# Real Number Proving in PVS

César A. Muñoz

munoz@nianet.org
http://research.nianet.org/~munoz
National Institute of Aerospace

PVS Class 2007

# Why Real Number Proving in PVS ?

- There are more numbers than integers in *real* life (despite what model-checkers are telling you).
- Conceptually, it is easier to reason on a continuous framework than on a discrete one.
- A lot of classical results in calculus, trigonometry, and continuous mathematics.
- Sometimes you cannot avoid them: hybrid systems, engineering applications, etc.

# I Use a CAS, Why Should I Bother with PVS?

Computer Algebra Systems (CAS):

- Mathematica, Maple, Matlab, Scilab,... offer very powerful symbolic and numerical engines.
- CAS do not aim *soundness*. Singularities and exceptions are well-known problems of CAS.
- CAS do not support specification languages but programming languages.

# CAS vs. Theorem Provers

- Real analysis is not a traditional strength of theorem provers.
- Theorem provers and CAS can be integrated in useful ways:
  - Computer algebra systems can be used to perform mechanical simplifications and find potential solutions.
  - Theorem prover are then used to verify the correctness of a particular solution.

# I. Real Numbers in PVS

- Reals are defined as an uninterpreted subtype of `number` in the prelude library:

  ```
  real: TYPE+ FROM number
  ```

- All numeric constants are `real`:
  - naturals: `0,1,...`
  - integers: `...,-1,0,1,...`
  - rationals: `...,-1/10,...,3/2,...`

- Decimal notation is supported in PVS 4: The decimal number `3.141516` is syntactic sugar for the rational number `31416/10000`.

# PVS's `real` numbers are $\mathbb{R}$

(Rather than floating point numbers)

- All the standard properties: infinite, non-enumerable, $\mathbb{N} \subseteq \mathbb{Z} \subseteq \mathbb{Q} \subseteq \mathbb{R}$, . . . .
- Exact arithmetic: `1/3 + 1/3 + 1/3 = 1`.
- The type `real` is unbounded:

```
googol     : real = 10^100
googolplex : real = 10^googol

googol_prop : LEMMA
  googolplex > googol * googol
```

# PVS's `real` is Built-in

- Numerical expressions can be automatically reduced by the theorem prover (no need to prove `1+1=2`), ...
- ... except for *machine physical limitations*: Try to prove `googol_prop` with `grind`.

*You can still prove `googol_prop` using analytical methods.*

# Subtypes of `real`

```
nzreal   : TYPE+ = {r:real| r /= 0} % Nonzero reals
nnreal   : TYPE+ = {r:real| r >= 0} % Nonnegative reals
npreal   : TYPE+ = {r:real| r <= 0} % Nonpositive reals
negreal  : TYPE+ = {r:real| r < 0} % Negative reals
posreal  : TYPE+ = {r:real| r > 0} % Positive reals

rat      : TYPE+ FROM real
int      : TYPE+ FROM rat
nat      : TYPE+ FROM int
```

*The uninterpreted type `number` is the only `real`'s supertype predefined in PVS: no complex numbers, no hyper-reals, no* $\mathbb{R}^{\infty}$*, ...*

# Predefined Operations

```
+, -, *: [real, real -> real]
/: [real, nzreal -> real]
-: [real -> real]

sgn(x:real)  : int = IF x >= 0 THEN 1 ELSE -1 ENDIF
abs(x:real)  : {nny: nnreal | nny >= x} = ...
max(x,y:real): {z: real | z >= x AND z >= y} = ...
min(x,y:real): {z: real | z <= x AND z <= y} = ...
^(x: real,i:{i:int | x /= 0 OR i >= 0}): real = ...
```

*. . . and what about* $\sqrt{}, \int$, log, exp, sin, cos, tan, $\pi$, lim, . . . *?*

# NASA PVS Libraries

Among many others, the following libraries are available at http://shemesh.larc.nasa.gov/ftp/larc/PVS-library/pvslib.html

- `reals`: Square, square root, quadratic formula, polynomials.
- `analysis`: Real analysis, limits, continuity, derivatives, integrals.
- `series`: Power series, Taylor's theorem.
- `lnexp` and `lnexp_fnd`: Logarithm, exponential, and hyperbolic functions.
- `trig` and `trig_fnd`: Trigonometry.
- `complex`: Complex numbers.
- `float`: Floating point numbers.

# To Be Or Not To Be (Foundational) ?

- Axiomatic theories `trig` and `lnexp` typechek faster.
- Fundational theories `trig_fnd` and `lnexp_fnd` have no axioms, and are updated regularly.
- Careful what you wish for:

```
  |-----
{1}   sin(pi / 2) > 1 / 2

Rule? (grind)
Integral rewrites Integral[real](0, 1, atan_deriv_fn)
  to integral(0, 1, atan_deriv_fn)
atan_value rewrites atan_value(1)
  to integral(0, 1, atan_deriv_fn)
atan rewrites atan(1)
  to integral(0, 1, atan_deriv_fn)
pi rewrites pi
  to 4 * integral(0, 1, atan_deriv_fn)
sin_value rewrites sin_value
  to ...
```

# To Be Or Not To Be (Foundational) ?

- Axiomatic theories `trig` and `lnexp` typechek faster.
- Fundational theories `trig_fnd` and `lnexp_fnd` have no axioms, and are updated regularly.
- Careful what you wish for:

```
  |-----
{1}   sin(pi / 2) > 1 / 2

Rule? (grind)
Integral rewrites Integral[real](0, 1, atan_deriv_fn)
  to integral(0, 1, atan_deriv_fn)
atan_value rewrites atan_value(1)
  to integral(0, 1, atan_deriv_fn)
atan rewrites atan(1)
  to integral(0, 1, atan_deriv_fn)
pi rewrites pi
  to 4 * integral(0, 1, atan_deriv_fn)
sin_value rewrites sin_value
  to ...
```

# II. Low Level Real Number Proving

Real numbers in PVS are axiomatically defined in the PVS prelude:

- Theory `real_axioms`:
  Commutativity, associativity, identity, etc. These properties are known to the decision procedures, so they rarely need to be cited.

