Implementation of First-Order Theorem Provers

Summer School 2009: Verification Technology, Systems & Applications

Stephan Schulz

schulz@eprover.org

First-Order Theorem Proving

Given: A set axioms and a hypothesis in first-order logic

$$A = \{A_1, \ldots, A_n\},\ H$$

Question: Do the axioms logically imply the hypothesis?

$$A \overset{?}{\models} H$$

An automated theorem prover tries to solve this question!

First-Order Logic with Equality

- First order logic deals with
  - Elements
  - Relations between elements
  - Functions over elements
  - . . . and their combination

- Allows general statements using quantified variables
  - There exists an X so that property P holds ($\exists X : P(X)$)
  - For all possible values of X property P holds ($\forall X : P(X)$)

- Function and predicate symbols are uninterpreted
  - No implicit background theory
  - All properties have to be specified explicitely
  - Exception: Equality is interpreted (as a congruence relation)

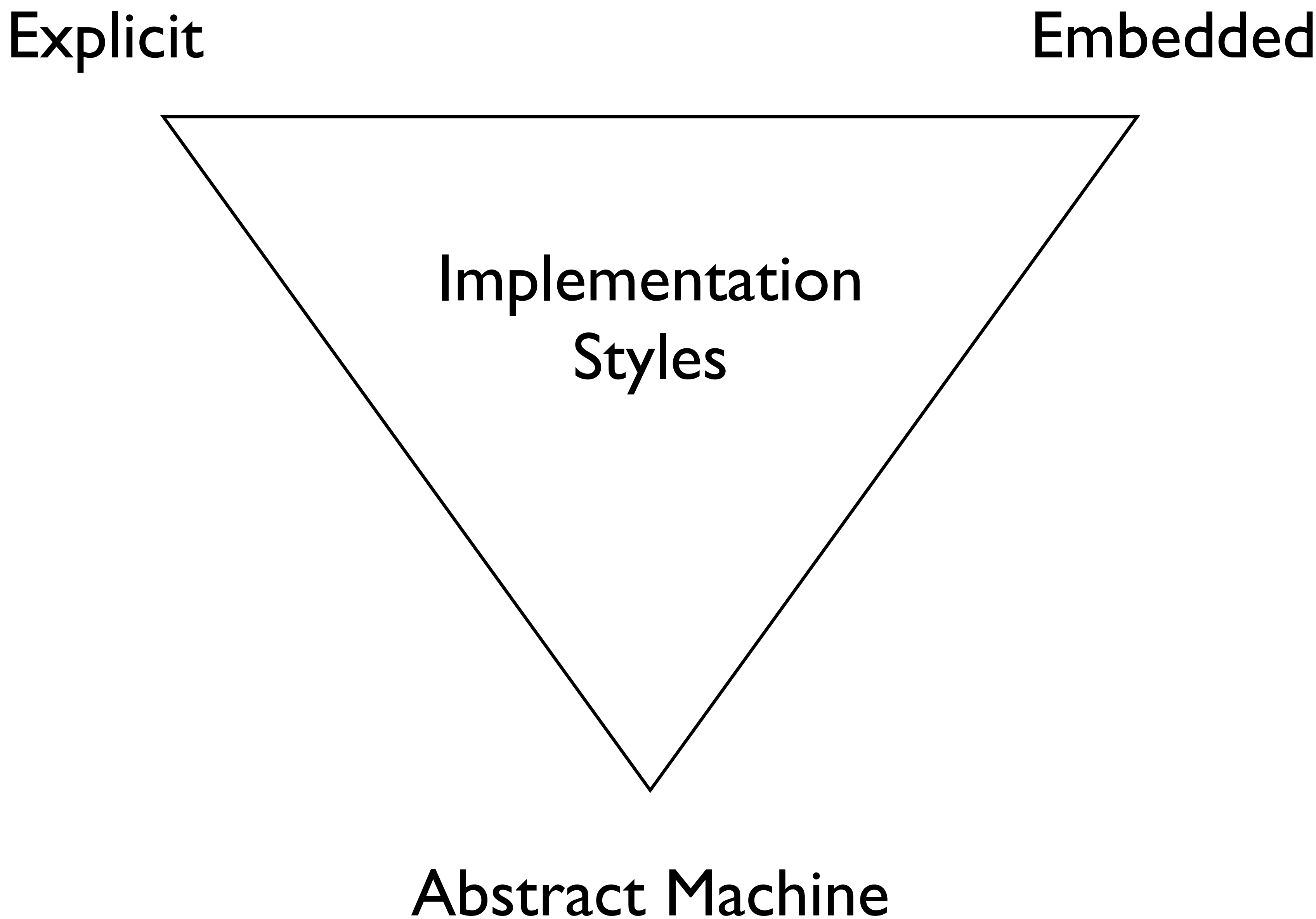
Why First-Order Logic?

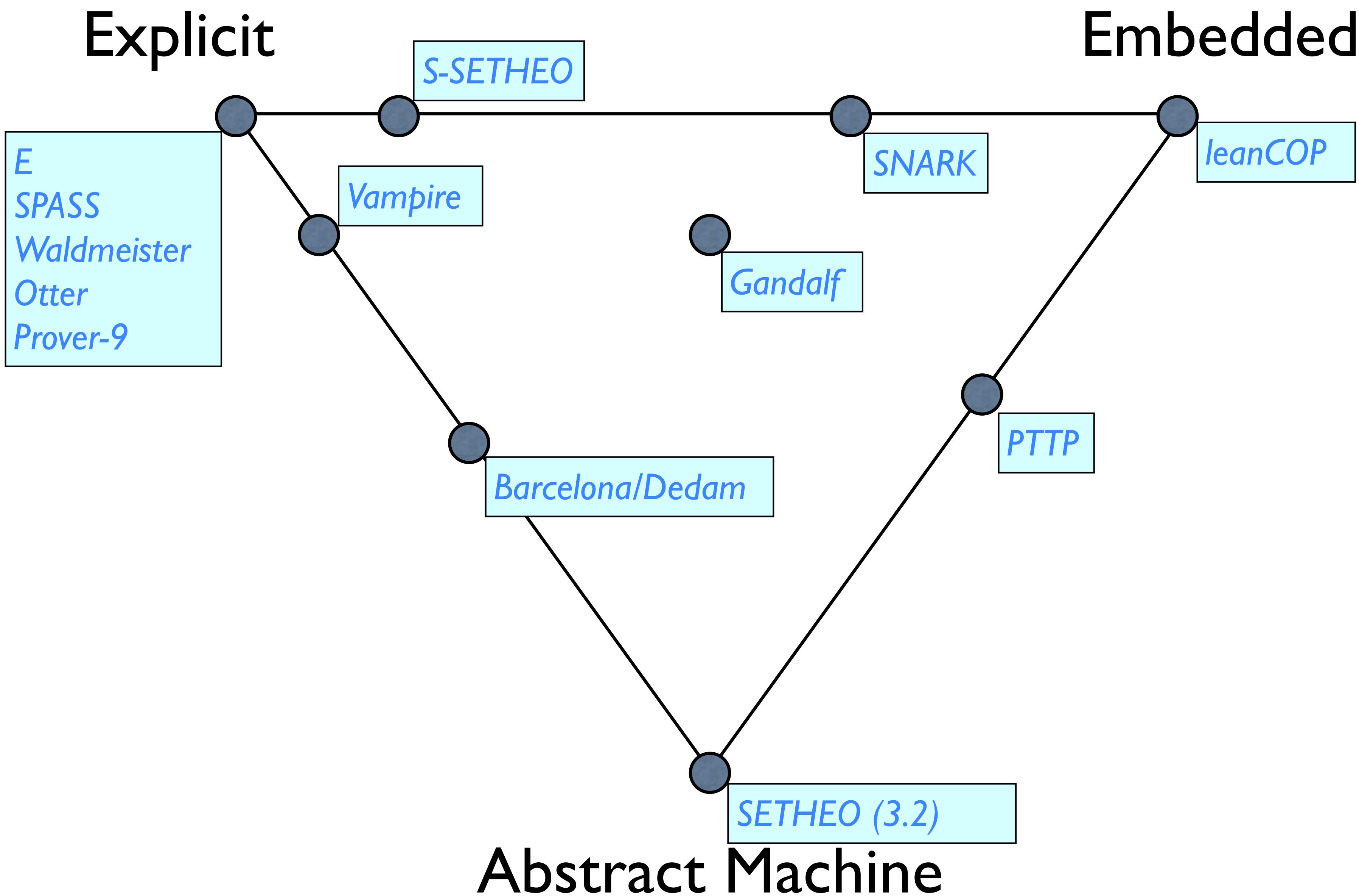
- Expressive:
  - Can encode any computable problem
  - Most tasks can be specified reasonably naturally
  - Many other logics can be reasonably translated to first-order logic
- Automatizable:
  - Sound and complete calculi for proof search exist
  - Search procedures are reasonably efficient
- Stable:
  - Logic is well-known and well-understood
  - Semantics are clear (and somewhat intuitive)

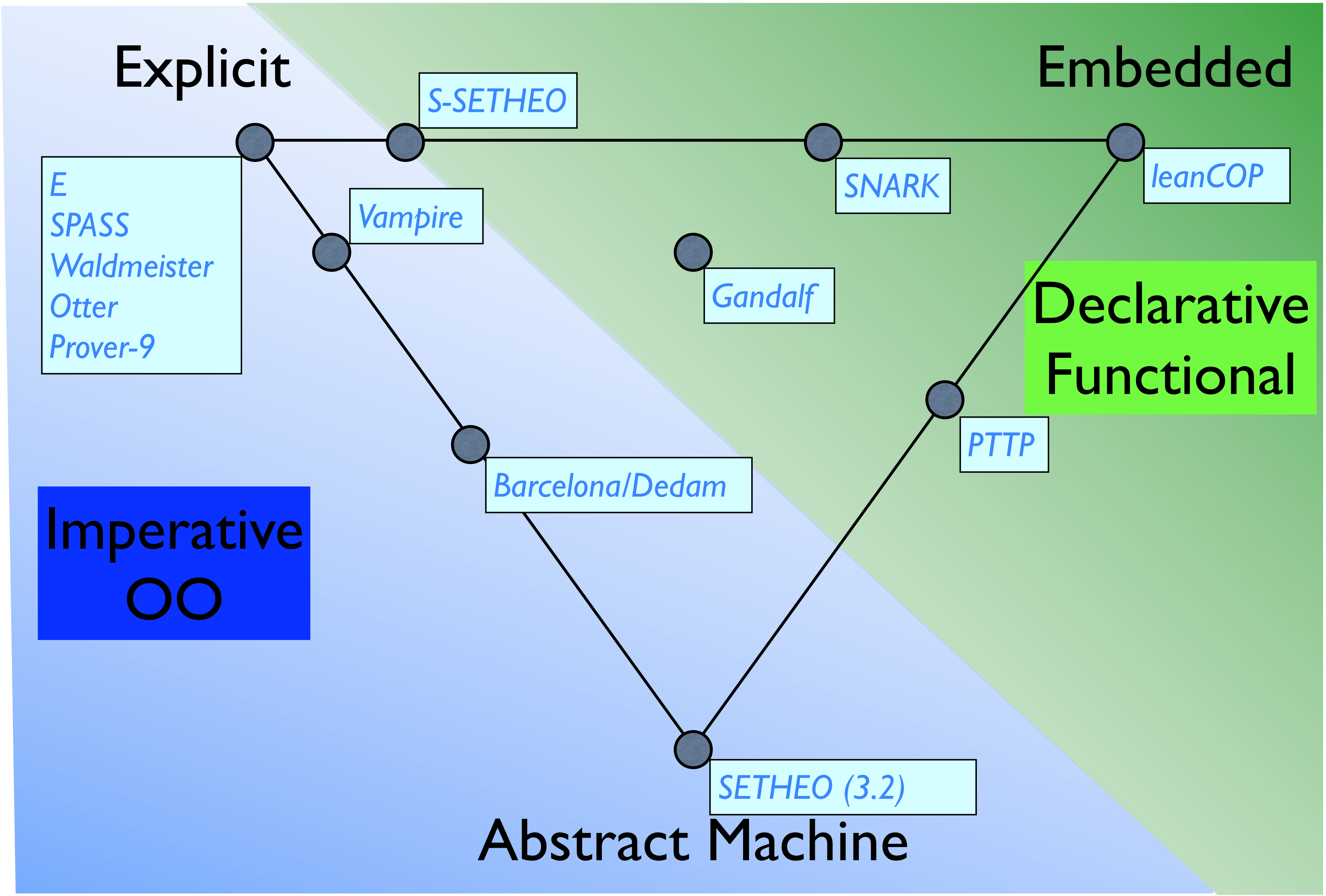
First-order logic is a good compromise between expressiveness and automatizability

Mainstream Milestones

- Herbrand-Universe Enumeration+SAT [DP60]
- Resolution [Rob65]
- Model Elimination [Lov68]
- Paramodulation [RW69]
- Completion [KB70]
- Otter 1.0 (1989, McCune)
- Unfailing completion [BDP89, HR87]
- Superposition [BG90, NR92, BG94]
- SETHEO [LSBB92]
- Vampire [Vor95] (but kept hidden for years)
- First CASC competition at Rutgers, FLOC'96 (Sutcliffe, Suttner)
- Waldmeister [BH96]
- SPASS [WGR96]
- E [Sch99]


Explicit
Embedded
Implementation
Styles
Abstract Machine


Explicit
Embedded
S-SETHEO
E
SPASS
Waldmeister
Otter
Prover-9
Vampire
SNARK
leanCOP
Gandalf
PTTP
Barcelona/Dedam
SETHEO (3.2)
Abstract Machine


Explicit
Embedded
S-SETHEO
E
SPASS
Waldmeister
Otter
Prover-9
Vampire
SNARK
leanCOP
Gandalf
Declarative
Functional
PTTP
Barcelona/Dedam
Imperative
OO
SETHEO (3.2)
Abstract Machine

Implementation Style (References)

Barcelona/Dedam [NRV97]
Gandalf [Tam97]
PTTP [Sti92, Sti89]
S-SETHEO [LS01b]
SPASS [Wei01, WSH$^+$07]
Vampire [RV02]
leanCOP [OB03, Ott08]
E [Sch02, Sch04b]
Otter [MW97]
Prover-9 [McC08]
SETHEO [LSBB92, MIL$^+$97]
Snark [E.S08]
Waldmeister [LH02, GHLS03]

Formulae

- Formulas are recursively defined:
  - Literals (elementary statements) are formulae
  - If F is a formula, $\forall X : F$ and $\exists X : F$ are formulae
  - Boolean combinations of formulae are formulae
  - Parentheses are applied wherever necessary
- Example:
  - $\forall X : (\forall Y : ((odd(X) \land odd(Y)) \rightarrow X \not\simeq add(Y, 1)))$

Clauses

- Clauses are multisets written and interpreted as disjunctions of literals
  - All variables implicitly universally quantified
- Example:

$$X \not\simeq add(Y,1) \vee odd(X) \vee odd(Y)$$

- Alternative views: Implicational

$$X \simeq add(Y,1) \implies (odd(X) \vee odd(Y))$$

or

$$(X \simeq add(Y,1) \wedge \neg odd(X)) \implies odd(Y))$$

or

$$(X \simeq add(Y,1) \wedge \neg odd(Y)) \implies odd(X))$$

or (weirdly)

$$(\neg odd(Y) \wedge \neg odd(X)) \implies X \not\simeq add(Y,1)$$

Literals

- $X \not\simeq add(Y, 1) \vee odd(X) \vee odd(Y)$

- – $X \not\simeq add(Y, 1)$ is a negative equational literal
  – $odd(X)$ and $odd(X)$ are positive non-equational literals

- Conventions:

  – $s \not\simeq t$ is a more convenient way of writing $\neg s \simeq t$
  – We write $s \dot{\simeq} t$ to denote an equational literal that may be either positive or negative
  – $s \simeq t$ is a more convenient way of writing $\simeq (s, t)$

Literals

- $X \not\simeq add(Y, 1) \vee odd(X) \vee odd(Y)$

- – $X \not\simeq add(Y, 1)$ is a negative equational literal
  – $odd(X)$ and $odd(X)$ are positive non-equational literals

- Convention:

  – $s \not\simeq t$ is a more convenient way of writing $\neg s \simeq t$
  – We write $s \dot{\simeq} t$ to denote an equational literal that may be either positive or negative
  – Heresy: $s \simeq t$ is a more conventient way of writing $\simeq (s, t)$
  – Truth: $odd(X)$ is a more convenient way of writing $odd(X) \simeq \top$

Equational Encoding Snag

- Problem:
  - $\{X \simeq a), \neg p(a)\}$ is satisfiable
  - What about $\{X \simeq a), p(a) \not\simeq \top\}$?
- Solution:
  - Two sorts: Individuals and Bools
  - Variables range over individuals only
  - Predicate terms are sort Bool
- Implemented that way in E

Terms

- $X \not\simeq add(Y, 1) \vee odd(X) \vee odd(Y)$

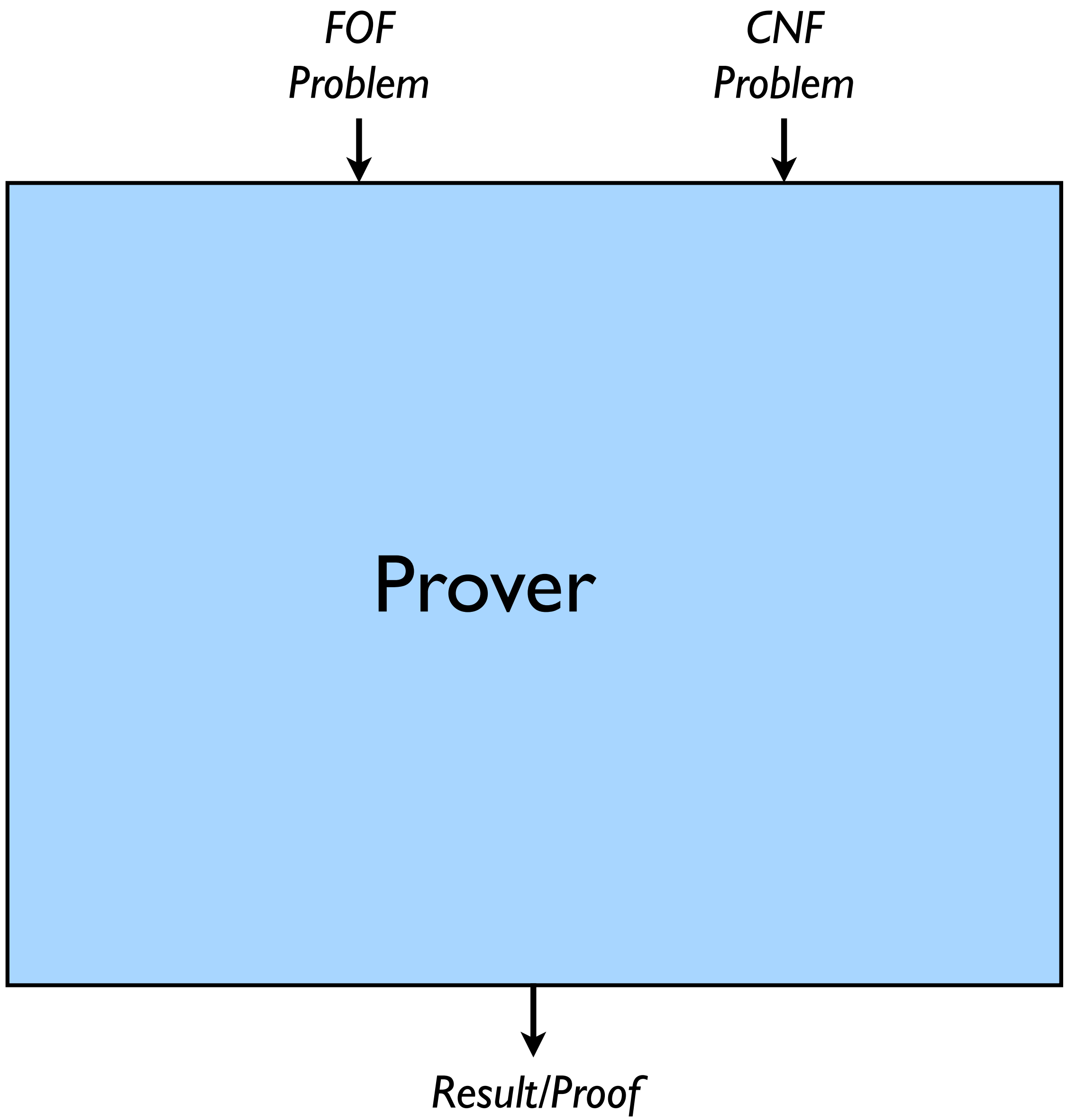
- – $X, add(Y, 1), 1$, and Y are terms
  – X and Y are variables
  – 1 is a constant term
  – $add(Y, 1)$ is a composite term with proper subterms 1 and Y

Concrete Syntax

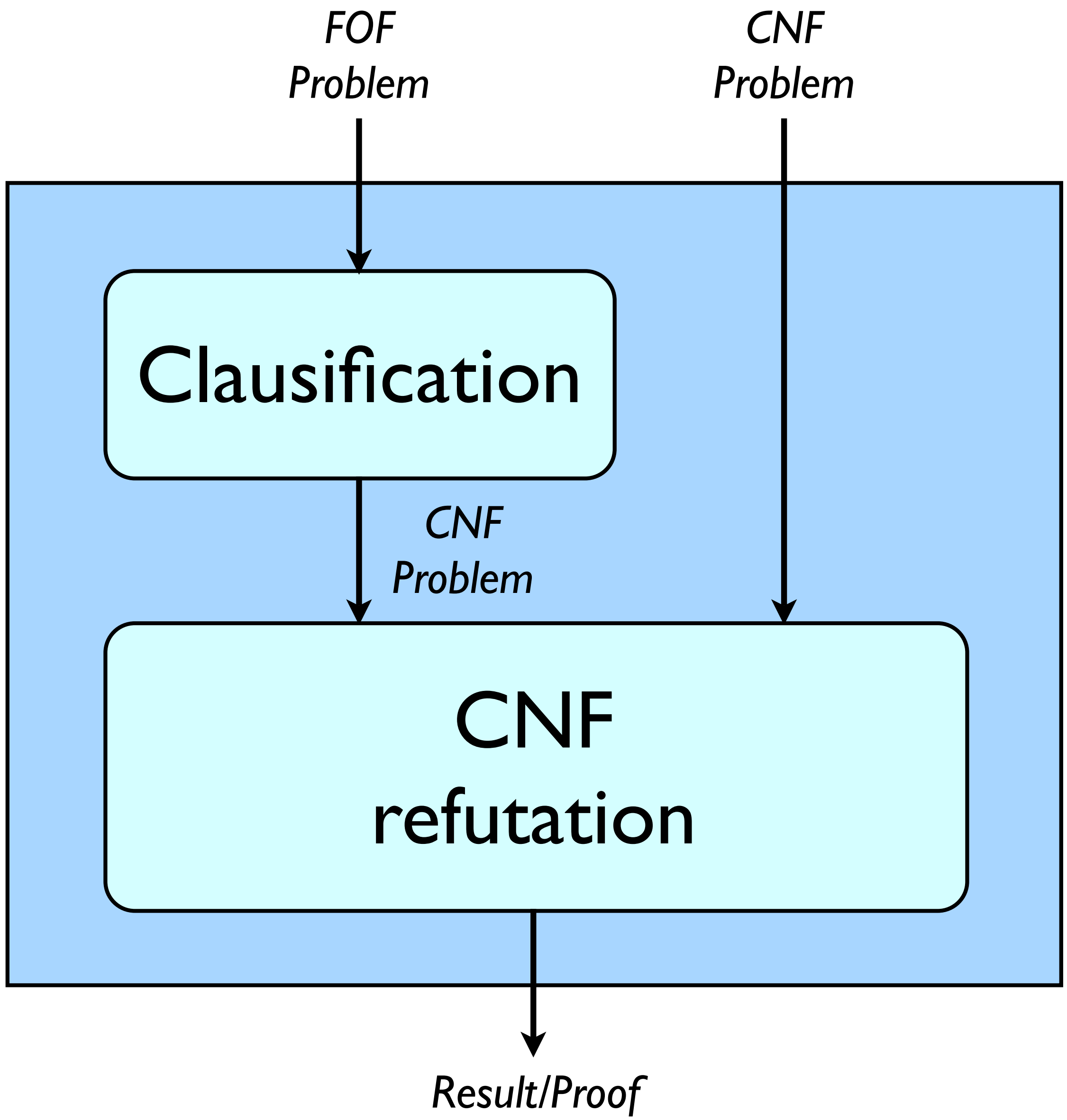
- Historically: Large variety of syntaxes
  - Prolog-inspired, e.g. LOP (SETHEO, E)
  - By committee, e.g. DFG-Syntax (SPASS)
  - LISP-inspired (SNARK)
  - Home-grown (Otter, Prover-9)
  - TPTP-1/2 syntax (with TPTP2X converter)

