical data types

Generic data types

Low level support code

Language API/Libraries

Operating System (POSIX)

# Alternative Term Implementations

- Shared terms (e.g. in E)
  - Structure as for normal terms
  - Only one copy of structurally identical terms ($f(a, a)$ has two nodes)
  - Can safe 80% to 99.9% of term nodes
  - Can be used to cache values/results
  - Can use garbage collection (even in C) - saves a lot of time and headaches!
- Flat terms (e.g. in Waldmeister, Twee)
  - Term is flat string of symbol encoding
  - Very space-efficient
  - Very fast traversal for matching
  - Harder to manipulate

# Implementing Term Orderings

- Lexicographic Path Ordering (LPO)
  - Based on function symbol precedence
  - Top symbols and symbol bags decide
  - Lexicographic decomposition for identical top symbols
  - LPO has theoretical advantages
    - Can orient equations towards sides with more variable occurrences
    - Can orient distributively *the right way* with $\times > +$:
      $X \times (Y + Z) \rightarrow (X \times Y) + (X \times Z)$
- Knuth-Bendix-Ordering (KBO)
  - Based on symbols weights and precedence
  - Weight, top symbol, decomposition, *variable condition*
  - KBO has practical advantages
    - Orients towards syntactically small terms (keeps terms smaller)
    - More efficient to evaluate
- For efficient algorithms: Löchner's *Things to Know When Implementing [KL]PO* [5, 6]

# Conclusion

# Conclusion

- Building a basic saturating prover is not hard
  - Binary resolution+factoring
  - Unification
  - (Subsumption)
- Building a superposition prover is harder
  - Term orderings
  - Rewriting/Simplification
- Being competitive is harder still
  - Indexing
  - Search heuristics
  - Experimental infrastructure

# Conclusion

- Building a basic saturating prover is not hard
    - Binary resolution+factoring
    - Unification
    - (Subsumption)
- Building a superposition prover is harder
    - Term orderings
    - Rewriting/Simplification
- Being competitive is harder still
    - Indexing
    - Search heuristics
    - Experimental infrastructure

**The knowledge needed to write a decent prover is now mostly publicly available!**

# Conclusion II

- Maintenance and evolution are hardest!
    - Users want new features
    - Conference/competition deadline always loom
    - Result: Code clutter
    - Unexpected dependencies and side effects
- . . . so keep it clean!
    - Have a plan
    - Code clean and general building blocks
    - Take the time to refactor/cleanup (yeah, right!)
    - . . . or throw it away and start over
- Remember: 90% of the iceberg. . .

# Conclusion II

- Maintenance and evolution are hardest!
  - Users want new features
  - Conference/competition deadline always loom
  - Result: Code clutter
  - Unexpected dependencies and side effects
- . . . so keep it clean!
  - Have a plan
  - Code clean and general building blocks
  - Take the time to refactor/cleanup (yeah, right!)
  - . . . or throw it away and start over
- Remember: 90% of the iceberg. . .

**Most Important: Have Fun!**

# References

# References I

Leo Bachmair and Harald Ganzinger.
On Restrictions of Ordered Paramodulation with Simplification.
In M.E. Stickel, editor, *Proc. of the 10th CADE, Kaiserslautern*, volume 449 of *LNAI*, pages 427—441. Springer, 1990.

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

J. Christian.
Flat Terms, Discrimination Nets and Fast Term Rewriting.
*Journal of Automated Reasoning*, 10(1):95–113, 1993.

Laura Kovács and Andrei Voronkov.
First-order theorem proving and Vampire.
In Natasha Sharygina and Helmut Veith, editors, *Proc. of the 25th CAV*, volume 8044 of *LNCS*, pages 1–35. Springer, 2013.

Bernd Löchner.
Things to Know when Implementing KBO.
*Journal of Automated Reasoning*, 36(4):289–310, 2006.

Bernd Löchner.
Things to Know When Implementing LPO.
*International Journal on Artificial Intelligence Tools*, 15(1):53–80, 2006.

E.L. Lusk and R.A. Overbeek.
A Short Problem Set for Testing Systems that Include Equality Reasoning.
Technical report, Argonne National Laboratory, Illinois, 1982.

# References II

R. Nieuwenhuis and A. Rubio.
Paramodulation-Based Theorem Proving.
In A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*, volume I, chapter 7, pages 371–443. Elsevier Science and MIT Press, 2001.

A. Nonnengart and C. Weidenbach.
Computing Small Clause Normal Forms.
In A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*, volume I, chapter 5, pages 335–367. Elsevier Science and MIT Press, 2001.

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

Stephan Schulz.
Fingerprint Indexing for Paramodulation and Rewriting.
In Bernhard Gramlich, Ulrike Sattler, and Dale Miller, editors, *Proc. of the 6th IJCAR, Manchester*, volume 7364 of *LNAI*, pages 477–483. Springer, 2012.

Stephan Schulz.
Simple and Efficient Clause Subsumption with Feature Vector Indexing.
In Maria Paola Bonacina and Mark E. Stickel, editors, *Automated Reasoning and Mathematics: Essays in Memory of William W. McCune*, volume 7788 of *LNAI*, pages 45–67. Springer, 2013.

Stephan Schulz.
System Description: E 1.8.
In Ken McMillan, Aart Middeldorp, and Andrei Voronkov, editors, *Proc. of the 19th LPAR, Stellenbosch*, volume 8312 of *LNCS*, pages 735–743. Springer, 2013.

# References III

Stephan Schulz.
E 2.0 user manual.
EasyChair Preprint no. 8, 2018.

Stephan Schulz and Martin Möhrmann.
Performance of clause selection heuristics for saturation-based theorem proving.
In Nicola Olivetti and Ashish Tiwari, editors, *Proc. of the 8th IJCAR, Coimbra*, volume 9706 of *LNAI*, pages 330–345. Springer, 2016.

R. Sekar, I.V. Ramakrishnan, and A. Voronkov.
Term Indexing.
In A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*, volume II, chapter 26, pages 1853–1961. Elsevier Science and MIT Press, 2001.

Christoph Weidenbach, Dilyana Dimova, Arnaud Fietzke, Rohit Kumar, Martin Suda, and Patrick Wischnewski.
SPASS Version 3.5.
In Renate Schmidt, editor, *Proc. of the 22nd CADE, Montreal, Canada*, volume 5663 of *LNAI*, pages 140–145. Springer, 2009.

# Image Sources

- Public domain via the Wikimedia Commons
  - Albert Bonsack's Cigarette Rolling Machine `https://commons.wikimedia.org/wiki/File:Bonsack_machine.png`
- Clipart: `http://openclipart.org`
- Others: Painstakingly drawn by the author