Subsumption Algorithms for Three-Valued Geometric Resolution

Hans de Nivelle

Instytut Informatyki, University of Wrocław, Poland, nivelle@ii.uni.wroc.pl

Abstract. In the implementation of geometric resolution, the most costly operation is subsumption (or matching). In the matching problem, one has to decide for a three-valued, geometric formula, whether this formula is false in a given interpretation. The formula contains only atoms with variables, equality, and existential quantifiers. The interpretation contains only atoms with constants, which are assumed to be distinct.
Because matching is not restricted by term structure, matching for geometric resolution is a hard problem. We translate the matching problem into a generalized constraint satisfaction problem, and give an algorithm that solves it efficiently. The algorithm uses learning teachings, similar to clause learning in propositional logic. After that, we adapt the algorithm in such a way that it finds solutions that use a minimal subset of the interpretation.
The techniques presented in this paper may also have applications in constraint solving.

1 Introduction

Geometric logic as a theorem proving strategy was introduced in [1]. (The infinitary variant is called *coherent logic*.) Bezem and Coquand were motivated mostly by the desire to obtain a theorem proving strategy with a simple normal form transformation, which makes that many natural problems need no transformation at all, others have a much simpler transformation, and which makes that in all cases Skolemization can be avoided. This results in more readable proofs, and proofs that can be backtranslated more easily.

Our motivation for using geometric resolution is different, more engineering-oriented. We hope that three-valued geometric resolution can be used as a generic reasoning core, into which different kinds of two- or three-valued reasoning problems (e.g. problems representing type correctness, two-valued decision problems, simply typed classical problems) can be translated. Because we want the geometric reasoning core to be generic, we are willing to accept transformations that do not preserve much of the structure of the original formula. Subformulas are freely renamed, and functional expressions are flattened and replaced by relations.

We first give a definition of three-valued, geometric formulas. The definition that we give here is too general, but it is easier to understand than the correct definition in [3], because we removed some technical restrictions that are required

by other parts of the geometric search algorithm, which are not relevant in this paper.

Definition 1. *A* geometric literal *has one of the following four forms:*

1. *A simple atom of form* $p_\lambda(x_1, \ldots, x_n)$*, where* $x_1, \ldots, x_n$ *are variables (with repetitions allowed) and* $\lambda \in \{\mathbf{f}, \mathbf{e}, \mathbf{t}\}$.
2. *An equality atom of form* $x_1 \approx x_2$*, with* x_1, x_2 *distinct variables.*
3. *A domain atom* $\#_\mathbf{f}\, x$*, with* x *a variable.*
4. *An* existential atom *of form* $\exists y\, p_\lambda(x_1, \ldots, x_n, y)$ *with* $\lambda \in \{\mathbf{f}, \mathbf{e}, \mathbf{t}\}$*, and such that* y *occurs at least once in the atom, not necessarily on the last place.*

A geometric formula *has form* $A_1, \ldots, A_p \mid B_1, \ldots, B_q$*, where the* A_i *are simple or domain atoms, and the* B_j *are atoms of arbitrary type.*

We require that geometric formulas are range restricted*, which means that every variable that occurs free in a* B_j *must occur in an* A_i *as well.*

The intuitive meaning of $A_1, \ldots, A_p \mid B_1, \ldots, B_q$ *is* $\forall \overline{x}\ A_1 \vee \cdots \vee A_p \vee B_1 \vee \cdots \vee B_q$*, where* $\overline{x}$ *are all the free variables. The bar* $\mid$ *has no logical meaning. Its only purpose is to separate the two types of atoms.*

A geometric formula that is not range restricted, can always be made range restricted by inserting suitable $\#_\mathbf{f}$ atoms into the left hand side. Atoms are always flat atoms, labeled with truth-values, as in [11]. It is shown in [4] and [3] that formulas in classical logic with partial functions ([2]) can be translated into sets of geometric formulas.

Having defined formulas, we can define interpretations.

Definition 2. *We define an* interpretation I *as a finite set of atoms of forms* $\#\, c$ *with* c *a constant, or form* $p_\lambda(c_1, \ldots, c_n)$*, where* $c_1, \ldots, c_n$ *are constants (repetitions allowed). Interpretations must be* range restricted *as well. This means that every constant* c *occurring in the interpretation must occur in an atom of form* $\#_\mathbf{t}\, c$*.*

Matching searches for false formulas. These are formulas whose premises $A_1, \ldots, A_p$ clash with I, while none of the B_j is true in I.

Definition 3. *Let* I *be an interpretation. Let* A *be a geometric literal. Let* Θ *be a substitution that assigns constants to variables, and that is defined on the variables in* A*.*

We say that $A\Theta$ conflicts *(or* is in conflict with*)* I *if* A *has form* $p_\lambda(x_1, \ldots, x_n)$*, and there is an atom of form* $p_\mu(x_1\Theta, \ldots, x_n\Theta) \in I$ *with* $\lambda \neq \mu$*,* A *has form* $x_1 \approx x_2$ *and* $x_1\Theta \neq x_2\Theta$*, or* A *has form* $\#_\mathbf{f}\, x$ *and* $(\#_\mathbf{t}\, x\Theta) \in I$*. If* B *is a geometric literal, s.t.* Θ *is defined on all variables of* B*, we say that* $B\Theta$ *is true in* I *if* **(1)** A *has form* $p_\lambda(x_1, \ldots, x_n)$ *and* $p_\lambda(x_1\Theta, \ldots, x_n\Theta) \in I$*,* **(2)** A *has form* $x_1 \approx x_2$ *and* $x_1\Theta = x_2\Theta$*,* **(3)** A *has form* $\#_\mathbf{t}\, x$ *and* $(\#_\mathbf{t}\, x\Theta) \in I$*, or* **(4)** A *has form* $\exists y\, B_\lambda(x_1, \ldots, x_n, y)$ *and there exists a constant* c*, s.t.* $B_\lambda(x_1\Theta, \ldots, x_n\Theta, c) \in I$*.*

In the definitions of truth and conflict, # is treated as a usual predicate.

