Automated Theorem Proving

Lecture 14: Efficient Saturation Procedures and Outlook

Prof. Dr. Jasmin Blanchette

based on slides by Dr. Uwe Waldmann

Winter Semester 2026/27

Part 6: Efficient Saturation Procedures

Problem:

Refutational completeness is nice in theory, but . . .

. . . it guarantees only that proofs will be found eventually,
not that they will be found quickly.

Even though orderings and selection functions reduce the number of possible inferences, the search space problem is enormous.

First-order provers look for a needle in a haystack:
It may be necessary to make some millions of inferences to find a proof that is only a few dozens of steps long.

Coping with Large Sets of Formulas

Consequently:

- We must deal with large sets of formulas.
- We must use efficient techniques to find formulas that can be used as partners in an inference.
- We must simplify/eliminate as many formulas as possible.
- We must use efficient techniques to check whether a formula can be simplified/eliminated.

Coping with Large Sets of Formulas

Note:

Often there are several competing implementation techniques.

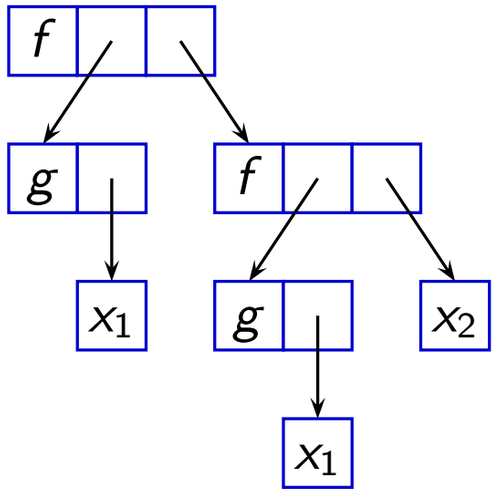
Design decisions depend on each other.

Design decisions depend on the particular class of problems we want to solve
(first-order logic without or with equality/unit equations,
size of the signature,
special algebraic properties like associativity and commutativity, etc.).

6.1 Term Representations

The obvious data structure for terms: Trees

$f(g(x_1), f(g(x_1), x_2))$

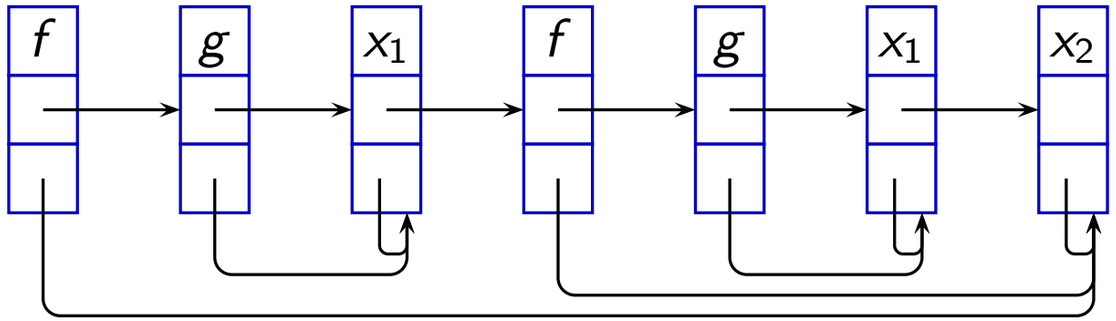


optionally: (full) sharing

Term Representations

An alternative: Flatterms

$f(g(x_1), f(g(x_1), x_2))$



need more memory;

but: better suited for preorder term traversal

and easier memory management.

6.2 Index Data Structures

Problem:

For a term t, we want to find all terms s such that

- s is an instance of t,
- s is a generalization of t (i.e., t is an instance of s),
- s and t are unifiable,
- s is a generalization of some subterm of t,
- ...

Index Data Structures

Requirements:

- fast insertion,
- fast deletion,
- fast retrieval,
- small memory consumption.

Index Data Structures

Many different approaches:

- Path indexing
- Discrimination trees
- Substitution trees
- Context trees
- Feature vector indexing
- ...

Index Data Structures

Perfect filtering:

The indexing technique returns exactly those terms satisfying the query.

Imperfect filtering:

The indexing technique returns some superset of the set of all terms satisfying the query.

Retrieval operations must be followed by an additional check,
but the index can often be implemented more efficiently.

Frequently: All occurrences of variables are treated as different variables.

Path Indexing

Path indexing:

Paths of terms are encoded in a trie ("retrieval tree").

A star $*$ represents arbitrary variables.

Example: Paths of $f(g(*, b), *)$: $f.1.g.1.*$

$f.1.g.2.b$

$f.2.*$

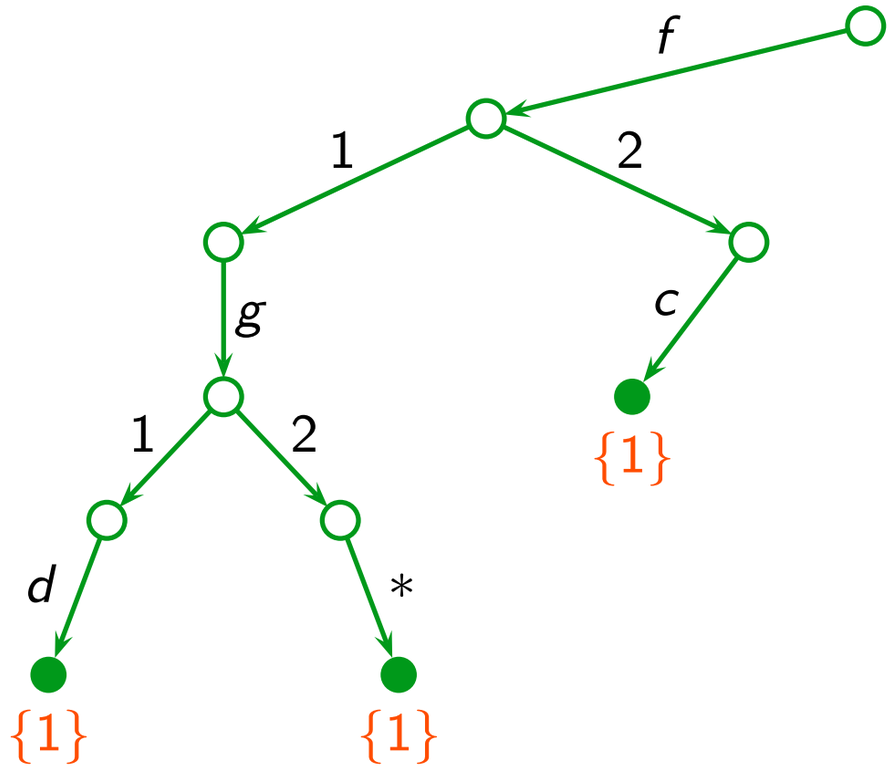
Each leaf of the trie contains the set of (pointers to) all terms that contain the respective path.

