Implementation of Resolution

Hans de Nivelle
Max Planck Institut für Informatik
Im Stadtwald
D-66123, Saarbrücken
Germany

07.05.1999

Goals of Bliksem

Bliksem is designed with the following objectives in mind:

- Technically efficient.
- Theoretically up to date.
- Portable, easy to install.

General words of Wisdom

- Die Kunst des Lassens ist ebenso wichtig wie die Kunst des Tuns.
- Try to find the real solutions for engineering problems, not just a solution.
- Make many profiles.

Calculus Used

Derivation Rules:

Resolution From A, R_1 and $\neg\, A, R_2$ derive R_1, R_2.

Paramodulation From $t_1 \approx t_2, R_1$ and $A[t_1], R_2$ derive $A[t_2], R_2$.

Factoring From A, A, R derive A, R.

Eqrefl From $\neg t \approx t, R$ derive R.

Eqfact From $t_1 \approx t_2, t_1 \approx t_3, R$ derive $\neg t_2 \approx t_3, t_1 \approx t_3, R$.

We omitted the substitutions, and maximality conditions.

Implementation of Terms

This is probably the most important design decision. One should evaluate the alternatives from the following perspectives:

- Space used
- Time used by standard operations
- Flexibility.

Space is increasingly less important. However there is one point to consider: The space used influences the time, as observed in [**?**] due to hardware optimizations.

It is not quite clear what the standard operations are. They should at least include left/right traversal, and access of subterms.

With some representations it is hard to reuse space occupied by a term that is not in use anymore.

We give examples, using the term $f(s(X), Y, s(a))$.

Prefix Representation

address:	terms
$100:$	f
$101:$	s
$102:$	X
$103:$	Y
$104:$	s
$105:$	a

Uses little memory.

Left/right traversal requires some administration, in order to know where the term ends.

Accessing subterms requires administration.

Prefix with End Representation

address:	terms	end
100 :	f	106
101 :	s	103
102 :	X	103
103 :	Y	104
104 :	s	106
105 :	a	106

Obtained from the prefix representation, by adding end references.

No administration needed.

Uses 2 times more memory than the (pure) prefix representation.

Flatterms

address:	terms	next	end
100 :	f	101	106
101 :	s	102	103
102 :	X	103	103
103 :	Y	104	104
104 :	s	105	106
105 :	a	106	106

Very similar to the prefix with end representation.

Uses 3 times more memory than the prefix representation.

No sharing is possible.

Deep Terms

address:	first
70 :	Y
80 : 81 :	s 82
82 :	X
90 : 91 :	s 95
95 :	a
110 : 111 : 112 : 113 :	f 80 70 90

Uses app. 2 times more memory than the prefix representation.
Left/right traversal involves administration, or recursion.
Allows sharing

Deep Terms with Argument List

address:	first	rest
100 :	f	101
101 :	110	102
102 :	120	103
103 :	130	nil
110 :	s	112
112 :	115	nil
115 :	X	nil
120 :	Y	nil
130 :	s	131
131 :	134	nil
134 :	a	nil

Uses app. 3 times more memory than the prefix representation.

Records have equal length.

Left/right traversal involves administration, or recursion.

Allows sharing.

Used in Spass/Otter.

Unification Times in 10^{-6} seconds

term1	term2	prefix	prefixwe	deep	deepwal	flat
$r(a,a,a)$	$q(a,a)$	0.50	0.43	0.34	0.35	0.43
$r(a,a,a)$	$r(b,a,a)$	0.82	0.59	0.79	0.78	0.60
$q(f(a,a),f(a,a))$	$q(X,X)$	3.27	2.01	3.10	3.41	2.15
$q(X,X)$	$q(f(a,a),f(a,a))$	3.19	1.96	3.12	3.43	2.11
$q(f(a,a),f(a,a))$	$q(X,Y)$	3.13	2.10	3.47	3.69	2.26
$q(f(a,a),f(a,b))$	$q(f(a,a),f(a,b))$	2.61	1.41	2.74	3.17	1.51
$q(f(a,a),f(a,b))$	$q(f(a,a),f(a,a))$	2.36	1.37	2.71	3.00	1.46
$q(f(s(s(a)),s(s(b))),$ $\ldots\ f(s(s(X)),s(s(Y))))$	$q(X,X)$	6.48	4.19	6.75	8.06	4.52
$q(f(s(s(a)),s(s(b))),$ $\ldots\ f(s(s(X)),s(s(X))))$	$q(X,X)$	6.10	3.77	6.12	7.31	4.03
$q(f(X,Y),f(X,Z))$	$q(f(Y,X),f(Z,X))$	6.08	4.28	5.30	5.61	4.38
$q(s(s(f(X,Y))),s(Y))$	$q(X,s(s(s(X))))$	3.79	2.79	4.92	17.74	3.12

