# 5 Termination Revisited

So far: Termination as a subordinate task for entailment checking.

TRS is generated by some saturation process; ordering must be chosen before the saturation starts.

Now: Termination as a main task (e. g., for program analysis).

TRS is fixed and known in advance.

Literature:

Nao Hirokawa and Aart Middeldorp: Dependency Pairs Revisited, RTA 2004, pp. 249-268 (in particular Sect. 1–4).

Thomas Arts and Jürgen Giesl: Termination of Term Rewriting Using Dependency Pairs, Theoretical Computer Science, 236:133-178, 2000.

## 5.1 Dependency Pairs

Invented by T. Arts and J. Giesl in 1996, many refinements since then.

Given: finite TRS $R$ over $\Sigma = (\Omega, \emptyset)$.

$T_0 := \{\, t \in \mathrm{T}_\Sigma(X) \mid$ there is an infinite derivation $t \to_R t_1 \to_R t_2 \to_R \dots \}$.

$T_\infty := \{\, t \in T_0 \mid \forall p > \varepsilon : t|_p \notin T_0 \,\}$ = minimal elements of $T_0$ w. r. t. $\rhd$.

$t \in T_0 \Rightarrow$ there exists a $t' \in T_\infty$ such that $t \unrhd t'$.

$R$ is non-terminating iff $T_0 \neq \emptyset$ iff $T_\infty \neq \emptyset$.

Assume that $T_\infty \neq \emptyset$ and consider some non-terminating derivation starting from $t \in T_\infty$. Since all subterms of $t$ allow only finite derivations, at some point a rule $l \to r \in R$ must be applied at the root of $t$ (possibly preceded by rewrite steps below the root):

$$t = f(t_1, \dots, t_n) \xrightarrow{>\varepsilon}{}^{*}_{R} f(s_1, \dots, s_n) = l\sigma \xrightarrow{\varepsilon}_{R} r\sigma.$$

In particular, $\mathrm{root}(t) = \mathrm{root}(l)$, so we see that the root symbol of any term in $T_\infty$ must be contained in $D := \{\, \mathrm{root}(l) \mid l \to r \in R \,\}$. $D$ is called the set of *defined symbols* of $R$; $C := \Omega \setminus D$ is called the set of *constructor symbols* of $R$.

The term $r\sigma$ is contained in $T_0$, so there exists a $v \in T_\infty$ such that $r\sigma \unrhd v$.

If $v$ occurred in $r\sigma$ at or below a variable position of $r$, then $x\sigma|_p = v$ for some $x \in \mathrm{var}(r) \subseteq \mathrm{var}(l)$, hence $s_i \unrhd x\sigma$ and there would be an infinite derivation starting from some $t_i$. This contradicts $t \in T_\infty$, though.

Therefore, $v = u\sigma$ for some non-variable subterm $u$ of $r$. As $v \in T_\infty$, we see that $\text{root}(u) = \text{root}(v) \in D$. Moreover, $u$ cannot be a proper subterm of $l$, since otherwise again there would be an infinite derivation starting from some $t_i$.

Putting everything together, we obtain

$$t = f(t_1, \dots, t_n) \xrightarrow{>\varepsilon}{}_R^* f(s_1, \dots, s_n) = l\sigma \xrightarrow{\varepsilon}{}_R r\sigma \trianglerighteq u\sigma$$

where $r \trianglerighteq u$, $\text{root}(u) \in D$, $l \ntrianglerighteq u$, $u$ is not a variable.

Since $u\sigma \in T_\infty$, we can continue this process and obtain an infinite sequence.