Definition 4. *Let I be an interpretation. Let B be a geometric atom. Let Θ be a substitution that instantiates all free variables of B, and for which $B\Theta$ is not true in I. We define the* extension set *$E(B, \Theta)$ as follows:*

- *If B has form $p_\lambda(x_1, \ldots, x_n)$ or $\#_{\mathbf{t}}\ x$, then $E(B, \Theta) = \{B\Theta\}$.*
- *If B has form $x_1 \approx x_2$, then $E(B, \Theta) = \emptyset$.*
- *If B has form $\exists y\ B_\lambda(x_1, \ldots, x_n, y)$, then*

$$E(B, \Theta) = \bigcup \begin{Bmatrix} \{\ B\Theta\{y := e\} \mid c \in I\ \} \\ \{\ B\Theta\{y := \hat{c}\}\ \},\ \textit{for some } \hat{c} \notin I. \end{Bmatrix}$$

(By $c \in I$, we mean: c is a constant occuring in an atom in I)

Intuitively, if for a geometric formula $\phi = A_1, \ldots, A_p \mid B_1, \ldots, B_q$ and a substitution Θ, the $A_i\Theta$ are in conflict with I, while none of the $B_j\Theta$ is true in I, then $\phi\Theta$ is false in I. If there exist a B_j and an atom $C \in E(B_j, \Theta)$ that is not in conflict with I, then $\phi\Theta$ can be made true by adding C. If no such C exists, a conflict was found. If more than one C exists, the search algorithm has to backtrack through all possibilities. The search algorithm tries to extend an initial interpretation I into an interpretation $I' \supset I$ that makes all formulas true by backtracking as described above. At each stage of the search, it looks for a formula and a substitution that make the formula false. If no formula and substitution can be found, the current interpretation is a model. Otherwise, search continues either by extending I, or by backtracking. Details of the procedure are described in [5] for the two-valued case, and in [3] for the three-valued case. Experiments with the current three-valued version, and the previous two-valued version ([6]) show that the search for false formulas consumes nearly all of the resources of the prover.

Definition 5. *An instance of* the matching problem *consists of an interpretation I and a geometric formula $A_1, \ldots, A_p \mid B_1, \ldots, B_q$.*

Determine if there exists a substitution Θ that brings all A_i in conflict with I, and makes none of the B_j true in Θ. If yes, then return such substitution.

Example 1. Consider an interpretation I consisting of atoms

$$P_{\mathbf{t}}(c_0, c_0),\ P_{\mathbf{e}}(c_0, c_1),\ P_{\mathbf{t}}(c_1, c_1),\ P_{\mathbf{e}}(c_1, c_2),\ Q_{\mathbf{t}}(c_2, c_0).$$

The formula $\phi_1 = P_{\mathbf{f}}(X, Y),\ P_{\mathbf{f}}(Y, Z) \mid Q_{\mathbf{t}}(Z, X)$ can be matched in five ways:

$$\begin{array}{l} \Theta_1 = \{\ X := c_0,\ Y := c_0,\ Z := c_0\ \} \\ \Theta_2 = \{\ X := c_0,\ Y := c_0,\ Z := c_1\ \} \\ \Theta_3 = \{\ X := c_0,\ Y := c_1,\ Z := c_1\ \} \\ \Theta_4 = \{\ X := c_1,\ Y := c_1,\ Z := c_1\ \} \\ \Theta_5 = \{\ X := c_1,\ Y := c_1,\ Z := c_2\ \} \end{array}$$

The substitution $\Theta_6 = \{\ X := c_0,\ Y := c_1,\ Z := c_2\ \}$ would make the conclusion $Q_{\mathbf{t}}(Z, X)$ true.

Next consider the formula $\phi_2 = \ P_{\mathbf{f}}(X,Y),\ P_{\mathbf{t}}(Y,Z) \mid X \approx Y$.
The substitution $\Theta = \{\ X := c_0,\ Y := c_1,\ Z := c_2\ \}$ is the only matching of ϕ_2 into I.
Finally, the formula $\phi_3 = \ P_{\mathbf{t}}(X,Y) \mid \exists Z\ Q_{\mathbf{t}}(Y,Z)$ can be matched with $\Theta = \{\ X := c_0,\ Y := c_1\ \}$, and in no other way.

The first formula in Example 1 has five matchings. In case more than one matching exists, it matters for the continuation of the search process, which matching is returned. The prover analyses which ground atoms in I contributed to the matching, and only those will be considered in backtracking. In general, the set of conflicting atoms in I should be as small as possible, and should depend on as few as possible decisions. (Decisions in the sense of propositonal reasoning ([10]) The simplest solution to finding the best matching would be to define some preference relation $\preceq$ on matchings, enumerate all matchings, and use $\preceq$ to choose the best one. Unfortunately, this method is not practical because the number of matchings can be extremely high. We will address this problem in Section 5.

Even, if one is interested in the decision problem only, it is still intractable because the decision problem is NP-complete. This can be shown by a simple reduction from SAT.

In this paper, we will introduce an algorithm for efficiently solving the matching problem. Parts of the algorithm have been incompletely implemented in the two-valued version of **Geo** ([6]). The three-valued version of **Geo** that took part in CASC 25 (see [13]) uses a very naive implementation of matching.

In the rest of this paper, we will translate the matching problem into a structure called *generalized constraint satisfaction problem* (GCSP). The generalization consists of the fact that it contains additional constraints that a solution must not make true. These constraints correspond to the conclusions of the geometric formula that one is trying to match.

After that, we give in Section 3 a backtracking algorithm for solving GCSP, which is based on local consistency checking and backtracking. It makes use of a data structure that we call *choice stack*. Choice stacks can be used for controlling the backtracking process, but also for keeping check of changes that occur during local consistency checking, and which may induce further changes.

In Section 4 we add lemma learning to the algorithm of Section 3. In Section 5, we show how the algorithm of Section 3, or any other algorithm for solving GCSP, can be used for finding optimal matchings.

2 Translation into Generalized Constraint Satisfaction Problem

We introduce the generalized constraint satisfaction problem, and show how an instance of the matching problem can be translated. It is 'generalized' because there are additional, negative constraints (called *blockings*) which a solution is not allowed to satisfy. The blockings originate from translations of the $B_1, \ldots, B_q$.

Definition 6. *A* substlet *s is a (small) substitution. We usually write s in the form $\overline{v}/\overline{c}$, where $\overline{v}$ is a sequence of variables without repetitions, and $\overline{c}$ is a sequence of constants of same length as $\overline{v}$.*

We say that two substlets $\overline{v}_1/\overline{c}_1$ and $\overline{v}_2/\overline{c}_2$ are in conflict *if there exist i, j s.t. $v_{1,i} = v_{2,j}$ and $c_{1,i} \neq c_{2,j}$.*

If $\overline{v}_1/\overline{c}_1, \ldots, \overline{v}_n/\overline{c}_n$ is a sequence of substlets not containing a conflicting pair, then one can read off a substitution as follows: $\bigcup\{\overline{v}_1/\overline{c}_1, \ldots, \overline{v}_n/\overline{c}_n\} = \{v_{i,j} := c_{i,j} \mid 1 \leq i \leq n,\ 1 \leq j \leq \|\overline{v}_i\|\}$.

