

Developing and Proving Algorithms with PVS

(Part II)

César A. Muñoz

NASA's Langley Research Center

`Cesar.A.Munoz@nasa.gov`

`http://shemesh.larc.nasa.gov/people/cam`



Fifteenth Summer School on Formal Techniques
May 23–29, 2026

PVS Logical Foundation

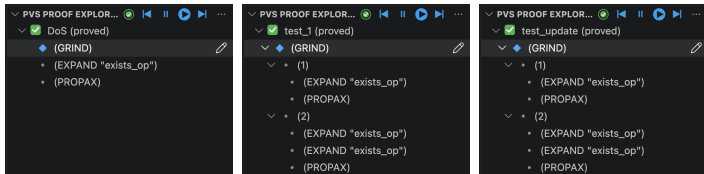
HOL-Based Proof Assistants	Classical	Intuitionistic
Simple Types	Isabelle/HOL	
Dependent Types	PVS	Lean, Rocq

In PVS, in contrast to intuitionistic proof assistants,

- ▶ No Curry-Howard Isomorphism (so proofs are not lambda terms).
- ▶ Types are not first-class terms (so no need for type universes).
- ▶ Types are not unique and typing is undecidable.
- ▶ Functions are not necessarily computable.
- ▶ In the sequent $\Gamma \vdash \Delta$, Δ may have more than 1 formula (and $A, \Gamma \vdash \Delta$ iff $\Gamma \vdash \neg A, \Delta$).
- ▶ Axiom of Choice, Law of Middle Excluded, Double Negation Law are built-in.
- ▶ $1 = 1.0 = 2/2 = \text{sqrt}(1) = 1 = 1+0$

Previously on This Talk ...

- ▶ Types, including records, sub-types, and dependent types.
- ▶ Functions, including recursive functions.
- ▶ Inductive predicates.
- ▶ Type judgements, including recursive type judgements.
- ▶ Proofs:



Recap: UAS Service Supplier

USS provides surveillance services for a set of aircraft within a given operational volume.

- ▶ **Create** USS, including maximum capacity and operational volume (area & height).
- ▶ **Subscribe/unsubscribe** aircraft operations.
- ▶ **Check** if an aircraft is subscribed.
- ▶ **Get** information about aircraft operation.
- ▶ **Check** if maximum capacity has been reached.
- ▶ Update aircraft state information (position and velocity).
- ▶ Retrieve aircraft state.
- ▶ Monitor traffic conflicts.
- ▶ Monitor keep-in geofencing.

Is This Model Still Useful?

```
UTMO[AircraftOp:TYPE+,  
      OperationalArea:TYPE+] : THEORY  
BEGIN  
  
  % Type alias  
  AircraftId : TYPE = string  
  
  % Association list of tuples (id,info)  
  Operations : TYPE = list[[AircraftId,AircraftOp]]  
  
  % Record type  
  USS : TYPE = [#  
    max_capacity : posnat,  
    op_alt      : nnreal,  
    op_area : OperationalArea,  
    operations : Operations  
  #]
```

What About This One?

```
UTM1[AircraftOp:TYPE+,  
      OperationalArea:TYPE+] : THEORY  
BEGIN  
  
  % Type alias  
  AircraftId : TYPE = string  
  
  % Association list of tuples (id,info)  
  Operations : TYPE = [AircraftId->AircraftOp]  
  
  ...
```

Maybe This One?

(Pun Intended)

```
UTM2[AircraftOp:TYPE+,  
      OperationalArea:TYPE+] : THEORY  
BEGIN  
  
  IMPORTING structures@Maybe  
  
  % Type alias  
  AircraftId : TYPE = string  
  
  % Association list of tuples (id,info)  
  Operations : TYPE = [AircraftId->Maybe[AircraftOp]]  
  
  ...
```

Maybe This One?

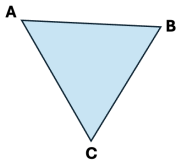
(Pun Intended)

```
UTM2[AircraftOp:TYPE+,  
      OperationalArea:TYPE+] : THEORY  
BEGIN  
  
  IMPORTING structures@Maybe  
  
  % Type alias  
  AircraftId : TYPE = string  
  
  % Association list of tuples (id,info)  
  Operations : TYPE = [AircraftId->Maybe[AircraftOp]]  
  
  ...
```

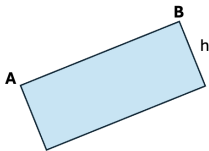
Or This One

```
UTMn[AircraftOp:TYPE+,  
      OperationalArea:TYPE+] : THEORY  
BEGIN  
  
  IMPORTING HashMap  
  
  AircraftId : TYPE = string  
  
  Operations : TYPE = HashMap[AircraftId,AircraftOp]  
  
  ...
```