- Theory `real_props`:
  Order and cancellation laws. These lemmas are not used automatically by the standard decision procedures.

# If You Really Want to Know . . .

```
real_props: THEORY
 BEGIN
  both_sides_plus_le1: LEMMA x + z <= y + z IFF x <= y
  both_sides_plus_le2: LEMMA z + x <= z + y IFF x <= y
  both_sides_minus_le1: LEMMA x - z <= y - z IFF x <= y
  both_sides_minus_le2: LEMMA z - x <= z - y IFF y <= x
  both_sides_div_pos_le1: LEMMA x/pz <= y/pz IFF x <= y
  both_sides_div_neg_le1: LEMMA x/nz <= y/nz IFF y <= x
  ...
  abs_mult: LEMMA abs(x * y) = abs(x) * abs(y)
  abs_div: LEMMA abs(x / n0y) = abs(x) / abs(n0y)
  abs_abs: LEMMA abs(abs(x)) = abs(x)
  abs_square: LEMMA abs(x * x) = x * x
  abs_limits: LEMMA -(abs(x) + abs(y)) <= x + y AND
                    x + y <= abs(x) + abs(y)
END real_props
```

# Tip 1: Avoid real_props

```
  |-----
{1}   nnx / (nnx + 1) <= 1

Rule? (grind)

  |-----
{1}   nnx / (1 + nnx) <= 1

Rule? (grind :theories "real_props")
div_mult_pos_le1 rewrites nnx / (1 + nnx) <= 1
  to TRUE
Q.E.D.
```

# A Toy Example

```
toy :

  |-----
{1}   x * (1 - x) <= 1

Rule? (grind :theories "real_props")

toy :

  |-----
{1}   x - x * x <= 1
```

# A Toy Example

```
toy :

  |-----
{1}   x * (1 - x) <= 1

Rule?  (grind :theories "real_props")

toy :

  |-----
{1}   x - x * x <= 1
```

# Tip 2: Use `both-sides` To Operate Both Sides of a Formula

(But **only** to **add/subtract** both sides of a formula)

```
toy :

  |-----
{1}   x - x * x <= 1

Rule? (both-sides "-" "1/4")
Applying - 1 / 4 to both sides of an inequality/equality
conjunction, this simplifies to:
toy :

  |-----
{1}   x - x * x - 1 / 4 <= 1 - 1 / 4

Rule?
```

# Tip 2: Use `both-sides` To Operate Both Sides of a Formula

(But **only** to **add/subtract** both sides of a formula)

```
toy :

  |-----
{1}   x - x * x <= 1

Rule? (both-sides "-" "1/4")
Applying - 1 / 4 to both sides of an inequality/equality
conjunction, this simplifies to:
toy :

  |-----
{1}   x - x * x - 1 / 4 <= 1 - 1 / 4

Rule?
```

# Tip 3: Use `case` to Prove What You Want . . .

(No what PVS offers you)

```
Rule? (case "x - x * x - 1 / 4 <= 0")
this yields  2 subgoals:
toy.1 :
{-1}  x - x * x - 1 / 4 <= 0
  |-----
[1]   x - x * x - 1 / 4 <= 1 - 1 / 4

Rule? (assert)
This completes the proof of toy.1.

toy.2 :
  |-----
{1}   x - x * x - 1 / 4 <= 0
[2]   x - x * x - 1 / 4 <= 1 - 1 / 4

Rule? (hide 2)
```

# Tip 3: Use case to Prove What You Want ...

(No what PVS offers you)

```
Rule? (case "x - x * x - 1 / 4 <= 0")
this yields  2 subgoals:
toy.1 :
{-1}  x - x * x - 1 / 4 <= 0
  |-----
[1]   x - x * x - 1 / 4 <= 1 - 1 / 4

Rule? (assert)
This completes the proof of toy.1.

toy.2 :
  |-----
{1}   x - x * x - 1 / 4 <= 0
[2]   x - x * x - 1 / 4 <= 1 - 1 / 4

Rule? (hide 2)
```

# Tip 3: Use case to Prove What You Want . . .

(No what PVS offers you)

```
Rule? (case "x - x * x - 1 / 4 <= 0")
this yields  2 subgoals:
toy.1 :
{-1}  x - x * x - 1 / 4 <= 0
  |-----
[1]   x - x * x - 1 / 4 <= 1 - 1 / 4

Rule? (assert)
This completes the proof of toy.1.

toy.2 :
  |-----
{1}   x - x * x - 1 / 4 <= 0
[2]   x - x * x - 1 / 4 <= 1 - 1 / 4

Rule? (hide 2)
```

# . . . And Arrange Expressions With `case-replace`

```
toy.2 :
  |-----
[1]   x - x * x - 1 / 4 <= 0
Rule? (case-replace
        "x - x * x - 1 / 4 = -(x-1/2)*(x-1/2)"
        :hide? t)
this yields  2 subgoals:
toy.2.1 :
  |-----
{1}   -(x - 1 / 2) * (x - 1 / 2) <= 0

Rule? (grind :theories "real_props")
this simplifies to:
toy.2.1 :
  |-----
{1}   -(x - 1 / 2) * x - (1 * -(x - 1 / 2)) / 2 <= 0
```

