Subsumption for Three-Valued Geometric Resolution

ZJP Seminar, Wrocław, 20.05.2016.

Context

Theorem proving: We try to find logical proofs automatically. Frequently used calculi:

Untyped, first-order logic $\Rightarrow$ Resolution, Paramodulation, Superposition.

In 2005-2006, I developed a calculus called geometric resolution. It is good at finding finite models, and reasonably good at finding proofs.

Context (2)

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

Geometric resolution has been implemented in a theorem prover, which was (creatively) called **Geo**, and written in C^{++}.

Dependent on how you interpret the results, one could say that Geo was 4th in CASC 2006 in FOF and CNF categories, and third in the SAT category.

Not great, but not also not hopeless.

My real aim, from the beginning, was not to win CASC, but to develope a theorem prover that is good at dealing with partial functions.

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!";
}
```

Partial Classical Logic

It was not so easy as I thought: In 2010-2014, I developed an extension of classical logic, with supports partial functions.

I am totally not going to speak about this logic, but it is a nice logic, and I believe it is 'right'.

- Theorem Proving for Classical Logic with Partial Functions by Reduction to Kleene Logic. Journal of Logic and Computation, 2014.
- Classical Logic with Partial Functions, Journal of Automated Reasoning, 2011.

(Don't read the 2011 paper, because the logic has evolved)

Three-Valued Logic

The theorem proving method in the 2014 paper is a 3-valued adaptation of Geometric Resolution.

I want an implementation of this calculus, that is so effective, that somebody could actually consider using it.

The implementation is dominated by one miserable problem, where the prover spends most its time, which is matching Horn clauses into interpretations.

So we finally arrived at our main topic!

Matching

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

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

in interpretation

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

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

Unfortunately:

theorem: Matching is very NP-complete.

The proof is easy, by reduction from SAT.

Substitutions, Substlets

Substitutions are defined a usual. A substlet is a (small) substitution. We usually 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 $\Theta_2 \subseteq \Theta_1$.

(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

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

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

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

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

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

Example (3)

Matching

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

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 without variables, then (Σ^+, Σ^-) is trivially unsolvable.
2. If Σ^+ contains a clause without 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**(Θ) is a recursive algorithm. The parameter Θ is initially empty. In addition, it has access to (Σ^+, Σ^-).

- 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.

Optional Preprocessing: Filtering

Procedure **filter**$(c, \Sigma^+, \Sigma^-, U)$ checks local consistency of clauses in Σ^+ against c. If any clauses lose substlets, they are added to U.

- Generate all connected subsets $C \subseteq \Sigma^+$ of size k that contain c. For every generated subset C, do the following:
- Write C in the form $\{c_1, \ldots, c_k\}$. Initialize $(s_1, \ldots, s_k) = (\emptyset, \ldots, \emptyset)$.
- Generate all solutions Θ of (C, Σ^-). For every solution Θ found, do
  - For every c_i $(1 \leq i \leq k)$, let $s \in c_i$ be the substlet that is consistent with Θ.
  - Replace s_i by $s_i \cup s$.
- For every s_i that is different from c_i, replace c_i by s_i, and add c_i to U.

Filtering (2)

Filtering proceeds as follows:

- Start with $U = \Sigma^+$.
- As long as U is not empty, pick a $c \in U$, and remove c from U. After that, call **filter**$(c, \Sigma^+, \Sigma^-, U)$.

A concrete implementation uses indices instead of clauses.

Reasonable choices for k are $2, 3, 4$.

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

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

Learning from Conflicts (Borrowed from DPLL)

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.

