Computational Reflection:
Automatically Proving Difficult Things

Anthony Narkawicz

NASA Langley Formal Methods Group
anthony.narkawicz@nasa.gov
October 2012
Suppose we are proving the correctness of some system,

Part way through the proof, we must show:

$$\vdash \{1\} \text{ EXISTS (x:real): } 3x^2 -5x + 2 = 0$$
A little while later, we have to prove:

|⁻
{1} EXISTS (x:real): 2 x² - 3x + 1 = 0

And later,

|⁻
{1} EXISTS (x:real): 12 x² - 10x + 2 = 0

In each case, we prove this by instantiating formula 1 with a real number x that makes the equality true.

But we don’t need to know the exact solutions to these equations to know that they are true.
Computational Reflection

By the quadratic formula, the equation

\[ ax^2 + bx + c = 0 \]

has a solution if and only if \( b^2 - 4ac \geq 0 \).

In PVS, we can define a function on \( a \), \( b \), and \( c \) that returns a boolean:

\[ D(a,b,c:\text{real}) : \text{bool} = b^2 - 4a*c \geq 0 \]

We can then prove the following lemma in PVS

quadratic_solvable : LEMMA

\[
\text{FORALL (a,b,c:\text{real})}:
\begin{align*}
(\exists x:\text{real}) : ax^2 - bx + c &= 0 \\
\iff D(a,b,c)
\end{align*}
\]
Computational Reflection

Now we can solve all of those lemmas by just evaluating $D$.

The next time we have to prove something like

$$\vdash \{1\} \text{ EXISTS (x:real): } 2 \, x^2 - 3 \, x + 1 = 0$$

we can just type

```
(lemma 'quadratic_solvable')
(inst?)
(assert)
(hide (-1 -2))
```

which turns the sequent into

$$\vdash \{1\} \, D(2,-3,1)$$

This proves with

```
(grind)
```
What if we tried to prove something that is false???

Part way through the proof, we must show:

\[ |- \{1\} \text{EXISTS } (x:\text{real}): 10 \ x^2 -2x+1 = 0 \]

This **FAILS**:

(lemma `'quadratic_solvable'`)
(inst?)
(assert)
(hide (-1 -2))
(grind)
Computational Reflection

Proving that a quadratic has a root:

Can't be evaluated

\[ \{1\} \text{ EXISTS (x:real): } 2x^2 -3x+1 = 0 \]

Can be evaluated (e.g. with grind)

\[ \{1\} \text{ } D(2,-3,1) \]

A few simple commands

Computational reflection is similar:

Can't be evaluated

\[ \{1\} \text{ P(p)} \]

Can be evaluated (with (eval-formula))

\[ \{1\} \text{ Q(p)} \]

A few simple commands
Computational Reflection

Computational reflection:

- We have some type $T$ (e.g. quadratics)
- We often want to prove a property of $P(p)$ for some $p \in T$
- The property $P(p)$ can not be evaluated
- $Q(p)$ is equivalent to $P(p)$
- $Q(p)$ can be evaluated!
Computational Reflection

Computational reflection:

- $Q(p)$ is equivalent to $P(p)$ and can be evaluated

Sometimes (grind) can be inefficient.

Let’s prove

$$\vdash \{1\} \exists (x:\text{real}): 2^{400} * x^2 + 2^{600}x + 2^{100} = 0$$

The same proof works as before. The sequent is reduced to proving

$$\vdash \{1\} D(2^{400}, 2^{600}, 2^{100})$$

This proves with (grind)
Computational Reflection

\[
\begin{align*}
\vdash \\
\{1\} & \quad D(2^{400}, 2^{600}, 2^{100})
\end{align*}
\]

This proves with (grind)... but it takes more than a minute.

- What if we have to prove many results like this?
- What if the function D were significantly more complicated?

(grind) is not very efficient for evaluating complicated expressions

PVS is not really a programming language. We’d like to evaluate this expression as fast as we could in a programming language.
We can evaluate

\[ \vdash \{1\} D(2^{400}, 2^{600}, 2^{100}) \]

using

(eval-formula)

THIS is computational reflection
The property $Q(p)$ is equivalent to $P(p)$ and can be evaluated.
Why is it Called Reflection?

- PVS is built on top of LISP
- The problem is reflected down to LISP
- ... and computed there
Ground Terms

To compute $Q(p)$ in LISP using (eval-formula), all of the atoms involved must be ground terms.

That is, it has to be something that the programming language can compute.