# ...And Arrange Expressions With case-replace

```
toy.2 :
  |-----
[1]   x - x * x - 1 / 4 <= 0
Rule? (case-replace
       "x - x * x - 1 / 4 = -(x-1/2)*(x-1/2)"
       :hide? t)
this yields  2 subgoals:
toy.2.1 :
  |-----
{1}   -(x - 1 / 2) * (x - 1 / 2) <= 0

Rule? (grind :theories "real_props")
this simplifies to:
toy.2.1 :
  |-----
{1}   -(x - 1 / 2) * x - (1 * -(x - 1 / 2)) / 2 <= 0
```

## ...And Arrange Expressions With case-replace

```
toy.2 :
  |-----
[1]   x - x * x - 1 / 4 <= 0
Rule? (case-replace
       "x - x * x - 1 / 4 = -(x-1/2)*(x-1/2)"
       :hide? t)
this yields  2 subgoals:
toy.2.1 :
  |-----
{1}   -(x - 1 / 2) * (x - 1 / 2) <= 0

Rule? (grind :theories "real_props")
this simplifies to:
toy.2.1 :
  |-----
{1}   -(x - 1 / 2) * x - (1 * -(x - 1 / 2)) / 2 <= 0
```

# Tip 4: Introduce New Names to Avoid Distribution Laws

```
toy.2.1 :
  |-----
{1}   -(x - 1 / 2) * (x - 1 / 2) <= 0

Rule? (name-replace "X" "(x-1/2)" :hide? nil)
this simplifies to:
toy.2.1 :
{-1}  (x - 1 / 2) = X
  |-----
{1}   -X * X <= 0
```

# Tip 4: Introduce New Names to Avoid Distribution Laws

```
toy.2.1 :
  |-----
{1}   -(x - 1 / 2) * (x - 1 / 2) <= 0

Rule? (name-replace "X" "(x-1/2)" :hide? nil)
this simplifies to:
toy.2.1 :
{-1}  (x - 1 / 2) = X
  |-----
{1}   -X * X <= 0
```

# Finally . . .

```
toy.2.1 :
{-1}  (x - 1 / 2) = X
  |-----
{1}   -X * X <= 0

Rule? (grind :theories "real_props")
This completes the proof of toy.2.1.

toy.2.2 :

  |-----
{1}   x - x * x - 1 / 4 = -(x - 1 / 2) * (x - 1 / 2)
[2]   x - x * x - 1 / 4 <= 0

Rule? (assert)
Q.E.D.
```

# Tip 5: Don't Reinvent the Wheel

(Look into the NASA libraries first!)

Theory `reals@quadratic`:

```
quadratic_le_0 : LEMMA
  a*sq(x) + b*x + c <= 0 IFF
  ((discr(a,b,c) >= 0 AND
    ((a > 0 AND x2(a,b,c) <= x AND x <= x1(a,b,c)) OR
     (a < 0 AND (x <= x1(a,b,c) OR x2(a,b,c) <= x)))) OR
   (discr(a,b,c) < 0 AND c <= 0))
```

# A Simpler Proof

```
  |-----
{1}   x * (1 - x) <= 1

Rule? (lemma "quadratic_le_0"
                ("a" "-1" "b" "1" "c" "-1" "x" "x"))
      (grind)
Trying repeated skolemization, instantiation, and
if-lifting,
Q.E.D.
```

# III. Strategies for Algebraic Manipulations

- Manip: Package for algebraic manipulations of real-valued expressions.
- Developed by B. Di Vito (NASA LaRC).
- Included as part of the PVS NASA Libraries.
- The package consists of:
  - Strategies.
  - Extended notations for formulas and expressions.
  - Emacs extensions.
  - Support functions for strategy developers.

# Manip Strategies: Basic Manipulations

| Strategy | Description |
|---|---|
| `(swap-rel fnums)` | Swap sides and reverse relations |
| `(swap! expr-loc)` | $x \circ y \Rightarrow y \circ x$ |
| `(group! expr-loc LR)` | $(x \circ y) \circ z \Rightarrow x \circ (y \circ z)$ |
| `(flip-ineq fnums)` | Negate and move inequalities |
| `(split-ineq fnum)` | Split $\leq$ ($\geq$) into $<$ ($>$) and $=$ |

# Extended Formula Notation

- Standard
  - `*`: All formulas.
  - `-`: All formulas in the antecedent.
  - `+`: All formulas in the consequent.

- Extended (Manip strategies only)
  - `(^ n1...nk)`: All formulas but `n1,...,nk`
  - `(-^ n1...nk)`: All antecedent formulas but `n1,...,nk`
  - `(+^ n1...nk)`: All consequent formulas but `n1,...,nk`

# (Basic) Extended Expression Notation

- Term indexes:
  - `L,R`: Left- or right-hand side of a formula.
  - `n`: `n`-th term from left to right in a formula.
  - `-n`: `n`-th term from right to left in a formula.
  - `*`: All terms in a formula.
  - `(^ n1...nk)`: All terms in a formula but `n1,...,nk`.

- Location references:
  - `(!  fnum LR i1...in)`: Term in formula `fnum`, Left- or Right-hand side, at recursive path location `i1...ik`.