If we define $S := \{\, l \to u \mid l \to r \in R,\ r \trianglerighteq u,\ \text{root}(u) \in D,\ l \ntrianglerighteq u,\ u \notin X \,\}$, we can combine the rewrite step at the root and the subterm step and obtain

$$t \xrightarrow{>\varepsilon}{}_R^* l\sigma \xrightarrow{\varepsilon}{}_S u\sigma.$$

To get rid of the superscripts $\varepsilon$ and $>\varepsilon$, it turns out to be useful to introduce a new set of function symbols $f^\sharp$ that are only used for the root symbols of this derivation:

$$\Omega^\sharp := \{\, f^\sharp/n \mid f/n \in \Omega \,\}.$$

For a term $t = f(t_1, \dots, t_n)$ we define $t^\sharp := f^\sharp(t_1, \dots, t_n)$; for a set of terms $T$ we define $T^\sharp := \{\, t^\sharp \mid t \in T \,\}$.

The set of *dependency pairs* of a TRS $R$ is then defined by

$$\text{DP}(R) := \{\, l^\sharp \to u^\sharp \mid l \to r \in R,\ r \trianglerighteq u,\ \text{root}(u) \in D,\ l \ntrianglerighteq u,\ u \notin X \,\}.$$

For $t \in T_\infty$, the sequence using the $S$-rule corresponds now to

$$t^\sharp \to_R^* l^\sharp\sigma \to_{\text{DP}(R)} u^\sharp\sigma$$

where $t^\sharp \in T_\infty^\sharp$ and $u^\sharp\sigma \in T_\infty^\sharp$.

(Note that rules in $R$ do not contain symbols from $\Omega^\sharp$, whereas all roots of terms in $\text{DP}(R)$ come from $\Omega^\sharp$, so rules from $R$ can only be applied below the root and rules from $\text{DP}(R)$ can only be applied at the root.)

Since $u^\sharp\sigma$ is again in $T_\infty^\sharp$, we can continue the process in the same way. We obtain: $R$ is non-terminating iff there is an infinite sequence

$$t_1 \to_R^* t_2 \to_{\text{DP}(R)} t_3 \to_R^* t_4 \to_{\text{DP}(R)} \cdots$$

with $t_i \in T_\infty^\sharp$ for all $i$.

Moreover, if there exists such an infinite sequence, then there exists an infinite sequence in which all DPs that are used are used infinitely often. (If some DP is used only finitely often, we can cut off the initial part of the sequence up to the last occurrence of that DP; the remainder is still an infinite sequence.)

### Dependency Graphs

Such infinite sequences correspond to "cycles" in the "dependency graph":

*Dependency graph* $\mathrm{DG}(R)$ of a TRS $R$:

directed graph

nodes: dependency pairs $s \to t \in \mathrm{DP}(R)$

edges: from $s \to t$ to $u \to v$ if there are $\sigma$, $\tau$ such that $t\sigma \to_R^* u\tau$.

Intuitively, we draw an edge between two dependency pairs, if these two dependency pairs can be used after another in an infinite sequence (with some $R$-steps in between). While this relation is undecidable in general, there are reasonable overapproximations:

The functions cap and ren are defined by:

$$\mathrm{cap}(x) = x$$
$$\mathrm{cap}(f(t_1, \dots, t_n) = \begin{cases} y & \text{if } f \in D \\ f(\mathrm{cap}(t_1), \dots, \mathrm{cap}(t_n)) & \text{if } f \in C \cup D^\sharp \end{cases}$$

$$\mathrm{ren}(x) = y, \quad y \text{ fresh}$$
$$\mathrm{ren}(f(t_1, \dots, t_n) = f(\mathrm{ren}(t_1), \dots, \mathrm{ren}(t_n))$$

The overapproximated dependency graph contains an edge from $s \to t$ to $u \to v$ if $\mathrm{ren}(\mathrm{cap}(t))$ and $u$ are unifiable.

A *cycle* in the dependency graph is a non-empty subset $K \subseteq \mathrm{DP}(R)$ such that there is a non-empty path from every DP in $K$ to every DP in $K$ (the two DPs may be identical).

Let $K \subseteq \mathrm{DP}(R)$. An infinite rewrite sequence in $R \cup K$ of the form

$$t_1 \to_R^* t_2 \to_K t_3 \to_R^* t_4 \to_K \dots$$

with $t_i \in T_\infty^\sharp$ is called $K$-minimal, if all rules in $K$ are used infinitely often.

