Part 4: Implementation Issues

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.)

4.1 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.

- W = "Worked-off" (or "active") clauses:
  Have already been selected as "given clause".
  (So all inferences between these clauses have already been computed.)
- U = "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 U;
$U := U \setminus \{C\}$.

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

The Main Loop

Additionally:

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

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

The Main Loop

Design decision: should one also simplify U using W ?

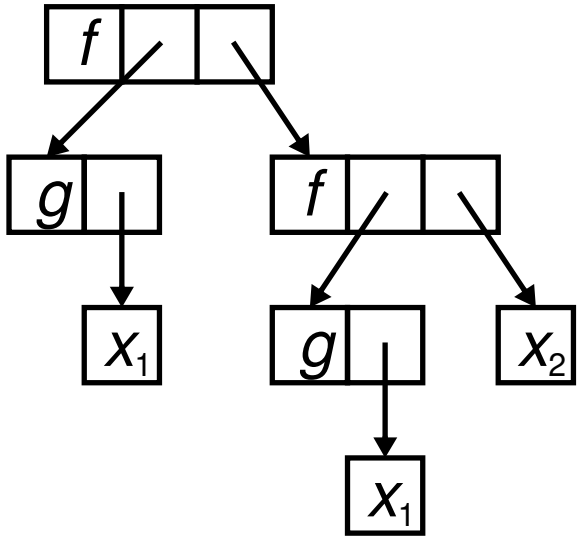
yes $\leadsto$ "Otter loop":
Advantage: simplifications of U may be useful to derive the empty clause.

no $\leadsto$ "Discount loop":
Advantage: clauses in U are really passive;
only clauses in W 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.)

4.2 Term Representations

The obvious data structure for terms: Trees

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

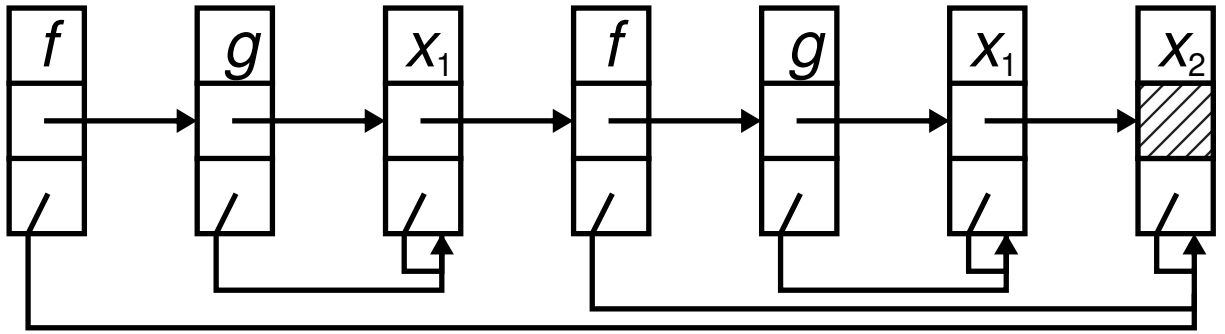


optionally: (full) sharing

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.