# Examples

```
{-1}  x * r + y * r + 1 >= r - 1
  |-----
{1}   r = y * 2 * x + 1

Rule? (swap-rel -1)

{-1}  r - 1 <= x * r + y * r + 1
  |-----
[1]   r = y * 2 * x + 1

Rule? (swap! (! -1 R 1))

{-1}  r - 1 <= r * x + y * r + 1
  |-----
[1]   r = y * 2 * x + 1
```

# Examples

```
{-1}  x * r + y * r + 1 >= r - 1
  |-----
{1}   r = y * 2 * x + 1

Rule? (swap-rel -1)

{-1}  r - 1 <= x * r + y * r + 1
  |-----
[1]   r = y * 2 * x + 1

Rule? (swap! (! -1 R 1))

{-1}  r - 1 <= r * x + y * r + 1
  |-----
[1]   r = y * 2 * x + 1
```

# Examples

```
{-1}  x * r + y * r + 1 >= r - 1
  |-----
{1}   r = y * 2 * x + 1

Rule? (swap-rel -1)

{-1}  r - 1 <= x * r + y * r + 1
  |-----
[1]   r = y * 2 * x + 1

Rule? (swap! (! -1 R 1))

{-1}  r - 1 <= r * x + y * r + 1
  |-----
[1]   r = y * 2 * x + 1
```

# Examples

```
{-1}  x * r + y * r + 1 >= r - 1
  |-----
{1}   r = y * 2 * x + 1

Rule? (swap-rel -1)

{-1}  r - 1 <= x * r + y * r + 1
  |-----
[1]   r = y * 2 * x + 1

Rule? (swap! (! -1 R 1))

{-1}  r - 1 <= r * x + y * r + 1
  |-----
[1]   r = y * 2 * x + 1
```

```
{-1}  r - 1 <= r * x + y * r + 1
  |-----
[1]   r = y * 2 * x + 1

Rule? (group! (! 1 R 1) R)

[-1]  r - 1 <= r * x + y * r + 1
  |-----
{1}   r = y * (2 * x) + 1

Rule? (flip-ineq -1)

  |-----
{1}   r - 1 > r * x + y * r + 1
[2]   r = y * (2 * x) + 1
```

```
{-1}  r - 1 <= r * x + y * r + 1
  |-----
[1]   r = y * 2 * x + 1

Rule? (group! (! 1 R 1) R)

[-1]  r - 1 <= r * x + y * r + 1
  |-----
{1}   r = y * (2 * x) + 1

Rule? (flip-ineq -1)

  |-----
{1}   r - 1 > r * x + y * r + 1
[2]   r = y * (2 * x) + 1
```

```
{-1}  r - 1 <= r * x + y * r + 1
  |-----
[1]   r = y * 2 * x + 1

Rule? (group! (! 1 R 1) R)

[-1]  r - 1 <= r * x + y * r + 1
  |-----
{1}   r = y * (2 * x) + 1

Rule? (flip-ineq -1)

  |-----
{1}   r - 1 > r * x + y * r + 1
[2]   r = y * (2 * x) + 1
```

```
{-1}  r - 1 <= r * x + y * r + 1
  |-----
[1]   r = y * 2 * x + 1

Rule? (group! (! 1 R 1) R)

[-1]  r - 1 <= r * x + y * r + 1
  |-----
{1}   r = y * (2 * x) + 1

Rule? (flip-ineq -1)

  |-----
{1}   r - 1 > r * x + y * r + 1
[2]   r = y * (2 * x) + 1
```

```
  |-----
{1}   r - 1 > r * x + y * r + 1
[2]   r = y * (2 * x) + 1

Rule? (split-ineq 1)
Splitting off the equality case from formula 1,
this yields  2 subgoals:
{-1}  r - 1 = r * x + y * r + 1
  |-----
[1]   r - 1 > r * x + y * r + 1
[2]   r = y * (2 * x) + 1

Rule? (postpone)
  |-----
{1}   r - 1 = r * x + y * r + 1
[2]   r - 1 > r * x + y * r + 1
[3]   r = y * (2 * x) + 1
```

```
  |-----
{1}   r - 1 > r * x + y * r + 1
[2]   r = y * (2 * x) + 1

Rule? (split-ineq 1)
Splitting off the equality case from formula 1,
this yields  2 subgoals:
{-1}  r - 1 = r * x + y * r + 1
  |-----
[1]   r - 1 > r * x + y * r + 1
[2]   r = y * (2 * x) + 1

Rule? (postpone)
  |-----
{1}   r - 1 = r * x + y * r + 1
[2]   r - 1 > r * x + y * r + 1
[3]   r = y * (2 * x) + 1
```

# More Strategies

| Strategy | Description |
|---|---|
| `(mult-by fnums term)` | Multiply formula by term |
| `(div-by fnums term)` | Divide formula by term |
| `(move-terms fnum L\|R tnums)` | Move additive terms left and right |
| `(isolate fnum L\|R tnum)` | Isolate additive terms |
| `(cross-mult fnums)` | Perform cross-multiplications |
| `(factor fnums)` | Factorize formulas |
| `(factor! expr-loc)` | Factorize terms |
| `(mult-eq fnum fnum)` | Multiply equalities |
| `(mult-ineq fnum fnum)` | Multiply inequalities |

# More Examples

```
{-1}  (x * r + y) / pa > (r - 1) / pb
  |-----
{1}   r - y * 2 * x = 1

Rule? (cross-mult -1)

{-1}  pb * r * x + pb * y > pa * r - pa
  |-----
[1]   r - y * 2 * x = 1

Rule? (isolate 1 L 1)

[-1]  pb * r * x + pb * y > pa * r - pa
  |-----
{1}   (r = 1 + y * 2 * x)
```

# More Examples

```
{-1}  (x * r + y) / pa > (r - 1) / pb
  |-----
{1}   r - y * 2 * x = 1

Rule? (cross-mult -1)

{-1}  pb * r * x + pb * y > pa * r - pa
  |-----
[1]   r - y * 2 * x = 1

Rule? (isolate 1 L 1)

[-1]  pb * r * x + pb * y > pa * r - pa
  |-----
{1}   (r = 1 + y * 2 * x)
```