Path Indexing

Example: Path index for $\{f(g(d,*),c),\ g(b,h(c)),\ f(g(*,c),c),$
$f(b,g(c,b)),\ f(b,g(*,b)),\ f(*,c),\ f(*,g(c,b))\}$

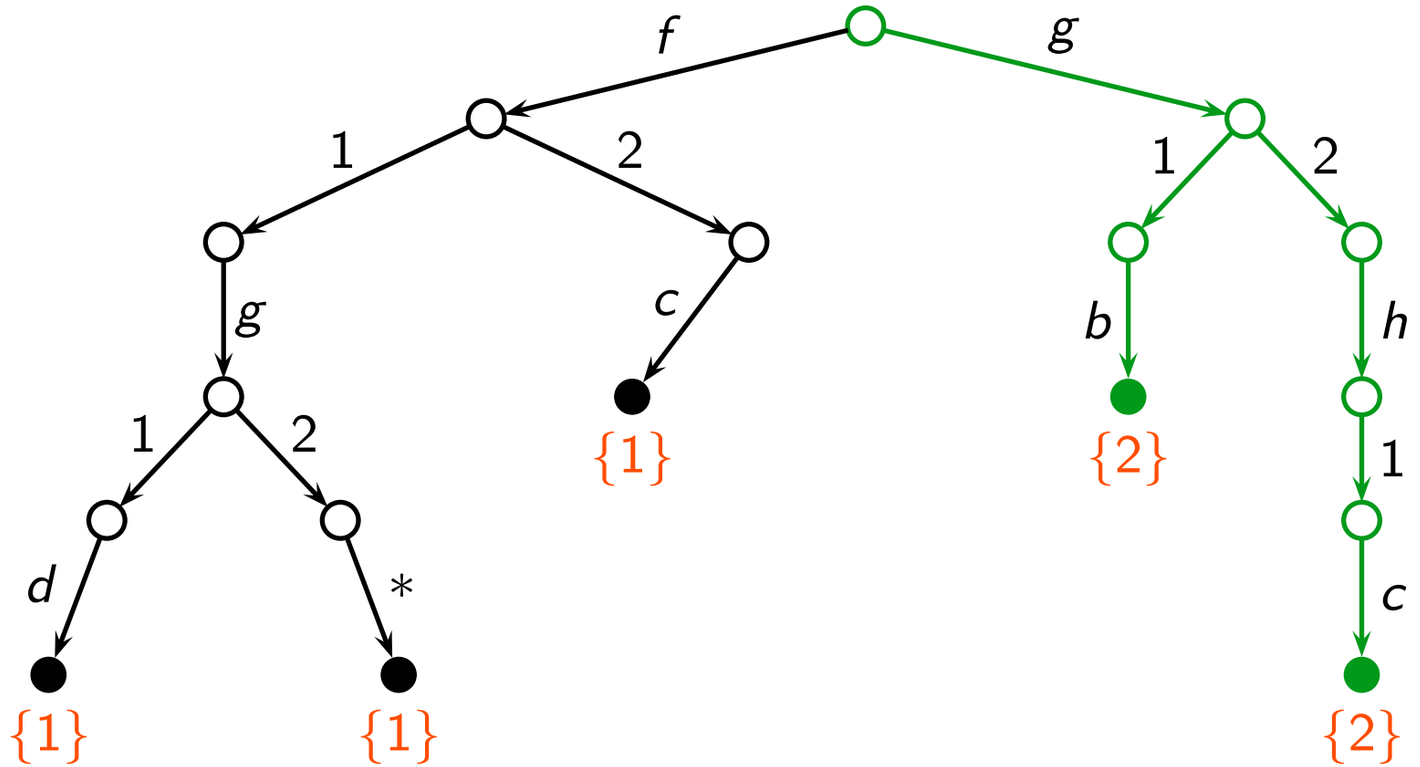
Path Indexing

Example: Path index for $\{f(g(d,*),c),\ g(b,h(c)),\ f(g(*,c),c),$
$f(b,g(c,b)),\ f(b,g(*,b)),\ f(*,c),\ f(*,g(c,b))\}$



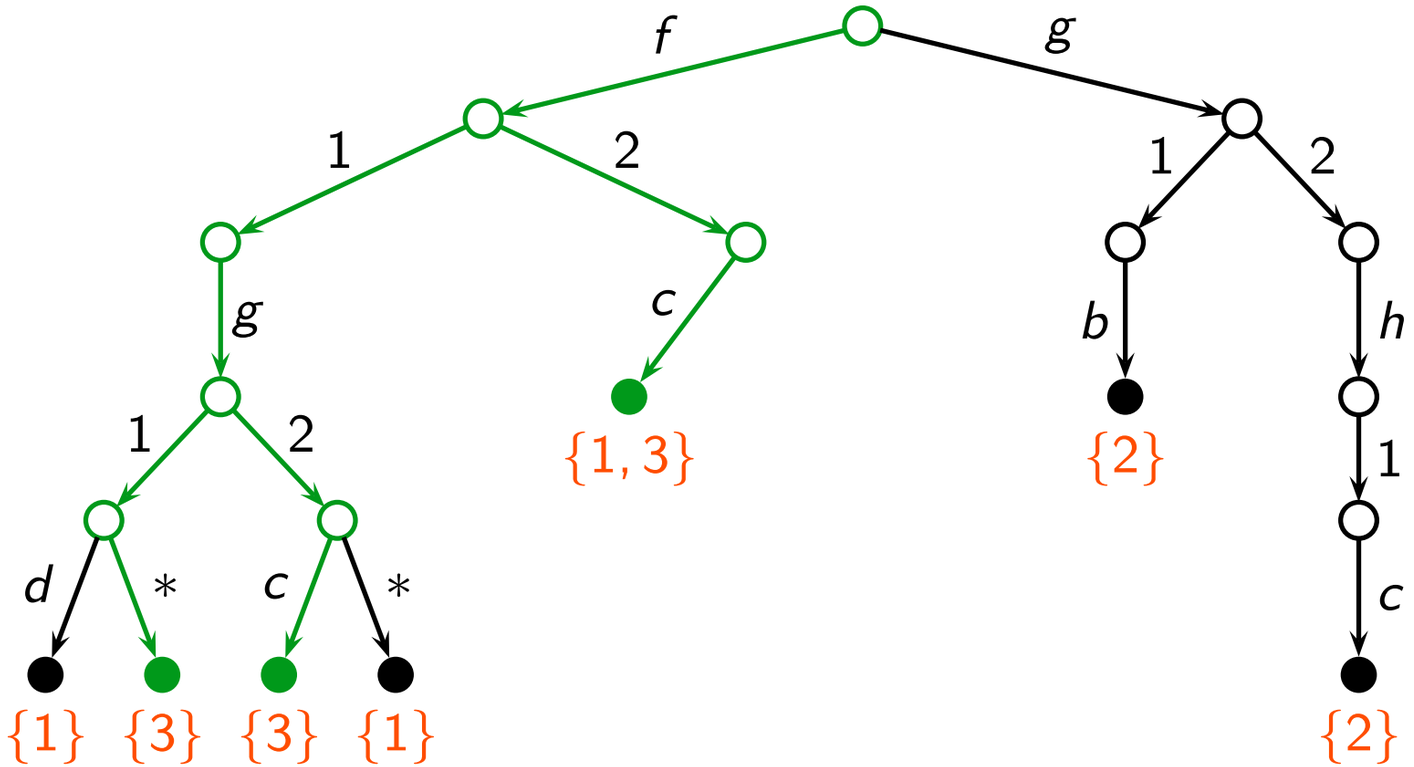
Path Indexing

Example: Path index for $\{f(g(d,*),c),\ g(b,h(c)),\ f(g(*,c),c),$ $f(b,g(c,b)),\ f(b,g(*,b)),\ f(*,c),\ f(*,g(c,b))\}$