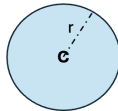
Concretizing The Operational Area



```
mk_triangle(A,B,C)  
: shapes@Triangle_2D
```



```
mk_rectangle(A,B,h)  
: shapes@Rectangle_2D
```



```
mk_circle(C,r)  
: shapes@Circle_2D
```

Data Types

```
UTM : THEORY
```

```
BEGIN
```

```
    IMPORTING shapes@triangle_2D,  
              shapes@circle_2D,  
              shapes@rectangle_2D
```

```
OperationalArea : DATATYPE
```

```
BEGIN
```

```
    CircularArea(get_circle:Circle_2D)  
      : circular_area?
```

```
    RectangularArea(get_rectangle  
      : Rectangle_2D): rectangular_area?
```

```
    TriangularArea(get_triangle:Triangle_2D)  
      : triangular_area?
```

```
END OperationalArea
```

Data Types

```
UTM : THEORY
BEGIN
    IMPORTING shapes@triangle_2D,
              shapes@circle_2D,
              shapes@rectangle_2D

    OperationalArea : DATATYPE
    BEGIN
        CircularArea(get_circle:Circle_2D)
            : circular_area?
        RectangularArea(get_rectangle
            : Rectangle_2D): rectangular_area?
        TriangularArea(get_triangle:Triangle_2D)
            : triangular_area?
    END OperationalArea
```

Data Types

```
UTM : THEORY
BEGIN
    IMPORTING shapes@triangle_2D,
              shapes@circle_2D,
              shapes@rectangle_2D

    OperationalArea : DATATYPE
    BEGIN
        CircularArea(get_circle:Circle_2D)
            : circular_area?
        RectangularArea(get_rectangle
            : Rectangle_2D): rectangular_area?
        TriangularArea(get_triangle:Triangle_2D)
            : triangular_area?
    END OperationalArea
```

Data Types

```
UTM : THEORY
BEGIN
  IMPORTING shapes@triangle_2D,
            shapes@circle_2D,
            shapes@rectangle_2D

  OperationalArea : DATATYPE
  BEGIN
    CircularArea(get_circle:Circle_2D)
      : circular_area?
    RectangularArea(get_rectangle
      : Rectangle_2D): rectangular_area?
    TriangularArea(get_triangle:Triangle_2D)
      : triangular_area?
  END OperationalArea
```

Data Types

```
UTM : THEORY
BEGIN
    IMPORTING shapes@triangle_2D,
              shapes@circle_2D,
              shapes@rectangle_2D

    OperationalArea : DATATYPE
    BEGIN
        CircularArea(get_circle:Circle_2D)
            : circular_area?
        RectangularArea(get_rectangle
            : Rectangle_2D): rectangular_area?
        TriangularArea(get_triangle:Triangle_2D)
            : triangular_area?
    END OperationalArea
```

Enumeration Types

```
UASClass : TYPE = { SMALL, MEDIUM, LARGE }
```

Enumeration types are syntactic sugar for a data type, i.e.,

```
UASClass : DATATYPE =  
BEGIN  
    SMALL: SMALL?  
    MEDIUM: MEDIUM?  
    LARGE: LARGE?  
END UASClass
```

(Inductive Data Types)

- ▶ Data types are inductively defined as the smallest fix point of objects built only using the constructor functions.
- ▶ Data types can be parametric.
- ▶ Data types generate several definitions, e.g., `ord`, `JValue_induction`, `every`, `some`, `<<`, ...

% JSON values can be represented as follows.

JValue : DATATYPE

BEGIN

 JObject(jobject : (injective_dict?[JValue])) : jobject?

 JArray(jarray : list[JValue]) : jarray?

 JNum(jnum : real) : jnum?

 JStr(jstr : string) : jstr?

 JBool(jbool : bool) : jbool?

 JNull : jnull?

END JValue

Putting Everything Together

```
IMPORTING vectors@vect_3D_2D
%% Vect3 : TYPE = [# x:real, y:real, z:real #]
FlightPlan : TYPE = list[Vect3]
```

```
AircraftOp : TYPE = [#
    uas_class    : UASClass,
    callsign     : string,
    operator_id  : string,
    flight_plan  : FlightPlan
#]
```

```
IMPORTING UTM0[AircraftOp,OperationalArea]
```

Exercise

Define the function

`in_operational_volume?(uss:USS)(p:Vect3):bool` that returns TRUE if the point `p` is inside the operational volume of `uss`.

Hints: Check that

1. The vertical component of `p`, i.e., `p.z`, is below `uss.op_alt`.
2. The horizontal component of `p`, i.e., `vect2(p)`, is inside `uss.op_area`.

Look for the names of the functions in `NASALib/shapes` that check if a point is inside a circle, rectangle, and triangle.

