Outline

- Geometry Theorem Proving
- CVC Lite

Sources:

Harrison, John. *Introduction to Logic and Automated Theorem Proving*. Unpublished manuscript. Used by permission.


Review

- Real Algebra
- Combining Decision Procedures

Geometry Theorem Proving

Consider the following theorem in geometry.

**Theorem**

If \( D \) is the midpoint of \( AB \) and \( E \) is the midpoint of \( AC \), then \( BC \parallel DE \).

At first, a theorem like this may seem significantly different from the kinds of first order theorems we have dealt with so far.

However, with a little bit of effort, we can leverage our previous results to prove such theorems.
Geometry Theorem Proving

The first step is figuring out how to translate geometric statements into formulas of first order logic.

The key idea is to use Cartesian coordinates, representing every point as a pair of real numbers.

With this representation, many standard geometrical assertions can be encoded as simple polynomial equations.

For example, three points $A$, $B$, and $C$ are collinear iff

$$(A_x - B_x)(B_y - C_y) = (A_y - B_y)(B_x - C_x).$$

Similarly, $A$ is the midpoint of the line segment joining $B$ and $C$ iff

$$2A_x = B_x + C_x \land 2A_y = B_y + C_y.$$
Many geometry theorems are **constructive**: starting with an initial set of arbitrary points, \( P_1, \ldots, P_k \), a set of new points \( P_{k+1}, \ldots, P_n \) is constructed. The conclusion is then some assertion about the total set of points.

The key insight is that because points are constructed in a certain order, this order can be exploited to make the theorem-proving process more efficient.

Wu’s method is based on triangulation. A set of polynomial equations is said to be **triangular** when it has the following form:

\[
p_m(x_1, \ldots, x_i, x_{k+1}, \ldots, x_{k+m}) = 0 \\
\vdots \\
p_1(x_1, \ldots, x_k, x_{k+1}) = 0 \\
p_0(x_1, \ldots, x_k) = 0
\]

In other words, each polynomial, \( p_i \) contains a variable \( x_{k+i} \) which does not appear in any polynomial \( p_j \), where \( j < i \).

Any set of polynomial equations can be used to derive a triangular set which are true whenever the initial set is true.

The elimination uses our favorite trick of pseudo-division.

Suppose we start with \( s_m(x_1, \ldots, x_{k+m}) \). The goal is to obtain a simple set of conditions which imply \( s_m = 0 \).

We can start by pseudo-dividing \( s_m \) by \( p_m \) to obtain:

\[
a_m(x_1, \ldots, x_{k+1}) s_m = p_m q_m + s_{m-1}(x_1, \ldots, x_{k+m})
\]

where \( a_m \) is the leading coefficient of \( p_m \), considered as a polynomial in \( x_{k+m} \).

Since \( p_m = 0 \) is in our triangular set, we can deduce that:

\[
s_{m-1}(x_1, \ldots, x_{k+m}) = 0 \iff a_m = 0 \lor s_m = 0.
\]

It follows that \( (s_{m-1} = 0 \land a_m \neq 0) \rightarrow s_m = 0 \).

If we are lucky (or if \( p_m \) is linear in \( x_{k+m} \)), \( s_{m-1} \) will not contain \( x_{k+m} \).

Otherwise, we collect the coefficients \( c_i \) of \( x_{k+m} \) in \( s_{m-1} \) to get:

\[
(\land c_i = 0 \land a_m \neq 0) \rightarrow s_m = 0.
\]

In either case, the equations in the left hand side of the implication no longer contain \( x_{k+m} \), and we can thus apply the elimination procedure recursively to each of them.

Suppose we have a set \( q_0, \ldots, q_m \) of polynomials that we wish to triangulate.

We simply pick some order on the variables. Let \( x_{k+m} \) be the first variable in the order. If there is only one polynomial containing \( x_{k+m} \), we simply let this be \( p_m \).

Otherwise, we pick the polynomial of least degree in \( x_{k+m} \) and pseudo-divide all other polynomials containing \( x_{k+m} \) by it. We continue this until only one polynomial contains \( x_{k+m} \), which then becomes \( p_m \).

By repeating this process for each variable, we obtain a triangular set of polynomials \( p_0, \ldots, p_m \).

We next describe how such a set can be used to successively “eliminate” variables in a polynomial \( s(x_1, \ldots, x_{k+m}) \).

Thus, we have a theorem that gives a set of conditions under which \( s_m = 0 \).