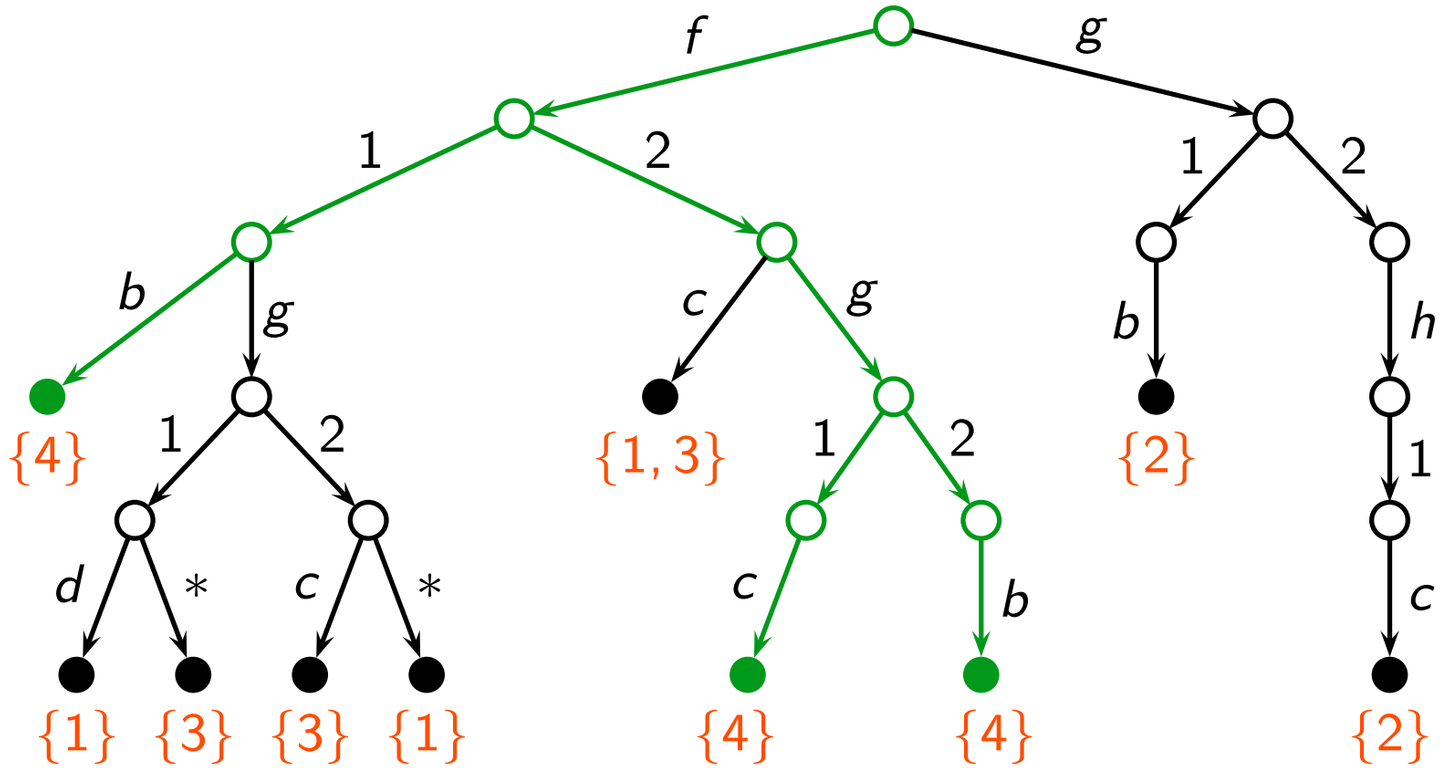
Path Indexing

Example: Path index for $\{f(g(d,*),c),\ g(b,h(c)),\ f(g(*,c),c),$ $f(b,g(c,b)),\ f(b,g(*,b)),\ f(*,c),\ f(*,g(c,b))\}$



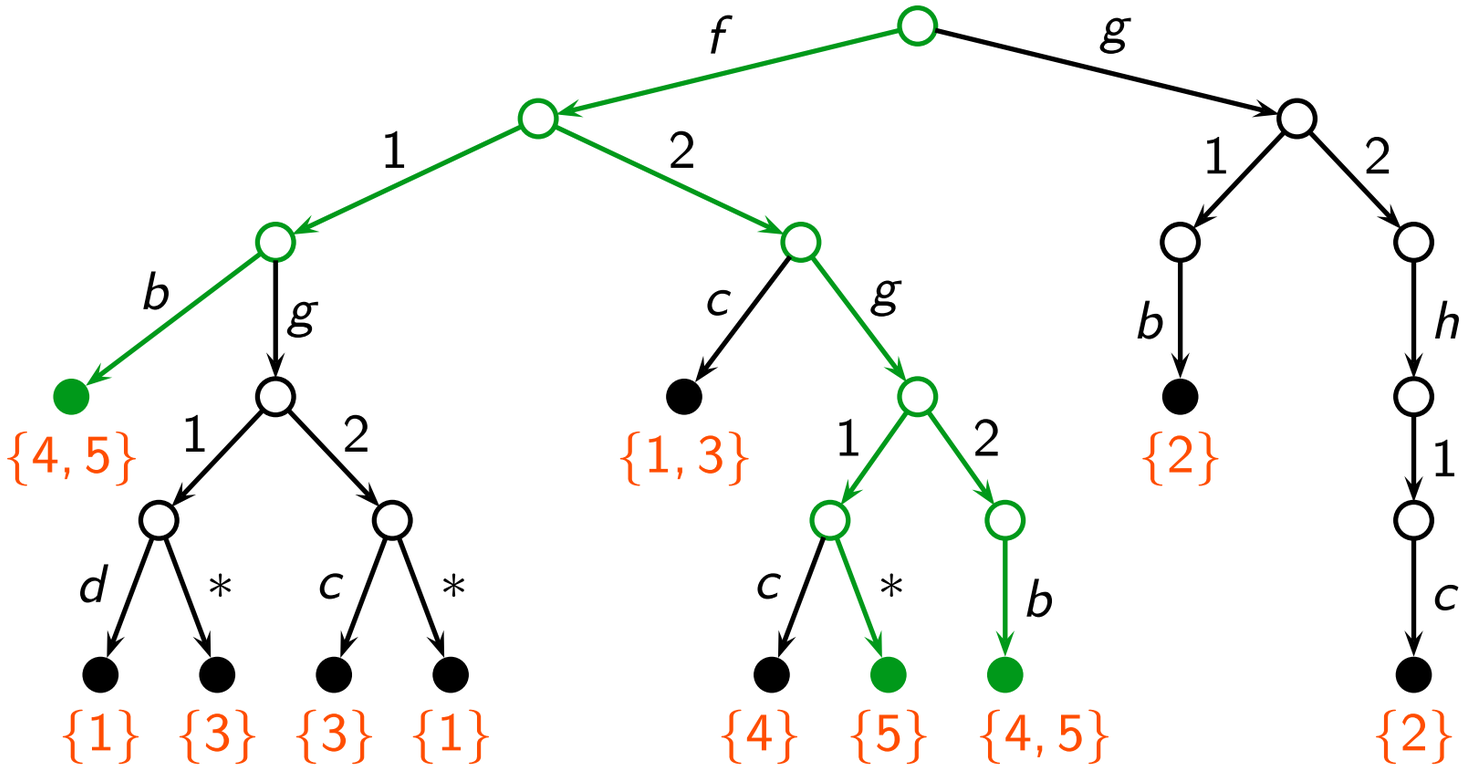
Path Indexing

Example: Path index for $\{f(g(d,*),c),\ g(b,h(c)),\ f(g(*,c),c),$ $f(b,g(c,b)),\ f(b,g(*,b)),\ f(*,c),\ f(*,g(c,b))\}$