$R$ is non-terminating iff there is a cycle $K \subseteq \mathrm{DP}(R)$ and a $K$-minimal infinite rewrite sequence.

## 5.2 Subterm Criterion

Our task is to show that there are no $K$-minimal infinite rewrite sequences.

Suppose that every dependency pair symbol $f^\sharp$ in $K$ has positive arity (i. e., no constants). A *simple projection* $\pi$ is a mapping $\pi : \Omega^\sharp \to \mathbb{N}$ such that $\pi(f^\sharp) = i \in \{1, \dots, \mathrm{arity}(f^\sharp)\}$.

We define $\pi(f^\sharp(t_1, \dots, t_n)) = t_{\pi(f^\sharp)}$.

**Theorem 5.1 (Hirokawa and Middeldorp)** *Let $K$ be a cycle in* DG($R$). *If there is a simple projection $\pi$ for $K$ such that $\pi(l) \trianglerighteq \pi(r)$ for every $l \to r \in K$ and $\pi(l) \rhd \pi(r)$ for some $l \to r \in K$, then there are no $K$-minimal sequences.*

**Proof.** Suppose that

$$t_1 \to_R^* u_1 \to_K t_2 \to_R^* u_2 \to_K \ldots$$

is a $K$-minimal infinite rewrite sequence. Apply $\pi$ to every $t_i$:

Case 1: $u_i \to_K t_{i+1}$. There is an $l \to r \in K$ such that $u_i = l\sigma$, $t_{i+1} = r\sigma$. Then $\pi(u_i) = \pi(l)\sigma$ and $\pi(t_{i+1}) = \pi(r)\sigma$. By assumption, $\pi(l) \trianglerighteq \pi(r)$. If $\pi(l) = \pi(r)$, then $\pi(u_i) = \pi(t_{i+1})$. If $\pi(l) \rhd \pi(r)$, then $\pi(u_i) = \pi(l)\sigma \rhd \pi(r)\sigma = \pi(t_{i+1})$. In particular, $\pi(u_i) \rhd \pi(t_{i+1})$ for infinitely many $i$ (since every DP is used infinitely often).

Case 2: $t_i \to_R^* u_i$. Then $\pi(t_i) \to_R^* \pi(u_i)$.

By applying $\pi$ to every term in the $K$-minimal infinite rewrite sequence, we obtain an infinite $(\to_R \cup \rhd)$-sequence containing infinitely many $\rhd$-steps. Since $\rhd$ is well-founded, there must also exist infinitely many $\to_R$-steps (otherwise the infinite sequence would have an infinite tail consisting only of $\rhd$-steps, contradicting well-foundedness.)

Now note that $\rhd \circ \to_R \subseteq \to_R \circ \rhd$. Therefore we can commute $\rhd$-steps and $\to_R$-steps and move all $\to_R$-steps to the front. We obtain an infinite $\to_R$-sequence that starts with $\pi(t_1)$. However $t_1 \rhd \pi(t_1)$ and $t_1 \in T_\infty$, so there cannot be an infinite $\to_R$-sequence starting from $\pi(t_1)$. □

Problem: The number of cycles in DG($R$) can be exponential.

Better method: Analyze strongly connected components (SCCs).

SCC of a graph: maximal subgraph in which there is a non-empty path from every node to every node. (The two nodes can be identical.)[3]

Important property: Every cycle is contained in some SCC.

Idea: Search for a simple projection $\pi$ such that $\pi(l) \trianglerighteq \pi(r)$ for all DPs $l \to r$ in the SCC. Delete all DPs in the SCC for which $\pi(l) \rhd \pi(r)$ (by the previous theorem, there cannot be any $K$-minimal infinite rewrite sequences using these DPs). Then re-compute SCCs for the remaining graph and re-start.

No SCCs left $\Rightarrow$ no cycles left $\Rightarrow$ $R$ is terminating.

Example: See Ex. 13 from Hirokawa and Middeldorp.

[3]There are several definitions of SCCs that differ in the treatment of edges from a node to itself.