# More Examples

```
{-1}  (x * r + y) / pa > (r - 1) / pb
  |-----
{1}   r - y * 2 * x = 1

Rule? (cross-mult -1)

{-1}  pb * r * x + pb * y > pa * r - pa
  |-----
[1]   r - y * 2 * x = 1

Rule? (isolate 1 L 1)

[-1]  pb * r * x + pb * y > pa * r - pa
  |-----
{1}   (r = 1 + y * 2 * x)
```

# More Examples

```
{-1}  (x * r + y) / pa > (r - 1) / pb
  |-----
{1}   r - y * 2 * x = 1

Rule? (cross-mult -1)

{-1}  pb * r * x + pb * y > pa * r - pa
  |-----
[1]   r - y * 2 * x = 1

Rule? (isolate 1 L 1)

[-1]  pb * r * x + pb * y > pa * r - pa
  |-----
{1}   (r = 1 + y * 2 * x)
```

```
{-1}  x * y - pa + na < x * na * pa
{-2}  r - y * 2 * x = 1
  |-----
{1}   2 * pa = 2 * x + 2 * y

Rule? (move-terms -1 L (2 3))

{-1}  (x * y < x * na * pa + pa - na)
[-2]  r - y * 2 * x = 1
  |-----
[1]   2 * pa = 2 * x + 2 * y

Rule? (factor 1)

[-1]  (x * y < x * na * pa + pa - na)
[-2]  r - y * 2 * x = 1
  |-----
{1}   2 * pa = 2 * (x + y)
```

```
{-1}  x * y - pa + na < x * na * pa
{-2}  r - y * 2 * x = 1
  |-----
{1}   2 * pa = 2 * x + 2 * y

Rule? (move-terms -1 L (2 3))

{-1}  (x * y < x * na * pa + pa - na)
[-2]  r - y * 2 * x = 1
  |-----
[1]   2 * pa = 2 * x + 2 * y

Rule? (factor 1)

[-1]  (x * y < x * na * pa + pa - na)
[-2]  r - y * 2 * x = 1
  |-----
{1}   2 * pa = 2 * (x + y)
```

```
{-1}  x * y - pa + na < x * na * pa
{-2}  r - y * 2 * x = 1
  |-----
{1}   2 * pa = 2 * x + 2 * y

Rule? (move-terms -1 L (2 3))

{-1}  (x * y < x * na * pa + pa - na)
[-2]  r - y * 2 * x = 1
  |-----
[1]   2 * pa = 2 * x + 2 * y

Rule? (factor 1)

[-1]  (x * y < x * na * pa + pa - na)
[-2]  r - y * 2 * x = 1
  |-----
{1}   2 * pa = 2 * (x + y)
```

```
{-1}  x * y - pa + na < x * na * pa
{-2}  r - y * 2 * x = 1
  |-----
{1}   2 * pa = 2 * x + 2 * y

Rule? (move-terms -1 L (2 3))

{-1}  (x * y < x * na * pa + pa - na)
[-2]  r - y * 2 * x = 1
  |-----
[1]   2 * pa = 2 * x + 2 * y

Rule? (factor 1)

[-1]  (x * y < x * na * pa + pa - na)
[-2]  r - y * 2 * x = 1
  |-----
{1}   2 * pa = 2 * (x + y)
```

```
[-1]  (x * y < x * na * pa + pa - na)
[-2]  r - y * 2 * x = 1
  |-----
{1}   2 * pa = 2 * (x + y)

Rule? (mult-eq -1 -2)

{-1}  (x * y)*(r - y * 2 * x) < (x * na * pa + pa - na)*1
[-2]  (x * y < x * na * pa + pa - na)
[-3]  r - y * 2 * x = 1
  |-----
[1]   2 * pa = 2 * (x + y)

Rule? (div-by 1 "2")


...
  |-----
{1}   (pa = (x + y))
```

```
[-1]  (x * y < x * na * pa + pa - na)
[-2]  r - y * 2 * x = 1
  |-----
{1}   2 * pa = 2 * (x + y)

Rule? (mult-eq -1 -2)

{-1}  (x * y)*(r - y * 2 * x) < (x * na * pa + pa - na)*1
[-2]  (x * y < x * na * pa + pa - na)
[-3]  r - y * 2 * x = 1
  |-----
[1]   2 * pa = 2 * (x + y)

Rule? (div-by 1 "2")

...
  |-----
{1}   (pa = (x + y))
```

```
[-1]  (x * y < x * na * pa + pa - na)
[-2]  r - y * 2 * x = 1
  |-----
{1}   2 * pa = 2 * (x + y)

Rule? (mult-eq -1 -2)

{-1}  (x * y)*(r - y * 2 * x) < (x * na * pa + pa - na)*1
[-2]  (x * y < x * na * pa + pa - na)
[-3]  r - y * 2 * x = 1
  |-----
[1]   2 * pa = 2 * (x + y)

Rule? (div-by 1 "2")

...
  |-----
{1}   (pa = (x + y))
```

```
[-1]  (x * y < x * na * pa + pa - na)
[-2]  r - y * 2 * x = 1
  |-----
{1}   2 * pa = 2 * (x + y)

Rule? (mult-eq -1 -2)

{-1}  (x * y)*(r - y * 2 * x) < (x * na * pa + pa - na)*1
[-2]  (x * y < x * na * pa + pa - na)
[-3]  r - y * 2 * x = 1
  |-----
[1]   2 * pa = 2 * (x + y)

Rule? (div-by 1 "2")

...
  |-----
{1}   (pa = (x + y))
```

# The Field Package

- Field: A simplification procedure for the field of real numbers.
- Included as part of the PVS NASA Libraries.
- The package consists of:
  - The strategy `field`.
  - Several *extra-tegies*.