If Θ is a substitution and $s = \overline{v}/\overline{c}$ is a substlet, we say that Θ makes s true *if every $v_i := c_i$ is present in Θ.*

We say that Θ and s are in conflict if there is a v_i/c_i with $1 \leq i \leq \|v\|$, s.t. $v_i\Theta$ is defined and distinct from c_i.

A clause *C is a finite set of substlets. We say that a substitution Θ* makes *C* true *(notation $\Theta \models C$) if Θ makes a substlet $(\overline{v}/\overline{c}) \in C$ true. We say that Θ* makes *C* false *(notation $\Theta \models \neg C$) if every substlet $(\overline{v}/\overline{c}) \in C$ is in conflict with Θ. In the remaining case, we call C* undecided by *Θ.*

Definition 7. *A* generalized constraint satisfaction problem *(GCSP) is a pair of form (Σ^+, Σ^-) in which Σ^+ is a finite set of clauses, and Σ^- is a finite set of substlets.*

A solution of (Σ^+, Σ^-) is a substitution Θ s.t. every clause in Σ^+ is true in Θ and there is no $\sigma \in \Sigma^-$, s.t. $\Theta \models \sigma$.

Definition 8. *Let (Σ^+, Σ^-) a GCSP. We call (Σ^+, Σ^-)* range restricted *if for every variable v that occurs in a substlet $\sigma \in \Sigma^-$, there exists a clause $c \in \Sigma^+$ s.t. every substlet $s \in c$ has v in its domain.*

We now explain how a matching instance is translated into a generalized constraint satisfaction problem.

Definition 9. *Assume that I and $\phi = A_1, \ldots, A_p \mid B_1, \ldots, B_q$ together form an instance of the matching problem. The* translation *(Σ^+, Σ^-) of (I, ϕ) into* into *GCSP is obtained as follows:*

- *For every A_i, let $\overline{v}_i$ denote the variables of A_i. Then Σ^+ contains the clause*

$$\{\ \overline{v}_i/\overline{v}_i\Theta \mid A_i\Theta \text{ is in conflict with } I\ \}.$$

- *For every B_j, let $\overline{w}_j$ denote the variables of B_j. For every Θ that makes $B_j\Theta$ true in I, Σ^- contains the substlet $\overline{w}_j/(\overline{w}_j\Theta)$.*

Theorem 1. *A matching instance (I, ϕ) has a matching iff its corresponding GCSP has a solution.*

In theory, the set of blockings Σ^- is redundant, because a blocking σ can always be replaced by a clause as follows: Let σ be a blocking, let $\overline{v}$ be its variables. Define $\sigma_1 = \sigma$, and let $\sigma_2, \ldots, \sigma_n \in \Sigma^-$ be the blockings whose domain is also $\overline{v}$. One can replace $\sigma_1, \ldots, \sigma_n$ by the clause $\{\ \overline{v}/\overline{c} \mid \overline{v}/\overline{c}$ conflicts all $\sigma_i\ (1 \leq i \leq n)\ \}$.

We prefer to keep Σ^-, because in general, the resulting clause is quadratic in the size of $\sigma_1, \ldots, \sigma_n$. For example, if $\sigma_1, \ldots, \sigma_n$ result from an equality $X \approx Y$, then each σ_i has form $(X, Y)/(c_i, c_i)$. The resulting clause $C = \{(X, Y)/(c_i, c_j) \mid i \neq j\}$ has size $n(n-1)$.

Clauses resulting from a matching problem have the following trivial, but important property:

Lemma 1. *Let (Σ^+, Σ^-) be obtained by the translation in Definition 9. Let $s_1, s_2 \in C \in \Sigma^+$. Then either $s_1 = s_2$, or s_1 and s_2 are in conflict with each other.*

Lemma 1 holds because s_1 and s_2 have the same domain.

Example 2. In Example 1, the matching problem (I, ϕ_1) can be translated into the GCSP below. The clauses are above the horizontal line, and the blockings are below it. Because substlets in the same clause always have the same variables, we write the variables of a clause only once.

$$\begin{array}{l} (X, Y) \ / \ (c_0, c_0) \mid (c_0, c_1) \mid (c_1, c_1) \mid (c_1, c_2) \\ (Y, Z) \ / \ (c_0, c_0) \mid (c_0, c_1) \mid (c_1, c_1) \mid (c_1, c_2) \\ \hline (X, Z) \ / \ (c_0, c_2) \end{array}$$

Translating (I, ϕ_2) results in:

$$\begin{array}{l} (X, Y) \ / \ (c_0, c_0) \mid (c_0, c_1) \mid (c_1, c_1) \mid (c_1, c_2) \\ (Y, Z) \ / \ (c_0, c_1) \mid (c_1, c_2) \\ \hline (X, Y) \ / \ (c_0, c_0) \\ (X, Y) \ / \ (c_1, c_1) \\ (X, Y) \ / \ (c_2, c_2) \end{array}$$

Translation of (I, ϕ_3) results in:

$$\frac{(X, Y) \ / \ (c_0, c_1) \mid (c_1, c_2)}{(Y) \ / \ (c2)}$$

Before one runs any algorithms on a GCSP, it is useful to do some simplifications. If the GCSP contains a propositional clause (not containing any variables), this clause is either the empty clause, or a tautology. In the first case, the problem is trivially unsolvable. In the second case, the clause can be removed.

Similarly, if Σ^- contains a propositional blocking, then (Σ^+, Σ^-) is trivially unsolvable. Such blockings originate from a B_j that is purely propositional, or that has form $\exists y \ P_\lambda(y)$.

A third important preprocessing step is *removal of unit blockings*. Let $\sigma \in \Sigma^-$ be a blocking whose domain is included in the domain of some clause $C \in \Sigma^+$. In that case, one can remove every substlet $\overline{v}/\overline{c}$ from C, that has $\bigcup\{\overline{v}/\overline{c}\} \models \sigma$. If this results in C being empty, then (Σ^+, Σ^-) trivially has no solution. If no $\overline{v}/\overline{c}$ in any clause $C \in \Sigma^+$ implies σ, then σ can be removed from Σ^-, because of Lemma 1.

Applying removal of unit blockings to the translation of (I, ϕ_2) above results in

$$\underline{\begin{array}{l}(X,Y)\ /\ (c_0,c_1)\mid(c_1,c_2)\\(Y,Z)\ /\ (c_0,c_1)\mid(c_1,c_2)\end{array}}$$

It is worth noting that removal of propositional blockings can be viewed as a special case of removal of unit blockings.

A GCSP can be solved by backtracking, similar to SAT solving. A backtracking algorithm for GCSP can be either variable or clause based. A variable based algorithm maintains a substitution Θ which it tries to extend into a solution. It backtracks by picking a variable V and trying to assign it in all possible ways. It considers a new assignment when Θ makes a clause $C \in \Sigma^+$ false, or a blocking $\sigma \in \Sigma^-$ true.

