Computational Reflection: Automatically Proving Difficult Things

Anthony Narkawicz

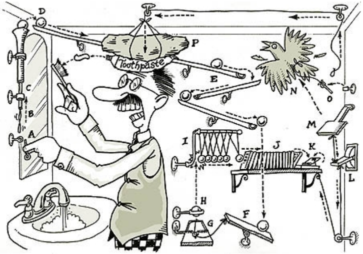
NASA Langley Formal Methods Group

anthony.narkawicz@nasa.gov

October 2012

Computational Reflection

Suppose we are proving the correctness of some system,



http://hhe.wikispaces.com/Rube+Goldberg+Machines

Part way through the proof, we must show:

```
    |-
{1} EXISTS (x:real): 3 x^2 -5*x+2 = 0
```

Computational Reflection

A little while later, we have to prove:

```
    |-
{1} EXISTS (x:real): 2 x^2 -3*x+1 = 0
```

And later,

```
    |-
{1} EXISTS (x:real): 12 x^2 -10*x+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:real): bool = b^2 - 4*a*c >= 0
```

We can then prove the following lemma in PVS

```
quadratic_solvable : LEMMA
  FORALL (a,b,c:real):
     (EXISTS (x:real): a x^2 -b*x+c = 0)
        IFF
     D(a,b,c)
```

Computational Reflection

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

The next time we have to prove something like

```
    |-
{1} EXISTS (x:real): 2 x^2 -3*x+1 = 0
```

we can just type

```
(lemma ‘‘quandratic_solvable’’)
(inst?)
(assert)
(hide
(-1 -2))
```

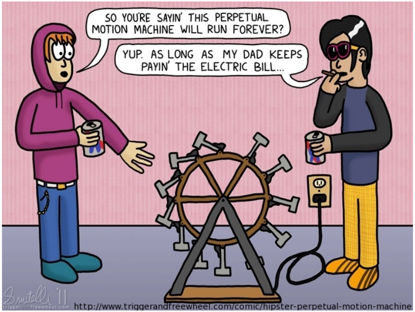
which turns the sequent into

```
    |-
{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} EXISTS (x:real): 10 x^2 -2*x+1 = 0
```

This FAILS:

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

Computational Reflection

Proving that a quadratic has a root:

Can't be evaluated

Can be evaluated (e.g. with grind)

```
|-
{1} EXISTS (x:real): 2 x^2 -3*x+1 = 0
```

A few simple commands

```
|-
{1} D(2,-3,1)
```

Computational reflection is similar:

Can't be evaluated

Can be evaluated (with (eval-formula))

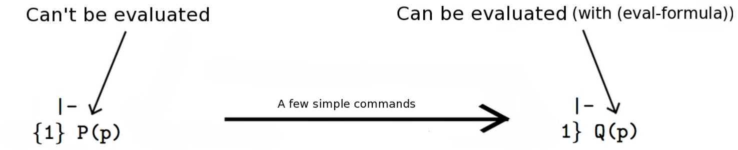
```
|-
{1} P(p)
```

A few simple commands

```
|-
1} Q(p)
```

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

```
    |-
{1} EXISTS (x:real): 2^400 * x^2 +2^600*x+2^100 = 0
```

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

```
    |-
{1} D(2^400,2^600,2^100)
```

This proves with (grind)

Computational Reflection

```
    |-
{1} D(2^400,2^600,2^100)
```

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.

Computational Reflection

We can evaluate

```
    |-
{1} D(2^400,2^600,2^100)
```

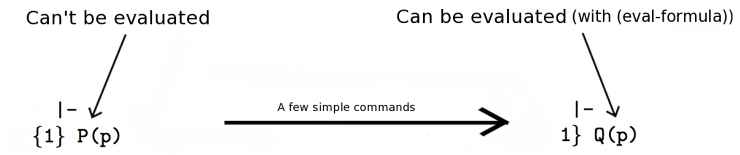
using

```
 (eval-formula)
```

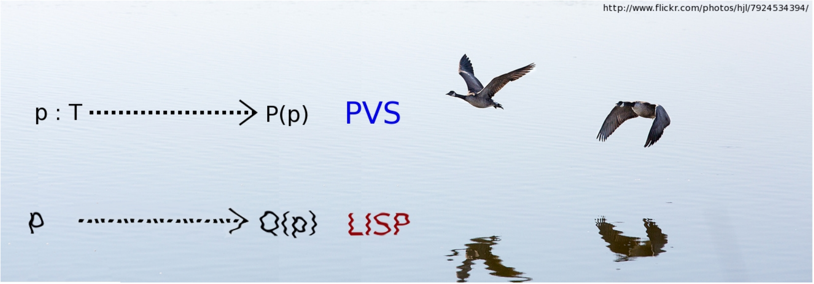
THIS is computational reflection

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

$$\text{IF} \quad a^2 \geq 0 \quad \text{THEN} \quad 1 \quad \text{ELSE} \quad 0 \quad \text{ENDIF}$$