- Recently: Quasi-standardizaton on TPTP-3 syntax [SSCG06, Sut09]
  - Annotated clauses/formulas
  - Can represent problems and proofs
  - Support in Vampire, SPASS, E, E-SETHEO, iProver,

A First-Order Prover - Bird's Eye Perspective



A First-Order Prover - Bird's X-Ray Perspective



Clausification

$A \stackrel{?}{\models} H$ $\quad\Longrightarrow\quad$ $\boxed{\textbf{Classifier}}$ $\quad\Longrightarrow\quad$ $\{C_1, C_2, \ldots, C_3\}$

...such that

$\{C_1, C_2, \ldots, C_3\}$ is unsatisfiable

iff

$A \models H$ holds

Clausification

$A \overset{?}{\models} H \quad \Longrightarrow \quad \boxed{\textbf{Magic}} \quad \Longrightarrow \quad \{C_1, C_2, \ldots, C_3\}$

...such that

$\{C_1, C_2, \ldots, C_3\}$ is unsatisfiable

iff

$A \models H$ holds

Clausification

$A \overset{?}{\models} H \quad \Longrightarrow \quad \boxed{\textbf{Magic}} \quad \Longrightarrow \quad \{C_1, C_2, \ldots, C_3\}$

White Magic: Standard conjunctive normal form with Skolemization [Lov78] [NW01] (read once)

- Straightforward
- CNF can explode (and does, occasionally)

Black Magic: Miniscoping and definitions [NW01] (Read twice)

- Smaller CNF, exponential growths can be controlled
- Better (smaller) terms, less arity in Skolem functions
- Implemented in **E**

Forbidden Magic: Advanced Skolemization [NW01](Read five times)

- Implemented in FLOTTER
- Theoretically superior, but advantage in practice unclear

Why FOF at all?

```
% All aircraft are either in lower or in upper airspace
fof(low_up_is_exhaustive, axiom,
    (![X]:(lowairspace(X)|uppairspace(X)))).

fof(filter_equiv, conjecture, (
% Naive version: Display aircraft in the Abu Dhabi Approach area in
% lower airspace, display aircraft in the Dubai Approach area in lower
% airspace, display all aircraft in upper airspace, except for
% aircraft in military training region if they are actual military
% aircraft.
    (![X]:(((a_d_app(X) & lowairspace(X))|(dub_app(X) & lowairspace(X))
    |uppairspace(X))&
    (~milregion(X)|~military(X))))
    <=>
% Optimized version: Display all aircraft in either Approach, display
% aircraft in upper airspace, except military aircraft in the military
% training region
    (![X]:((uppairspace(X) | dub_app(X) | a_d_app(X)) &
    (~military(X) | ~milregion(X)))))).
```

Why FOF at all?

```
cnf(i_0_1,plain,(lowairspace(X1)|uppairspace(X1))).
cnf(i_0_12,negated_conjecture,(milregion(esk1_0)|milregion(esk2_0)|~uppairspace(esk1_0)|~uppairspace(esk2_0))).
cnf(i_0_8,negated_conjecture,(milregion(esk1_0)|milregion(esk2_0)|~uppairspace(esk1_0)|~a_d_app(esk2_0))).
cnf(i_0_10,negated_conjecture,(milregion(esk1_0)|milregion(esk2_0)|~uppairspace(esk1_0)|~dub_app(esk2_0))).
cnf(i_0_13,negated_conjecture,(milregion(esk1_0)|military(esk2_0)|~uppairspace(esk1_0)|~uppairspace(esk2_0))).
cnf(i_0_9,negated_conjecture,(milregion(esk1_0)|military(esk2_0)|~uppairspace(esk1_0)|~a_d_app(esk2_0))).
cnf(i_0_11,negated_conjecture,(milregion(esk1_0)|military(esk2_0)|~uppairspace(esk1_0)|~dub_app(esk2_0))).
cnf(i_0_6,negated_conjecture,(milregion(esk2_0)|military(esk1_0)|~uppairspace(esk1_0)|~uppairspace(esk2_0))).
cnf(i_0_2,negated_conjecture,(milregion(esk2_0)|military(esk1_0)|~uppairspace(esk1_0)|~a_d_app(esk2_0))).
cnf(i_0_4,negated_conjecture,(milregion(esk2_0)|military(esk1_0)|~uppairspace(esk1_0)|~dub_app(esk2_0))).
cnf(i_0_7,negated_conjecture,(military(esk1_0)|military(esk2_0)|~uppairspace(esk1_0)|~uppairspace(esk2_0))).
cnf(i_0_3,negated_conjecture,(military(esk1_0)|military(esk2_0)|~uppairspace(esk1_0)|~a_d_app(esk2_0))).
cnf(i_0_5,negated_conjecture,(military(esk1_0)|military(esk2_0)|~uppairspace(esk1_0)|~dub_app(esk2_0))).
cnf(i_0_36,negated_conjecture,(milregion(esk1_0)|milregion(esk2_0)|~lowairspace(esk1_0)|~uppairspace(esk2_0)|
    ~a_d_app(esk1_0))).
cnf(i_0_24,negated_conjecture,(milregion(esk1_0)|milregion(esk2_0)|~lowairspace(esk1_0)|~uppairspace(esk2_0)|
    ~dub_app(esk1_0))).
cnf(i_0_32,negated_conjecture,(milregion(esk1_0)|milregion(esk2_0)|~lowairspace(esk1_0)|~a_d_app(esk1_0)|
    ~a_d_app(esk2_0))).
cnf(i_0_34,negated_conjecture,(milregion(esk1_0)|milregion(esk2_0)|~lowairspace(esk1_0)|~a_d_app(esk1_0)|
    ~dub_app(esk2_0))).
cnf(i_0_20,negated_conjecture,(milregion(esk1_0)|milregion(esk2_0)|~lowairspace(esk1_0)|~a_d_app(esk2_0)|
    ~dub_app(esk1_0))).
cnf(i_0_22,negated_conjecture,(milregion(esk1_0)|milregion(esk2_0)|~lowairspace(esk1_0)|~dub_app(esk1_0)|
    ~dub_app(esk2_0))).
cnf(i_0_37,negated_conjecture,(milregion(esk1_0)|military(esk2_0)|~lowairspace(esk1_0)|~uppairspace(esk2_0)|
    ~a_d_app(esk1_0))).
cnf(i_0_25,negated_conjecture,(milregion(esk1_0)|military(esk2_0)|~lowairspace(esk1_0)|~uppairspace(esk2_0)|
    ~dub_app(esk1_0))).
cnf(i_0_33,negated_conjecture,(milregion(esk1_0)|military(esk2_0)|~lowairspace(esk1_0)|~a_d_app(esk1_0)|
    ~a_d_app(esk2_0))).
```

```
cnf(i_0_35,negated_conjecture,(milregion(esk1_0)|military(esk2_0)|~lowairspace(esk1_0)|~a_d_app(esk1_0)|
    ~dub_app(esk2_0))).
cnf(i_0_21,negated_conjecture,(milregion(esk1_0)|military(esk2_0)|~lowairspace(esk1_0)|~a_d_app(esk2_0)|
    ~dub_app(esk1_0))).
cnf(i_0_23,negated_conjecture,(milregion(esk1_0)|military(esk2_0)|~lowairspace(esk1_0)|~dub_app(esk1_0)|
    ~dub_app(esk2_0))).
cnf(i_0_30,negated_conjecture,(milregion(esk2_0)|military(esk1_0)|~lowairspace(esk1_0)|~uppairspace(esk2_0)|
    ~a_d_app(esk1_0))).
cnf(i_0_18,negated_conjecture,(milregion(esk2_0)|military(esk1_0)|~lowairspace(esk1_0)|~uppairspace(esk2_0)|
    ~dub_app(esk1_0))).
cnf(i_0_26,negated_conjecture,(milregion(esk2_0)|military(esk1_0)|~lowairspace(esk1_0)|~a_d_app(esk1_0)|
    ~a_d_app(esk2_0))).
cnf(i_0_28,negated_conjecture,(milregion(esk2_0)|military(esk1_0)|~lowairspace(esk1_0)|~a_d_app(esk1_0)|
    ~dub_app(esk2_0))).
cnf(i_0_14,negated_conjecture,(milregion(esk2_0)|military(esk1_0)|~lowairspace(esk1_0)|~a_d_app(esk2_0)|
    ~dub_app(esk1_0))).
cnf(i_0_16,negated_conjecture,(milregion(esk2_0)|military(esk1_0)|~lowairspace(esk1_0)|~dub_app(esk1_0)|
    ~dub_app(esk2_0))).
cnf(i_0_31,negated_conjecture,(military(esk1_0)|military(esk2_0)|~lowairspace(esk1_0)|~uppairspace(esk2_0)|
    ~a_d_app(esk1_0))).
cnf(i_0_19,negated_conjecture,(military(esk1_0)|military(esk2_0)|~lowairspace(esk1_0)|~uppairspace(esk2_0)|
    ~dub_app(esk1_0))).
cnf(i_0_27,negated_conjecture,(military(esk1_0)|military(esk2_0)|~lowairspace(esk1_0)|~a_d_app(esk1_0)|
    ~a_d_app(esk2_0))).
cnf(i_0_29,negated_conjecture,(military(esk1_0)|military(esk2_0)|~lowairspace(esk1_0)|~a_d_app(esk1_0)|
    ~dub_app(esk2_0))).
cnf(i_0_15,negated_conjecture,(military(esk1_0)|military(esk2_0)|~lowairspace(esk1_0)|~a_d_app(esk2_0)|
    ~dub_app(esk1_0))).
cnf(i_0_17,negated_conjecture,(military(esk1_0)|military(esk2_0)|~lowairspace(esk1_0)|~dub_app(esk1_0)|
    ~dub_app(esk2_0))).
cnf(i_0_44,negated_conjecture,(lowairspace(X2)|uppairspace(X2)|uppairspace(X1)|a_d_app(X1)|
    dub_app(X1))).
cnf(i_0_39,negated_conjecture,(lowairspace(X2)|uppairspace(X2)|~milregion(X1)|~military(X1))).
cnf(i_0_46,negated_conjecture,(lowairspace(X2)|uppairspace(X2)|uppairspace(X1)|a_d_app(X2)|a_d_app(X1)|
```

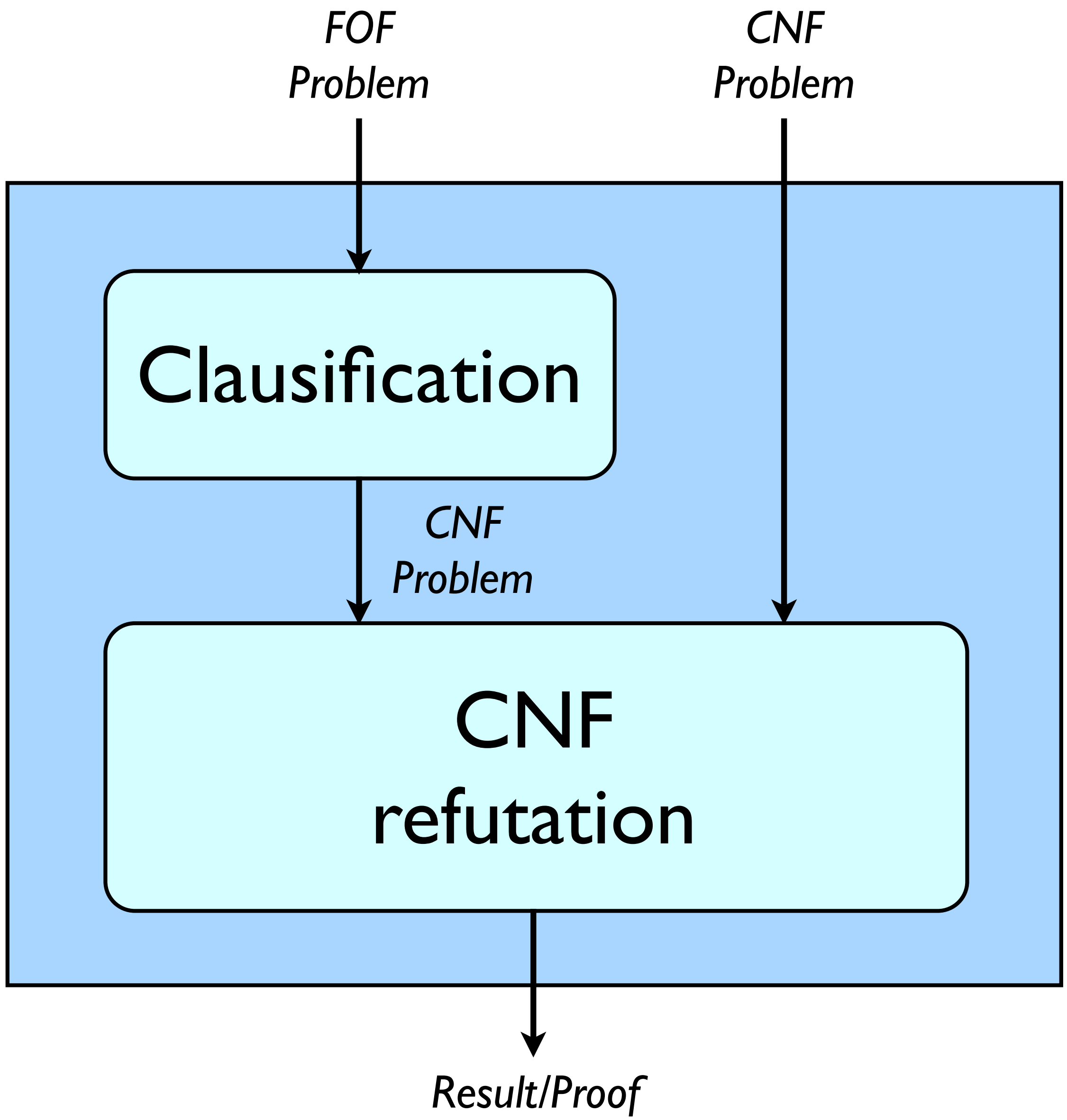
```
    dub_app(X1))).
cnf(i_0_45,negated_conjecture,(lowairspace(X2)|uppairspace(X2)|uppairspace(X1)|a_d_app(X1)|
    dub_app(X2)|dub_app(X1))).
cnf(i_0_47,negated_conjecture,(uppairspace(X2)|uppairspace(X1)|a_d_app(X2)|a_d_app(X1)|dub_app(X2)|
    dub_app(X1))).
cnf(i_0_41,negated_conjecture,(lowairspace(X2)|uppairspace(X2)|a_d_app(X2)|~milregion(X1)|~military(X1))).
cnf(i_0_40,negated_conjecture,(lowairspace(X2)|uppairspace(X2)|dub_app(X2)|~milregion(X1)|~military(X1))).
cnf(i_0_42,negated_conjecture,(uppairspace(X2)|a_d_app(X2)|dub_app(X2)|~milregion(X1)|~military(X1))).
cnf(i_0_43,negated_conjecture,(uppairspace(X1)|a_d_app(X1)|dub_app(X1)|~milregion(X2)|~military(X2))).
cnf(i_0_38,negated_conjecture,(~milregion(X2)|~milregion(X1)|~military(X2)|~military(X1))).
```

Lazy Developer's Clausification

$$A \overset{?}{\models} H \quad \Longrightarrow \quad \boxed{\begin{array}{c}\textbf{E}\\ \textbf{FLOTTER}\\ \textbf{Vampire}\end{array}} \quad \Longrightarrow \quad \{C_1, C_2, \ldots, C_3\}$$

- iProver (uses E, Vampire)
- E-SETHEO (uses E, FLOTTER)
- Fampire (uses FLOTTER)

A First-Order Prover - Bird's X-Ray Perspective



CNF Saturation

- Basic idea: Proof state is a set of clauses S
  - Goal: Show unsatisfiability of S
  - Method: Derive empty clause via deduction
  - Problem: Proof state explosion
- Generation: Deduce new clauses
  - Logical core of the calculus
  - Necessary for completeness
  - Lead to explosion is proof state size
    $\Longrightarrow$ Restrict as much as possible
- Simplification: Remove or simplify clauses from S
  - Critical for acceptable performance
  - Burns most CPU cycles
    $\Longrightarrow$ Efficient implementation necessary

Rewriting

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

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

- Conditions:
  - $u|_p = \sigma(s)$
  - $\sigma(s) > \sigma(t)$
  - Some restrictions on rewriting $>$-maximal terms in a clause apply

- Note: If $s > t$, we call $s \simeq t$ a rewrite rule
  - Implies $\sigma(s) > \sigma(t)$, no ordering check necessary

Paramodulation/Superposition

- Superposition: "Lazy conditional speculative rewriting"
  - Conditional: Uses non-unit clauses
    - One positive literal is seen as potential rewrite rule
    - All other literals are seen as (positive and negative) conditions
  - Lazy: Conditions are not solved, but appended to result
  - Speculative:
    - Replaces potentially larger terms
    - Applies to instances of clauses (generated by unification)
    - Original clauses remain (generating inference)

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

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

Subsumption

- Idea: Only keep the most general clauses
  - If one clause is subsumed by another, discard it

$$\frac{C \qquad \sigma(C) \vee R}{C}$$

- Examples:
  - $p(X)$ subsumes $p(a) \vee q(f(X), a)$ ($\sigma = \{X \leftarrow a\}$)
  - $p(X) \vee p(Y)$ does not multi-set-subsume $p(a) \vee q(f(X), a)$
  - $q(X, Y) \vee q(X, a)$ subsumes $q(a, a) \vee q(a, b)$

- Subsumption is hard (NP-complete)
  - $n!$ permutations in non-equational clause with n literals
  - $n!2^n$ permutations in equational clause with n literals

Term Orderings

- Superposition is instantiated with a ground-completable simplification ordering $>$ on terms
  - $>$ is Noetherian
  - $>$ is compatible with term structure: $t_1 > t_2$ implies $s[t_1]_p > s[t_2]_p$
  - $>$ is compatible with substitutions: $t_1 > t_2$ implies $\sigma(t_1) > \sigma(t_2)$
  - $>$ has the subterm-property: $s > s|_p$
  - In practice: LPO, KBO, RPO
- Ordering evaluation is one of the major costs in superposition-based theorem proving
- Efficient implementation of orderings: [Löc06, L0̈6]

Generalized Redundancy Elimination

- A clause is redundant in S, if all its ground instances are implied by $>$ smaller ground instances of other clauses in S
  - May require addition of smaller implied clauses!
- Examples:
  - Rewriting (rewritten clause added!)
  - Tautology deletion (implied by empty clause)
  - Redundant literal elimination: $l \vee l \vee R$ replaced by $l \vee R$
  - False literal elimination: $s \not\simeq s \vee R$ replaced by R
- Literature:
  - Theoretical results: [BG94, BG98, NR01]
  - Some important refinements used in E: [Sch02, Sch04b, RV01, Sch09]

The Basic Given-Clause Algorithm

- Completeness requires consideration of all possible persistent clause combinations for generating inferences
  - For superposition: All 2-clause combinations
  - Other inferences: Typically a single clause
- Given-clause algorithm replaces complex bookkeeping with simple invariant:
  - Proofstate $S = P \cup U$, P initially empty
  - All inferences between clauses in P have been performed
- The algorithm:

while $U \neq \{\}$
  $g =$ delete_best(U)
  if $g == \Box$
    SUCCESS, Proof found
  $P = P \cup \{g\}$
  $U = U \cup$generate(g, P)
SUCCESS, original U is satisfiable

DISCOUNT Loop

- Aim: Integrate simplification into given clause algorithm
- The algorithm (as implemented in E):

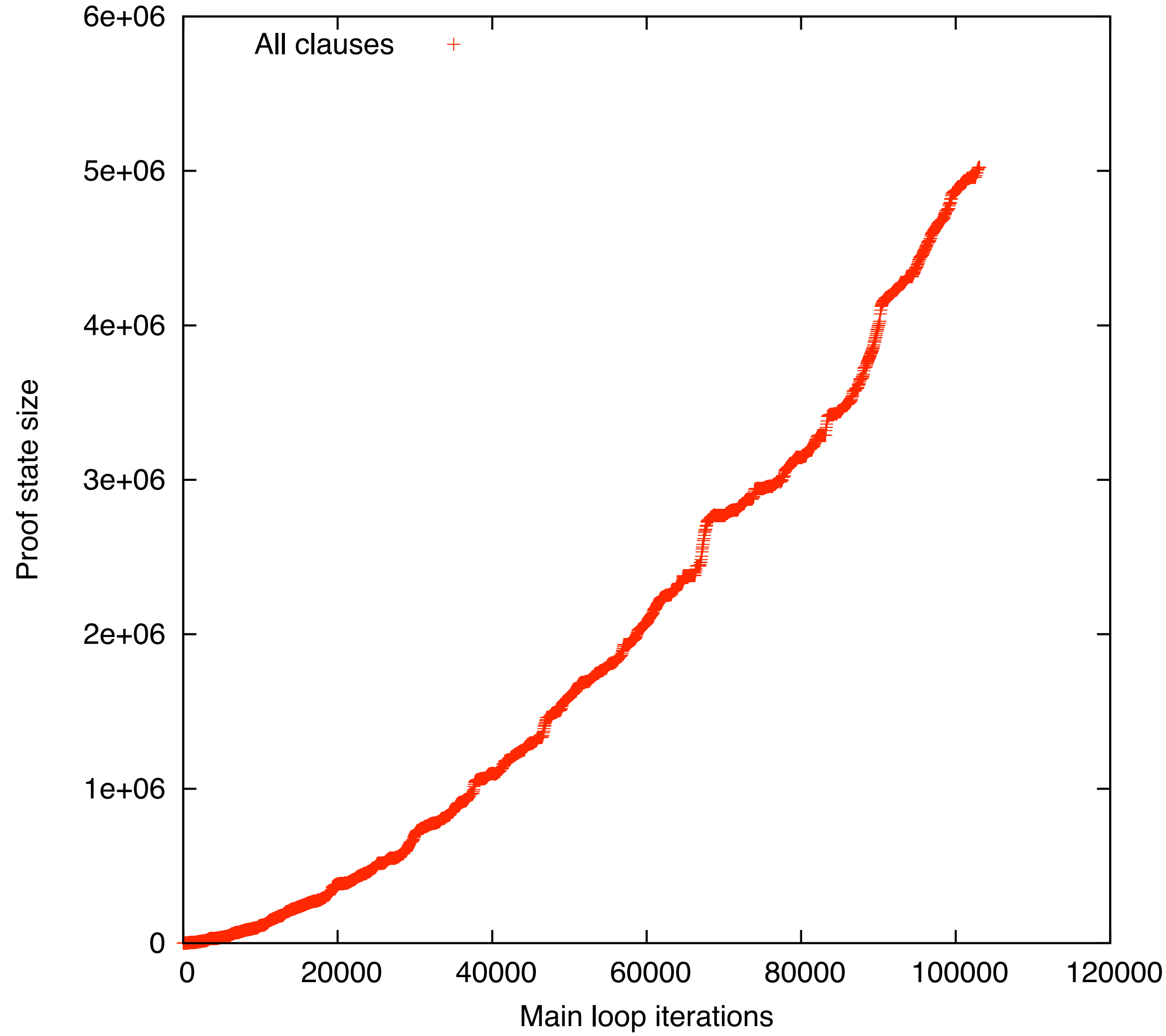
while $U \neq \{\}$
  $g =$ delete_best(U)
  $g =$ simplify(g,P)
  if $g == \square$
    SUCCESS, Proof found
  if g is not redundant w.r.t. P
    $T = \{c \in P | c$ redundant or simplifiable w.r.t. $g\}$
    $P = (P \backslash T) \cup \{g\}$
    $T = T \cup$generate(g, P)
    foreach $c \in T$
      $c =$cheap_simplify(c, P)
      if c is not trivial
        $U = U \cup \{c\}$
SUCCESS, original U is satisfiable

What is so hard about this?