## field

```
{-1}  vox > 0
{-2}  s * s - D*D > D
{-3}  s * vix * voy - s * viy * vox /= 0
{-4}  ((s * s - D*D) * voy - D * vox * sqrt(s*s - D*D))/
       (s * (vix * voy - vox * viy)) * s * vox /= 0
{-5}   voy * sqrt(s * s - D*D) - D * vox /= 0
  |-----
{1}  (viy * sqrt(s * s - D*D) - vix * D) /
      (voy * sqrt(s * s - D*D) - vox * D) =
      (D*D - s * s) / (((s * s - D*D) * voy - D * vox *
      sqrt(s * s - D*D)) /
      (s * (vix * voy - vox * viy)) * s * vox) +
      vix / vox

Rule? (field 1)
Q.E.D.
```

# field

```
{-1}  vox > 0
{-2}  s * s - D*D > D
{-3}  s * vix * voy - s * viy * vox /= 0
{-4}  ((s * s - D*D) * voy - D * vox * sqrt(s*s - D*D))/
       (s * (vix * voy - vox * viy)) * s * vox /= 0
{-5}   voy * sqrt(s * s - D*D) - D * vox /= 0
  |-----
{1}  (viy * sqrt(s * s - D*D) - vix * D) /
      (voy * sqrt(s * s - D*D) - vox * D) =
      (D*D - s * s) / (((s * s - D*D) * voy - D * vox *
      sqrt(s * s - D*D)) /
      (s * (vix * voy - vox * viy)) * s * vox) +
      vix / vox

Rule? (field 1)
Q.E.D.
```

# Some Extra-tegies

| Strategy | Description |
|---|---|
| (grind-reals) | grind + real_props |
| (cancel-by fnum term) | Cancel a common term in a formula |
| (skoletin fnum) | Skolemize let-in expressions |
| (skeep fnum) | Skolemize with same variable names |
| (neg-formula fnum) | Negate a formula |
| (add-formula fnum fnum) | Add two formulas |

# Forget Tip 1 and Tip 4, Use grind-reals

```
  |-----
{1}   (x - 1 / 2) * (x - 1 / 2) >= 0

Rule? (grind-reals :nodistrib 1)

Q.E.D.
```

# Forget Tip 1 and Tip 4, Use `grind-reals`

```
  |-----
{1}   (x - 1 / 2) * (x - 1 / 2) >= 0

Rule? (grind-reals :nodistrib 1)

Q.E.D.
```

# cancel-by

```
{-1}  4 * (pa * pb) + (pa * 6) * pa = pa * ((c + 1) * 2)
  |-----
{1}   2 * pb + 3 * pa = c

Rule? (cancel-by -1 "2*pa")

{-1}  (3 * pa) + (2 * pb) = 1 + c
  |-----
{1}   2 * pa = 0
{2}   3 * pa + 2 * pb = c
```

# cancel-by

```
{-1}  4 * (pa * pb) + (pa * 6) * pa = pa * ((c + 1) * 2)
  |-----
{1}   2 * pb + 3 * pa = c

Rule? (cancel-by -1 "2*pa")

{-1}  (3 * pa) + (2 * pb) = 1 + c
  |-----
{1}   2 * pa = 0
{2}   3 * pa + 2 * pb = c
```

# PVS's Let-in Expressions

- Let-in expressions are used in PVS to introduce local definitions.

- They are automatically unfolded by the theorem prover.

```
  |-----
{1}   LET a = a * y + 2 IN
        LET b = a + x IN
          LET c = a + b IN -b + 4 * a * c / 2 = 0

Rule? (assert)

  |-----
{1}   (32 + 8 * (x*x * y*y) + 4 * (x*x*y) + 16 * (x*y) +
        16 * (x*y) + 8*x) / 2 + -(2 + x*y + x) = 0
```

# PVS's Let-in Expressions

- Let-in expressions are used in PVS to introduce local definitions.

- They are automatically unfolded by the theorem prover.

```
  |-----
{1}   LET a = a * y + 2 IN
        LET b = a + x IN
          LET c = a + b IN -b + 4 * a * c / 2 = 0

Rule? (assert)

  |-----
{1}   (32 + 8 * (x*x * y*y) + 4 * (x*x*y) + 16 * (x*y) +
        16 * (x*y) + 8*x) / 2 + -(2 + x*y + x) = 0
```

# Let-in Expressions Go Wild

```
  |-----
{1}   LET a = (x + 1) IN LET b = a * a IN
      LET c = b * b IN c * c >= a

Rule? (assert)

  |-----
{1} 1 + x + (x*x*x*x*x*x*x*x + x*x*x*x*x*x*x)
        + (x*x*x*x*x*x*x + x*x*x*x*x*x)
        + (x*x*x*x*x*x*x + x*x*x*x*x*x)
      ...
        + (x*x + x)
        + (x*x + x)
        + (x*x + x)
        >= 1 + x
```

# Let-in Expressions Go Wild

```
  |-----
{1}   LET a = (x + 1) IN LET b = a * a IN
      LET c = b * b IN c * c >= a

Rule? (assert)

  |-----
{1} 1 + x + (x*x*x*x*x*x*x + x*x*x*x*x*x)
        + (x*x*x*x*x*x + x*x*x*x*x)
        + (x*x*x*x*x*x + x*x*x*x*x)
      ...
        + (x*x + x)
        + (x*x + x)
        + (x*x + x)
        >= 1 + x
```

