4.6 Unfailing Completion

Classical completion:

Try to transform a set E of equations into an equivalent convergent TRS.

Fail, if an equation can neither be oriented nor deleted.

Unfailing completion (Bachmair, Dershowitz and Plaisted):

If an equation cannot be oriented, we can still use *orientable instances* for rewriting.

Note: If $\succ$ is total on ground terms, then every *ground instance* of an equation is trivial or can be oriented.

Goal: Derive a *ground convergent* set of equations.

Unfailing Completion

Let E be a set of equations, let $\succ$ be a reduction ordering.

We define the relation $\rightarrow_{E\succ}$ by

$$s \rightarrow_{E\succ} t \quad \text{iff} \quad \begin{array}{l} \text{there exist } (u \approx v) \in E \text{ or } (v \approx u) \in E, \\ p \in \mathrm{pos}(s), \text{ and } \sigma : X \rightarrow \mathrm{T}_\Sigma(X), \\ \text{such that } s|_p = u\sigma \text{ and } t = s[v\sigma]_p \\ \text{and } u\sigma \succ v\sigma. \end{array}$$

Note: $\rightarrow_{E\succ}$ is terminating by construction.

Unfailing Completion

From now on let $\succ$ be a reduction ordering that is total on ground terms.

E is called ground convergent w. r. t. $\succ$, if for all ground terms s and t with $s \leftrightarrow_E^* t$ there exists a ground term v such that $s \rightarrow_{E\succ}^* v \;{}_{E\succ}^*\!\leftarrow t$.

(Analogously for $E \cup R$.)

Unfailing Completion

As for standard completion, we establish ground convergence by computing critical pairs.

However, the ordering $\succ$ is not total on non-ground terms.

Since $s\theta \succ t\theta$ implies $s \not\preceq t$, we approximate $\succ$ on ground terms by $\not\preceq$ on arbitrary terms.

Unfailing Completion

Let $u_i \mathrel{\dot{\approx}} v_i$ $(i = 1, 2)$ be equations in E whose variables have been renamed such that $\mathrm{vars}(u_1 \mathrel{\dot{\approx}} v_1) \cap \mathrm{vars}(u_2 \mathrel{\dot{\approx}} v_2) = \emptyset$. Let $p \in \mathrm{pos}(u_1)$ be a position such that $u_1|_p$ is not a variable, σ is an mgu of $u_1|_p$ and u_2, and $u_i\sigma \not\preceq v_i\sigma$ $(i = 1, 2)$. Then $\langle v_1\sigma, (u_1\sigma)[v_2\sigma]_p \rangle$ is called a semi-critical pair of E with respect to $\succ$.

The set of all semi-critical pairs of E is denoted by $\mathrm{SP}_\succ(E)$.

Semi-critical pairs of $E \cup R$ are defined analogously. If $\to_R \subseteq \succ$, then $\mathrm{CP}(R)$ and $\mathrm{SP}_\succ(R)$ agree.

Unfailing Completion

Note: In contrast to critical pairs, it may be necessary to consider overlaps of a rule with itself at the top.

For instance, if $E = \{f(x) \approx g(y)\}$, then $\langle g(y), g(y') \rangle$ is a non-trivial semi-critical pair.

Unfailing Completion

The *Deduce* rule takes now the following form:

Deduce

$$(E; R) \quad \Rightarrow_{UKBC} \quad (E \cup \{s \approx t\}; R)$$

if $\langle s, t\rangle \in \mathrm{SP}_{\succ}(E \cup R)$

The other rules are inherited from $\Rightarrow_{KBC}$. The fairness criterion for runs is replaced by

$$\mathrm{SP}_{\succ}(E_* \cup R_*) \subseteq E_\infty$$

(i. e., if every semi-critical pair between persisting rules or equations is computed at some step of the derivation).

Unfailing Completion

Analogously to Thm. 4.32 we obtain now the following theorem:

Theorem 4.33:
Let $(E_0; R_0) \Rightarrow_{UKBC} (E_1; R_1) \Rightarrow_{UKBC} (E_2; R_2) \Rightarrow_{UKBC} \ldots$ be a fair run; let $R_0 = \emptyset$. Then

(1) $E_* \cup R_*$ is equivalent to E_0, and

(2) $E_* \cup R_*$ is ground convergent.

Unfailing Completion

Moreover one can show that, whenever there exists a *reduced* convergent R such that $\approx_{E_0} = \downarrow_R$ and $\rightarrow_R \in \succ$, then for every fair *and simplifying* run $E_* = \emptyset$ and $R_* = R$ up to variable renaming.

Here R is called reduced, if for every $l \rightarrow r \in R$, both l and r are irreducible w. r. t. $R \setminus \{l \rightarrow r\}$. A run is called simplifying, if R_* is reduced, and for all equations $u \approx v \in E_*$, u and v are incomparable w. r. t. $\succ$ and irreducible w. r. t. R_*.