We can use this procedure to prove geometry theorems. The idea is to build a triangular set of polynomials \( p_i \) out of the formulas describing the construction of auxiliary points.

We then let \( s_m \) be the assertion in the conclusion of the theorem and use the above procedure to get a sufficient set of conditions under which the theorem is true.

The conditions obtained often correspond precisely to the non-degeneracy conditions we mentioned earlier.

We will look at some examples in geom.ml.
CVC Lite

CVC Lite is the successor to CVC (Cooperating Validity Checker) which was the successor to SVC (Stanford Validity Checker).

The original motivation for SVC was verification of microprocessors using the Burch-Dill method.

The Burch-Dill Method

- Prove that an implementation matches its specification
- Proof is by induction over all computations
- Induction step requires proving the validity of a large formula generated by symbolically simulating the implementation and the specification.

Validity Checker Philosophy

- Be able to handle large examples
- Language should include constructs useful for modeling real systems
- Validity checking should be fast and complete
- Increase confidence in result by producing a proof that can be checked separately

Deciding First-Order Theories

CVC Lite is a decision procedure for the quantifier-free validity problem for a theory $T$ that tries to capture constructs which are useful in a wide variety of verification applications.

The theory $T$ is actually a union of a number of simpler theories whose signatures are disjoint.

CVC Lite works by combining decision procedures for the individual theories using an implementation of the Nelson-Oppen method for combining decision procedures.

We will consider a few examples of theories which are included in CVC Lite and then briefly review the Nelson-Oppen method for combining theoreis.

History and Background

SVC and CVC were both successful systems, with many users and a wide variety of applications.

But there were gaps in both theory and software architecture that were easiest to address by starting over.

CVC Lite is a joint project of Stanford University and New York University. Its primary authors are Sergey Berezin (Stanford) and Clark Barrett (NYU).

CVC Lite already contains most of the functionality of the previous systems as well as some functionality not found in either previous system.

CVC Lite has proven to be easier to understand and extend.

The Theory $T_{\epsilon}$ of Equality

The theory $T_{\epsilon}$ of equality is the theory $\emptyset$.

Though the theory is empty, the signature is not.

Because the theory gives no information about the function or predicate symbols in the signature, this theory is sometimes called the theory of “uninterpreted functions”.

A well-known result in first-order logic is that the general validity problem for $T_{\epsilon}$ is undecidable.

However, the quantifier-free validity problem for $T_{\epsilon}$ is decidable via congruence closure.
The Theory $T_R$ of Reals

Let $\Sigma_R$ be the signature $(0, 1, +, -, \leq)$.

Let $\mathcal{A}_R$ be the standard model of the reals with domain $\mathbb{R}$.

Then $T_R$ is defined to be $Th_{\mathcal{A}_R}$.

The validity problem for $T_R$ is decidable.

The quantifier-free satisfiability problem for conjunctions of literals (atomic formulas or their negations) in $T_R$ is solvable in polynomial time, though exponential methods (like simplex or Fourier-Motzkin) tend to perform best in practice.

The Theory $T_A$ of Arrays

Let $\Sigma_A$ be the signature $(\text{read}, \text{write})$.

Let $T_A$ be the following axioms:

\[
\forall a \forall i \forall v \,(\text{read}(\text{write}(a, i, v), i) = v)
\]
\[
\forall a \forall i \forall j \forall v \,(i \neq j \rightarrow \text{read}(\text{write}(a, i, v), j) = \text{read}(a, j))
\]
\[
\forall a \forall i \forall b \,\left(\forall i \,(\text{read}(a, i) = \text{read}(b, i)) \rightarrow a = b\right)
\]

The validity problem for $T_A$ is undecidable, but the quantifier-free validity problem for $T_A$ is decidable (its complexity is NP).

Modeling Reactive Systems Using CVC Lite

Consider the following lines of code:

\[
\begin{align*}
I_0 & : a[i] := a[i] + 1; \\
I_1 & : a[i + 1] := a[i - 1] - 1; \\
I_2 & : 
\end{align*}
\]

This can be modeled in CVC Lite as follows:

\[
\begin{align*}
i_0, i_1, i_2 & : \text{REAL}; \\
a_0, a_1, a_2 & : \text{ARRAY REAL OF REAL}; \\
\text{ASSERT} & (a_1 = a_0 \text{ WITH } [i_0] := a_0[i_0]+1) \text{ AND } (i_1 = i_0); \\
\text{ASSERT} & (a_2 = a_1 \text{ WITH } [i_1+1] := a_1[i_1-1]-1) \text{ AND } (i_2 = i_1);
\end{align*}
\]