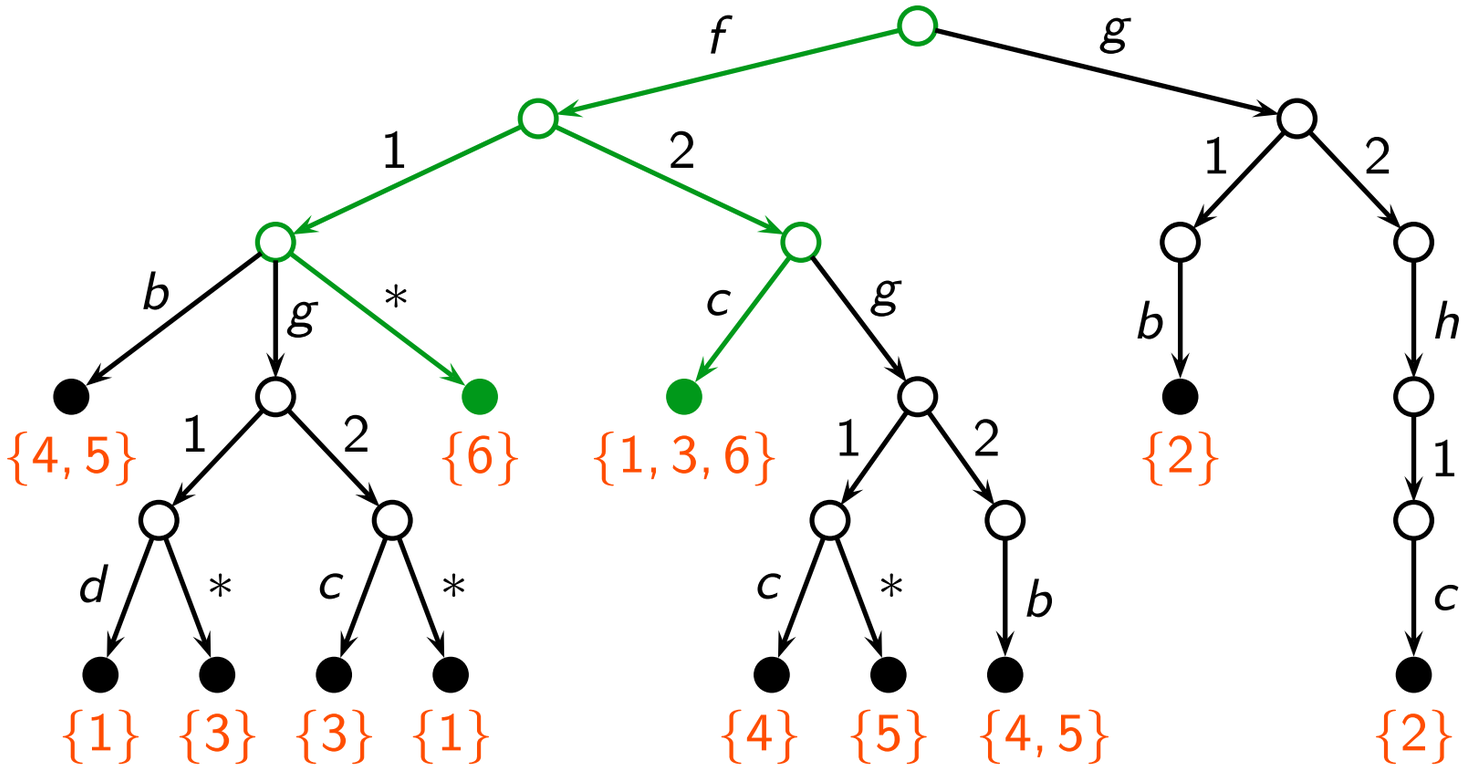
Path Indexing

Example: Path index for $\{f(g(d,*),c),\ g(b,h(c)),\ f(g(*,c),c),$
$f(b,g(c,b)),\ f(b,g(*,b)),\ f(*,c),\ f(*,g(c,b))\}$



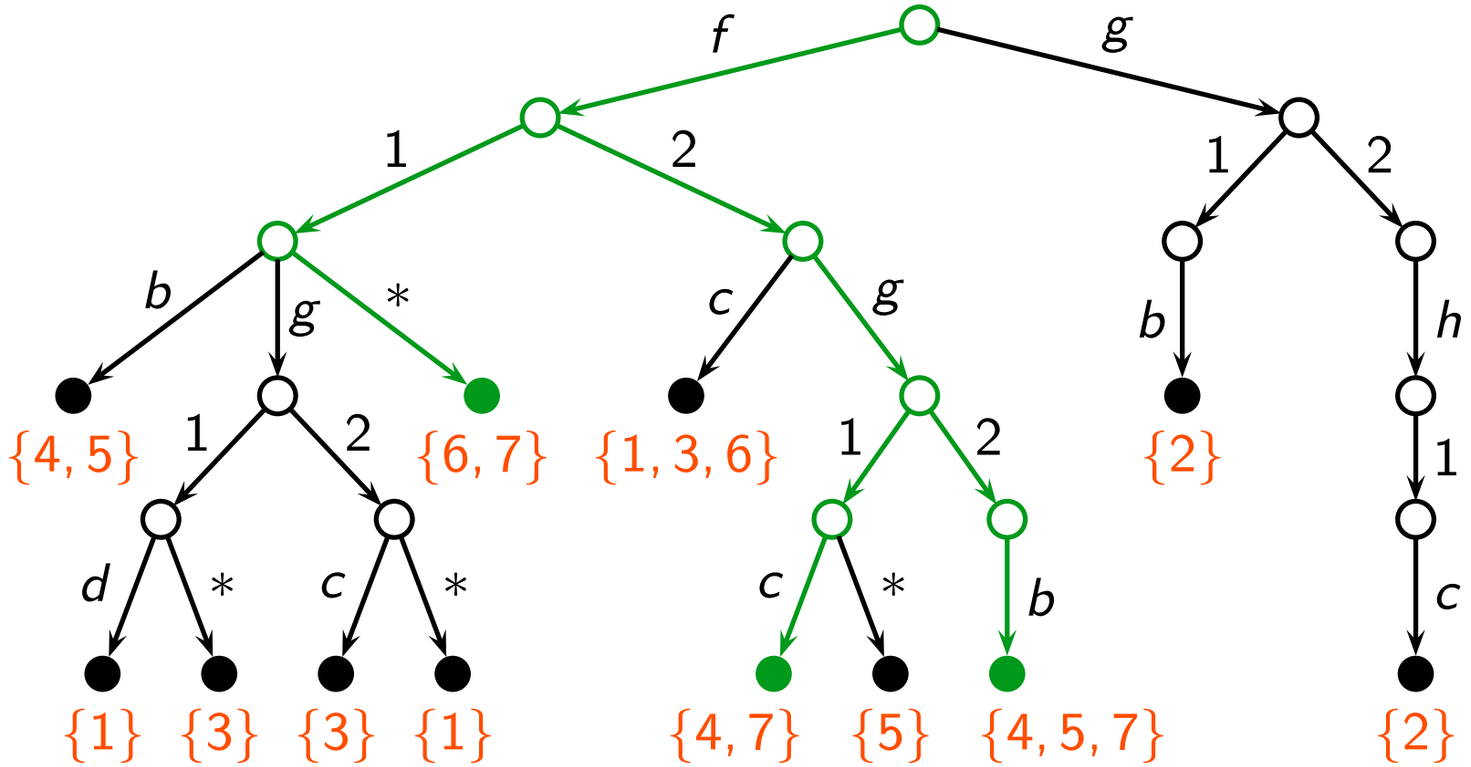
Path Indexing

Example: Path index for $\{f(g(d,*),c),\ g(b,h(c)),\ f(g(*,c),c),$
$f(b,g(c,b)),\ f(b,g(*,b)),\ f(*,c),\ f(*,g(c,b))\}$