What is so hard about this?

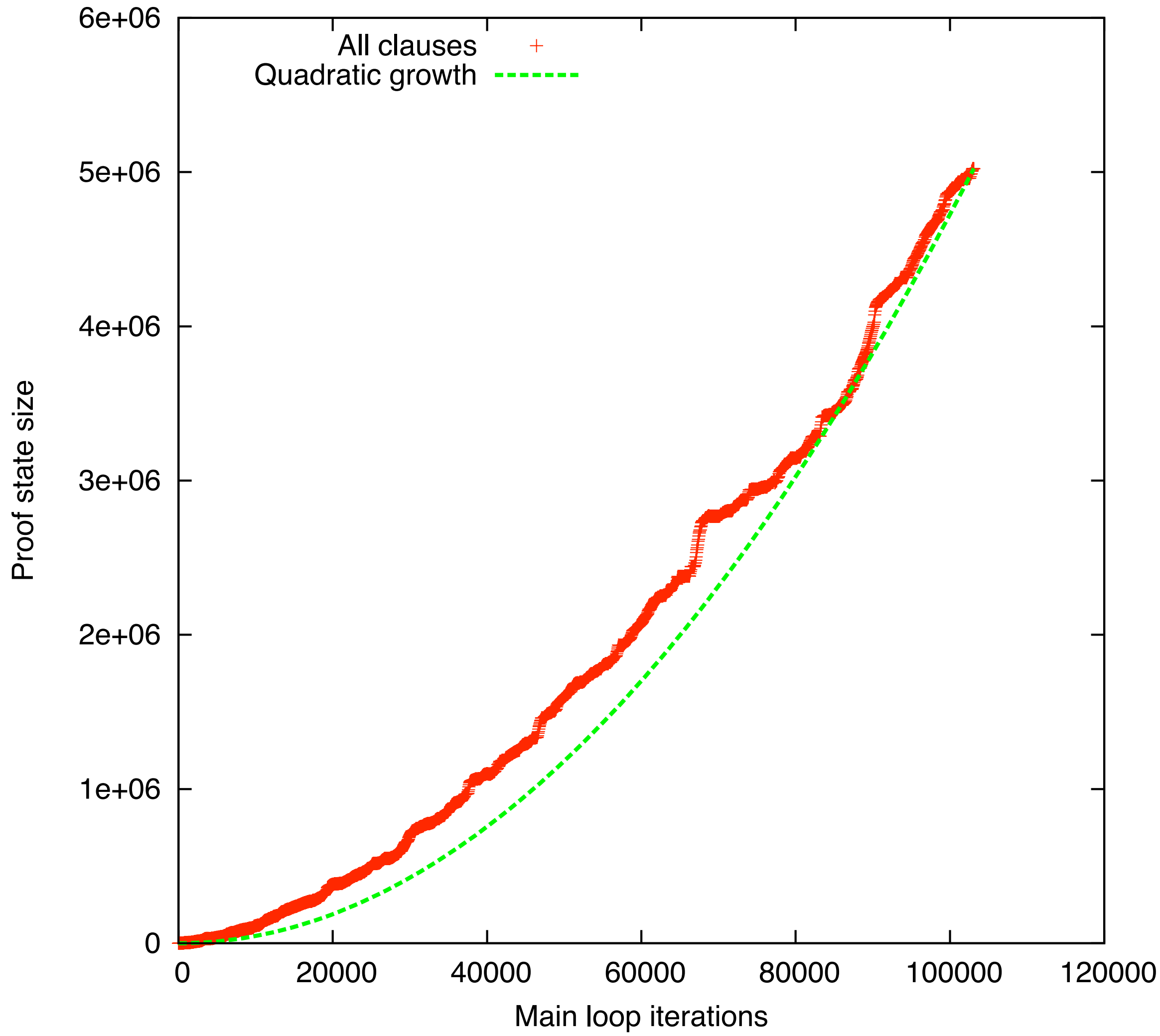
- Data from simple TPTP example NUM030-1+rm_eq_rstfp.lop (solved by E in 30 seconds on ancient Apple Powerbook):
  - Initial clauses: 160
  - Processed clauses: 16,322
  - Generated clauses: 204,436
  - Paramodulations: 204,395
  - Current number of processed clauses: 1,885
  - Current number of unprocessed clauses: 94,442
  - Number of terms: 5,628,929
- Hard problems run for days!
  - Millions of clauses generated (and stored)
  - Many millions of terms stored and rewritten
  - Each rewrite attempt must consider many ($>>$ 10000) rules
  - Subsumption must test many ($>>$ 10000) candidates for each subsumption attempt
  - Heuristic must find best clause out of millions

Proof State Development



Proof state behavior for ring theory example RNG043-2 (Default Mode)

Proof State Development



Proof state behavior for ring theory example RNG043-2 (Default Mode)

- Growth is roughly quadratic in the number of processed clauses

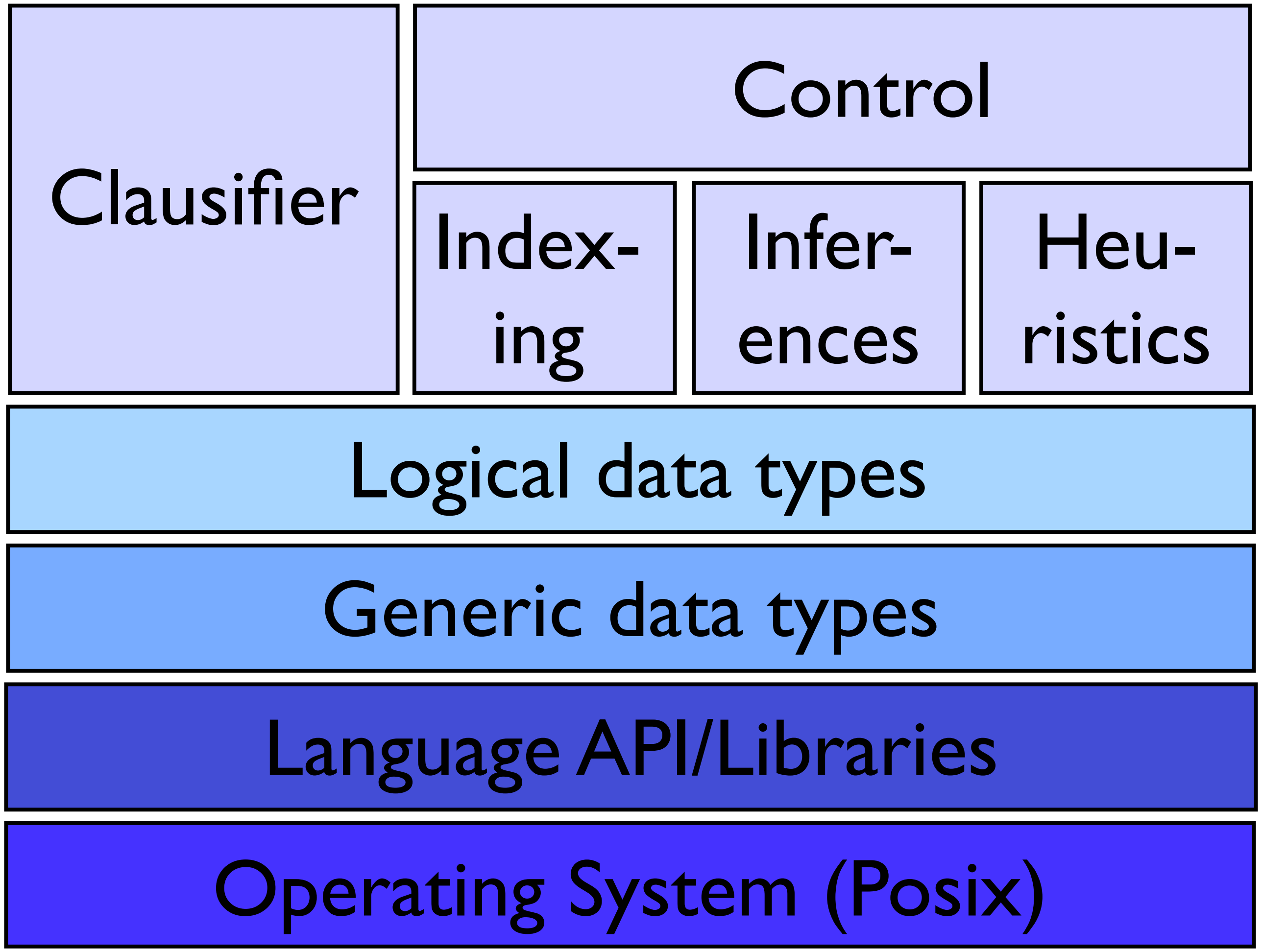
Literature on Proof Procedures

- New Waldmeister Loop: [GHLS03]
- Comparisons: [RV03]
- Best discussion of E Loop: [Sch02]

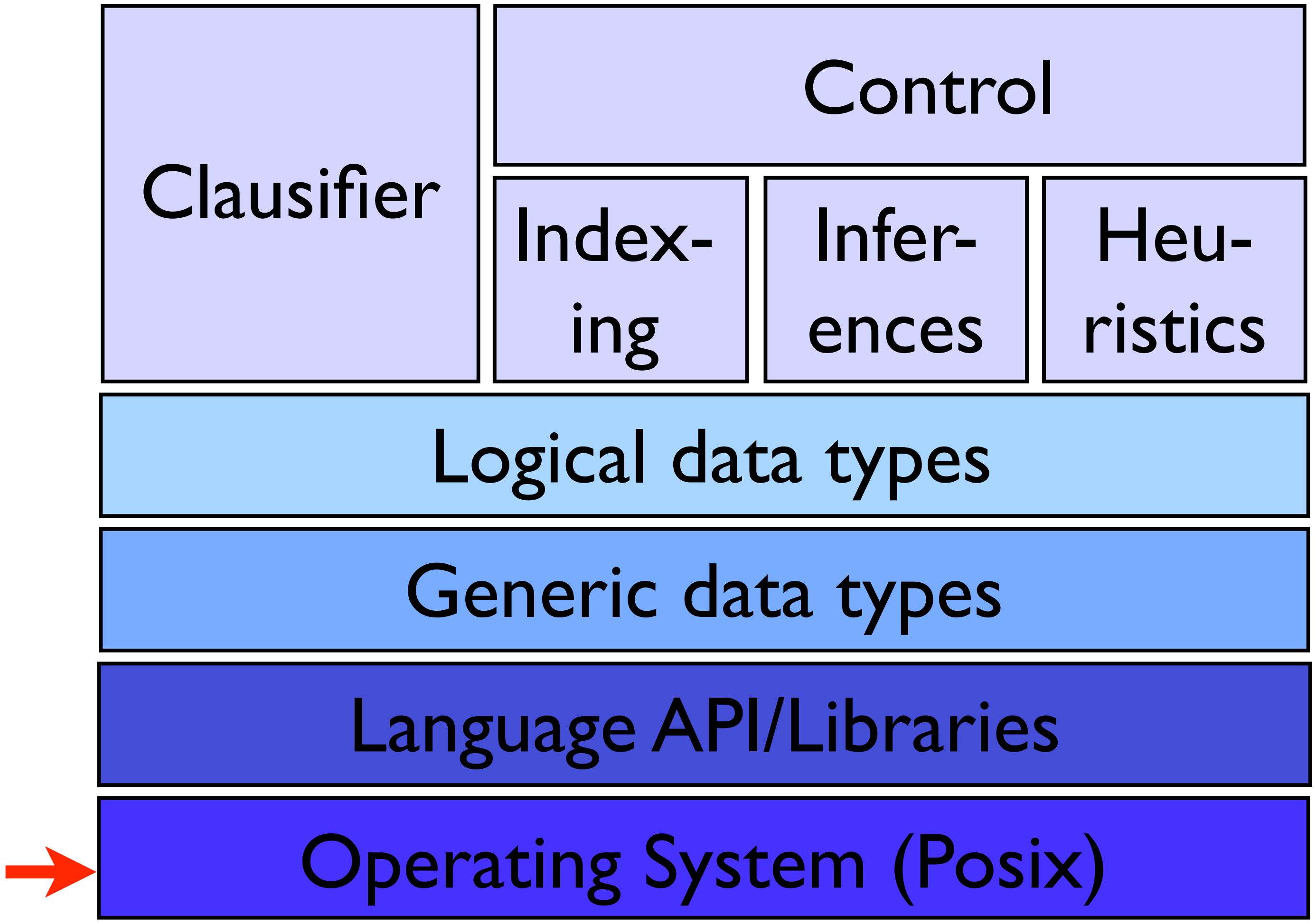
Exercise: Installing and Running E

- Goto `http://www.eprover.org`
- Find the download section
- Find and read the README
- Download the source tarball
- Following the README, build the system in a local user directory
- Run the prover on one of the included examples to demonstrates that it works.

Layered Architecture



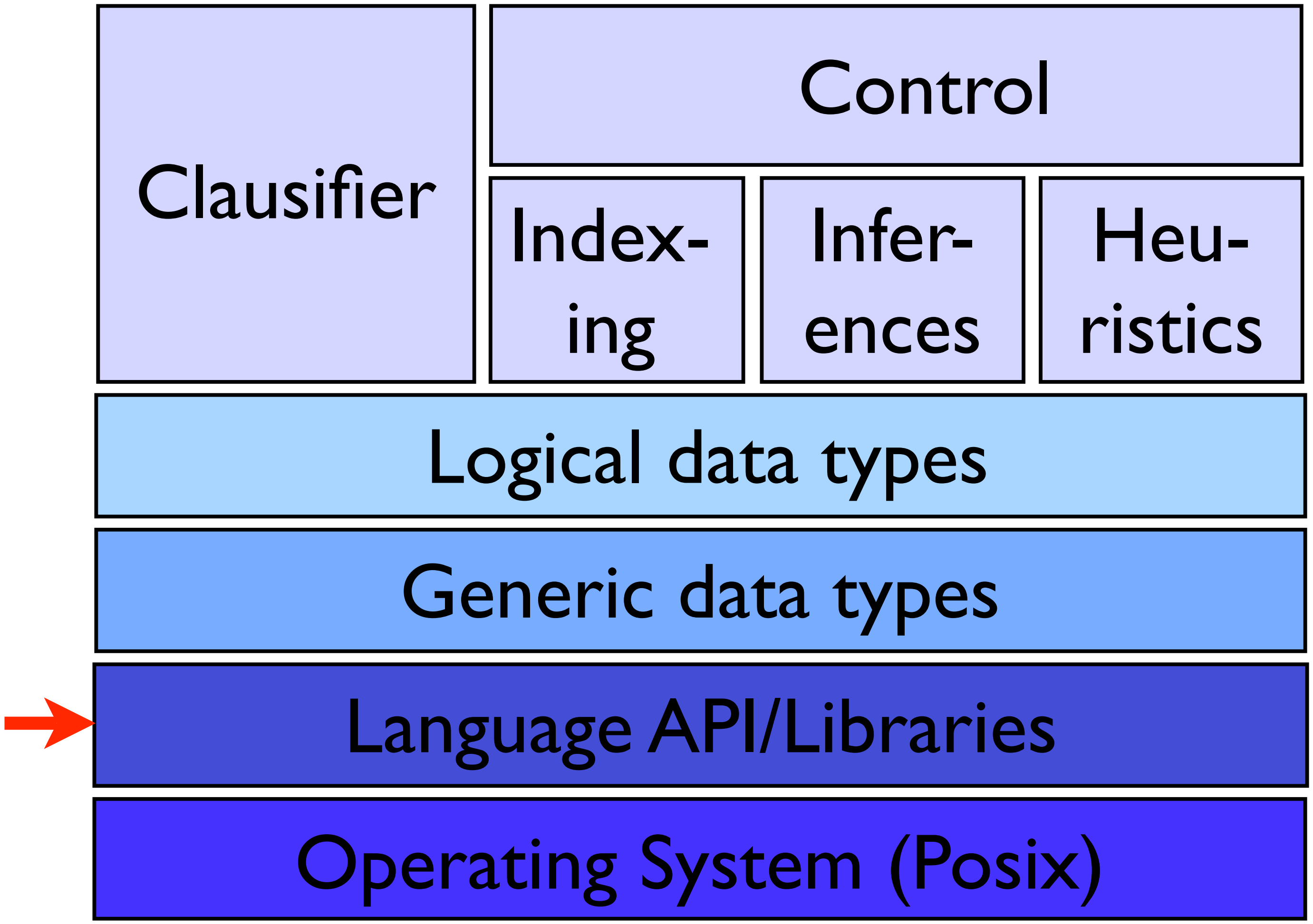
Layered Architecture



Operating System

- Pick a UNIX variant
  - Widely used
  - Free
  - Stable
  - Much better support for remote tests and automation
  - Everybody else uses it ;-)
- Aim for portability
  - Theorem provers have minimal requirements
  - Text input/output
  - POSIX is sufficient

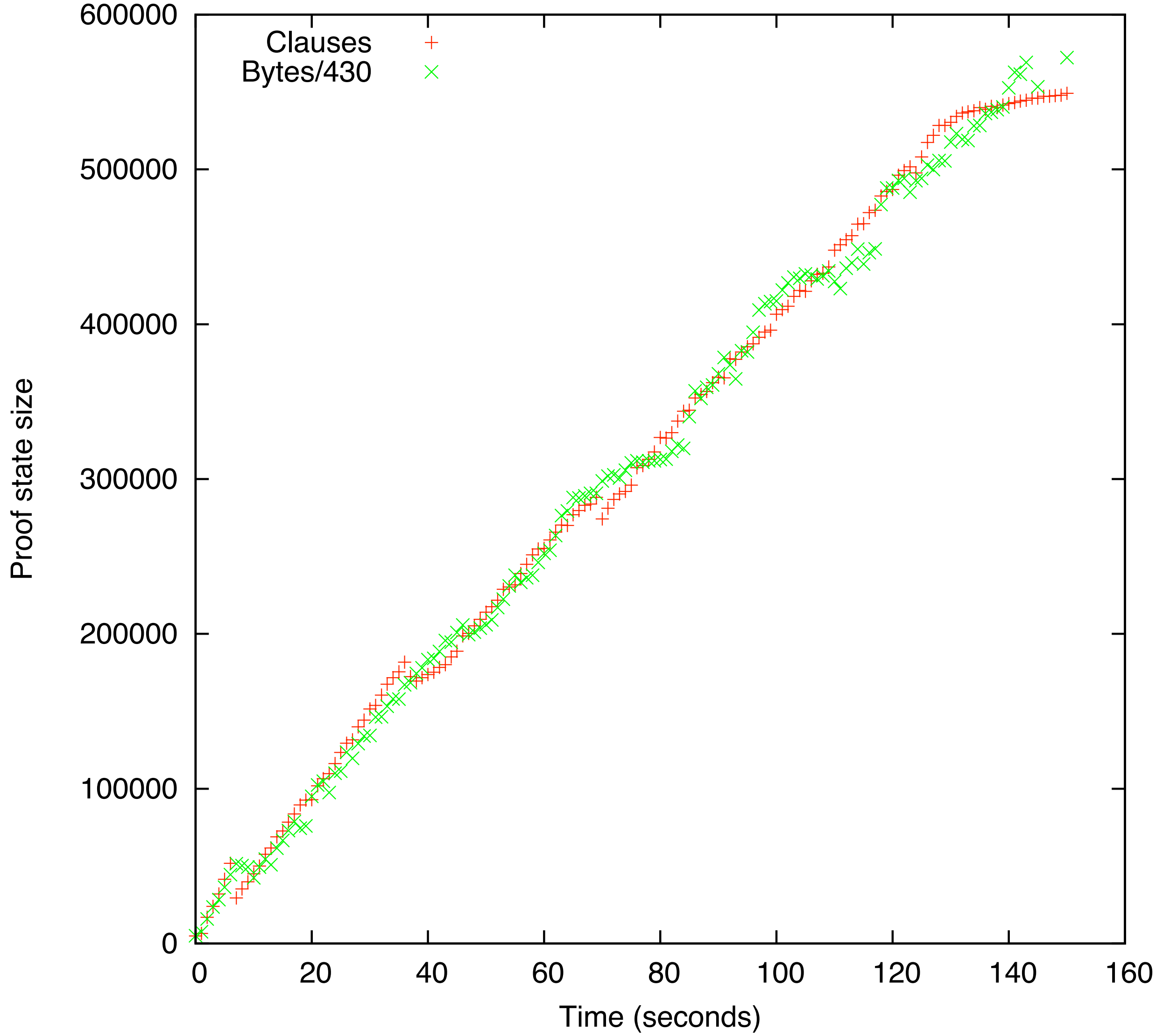
Layered Architecture



Language API/Libraries

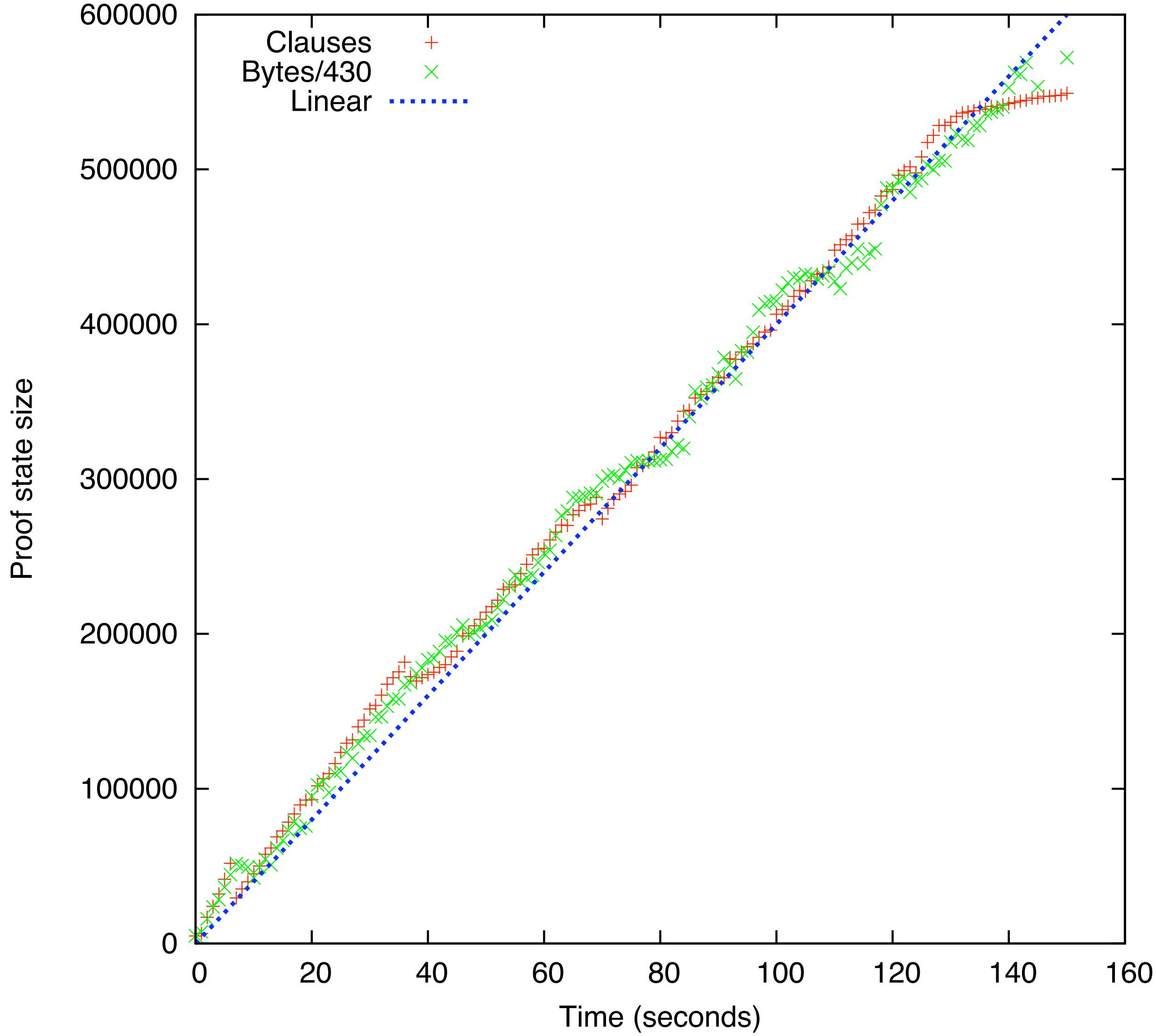
- Pick your language
- High-level/funtional or declarative languages come with rich datatypes and libraries
  - Can cover "Generic data types"
  - Can even cover 90% of "Logical data types"
- C offers nearly full control
  - Much better for low-level performance
  - . . . if you can make it happen!