Modeling Reactive Systems Using CVC Lite

To check whether the result is equivalent when the two statements are swapped, we can use the following CVC Lite \textit{QUERY}.

\[
\begin{align*}
i_0, i_1, i_2, i_3, i_4 & : \text{REAL}; \\
a_0, a_1, a_2, a_3, a_4 & : \text{ARRAY REAL OF REAL}; \\
\text{ASSERT} & (a_1 = a_0 \text{ WITH } [i_0] := a_0[i_0]+1) \text{ AND } (i_1 = i_0); \\
\text{ASSERT} & (a_2 = a_1 \text{ WITH } [i_1+1] := a_1[i_1-1]-1) \text{ AND } (i_2 = i_1); \\
\text{ASSERT} & (a_3 = a_0 \text{ WITH } [i_0+1] := a_0[i_0-1]-1) \text{ AND } (i_3 = i_0); \\
\text{ASSERT} & (a_4 = a_3 \text{ WITH } [i_3] := a_3[i_3]+1) \text{ AND } (i_4 = i_3); \\
\text{QUERY} & (i_2 = i_4 \text{ AND } a_2 = a_4);
\end{align*}
\]
Modeling Reactive Systems Using CVC Lite

A more efficient encoding ignores variables that do not change and uses the \textit{LET} construct to introduce temporary expressions.

\begin{verbatim}
i : REAL;
a : ARRAY REAL OF REAL;

QUERY
(LET a1 : ARRAY REAL OF REAL = a WITH [i] := a[i]+1 IN

  a1 WITH [i] := a1[i] =

  (LET a1 : ARRAY REAL OF REAL = a WITH [i+1] := a[i-1]-1 IN

  a1 WITH [i] := a1[i]+1);
\end{verbatim}

Software Architecture

The core algorithms of CVC Lite decide the satisfiability of conjunctions of literals.

What if the formula is not a conjunction of literals?

One approach would be to use propositional transformations (such as distributivity and DeMorgan’s laws) to transform the formula into disjunctive normal form (DNF) and then test each disjunct separately.

However, this can result in an exponential blow-up in the size of the formula and is thus too costly in practice.

Checking the Satisfiability of Arbitrary Formulas

Another approach is to look for a consistent assignment of truth values to the literals which makes the formula true.

Note that this is really the combination of two problems:

- A Boolean satisfiability problem to find an assignment of truth values to the Boolean structure of the formula.
- A non-Boolean satisfiability problem to check whether the assignment of truth values to literals is consistent.

We can use SAT-based techniques to solve the first problem, and Nelson-Oppen based techniques to solve the second.

In the remainder of this lecture, we assume that \textit{Sat} is an algorithm for determining the satisfiability of a conjunction of first-order literals.
Checking the Satisfiability of Arbitrary Formulas

SVC (the predecessor to CVC) uses a simple DPLL recursive search to solve the Boolean satisfiability portion of the problem.

CheckSat($\Gamma, \phi$)

IF $\neg Sat_{FO}(\Gamma)$ THEN RETURN $\emptyset$;
$\phi'$ := Simplify($\Gamma$, $\phi$);
IF $\phi'$ = FALSE THEN RETURN $\emptyset$;
IF $\phi'$ = TRUE THEN RETURN $\Gamma$;
$\alpha$ := FindLiteral($\phi'$);
$\Gamma'$ := CheckSat($\Gamma \cup \{\alpha\}, \phi'$);
IF $\Gamma'$ $\neq \emptyset$ THEN RETURN $\Gamma'$;
RETURN CheckSat($\Gamma \cup \{-\alpha\}, \phi'$);

Computing a Propositional Abstraction

Given a quantifier-free first-order formula $\phi$, the first step is to compute a propositional abstraction $Abs(\phi)$ of $\phi$.

We do this by replacing each atomic formula $\alpha$ in $\phi$ by a propositional variable $p_{\alpha}$.

The result is a propositional formula $Abs(\phi)$ which has the following property:

If $Abs(\phi)$ is unsatisfiable, then $\phi$ is unsatisfiable.

However, the converse is not true. It may be the case that $Abs(\phi)$ is satisfiable but $\phi$ is not.

A Naive Algorithm

Suppose we wish to check the satisfiability of a quantifier-free first-order formula $\phi$.