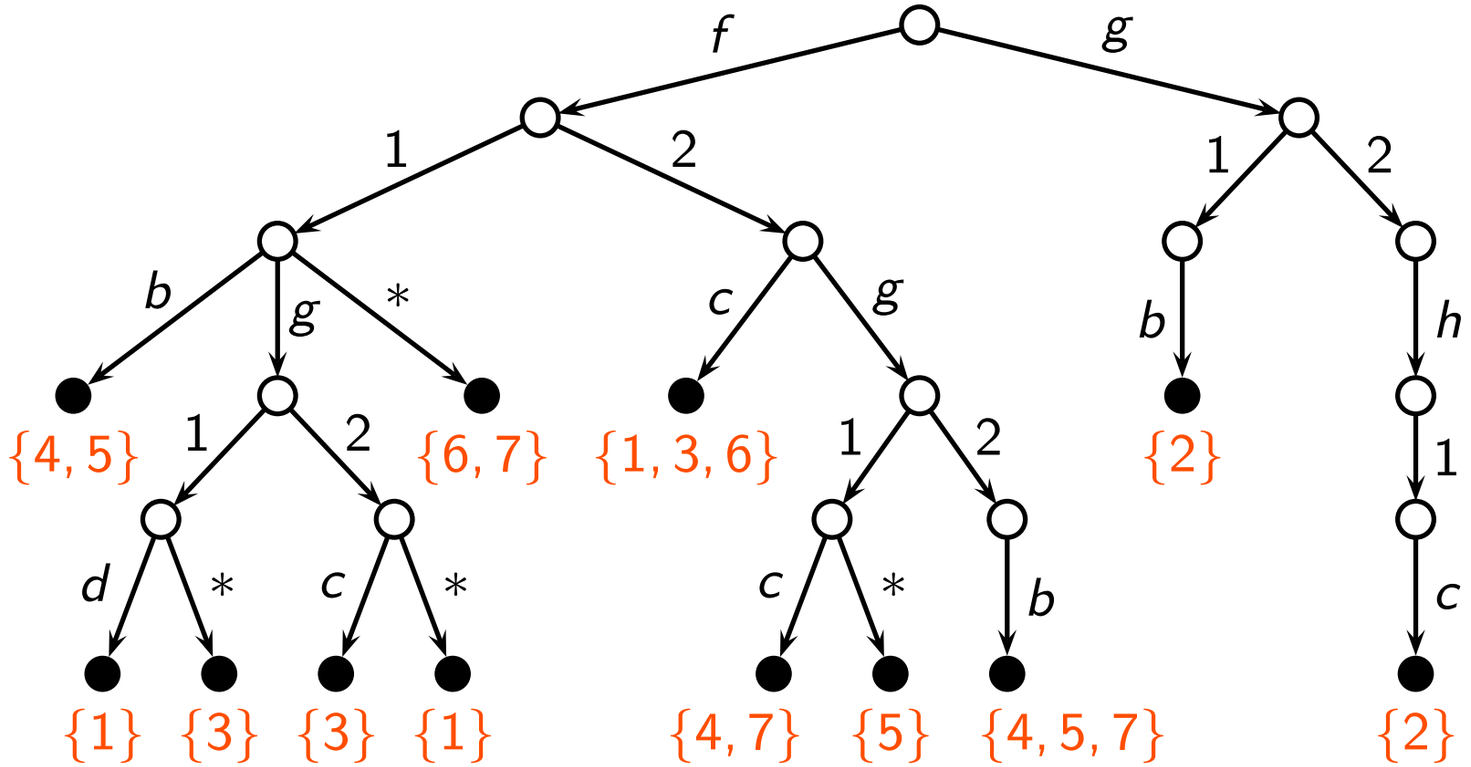
Path Indexing

Example: Path index for $\{f(g(d,*),c),\ g(b,h(c)),\ f(g(*,c),c),$ $f(b,g(c,b)),\ f(b,g(*,b)),\ f(*,c),\ f(*,g(c,b))\}$



Path Indexing

Example: Path index for $\{f(g(d,*),c),\ g(b,h(c)),\ f(g(*,c),c),$
$f(b,g(c,b)),\ f(b,g(*,b)),\ f(*,c),\ f(*,g(c,b))\}$



Path Indexing

Advantages:

- Uses little space.
- No backtracking for retrieval.
- Efficient insertion and deletion.
- Good for finding instances,
  also usable for finding generalizations.

Disadvantages:

- Retrieval requires combining intermediate results for all paths.

Discrimination Trees

Discrimination trees:

Preorder traversals of terms are encoded in a trie.

A star $*$ represents arbitrary variables.

Example: String of $f(g(*, b), *)$: $f.g.*.b.*$

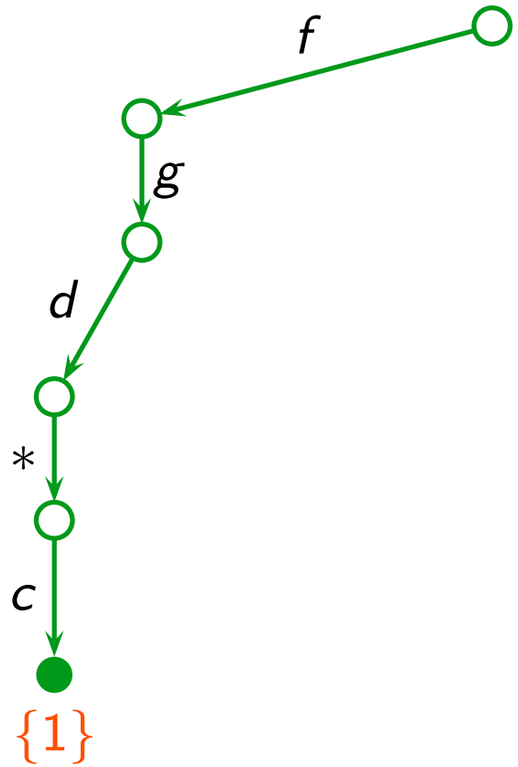
Each leaf of the trie contains (a pointer to) the term that is represented by the path.

Discrimination Trees

Example: Discrimination tree for $\{f(g(d,*),c),\ g(b,h(c)),\ f(g(*,c),c),$ $f(b,g(c,b)),\ f(b,g(*,b)),\ f(*,c),\ f(*,g(c,b))\}$

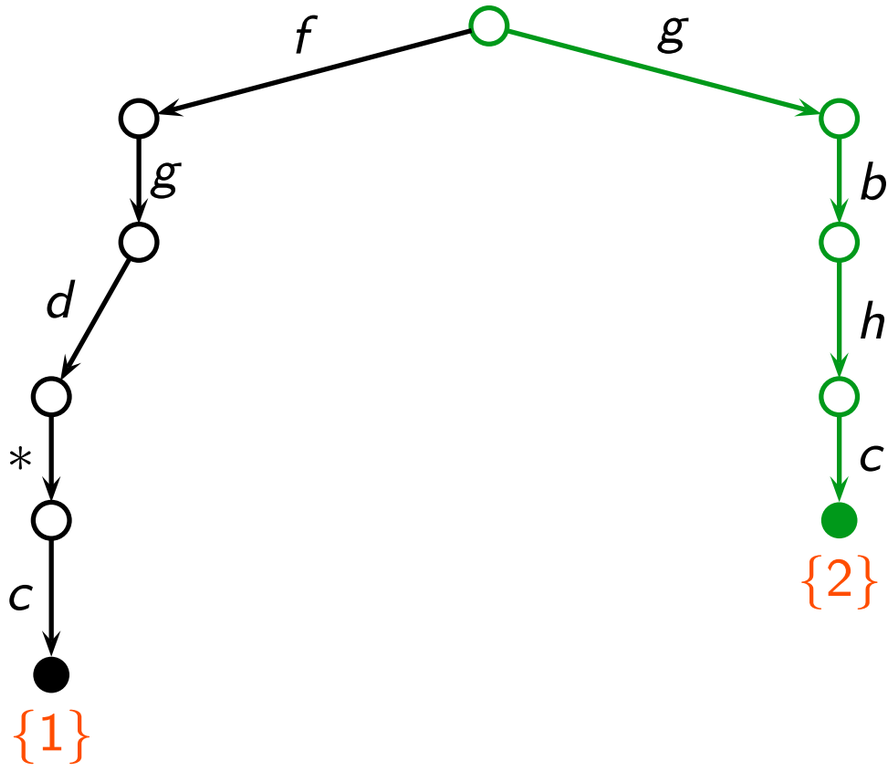
Discrimination Trees

Example: Discrimination tree for $\{f(g(d,*),c),\ g(b,h(c)),\ f(g(*,c),c),\ f(b,g(c,b)),\ f(b,g(*,b)),\ f(*,c),\ f(*,g(c,b))\}$