Unfailing Completion

Unfailing completion is refutationally complete for equational theories:

Theorem 4.34:
Let E be a set of equations, let $\succ$ be a reduction ordering that is total on ground terms. For any two terms s and t, let $\hat{s}$ and $\hat{t}$ be the terms obtained from s and t by replacing all variables by Skolem constants. Let $eq/2$, $true/0$ and $false/0$ be new operator symbols, such that $true$ and $false$ are smaller than all other terms. Let $E_0 = E \cup \{eq(\hat{s}, \hat{t}) \approx true,\ eq(x, x) \approx false\}$. If $(E_0; \emptyset) \Rightarrow_{UKBC} (E_1; R_1) \Rightarrow_{UKBC} (E_2; R_2) \Rightarrow_{UKBC} \ldots$ be a fair run of unfailing completion, then $s \approx_E t$ iff some $E_i \cup R_i$ contains $true \approx false$.

Unfailing Completion

Outlook:

Combine ordered resolution and unfailing completion to get a calculus for equational clauses:

compute inferences between (strictly) maximal literals as in ordered resolution,
compute overlaps between maximal sides of equations as in unfailing completion

$\Rightarrow$ Superposition calculus.

Part 5: Implementing 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 are not independent of each other.

Design decisions are not independent of the particular class of problems we want to solve. (FOL without equality/FOL with equality/unit equations, size of the signature, special algebraic properties like AC, etc.)

5.1 The Main Loop

Standard approach:

Select one clause ("Given clause").

Find many partner clauses that can be used in inferences together with the "given clause" using an appropriate index data structure.

Compute the conclusions of these inferences; add them to the set of clauses.

The Main Loop

Consequently: split the set of clauses into two subsets.

- *Wo* = "Worked-off" (or "active") clauses: Have already been selected as "given clause". (So all inferences between these clauses have already been computed.)
- *Us* = "Usable" (or "passive") clauses: Have not yet been selected as "given clause".

The Main Loop

During each iteration of the main loop:

Select a new given clause C from Us; $Us := Us \setminus \{C\}$.

Find partner clauses D_i from Wo; $New = Infer(\{ D_i \mid i \in I \}, C)$; $Us = Us \cup New$; $Wo = Wo \cup \{C\}$

The Main Loop

Additionally:

Try to simplify C using Wo. (Skip the remainder of the iteration, if C can be eliminated.)

Try to simplify (or even eliminate) clauses from Wo using C.

The Main Loop

Design decision: should one also simplify *Us* using *Wo*?

yes $\leadsto$ "Full Reduction":
Advantage: simplifications of *Us* may be useful to derive the empty clause.

no $\leadsto$ "Lazy Reduction":
Advantage: clauses in *Us* are really passive; only clauses in *Wo* have to be kept in index data structure. (Hence: can use index data structure for which retrieval is faster, even if update is slower and space consumption is higher.)

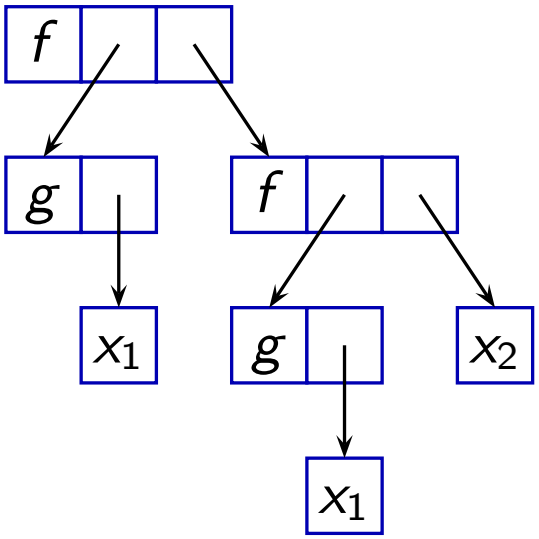
Main Loop Full Reduction

$Us = N$;
$Wo = \emptyset$;
while ($Us \neq \emptyset$ && $\bot \notin Us$) {
    Given = select clause from Us and move it from Us to Wo;
    New = all inferences between Given and Wo;
    Reduce New together with Wo and Us;
    $Us = Us \cup New$;}
if ($\bot \in Us$)
    return "unsatisfiable";
else
    return "satisfiable";

5.2 Term Representations

The obvious data structure for terms: Trees

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

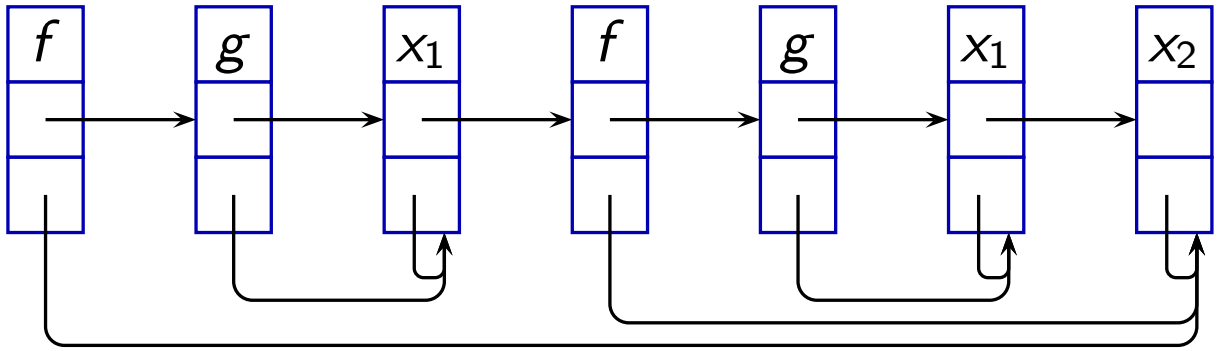


optionally: (full) sharing

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.