Exercise

Define the function

`check_flight_plan(uss:USS,ac_op:AircraftOp)` that returns

- ▶ OK, if every waypoint in the flight plan of `ac_op` lies inside the USS operational volume, or
- ▶ `Failure(wps)` otherwise, where `wps` is the list of waypoints in its flight plan that lie outside the USS operational volume.

Exercise

Modify the function `subscribe` in `UTM.pvs` so that only aircraft operations `ac_op` that satisfy `check_flight_plan(uss,ac_op) = OK` are subscribed. Throw an exception if `check_flight_plan(uss)` returns a `Failure`.

How to Validate If We Specified The Right Problem?

- ▶ Proving properties using the PVS theorem prover.
- ▶ Validating expected outputs using the PVS ground evaluator.

(Animation of Functional Specifications ...

PVSio is

- ▶ a *read-eval-loop* interface to evaluate ground expressions;
- ▶ an extension of the PVS specification language with imperative features;
- ▶ available as the standalone Unix command **pvsio**, in Emacs, through M-x **pvsio**, and in VSCode-PVS.



```
typecheck-file | evaluate-in-pvsio | view-as-markdown
```

```
UTM : THEORY
```

```
BEGIN
```

```
IMPORTING shapes@triangle_2D,  
          shapes@circle_2D,  
          shapes@rectangle_2D
```

Input/Output Operations

```
<PVSio> 1/7;
```

```
==>
```

```
0. $\overline{142857}$ 
```

```
<PVSio> sqrt(2);
```

```
==>
```

```
1.414213...
```

```
<PVSio> LET r = 10 IN
```

```
print("sqrt("+r+")="+sqrt(r));
```

```
sqrt(10)=3.162277
```

```
<PVSio> printf("Hello World ~s",today);
```

```
Hello World "05/10/2026"
```

Random Values

```
<PVSio> RANDOM;
```

```
==>
```

```
0.277062...
```

```
<PVSio> RANDOM=RANDOM;
```

```
==>
```

```
FALSE
```

```
<PVSio> LET r=RANDOM IN r=r;
```

```
==>
```

```
TRUE
```

Random Values

```
<PVSio> RANDOM;
```

```
==>
```

```
0.277062...
```

```
<PVSio> RANDOM=RANDOM;
```

```
==>
```

```
FALSE
```

```
<PVSio> LET r=RANDOM IN r=r;
```

```
==>
```

```
TRUE
```

Random Values

```
<PVSio> RANDOM;
```

```
==>
```

```
0.277062...
```

```
<PVSio> RANDOM=RANDOM;
```

```
==>
```

```
FALSE
```

```
<PVSio> LET r=RANDOM IN r=r;
```

```
==>
```

```
TRUE
```

PVSio Extensions Are Sound

- ▶ From the point of view of the type-checker and theorem prover, PVSio extensions are uninterpreted functions.
- ▶ PVSio extensions are not evaluated by the theorem prover,
- ▶ For PVS, `RANDOM` is just an uninterpreted constant and, therefore, `RANDOM = RANDOM` always holds.

More PVSio Features

- ▶ I/O files.
- ▶ Mutable and global variables.
- ▶ Exceptions.
- ▶ Type introspection.
- ▶ Unbounded loops (possibly non-terminating).
- ▶ Extensionality via user-defined semantic attachments.

PVSio Utilities

- ▶ **PVSioChecker**: Differential testing.
- ▶ **PVSioCSV**: Reader and writer of comma-separated value files.
- ▶ **PVSioRegex**: Regular expressions matcher.
- ▶ **PVSioPEG**: Lexer/parser via Parsing Expression Grammars (PEG).
- ▶ **PVSioJSON**: Reader and writer of JSON files.
- ▶ **PVSioKeyVal**: Persistent key/value store of PVS values.
- ▶ **PVS2C**: Memory efficient C generator for PVS functional specifications.

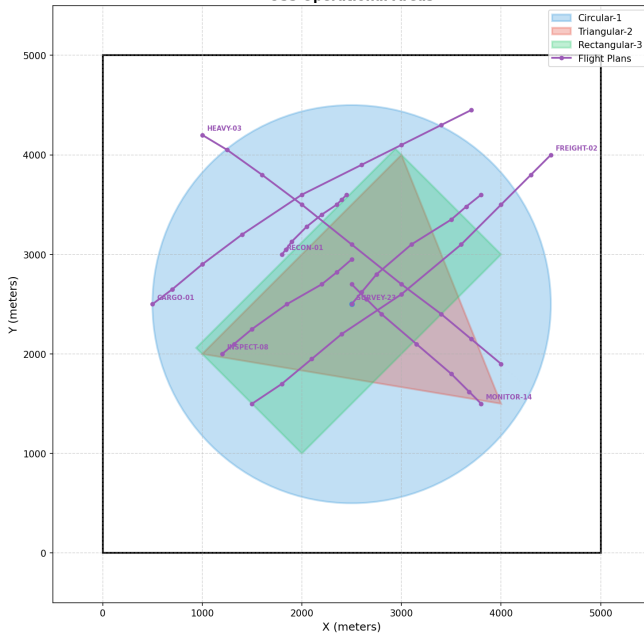
... PVSio References)