Matching Times in 10^{-6} seconds

term1	term2	prefix	prefixwe	deep	deepwal	flat
$r(a,a,a)$	$q(a,a)$	0.37	0.39	0.23	0.24	0.34
$r(a,a,a)$	$r(b,a,a)$	0.55	0.50	0.58	0.54	0.46
$r(a,a,a)$	$r(a,b,a)$	0.72	0.60	0.90	0.86	0.58
$r(a,b,a)$	$r(a,b,b)$	0.84	0.71	1.21	1.18	0.69
$r(a,b,a)$	$r(a,b,a)$	0.92	0.73	1.30	1.34	0.77
$r(X,Y,Z)$	$r(s(a),s(b),f(X,Y))$	2.22	1.30	1.78	1.87	1.34
$r(X,Y,X)$	$r(s(f(a,b)),s(f(b,b)),$ $\dots s(s(f(a,b))))$	2.64	1.51	2.41	2.75	1.63
$r(s(X),f(X,Y),f(Y,X))$	$r(s(a),f(a,b),f(b,a))$	2.86	2.48	3.70	4.03	2.58
$r(X,X,X)$	$r(s^{10}(a),s^{10}(a),s^{10}(a))$	5.26	3.02	6.11	41.26	3.62

Implementation of Substitutions

A substitution is an assignment of terms to a finite set of variables. There are two main ways of representing substitutions:

- Shallow Substitutions.
  In a *shallow substitution*, the variables are represented by small integers. This makes it possible to represent the values for $V_0, V_1, V_2, \ldots$ as values in an array $\Theta[V_0], \Theta[V_1], \Theta[V_2], \ldots$

- Deep Substitutions.
  In a *deep substitution*, the assignments $V := t$ are stored as an array of pairs (V, t).

Shallow substitutions are used in Otter, Spass.
With shallow substitutions, assignments can be added, deleted, and retrieved in constant time.
However initialization is costly.
Backtracking requires additional administration.

With deep substitutions, assignments can be added and deleted in constant time.
Assignments are retrieved in linear time.
The structure is well-suited to backtracking.

term1	term2	deep1	deep2	shallow2 / 20
$r(a,a,a)$	$q(a,a)$	0.45	0.46	0.93/2.91
$r(a,a,a)$	$r(b,a,a)$	0.58	0.63	1.10/3.19
$q(f(a,a),f(a,a))$	$q(X,X)$	1.99	2.10	2.52/4.30
$q(X,X)$	$q(f(a,a),f(a,a))$	1.96	2.09	2.53/4.31
$q(f(a,a),f(a,a))$	$q(X,Y)$	2.11	2.02	2.17/3.97
$q(f(a,a),f(a,b))$	$q(f(a,a),f(a,b))$	1.39	1.43	2.02/3.88
$q(f(a,a),f(a,b))$	$q(f(a,a),f(a,a))$	1.37	1.40	1.99/3.86
$q(f(s(s(a)),s(s(b))),$ $\ldots\ f(s(s(X)),s(s(Y))))$	$q(X,X)$	4.18	4.07	3.92/5.74
$q(f(s(s(a)),s(s(b))),$ $\ldots\ f(s(s(X)),s(s(X))))$	$q(X,X)$	3.79	3.64	3.88/5.69
$q(f(X,Y),f(X,Z))$	$q(f(Y,X),f(Z,X))$	4.30	3.62	3.33/5.14

Implementation of Discrimination Trees

Discrimination trees arise if one combines shared prefixes of terms/literals. This is useful for retrieving instances/unifiers/generalizations from a large set. In Bliksem discrimination trees are used only for retrieving generalizations. We have done no good benchmark tests.