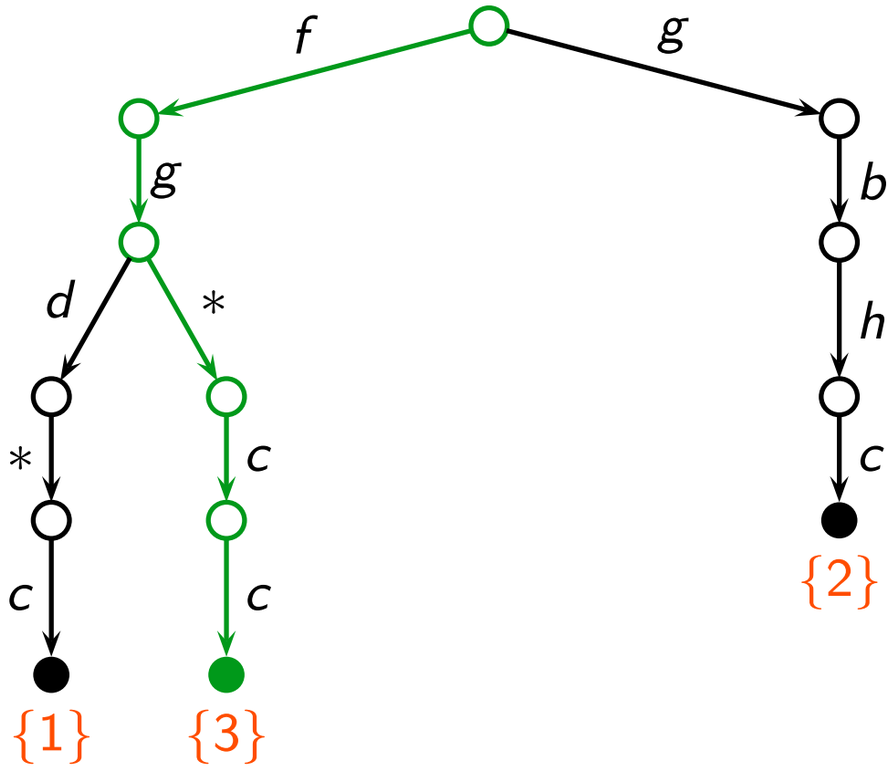
Discrimination Trees

Example: Discrimination tree for $\{f(g(d,*),c),\ g(b,h(c)),\ f(g(*,c),c),$ $f(b,g(c,b)),\ f(b,g(*,b)),\ f(*,c),\ f(*,g(c,b))\}$



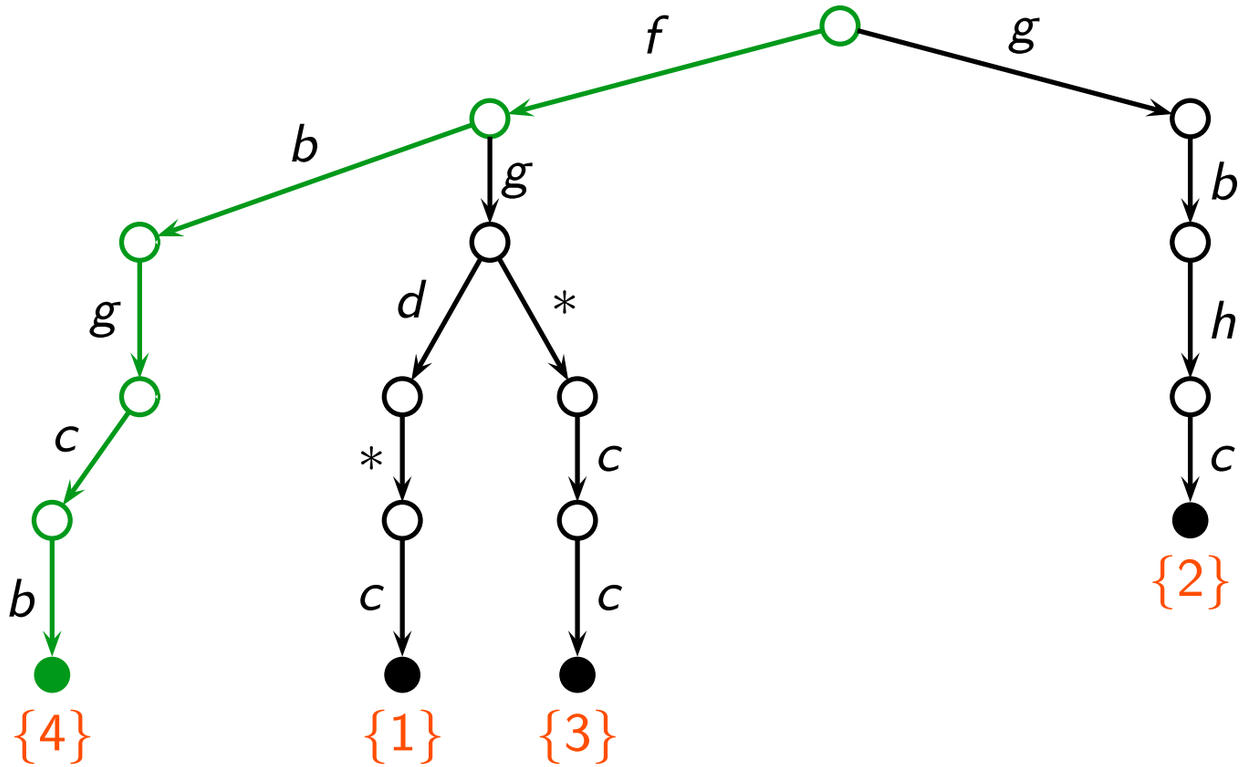
Discrimination Trees

Example: Discrimination tree for $\{f(g(d,*),c),\ g(b,h(c)),\ f(g(*,c),c),$ $f(b,g(c,b)),\ f(b,g(*,b)),\ f(*,c),\ f(*,g(c,b))\}$



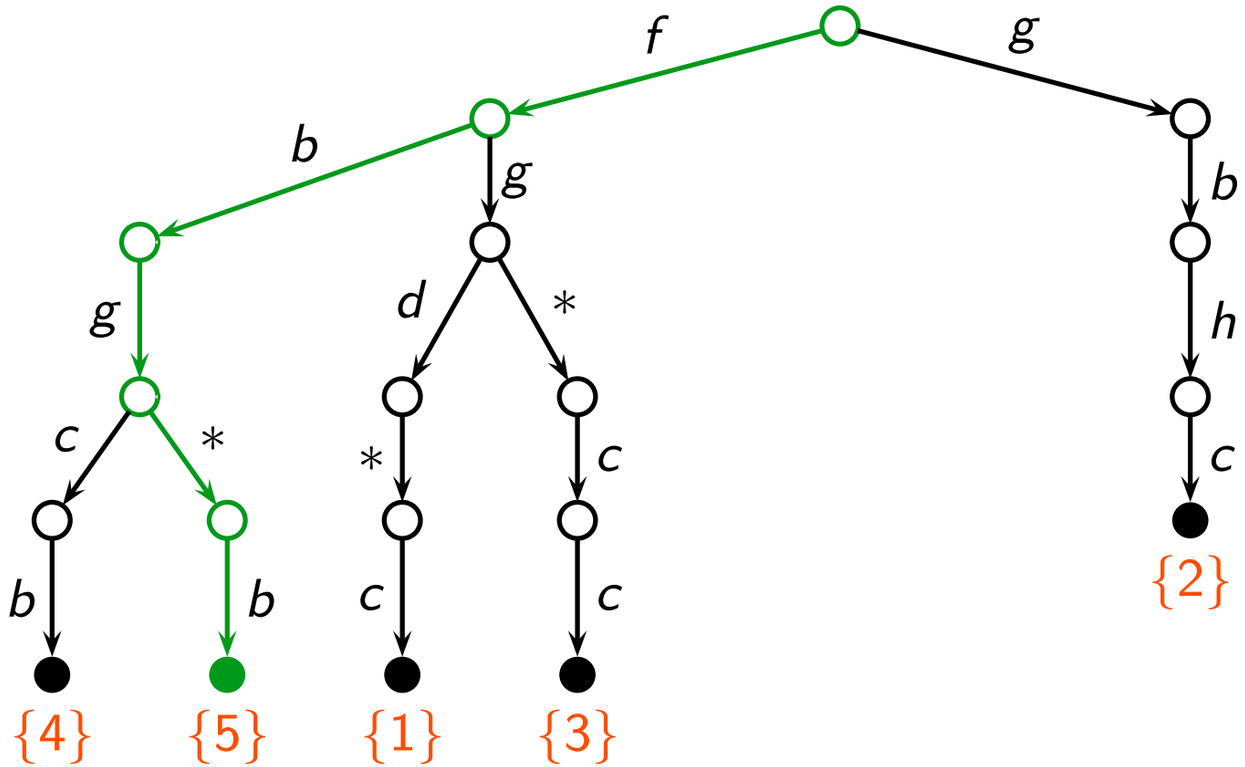
Discrimination Trees

Example: Discrimination tree for $\{f(g(d,*),c),\ g(b,h(c)),\ f(g(*,c),c),$ $f(b,g(c,b)),\ f(b,g(*,b)),\ f(*,c),\ f(*,g(c,b))\}$