Memory Consumption



- Proof state behavior for number theory example NUM030-1 (880 MHz SunFire)

Memory Consumption

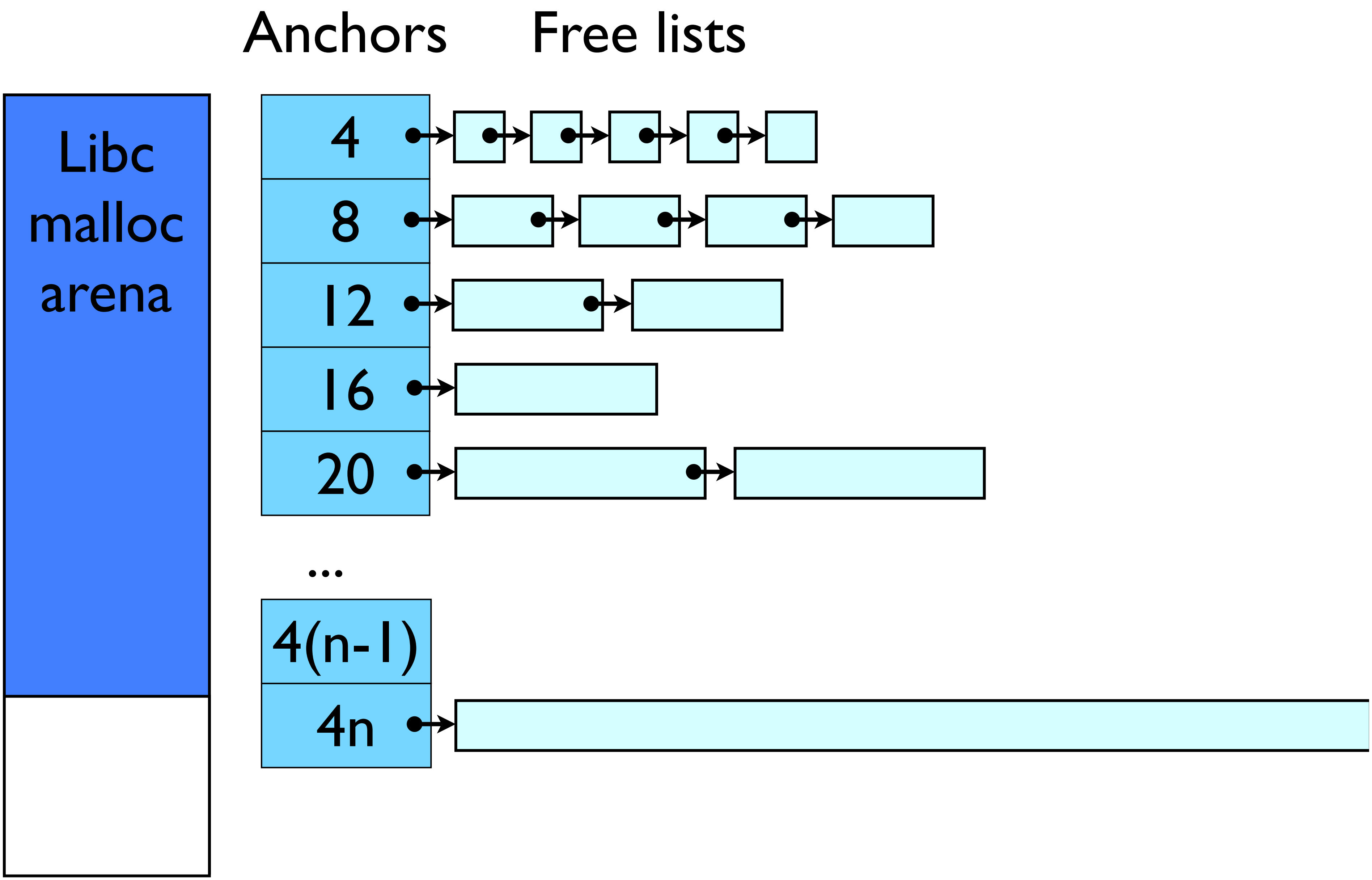


- Proof state behavior for number theory example NUM030-1 (880 MHz SunFire)

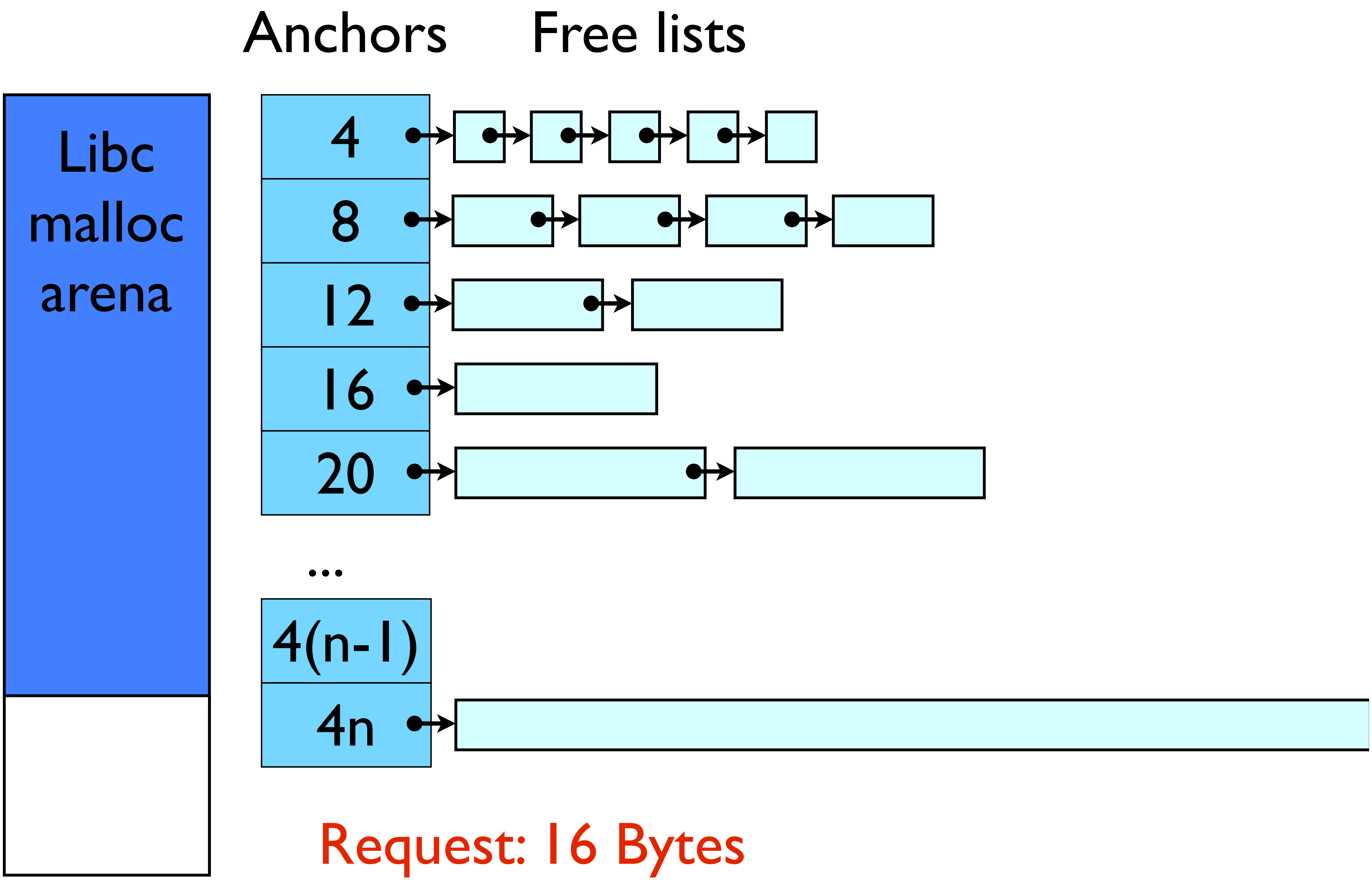
Memory Management

- Nearly all memory in a saturating prover is taken up by very few data types
  - Terms
  - Literals
  - Clauses
  - Clause evaluations
  - (Indices)
- These data types are frequently created and destroyed
  - Prime target for freelist based memory management
  - Backed directly by system malloc()
  - Allocating and chopping up large blocks does not pay off!
- Result:
  - Allocating temporary data structures is O(1)
  - Overhead is very small
  - Speedup 20%-50% depending on OS/processor/libC version

