**Lemma 4.57** *Let $N_0 \vdash N_1 \vdash N_2 \vdash \ldots$ be a run. Then $Red(N_i) \subseteq Red(N_\infty)$ and $Red(N_i) \subseteq Red(N_*)$ for every $i$.*

**Proof.** Exercise. □

**Corollary 4.58** $N_i \subseteq N_* \cup Red(N_*)$ *for every $i$.*

**Proof.** If $C \in N_i \setminus N_*$, then there is a $k \geq i$ such that $C \in N_k \setminus N_{k+1}$, so $C$ must be redundant w. r. t. $N_{k+1}$. Consequently, $C$ is redundant w. r. t. $N_*$. □

A run is called *fair,* if the conclusion of every inference from clauses in $N_* \setminus Red(N_*)$ is contained in some $N_i \cup Red(N_i)$.

**Lemma 4.59** *If a run is fair, then its limit is saturated up to redundancy.*

**Proof.** If the run is fair, then the conclusion of every inference from non-redundant clauses in $N_*$ is contained in some $N_i \cup Red(N_i)$, and therefore contained in $N_* \cup Red(N_*)$. Hence $N_*$ is saturated up to redundancy. □

**Theorem 4.60 (Refutational Completeness: Dynamic View)** *Let $N_0 \vdash N_1 \vdash N_2 \vdash \ldots$ be a fair run, let $N_*$ be its limit. Then $N_0$ has a model if and only if $\perp \notin N_*$.*

**Proof.** ($\Leftarrow$): By fairness, $N_*$ is saturated up to redundancy. If $\perp \notin N_*$, then it has a term-generated model. Since every clause in $N_0$ is contained in $N_*$ or redundant w. r. t. $N_*$, this model is also a model of $G_\Sigma(N_0)$ and therefore a model of $N_0$.

($\Rightarrow$): Obvious, since $N_0 \models N_*$. □

**Superposition: Extensions**

Extensions and improvements:

simplification techniques,

selection functions (as for ordered resolution),

redundancy for inferences,

basic strategies,

constraint reasoning.

## Theory Reasoning

Superposition vs. resolution + equality axioms:

    specialized inference rules, thus no inferences with theory axioms,

    computation modulo symmetry,

    stronger ordering restrictions,

    no variable overlaps,

    stronger redundancy criterion.

Similar techniques can be used for other theories:

    transitive relations,

    dense total orderings without endpoints,

    commutativity,

    associativity and commutativity,

    abelian monoids,

    abelian groups,

    divisible torsion-free abelian groups.

Observations:

    no inferences with theory axioms:
    yes, usually possible.

    computation modulo theory axioms:
    often possible, but requires unification and orderings modulo theory.

    stronger ordering restrictions, no variable overlaps:
    sometimes possible, but in many cases, certain variable overlaps remain necessary.

    stronger redundancy criterion:
    depends on the model construction.

    In many cases, integrating more theory axioms simplifies matters.

    Inefficient unification procedures may be replaced by constraints.

# 5 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.

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.

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".

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

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 $C$.

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

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

no $\rightsquigarrow$ "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.)

## 5.2 Term Representations

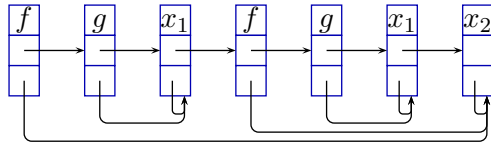The obvious data structure for terms: Trees

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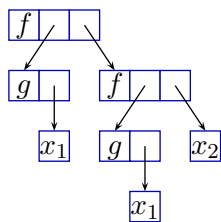


optionally: (full) sharing

An alternative: Flatterms

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

| $f$ | $g$ | $x_1$ | $f$ | $g$ | $x_1$ | $x_2$ |

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 $t$,

- ...

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