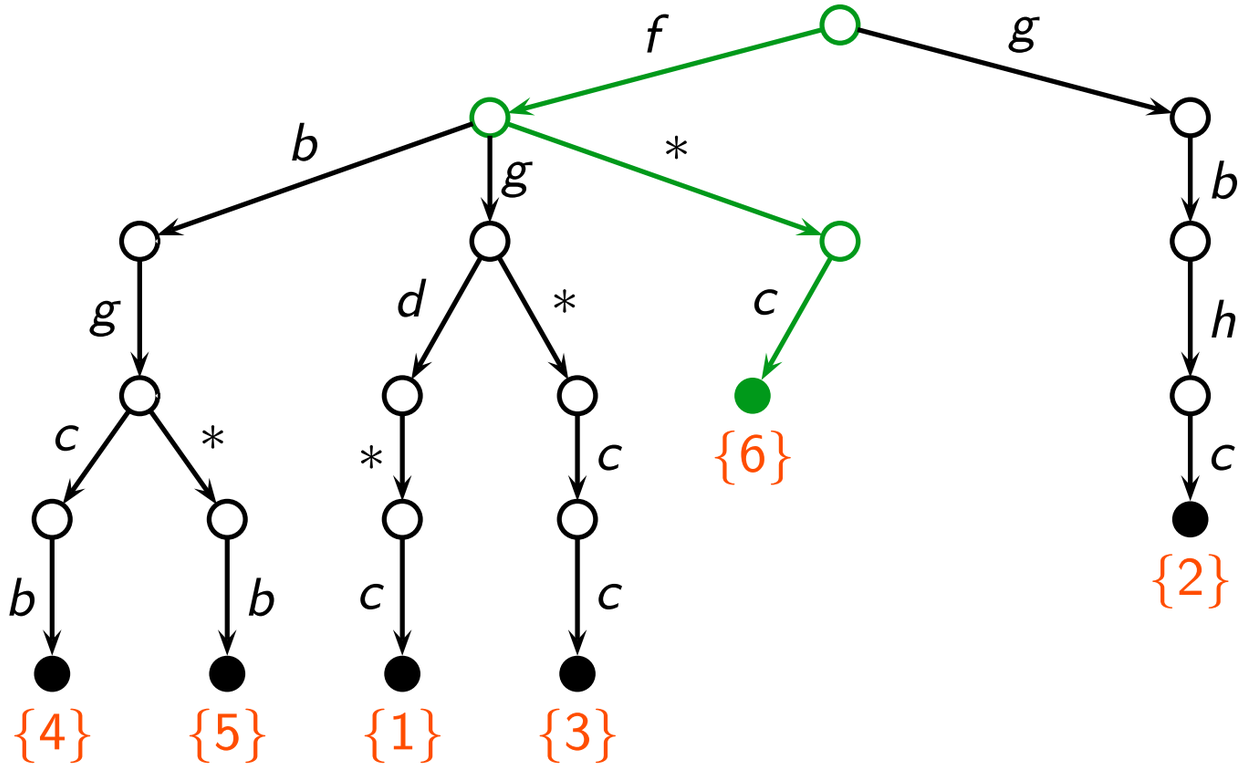
Discrimination Trees

Example: Discrimination tree for $\{f(g(d,*),c),\ g(b,h(c)),\ f(g(*,c),c),$ $f(b,g(c,b)),\ f(b,g(*,b)),\ f(*,c),\ f(*,g(c,b))\}$



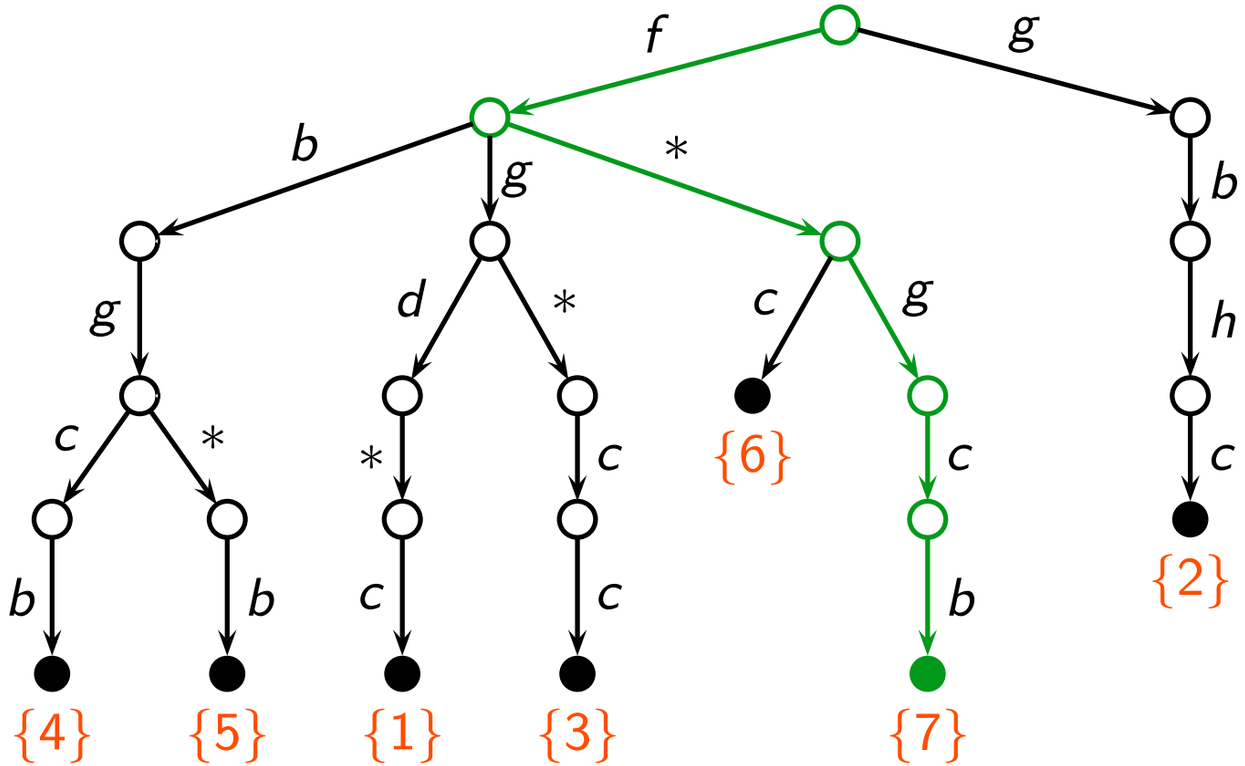
Discrimination Trees

Example: Discrimination tree for $\{f(g(d,*),c),\ g(b,h(c)),\ f(g(*,c),c),$ $f(b,g(c,b)),\ f(b,g(*,b)),\ f(*,c),\ f(*,g(c,b))\}$