## 5.3 Reduction Pairs and Argument Filterings

Goal: Show the non-existence of $K$-minimal infinite rewrite sequences

$$t_1 \to_R^* u_1 \to_K t_2 \to_R^* u_2 \to_K \dots$$

using well-founded orderings.

We observe that the requirements for the orderings used here are less restrictive than for reduction orderings:

$K$-rules are only used at the top, so we need stability under substitutions, but compatibility with contexts is unnecessary.

While $\to_K$-steps should be decreasing, for $\to_R$-steps it would be sufficient to show that they are not increasing.

This motivates the following definitions:

*Rewrite quasi-ordering* $\succsim$:

reflexive and transitive binary relation, stable under substitutions, compatible with contexts.

*Reduction pair* $(\succsim, \succ)$:

$\succsim$ is a rewrite quasi-ordering.

$\succ$ is a well-founded ordering that is stable under substitutions.

$\succsim$ and $\succ$ are compatible: $\succsim \circ \succ \subseteq \succ$ or $\succ \circ \succsim \subseteq \succ$.

(In practice, $\succ$ is almost always the strict part of the quasi-ordering $\succsim$.)

Clearly, for any reduction ordering $\succ$, $(\succeq, \succ)$ is a reduction pair. More general reduction pairs can be obtained using argument filterings:

*Argument filtering* $\pi$:

$\pi : \Omega \cup \Omega^\sharp \to \mathbb{N} \cup \text{list of } \mathbb{N}$

$$\pi(f) = \begin{cases} i \in \{1, \dots, \text{arity}(f)\}, \text{ or} \\ [i_1, \dots, i_k], \text{ where } 1 \leq i_1 < \dots < i_k \leq \text{arity}(f),\ 0 \leq k \leq \text{arity}(f) \end{cases}$$

Extension to terms:

$\pi(x) = x$

$\pi(f(t_1, \dots, t_n)) = \pi(t_i)$, if $\pi(f) = i$

$\pi(f(t_1, \dots, t_n)) = f'(\pi(t_{i_1}), \dots, \pi(t_{i_k}))$, if $\pi(f) = [i_1, \dots, i_k]$, where $f'/k$ is a new function symbol.

Let $\succ$ be a reduction ordering, let $\pi$ be an argument filtering. Define $s \succ_\pi t$ iff $\pi(s) \succ \pi(t)$ and $s \succsim_\pi t$ iff $\pi(s) \succeq \pi(t)$.

**Lemma 5.2** *$(\succsim_\pi, \succ_\pi)$ is a reduction pair.*

**Proof.** Follows from the following two properties:

$\pi(s\sigma) = \pi(s)\sigma_\pi$, where $\sigma_\pi$ is the substitution that maps $x$ to $\pi(\sigma(x))$.

$$\pi(s[u]_p) = \begin{cases} \pi(s), & \text{if } p \text{ does not correspond to any position in } \pi(s) \\ \pi(s)[\pi(u)]_q, & \text{if } p \text{ corresponds to } q \text{ in } \pi(s) \end{cases}$$ □

For interpretation-based orderings (such as polynomial orderings) the idea of "cutting out" certain subterms can be included directly in the definition of the ordering:

*Reduction pairs by interpretation:*

Let $\mathcal{A}$ be a $\Sigma$-algebra; let $\succ$ be a well-founded strict partial ordering on its universe.

Assume that all interpretations $f_\mathcal{A}$ of function symbols are *weakly monotone*, i. e., $a_i \succeq b_i$ implies $f(a_1, \dots, , a_n) \succeq f(b_1, \dots, b_n)$ for all $a_i, b_i \in U_\mathcal{A}$.

Define $s \succsim_\mathcal{A} t$ iff $\mathcal{A}(\beta)(s) \succeq \mathcal{A}(\beta)(t)$ for all assignments $\beta : X \to U_\mathcal{A}$; define $s \succ_\mathcal{A} t$ iff $\mathcal{A}(\beta)(s) \succ \mathcal{A}(\beta)(t)$ for all assignments $\beta : X \to U_\mathcal{A}$.