A clause based algorithm maintains a consistent set S of substlets (whose union defines a substitution). It backtracks by picking an undecided clause $C \in \Sigma^+$, and consecutively adding all atoms that are consistent with S into S. It backtracks when there is a clause C all of whose atoms are in conflict with S, or when $\bigcup S$ makes a blocking true.

Our experiments suggest that there is no significant difference in performance, nor in programming effort between the two variants. We will stick with clause based algorithms, because it seems that they can be more easily combined with local consistency checking.

Local consistency checking (see [7, 9, 12]) is a pre-check that comes in many variations. They all have in common that one enumerates small subsets Σ' of Σ^+, for each of the Σ' generates all solutions, and then checks for each substlet occurring in a $C \in \Sigma'$ whether it occurs in a solution for Σ'. If some substlet $s \in C$ does not occur in a solution for Σ', then it certainly does not occur in a solution for Σ.

Experiments with **geo** show that a simple filtering using all subsets Σ' of size 2 whose clauses share a variable, a priori rejects a large fraction of matching instances. On those that are not immediately rejected, the next stage of search based on backtracking is much more efficient.

In [7], (Chapter 3) local consistency checking is done with subsets of variables (instead of clauses). Using subsets of two variables is called checking for *arc consistency*, while considering subsets of three variables is called *path consistency*. In general, using bigger subsets is a more effective precheck, but also more costly because it approximates the original problem. It still may be worth it to try subsets of bigger size than size 2.

3 Matching Using Choice Stacks

We first present the matching algorithm without learning, and introduce learning in the next section. The non-learning algorithm is based on a combination of local consistency checking and backtracking. At each level, it first tries to reduce the set of clauses by local consistency checking. During local consistency checking it checks all subsets of size two for variable conflicts, and subsets of arbitrary

size, of which all but one are unit, for conflicts based on blockings. After local checking has been exhaustively applied, it picks a clause, splits it into two parts, and backtracks through the two resulting clauses.

Local consistency checking is a change-driven process. Whenever some clause C gets replaced by a $C' \subset C$, one has to check for all clauses D that share a variable with C', whether D now contains substlets that are in conflict with every substlet in C'. If yes, then D can be replaced by a $D' \subset D$, which in turn may result in further replacements. This means that the search algorithm needs to maintain a set of changed clauses, and using this set of changed clauses, check which more clauses it can change. When the set of changed clauses is empty, it either has a solution, or it needs to backtrack (which introduces a new single, changed clause.) Instead of maintaining a set of changed clauses, we use a data structure that we call choice stack.

Definition 10. *A* choice stack $\overline{C}$ *is a data structure consisting of a sequence of clauses* $C_1, \ldots, C_n$.

A choice stack supports refinement *of clauses: Refining* C_i *into* C' *means replacing* $\overline{C} = C_1, \ldots, C_n$ *by* $C_1, \ldots, [C_i], \ldots, C_n, C'$, *where* $C' \subset C_i$, *and the square brackets indicate that* C_i *has been refined. We write* $\overline{C}[i/C']$ *for the resulting choice stack.*

Restoring a choice stack to size n *means removing all clauses at positions* $> n$, *and removing all square brackets that resulted from adding the clauses at positions* $> n$. *We use a predicate* A_i *for querying whether the* i*-th clause is* actual *(is not in brackets). We use the notation* R_i *for the original, initial clause from which* C_i *is obtained by successive refinements.*

The size *of a choice stack is the total number of clauses in it (with or without brackets).*

Given a choice stack $\overline{C}$ *and a substitution* Θ, *we say that* Θ agrees with $\overline{C}$ *if* Θ *does not make any* C_i *false.*

Choice stacks can be efficiently implemented without need to copy clauses. They support all features needed by a search algorithm based on backtracking and local consistency checking. Change driven inspection is implemented by the fact that changed clauses are blocked and reinserted at a higher position. If one starts at the position of the latest change, and iteratively proceeds towards the end of the choice stack, checking all clauses on positions i that have A_i true, one will exhaustively inspect all changes.

Definition 11. *If* $\overline{v}/\overline{c}$ *is a substlet, and* C *a clause, we call* $\overline{v}/\overline{c}$ *in conflict with* C *if* $\overline{v}/\overline{c}$ *is in conflict with every* $s \in C$.

Definition 12. *A call to* **findmatch**$(\overline{C}, k, \Sigma^-)$ *either returns* $\bot$, *or constructs a solution of the GCSP* $(\overline{C}, \Sigma^-)$. *For simplicity, we assume that* **findmatch** *stops when it finds a solution, instead of returning it.* **findmatch**$(\overline{C}, k, \Sigma^-)$ *is recursively defined as follows:*

LOCAL: *As long as* $k \leq \|\overline{C}\|$, *do the following: If* A_k *holds, then*

LOC-CONFL: *For every C_i for which A_i holds, and which shares a variable with C_k, partition C_i into two parts as follows:*

$$C_i^- = \{s \in C_i \mid s \text{ is in conflict with } C_k\},$$
$$C_i^+ = \{s \in C_i \mid s \text{ is not in conflict with } C_k\}.$$

If $C_i^+ = \emptyset$, then return $\bot$. Otherwise, refine $\overline{C}$ into $\overline{C}[i/C_i^+]$.

LOC-BLOCKING: *If $\|C_k\| = 1$, then define $\Theta = \bigcup\{s_j \mid 1 \leq j \leq \|\overline{C}\|$ and C_j has form $\{s_j\}\}$. (Note that the unique element of C_k also contributes to Θ.) For every $\sigma \in \Sigma^-$ that shares a variable with C_k, for every C_i, for which A_i holds and which shares a variable with σ, partition C_i into two parts as follows:*

$$C_i^- = \{s' \in C_i \mid \Theta \cup \bigcup\{s'\} \models \sigma\},$$
$$C_i^+ = \{s' \in C_i \mid \Theta \cup \bigcup\{s'\} \not\models \sigma\}.$$

If $C_i^+ = \emptyset$, then return $\bot$. Otherwise, refine $\overline{C}$ into $\overline{C}[i/C_i^+]$.

Assign $k = k + 1$.

SOLUTION: *If all i for which A_i holds, have $\|C_i\| = 1$, then $\Theta = \bigcup\{s_j \mid 1 \leq j \leq \|\overline{C}\|$ and C_j has form $\{s_j\}\}$ is a solution.*

BACKTRACK: *Otherwise, pick an i for which A_i holds, and which has $\|C_i\| > 1$. Partition C_i into two parts C_1, C_2, s.t. neither C_1 nor C_2 is empty. Recursively call* **findmatch**$(\overline{C}[i/C_1],\ \|\overline{C}\|,\ \Sigma^-)$. *If this call does not result in a solution, then also call* **findmatch**$(\overline{C}[i/C_2],\ \|\overline{C}\|,\ \Sigma^-)$.