# Tip 8. To unfold a let-in, use `skoletin`

```
   |-----
{1}   LET a = (x + 1) IN LET b = a * a IN
      LET c = b * b IN c * c >= a

Rule? (skoletin 1)
{-1}  a = (x + 1)
   |-----
{1}   LET b = a * a IN LET c = b * b IN c * c >= a

Rule? (skoletin* 1)
{-1}  c = b * b
{-2}  b = a * a
[-3]  a = (x + 1)
   |-----
{1}   c * c >= a
```

# Tip 8. To unfold a let-in, use skoletin

```
  |-----
{1}   LET a = (x + 1) IN LET b = a * a IN
      LET c = b * b IN c * c >= a

Rule? (skoletin 1)
{-1}  a = (x + 1)
  |-----
{1}   LET b = a * a IN LET c = b * b IN c * c >= a

Rule? (skoletin* 1)
{-1}  c = b * b
{-2}  b = a * a
[-3]  a = (x + 1)
  |-----
{1}   c * c >= a
```

# Tip 8. To unfold a let-in, use `skoletin`

```
   |-----
{1}   LET a = (x + 1) IN LET b = a * a IN
      LET c = b * b IN c * c >= a

Rule? (skoletin 1)
{-1}  a = (x + 1)
   |-----
{1}   LET b = a * a IN LET c = b * b IN c * c >= a

Rule? (skoletin* 1)
{-1}  c = b * b
{-2}  b = a * a
[-3]  a = (x + 1)
   |-----
{1}   c * c >= a
```

# Tip 8. To unfold a let-in, use `skoletin`

```
  |-----
{1}   LET a = (x + 1) IN LET b = a * a IN
      LET c = b * b IN c * c >= a

Rule? (skoletin 1)
{-1}  a = (x + 1)
  |-----
{1}   LET b = a * a IN LET c = b * b IN c * c >= a

Rule? (skoletin* 1)
{-1}  c = b * b
{-2}  b = a * a
[-3]  a = (x + 1)
  |-----
{1}   c * c >= a
```

# More examples

```
  |-----
{1}   FORALL (nnx: nnreal, x: real):
        nnx > x - nnx*nnx AND x + 2 * nnx*nnx >= 4 * nnx
        IMPLIES nnx > 1

Rule? (skeep)
{-1}  nnx > x - nnx*nnx
{-2}  x + 2 *nnx*nnx >= 4 * nnx
  |-----
{1}   nnx > 1

Rule? (neg-formula -1)
{-1}  nnx*nnx - x > -nnx
[-2]  x + 2 * nnx*nnx >= 4 * nnx
  |-----
[1]   nnx > 1
```

# More examples

```
  |-----
{1}   FORALL (nnx: nnreal, x: real):
        nnx > x - nnx*nnx AND x + 2 * nnx*nnx >= 4 * nnx
        IMPLIES nnx > 1

Rule? (skeep)
{-1}  nnx > x - nnx*nnx
{-2}  x + 2 *nnx*nnx >= 4 * nnx
  |-----
{1}   nnx > 1

Rule? (neg-formula -1)
{-1}  nnx*nnx - x > -nnx
[-2]  x + 2 * nnx*nnx >= 4 * nnx
  |-----
[1]   nnx > 1
```

# More examples

```
  |-----
{1}   FORALL (nnx: nnreal, x: real):
        nnx > x - nnx*nnx AND x + 2 * nnx*nnx >= 4 * nnx
        IMPLIES nnx > 1

Rule? (skeep)
{-1}  nnx > x - nnx*nnx
{-2}  x + 2 *nnx*nnx >= 4 * nnx
  |-----
{1}   nnx > 1

Rule? (neg-formula -1)
{-1}  nnx*nnx - x > -nnx
[-2]  x + 2 * nnx*nnx >= 4 * nnx
  |-----
[1]   nnx > 1
```

# More examples

```
  |-----
{1}   FORALL (nnx: nnreal, x: real):
        nnx > x - nnx*nnx AND x + 2 * nnx*nnx >= 4 * nnx
        IMPLIES nnx > 1

Rule? (skeep)
{-1}  nnx > x - nnx*nnx
{-2}  x + 2 *nnx*nnx >= 4 * nnx
  |-----
{1}   nnx > 1

Rule? (neg-formula -1)
{-1}  nnx*nnx - x > -nnx
[-2]  x + 2 * nnx*nnx >= 4 * nnx
  |-----
[1]   nnx > 1
```

```
{-1}  nnx*nnx - x > -nnx
[-2]  x + 2 * nnx*nnx >= 4 * nnx
  |-----
[1]   nnx > 1

Rule? (add-formulas -1 -2)
{-1}  3 * (nnx*nnx) > -nnx + 4 * nnx
  |-----
[1]   nnx > 1

Rule? (cancel-by -1 "nnx")

Q.E.D.
```

```
{-1}  nnx*nnx - x > -nnx
[-2]  x + 2 * nnx*nnx >= 4 * nnx
  |-----
[1]   nnx > 1

Rule? (add-formulas -1 -2)
{-1}  3 * (nnx*nnx) > -nnx + 4 * nnx
  |-----
[1]   nnx > 1

Rule? (cancel-by -1 "nnx")

Q.E.D.
```

```
{-1}  nnx*nnx - x > -nnx
[-2]  x + 2 * nnx*nnx >= 4 * nnx
  |-----
[1]   nnx > 1

Rule? (add-formulas -1 -2)
{-1}  3 * (nnx*nnx) > -nnx + 4 * nnx
  |-----
[1]   nnx > 1

Rule? (cancel-by -1 "nnx")

Q.E.D.
```

```
{-1}  nnx*nnx - x > -nnx
[-2]  x + 2 * nnx*nnx >= 4 * nnx
  |-----
[1]   nnx > 1

Rule? (add-formulas -1 -2)
{-1}  3 * (nnx*nnx) > -nnx + 4 * nnx
  |-----
[1]   nnx > 1

Rule? (cancel-by -1 "nnx")

Q.E.D.
```

# IV. Strategies for Specialized Domains

- Linear arithmetic via Yices.
- Numerical calculations via Interval.