Memory Management illustrated

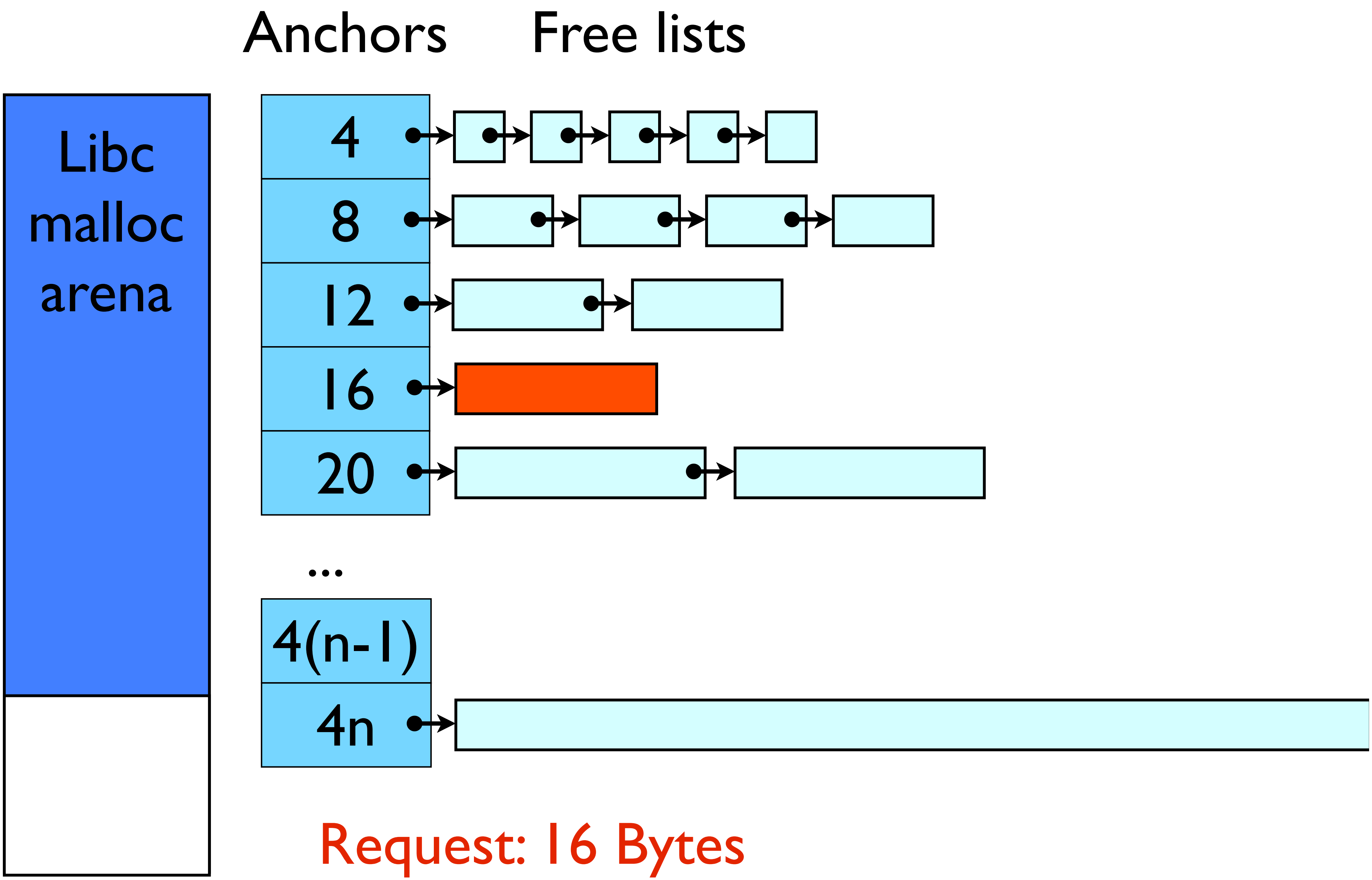


Memory Management illustrated



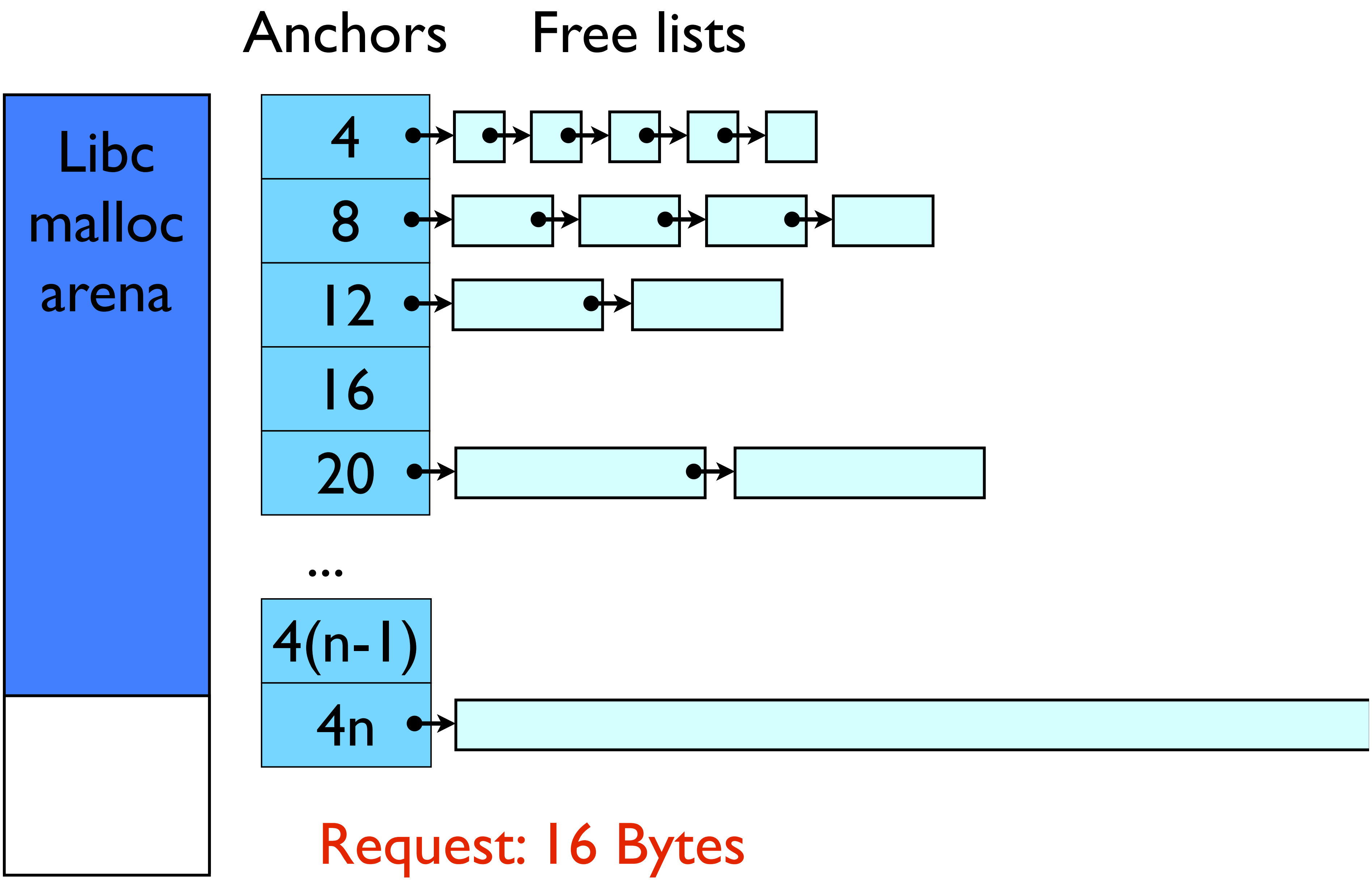
Request: 16 Bytes

Memory Management illustrated



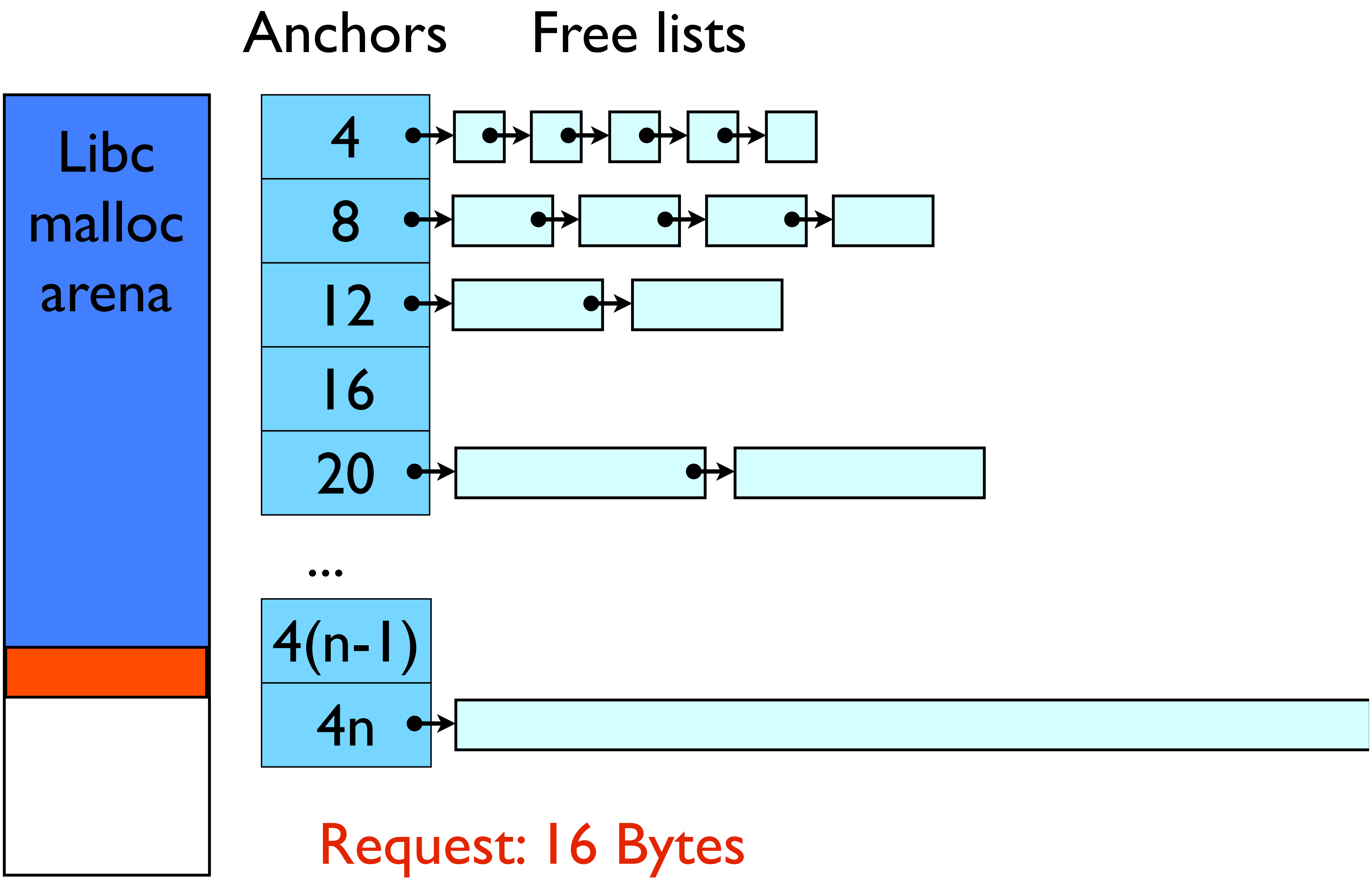
Request: 16 Bytes

Memory Management illustrated



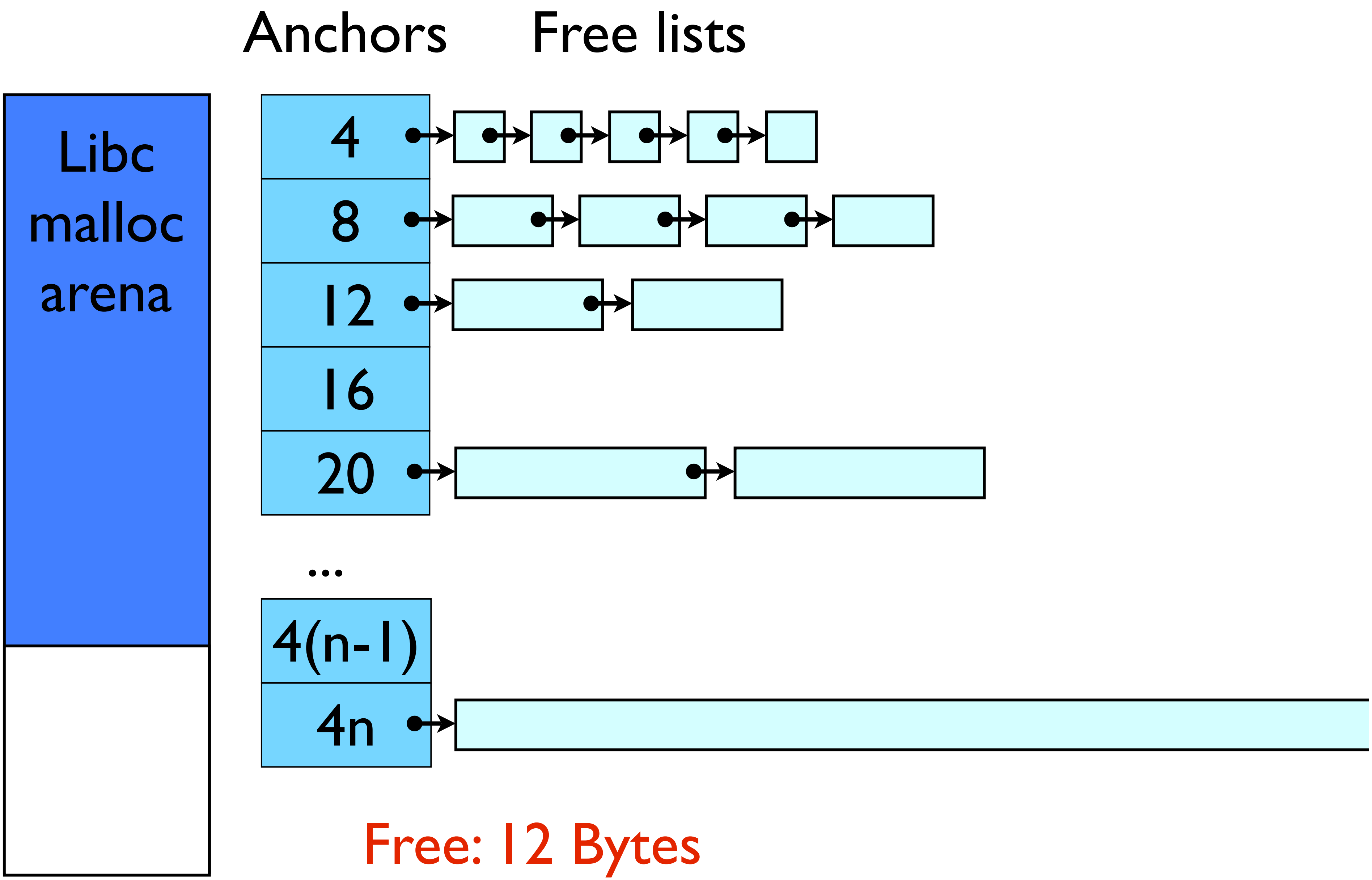
Request: 16 Bytes

Memory Management illustrated



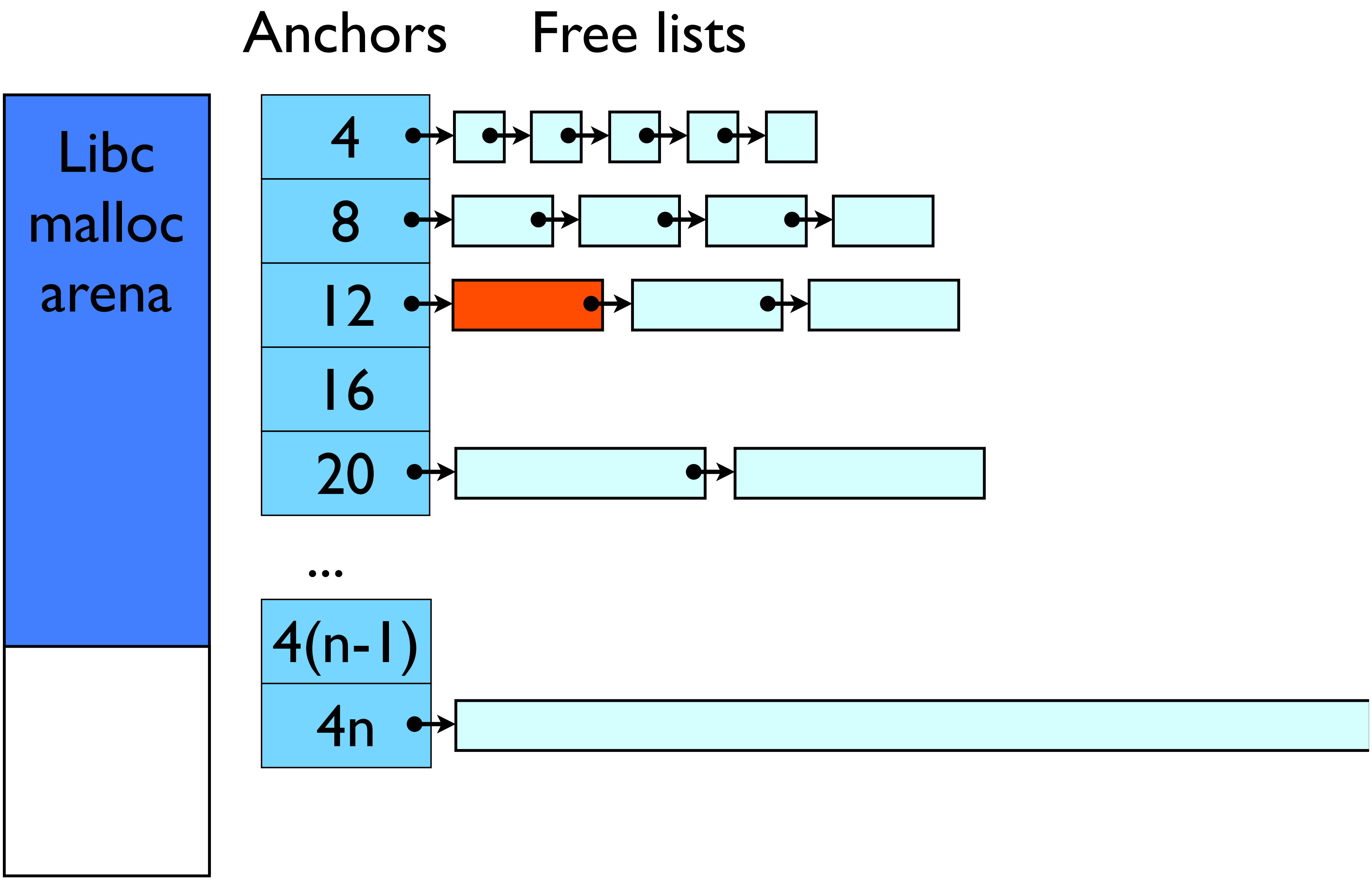
Request: 16 Bytes

Memory Management illustrated

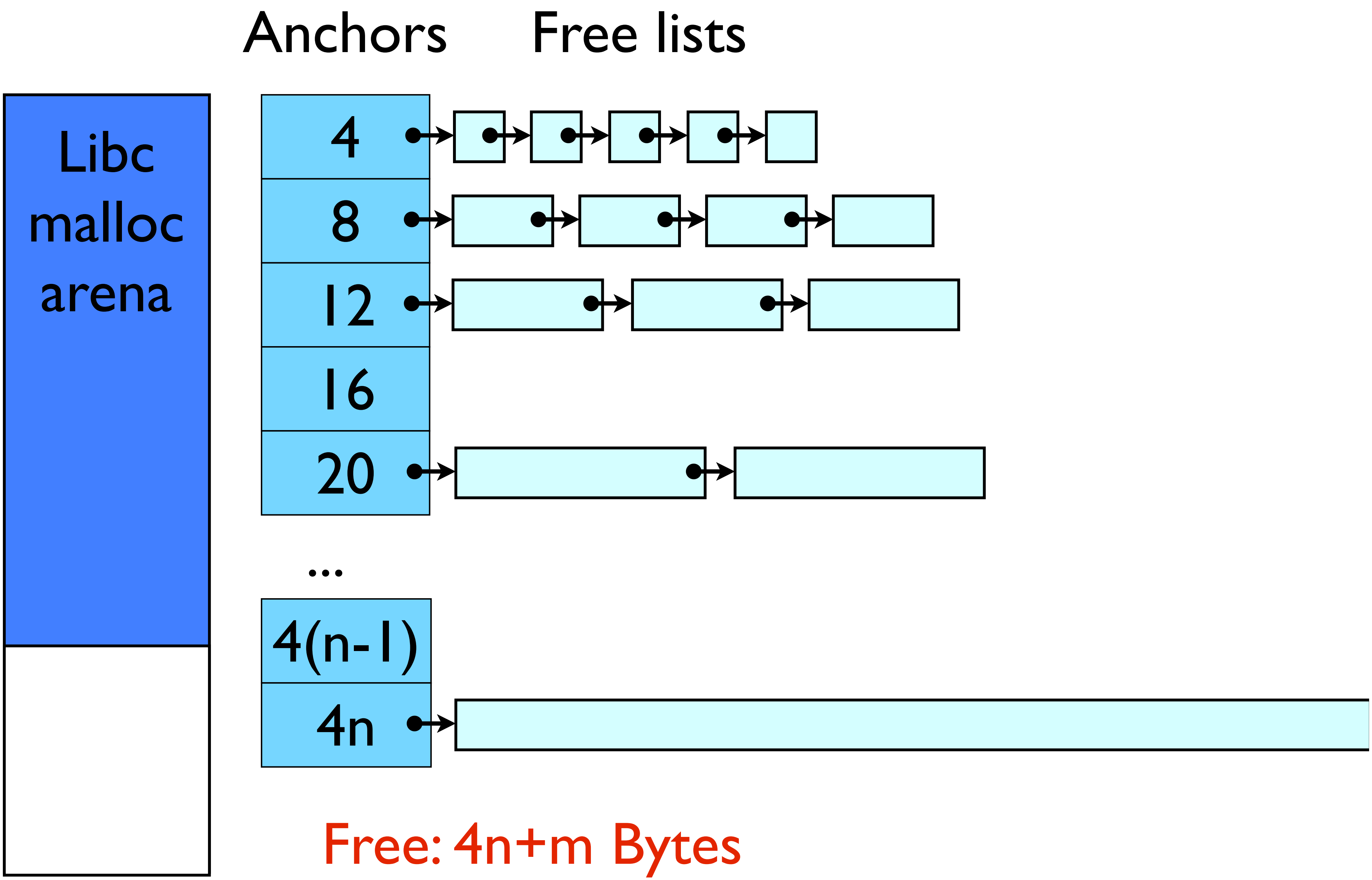


Free: 12 Bytes

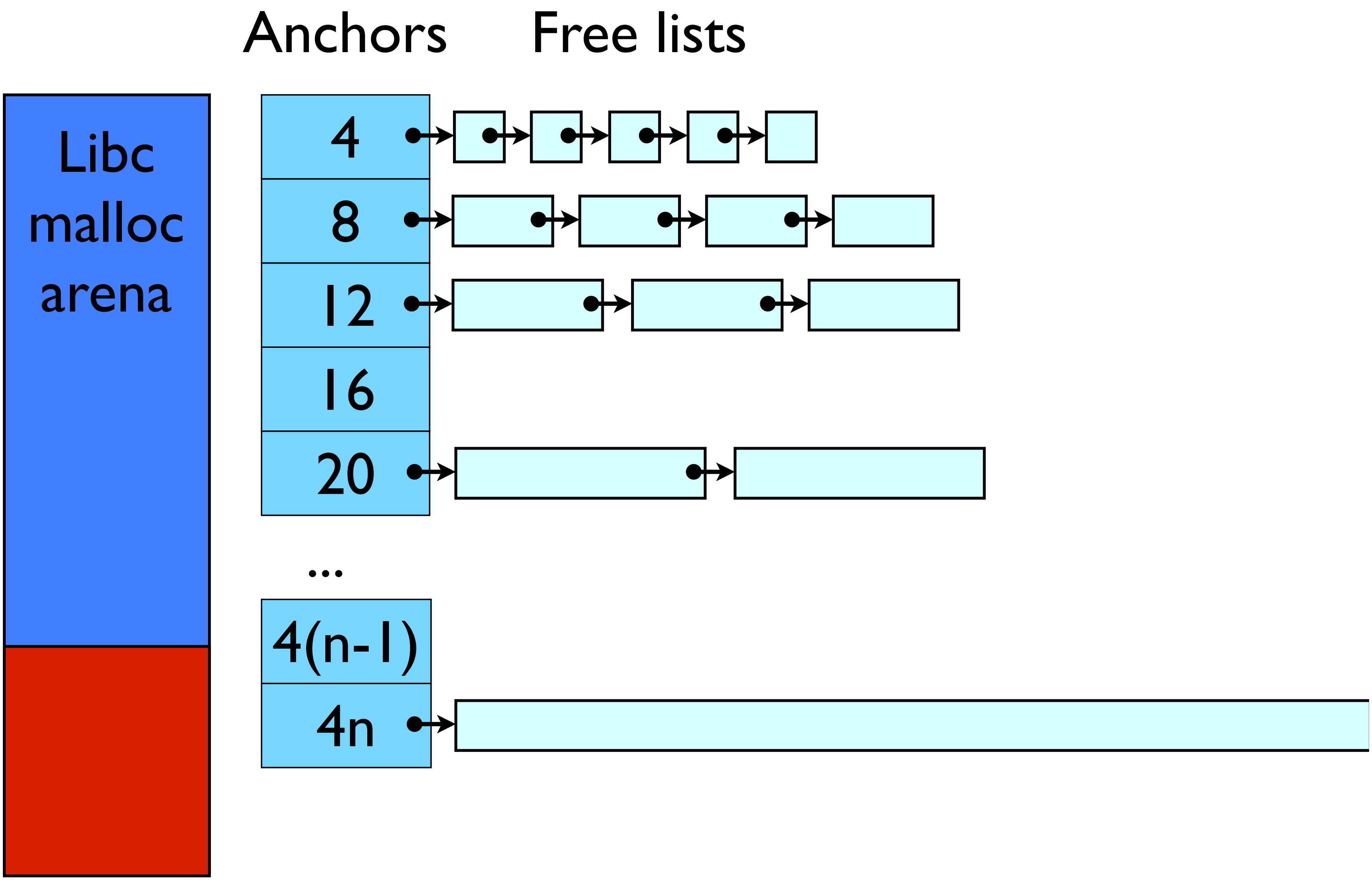
Memory Management illustrated



Memory Management illustrated



Memory Management illustrated



Exercise: Influence of Memory Management

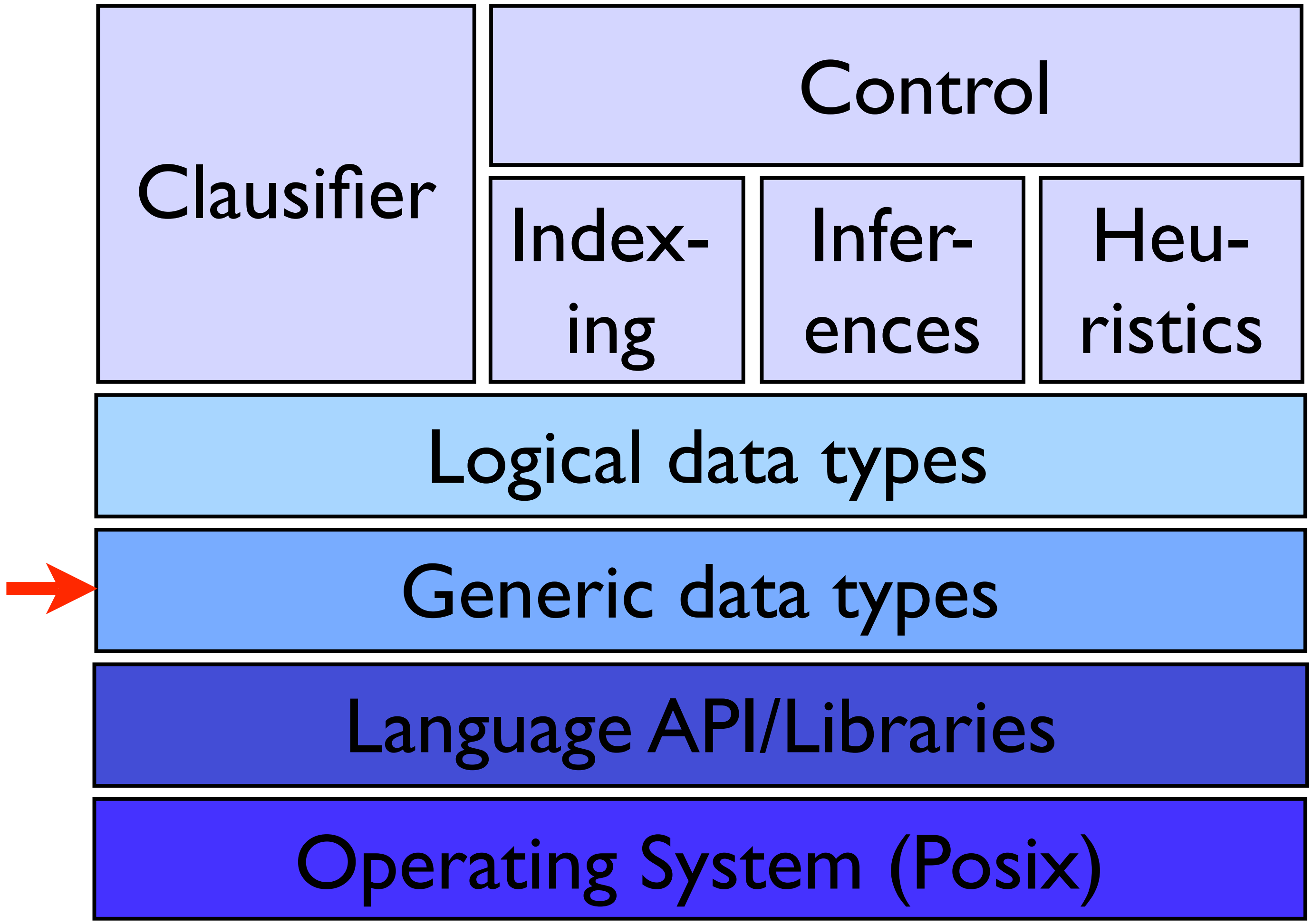
- E can be build with 2 different workin memory management schemes
  - Vanilla libC malloc()
    - Add compiler option `-DUSE_SYSTEM_MEM` in E/Makefile.vars
  - Freelists backed by malloc() (see above)
    - Default version

- Compare the performance yourself:
  - Run default E a couple of times with output disabled
  - `eprover -s --resources-info LUSK6ext.lop`
  - Take note of the reported times
  - Enable use of system malloc(), then `make rebuild`
  - Rerun the tests and compare the times

Makefile.vars

```
...
BUILDFLAGS = -DPRINT_SOMEERRORS_STDOUT \
             -DMEMORY_RESERVE_PARANOID \
             -DPRINT_TSTP_STATUS \
     -DSTACK_SIZE=32768  \
             -DUSE_SYSTEM_MEM \
             # -DFULL_MEM_STATS\
             # -DPRINT_RW_STATE # -DMEASURE_EXPENSIVE
...
```

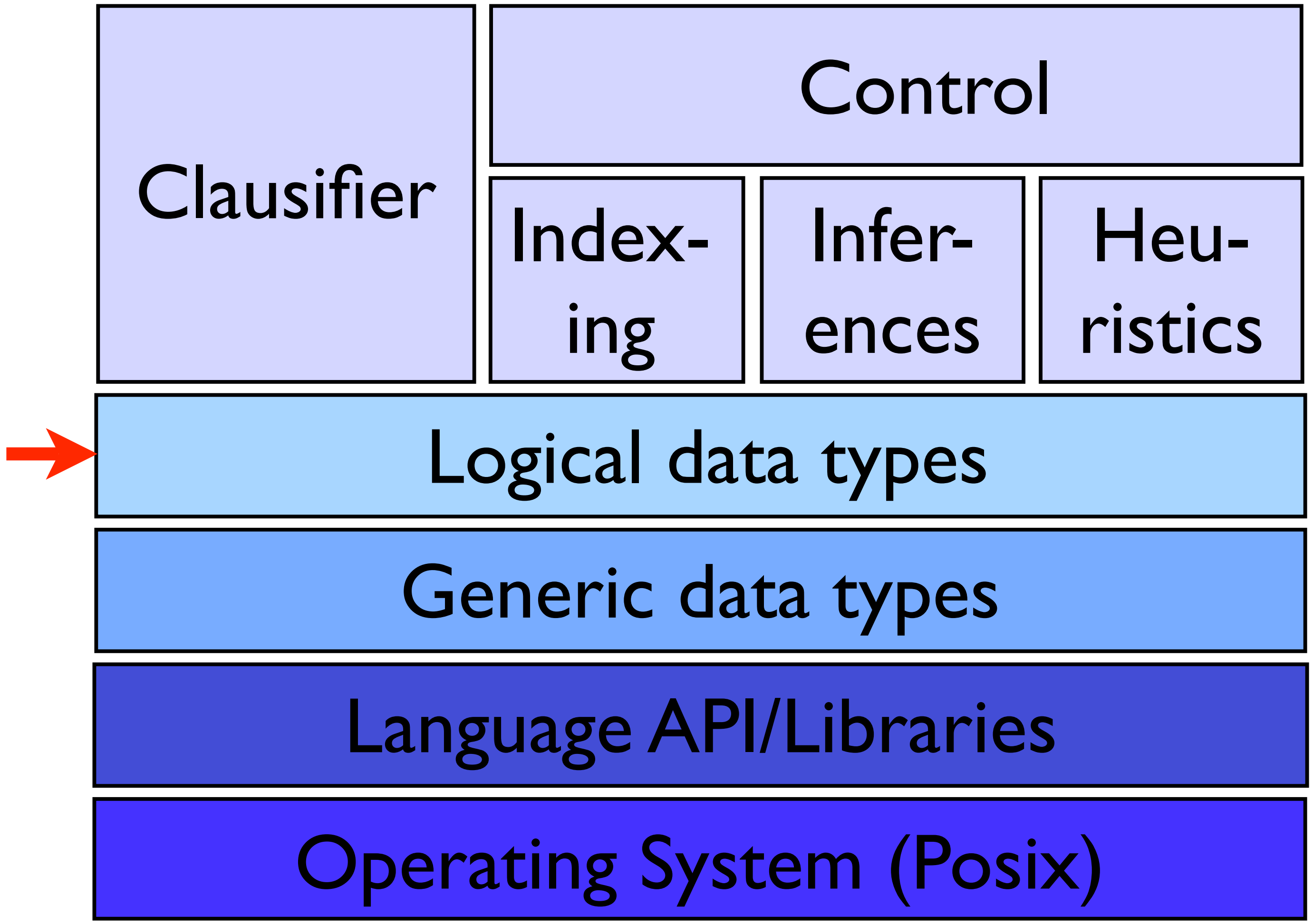
Layered Architecture



Generic Data types

- (Dynamic) Stacks
- (Dynamic) Arrays
- Hashes
- Singly linked lists
- Doubly linked lists
- Tries
- Splay trees [ST85]
- Skip lists [Pug90]

Layered Architecture



First-Order Terms

- Terms are words over the alphabet $F \cup V \cup \{'(', ')', ','\}$, where...