Before **findmatch** *can be called on (Σ^+, Σ^-), check for propositional clauses, propositional blockings and apply unit blocking removal. Let (Σ'^+, Σ'^-) be the result. Call* **findmatch**$(\Sigma'^+, 1, \Sigma'^-)$.

Algorithm **findmatch** is similar to DPLL in that it tries to postpone backtracking as long as possible by giving preference to deterministic reasoning. It differs from DPLL, because it is not based on interpretations (which would be substitutions Θ in our case.) Instead of gradually building an interpretation by adding assignments, algoritm **findmatch** gradually refines a set of clauses into an interpretation by removing substlets from the clauses, until every clause is unit.

Deterministic reasoning is done by deleting substlets from clauses that are in conflict with one of the other clauses. When deterministic reasoning fails to solve the problem, we pick a non-unit clause C_i, and remove some part C_1 from it, and apply deterministic reasoning again. If this fails to give a solution, we replace C_1 by $C \backslash C_1$.

4 Matching with Conflict Learning

The matching algorithm in the two-valued version of Geo ([6]) was already equipped with a weak form of conflict learning. Before releasing it, we had experimented with naive matching, the algorithm in [8], and a lot of ad hoc methods.

Matching with conflict learning is the only approach that gives acceptable performance. Despite this, matching is still a critical operation in the two-valued version of Geo, which will need significant improvement in the three-valued version. The algorithm in the current paper tries to obtain this in several, important ways: Firstly, algorithm **findmatch** mixes local consistency checking with backtracking. The algorithm in the two-valued version never attempted any deterministic reasoning. It always backtracked on a randomly picked variable. Secondly, in the two-valued version of Geo, lemmas have form $v_1/c_1, \ldots, v_n/c_n \rightarrow \bot$, i.e. they consist of a single, negated substlet. The lemmas that we will introduce shortly, are more expressive because they are positive. A negative substlet refutes only the substlets that it is included in, while a positive substlet rejects all substlets that it conflicts with. We hence expect that a single lemma will reject more matching attempts, and that, as a consequence, less lemmas will be generated. Thirdly, the algorithm in the two-valued version of Geo is unable to find optimal solutions. It always stops on the first solution, which frequently causes unpredictable behaviour. In Section 5, we give an algorithm that will turn every algoritm that can find some solution, into an algorithm that can find an optimal solution.

Definition 13. *A* lemma *is defined in the same way as a clause. For a substitution Θ, the notions of truth, falsehood, and undecidedness are defined in the same way as for clauses.*

We distinguish between clauses and lemmas because they serve different functions in the algorithm, which makes it useful to implement them as different classes. There is also a technical distinction, namely that all substlets in a clause always have the same domain, while in a lemma they can be different.

Definition 14. *Let (Σ^+, Σ^-) be a GCSP. Let λ be a lemma. We say that λ is* valid in *(Σ^+, Σ^-) if every solution Θ of (Σ^+, Σ^-) makes λ true.*

Let $\overline{C}$ be a choice stack, let λ be a lemma. We say that $\overline{C}$ makes λ false if λ is false in every substitution Θ that agrees with $\overline{C}$.

If $\overline{C}$ is choice stack, and there exists a valid lemma that is false in $\overline{C}$, then there exists no solution Θ of (Σ^+, Σ^-) that agrees with $\overline{C}$, and one can backtrack.

Our extended matching algorithm will be similar to DPLL with conflict learning. It derives valid lemmas that are used to guide the search process. For the derivation of lemmas, we use two variants of lemma resolution that we will introduce shortly.

There is a technical complication arising from the fact that algorithm **findmatch** is not based on substitutions, but on choice stacks. A substitution can play the same role as an interpretation in DPLL, and checking whether a lemma is false in a substitution is easy. A choice stack lazily refines a set of clauses into a substitution. This has the advantage that commitment can be postponed, but it has a major disadvantage that checking falsehood of a lemma becomes harder. In fact, if the choice stack $\overline{C}$ has no solutions, then every lemma is false in $\overline{C}$, which renders checking falsehood NP-complete. It follows that one needs a stronger notion then falsehood, one that one can actually use:

Definition 15. *Let $\overline{C} = (C_1, \ldots, C_n)$ be a choice stack. Let λ be a lemma. We call λ* a conflict lemma of $\overline{C}$ *if for every substlet $s \in \lambda$, there is a C_i in $\overline{C}$, such that s is in conflict with C_i.*

Checking whether a given λ is a conflict lemma is a cheap operation. In addition, one can use a watching scheme based on the changes in $\overline{C}$, and recheck only the lemmas whose watched substlets might be affected by a refinement. (See [10]). Conflict lemmas are created by resolution rules:

Definition 16. *Let λ_1 and λ_2 be lemmas. Let $\mu_1 \subseteq \lambda_1$. We define* $\mathrm{RES}(\lambda_1, \mu_1, \lambda_2)$ *the* conflict resolvent of λ_1 *and* λ_2 based on μ_1. *First define μ_2 and μ_1' as follows:*

$$\begin{cases} \mu_2 = \{\ (\overline{v}_2/\overline{c}_2) \in \lambda_2 \mid (\overline{v}_2/\overline{c}_2)\ \textit{conflicts every}\ (\overline{v}_1/\overline{c}_1) \in \mu_1\ \}, \\ \mu_1' = \{\ (\overline{v}_1/\overline{c}_1) \in \lambda_1 \mid (\overline{v}_1/\overline{c}_1)\ \textit{conflicts every}\ (\overline{v}_2/\overline{c}_2) \in \mu_2\ \}. \end{cases}$$

Then $\mathrm{RES}(\lambda_1, \mu_1, \lambda_2) = (\lambda_1 \backslash \mu_1') \cup (\lambda_2 \backslash \mu_2)$.

Suppose that one wants to resolve $\lambda_1 = \{\ (x,y)/(1,2),\ (x,y)/(2,1),\ (x,y)/(3,3)\ \}$ with $\lambda_2 = \{\ (y,z)/(1,2),\ (y,z)/(2,1)\ \}$ based on $\mu_1 = \{\ (x,y)/(1,2)\ \}$. In that case, $\mu_2 = \{\ (y,z)/(1,2)\ \}$. It turns out that $(x,y)/(3,3)$ can also resolve with μ_2, so we have $\mu_1' = \{\ (x,y)/(1,2),\ (x,y)/(3,3)\ \}$. The resolvent is $\{\ (x,y)/(2,1),\ (y,z)/(1,2)\ \}$.