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

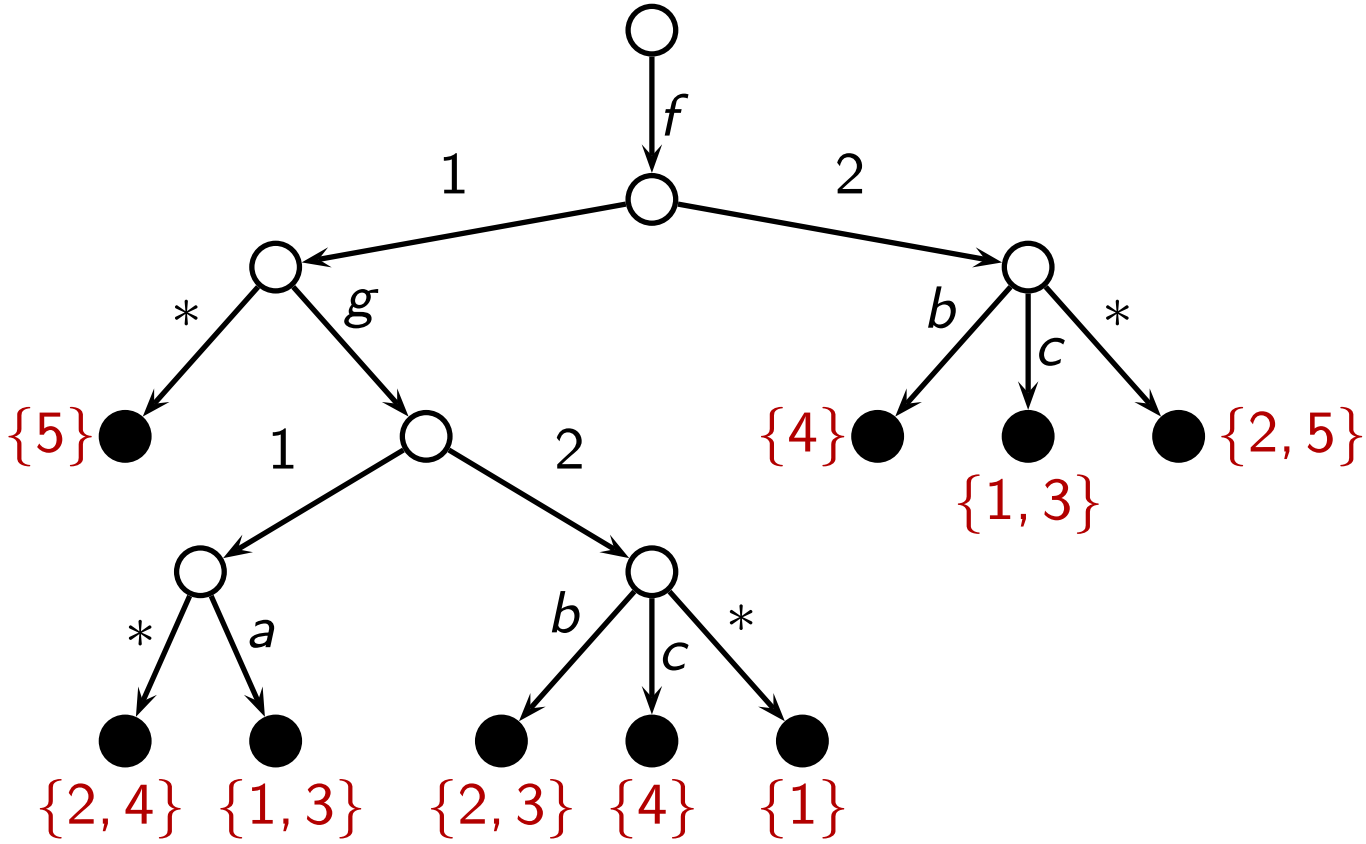
$f.1.g.2.b$

$f.2.*$

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

Path Indexing

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



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.

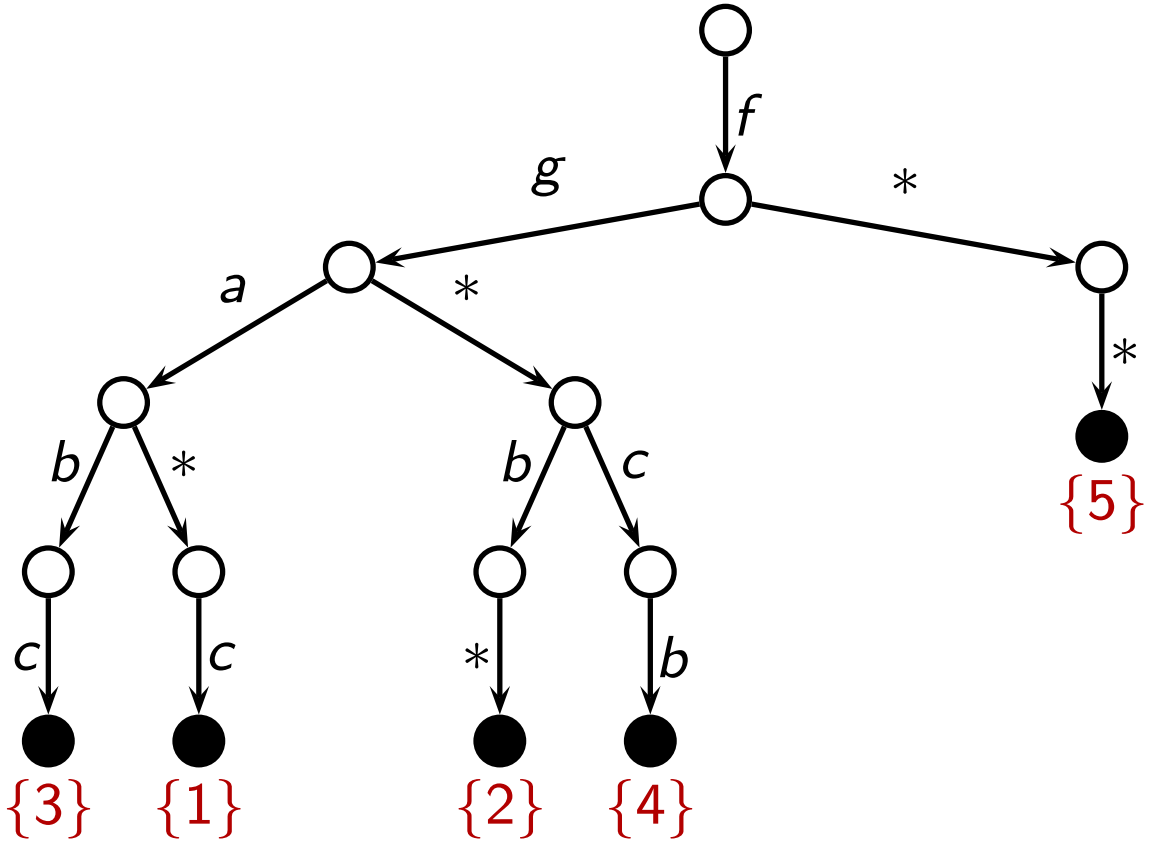
A star $*$ represents arbitrary variables.

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 $\{f(g(a,*),c),\ f(g(*,b),*),$ $f(g(a,b),c),\ f(g(*,c),b),\ f(*,*)\}$



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.

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.