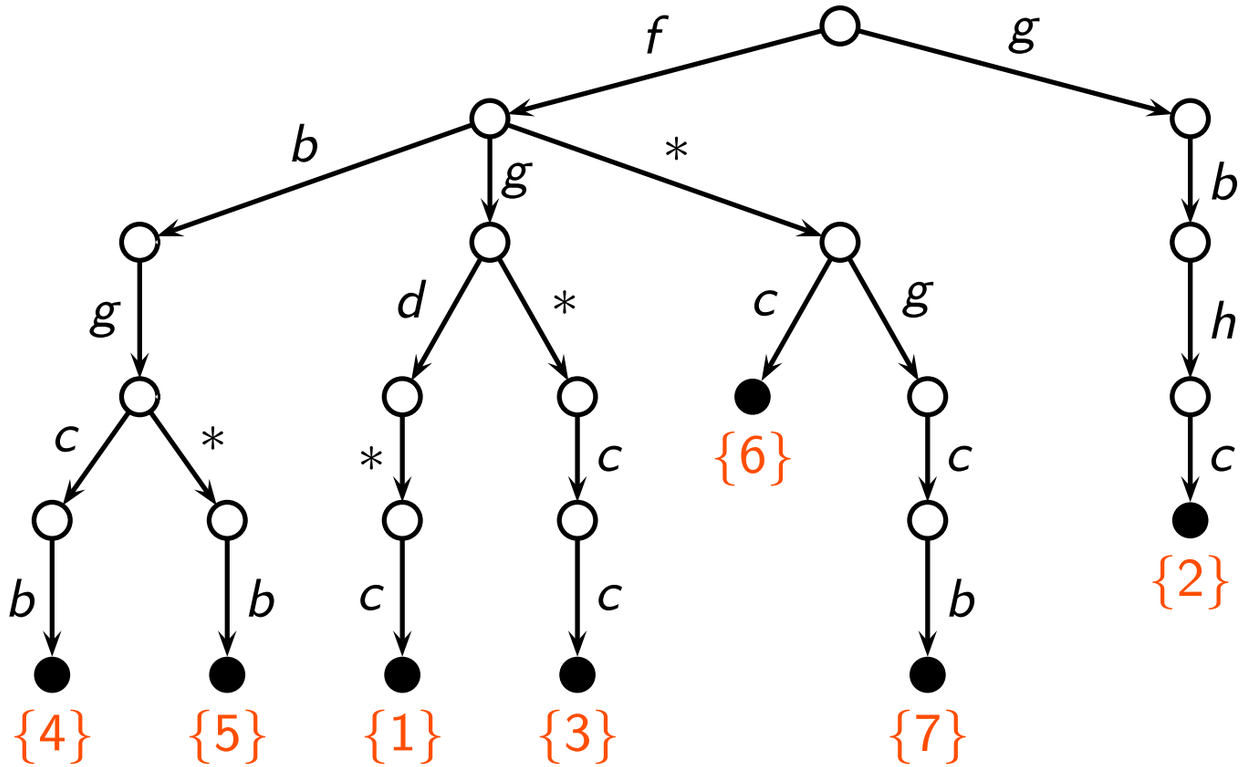
Discrimination Trees

Example: Discrimination tree for $\{f(g(d,*),c),\ g(b,h(c)),\ f(g(*,c),c),$ $f(b,g(c,b)),\ f(b,g(*,b)),\ f(*,c),\ f(*,g(c,b))\}$



Discrimination Trees

Example: Discrimination tree for $\{f(g(d,*),c),\ g(b,h(c)),\ f(g(*,c),c),$ $f(b,g(c,b)),\ f(b,g(*,b)),\ f(*,c),\ f(*,g(c,b))\}$



Discrimination Trees

Advantages:

Each leaf yields one term, hence retrieval does not require intersections of intermediate results for all paths.

Good for finding generalizations,
not so good for finding instances.

Disadvantage:

Uses more storage than path indexing (due to less sharing).

Feature Vector Indexing

Goal:

C' is subsumed by C if $C' = C\sigma \vee D$.

Find all clauses C' for a given C or vice versa.

Feature Vector Indexing

If C' is subsumed by C, then

- C' contains at least as many literals as C.
- C' contains at least as many positive literals as C.
- C' contains at least as many negative literals as C.
- C' contains at least as many function symbols as C.
- C' contains at least as many occurrences of f as C.
- C' contains at least as many occurrences of f in negative literals as C.
- the deepest occurrence of f in C' is at least as deep as in C.
- ...

Feature Vector Indexing

Idea:

Select a list of these "features."

Compute the "feature vector" (a list of natural numbers)
for each clause and store it in a trie.

When searching for a subsuming clause:
Traverse the trie, check all clauses for which all features are smaller or equal. (Stop if a subsuming clause is found.)

When searching for subsumed clauses:
Traverse the trie, check all clauses for which all features are larger or equal.

Feature Vector Indexing

Advantages:

- Works on the clause level, rather than on the term level.
- Specialized for subsumption testing.

Disadvantages:

- Needs to be complemented by other index structure for other operations.

Literature

R. Sekar, I. V. Ramakrishnan, and Andrei Voronkov:
Term Indexing, Ch. 26 in Robinson and Voronkov (eds.), *Handbook of Automated Reasoning, Vol. II*, Elsevier, 2001.

Stephan Schulz:
Simple and Efficient Clause Subsumption with Feature Vector Indexing, in Bonacina and Stickel (eds.), *Automated Reasoning and Mathematics*, LNCS 7788, Springer, 2013.

Christoph Weidenbach:
Combining Superposition, Sorts and Splitting, Ch. 27 in Robinson and Voronkov (eds.), *Handbook of Automated Reasoning, Vol. II*, Elsevier, 2001.

Part 7: Outlook

Further topics in automated reasoning.

7.1 Satisfiability Modulo Theories (SMT)

DPLL and CDCL check satisfiability of propositional formulas.

DPLL and CDCL can also be used for ground first-order formulas without equality:

- Ground first-order atoms are treated like propositional variables.
- Truth values of $P(b), Q(b), Q(f(b))$ are independent.

Satisfiability Modulo Theories (SMT)

For ground formulas with equality, independence does not hold:

If $b \approx c$ is true, then $f(b) \approx f(c)$ must also be true.

Similarly for other theories, e.g. linear arithmetic:
$b > 5$ implies $b > 3$.

We can still use CDCL, but we must combine it with a decision procedure for the theory part T:

$M \models_T C$ if and only if M and the theory axioms T entail C.

7.2 Sorted Logics

So far, we have considered only unsorted first-order logic.

In practice, one often considers many-sorted logics:

$read/2$ becomes $read : array \times nat \to data$.

$write/3$ becomes $write : array \times nat \times data \to array$.

Variables: $x : data$

Only one declaration per function/predicate/variable symbol.

All terms, atoms, substitutions must be well-sorted.

Sorted Logics

Algebras:

Instead of universe $U_{\mathcal{A}}$, one set per sort: $array_{\mathcal{A}}$, $nat_{\mathcal{A}}$.

Interpretations of function and predicate symbols correspond to their declarations:

$read_{\mathcal{A}} : array_{\mathcal{A}} \times nat_{\mathcal{A}} \to data_{\mathcal{A}}$

Sorted Logics

Proof theory, calculi, etc.:

Essentially as in the unsorted case.

More difficult:

Subsorts

Overloading

7.3 Splitting

Tableau-like rule within resolution to eliminate variable-disjoint (positive) disjunctions:

$$\frac{N \cup \{C_1 \vee C_2\}}{N \cup \{C_1\} \;\mid\; N \cup \{C_2\}}$$

if $\mathrm{var}(C_1) \cap \mathrm{var}(C_2) = \emptyset$.

Split clauses are smaller and more likely to be usable for simplification.

The splitting tree is explored using intelligent backtracking.

Splitting

Improvement:

Use a SAT solver to manage the selection of split clauses.

$\Rightarrow$ AVATAR.

7.4 Higher-Order Logics

What changes if we switch to higher-order logics?

Applied variables: $x\ b$.

Partially applied functions: *times* z.

Lambda-expressions with $\alpha\beta\eta$-conversion:
$(\lambda x.\, f\ (x\ b)\ c)\ (\lambda y.\, d) = f\ d\ c$.

Embedded booleans: $(\lambda x.\, \text{if } x \text{ then } f \text{ else } g)\ (p \vee q)$

Higher-Order Logics

Problems:

Orderings cannot have all desired compatibility properties.
$\Rightarrow$ additional inferences

Most general unifiers need not exist anymore.
$\Rightarrow$ interleave enumeration of unifiers and computation of inferences

CNF transformation by preprocessing is no longer sufficient.
$\Rightarrow$ need calculus with embedded clausification