Then $(\succsim_\mathcal{A}, \succ_\mathcal{A})$ is a reduction pair.

For polynomial orderings, this definition permits interpretations of function symbols where some variable does not occur at all (e. g., $P_f(X, Y) = 2X + 1$ for a *binary* function symbol). It is no longer required that *every* variable must occur with some positive coefficient.

**Theorem 5.3 (Arts and Giesl)** *Let $K$ be a cycle in the dependency graph of the TRS $R$. If there is a reduction pair $(\succsim, \succ)$ such that*

- *$l \succsim r$ for all $l \to r \in R$,*
- *$l \succsim r$ or $l \succ r$ for all $l \to r \in K$,*
- *$l \succ r$ for at least one $l \to r \in K$,*

*then there is no $K$-minimal infinite sequence.*

**Proof.** Assume that

$$t_1 \rightarrow_R^* u_1 \rightarrow_K t_2 \rightarrow_R^* u_2 \rightarrow_K \dots$$

is a $K$-minimal infinite rewrite sequence.

As $l \succsim r$ for all $l \rightarrow r \in R$, we obtain $t_i \succsim u_i$ by stability under substitutions, compatibility with contexts, reflexivity and transitivity.

As $l \succsim r$ or $l \succ r$ for all $l \rightarrow r \in K$, we obtain $u_i$ $(\succsim \cup \succ)$ $t_{i+1}$ by stability under substitutions.

So we get an infinite $(\succsim \cup \succ)$-sequence containing infinitely many $\succ$-steps (since every DP in $K$, in particular the one for which $l \succ r$ holds, is used infinitely often).

By compatibility of $\succsim$ and $\succ$, we can transform this into an infinite $\succ$-sequence, contradicting well-foundedness. □

The idea can be extended to SCCs in the same way as for the subterm criterion:

Search for a reduction pair $(\succsim, \succ)$ such that $l \succsim r$ for all $l \rightarrow r \in R$ and $l \succsim r$ or $l \succ r$ for all DPs $l \rightarrow r$ in the SCC. Delete all DPs in the SCC for which $l \succ r$. Then re-compute SCCs for the remaining graph and re-start.

Example: Consider the following TRS $R$ from [Arts and Giesl]:

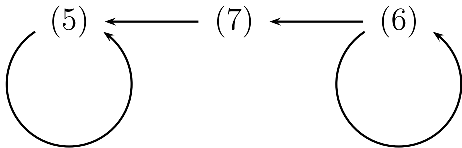
$$minus(x, 0) \rightarrow x \qquad (1)$$
$$minus(s(x), s(y)) \rightarrow minus(x, y) \qquad (2)$$
$$quot(0, s(y)) \rightarrow 0 \qquad (3)$$
$$quot(s(x), s(y)) \rightarrow s(quot(minus(x, y), s(y))) \qquad (4)$$

($R$ is not contained in any simplification ordering, since the left-hand side of rule (4) is embedded in the right-hand side after instantiating $y$ by $s(x)$.)

$R$ has three dependency pairs:

$$minus^\sharp(s(x), s(y)) \rightarrow minus^\sharp(x, y) \qquad (5)$$
$$quot^\sharp(s(x), s(y)) \rightarrow quot^\sharp(minus(x, y), s(y)) \qquad (6)$$
$$quot^\sharp(s(x), s(y)) \rightarrow minus^\sharp(x, y) \qquad (7)$$

The dependency graph of $R$ is



There are exactly two SCCs (and also two cycles). The cycle at (5) can be handled using the subterm criterion with $\pi(minus^\sharp) = 1$. For the cycle at (6) we can use an argument filtering $\pi$ that maps *minus* to 1 and leaves all other function symbols unchanged (that is, $\pi(g) = [1, \dots, \text{arity}(g)]$ for every $g$ different from *minus*.) After applying the argument filtering, we compare left and right-hand sides using an LPO with precedence $quot > s$ (the precedence of other symbols is irrelevant). We obtain $l \succ r$ for (6) and $l \succsim r$ for (1), (2), (3), (4), so the previous theorem can be applied.