At present the implementation is ad hoc, (essentially deep terms with argument lists)

Discrimination trees are dynamic.

The best implementation depends on the branching degree, and the number variable branches.

The algorithm is more important, then is the case with terms.

Main Loop of Bliksem

The main loop in Bliksem has the following structure. There are three lists of clauses, **unfactored, inuse, clauses**. **unfactored** contains clauses have to be checked for the unary rules of factoring, equality factoring, and equality reflexitivity. **inuse** contains clauses that can be used for the binary rules of resolution, and paramodulation.

1. As long as **unfactored** is nonempty, pick a clause from **unfactored**, move it to **clauses**, and process all 1-derivable clauses that can be constructed from it.

2. If **clauses** is nonempty, select the lightest/oldest clause from **clauses**, and reprocess it. If it survives, then move it to **inuse**. Construct all 2-derivable clauses that have the other parent in **inuse**, and process them.

Processing a New Clause:

1. Try to simplify the clause by rewriting, and by resolution simplification.

2. If the clause is a tautology, then don't keep it.

3. If the clause is too heavy, then don't keep it.

4. Check if the clause is subsumed. If it is subsumed, then don't keep it.

5. The clause is kept. Put put it in **unfactored**.

Occasionally, a resimplification sweep is made. Clauses in **inuse** are reprocessed as if they were just generated.

Simplification and Subsumption

Subsumption Let c_1 and c_2 be clauses. Clause c_1 subsumes clause c_2 if there exists a substitution Θ, such that $c_1\Theta \subseteq c_2$, and $\mid c_1 \mid \ \leq \ \mid c_2 \mid$. In that case c_2 can be deleted.

Demodulation If c_1 has form $\{t_1 \approx t_2\} \cup R_1$, and c_2 has form $\{A[u]\} \cup R_2$, there exists a substitution Θ, s.t. $R_1\Theta \subseteq R_2$, and $u = t_1\Theta$, and moreover $t_1 \succ t_2$, and $\mid c_1 \mid \ \leq \ \mid c_2 \mid$, then c_2 can be deleted, and replaced by $\{A[t_2\Theta]\} \cup R_2$.

Resolution If c_1 has form $\{\pm A\} \cup R_1$ c_2 has form $\{\mp B\} \cup R_2$, there is a substitution Θ, such that $R_1 \subseteq R_2$, $A\Theta = B$, and $\mid c_1 \mid \ \leq \ \mid c_2 \mid$, then c_2 can be deleted, and replaced by R_2.

Tautology Clause $c = \{\neg a_1, \ldots, \neg a_p, b_1, \ldots, b_q\}$ is a tautology, if $a_1, \ldots, a_p \vdash b_j$, for a j.

The checks interact. It is possible that a clause becomes redundant, or a tautology, after simplification.

The ways in which simplification and subsumption are implemented, are closely related.

Simplification and subsumption are essential if one hopes to find a saturation.

I don't know the right order.

Bad News

Resolution/Demodulation/Subsumption are NP-complete.

Good News

This never shows up in practice

The Real Bad News

There are too many candidates (≥ 100000)

Some optimization is needed.

- Local Optimization
- Global Optimization

Local Optimization

Basic Algorithm:

Check if $\{A_1, \ldots, A_p\}$ subsumes $\{B_1, \ldots, B_q\}$ by backtracking: Non-deterministically try all B_j for each A_i.

Improved algorithm: (Gottlob and Leitsch)

First try to separate $\{A_1, \ldots, A_p\}$ into variable disjoint components, and treat these independently.
Try to split at each stage of the algoritm.

Example

$\{p(X), q(Y)\}$ does not subsume $\{p(0), p(1), p(2), r(1)\}$.

Algorithm in Bliksem

Observe that the subsumption test, is essentially the same as propositional satisfiability checking. Use semantic tableaux with backjumping.
Algorithm:

Non-deterministically try all B_j for each A_i. Mark for each A_i, whether or not it contributed to the failure of an $A_{i'}$ with $i' > i$. It it did not, then do not backtrack on it.

Global Optimization

We have no good solution here.

Bad Solution

In order to decide whether or not c_2 is subsumed, use a discrimination tree to find a c_1, and an $A \in c_1$, together with a substitution Θ, such that $A\Theta \in c_2$. Do the full subsumption test on c_1.