- We first form the propositional abstraction $Abs(\phi)$.
- We check $Abs(\phi)$ for satisfiability using a Boolean SAT solver.
- If $Abs(\phi)$ is unsatisfiable, $\phi$ is unsatisfiable.
- Otherwise, let $\psi$ be a variable assignment satisfying $Abs(\phi)$.
- Let $Abs^{-1}(\psi)$ be the conjunction of first-order literals corresponding to $\psi$.
- If $Sat_{FO}(Abs^{-1}(\psi))$, then $\phi$ is satisfiable.
- Otherwise, we refine $Abs(\phi)$ by adding $\neg \psi$ and repeat.

Since there are only a finite number of possible variable assignments to $Abs(\phi)$, the algorithm will eventually terminate.
Problems with the Naive Approach

Although simple and elegant, the approach just described is not practical.
The following is a list of the main issues that have to be addressed in order to make the algorithm efficient in practice.

- Redundant clauses
- Lazy notification
- Decision heuristics
- Sat heuristics and completeness

Redundant Clauses

The main difficulty with the naive approach is that the clauses added in the refinement step can be highly redundant.

Suppose that the Boolean abstraction $Abs(\phi)$ contains $n + 2$ propositional variables.

When a Boolean assignment is returned by the SAT solver, all $n + 2$ variables will have an assignment.

But what if only 2 of these assignments are sufficient to result in an inconsistency in the atomic formulas associated with the variables?

For each assignment of values to the other $n$ propositional variables which leads to a satisfying solution, the refinement loop will have to add another clause.

In the worst case, $2^n$ clauses will be added when a single clause containing the two offending variables would have sufficed.

Lazy Notification

The naive algorithm is lazy in the sense that the SAT solver is used as a black box and $Sat_{FO}$ is not invoked until a complete solution is obtained.

In contrast, an eager approach is to notify $Sat_{FO}$ incrementally of every decision made by the SAT solver.

Experimental results show that the eager approach is significantly better.

Eager notification requires that $Sat_{FO}$ be online: able quickly to determine the consistency of incrementally more or fewer literals.

Eager notification also requires that the SAT solver be instrumented to inform $Sat_{FO}$ every time it assigns a variable.
Naive, Lazy, and Eager Implementations

<table>
<thead>
<tr>
<th>Example</th>
<th>Naive</th>
<th>Lazy</th>
<th>Eager</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Iterations</td>
<td>Time (s)</td>
<td>Iterations</td>
</tr>
<tr>
<td>read0</td>
<td>77</td>
<td>0.14</td>
<td>17</td>
</tr>
<tr>
<td>pp-pc-s2i</td>
<td>?</td>
<td>&gt; 10000</td>
<td>82</td>
</tr>
<tr>
<td>pp-invariant</td>
<td>?</td>
<td>&gt; 10000</td>
<td>239</td>
</tr>
<tr>
<td>v-dlx-pc</td>
<td>?</td>
<td>&gt; 10000</td>
<td>6158</td>
</tr>
</tbody>
</table>

Decision Heuristics

SAT solvers like Chaff have sophisticated heuristics for determining which variable to split on.

However, for some first-order examples, the structure of the original formula is an important consideration when determining which literal to split on.

For example, CVC includes the `ite` (if-then-else) construct.

Suppose an `ite` expression of the form `ite(α, t₁, t₂)` appears in the formula being checked.

If α is set to `T`, then all of the literals in t₂ can be ignored since they no longer affect the formula.

Unfortunately, the SAT solver doesn’t know this and as a result, it can waste a lot of time choosing irrelevant variables.

We found that for such examples, it was better to use a depth-first traversal of the original formula to choose splitters than the built-in SAT heuristic.

Again, this requires tighter integration and communication between the two solvers.

Variable Selection Results

<table>
<thead>
<tr>
<th>Example</th>
<th>SAT</th>
<th>DFS</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Decisions</td>
<td>Time (s)</td>
</tr>
<tr>
<td>bool-dlx1-c</td>
<td>1309</td>
<td>0.69</td>
</tr>
<tr>
<td>bool-dlx2-aa</td>
<td>4974</td>
<td>2.36</td>
</tr>
<tr>
<td>bool-dlx2-cc-bug01</td>
<td>10903</td>
<td>11.4</td>
</tr>
<tr>
<td>v-dlx-pc</td>
<td>4387</td>
<td>3.22</td>
</tr>
<tr>
<td>v-dlx-dmem</td>
<td>5221</td>
<td>4.12</td>
</tr>
<tr>
<td>v-dlx-regfile</td>
<td>6802</td>
<td>5.85</td>
</tr>
<tr>
<td>dlx-pc</td>
<td>39833</td>
<td>19.0</td>
</tr>
<tr>
<td>dlx-dmem</td>
<td>34320</td>
<td>18.8</td>
</tr>
<tr>
<td>dlx-regfile</td>
<td>47822</td>
<td>35.5</td>
</tr>
<tr>
<td>pp-bloaddata-a</td>
<td>8695</td>
<td>5.47</td>
</tr>
<tr>
<td>pp-bloaddata</td>
<td>9016</td>
<td>5.56</td>
</tr>
<tr>
<td>pp-dmem2</td>
<td>3167</td>
<td>2.24</td>
</tr>
</tbody>
</table>