- Variables: $V = \{X, Y, Z, X1, \ldots\}$

- Function symbols: $F = \{f/2, g/1, a/0, b/0, \ldots\}$

- Definition of terms:

  – $X \in V$ is a term
  – $f/n \in F, t_1, \ldots, t_n$ are terms $\rightsquigarrow f(t_1, \ldots, t_n)$ is a term
  – Nothing else is a term

Terms are by far the most frequent objects in a typical proof state!
$\rightsquigarrow$ Term representation is critical!

Representing Function Symbols and Variables

- Naive: Representing function symbols as strings: `"f"`, `"g"`, `"add"`
  - May be ok for f, g, add
  - Users write $unordered\_pair, universal\_class, \ldots$
- Solution: Signature table
  - Map each function symbol to unique small positive integer
  - Represent function symbol by this integer
  - Maintain table with meta-information for function symbols indexed by assigned code
- Handling variables:
  - Rename variables to $\{X_1, X_2, \ldots\}$
  - Represent X_i by $-i$
  - Disjoint from function symbol codes!

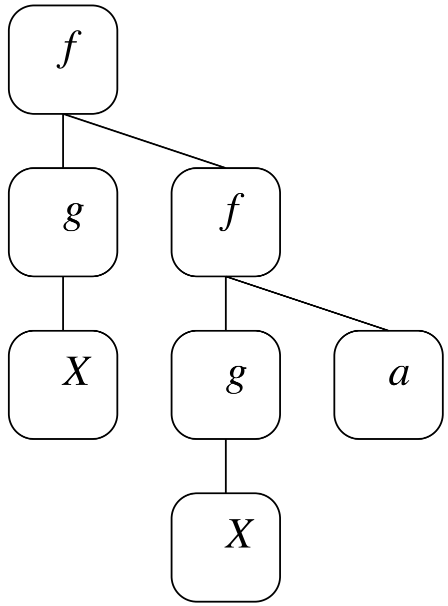
$\boxed{\text{From now on, assume this always done!}}$

Representing Terms

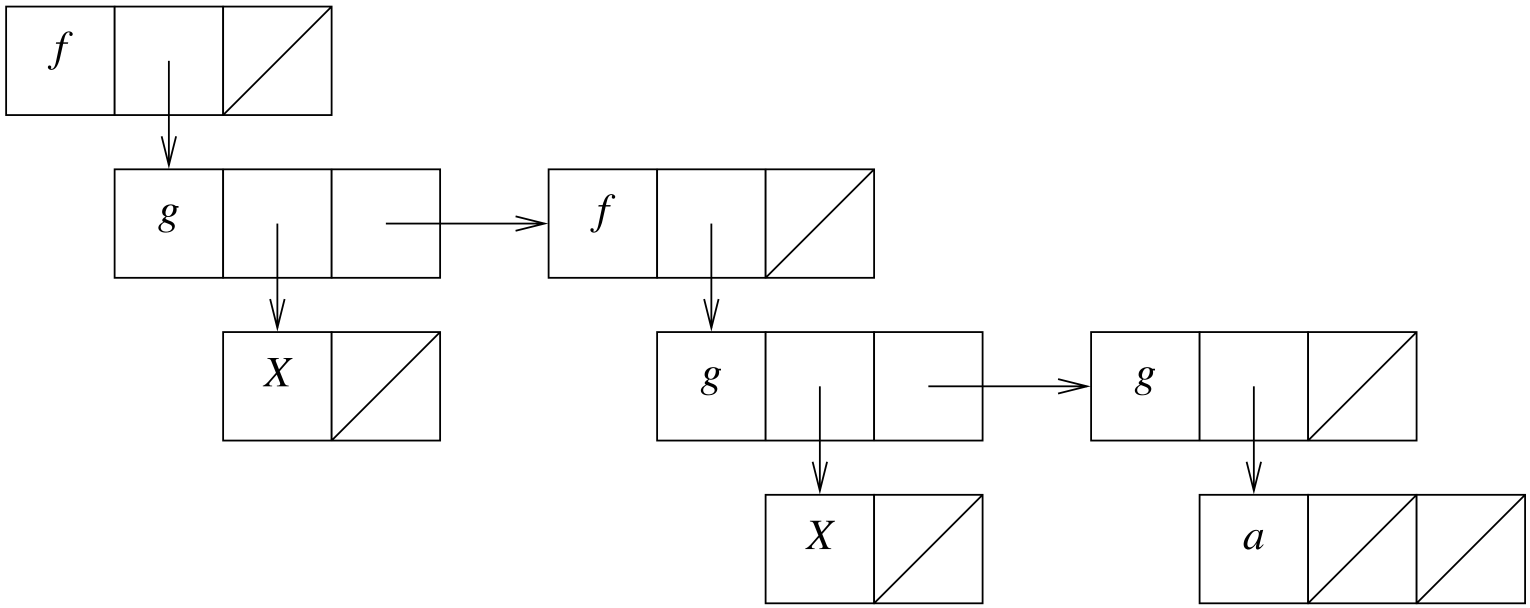
- Naive: Represent terms as strings `"f(g(X), f(g(X),a))"`
- More compact: `"fgXfgXa"`
  - Seems to be very memory-efficient!
  - But: Inconvenient for manipulation!
- Terms as ordered trees
  - Nodes are labeled with function symbols or variables
  - Successor nodes are subterms
  - Leaf nodes correspond to variables or constants
  - Obvious approach, used in many systems!

Abstract Term Trees

- Example term: $f(g(X), f(g(X), a))$

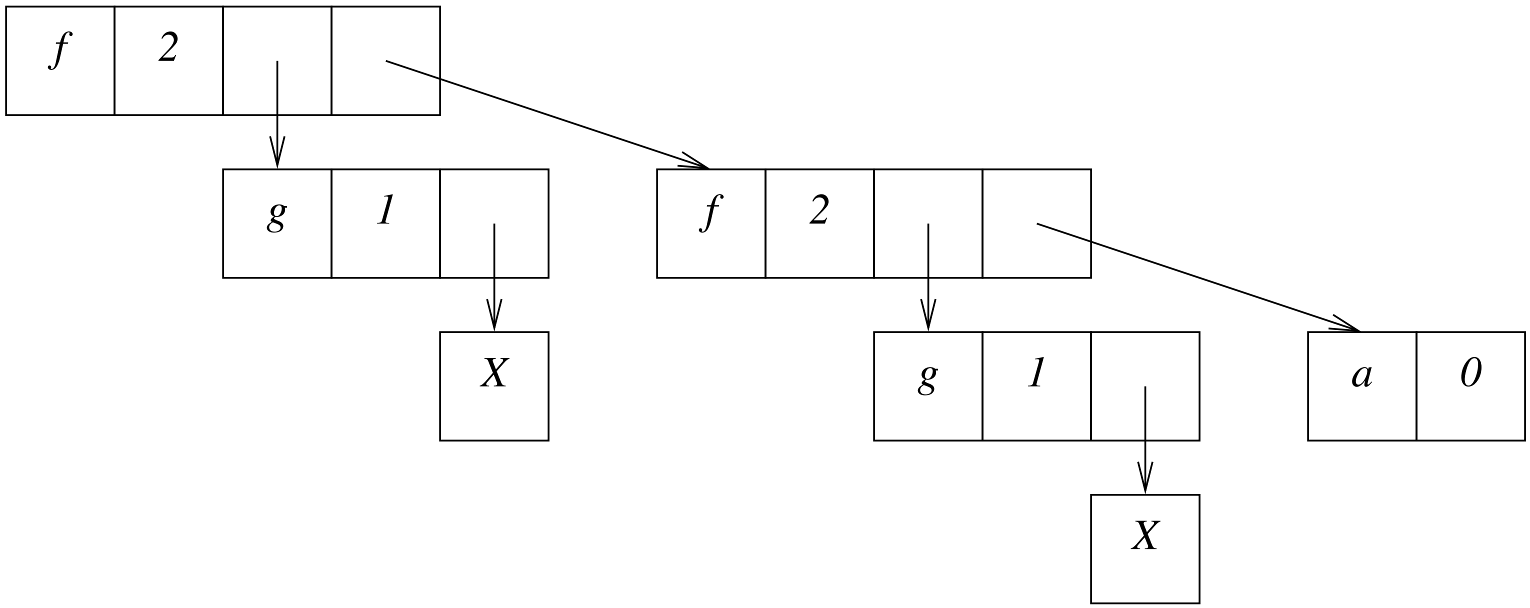


LISP-Style Term Trees



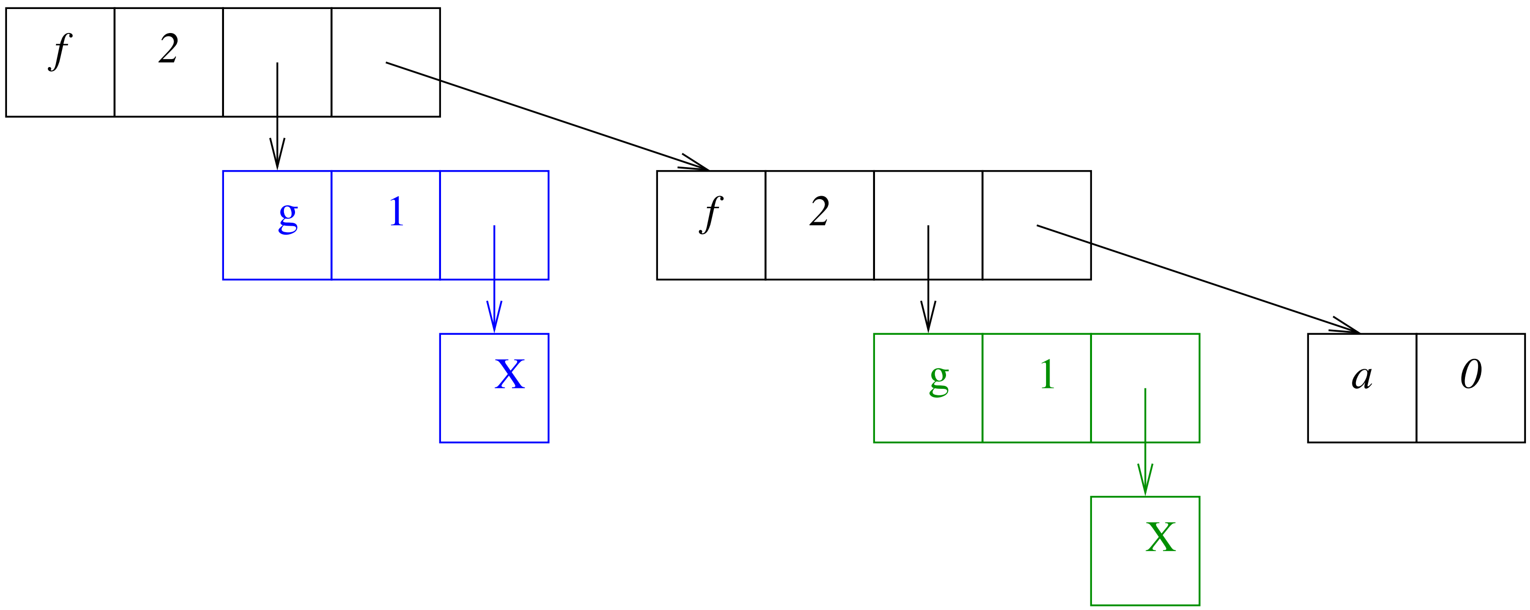
- Argument lists are represented as linked lists
- Implemented e.g. in PCL tools for DISCOUNT and Waldmeister

C/ASM Style Term Trees



- Argument lists are represented by arrays with length
- Implemented e.g. in DISCOUNT (as an evil hack)

C/ASM Style Term Trees

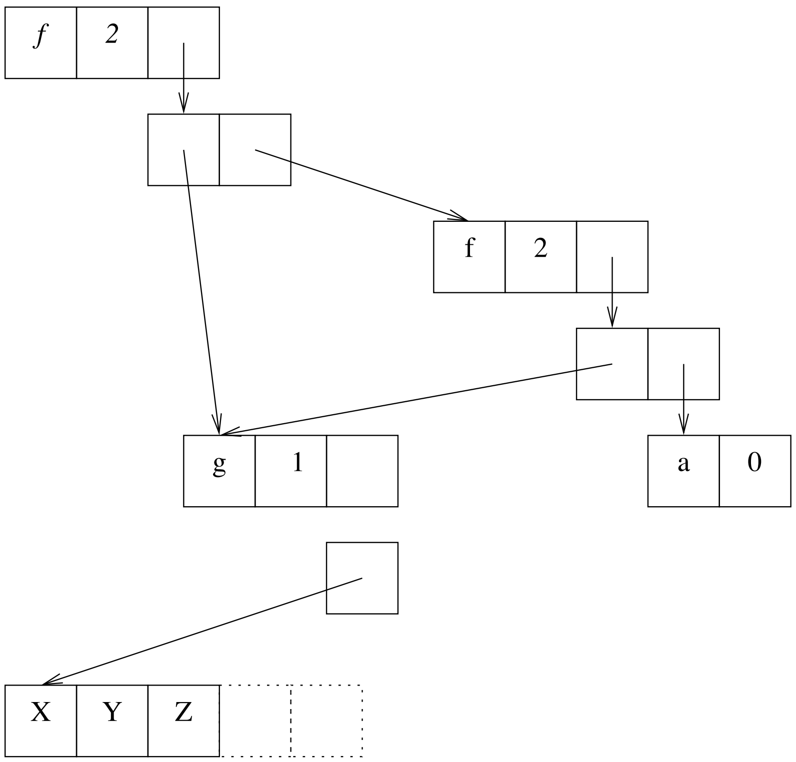


- In this version: Isomorphic subterms have isomorphic representation!

Exercise: Term Datatype in E

- E's basic term data type is defined in `E/TERMS/cte_termtypes.h`
  - Which term representation does E use?

Shared Terms (E)



- Idea: Consider terms not as trees, but as DAGs
  - Reuse identical parts
  - Shared variable banks (trivial)
  - Shared term banks maintained bottom-up

Shared Terms

- Disadvantages:
  - More complex
  - Overhead for maintaining term bank
  - Destructive changes must be avoided

- Direct Benefits:
  - Saves between 80% and 99.99% of term nodes
  - Consequence: We can afford to store precomputed values
    - Term weight
    - Rewrite status (see below)
    - Groundness flag
    - . . .
  - Term identity: One pointer comparison!

Literal Datatype

- See `E/CLAUSES/ccl_eqn.h`
- Equations are basically pairs of terms with some properties

```
/* Basic data structure for rules, equations, literals. Terms are
   always assumed to be shared and need to be manipulated while taking
   care about references! */

typedef struct eqncell
{
   EqnProperties  properties;/* Positive, maximal, equational */
   Term_p         lterm;
   Term_p         rterm;
   int            pos;
   TB_p           bank;      /* Terms are from this bank */
   struct eqncell *next;     /* For lists of equations */
}EqnCell, *Eqn_p, **EqnRef;
```

Clause Datatype

- See `E/CLAUSES/ccl_clause.h`

- Clauses are containers with Meta-information and literal lists

```
typedef struct clause_cell
{
   long                  ident;        /* Hopefully unique ident for
 all clauses created during
 proof run */
   SysDate               date;         /* ...at which this clause
 became a demodulator */
   Eqn_p                 literals;     /* List of literals */
   short                 neg_lit_no;   /* Negative literals */
   short                 pos_lit_no;   /* Positive literals */
   long                  weight;       /* ClauseStandardWeight()
 precomputed at some points in
 the program */
   Eval_p                evaluations; /* List of evauations */
```

```
   ClauseProperties      properties;  /* Anything we want to note at
 the clause? */
...
   struct clausesetcell* set;         /* Is the clause in a set? */
   struct clause_cell*   pred;        /* For clause sets = doubly  */
   struct clause_cell*   succ;        /* linked lists */
}ClauseCell, *Clause_p;
```

Summary Day 1

- First-order logic with equality
- Superposition calculus
  - Generating inferences ("Superposition rule")
  - Rewriting
  - Subsumption
- Proof procedure
  - Basic given-clause algorithm
  - DISCOUNT Loop
- Software architecture
  - Low-level components
  - Logical datetypes

Literature Online

- My papers are at `http://www4.informatik.tu-muenchen.de/~schulz/bibliography.html`
- The Workshop versions of Bernd Löchners LPO/KBO papers [Löc06, L0̈6] are published in the "Empricially Successful" series of Workshops. Proceedings are at `http://www.eprover.org/EVENTS/es_series.html`
  - "Things to know when implementing LPO": Proceedings of Empirically Successful First Order Reasoning (2004)
  - "Things to know when implementing KPO": Proceedings of Empirically Successful Classical Automated Reasoning (2005)
- Technical Report version of [BG94]:
  - `http://domino.mpi-inf.mpg.de/internet/reports.nsf/c125634c000710d4c12560410043ec01/c2de67aa270295ddc12560400038fcc3!OpenDocument`
  - . . . or Google "Bachmair Ganzinger 91-208"

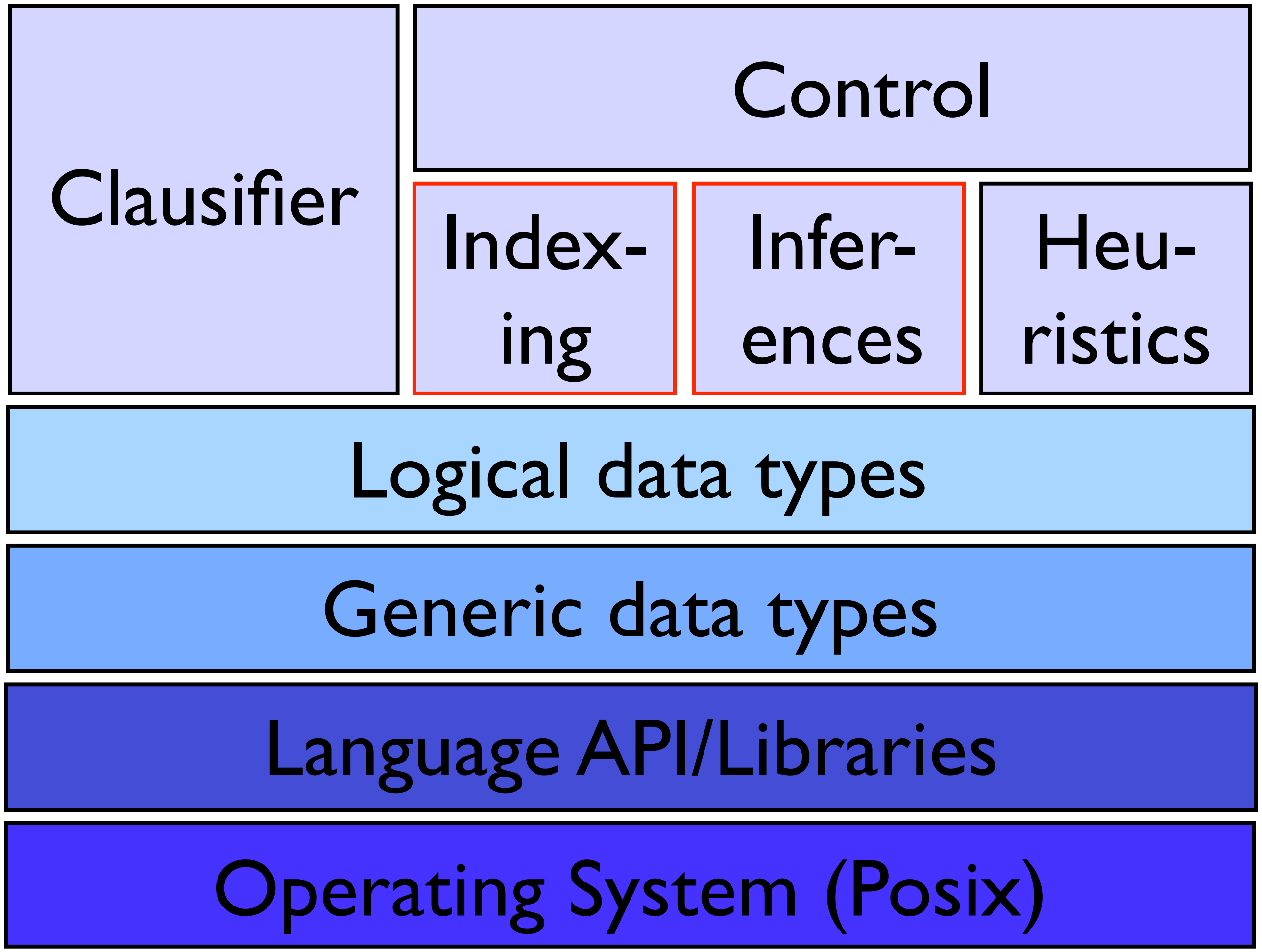
"LUSK6" Example

```
#   Problem:    In a ring, if x*x*x = x for all x
#               in the ring, then
#               x*y = y*x for all x,y in the ring.
#
#               Functions:      f   : Multiplikation *
#                               J   : Addition +
#                               g   : Inverses
#                               e   : Neutrales Element
#                               a,b : Konstanten

j (0,X)         = X.                        # 0 ist a left identity for sum
j (X,0)         = X.                        # 0 ist a right identity for sum
j (g (X),X)     = 0.                        # there exists a left inverse for sum
j (X,g (X))     = 0.                        # there exists a right inverse for sum
j (j (X,Y),Z) = j (X,j (Y,Z)).              # associativity of addition
j (X,Y)         = j(Y,X).                   # commutativity of addition
f (f (X,Y),Z) = f (X,f (Y,Z)).              # associativity of multiplication
f (X,j (Y,Z)) = j (f (X,Y),f (X,Z)).        # distributivity axioms
f (j (X,Y),Z) = j (f (X,Z),f (Y,Z)).        #
f (f(X,X),X)   = X.                         # special hypothese: x*x*x = x

f (a,b) != f (b,a).                         # (Skolemized) theorem
```

LUSK6 in TPTP-3 syntax

```
cnf(j_neutral_left,        axiom, j(0,X)          = X).
cnf(j_neutral_right,       axiom, j(X,0)          = X).
cnf(j_inverse_left,        axiom, j(g(X),X)       = 0).
cnf(j_inverse_right,       axiom, j(X,g(X))       = 0).
cnf(j_commutes,            axiom, j(X,Y)          = j(Y,X)).
cnf(j_associates,          axiom, j(j(X,Y),Z) = j(X,j(Y,Z))).
cnf(f_associates,          axiom, f(f(X,Y),Z) = f(X,f(Y,Z))).
cnf(f_distributes_left,    axiom, f(X,j(Y,Z)) = j(f(X,Y),f(X,Z))).
cnf(f_distributes_right, axiom, f(j(X,Y),Z) = j(f(X,Z),f(Y,Z))).
cnf(x_cubedequals_x,       axiom, f(f(X,X),X) = X).

fof(mult_commutes,conjecture,![X,Y]:(f(X,Y) = f(Y,X))).
```

Layered Architecture



Efficient Rewriting

- Problem:
  - Given term t, equations $E = \{l_1 \simeq r_1 \ldots l_n \simeq r_n\}$
  - Find normal form of t w.r.t. E
- Bottlenecks:
  - Find applicable equations
  - Check ordering constraint ($\sigma(l) > \sigma(r)$)
- Solutions in E:
  - Cached rewriting (normal form date, pointer)
  - Perfect discrimination tree indexing with age/size constraints

Shared Terms and Cached Rewriting

- Shared terms can be long-term persistent!
- Shared terms can afford to store more information per term node!
- Hence: Store rewrite information
  - Pointer to resulting term
  - Age of youngest equation with respect to which term is in normal form
- Terms are at most rewritten once!
- Search for matching rewrite rule can exclude old equations!