For instance, if $a \in \mathbb{R}$, it can’t compute

\[
\text{IF } a^2 \geq 0 \text{ THEN } 1 \text{ ELSE } 0 \text{ ENDIF}
\]

which is equal to 1, because $a^2$ is not ground.

However, it can compute

\[
\text{IF } 2^2 \geq 0 \text{ THEN } 1 \text{ ELSE } 0 \text{ ENDIF}
\]
Example: Conflict Detection

Conflicts

- Minimum Horizontal Distance $D$
- Minimum Vertical Distance $H$

*The Protected Zone: A Cylinder*
The Problem: Detect Conflicts Within a Lookahead Time $T$

Conflict: \textit{Exists a time }$t \in [0, T]$ \textit{such that the red plane is inside the cylinder at time }$t$.\"
### Example: Conflict Detection

<table>
<thead>
<tr>
<th>Aircraft</th>
<th>Position</th>
<th>Velocity</th>
</tr>
</thead>
<tbody>
<tr>
<td>ownship</td>
<td>(s_o)</td>
<td>(v_o)</td>
</tr>
<tr>
<td>intruder</td>
<td>(s_i)</td>
<td>(v_i)</td>
</tr>
<tr>
<td>relative</td>
<td>(s = s_o - s_i)</td>
<td>(v = v_o - v_i)</td>
</tr>
</tbody>
</table>

\[
\text{conflict?}(D, H, s, v) \equiv \exists t \geq 0 : \|s + tv\| < D \quad \text{and} \quad |s_z + tv_z| < H
\]

This is not computable
Example: Conflict Detection

Project: Take 10K examples of near-conflicts and prove that none of them are actual conflicts.

Problem: Analyzing them individually would be very slow since conflict?((D, H, s, v) can’t be evaluated.

Solution: Replace conflict?((D, H, s, v) with something equivalent that can be evaluated (computational reflection)
Example: Conflict Detection

cd3d is an algorithm that computes whether $\text{conflict?}(D, H, s, v)$ holds.

\[
cd3d(s,v) : \text{bool} = \\
\begin{align*}
\text{IF } v'z &= \emptyset \text{ AND } \text{abs}(s'z) < H \\
\text{THEN }\quad \text{sqv}(\text{vect2}(s) + B * \text{vect2}(v)) < \text{sqv}(D) \text{ OR} \\
\qquad \text{IF } \text{sqv}(\text{vect2}(s)) = \text{sqv}(D) \text{ AND } B = \emptyset \text{ THEN } \text{vect2}(s) \neq \text{vect2}(v) \\
\qquad \text{ELSE } \begin{aligned}
\text{sqv}(\min(\max(B * \text{sqv}(\text{vect2}(v)), -(\text{vect2}(s) \neq \text{vect2}(v))), \\
T \neq \text{sqv}(\text{vect2}(v)))) + \\
\text{sqv}(\text{vect2}(v)) * \text{sqv}(\text{vect2}(s)) + 2 * (\min(\max(B * \text{sqv}(\text{vect2}(v)), -(\text{vect2}(s) \neq \text{vect2}(v))), \\
T \neq \text{sqv}(\text{vect2}(v)))) * (\text{vect2}(s) \neq \text{vect2}(v)) - \text{sqv}(D) * \text{sqv}(\text{vect2}(v)) \\
\end{aligned} \\
\text{ENDIF} < 0 \\
\text{ELSIF } v'z \neq \emptyset \text{ AND} \\
\qquad \max(-H - \text{sign}(v'z) * s'z, s', T) * \text{abs}(v'z) < \min(-H - \text{sign}(v'z) * s'z, s', T) * \text{abs}(v'z)) \\
\text{THEN } \begin{aligned}
\text{sqv}(\text{vect2}(abs(v'z) * s)) + \\
\text{sqv}(\text{vect2}(v)) * \text{sqv}(\text{vect2}(v)) + 2 * (\min(\max(-H - \text{sign}(v'z) * s'z, s'), T) * \text{abs}(v'z)) + \\
\text{sqv}(\text{vect2}(v)) * \text{sqv}(\text{vect2}(v)) + 2 * (\min(\max(-H - \text{sign}(v'z) * s'z, T) * \text{abs}(v'z))) \\
\end{aligned} < \text{sqv}(D) * \text{abs}(v'z)) \\
\text{OR} \\
\text{IF } \text{sqv}(\text{vect2}(abs(v'z) * s)) = \text{sqv}(D) * \text{abs}(v'z)) \text{ AND} \\
\qquad \max(-H - \text{sign}(v'z) * s'z, s', T) * \text{abs}(v'z) = 0 \\
\text{THEN } \begin{aligned}
\text{sqv}(\text{vect2}(abs(v'z) * s)) * \text{vect2}(v) \\
\end{aligned} \\
\text{ELSE } \begin{aligned}
\text{sqv}(\min(\max(-H - \text{sign}(v'z) * s'z, s'), T) * \text{abs}(v'z))) + \\
\text{sqv}(\text{vect2}(v)) * \text{sqv}(\text{vect2}(abs(v'z) * s)) + 2 * (\min(\max(-H - \text{sign}(v'z) * s'z, s'), T) * \text{abs}(v'z))) + \\
\text{sqv}(\text{vect2}(v)) * \text{sqv}(\text{vect2}(abs(v'z) * s)) * \text{vect2}(v)) \\
\end{aligned} \\
\text{ENDIF} < 0 \\
\text{ELSE FALSE} \\
\text{ENDIF}
\end{align*}
\]