Definition 17. *Let $\sigma \in \Sigma^-$. Let $C_1, \ldots, C_n$ be a sequence of clauses. Let $\mu_1 \subseteq C_1, \ldots, \mu_n \subseteq C_n$ be subsets of $C_1, \ldots, C_n$, s.t. for every sequence of substlets $s_1 \in \mu_1, \ldots, s_n \in \mu_n$, either*

1. *Two s_i, s_j $(1 \leq i, j \leq n)$ are in conflict, or*
2. *$\bigcup\{s_1, \ldots, s_n\} \models s$.*

Then $(C_1 \backslash \mu_1) \cup \cdots \cup (C_n \backslash \mu_n)$ is a σ-resolvent *of $C_1, \ldots, C_n$. We will write* $\mathrm{RES}(C_1, \ldots, C_n, \mu_1, \ldots, \mu_n, s)$ *for the result.*

It is easy to show that conflict resolution and σ-resolution are valid reasoning rules. It would be possible to merge conflict resolution and σ-resolution into a single rule. One could also generalize σ-resolution to sets of blockings. Although we may do this the implementation, we will not merge the rules in the paper for clarity of presentation.

In order to extend algorithm **findmatch**, so that it will generate conflict lemmas, we assume a set of conflict lemmas Λ, which is initially empty. Whenever the modified algorithm **findmatch** backtracks, it extends Λ with a conflict lemma λ that conflicts the current choice stack $\overline{C}$. The following extensions are be made to **findmatch**:

- If at any stage, there is a lemma $\lambda \in \Lambda$, that is in conflict with $\overline{C}$, the algorithm backtracks.
- Assume that the algorithm passes through **LOC-CONFL**. If $C^+ = \emptyset$, the algorithm can insert the original version R_i of C_i into Λ. This is a conflict

lemma of C, because it can be written as $C_i \cup (R_i \backslash C_i)$. Choice stack $\overline{C}$ contains the clause C_i, with which every $s \in (R_i \backslash C_i)$ is in conflict, by Lemma 1. Every $s \in C_i$ is in conflict with C_k, which follows from the fact that $C_i = C_i^-$. If $C^+ \neq \emptyset$, algorithm **findmatch** continues, and by induction, one can assume that it inserts a conflict lemma of $\overline{C}[i/C_i^+]$ into to Λ.
If λ already is a closing lemma of $\overline{C}$, nothing needs to be done. Otherwise, let $\mu \subseteq \lambda$ be the substlets in λ that are in conflict with C_i^+. Let $\lambda' = \lambda \backslash \mu$. Every substlet in λ' is in conflict with $\overline{C}$, because λ conflicts $\overline{C}[i/C_i^+]$. We can insert $\mathrm{RES}(\lambda, \mu, C_i)$ into Λ. The resolvent has form $(\lambda \backslash \mu') \cup (C_i \backslash \mu_2)$, where $\mu \subseteq \mu'$ and $\mu_2 \subseteq C_i^+$. It follows that $\lambda \backslash \mu' \subseteq \lambda'$ and $(C_i \backslash \mu_2) \subseteq C_i^-$, which is by its construction in conflict with C_k.

– Assume that clause C_i was partitioned into two parts (C_i^+, C_i^-) at **LOC-BLOCKING**. We first apply σ-resolution: Let

$$S \subseteq \{ s_j \mid 1 \leq j \leq \|\overline{C}\| \text{ and } C_j \text{ has form } \{s_j\} \}$$

be the set of singletons that contribute to truth of the blocking σ, i.e. for every $s' \in C_i^-$, $\bigcup S \cup \{s'\} \models \sigma$, and there is no $S' \subset S$ for which this is still the case.
Let $j_1, \ldots, j_m$ be an enumeration of the clauses from which the elements of S originate. We can construct the σ-resolvent

$$\lambda_\sigma = (R_{j_1} \backslash \{s_{j_1}\}) \cup \cdots \cup (R_{j_m} \backslash \{s_{j_m}\}) \cup (R_i \backslash C_i^-).$$

It follows from the construction of S, that λ_σ is indeed a σ-resolvent. By Lemma 1, the elements in each $R_{j_z} \backslash \{s_{j_z}\}$ are in conflict with $\{s_{j_z}\}$. By the same lemma, the substlets in $R_i \backslash C_i^-$ that are not in conflict with C_i, are contained in C_i^+.
If C_i^+ is empty, we are done. Otherwise, algorithm **findmatch** continues, and by induction it inserts a conflict lemma λ of $\overline{C}[i/C^+]$ into Λ. If λ is a closing lemma of $\overline{C}$, then nothing needs to be done. (and we constructed λ_σ without reason.)
If λ is not a closing lemma of $\overline{C}$, we can define $\mu \subseteq \lambda$ as the substlets in λ that are in conflict with C_i^+. Construct $\lambda' = \mathrm{RES}(\lambda, \mu, \lambda_\sigma)$ and insert λ' into Λ. The argument that λ' is a conflict lemma, is similar to the case for LOC-CONFLICT.

– At **BACKTRACK**, one can assume by induction that the first recursive call inserts a closing lemma λ_1 of $\overline{C}[i/C_1]$ into Λ. If λ_1 is a closing lemma of C, nothing more needs to be done. Otherwise, the second recursive call will insert a closing lemma λ_2 of $\overline{C}[i/C_2]$ into Λ. Again, if λ_2 is a closing lemma of C, we are done. Otherwise, we can insert $\lambda' = \mathrm{RES}(\mathrm{RES}(R_i, C_1, \lambda_1), C_2, \lambda_2)$ into Λ. The proof that λ' is indeed a closing lemma is analogous to the proof for LOC-CONFLICT.

Example 3. Consider the choice stack $\overline{C}$, consisting of $C_1 = \{ (X, Y) \,/\, (c_0, c_1) \mid (c_1, c_2) \}$ and $C_2 = \{ (Y, Z) \,/\, (c_0, c_1) \mid (c_1, c_2) \}$. At $k = 1$, C_2 will be refined into $C_3 = \{ (Y, Z) \,/\, (c_1, c_2) \}$. At $k = 2$, we have A_2 is false, so it is skipped. At

$k = 3$, the clause C_1 is refined into $C_4 = \{\ (X, Y)\ /\ (c_1, c_2)\ \}$. At $k = 4$, nothing is done, and after that we reach **SOLUTION**.

Example 4. Consider $C_1 = \{\ (X, Y)\ /\ (c_0, c_1) \mid (c_1, c_2)\ \}$, $C_2 = \{\ (Y, Z, T)\ /\ (c_1, c_2, c_3) \mid (c_1, c_2, c_4)\ \}$ with a blocking $\sigma = (X, Z)\ /\ (c_0, c_2)$.