# Yices

- Yices is a SMT (Satisfiability Modulo Theories) solver developed at SRI.

- Background theories supported by Yices:
  - Linear arithmetic (real and integer): addition and multiplication by scalar.
  - Arrays.
  - Uninterpreted functions.
  - Datatypes.
  - Bit vectors.
  - Quantifiers.

# yices

```
  |-----
{1}   EXISTS (x, y: real): x /= y

Rule? (yices)
(assert
  (not (exists ( x_1::real y_2::real) (/=  x_1 y_2))))

Result = unsat
Logical context is inconsistent. Use (reset) to reset.
unsat
Yices translation of negation is unsatisfiable
Simplifying with Yices,
Q.E.D.
```

# yices

```
  |-----
{1}   EXISTS (x, y: real): x /= y

Rule? (yices)
(assert
  (not (exists ( x_1::real y_2::real) (/=  x_1 y_2))))

Result = unsat
Logical context is inconsistent. Use (reset) to reset.
unsat
Yices translation of negation is unsatisfiable
Simplifying with Yices,
Q.E.D.
```

# yices

```
  |-----
{1}   EXISTS (x, y: real):
      1 <= 3 * x - 3 * y AND 3 * x - 3 * y <= 2

Rule? (yices)
Yices translation of negation is unsatisfiable
Simplifying with Yices,

Q.E.D.
```

*Behind this QED there is* **no** *a "real" PVS proof!*

# yices

```
  |-----
{1}   EXISTS (x, y: real):
      1 <= 3 * x - 3 * y AND 3 * x - 3 * y <= 2

Rule? (yices)
Yices translation of negation is unsatisfiable
Simplifying with Yices,

Q.E.D.
```

*Behind this QED there is **no** a "real" PVS proof!*

# yices

```
  |-----
{1}   EXISTS (x, y: real):
      1 <= 3 * x - 3 * y AND 3 * x - 3 * y <= 2

Rule? (yices)
Yices translation of negation is unsatisfiable
Simplifying with Yices,

Q.E.D.
```

*Behind this QED there is* **no** *a "real" PVS proof!*

# The Interval Package

- A package for interval analysis.
- Exact real calculations including trigonometric and transcendental functions.
- http://research.nianet.org/~munoz/Interval

# numerical

```
  |-----
{1}   sin(6 * pi / 180) + sqrt(2) <= 2.11

Rule? (numerical)

Evaluating formula using numerical approximations,
Q.E.D.
```

*Behind this QED there is a "real" PVS proof!*

# numerical

```
  |-----
{1}   sin(6 * pi / 180) + sqrt(2) <= 2.11

Rule? (numerical)

Evaluating formula using numerical approximations,
Q.E.D.
```

*Behind this QED there is a “real” PVS proof!*

# numerical

```
  |-----
{1}   sin(6 * pi / 180) + sqrt(2) <= 2.11

Rule? (numerical)

Evaluating formula using numerical approximations,
Q.E.D.
```

*Behind this QED there is a “real” PVS proof!*

# instint

```
{-1}  x ## [| 0, 2 |]
  |-----
{1}   sqrt(x) + sqrt(3) < 315 / 100

Rule? (instint)
Proving that an expression is in a given interval,
Q.E.D.
```

# instint

```
{-1}  x ## [| 0, 2 |]
  |-----
{1}   sqrt(x) + sqrt(3) < 315 / 100

Rule? (instint)
Proving that an expression is in a given interval,
Q.E.D.
```

# instint with Splitting

```
{-1}  x ## [| 0, 1 |]
  |-----
{1}   x * (1 - x) ## [| 0, 1 / 3 |]

Rule? (instint)
{-1}  x * (1 - x) ## Mult([|0, 1|], Sub([|1|], [|0, 1|]))
{-2}  x ## [|0, 1|]
  |-----
{1}   Mult([|0, 1|], Sub([|1|], [|0, 1|])) < [|0, 1 / 3|]

Rule? (undo)
Rule? (instint :splitting 6)
Q.E.D.
```

# instint with Splitting

```
{-1}  x ## [| 0, 1 |]
  |-----
{1}   x * (1 - x) ## [| 0, 1 / 3 |]

Rule? (instint)
{-1}  x * (1 - x) ## Mult([|0, 1|], Sub([|1|], [|0, 1|]))
{-2}  x ## [|0, 1|]
  |-----
{1}   Mult([|0, 1|], Sub([|1|], [|0, 1|])) < [|0, 1 / 3|]

Rule? (undo)
Rule? (instint :splitting 6)
Q.E.D.
```

# instint with Splitting

```
{-1}  x ## [| 0, 1 |]
  |-----
{1}   x * (1 - x) ## [| 0, 1 / 3 |]

Rule? (instint)
{-1}  x * (1 - x) ## Mult([|0, 1|], Sub([|1|], [|0, 1|]))
{-2}  x ## [|0, 1|]
  |-----
{1}   Mult([|0, 1|], Sub([|1|], [|0, 1|])) < [|0, 1 / 3|]

Rule? (undo)
Rule? (instint :splitting 6)
Q.E.D.
```

# Essential Tools for the Real Practitioner

- PVS NASA libraries.
- Manip/Field packages.
- Depending on your application: Yices and Interval.