Subsumption Algorithms for Three-Valued Geometric Resolution

Hans de Nivelle

Instytut Informatyki Uniwersytetu Wrocławskiego

Polska

IJCAR 2016, Coimbra, 30.06.2016.

Context

Theorem proving: We try to find formal proofs automatically, so that eventually the mathematicians will be unemployed.

Three-Valued Logic: Logics that can handle partial functions, subtyping, and polymorphism in a unified fashion. In 2010-2013, I developed Partial Classical Logic.

PCL is continuously connected to classical logic. The less you use of the advanced type system, the closer you get to classical logic. If you don't the extra features, it becomes classical logic.

Context (2)

- Geometric Resolution: A Proof Procedure Based on Finite Model search, (coauthored by Jia Meng), International Joint Conference on Automated Reasoning, 2006.

Aims are: To try something exotic (that's what scientists should do), to obtain a theorem proving method that is good at finding proofs, and good at finding counter models, and to obtain an implementation of PCL, which is good enough, so that somebody could actually consider using it.

Current implementation at CASC is three-valued, (although TPTP uses only $66\frac{2}{3}\%$ of its power.)

Partial Functions

Assume that $X = 1,\ Y = 1$: It follows that
$X^2 - Y^2 = X^2 - XY \Rightarrow (X - Y)(X + Y) = (X - Y)X \Rightarrow$
$X + Y = X \Rightarrow 1 + 1 = 1.$ (QVOD ERAT DEMONSTRANDVM)

(Wie deelt door nul is een ...)

```
template<typename A>
void misbehave( )
{ std::list<A> lst1, lst2;
  if( lst1.front( ) == list2.front( ))
     std::cout << "first elements equal!";
  else
     std::cout << "first elements differ!";
}
```

Matching

We want to know whether a 3-valued formula without functions is false in a 3-valued interpretation, e.g. formulas

$$\begin{aligned}\phi_1 = &\ P_{\mathbf{f}}(X,Y) \vee P_{\mathbf{f}}(Y,Z) \vee Q_{\mathbf{t}}(Z,X).\\ \phi_2 = &\ P_{\mathbf{f}}(X,Y) \vee P_{\mathbf{t}}(Y,Z) \vee X \approx Y.\\ \phi_3 = &\ P_{\mathbf{t}}(X,Y) \vee \exists Z\ Q_{\mathbf{t}}(Y,Z)\end{aligned}$$

in interpretation

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

$\phi_1[X := c_0, Y := c_0, Z := c_1]$ is false, can be repaired by adding $Q_{\mathbf{t}}(c_1,c_0)$.

$\phi_2[X := c_0, Y := c_1, Z := c_2]$ is false, irrepairable.

$\phi_3[X := c_1, Y := c_1]$ and $\phi_3[X := c_1, Y := c_2]$ are true.

Matching (2)

Geo tries to construct a satisfying interpretation of a set of 3-valued formulas by backtracking. Matching formulas into interpretations is the key operation.

Unfortunately:

theorem: Matching is very NP-complete.

The proof is easy, by reduction from SAT.

Efficiency of matching will decide about the fate of geometric resolution.

Substitutions, Substlets

Substitutions are defined a usual. A substlet is a (small) substitution. We write substlets in form $\overline{v}/\overline{c}$.

We say that substitutions Θ_1 and Θ_2 are consistent if for every variable v occurring in the domain of Θ_1 and Θ_2, we have $v\Theta_1 = v\Theta_2$.

If Θ_1 and Θ_2 are not consistent, they are in conflict.

If $\Theta_1, \ldots, \Theta_n$ is a set of pairwise consistent substitutions, then $\bigcup\{\Theta_1, \ldots, \Theta_n\}$ is also a substitution.

We say that Θ_1 implies Θ_2 if $\Theta_2 \subseteq \Theta_1$.

(By inheritance, these definitions also apply to substlets.)

Lemmas, Clauses

A lemma λ is a finite set of substlets. If the substlets in the lemma have the same variables, we call λ a clause.

A substitution Θ implies a lemma λ if there is a substlet $(\overline{v}/\overline{c}) \in \lambda$, s.t. Θ implies $\overline{v}/\overline{c}$.

A substitution Θ is in conflict/conflicts λ if Θ conflicts every $(\overline{v}/\overline{c}) \in \lambda$.