At $k = 1$, nothing is changed. At $k = 2$, clause C_1 is refined into $C_3 = \{\ (X, Y)\ /\ (c_0, c_1)\ \}$ by LOC-CONFLICT. At $k = 3$, clause C_2 can be refined into the empty clause by LOC-BLOCKING.

In order to obtain a conflict lemma, we σ-resolve $R_3 = C_1$ with $R_2 = C_2$. The result is $\lambda_1 = \{\ (X, Y)\ /\ (c_1, c_2)\ \}$ which conflicts (C_2, C_3). Back at level 2, the returned lemma λ_1 is not a conflict lemma of (C_1, C_2). It can resolve with R_1 and the result is $\{\ \}$.

5 Finding Optimal Matchings

In this section we address the problem of finding optimal matchings. For the effectiveness of geometric resolution, it is important that a minimal matching is returned, in case more than one exists. A minimal matching is a matching that uses the smallest possible set of assumptions. In terminology of DPLL, assumptions represent decision levels. The assumptions contributing to a conflict represent choice options, which will be replaced by other options during backtracking. In addition to being as few as possible, assumptions at a lower decision level should always be preferred over assumptions at a higher decision level. The reason for this is the fact that in other branches of the search tree, there is a risk that more assumptions will be used, and when assumptions are at a lower level, there is less room to do this.

Definition 18. *Let I be an interpretation. An weight function α is a function that assigns finite subsets of natural numbers to the atoms of I.*

Let A be a geometric literal. Let Θ be a substitution such that $A\Theta$ is in conflict with I. Referring to Definition 3, we define $\alpha(\ p_\lambda(x_1, \ldots, x_n)\Theta, I) = \alpha(\ p_\mu(x_1\Theta, \ldots, x_n\Theta)\ )$, $\alpha(\ (x_1 \approx x_2)\Theta, I) = \{\}$, *and* $\alpha(\ (\#_{\mathbf{f}} x)\Theta, I) = \alpha(\ (\#_{\mathbf{t}} x\Theta)\ )$.

Definition 19. *Let I and $\phi = \ A_1, \ldots, A_p \mid B_1, \ldots, B_q$ together form an instance of the matching problem (Definition 5). Assume that Θ is a solution. The* weight of Θ, *for which we write $\alpha(I, \phi, \Theta)$, is defined as*

$$\bigcup \begin{cases} \{\ \alpha(A_i\Theta, I) \mid 1 \le i \le p\} \\ \{\ \alpha(C, I) \mid 1 \le j \le q,\ \ C \in E(B_j, \Theta),\ \textit{and } C \textit{ conflicts } I\ \} \end{cases}$$

Solving optimal matching means: First establish if (I, ϕ) has a solution. If it has, then find a solution Θ for which $\alpha(I, \phi, \Theta)$ is multiset minimal.

One could try to impose further selection criteria that are harder to explain and whose advantage is less evident.

Solving the minimal matching problem is non-trivial, because the number of possible solutions can be very large. The straightforward solution is to use some

efficient algorithm (e.g. the one in this paper) that enumerates all solutions, and keeps the best solution. Unfortunately, this approach is completely impractical because some instances have a very high number of solutions. One frequently encounters instances with $> 10^9$ solutions.

In order to find a minimal solution without enumerating all solutions, one can use any algorithm that stops on the first solution. It is used as follows: The first call is used to find out whether a solution exists. If not, then we are done. Otherwise, the algorithm is called again with its input restricted in such a way that it has to find a better solution than the previous, if one exists. One can continue doing this, until all possibilities to improve the solution have been exhausted. It can be shown that the number of calls needed to obtain an optimal solution is linear in the size of the assumption set of solution. In this way, it can be avoided that all solutions have to be enumerated.

Definition 20. *Let I be an interpretation that is equipped with a weight function α. Let $\phi = A_1, \ldots, A_p \mid B_1, \ldots, B_q$ be a geometric formula. Let α be a fixed set of natural numbers.*
We define the α-restricted translation of (I, ϕ) into (Σ^+, Σ^-) as follows:

- *For every A_i, let $\overline{v}_i$ be the variables of A_i. Then Σ^+ contains the clause*

$$\{\overline{v}_i/\overline{v}_i\Theta \mid A_i\Theta \text{ is in conflict with } I \text{ and } \alpha(A_i\Theta, I) \subseteq \alpha\}.$$

- *For each B_j, let $\overline{w}_j$ denote the variables of B_j. For every Θ that makes $B_j\Theta$ true in I, Σ^- contains the substlet $\overline{w}_j/(\overline{w}_j\Theta)$. In addition, if there exists a $C \in E(B_j, \Theta)$ that is in conflict with I and for which $\alpha(C, \Theta) \not\subseteq \alpha$, then Σ^- contains the substlet $\overline{w}_j/(\overline{w}_j\Theta)$.*

The α-restricted translation ensures that only conflicts involving atoms C with $\alpha(C) \subseteq \alpha$ are considered, and (independently of α), that no B_j is made true. The translation of Definition 9 can be viewed as a special case of α-restricted translation with $\alpha = \mathcal{N}$.

Theorem 2. *Let (Σ^+, Σ^-) be obtained by α-restricted translation of (I, ϕ). For every substitution Θ, Θ is a solution of (Σ^+, Σ^-) iff Θ is a solution of (I, ϕ), and it has $\alpha(I, \phi, \Theta) \subseteq \alpha$.*

Using α-restricted translation, we can define the **optimal** matching algorithm:

Definition 21. *Let* **solve**(Σ^+, Σ^-) *be a function that returns some solution of (Σ^+, Σ^-) if it has a solution, and $\perp$ otherwise.*

We define the algorithm **optimal**$(\ I, \phi\ )$ *that returns an optimal solution of (I, ϕ) if one exists and $\perp$ otherwise.*

- *Let (Σ^+, Σ^-) be the GCSP obtained by the translation of Definition 9. If Σ^+ contains an empty clause, then return $\perp$. If Σ^- contains a propositional blocking, then return $\perp$. Otherwise, remove unit blockings from (Σ^+, Σ^-). If this results in Σ^+ containing an empty clause, then return $\perp$.*

– *Let* $\Theta = \mathbf{solve}(\Sigma^+, \Sigma^-)$. *If* $\Theta = \bot$, *then* **return** $\bot$.
– *Let* $\alpha = \alpha(I, \phi, \Theta)$, *and let* $k := \sup(\alpha)$.
– *As long as* $k \neq 0$, *do the following:*
  • *Set* $k = k - 1$. *If* $k \in \overline{\alpha}$, *then do*
    ∗ *Let* $\alpha' = (\alpha \backslash \{k\}) \cup \{0, 1, 2, \ldots, k-1\}$.
    ∗ *Let* (Σ'^+, Σ'^-) *be the* α'*-restricted translation, of* (I, ϕ).
    ∗ *If* Σ^+ *contains an empty clause or* Σ^- *contains a propositional blocking, then skip the rest of the loop. Otherwise, remove the unit blockings from* (Σ^+, Σ^-). *If this results in* Σ^+ *containing the empty clause, then skip the rest of the loop.*
    ∗ *Let* $\Theta' = \mathbf{solve}(\Sigma^+, \Sigma^-)$. *If* $\Theta' \neq \bot$, *then set* $\Theta = \Theta'$ *and* $\alpha = \alpha(I, \phi, \Theta)$.
– *Now* Θ *is an optimal solution, so we can* **return** Θ.