Better Solution?

Store the complete c_1 in the discrimination tree. The problem is that this cannot be combined with the local optimizations.

When to Simplify

This is not at all clear. There are the following solutions:

Waldmeister Keep **inuse** completely simplified. Simplify only with **inuse**.

Spass/Otter/Barcelona Keep the total set of clauses completely simplified.

Bliksem Simplify generated clauses. Simplify clauses when they are selected. Occassionaly resimplify all clauses.

Gandalf Simplify generated clauses. Keep **inuse** completely simplified, but only by the clauses in **inuse**.

Clause Selection

In the main loop, Bliksem selects the lightest/oldest clause from **clauses**. This is done by going through the list. On some problems this process consumes $\geq 60\%$ of the total running time, which is unacceptable.

A possible solution would be to separate **clauses** into n lists
clauses[1], . . . **clauses**[n],
together with **lightest**[1], . . . , **lightest** [n].

Optimization of Binary Rules

For the binary rules one has to retrieve all literals from **inuse**, that are possibly unifiable with a literal in the given clause. One might be tempted to try to optimize this process, but it never shows up on profiles. Bliksem simply goes through the list of clauses.

Normal Form Transformation

In order to handle first order logic one needs a transformation clausal normal form. The standard transformation has the following form:

1. Transform the formula to NNF.

2. Try to antiprenex.

3. Skolemize.

4. Factor into clauses.

Works often nice, but sometimes not:

$$a_0 \leftrightarrow (a_1 \leftrightarrow (a_2 \leftrightarrow \cdots a_n)).$$

$$(a_0 \wedge b_0) \vee (a_1 \wedge b_1) \vee \cdots \vee (a_n \wedge b_n).$$

The problem is not that the formula blows up, but that proof tasks are duplicated, and that in disjunctions alternatives might resolve with each other.

Solution:
Replace a problematic subformula by a fresh name with a definition of this name:

$$\Phi(A(x))$$

is replaced by

$$\Phi(\alpha(x)),$$

and

$$\forall x(\alpha(x) \leftrightarrow (\rightarrow)A(x)).$$

What should be replaced? Consider:
$\forall x[a(x) \wedge b(x)] \vee \forall x[c(x) \wedge d(x)]$.

$\alpha \vee \forall x[c(x) \wedge d(x)]$,
$\alpha \rightarrow \forall x[a(x) \wedge b(x)]$.

$\forall x[\alpha(x)] \vee \forall x[c(x) \wedge d(x)]$,
$\forall x[\alpha(x) \rightarrow a(x) \wedge b(x)]$.

Answer: formulae that help, and that have little free variables.

Estimating the size of the CNF. Assume that the formula is already in CNF.

$$\begin{array}{ll}
\text{Fact}(A \vee B) = & \text{Fact}(A) * \text{Fact}(B) \\
\text{Fact}(A \wedge B) = & \text{Fact}(A) + \text{Fact}(B) \\
\text{Fact}(\forall x A) = & \text{Fact}(A) \\
\text{Fact}(\exists x A) = & \text{Fact}(A) \\
\text{Fact}(\pm a) = & 1.
\end{array}$$

Estimating the effect of replacing subformula F :

If $F = A$, then $\text{Fact}(F, A) = 1$. Otherwise

$$\begin{array}{ll}
\text{Fact}(A \vee B, F) = & \text{Fact}(A, F) * \text{Fact}(B, F) \\
\text{Fact}(A \wedge B, F) = & \text{Fact}(A, F) + \text{Fact}(B, F) \\
\text{Fact}(\forall x A, F) = & \text{Fact}(A, F) \\
\text{Fact}(\exists x A, F) = & \text{Fact}(A, F) \\
\text{Fact}(\pm A, F) = & 1.
\end{array}$$

Subformula F of A is *replacable* if

1 $\text{Fact}(F) > \mathbf{acceptable}$,

2 $\text{Fact}(F, A) + \text{Fact}(A) < \text{Fact(F)}$.

Among the replacable subformulae choose one with the smallest number of free variables. Iterate this until there are no replacable formulae left.

Conclusions

- Bliksem is now quite good on the basic level,
- Strategy selection needs to be improved,
- `http://mpi-sb.mpg.de/~nivelle`