### DP Processors

The methods described so far are particular cases of *DP processors:*

A DP processor

$$\frac{(G, R)}{(G_1, R_1), \ \dots, \ (G_n, R_n)}$$

takes a graph $G$ and a TRS $R$ as input and produces a set of pairs consisting of a graph and a TRS.

It is sound and complete if there are $K$-minimal infinite sequences for $G$ and $R$ if and only if there are $K$-minimal infinite sequences for at least one of the pairs $(G_i, R_i)$.

Examples:

$$\frac{(G, R)}{(SCC_1, R), \ \dots, \ (SCC_n, R)}$$

where $SCC_1, \dots, SCC_n$ are the strongly connected components of $G$.

$$\frac{(G, R)}{(G \setminus N, R)}$$

if there is an SCC of $G$ and a simple projection $\pi$ such that $\pi(l) \trianglerighteq \pi(r)$ for all DPs $l \to r$ in the SCC, and $N$ is the set of DPs of the SCC for which $\pi(l) \triangleright \pi(r)$.

(and analogously for reduction pairs)

### Innermost Termination

The dependency method can also be used for proving termination of *innermost rewriting:* $s \xrightarrow{\text{i}}_R t$ if $s \to_R t$ at position $p$ and no rule of $R$ can be applied at a position strictly below $p$. (DP processors for innermost termination are more powerful than for ordinary termination, and for program analysis, innermost termination is usually sufficient.)

# 6 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.

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.)

## 6.1 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.

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.)
- $U$ = "Usable" (or "passive") clauses: Have not yet been selected as "given clause".

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 $WO$; $\mathit{New} = \mathit{Infer}(\{ D_i \mid i \in I \}, C)$; $U = U \cup \mathit{New}$; $WO = WO \cup \{C\}$

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 $C$.

Design decision: should one also simplify $U$ using $WO$?

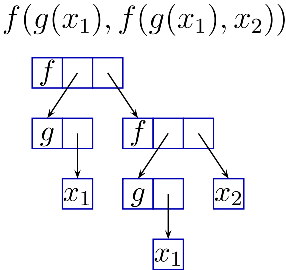
yes $\leadsto$ "Otter loop":
Advantage: simplifications of $U$ may be useful to derive the empty clause.

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

## 6.2 Term Representations

The obvious data structure for terms: Trees

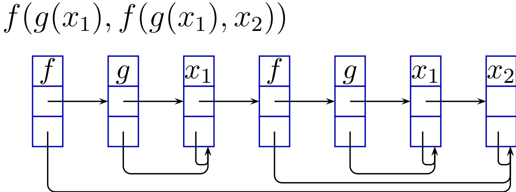
$f(g(x_1), f(g(x_1), x_2))$




optionally: (full) sharing

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.

## 6.3 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 $t$,
- ...

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).

Many different approaches:

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

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:

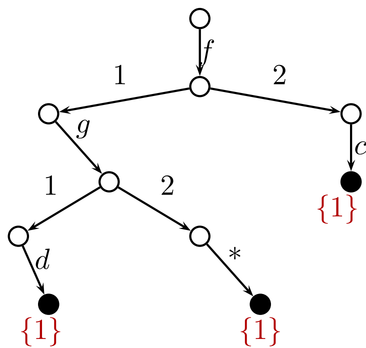
Paths of terms are encoded in a trie ("retrieval tree").

A star $*$ represents arbitrary variables.