Indexing

- Quickly find inference partners in large search states
  - Replace linear search with index access
  - Especially valuable for simplifying inferences

- More concretely (or more abstractly?):
  - Given a set of terms or clauses S
  - and a query term or query clause
  - and a retrieval relation R
  - Build a data structure to efficiently find (all) terms or clauses t from S such that $R(t, S)$ (the retrieval relation holds)

Introductory Example: Text Indexing

- Problem: Given a set `D` of text documents, find all documents that contain a certain word `w`
- Obviously correct implementation:

```
result = {}
for doc in D
    for word in doc
        if w == word
            result = result ∪{ doc }
            break;
return result
```

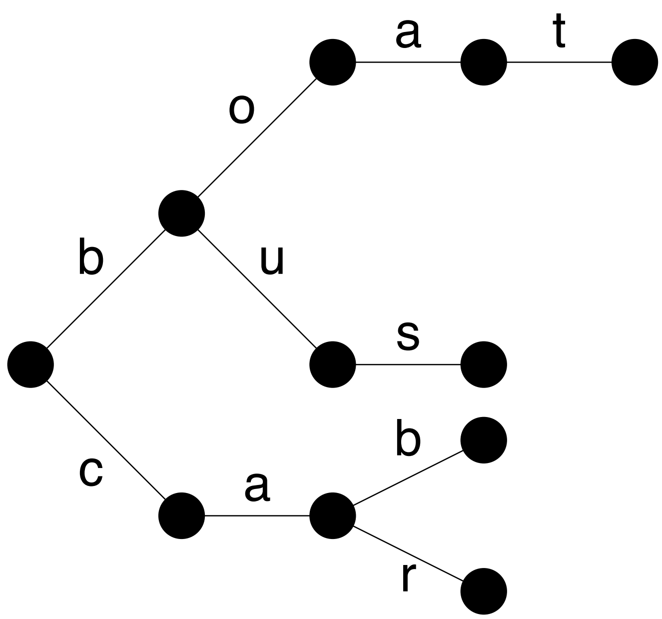
- Now think of Google. . .
  - Obvious approach (linear scan through documents) breaks down for large D
  - Instead: Precompiled Index $I : words \rightarrow documents$
  - Requirement: I efficiently computable for large number of words!

The Trie Data Structure

- Definition: Let Σ be a finite alphabet and Σ^* the set of all words over Σ
  - We write $|w|$ for the length of w
  - If $u, v \in \Sigma^*$, $w = uv$ is the word with prefix u
- A trie is a finite tree whose edges are labelled with letters from Σ
  - A node represents a set of words with a common prefix (defined by the labels on the path from the root to the node)
  - A leaf represents a single word
  - The whole trie represents the set of words at its leaves
  - Dually, for each set of words S (such that no word is the prefix of another), there is a unique trie T
- Fact: Finding the leaf representing w in T (if any) can be done in $O(|w|)$
  - This is independent of the size of S!
  - Inserting and deleting of elements is just as fast

Trie Example

- Consider $\Sigma = \{a, b, ..., z\}$ and $S = \{car, cab, bus, boat\}$
- The trie for S is:



- Tries can be built incrementally
- We can store extra infomation at nodes/leaves
  - E.g. all documents in which $boat$ occurs
  - Retrieving this information is fast and simple

Indexing Techniques for Theorem Provers

- Term Indexing standard technique for high performance theorem provers
  - Preprocess term sets into index
  - Return terms in a certain relation to a query term
    - Matches query term (find generalizations)
    - Matched by query term (find specializations)

- Perfect indexing:
  - Returns exactly the desired set of terms
  - May even return substitution

- Non-perfect indexing:
  - Returns candidates (superset of desired terms)
  - Separate test if candiate is solution

Frequent Operations

- Let S be a set of clauses
- Given term t, find an applicable rewrite rule in S
  - Forward rewriting
  - Reduced to: Given t, find $l \simeq r \in S$ such that $l\sigma = t$ for some σ
  - Find generalizations
- Given $l \rightarrow r$, find all rewritable clauses in S
  - Backward rewriting
  - Reduced to: Given l, find t such that $C|_p\sigma = l$
  - Find instances
- Given C, find a subsuming clause in S
  - Forward subsumption
  - Not easily reduced. . .
  - Backward subsumption analoguous

Classification of Indexing Techniques

- Perfect indexing
  - The index returns exactly the elements that fullfil the retrieval condition
  - Examples:
    - Perfect discrimination trees
    - Substitution trees
    - Context trees
- Non-perfect indexing:
  - The index returns a superset of the elements that fullfil the retrieval condition
  - Retrieval condition has to be verified
  - Examples:
    - (Non-perfect) discrimination trees
    - (Non-perfect) Path indexing
    - Top-symbol hashing
    - Feature vector-indexing

The Given Clause Algorithm

U: Unprocessed (passive) clauses (initially Specification)
P: Processed (active) clauses (initially: empty)

while $U \neq \{\}$
    $g =$ delete_best(U)
    $g =$ simplify(g,P)
    if $g == \square$
        SUCCESS, Proof found
    if g is not redundant w.r.t. P
        $T = \{c \in P | c$ redundant or simplifiable w.r.t. $g\}$
        $P = (P \backslash T) \cup \{g\}$
        $T = T \cup$generate(g, P)
        foreach $c \in T$
            $c =$cheap_simplify(c, P)
            if c is not trivial
                $U = U \cup \{c\}$
SUCCESS, original U is satisfiable

Typically, $|U| \sim |P|^2$ and $|U| \approx \sum |T|$

The Given Clause Algorithm

U: Unprocessed (passive) clauses (initially Specification)
P: Processed (active) clauses (initially: empty)

while $U \neq \{\}$
    $g =$ delete_best(U)
    $g =$ simplify(g,P)
    if $g == \Box$
        SUCCESS, Proof found
    if g is not redundant w.r.t. P
        $T = \{c \in P | c$ redundant or simplifiable w.r.t. $g\}$
        $P = (P \backslash T) \cup \{g\}$
        $T = T \cup$generate(g, P)
        foreach $c \in T$
            $c =$cheap_simplify(c, P)
            if c is not trivial
                $U = U \cup \{c\}$
SUCCESS, original U is satisfiable

Simplification of new clauses is bottleneck

Sequential Search for Forward Rewriting

- Given t, find $l \simeq r \in S$ such that $l\sigma = t$ for some σ

- Naive implementation (e.g. DISCOUNT):

  function find_matching_rule(t, S)
      for $l \simeq r \in S$
          $\sigma =$ match(l, t)
          if σ and $l\sigma > r\sigma$
              return (σ, $l \simeq r$)

- Remark: We assume that for unorientable $l \simeq r$, both $l \simeq r$ and $r \simeq l$ are in S

Conventional Matching

```
match(s,t)
    return match_list([s], [t], {})

match_list(ls, lt, σ)
    while ls ≠ []
        s = head(ls)
        t = head(lt)
        if s == X ∈ V
            if X ← t' ∈ σ
                if t ≠ t' return FAIL
                    else
                        σ = σ ∪ {X ← t}
        else if t == X ∈ V return FAIL
        else
            let s = f(s_1, ..., s_n)
            let t = g(t_1, ..., t_m)
            if f ≠ g return FAIL /* Otherwise n = m! */
        ls = append(tail(ls), [s_1, ... s_n]
        lt = append(tail(lt), [t_1, ... t_m])
    return σ
```

The Size of the Problem

- Example LUSK6:
  - Run time with E on 1GHz Powerbook: 1.7 seconds
  - Final size of P: 265 clauses (processed: 1542)
  - Final size of U: 26154 clauses
  - Approximately 150,000 successful rewrite steps
  - Naive implementation: $\approx$ 50-150 times more match attempts!
  - $\approx$ 100 machine instructions/match attempt
- Hard examples:
  - Several hours on 3+GHz machines
  - Billions of rewrite attempts
- Naive implementations don't cut it!

Top Symbol Hashing

- Simple, non-perfect indexing method for (forward-) rewriting

- Idea: If $t = f(t_1, \ldots, t_n)$ ($n \geq 0$), then any s that matches t has to start with f
  - $top(t) = f$ is called the top symbol of t

- Implementation:
  - Organize $S = \cup S_f$ with $S_f = \{l \simeq r \in S | top(l) = f\}$
  - For non-variable query term t, test only rewrite rules from $S_{top(t)}$

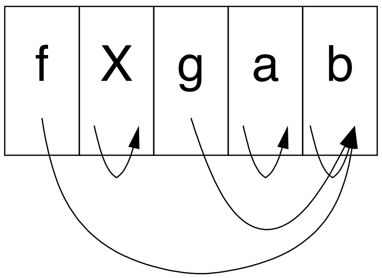
- Efficiency depends on problem composition
  - Few function symbols: Little improvement
  - Large signatures: Huge gain
  - Typically: Speed-up factor 5-15 for matching

String Terms and Flat Terms

- Terms are (conceptually) ordered trees
  - Recursive data structure
  - But: Conventional matching always does left-right traversal
  - Many other operations do likewise

- Alternative representation: String terms
  - $f(X, g(a, b))$ already is a string...
  - If arity of function symbols is fixed, we can drop braces: f$Xgab$
  - Left-right iteration is much faster (and simpler) for string terms

- Flat terms: Like string terms, but with term end pointers



  - Allows fast jumping over subterms for matching

Perfect discrimination tree indexing

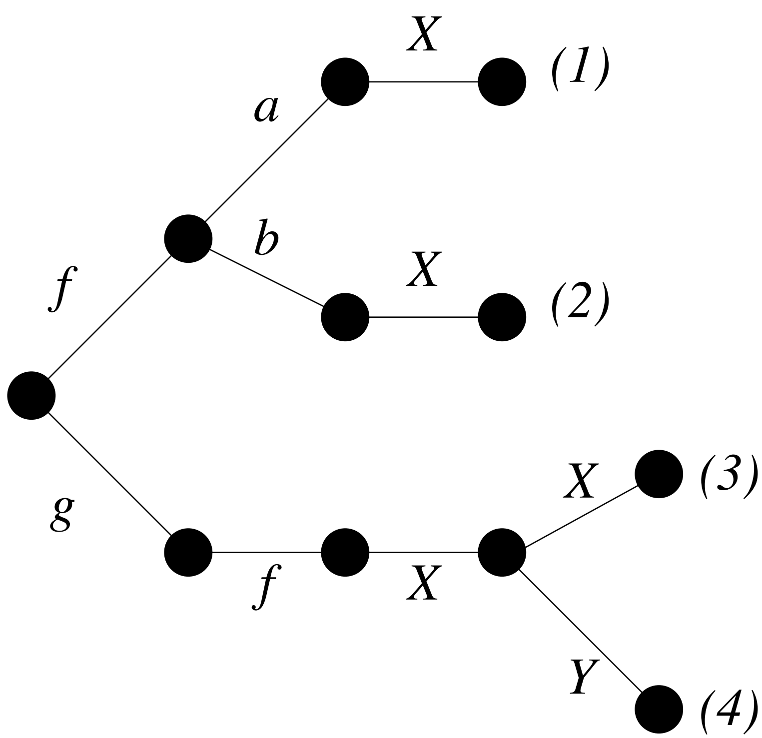
- Generalization of top symbol hashing

- Idea: Share common prefixes of terms in string representation
  - Represent terms as strings
  - Store string terms (left hand sides of rules) in trie (perfect discrimination tree)
  - Recursively traverse trie to find matching terms for a query:
    - At each node, follow all compatible vertices in turn
    - If following a variable branch, add binding for variable
    - If no valid possibility, backtrack to last open choice point
    - If leaf is reached, report match

- Currently most frequently used indexing technique
  - E (rewriting, unit subsumption)
  - Vampire (rewriting, unit- and non-unit subsumption (as code trees))
  - Waldmeister (rewriting, unit subsumption, paramodulation)
  - Gandalf (rewriting, subsumption)
  - ...

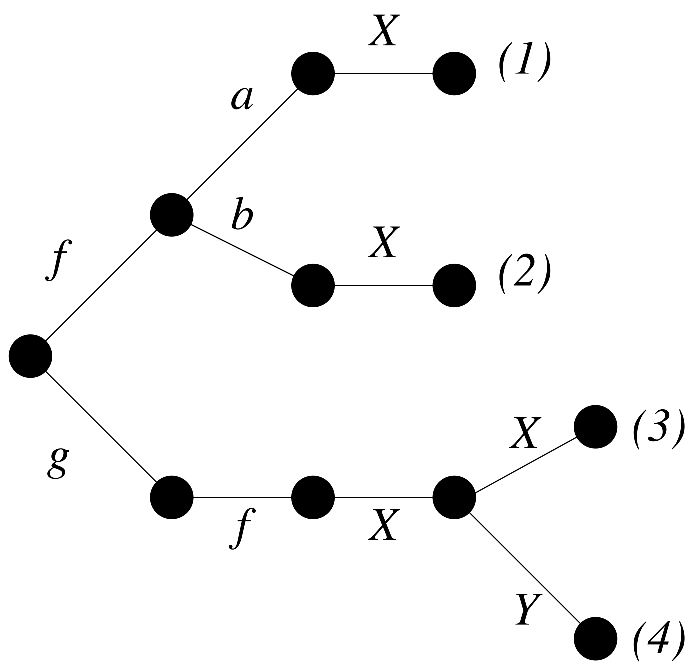
Example

- Consider $S = \{(1)f(a,X) \simeq a, (2)f(b,X) \simeq X,$
$(3)g(f(X,X)) \simeq f(Y,X), (4)g(f(X,Y)) \simeq g(X)\}$
  - String representation of left hand sides: $faX, fbX, gfXX, gfXY$
  - Corresponding trie:



Find matching rule for $g(f(a,g(b)))$

Example Continued



- Start with $g(f(a,g(b)))$, root node, $\sigma = \{\}$

  $\color{red}{g}(f(a,g(b)))$ Follow g vertex
  $g(\color{red}{f}(a,g(b)))$ Follow f vertex
  $g(f(\color{red}{a},g(b)))$ Follow X vertex, $\sigma = \{X \leftarrow a\}$, jump over a
  $g(f(a,\color{red}{g}(b)))$

  – Follow X vertex - Conflict! X already bound to a
  – Follow Y, $\sigma = \{X \leftarrow a, Y \leftarrow g(b)\}$, jump over $g(b)$ Rule 4 matches

Subsumption Indexing

- Subsumption: Important simplification technique for first-order reasoning
  - Drop less general (redundant) clauses
  - Keep more general clause
- Problem: Efficiently detecting subsumed clauses
  - Individual clause-clause subsumption is in NP
  - Large number of subsumption relations must be tested
- Major Approach: Indexing
  - Use precompiled data structures to efficiently select
    - subsuming clauses (forward subsumption)
    - subsumes clause (backward subsumption)

    from large (and fairly static) clause sets
- Usual: Different and complex indexing approaches for forward- and backward subsumption

Subsumption

- Idea: Only keep the most general clauses
  - If one clause is subsumed by another, discard it
- Formally: A clause C subsumes C' if:
  - There exists a substitution σ such that $C\sigma \subseteq C'$
  - Note: In that case $C \models C'$
  - $\subseteq$ usually is the multi-subset relation
- Examples:
  - $p(X)$ subsumes $p(a) \vee q(f(X), a)$ ($\sigma = \{X \leftarrow a\}$)
  - $p(X) \vee p(Y)$ does not multi-set-subsume $p(a) \vee q(f(X), a)$
  - $q(X, Y) \vee q(X, a)$ subsumes $q(a, a) \vee q(a, b)$
- Subsumption is hard (NP-complete)
  - $n!$ permutations in non-equational clause with n literals
  - $n!2^n$ permutations in equational clause with n literals

Forward- and Backward Subsumption

- Assume a set of clauses P and a given clause p

- Forward subsumption: Is there any clause in P that subsumes g?

- Backward subsumption: Find/remove all clauses in P subsumed by g

- Notice that these are clause–clause set operations

- Naive implementation: Sequence of clause-clause operations
  - Good implementation can speed up (average case) individual subsumption
  - Number of attempts still very high

- Smarter: Avoid many of the subsumption calls up front
  - Use indexing techniques to reduce number of candidates

Feature Vector Indexing

- New clause indexing technique
  - Not lifted from term indexing
- Advantages:
  - Small index (memory footprint)
  - Same index for forward- and backward subsumption
  - Very simple
  - Efficient in practice
  - Variants for different subsumption relations
- Disadvantages:
  - Non-perfect
  - Requires fixed signature for optimal performance

How does it work?

Properties of the Subsumption Relation

Definitions:

- Let C and C' be clauses
- C^+ is the (multi-)set (a clause) of positive literals in C
- C^- is the (multi-)set of negative literals in C
- $|C|_f$ is the number of occurences of (function or predicate) symbol f in C

Facts: If C subsumes C', then

- $|C^+| \leq |C'^+|$
- $|C^-| \leq |C'^-|$
- $|C^+|_f \leq |C'^+|_f$ for all f
- $|C^-|_f \leq |C'^-|_f$ for all f
- (Similar results exist for term depths)
- The same holds for all linear combination of these features

Remark: Composite critera are often used to detect subsumption failure early

- $|C| \leq |C'|$ (C cannot have more literals than C')
- $\sum_{f \in F} |C|_f \leq \sum_{f \in F} |C'|$ (C cannot have more symbols than C')

Feature Vectors

Definitions:

- A feature function f is a function from the set of clauses to $\mathbf{N}$
- f is subsumption-compatible, if C subsumes C' implies $f(C) \leq f(C')$
- A (subsumption-compatible) feature vector function F is a function from the set of clauses to $\mathbf{N}^n$ such that $\Pi_n^i \circ F$ (the projection of F to the ith component) is a subsumption-compatible feature function
- If v_1 and v_2 are feature vectors, we write $v_1 \leq_s v_2$, if $v_1[i] \leq v_2[i]$ for all i.

Fact:

- Assume F is a (subsumption-compatible) feature vector function
- Assume C subsumes C'
- By construction, $F(C) \leq_s F(C')$

Basic Principle of Feature Vector Indexing:

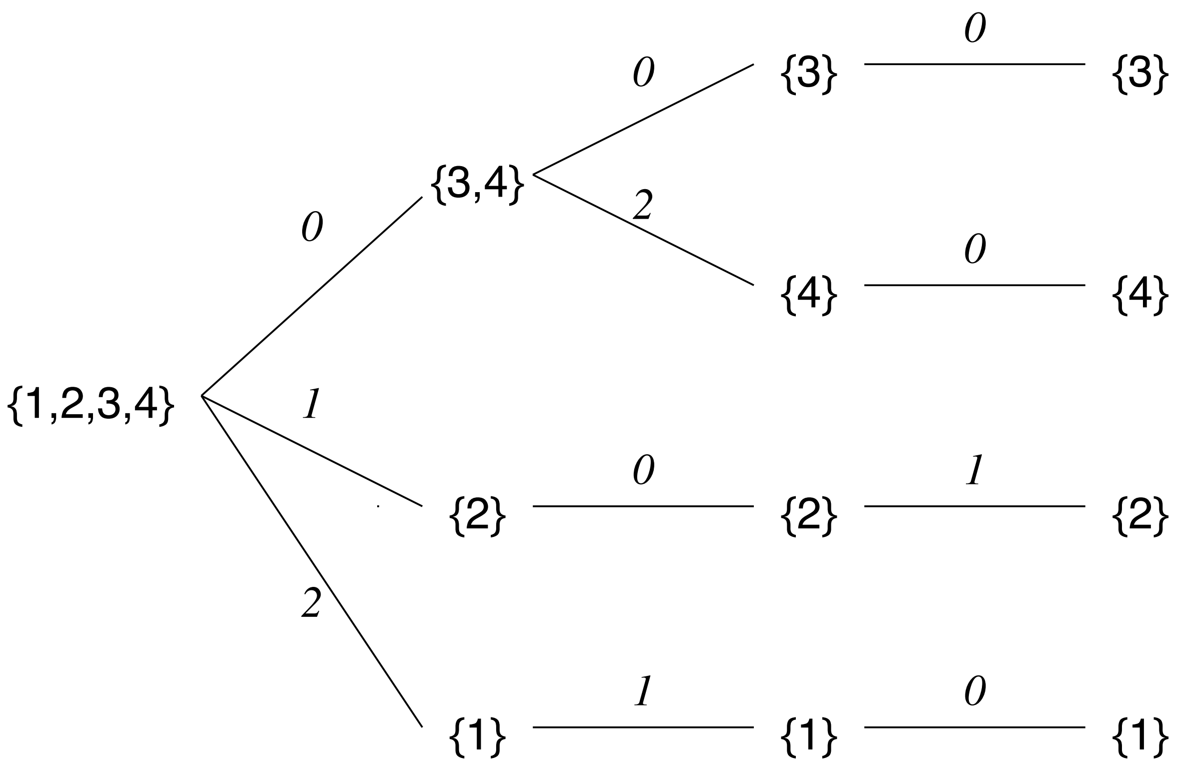
- For forward-subsumption: $candFS_F(P, g) = \{c \in P | F(c) \leq_s F(g)\}$
- For backward-subsumption: $candBS_F(P, g) = \{c \in P | F(g) \leq_s F(c)\}$

Feature Vector Indexing

- **Aim:** Efficiently compute $candFS_F(P, g)$ and $candBS_F(P, g)$
- **Solution:** Frequency vectors for P are compiled into a trie, clauses are stored in leaves
  - Tree of depth n (number of features in vector)
  - Nodes at depth d split according to feature $F(C)[d]$ (one successor per value)
  - All vectors with value $F(C)[d] = k$ associated with corresponding subtree
  - Construction continues recursively
- **Example:** Assume $F(C) := \langle |C^+|_a, |C^+|_f, |C^-|_b| \rangle$
  - Clause set $P = \{$1,2,3,4 $\}$ with
    1. $F(p(a) \vee p(f(a))) = \langle 2, 1, 0 \rangle$
    2. $F(p(a) \vee \neg p(b)) = \langle 1, 0, 1 \rangle$
    3. $F(\neg p(a) \vee p(b)) = \langle 0, 0, 0 \rangle$
    4. $F(p(X) \vee p(f(f(b)))) = \langle 0, 2, 0 \rangle$
  - Query $g = p(f(a))$
    * $F(g) = \langle 1, 1, 0 \rangle$