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

Modify **solve**(Θ) in such a way that, whenever it fails, it constructs a lemma λ, s.t.

1. λ is in conflict with Θ.
2. (Σ^+, Σ^-) implies λ.

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

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^+.

Combination of Filtering and Backtracking

A weak version of the previous algorithm is present in the two-valued version of **Geo**, and it works reasonably well.

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 the algorithm on

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

More precisely:

Algorithm **refine**(Σ^+) is a recursive algorithm. Initially, (Σ^+, Σ^-) is filtered into (Σ'^+, Σ^-). After that, if Σ'^+ does not contain an empty clause, **refine**(Σ'^+) is called.

- If the $c \in \Sigma^+$ that have $\|c\| = 1$ imply a $\sigma \in \Sigma^-$, then fail.
- If all $c \in \Sigma^+$ have $\|c\| = 1$, then report solution $\bigcup \Sigma^+$.
- Select a $c \in \Sigma^+$ and partition it as $c_1 \cup \cdots \cup c_m$, for some $m \geq 2$. For each c_i do the following:
- – Set $\Sigma_i^+ = (\Sigma^+ \backslash c) \cup c_i$. Set $U = \{c_i\}$.
  – Do a complete filtering: As long as U is not empty, pick a c from U, and remove c from U. Call **filter**$(c, \Sigma_i^+, \Sigma^-, U)$. If this results in an empty clause, then return fail.
  – Otherwise, call **refine**(Σ_i^+).

Addition of Learning

Describing algorithms in mathematical notation is always tricky because variables are reassigned. One could solve this by indexing, but that is ugly.

Let c be a clause occurring in a version of Σ^+ during execution of the algorithm. There exists an initial clause c in the initial Σ^+ that c is obtained from. Write $I(c)$ for this clause.

refine(Σ^+) is modified in such a way that, whenever it fails, it constructs a lemma λ, s.t.

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

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

New version of **refine**:

Let Λ be the current set of lemmas, initially $\Lambda = \emptyset$.

- If Λ contains a lemma λ that can be partitioned as $\lambda^+ \cup \lambda^-$, where λ^- are the substlets that are in conflict with some $c \in \Sigma^+$, and all substlets in λ^- have the same set of unassigned variables, then
  - If $\lambda^+ = \emptyset$, then λ is a conflict lemma.
  - Otherwise, partition every $c \in \Sigma^+$ as $c^+ \cup c^-$, where c^- are the substlets that are in conflict with λ^+. Replace c by c^+. If c^+ is empty, then add $\mathrm{RES}(c, c^-; \lambda, \lambda^+)$ to Λ.
- If the $c \in \Sigma^+$ that have $\|c\| = 1$ imply a $\sigma \in \Sigma^-$, then let $c_1, \ldots, c_m \in \Sigma^+$ be a minimal set of clauses that implies σ. Add $\mathrm{RES}(I(c_1), \ldots, I(c_m);\ c_1, \ldots, c_m)$ to Λ.

- By induction, each of the recursive calls of **solve**(Σ_i^+) produces a conflict lemma of Σ_i^+).

  Let $\lambda_1, \ldots, \lambda_m$ be the resulting conflict lemma.

  If one of $\lambda_1, \ldots, \lambda_m$ already is a conflict lemma of Σ, then nothing needs to be done.

  Otherwise, construct

$$\mathrm{RES}(I(c), \lambda_1, \ldots, \lambda_m;\ c, \mu_1, \ldots, \mu_m),$$

  where μ_i are the substlets in λ_i that are in conflict with c.

- (I skip the construction for **filter**)

Filtering

Consider problem again:

$$\begin{cases} (X,Y) & / & (0,0) \mid (0,1) \mid (1,1) \mid (1,2) \\ (Y,Z) & / & (0,0) \mid (1,0) \mid (2,1) \\ (Z,T) & / & (1,0) \mid (1,1) \end{cases}$$

By staring long at (Y,Z) and (Z,T), one sees that only $(Y,Z) := (2,1)$ is possible.

After that, by looking at the first two substclauses, one sees that only $(X,Y) := (1,2)$ is possible.

Problems solved without backtracking!

Efficient Implementation

- 2-Literal watching can be used for finding conflict lemmas. One picks two literals with different set of variables, and watches all variables in the literals.
- Clauses $c \in \Sigma^+$ are replaced by subsets, and restored all the time. At the same time, one has to remember $I(c)$.

  Represent $I(c)$ as an array, and c as an interval in $I(c)$.

  Deletion of $\sigma \in c$. Swap σ to the end of the interval.

  Invariant: Substlets outside of the current interval are not moved.

Conclusions

One of these algorithms will be good. I have no real time to try them out systematically.

NP-completeness is caused by increased expressivity of geometric formulas. It may result in shorter proofs.

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