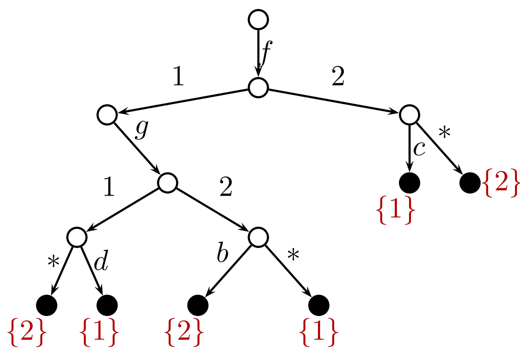
Example: Paths of $f(g(*, b), *)$: $f.1.g.1.*$
$f.1.g.2.b$
$f.2.*$

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

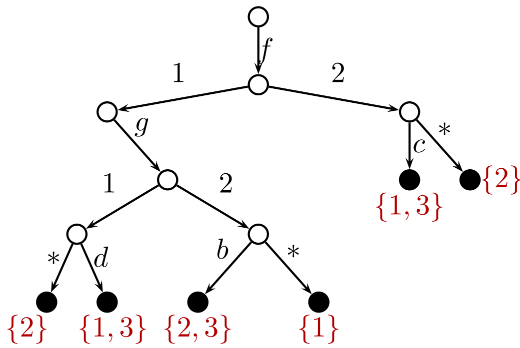
Example: Path index for $\{f(g(d, *), c)\}$



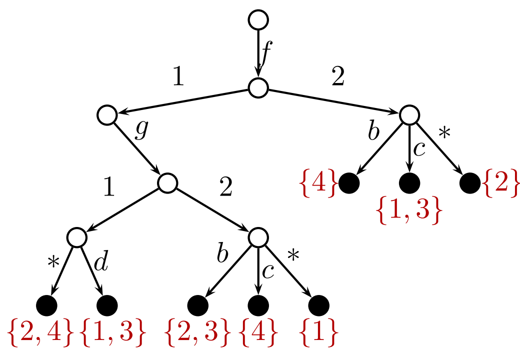

Example: Path index for $\{f(g(d, *), c),\ f(g(*, b), *)\}$



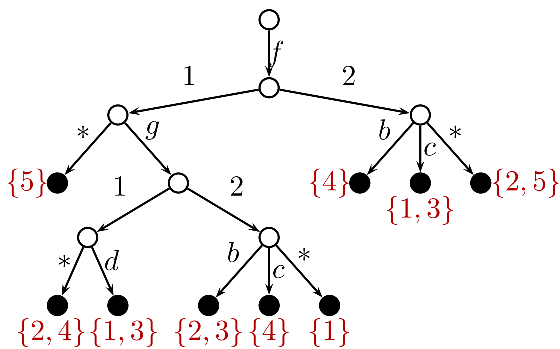
Example: Path index for $\{f(g(d,*),c),\ f(g(*,b),*),\ f(g(d,b),c)\}$




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




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




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:

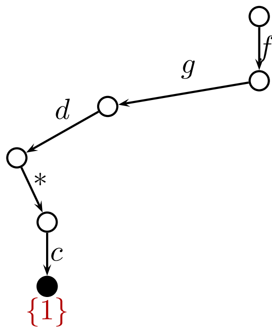
Preorder traversals of terms are encoded in a trie.

A star $*$ represents arbitrary variables.

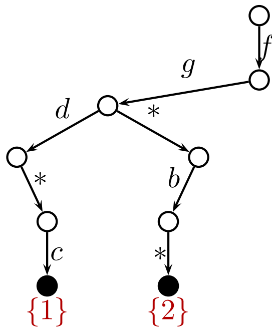
Example: String of $f(g(*, b), *)$: $\quad f.g.*.b.*$

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

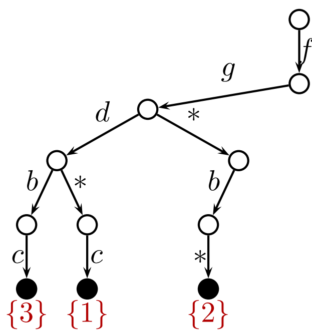
Example: Discrimination tree for $\{f(g(d, *), c)\}$