Algorithm **optimal** first solves (I, ϕ) without restriction. If this results in a solution Θ, it checks for each $k \in \alpha(I, \phi, \Theta)$ if k can be removed. The invariant of the main loop is: There exists no $k' \geq k$ that occurs in $\alpha(I, \phi, \Theta)$, and no Θ', that is a solution of (I, ϕ) with $k' \notin \alpha(I, \phi, \Theta')$. In addition, the invariant $\alpha = \alpha(I, \phi, \Theta)$ is maintained.

Example 5. Assume that in Example 2, the atoms are weighted as follows:

$$\alpha(\ P_{\mathbf{t}}(c_0, c_0)\ ) = \{1\},\quad \alpha(\ P_{\mathbf{e}}(c_0, c_1)\ ) = \{2\},\quad \alpha(\ P_{\mathbf{t}}(c_1, c_1)\ ) = \{3\},$$
$$\alpha(\ P_{\mathbf{e}}(c_1, c_2)\ ) = \{4\},\quad \alpha(\ Q_{\mathbf{t}}(c_2, c_0)\ ) = \{5\}.$$

We have $\alpha(I, \phi_1, \Theta_1) = \{1\}$, $\alpha(I, \phi_1, \Theta_2) = \{1, 2\}$, and $\alpha(I, \phi_1, \Theta_5) = \{2, 3\}$. If Θ_5 is the first solution generated, **solve** will construct the $\{1, 2\}$-restricted translation of (I, ϕ_2), which equals

$$\frac{\begin{array}{l}(X, Y)\ /\ (c_0, c_0) \mid (c_0, c_1) \\ (Y, Z)\ /\ (c_0, c_0) \mid (c_0, c_1)\end{array}}{(X, Z)\ /\ (c_0, c_2)}$$

If the next solution found is Θ_2, then **solve** will construct the $\{1\}$-restricted translation

$$\frac{\begin{array}{l}(X, Y)\ /\ (c_0, c_0) \\ (Y, Z)\ /\ (c_0, c_0)\end{array}}{(X, Z)\ /\ (c_0, c_2)}$$

whose only solution is Θ_1.

6 Conclusions

The problem of matching a geometric formula into an interpretation is currently the most time consuming part of implementations of geometric resolution. We gave a method for solving the matching problem by translating it into GCSP, and by providing efficient algorithms for GCSP. At the moment of submission,

there is no complete implementation, but different components have been implemented in different versions of the prover. A primitive version of learning was present in **Geo** 2007. The current, non-released, three-valued version of **Geo** uses translation into GCSP, together with the combination of **findmatch** without learning and **optimal**.

One might argue that a calculus that uses an NP-complete problem as its basic operation is not viable, but there is room for interpretation: The complexity of the matching problem is caused by the fact that as result of flattening, geometric formulas and interpretations have DAG-structure instead of tree-structure. This increased expressiveness may very well result in shorter proofs. Only experiments can determine which of the two effects will be stronger.

7 Funding

This work was supported by the Polish National Science Center (Narodowe Centrum Nauki) [grant number DEC-2011/03/B/ST6/00346].

References

1. Marc Bezem and Thierry Coquand. Automating coherent logic. In Geoff Sutcliffe and Andrei Voronkov, editors, *LPAR*, volume 3835 of *LNCS*, pages 246–260. Springer, 2005.
2. Hans de Nivelle. Classical logic with partial functions. *Journal of Automated Reasoning*, 47(4):399–425, 2011.
3. Hans de Nivelle. Theorem proving for classical logic with partial functions by reduction to Kleene logic. *Journal of Logic and Computation*, pages 1–42, 2014. (Accepted in April 2014, available from `http://logcom.oxfordjournals.org/`).
4. Hans de Nivelle. Theorem proving for logic with partial functions by reduction to Kleene logic. In Christoph Benzmüller and Jens Otten, editors, *Automated Reasoning in Quantified Non-Classical Logics (ARQNL) 2014*, pages 71–85. VSL Workshop Proceedings, 2014.
5. Hans de Nivelle and Jia Meng. Geometric resolution: A proof procedure based on finite model search. In John Harrison, Ulrich Furbach, and Natarajan Shankar, editors, *International Joint Conference on Automated Reasoning 2006*, volume 4130 of *Lecture Notes in Artificial Intelligence*, pages 303–317, Seattle, USA, August 2006. Springer.
6. Hans de Nivelle and Jia Meng. theorem prover Geo 2007f. Can be obtained from my homepage, September 2006.
7. Rina Dechter. *Constraint Processing*. Morgan Kaufmann Publishers, 2003.
8. Georg Gottlob and Alexander Leitsch. On the efficiency of subsumption algorithms. *Journal of the ACM*, 32(2):280–295, 1985.
9. Jérôme Maloberti and Michèle Sebag. Fast theta-subsumption with constraint satisfaction algorithms. *Machine Learning*, 55:137–174, 2004.
10. Joao Marques-Silva, Ines Lynce, and Sharad Malik. Conflict-driven clause learning SAT solvers. In Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh, editors, *Handbook of Satisfiability*, chapter 4, pages 131–153. IOS Press, 2009.

11. Neil Murray and Erik Rosenthal. Signed formulas: A liftable meta-logic for multiple-valued logics. In Jan Komorowski and Zbigniew Raś, editors, *Proceedings of the 7th International Symposium on Methodologies for Intelligent Systems (ISMIS)*, volume LNCS 689, pages 275–284, 1993.
12. Tobias Scheffer, Ralf Herbrich, and Fritz Wysotzki. Efficient theta-subsumption based on graph algorithms. In Stephen Muggleton, editor, *Inductive Logic Programming, 6th International Workshop, Selected Papers*, LNAI, pages 212–228. Springer Verlag Berlin, 1996.
13. Geoff Sutcliffe. The CADE ATP system competition. `http://www.cs.miami.edu/~tptp/CASC/25/`, August 2015.