GCSP, Generalized Constraint Solving Problem

Definition A 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.

We assume that (Σ^+, Σ^-) is range-restricted: Every variable v occurring in an $(\overline{v}/\overline{c}) \in \Sigma^-$, also occurs in a clause $c \in \Sigma^+$.

A substitution Θ is a solution of (Σ^+, Σ^-) if Θ makes every clause $c \in \Sigma^+$ true, and Θ makes no $\sigma \in \Sigma^-$ true.

Example (1)

Matching

$$P_{\mathbf{f}}(X,Y) \vee P_{\mathbf{f}}(Y,Z) \vee Q_{\mathbf{t}}(Z,X)$$

into

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

gives:

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

Example (2)

Matching

$$P_{\mathbf{f}}(X, Y) \vee P_{\mathbf{t}}(Y, Z) \vee X \approx Y$$

into

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

gives:

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

Example (3)

Matching

$$P_{\mathbf{t}}(X, Y) \vee \exists Z \; Q_{\mathbf{t}}(Y, Z)$$

into

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

gives

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

Essential Preprocessing

1. If Σ^- contains a substlet that has no variables, then (Σ^+, Σ^-) is trivially unsolvable.
2. If Σ^+ contains a clause that has no variables, then (Σ^+, Σ^-) is trivially unsolvable if this clause is empty. Otherwise, the clause can be removed from Σ^+.
3. If Σ^+ contains a clause c containing a substlet λ that implies a $\sigma \in \Sigma^-$, then λ can be removed from c. If this makes c empty, then (Σ^+, Σ^-) is unsolvable.

A Simple Algorithm

Algorithm **solve**(**substitution**& Θ) is defined recursively. Θ is initially empty. It has access to (Σ^+, Σ^-) as global variable.

- Every clause $c \in \Sigma^+$ can be partitioned into c^+ and c^-, where c^- are the substlets that are in conflict with Θ, and c^+ are the remaining substlets.
- If there is a clause $c \in \Sigma^+$ with $c^+ = \emptyset$, then **fail**.
- If there are no clauses with unassigned variables, then report Θ as a solution.
- From the clauses containing unassigned variables, pick a clause c with minimal $\|c^+\|$. For each $(\overline{v}/\overline{c}) \in c^+$ do the following:
  Let $\Theta' = \Theta \cup (\overline{v}/\overline{c})$. If Θ' does not imply a substlet in Σ^-, then recursively call **solve**(Θ').
- When all substlets of c^+ fail to produce a solution, then **fail**.

The algorithm on the previous slide is easy to implement, and performs reasonably well, if one uses a good picking function.

What if we want more?

Clause learning in SAT-solving is very effective, so it seems reasonable to try to use similar techniques here:

(Σ^+, Σ^-) implies a lemma λ if every solution Θ of (Σ^+, Σ^-) implies λ.

Let **solve** derive **(1)** lemmas λ that are implied by (Σ^+, Σ^-), and for which **(2)** Θ conflicts λ.

We call a lemma with properties $\mathbf{1}, \mathbf{2}$ a conflict lemma.

Resolution

Let $\lambda_1, \ldots, \lambda_n$ be a sequence of lemmas. Let $\mu_1 \subseteq \lambda_1, \ldots, \mu_n \subseteq \lambda_n$ be chosen in such a way that for every sequence of substlets $s_1 \in \mu_1, \ldots, s_n \in \mu_n$, we have

1. Two s_{i_1}, s_{i_2} are in conflict, or
2. $\bigcup\{s_1, \ldots, s_n\}$ implies a $\sigma \in \Sigma^-$.

Then

$$(\lambda_1 \backslash \mu_1) \cup \cdots \cup (\lambda_n \backslash \mu_n)$$

is a resolvent of $\lambda_1, \ldots, \lambda_n$.

We write $\mathrm{RES}(\lambda_1, \ldots, \lambda_n; \mu_1, \ldots, \mu_n)$ for the resolvent.

Theorem: If (Σ^+, Σ^-) implies all of $\lambda_1, \ldots, \lambda_n$, then (Σ^+, Σ^-) implies $\mathrm{RES}(\lambda_1, \ldots, \lambda_n; \mu_1, \ldots, \mu_n)$.