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

However, it can compute

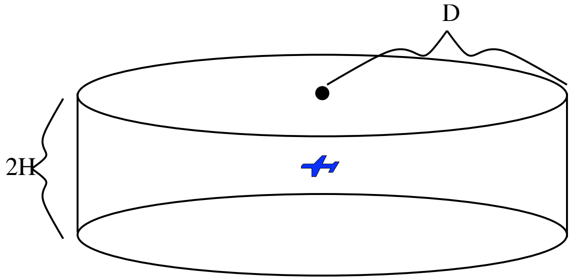
$$\text{IF} \quad 2^2 \geq 0 \quad \text{THEN} \quad 1 \quad \text{ELSE} \quad 0 \quad \text{ENDIF}$$

Example: Conflict Detection

<u>Conflicts</u>

- Minimum Horizontal Distance D
- Minimum Vertical Distance H

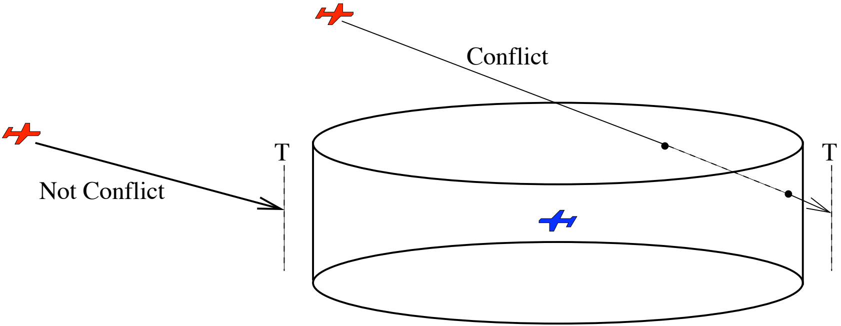
The Protected Zone: A Cylinder



Example: Conflict Detection

The Problem: Detect Conflicts Within a Lookahead Time T

Conflict: *Exists a time* $t \in [0, T]$ *such that the red plane is inside the cylinder at time* t.



Example: Conflict Detection

Aircraft	Position	Velocity
ownship	$\mathbf{s}_o$	$\mathbf{v}_o$
intruder	$\mathbf{s}_i$	$\mathbf{v}_i$
relative	$\mathbf{s} = \mathbf{s}_o - \mathbf{s}_i$	$\mathbf{v} = \mathbf{v}_o - \mathbf{v}_i$

$$\mathit{conflict?}(D, H, \mathbf{s}, \mathbf{v}) \equiv \exists t \geq 0 : \|\mathbf{s} + t\mathbf{v}\| < D \;\; \texttt{and} \;\; |s_z + t v_z| < H$$

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?*$(D, H, \mathbf{s}, \mathbf{v})$ can't be evaluated.

Solution: Replace *conflict?*$(D, H, \mathbf{s}, \mathbf{v})$ with something equivalent that can be evaluated (computational reflection)

Example: Conflict Detection

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

```
cd3D?(s,v) : bool =
    IF v`z = 0 AND abs(s`z) < H
      THEN sqv(vect2(s) + B * vect2(v)) < sq(D) OR
            IF sqv(vect2(s)) = sq(D) AND B = 0 THEN vect2(s) * vect2(v)
            ELSE sq(min(max(B * sqv(vect2(v)), -(vect2(s) * vect2(v))),
                        T * sqv(vect2(v))))
                  + sqv(vect2(v)) * sqv(vect2(s)) + 2 * (min(max(B * sqv(vect2(v)), -(vect2(s) * vect2(v))),
                        T * sqv(vect2(v))) * (vect2(s) * vect2(v))) - sq(D) * sqv(vect2(v))
            ENDIF < 0
    ELSIF v`z /= 0 AND
           max(-H - sign(v`z) * s`z, B * abs(v`z)) <
            min(H - sign(v`z) * s`z, T * abs(v`z))
      THEN sqv(vect2(abs(v`z) * s) + max(-H - sign(v`z) * s`z, B * abs(v`z)) * vect2(v))
            < sq(D * abs(v`z))
            OR
            IF sqv(vect2(abs(v`z) * s)) = sq(D * abs(v`z)) AND
                max(-H - sign(v`z) * s`z, B * abs(v`z)) = 0
              THEN vect2(abs(v`z) * s) * vect2(v)
            ELSE sq(min(max(max(-H - sign(v`z) * s`z, B * abs(v`z)) * sqv(vect2(v)),
                            -(vect2(abs(v`z) * s) * vect2(v))),
                        min(H - sign(v`z) * s`z, T * abs(v`z)) * sqv(vect2(v))))
                  + sqv(vect2(v)) * sqv(vect2(abs(v`z) * s)) + 2 *
                   (min(max(max(-H - sign(v`z) * s`z, B * abs(v`z)) * sqv(vect2(v)),
                            -(vect2(abs(v`z) * s) * vect2(v))),
                        min(H - sign(v`z) * s`z, T * abs(v`z)) * sqv(vect2(v)))
                    * (vect2(abs(v`z) * s) * vect2(v)))
                  - sq(D * abs(v`z)) * sqv(vect2(v))
            ENDIF < 0
    ELSE FALSE
    ENDIF
```

```
cd3d_correct : LEMMA
  FORALL (s,v:Vect3,D,H:posreal):
    conflict?(D,H,s,v)
    IFF
    cd3d(D,H,s,v)
```

Example: Conflict Detection

Given 10K lemmas of the form

```
not_conflict_8741: LEMMA
  D = 5 AND
  H = 1000 AND
  s = (21,-5,-100) AND
  v = (-551,-1,300)
  IMPLIES
  NOT conflict?(D,H,s,v)
```

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

conflict? is replaced by $cd3d$, which is then evaluated using (eval-formula).