5.3 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.

Note: In applications like functional or logic programming, the requirements are different (insertion and deletion are much less important).

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 $f(g(*, b), *)$: $f.1.g.1.*$

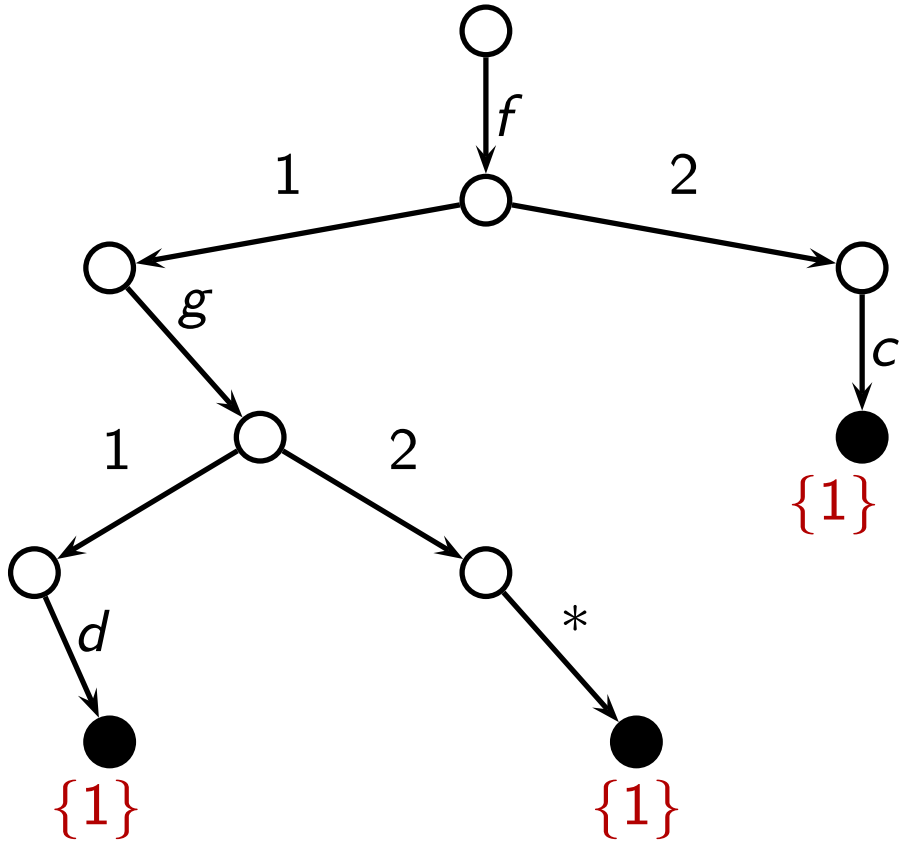