SAT Heuristics and Completeness

A somewhat surprising observation is that some heuristics used by SAT solvers must be disabled or the method will be incomplete.

An example of this is the pure literal rule.

This rule looks for propositional variables which are either always (or never) negated in the CNF formula being checked.

These variables can instantly be replaced by `T` (or `F`).

However, if such a variable is an abstraction of a first-order atomic formula, this is no longer the case.

This is because propositional literals are independent of each other, but first-order literals may not be.

Such heuristics must be carefully disabled in the SAT solver.
**CVC with SAT Compared to CVC without SAT**

<table>
<thead>
<tr>
<th>Example</th>
<th>cvc -sat</th>
<th>cvc +sat</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Decisions</td>
<td>Time (s)</td>
</tr>
<tr>
<td>bool-dlx1-c</td>
<td>?</td>
<td>&gt; 10000</td>
</tr>
<tr>
<td>bool-dlx2-aa</td>
<td>?</td>
<td>&gt; 10000</td>
</tr>
<tr>
<td>bool-dlx2-cc-bug01</td>
<td>?</td>
<td>&gt; 10000</td>
</tr>
<tr>
<td>v-dlx-pc</td>
<td>8642456</td>
<td>5082</td>
</tr>
<tr>
<td>v-dlx-dmem</td>
<td>2888268</td>
<td>2820</td>
</tr>
<tr>
<td>v-dlx-regfile</td>
<td>29435</td>
<td>37.6</td>
</tr>
<tr>
<td>dlx-pc</td>
<td>515</td>
<td>0.68</td>
</tr>
<tr>
<td>dlx-dmem</td>
<td>6031</td>
<td>4.50</td>
</tr>
<tr>
<td>dlx-regfile</td>
<td>6386</td>
<td>5.27</td>
</tr>
<tr>
<td>pp-bloaddata-a</td>
<td>93714</td>
<td>79.1</td>
</tr>
<tr>
<td>pp-bloaddata</td>
<td>345569</td>
<td>338</td>
</tr>
<tr>
<td>pp-dmem2</td>
<td>367877</td>
<td>338</td>
</tr>
</tbody>
</table>

**Related Work**

SRI’s *ICS* prover has a prototype implementation combining it with SAT. However, ICS is not an explicating prover, so they do not have a good mechanism for dealing with redundant clauses.

HP (formerly Compaq, formerly DEC) Systems Research Center (SRC) (a group which includes Greg Nelson of Nelson-Oppen fame) is developing a system called *Verifun* which uses a similar approach to the one outlined here.

However, Verifun uses custom annotations to explicate inconsistencies, whereas CVC uses its proof-production engine to accomplish the same thing.

**Integrating SAT Heuristics in CVC Lite**

Though the SAT technique was fairly successful for CVC, there were still problems with it:

- Because SAT solvers do not produce proofs, CVC cannot produce an overall proof in Sat mode.
- Non-convex theories can sometimes cause blow-up in Sat mode.
- Structural heuristics for choosing splitters are harder to apply because the structural information is lost in the translation to SAT.

*CVC Lite* addresses these issues.

**Proofs and Explication**

Having an automated theorem prover which produces proofs kills two birds with one stone.

First, proof-production improves robustness and increases confidence in the tool.

Second, proof-production provides a mechanism for explication which is critical to the success of the SAT-based approach just described.

To see this second point, suppose $Sat_{FO}$ is presented with a conjunction of literals and determines that they are inconsistent.

If this determination is accompanied by a proof, then an analysis of the proof yields the required information about which literals contributed to the inconsistency.

In CVCL, proof-production has two settings: a slow and thorough mode, in which full proofs are produced, and an assumption-tracking only mode, in which the proof-production machinery is used to track just enough information to make explication possible.
A Framework for Producing Proofs

Proofs are represented in CVCL by sequents.

A sequent is a pair $\Gamma \vdash \phi$, where $\Gamma$ is a set of assumptions and $\phi$ is a formula.