Example: Conflict Detection

```
{-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

```
not_conflict_8741: LEMMA
  D = 5 AND
  H = 1000 AND
  s = (21,-5,-100) AND
  v = (-551,-1,300)
  IMPLIES
  NOT conflict?(D,H,s,v)
```

can be proved by just typing

```
(noconflict)
```

Strategies and Computational Reflection

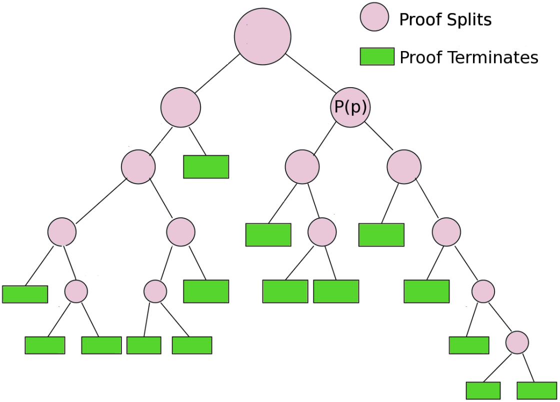
The command

```
(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

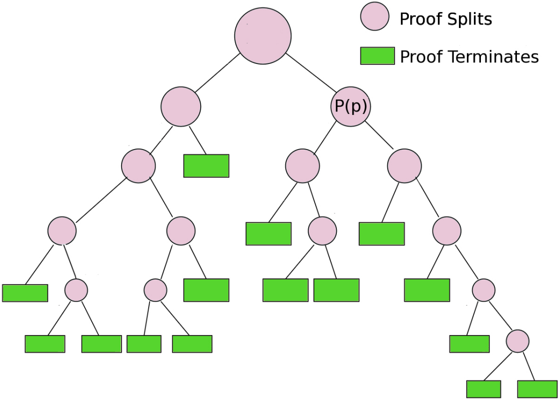
Recursion and Computational Reflection

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



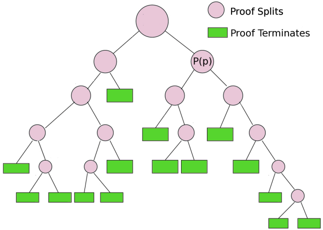
- The proof in PVS is as big as the tree
- All of this is done in PVS
- Even if we use reflection on the terminating nodes, forming a huge tree is slow in PVS

When Computational Reflection is Really Powerful

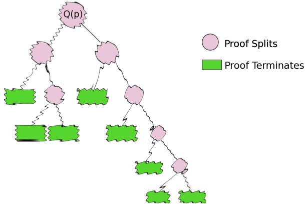
- Define the reflection function Q(p) as a recursive function that computes the whole tree
- ... and determines whether the result is true
- Then the proof in PVS is just reduced to evaluating Q(p) in PVS
- Which it does recursively in LISP by recreating the tree

When Computational Reflection is Really Powerful

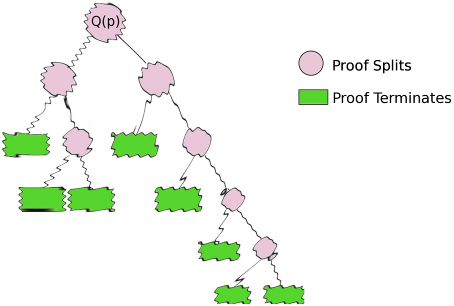
Instead of having PVS develop a proof that looks like



Define the recursive reflection function $Q(p)$ in LISP whose execution looks like



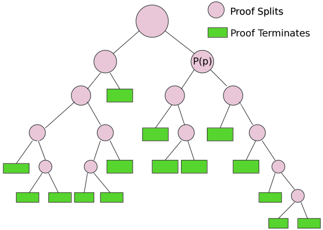
When Computational Reflection is Really Powerful



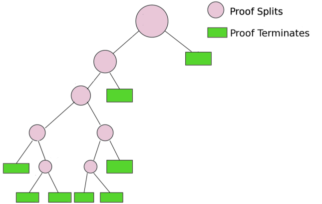
After proving $P(p)$ in this way, the branch of the proof tree where $P(p)$ was proved is now a single node

When Computational Reflection is Really Powerful

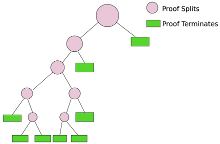
The proof tree



becomes



When Computational Reflection is Really Powerful



- 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