$f.1.g.2.b$

$f.2.*$

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

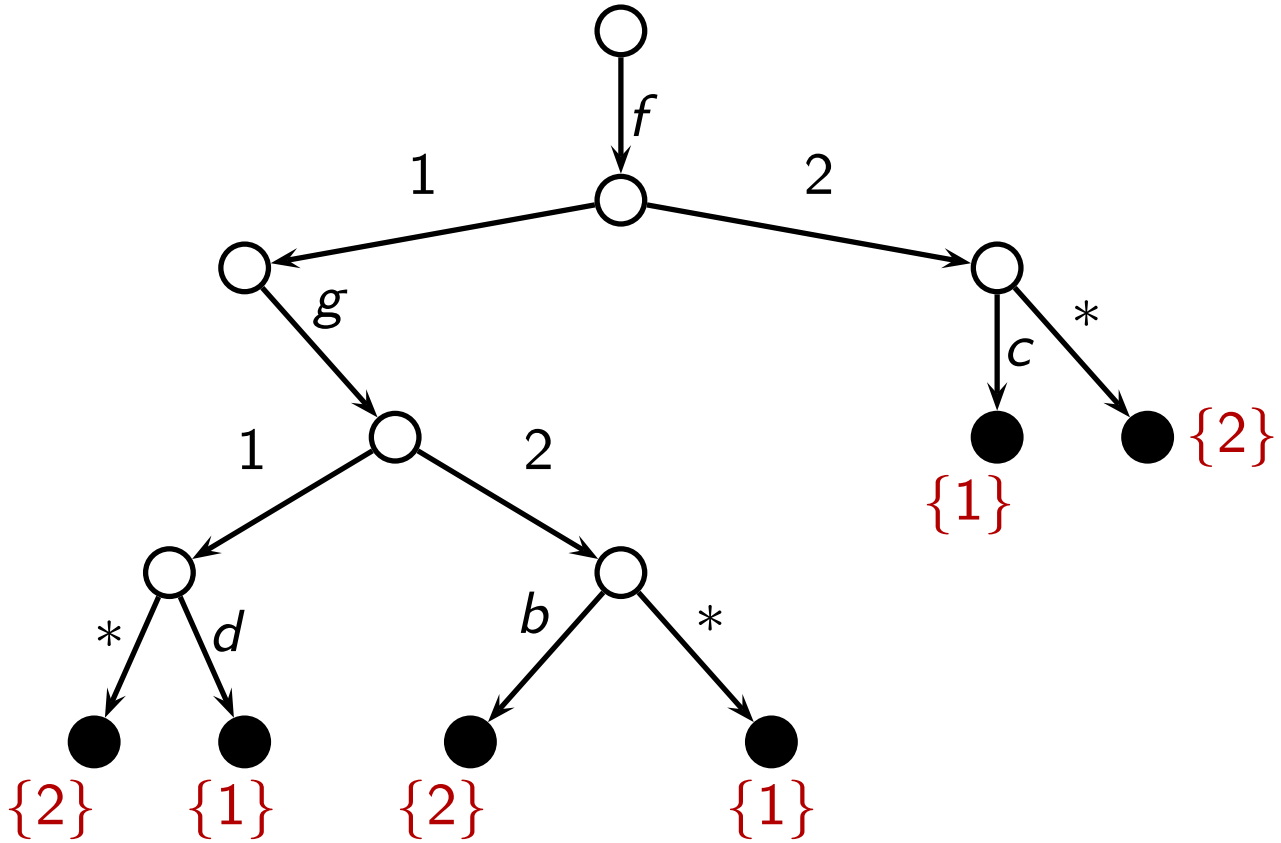
Path Indexing

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



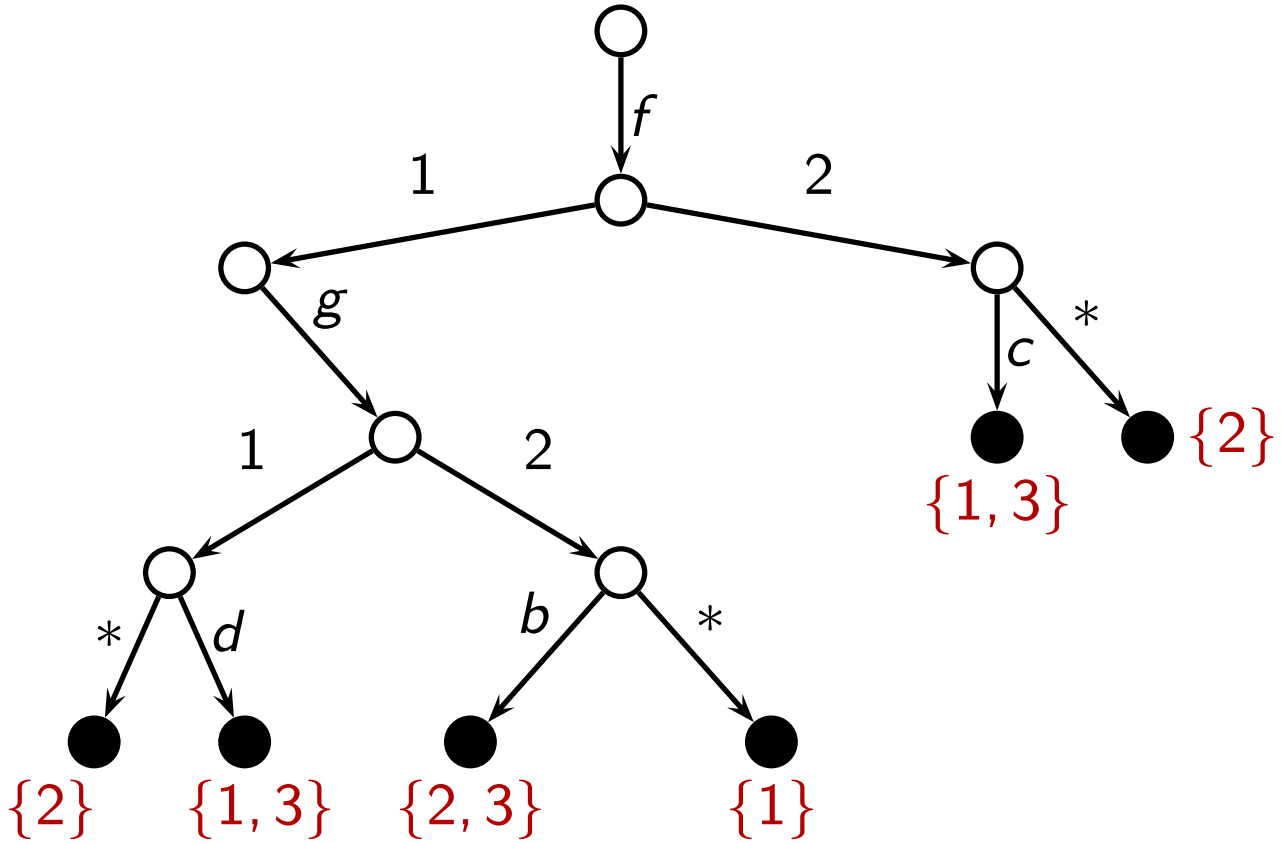
Path Indexing

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