\[cd3d\_correct : \text{LEMMA} \]
\[
\text{FORALL } (s,v: Vect3, D,H:posreal):
\text{conflict?}(D,H,s,v) \\
\text{IFF} \\
\text{cd3d}(D,H,s,v)
\]
Example: Conflict Detection

Given 10K lemmas of the form

\text{not\_conflict\_8741: LEMMA}
\begin{itemize}
    \item D = 5 AND
    \item H = 1000 AND
    \item s = (21,-5,-100) AND
    \item v = (-551,-1,300)
\end{itemize}
IMPLIES
\text{NOT conflict?}(D,H,s,v)

... the proofs are all the same and do not involve the actual numbers.

\text{conflict?} is replaced by \text{cd3d}, which is then evaluated using (eval-formula).
Example: Conflict Detection

```lisp
{-1} conflict?(D,H,s,v)
{-2} D = 5
{-3} H = 1000
{-4} s = (21,-5,-100)
{-5} v = (-551,-1,300)
   |- (replaces -2)
   (replaces -2)
   (replaces -2)
   (replaces -2)
   (lemma "cd3d_correct")
   (inst?)
   (assert)
   (hide -1)

{-1} cd3d(5,1000,(21,-5,-100),(-551,-1,300))
   |- (eval-formula)
```

Making the Proofs Even Easier

All of the proofs are the same.

We can create a single command that will execute the entire proof.

Let’s call it (noconflict).

This is called a strategy.

After defining it, every lemma of the form

\[
\text{not\_conflict\_8741: LEMMA}
\begin{align*}
D &= 5 \ \text{AND} \\
H &= 1000 \ \text{AND} \\
s &= (21,-5,-100) \ \text{AND} \\
v &= (-551,-1,300) \\
\text{IMPLIES} \\
\text{NOT conflict?}(D,H,s,v)
\end{align*}
\]

can be proved by just typing

(noconflict)
Strategies and Computational Reflection

The command

(noconflict)

is called a strategy.

The combination of a strategy with computational reflection is very powerful for proving results with complicated proofs very quickly.
Recursion and Computational Reflection

- A proof tree can be complicated
- A strategy can form the tree automatically in PVS
Recursion and Computational Reflection

It isn’t hard to decide when you need to split:

Yogi Berra: “When you come to a fork in the road, take it!”

- PVS can figure this out as well
- A strategy can tell PVS to split at each splitting node so that forming the tree is automatic
- It can also prove the result at each terminating node
Recursion and Computational Reflection

- The proof in PVS is as big as the tree
- All of this is done in PVS
- Even if we use reflection on the terminating nodes, forming a huge tree is slow in PVS
Define the reflection function $Q(p)$ as a recursive function that computes the whole tree.

... and determines whether the result is true.

Then the proof in PVS is just reduced to evaluating $Q(p)$ in PVS.

Which it does recursively in LISP by recreating the tree.
When Computational Reflection is Really Powerful

Instead of having PVS develop a proof that looks like

Define the recursive reflection function \( Q(p) \) in LISP whose execution looks like
When Computational Reflection is Really Powerful

After proving $P(p)$ in this way, the branch of the proof tree where $P(p)$ was proved is now a single node.
When Computational Reflection is Really Powerful

The proof tree becomes

becomes
Now the proof has the same length in PVS regardless of the size of the sub-tree where $P(p)$ is proved

But this is not always possible

Every node in the recursion of $Q(p)$ must be composed of only ground terms

So no variables, existential quantifiers, infinite universal quantifiers, or square roots

Coming up with a $Q$ so that its execution mimics the proof tree can be difficult
Examples

Sat Solving

Nonlinear Arithmetic

Any other problems with recursive proofs