Unrestricted resolution is NP-complete of course. (Given clauses $\lambda_1, \ldots, \lambda_n$, find $\mu_1, \ldots, \mu_n$, s.t. a resolvent is possible.)

If $\mu_1, \ldots, \mu_n$ are already known, then resolution is cheap.

Addition of Learning (2)

- If there is a clause $c \in \Sigma^+$ with $c^+ = \emptyset$, then c is a conflict lemma.
- By induction, each of the recursive calls **solve**(Θ') produces a conflict lemma of Θ'. Let $\lambda_1, \ldots, \lambda_m$ the conflict lemmas thus produced. If one of $\lambda_1, \ldots, \lambda_m$ is a conflict lemma of Θ, then nothing needs to be done.

  Otherwise, construct **RES**$(c, \lambda_1, \ldots, \lambda_m;\ c^+, \mu_1, \ldots, \mu_m)$, where each μ_i is the subset of λ_i, that shares a variable with c^+.

Simplifying the Problem: Filtering

Let (Σ^+, Σ^-) be a GCSP.

- Pick a clause $c \in \Sigma^+$.
- Pick a $C \subseteq (\Sigma^+ \backslash \{c\})$.
- Check for every substlet $s \in c$ if $(\{s\} \cup C, \Sigma^-)$ has a solution. If not, then remove s from c.
- c and C must form a circuit. This means that c and c_1 share a variable, c_1 and c_2 share a variable, $\cdots$, c_n and c share a variable.

Repeat this process until no further simplifications are possible.

Filtering (2)

In the paper is a clever datastructure for this, called choice stack.
It can also be used for implementing algorithm **solve**.

Reasonable choices for size of C are $1, 2, 3$.

If filtering results in an empty clause, the problem has no solution.

3-Filtering rejects 99% of the cases without search.

Combination of Filtering and Backtracking

Since filtering is succesful, it seems natural to mix filtering with backtracking in the following way:

1. Filter (Σ^+, Σ^-). If this results in an empty clause, then (Σ^+, Σ^-) has no solution.

2. Otherwise, pick a $c \in \Sigma^+$, and partition it into $m \geq 2$ parts $c_1, \ldots, c_m$. Define $\Sigma_i^+ = (\Sigma^+ \backslash c) \cup c_i$.

   Recursively apply this algorithm on

   $$(\Sigma_1^+, \Sigma^-), \ldots, (\Sigma_m^+, \Sigma^-).$$

Learning

The algorithm on the previous can be extended to include learning as well.

One needs an extended resolution rule:

Let $\lambda_1, \ldots, \lambda_n$ be a sequence of lemmas. Let $\mu_1 \subseteq \lambda_1, \ldots, \mu_n \subseteq \lambda_n$.

Let

$$S = \{ \bigcup\{s_1, \ldots, s_n\} \left| \begin{array}{l} \text{no } s_{i_1}, s_{i_2} \text{ are in conflict, and} \\ \bigcup\{s_1, \ldots, s_n\} \text{ does not imply a } \sigma \in \Sigma^- \\ s_1 \in \mu_1, \ldots, s_n \in \mu_n \end{array} \right. \}$$

Then

$$S \cup (\lambda_1 \backslash \mu_1) \cup \cdots \cup (\lambda_n \backslash \mu_n)$$

is a resolvent of $\lambda_1, \ldots, \lambda_n$.

Conflict Lemmas for the New Algorithm

λ is a conflict lemma if

1. every substlet $\sigma \in \lambda$ is in conflict with some clause c in the current version of Σ^+.
2. (Σ^+, Σ^-) implies λ.

As before, we call a lemma with properties $\mathbf{1}, \mathbf{2}$ a conflict lemma.

It can be shown that derivation of a conflict lemma is always possible.

Conclusions

I presented a collection of building blocks from which matching algorithms can be obtained. One instance is in the paper.

I need these matching algorithms for **geo**, but they may have usefulness in themselves.

I believe that one of the resulting algorithms will be good. I had no time try them out systematically, and the current CASC version has some obvious efficiency bugs (in lemma watching).

NP-completeness is caused by increased expressivity of geometric formulas. It may result in shorter proofs. It is not necessarily bad.

Computer science is an empirical science. It is closer to physics than to mathematics.