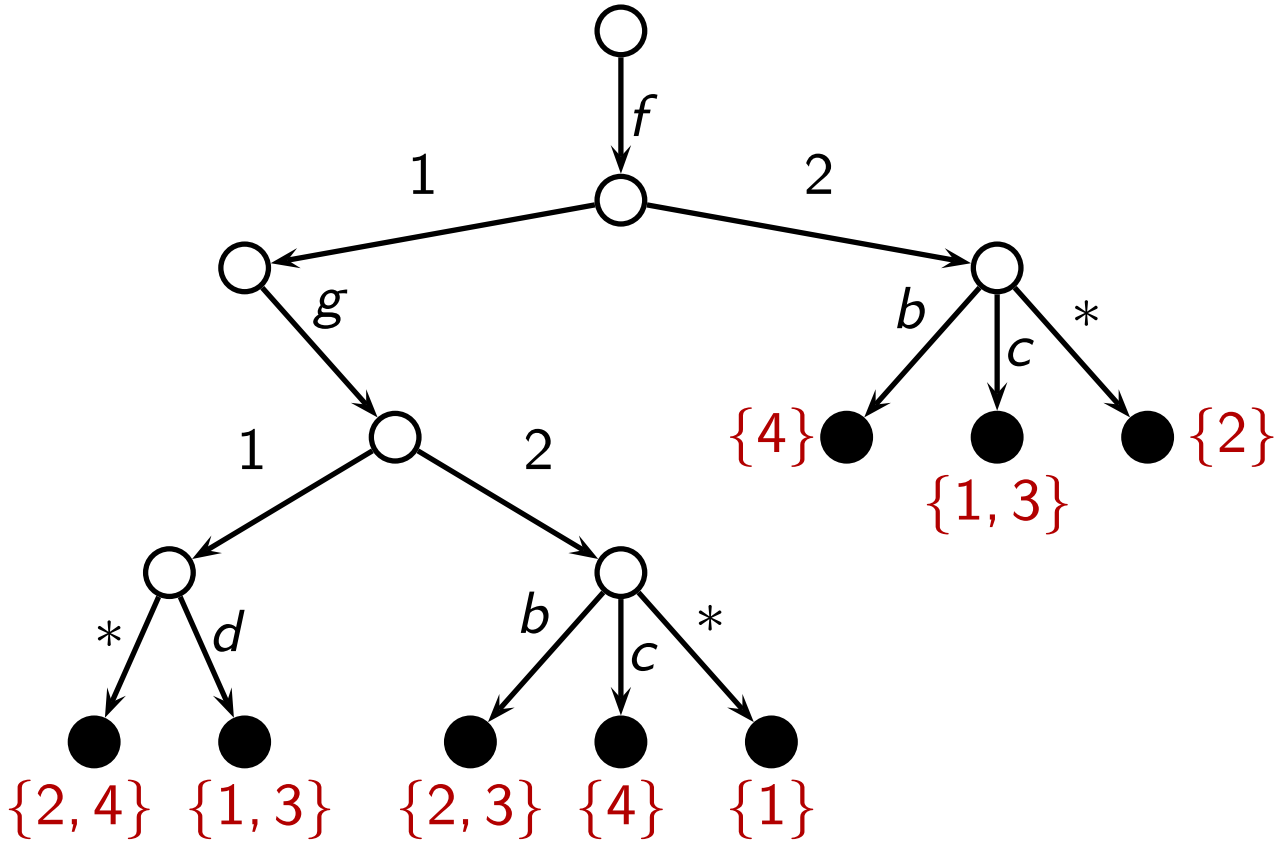
Path Indexing

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



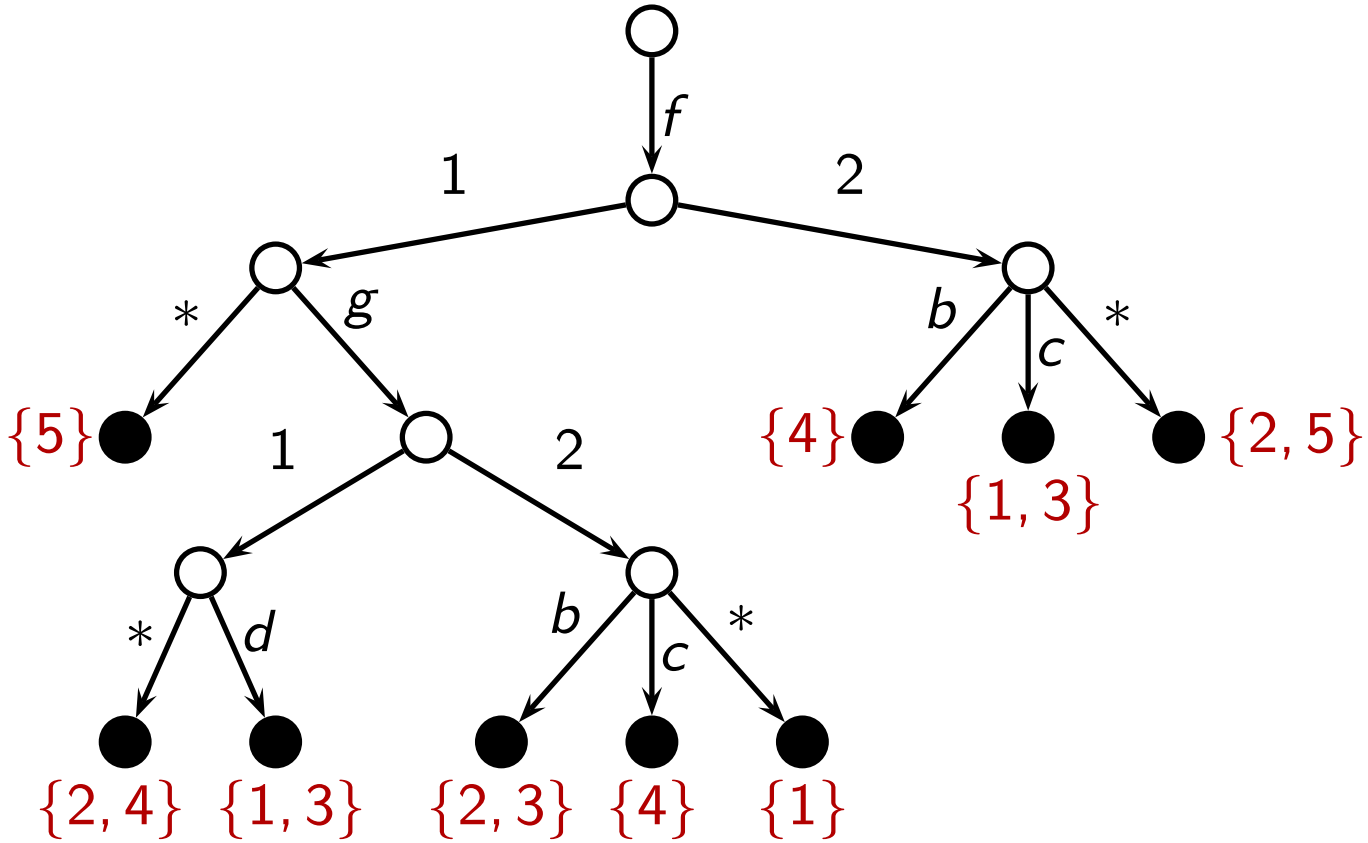
Path Indexing

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



Path Indexing

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



Path Indexing

Advantages:

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

Disadvantages:

- Retrieval requires combining intermediate results for subterms.

Discrimination Trees

Discrimination trees:

Preorder traversals of terms are encoded in a trie.

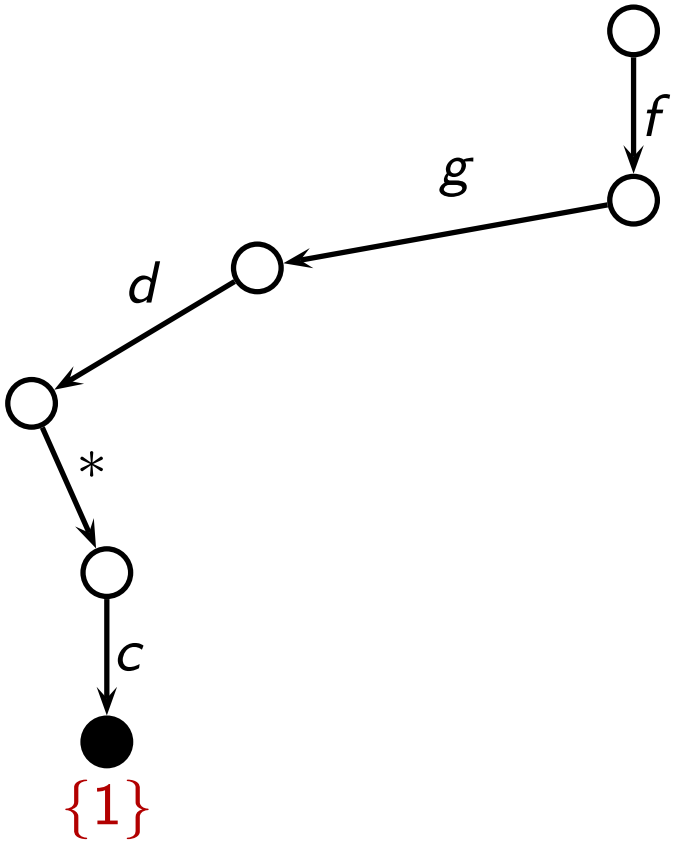
A star $*$ represents arbitrary variables.