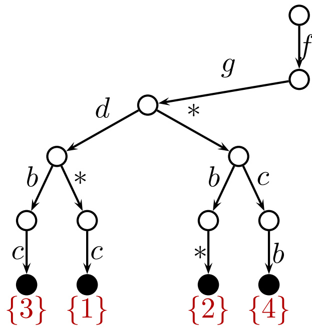

Example: Discrimination tree for $\{f(g(d, *), c),\ f(g(*, b), *)\}$



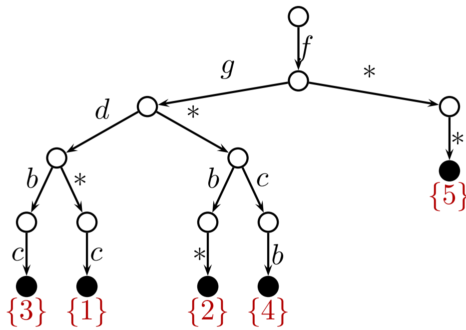

Example: Discrimination tree for $\{f(g(d, *), c),\ f(g(*, b), *),\ f(g(d, b), c)\}$



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




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




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.

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 $C$.
- ...

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.

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.

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

# 7 Outlook

## 7.1 Satisfiability Modulo Theories (SMT)

DPLL checks satisfiability of propositional formulas.

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

Ground first-order atoms are treated like propositional variables.

Truth values of $P(a), Q(a), Q(f(a))$ are independent.

For ground formulas with equality, independence is lost:

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 DPLL, but we must combine it with a decision procedure for the theory part $T$:

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

New DPLL rules:

$T$-Propagate:

$M \parallel N \;\Rightarrow_{\text{DPLL(T)}}\; M\, L \parallel N$

if $M \models_T L$ where $L$ is undefined in $M$ and $L$ or $\overline{L}$ occurs in $N$.

$T$-Learn:

$M \parallel N \;\Rightarrow_{\text{DPLL(T)}}\; M \parallel N \cup \{C\}$

if $N \models_T C$ and each atom of $C$ occurs in $N$ or $M$.

$T$-Backjump:

$$M\ L^{\mathrm{d}}\ M' \parallel N \cup \{C\} \ \Rightarrow_{\mathrm{DPLL(T)}}\ M\ L' \parallel N \cup \{C\}$$

if $M\ L^{\mathrm{d}}\ M' \models \neg C$
and there is some "backjump clause" $C' \vee L'$ such that
$N \cup \{C\} \models_T C' \vee L'$ and $M \models \neg C'$,
$L'$ is undefined under $M$, and
$L'$ or $\overline{L'}$ occurs in $N$ or in $M\ L^{\mathrm{d}}\ M'$.

## 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.

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:

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

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:

$$\frac{N \cup \{C_1 \vee C_2\}}{N \cup \{C_1\} \quad | \quad N \cup \{C_2\}}$$

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

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

Splitting tree is explored using intelligent backtracking.

## 7.4 Integrating Theories into Resolution

Certain kinds of axioms are

important in practice,

but difficult for theorem provers.

Most important case: equality

but also: orderings, (associativity and) commutativity, ...

Idea: Combine ordered resolution and critical pair computation.

Superposition (ground case):

$$\frac{D' \vee t \approx t' \qquad C' \vee s[t] \approx s'}{D' \vee C' \vee s[t'] \approx s'}$$

Superposition (non-ground case):

$$\frac{D' \vee t \approx t' \qquad C' \vee s[u] \approx s'}{(D' \vee C' \vee s[t'] \approx s')\sigma}$$

where $\sigma = \mathrm{mgu}(t, u)$ and $u$ is not a variable.

Advantages:

No variable overlaps (as in KB-completion).

Stronger ordering restrictions:
Only overlaps of (strictly) maximal sides of (strictly) maximal literals are required.

Stronger redundancy criteria.

Similarly for orderings:

Ordered chaining:

$$\frac{D' \ \vee \ t' < t \qquad C' \ \vee \ s < s'}{(D' \ \vee \ C' \ \vee \ t' < s')\sigma}$$

where $\sigma$ is a most general unifier of $t$ and $s$.

Integrating other theories:

Black box:

Use external decision procedure.

Easy, but works only under certain restrictions.

White box:

Integrate using specialized inference rules and theory unification.

Hard work.

Often: integrating more theory axioms is better.