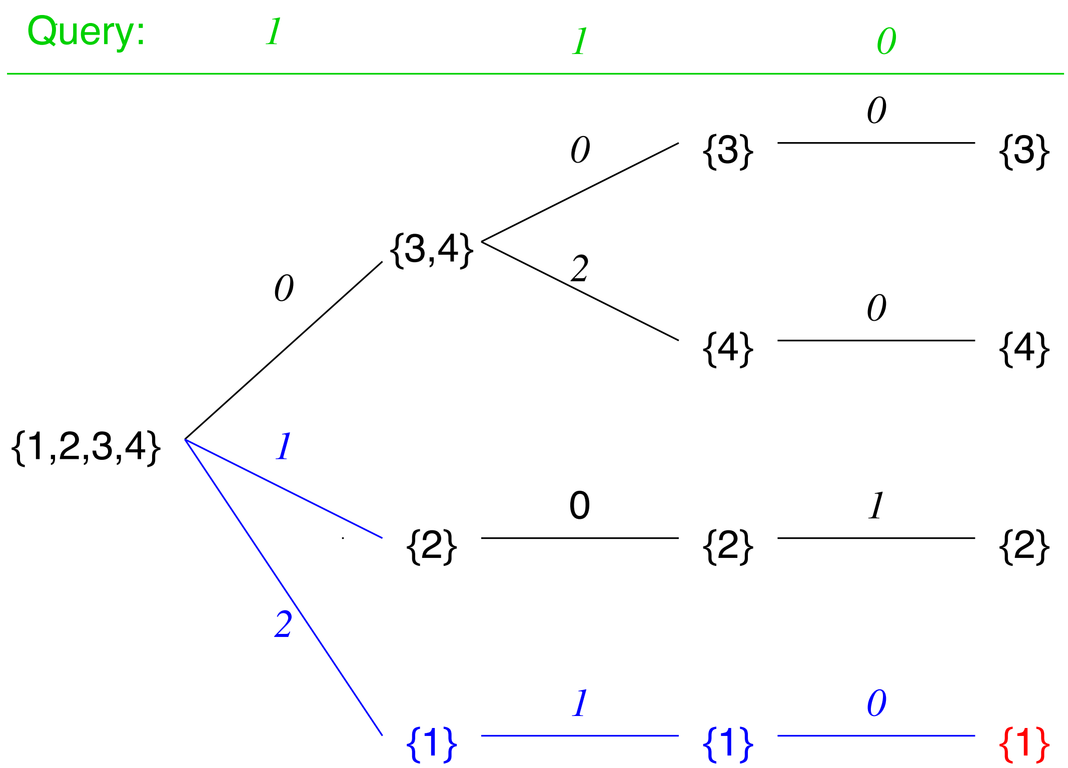
Example Index

1. $F(p(a) \vee p(f(a))) = \langle 2, 1, 0 \rangle$
2. $F(p(a) \vee \neg p(b)) = \langle 1, 0, 1 \rangle$
3. $F(\neg p(a) \vee p(b)) = \langle 0, 0, 0 \rangle$
4. $F(p(X) \vee p(f(f(b)))) = \langle 0, 2, 0 \rangle$



Example: Backward Subsumption

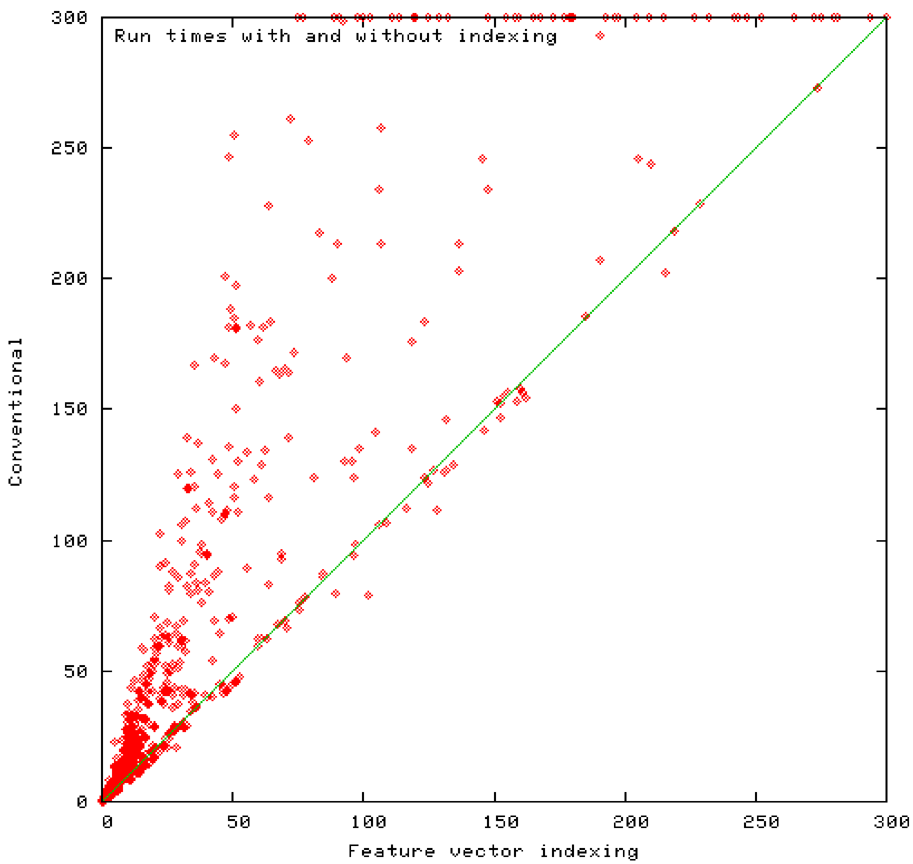
- Algorithm: At each node, only follow branches with larger or equal feature values



- Result: Just one subsumption candidate for $p(f(a))$

Performance 1

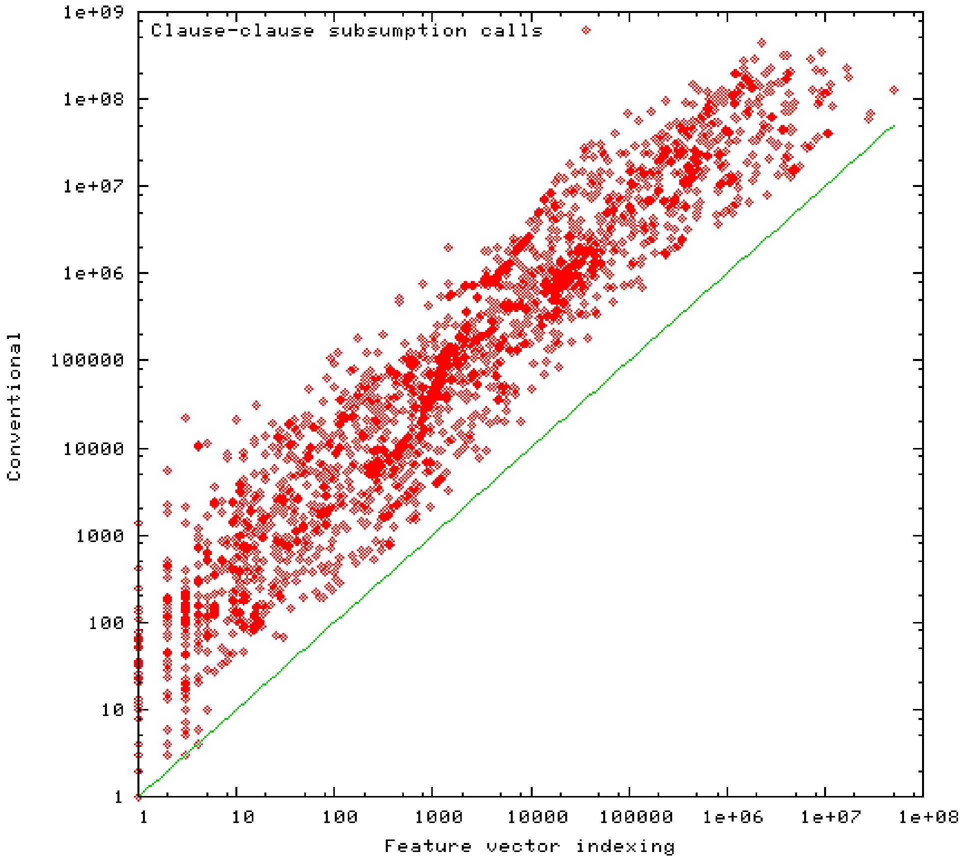
- Tested on 5180 examples from TPTP 2.5.1
  - Subsumption-heavy search strategy (contextual literal cutting)
  - Max. 75 features, 300MHz SUN Ultra 60, 300s time limit



- Speedup ca. 40%, overhead usually insignificant, 2717 vs. 2671 solutions found

Performance 2

- Number of subsumption attempts (notice double log scale)



- Average reduction: $1:60$, max: $1:8000 (1:\infty)$

Literature on Indexing

- Overview: [Gra95, SRV01]

- Classic paper: [McC92]

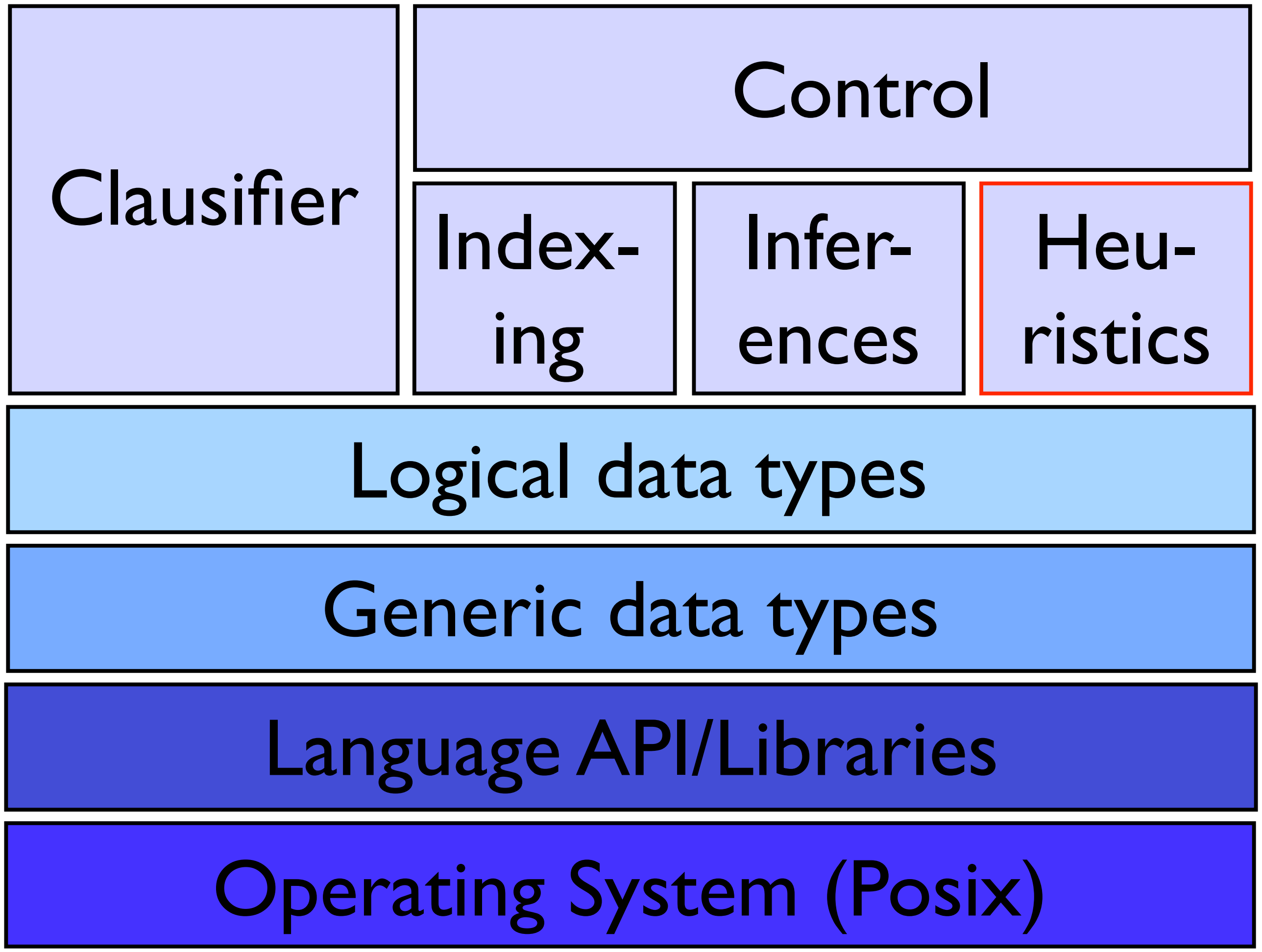
- Comparisons (for rewriting): [NHRV01]

- Feature vector indexing: [Sch04a]

Excercise: Unification

- E's unification code is SubstComputeMgu() in `E/TERMS/cte_match_mgu_1-1.[hc]`
  - Read and understand the code
  - Unification is broken down into subtassk
  - Subtasks are stored in a particular order
  - Why? Experiment with different orders!

Layered Architecture



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

- Important choice points for E:
  - Simplification ordering
  - Clause selection
  - Literal selection

- Other choice points:
  - Choice of rewrite relation (usually strongest, don't care which normal form)
  - Application of rewrite relation to terms (leftmost-innermost, strongly suggested by shared terms)

Simplification Orderings

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

Clause selection

- Most important choice point (?)
- Probably also hardest chocice (find best clause among millions)
- Implementation in E: Multiple priority queues sorted by heuristic evaluation and strategy-defined priority
- Selection in weighted round-robin-scheme (generalizes pick-given ratio)
- Example: 8*Refinedweight(PreferGoals,1,2,2,3,0.8),
  8*Refinedweight(PreferNonGoals,2,1,2,3,0.8),
  1*Clauseweight(ConstPrio,1,1,0.7),
  1*FIFOWeight(ByNegLitDist)
- Big win: Goal directed search
  – Symbols in the goal have low (=good) weights
  – Other symbols have increasingly large weight based on linking distance

Literal Selection

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

Literature on other Systems

- Real (saturating) provers: [LH02, RV02, Sch02, Wei01, WSH+07, Sti92, Sti89, LS01b]

- Significant alternative approaches:
  - DCTP [SL01, LS01a, LS02],
  - Model elimination: SETHEO [LSBB92, MIL+97], leanCOP [OB03, Ott08]
  - Instantiation-Based Reasoning: iProver: [Kor08, Kor09]
  - Model Evolution: Darwin [BFT06]

References

[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.

[BFT06] Peter Baumgartner, Alexander Fuchs, and Cesare Tinelli. Implementing the Model Evolution Calculus. *International Journal of Artificial Intelligence Tools*, 15(1):21–52, 2006.

[BG90] L. Bachmair and H. 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.

[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.

[BG98] L. Bachmair and H. Ganzinger. Equational Reasoning in Saturation-Based Theorem Proving. In W. Bibel and P.H. Schmitt, editors, *Automated Deduction — A Basis for Applications*, volume 9 (1) of *Applied Logic Series*, chapter 11, pages 353–397. Kluwer Academic Publishers, 1998.

[BH96] A. Buch and Th. Hillenbrand. Waldmeister: Development of a high performance completion-based theorem prover. SEKI-Report SR-96-01, Fachbereich Informatik, Universität Kaiserslautern, 1996. Available at `http://agent.informatik.uni-kl.de/waldmeister/`.

[DP60] M. Davis and H. Putnam. A Computing Procedure for Quantification Theory. *Journal of the ACM*, 7(1):215–215, 1960.

[E.S08] Mark E.Stickel. SNARK - SRI's New Automated Reasoning Kit. `http://www.ai.sri.com/~stickel/snark.html`, 2008. (acccessed 2009-10-04).

[GHLS03] J.M. Gaillourdet, Th. Hillenbrand, B. Löchner, and H. Spies. The New Waldmeister Loop At Work. In F. Bader, editor, *Proc. of the 19th CADE, Miami*, volume 2741 of *LNAI*, pages 317–321. Springer, 2003.

[Gra95] P. Graf. *Term Indexing*, volume 1053 of *LNAI*. Springer, 1995.

[HR87] J. Hsiang and M. Rusinowitch. On Word Problems in Equational Theories. In *Proc. of the 14th ICALP, Karlsruhe*, volume 267 of *LNCS*, pages 54–71. Springer, 1987.

[KB70] D.E. Knuth and P.B. Bendix. Simple Word Problems in Universal Algebras. In J. Leech, editor, *Computational Algebra*, pages 263–297. Pergamon Press, 1970.

[Kor08] Konstantin Korovin. iProver - An Instantiation-Based Theorem Prover for First-Order Logic (System Description). In A. Armando, P. Baumgartner, and G. Dowek, editors, *Proc. of the 4th IJCAR, Sydney*, volume 5195 of *LNAI*, pages 292–298. Springer, 2008.

[Kor09] Konstantin Korovin. An Invitation to Instantiation-Based Reasoning: From Theory to Practice. In *Volume in Memoriam of Harald Ganzinger*, LNCS. Springer, 2009. (to appear).

[LÖ6] Bernd Löchner. Things to Know when Implementing KBO. *Journal of Automated Reasoning*, 36(4):289–310, 2006.

[LH02] B. Löchner and Th. Hillenbrand. A Phytography of Waldmeister. *Journal of AI Communications*, 15(2/3):127–133, 2002.

[Löc06] Bernd Löchner. Things to Know When Implementing LPO. *International Journal on Artificial Intelligence Tools*, 15(1):53–80, 2006.

[Lov68] D.W. Loveland. Mechanical Theorem Proving by Model Elimination. *Journal of the ACM*, 15(2), 1968.

[Lov78] D.W. Loveland. *Automated Theorem Proving: A Logical Basis*. North Holland, Amsterdam, 1978.

[LS01a] R. Letz and G. Stenz. Proof and Model Generation with Disconnection Tableaux. In R. Nieuwenhuis and A. Voronkov, editors, *Proc. of the 8th LPAR, Havana*, volume 2250 of *LNAI*, pages 142–156. Springer, 2001.

[LS01b] Reinhold Letz and Gernot Stenz. Model Elimination and Connection Tableau Procedures. In A. Robinson and A. Voronkov, editors, *Handbook of automated reasoning*, volume II, chapter 28, pages 2015–2112. Elsevier Science and MIT Press, 2001.

[LS02] Reinhold Letz and Gernot Stenz. Integration of Equality Reasoning into the Disconnection Calculus. In Uwe Egly and Christian Fermüller, editors, *Proc. TABLEAUX'2002, Copenhagen, Denmark*, LNAI, pages 176–190. Springer, 2002.

[LSBB92] R. Letz, J. Schumann, S. Bayerl, and W. Bibel. SETHEO: A High-Performance Theorem Prover. *Journal of Automated Reasoning*, 1(8):183–212, 1992.

[McC92] W.W. McCune. Experiments with Discrimination-Tree Indexing and Path Indexing for Term Retrieval. *Journal of Automated Reasoning*, 9(2):147–167, 1992.

[McC08] William W. McCune. Prover9 and Mace4. `http://www.cs.unm.edu/~mccune/prover9/`, 2008. (acccessed 2009-10-04).

[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.

[NHRV01] R. Nieuwenhuis, Th. Hillenbrand, A. Riazanov, and A. Voronkov. On the Evaluation of Indexing Techniques for Theorem Proving. In R. Goré, A. Leitsch, and T. Nipkow, editors, *Proc. of the 1st IJCAR, Siena*, volume 2083 of *LNAI*, pages 257–271. Springer, 2001.

[NR92] R. Nieuwenhuis and A. Rubio. Theorem Proving with Ordering Constrained Clauses. In D. Kapur, editor, *Proc. of the 11th CADE, Saratoga Springs*, volume 607 of *LNAI*, pages 477–491. Springer, 1992.

[NR01] 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.

[NRV97] Robert Nieuwenhuis, José Miguel Rivero, and Miguel Ángel Vallejo. Dedam: A Kernel of Data Structures and Algorithms for Automated

Deduction with Equality Clauses. In W.W. McCune, editor, *Proc. of the 14th CADE, Townsville*, volume 1249 of *LNAI*, pages 49–52. Springer, 1997. Full version at `http://http://www.lsi.upc.es/~roberto/refs/cade1997.html`.

[NW01] 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.

[OB03] Jens Otten and Wolfgang Bibel. leanCoP: Lean Connection-Based Theorem Proving,. *Journal of Symbolic Computation*, 36:139–161, 2003.

[Ott08] Jens Otten. leanCoP 2.0 and ileanCoP 1.2: High Performance Lean Theorem Proving in Classical and Intuitionistic Logic. In A. Armando, P. Baumgartner, and G. Dowek, editors, *Proc. of the 4th IJCAR, Sydney*, volume 5195 of *LNAI*, pages 283–291. Springer, 2008.

[Pug90] William Pugh. Skip Lists: A Probabilistic Alternative to Balanced Trees. *Communications of the ACM*, 33(6):668–676, 1990. `ftp://ftp.cs.umd.edu/pub/skipLists/`.

[Rob65] J. A. Robinson. A Machine-Oriented Logic Based on the Resolution Principle. *Journal of the ACM*, 12(1):23–41, 1965.

[RV01] A. Riazanov and A. Voronkov. Splitting without Backtracking. In B. Nebel, editor, *Proc. of the 17th International Joint Conference on Artificial Intelligence (IJCAI-2001), Seattle*, volume 1, pages 611–617. Morgan Kaufmann, 2001.

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

[RV03] A. Riazanov and A. Voronkov. Limited resource strategy in resolution theorem proving. *Journal of Symbolic Computation*, 36(1–2):101–115, 2003.

[RW69] G. Robinson and L. Wos. Paramodulation and Theorem Proving in First-Order Theories with Equality. In B. Meltzer and D. Michie, editors, *Machine Intelligence 4*. Edinburgh University Press, 1969.

[Sch99] S. Schulz. System Abstract: E 0.3. In H. Ganzinger, editor, *Proc. of* the 16th CADE, Trento, volume 1632 of LNAI, pages 297–391. Springer, 1999.

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

[Sch04a] S. Schulz. Simple and Efficient Clause Subsumption with Feature Vector Indexing. In G. Sutcliffe, S. Schulz, and T. Tammet, editors, Proc. of the IJCAR-2004 Workshop on Empirically Successful First-Order Theorem Proving, Cork, Ireland, 2004.

[Sch04b] S. Schulz. System Description: E 0.81. In D. Basin and M. Rusinowitch, editors, Proc. of the 2nd IJCAR, Cork, Ireland, volume 3097 of LNAI, pages 223–228. Springer, 2004.

[Sch09] S. Schulz. The E Equational Theorem Prover – User Manual. `http://www.eprover.org`, 2009. (available with the E source distribution).

[SL01] G. Stenz and R. Letz. DCTP – A Disconnection Calculus Theorem Prover – System Abstract. In R. Goré, A. Leitsch, and T. Nipkow, editors, *Proc. of the 1st IJCAR, Siena*, volume 2083 of *LNAI*, pages 381–385. Springer, 2001.

[SRV01] 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.

[SSCG06] Geoff Sutcliffe, Stephan Schulz, Koen Claessen, and Allen Van Gelder. Using the TPTP Language for Writing Derivations and Finite Interpretations . In Ulrich Fuhrbach and Natarajan Shankar, editors, *Proc. of the 3rd IJCAR, Seattle*, volume 4130 of *LNAI*, pages 67–81, 4130, 2006. Springer.

[ST85] D.D. Sleator and R.E. Tarjan. Self-Adjusting Binary Search Trees. *Journal of the ACM*, 32(3):652–686, 1985.

[Sti89] Mark E. Stickel. A Prolog technology theorem prover: A new exposition and implementation in Prolog. Technical Note 464, Artificial Intelligence Center, SRI International, Menlo Park, California, June 1989.

[Sti92] Mark E. Stickel. A Prolog technology theorem prover: A new exposition and implementation in Prolog. *Theoretical Computer Science*, 104(1):109–128, 1992.

[Sut09] G. Sutcliffe. The TPTP Web Site. `http://www.tptp.org`, 2004–2009. (acccessed 2009-09-28).

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

[Vor95] A. Voronkov. The Anatomy of Vampire: Implementing Bottom-Up Procedures with Code Trees. *Journal of Automated Reasoning*, 15(2):238–265, 1995.

[Wei01] C. Weidenbach. SPASS: Combining Superposition, Sorts and Splitting. In A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*, volume II, chapter 27, pages 1965–2013. Elsevier Science and MIT Press, 2001.

[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.

[WSH+07] Christoph Weidenbach, Renate Schmidt, Thomas Hillenbrand, Dalibor Topić, and Rostislav Rusev. SPASS Version 3.0. In Frank Pfenning, editor, Proc. of the 21st CADE, Bremen, volume 4603 of LNAI, pages 514–520. Springer, 2007.