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.

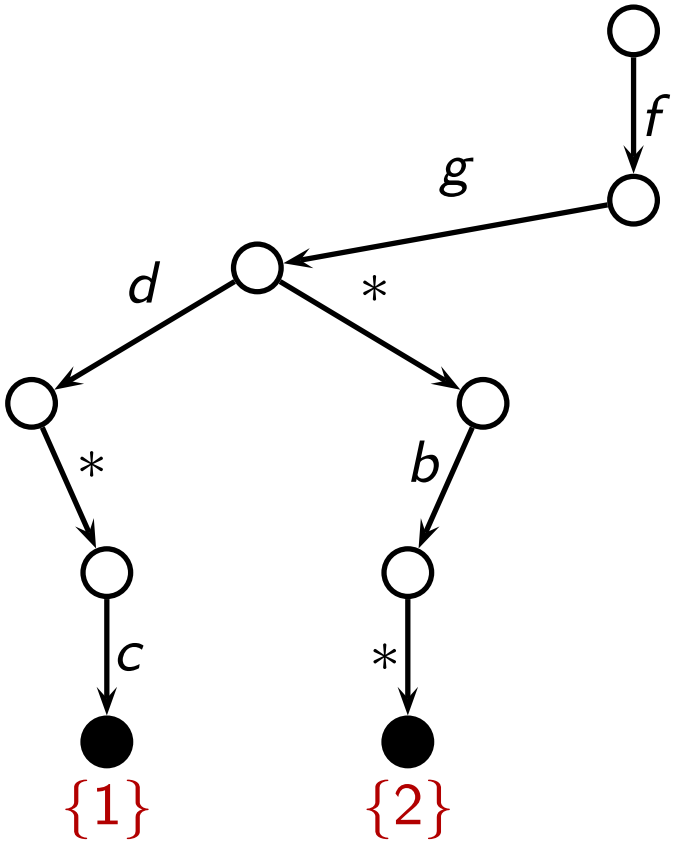
Discrimination Trees

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



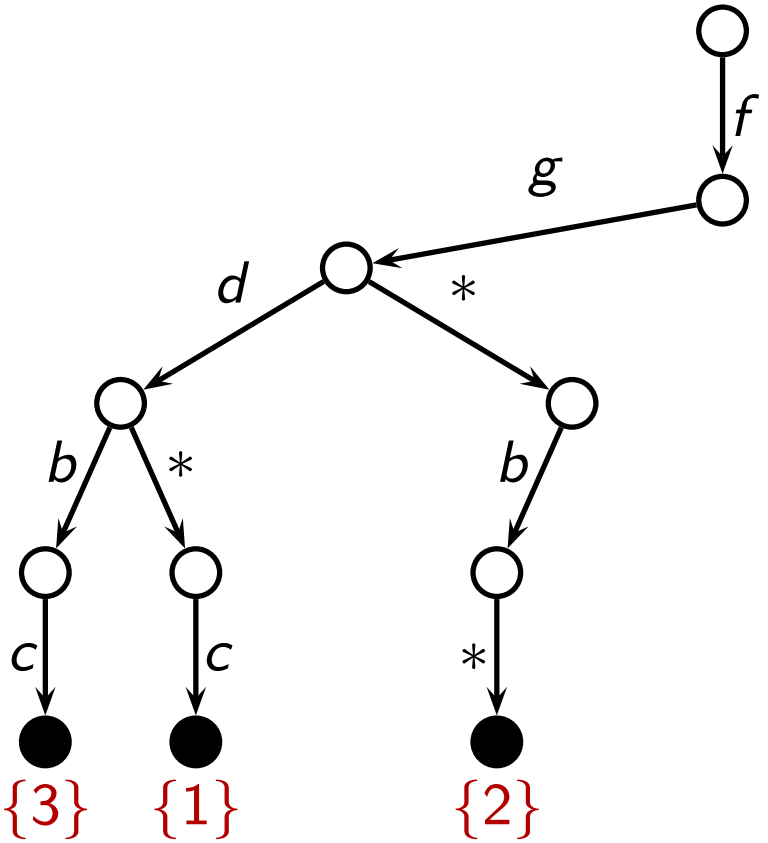
Discrimination Trees

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



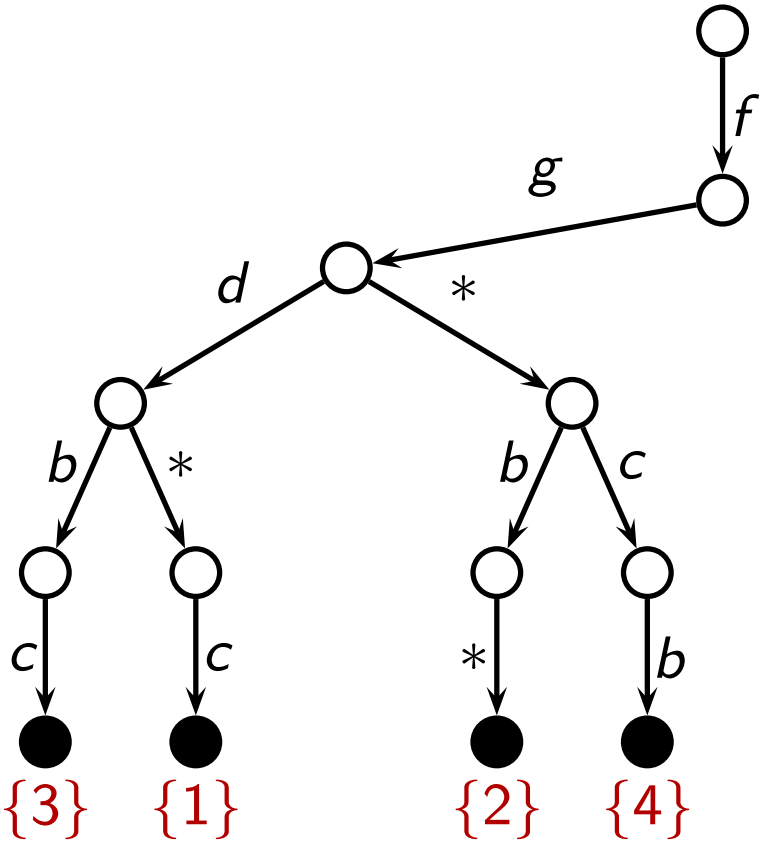
Discrimination Trees

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



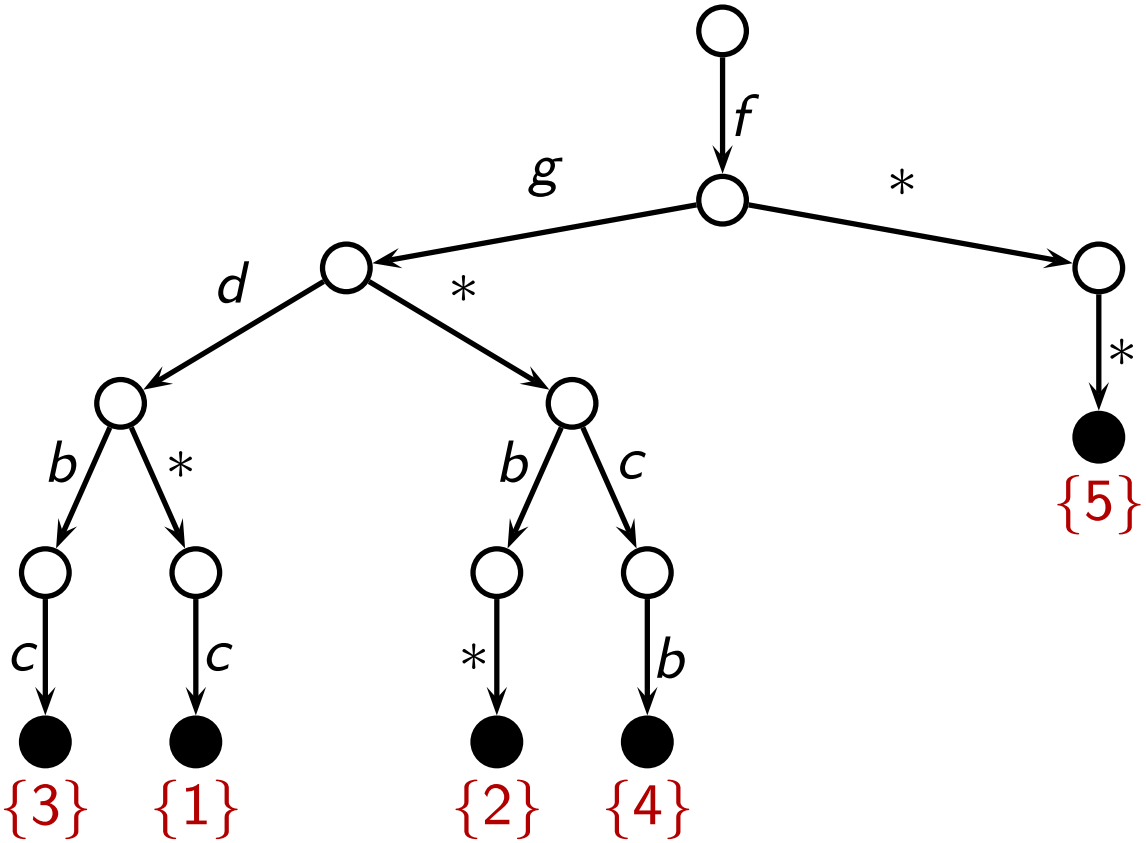
Discrimination Trees

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



Discrimination Trees

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



Discrimination Trees

Advantages:

- Each leaf yields one term, hence retrieval does not require intersections of intermediate results for subterms.
- Good for finding generalizations.

Disadvantages:

- Uses more storage than path indexing (due to less sharing).
- Uses still more storage, if jump lists are maintained to speed up the search for instances or unifiable terms.
- Backtracking required for retrieval.

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

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.

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