A sequent is valid if $1 \cup \Gamma \vdash \phi$, where $1$ is the deductive closure of the union of all theories participating in the cooperating framework of CVCL.

A proof rule, or inference rule is denoted as follows:

$$\frac{P_1 \cdots P_n}{C}$$

where the $P_i$’s are premises and $C$ is the conclusion of the rule (all are sequents).

A rule is sound if the validity of all premises implies the validity of the conclusion.

The set of premises may be empty, in which case the rule is called an axiom.

A proof or derivation of a sequent $C$ is a sequence of proof rule applications that forms a finite proof tree with $C$ as the root and axioms on the leaves.

Representing Theorems in CVCL

The main data structure in CVCL for dealing with proofs is the Theorem data structure.

A Theorem consists of a sequent and may or may not contain an accompanying proof.

In the slow and thorough mode, Theorem’s contain a full proof which gives a derivation of the sequent in the Theorem using CVCL’s inference rules.

In assumptions only mode, a Theorem contains only the derived sequent.

A Selection of Proof Rules

CVCL takes a very pragmatic approach to proof rules. The goal is to have a small trusted set of proof rules that can be independently checked.

The most basic rules are the assumption axiom, proof by contradiction (or negation elimination, implication and negation introduction, modus ponens for the $\leftrightarrow$ operator, and the cut rule:

$$\frac{\phi \vdash \phi}{\Gamma \vdash \phi} \text{ assume} \quad \frac{\Gamma, -\phi \vdash F}{\Gamma \vdash \phi} \text{ E} \quad \frac{\Gamma, \alpha \vdash \phi}{\Gamma, \alpha \vdash \phi} \text{ I} \quad \frac{\Gamma, \alpha \vdash F}{\Gamma \vdash \neg \alpha} \text{ I}$$

$$\frac{\Gamma_1 \vdash \phi \quad \Gamma_2 \vdash \phi \leftrightarrow \psi}{\Gamma_1 \cup \Gamma_2 \vdash \psi} \text{ MP} \quad \frac{\Gamma_1 \vdash \alpha \quad \Gamma_2, \alpha \vdash \phi}{\Gamma_1 \cup \Gamma_2 \vdash \phi} \text{ cut}$$

Implementing Proof Production

Instrumenting CVCL to produce proofs is easier than might be expected.

The implementation framework for CVCL (which is based on Nelson-Oppen) requires maintaining and communicating about facts which are true in the current context.

Instead of passing formulas around, we modify the framework to pass Theorem’s around instead.

Any time some reasoning step is performed, a Theorem is generated to encapsulate what was done.
Implementing Proof Production

Example

A union/find data structure is used to maintain equivalence classes of terms. Normally a call to find would return the representative term for the equivalence class.

We modify this so that a call to find returns the Theorem that a term is equivalent to its equivalence class representative.

When equivalence classes are merged, the Theorem's are combined using an inference rule for the transitivity of equality.

Using Theorems to Generate Conflict Clauses for SAT

In sat mode, CVCL uses Theorem's to generate conflict clauses as follows.

- Whenever SAT makes a decision, the corresponding literal is given as a new assumption to CVCL.
- If CVCL detects an inconsistency, it produces a Theorem whose sequent is $\Gamma \vdash F$ where $\Gamma$ is guaranteed to be a subset of the assumptions handed to CVCL.
- Because $\Gamma \vdash F$ is valid in CVCL's theory 1, we know that $1 \models \neg (\bigwedge \Gamma)$.
- $\neg (\bigwedge \Gamma)$ corresponds to a disjunction of literals and can be abstracted to obtain a conflict clause for SAT.

Integrating SAT Heuristics in CVC Lite

CVC Lite takes the dual-use of proof-production to the next level by using proofs to implement all of the standard optimizations used by SAT solvers.

CVC Lite does not translate its input into CNF, but instead retains the original formula.

However, learned conflict clauses are maintained as in SAT and are processed using fast Boolean Constraint Propagation using watched literals, just as in SAT.

Thus, there are two different sets of formulas, and different decision heuristics can be used for each of them.

Some additional overhead is incurred to instrument the fast SAT algorithms with proofs.

Current and Future Research

- Theoretical extension to sorted logic
- Dealing with sub.sorts and partial functions
- Support for quantifiers
- Integer arithmetic
- Better search heuristics
- Embedding in an interactive theorem prover
- Using CVC Lite for translation validation
- Support for non-linear arithmetic
- New theories
- New applications