- ▶ Website: <http://shemesh.larc.nasa.gov/fm/pvs/PVSio>.
- ▶ *Rapid prototyping in PVS*, C. Muñoz, NASA Contract Report.
- ▶ *Efficiently Executing PVS*, N. Shankar, SRI Technical Report.
- ▶ *Evaluating, Testing, and Animating PVS Specifications*, J. Crow, S. Owre, J. Rushby, N. Shankar, and D. Stringer-Calvert, SRI Technical Report.

Let's Animate UTM

- ▶ Claude generated JSON files representing information of USS Operational Areas and USS Aircraft Operations for different classes of UAS.
- ▶ Claude generated Python code to visualize JSON files.

USS Operational Areas



PVSioJSON@pvsio_json

```
<PVSio> json_from_file("USS-RECTANGULAR-03.json");
```

```
==>
```

```
JObject((: ("capacity", JNum(120)), ("altitude", JNum(125)),  
           ("area_type", JStr("rectangular")),  
           ("p1", JObject((: ("x", JNum(2000)), ("y", JNum(1000)) :))),  
           ("p2", JObject((: ("x", JNum(4000)), ("y", JNum(3000)) :))),  
           ("height", JNum(1500)) :))
```

```
<PVSio> json_from_file("MONITOR-14.json");
```

```
==>
```

```
JObject((: ("uas_class", JStr("MEDIUM")), ("callsign", JStr("MONITOR-14")),  
           ("operator_id", JStr("OP-2026-089")),  
           ("flight_plan",  
            JArray((: JObject((: ("x", JNum(3800)), ("y", JNum(1500)),  
                                   ("z", JNum(0)) :)),  
                    JObject((: ("x", JNum(3680)), ("y", JNum(1620)),  
                                   ("z", JNum(60)) :)),  
                    JObject((: ("x", JNum(3500)), ("y", JNum(1800)),  
                                   ("z", JNum(120)) :)),  
                    JObject((: ("x", JNum(3150)), ("y", JNum(2100)),  
                                   ("z", JNum(120)) :)),  
                    JObject((: ("x", JNum(2800)), ("y", JNum(2400)),  
                                   ("z", JNum(120)) :)),  
                    JObject((: ("x", JNum(2650)), ("y", JNum(2550)),  
                                   ("z", JNum(60)) :)),  
                    JObject((: ("x", JNum(2500)), ("y", JNum(2700)),  
                                   ("z", JNum(0)) :)) :)) :)) :))
```

Parse Exceptions

```
<PVSio> json_from_file("NOTAFILE.json");  
[PVSioException::FileNotFound] File not found (NOTAFILE.json)  
  
<PVSio> json_from_file("USS-MALFORMED-01.json");  
[PVSioException::PEGException] Expecting <,>, or <}>, at line 6, column 2.  
  
<PVSio> |
```

```
❏ PVS > {} USS-MALFORMED-01.json > ...  
1  {  
2    "capacity": 100,  
3    "altitude": 120,  
4    "area_type": "circular",  
5    "center": { "x": 2500.0, "y": 2500.0 }  
6    "radius": 1000.0  
7  }  
8
```

I Prompted Claude...

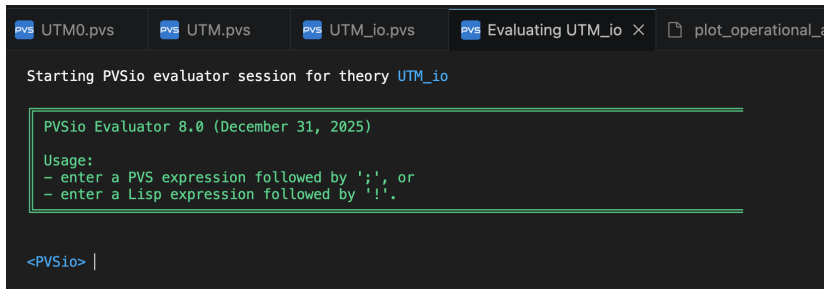
- ▶ To generate PVS functions to translate JSON JValues into USS and AircraftOp data structures.
- ▶ To use PVSExceptions to signal errors, e.g., “Triangle points must be non-collinear”, “Expected array for flight_plan”, etc.
- ▶ To create test cases where exceptions were raised.
- ▶ To create constants of type USS and AircraftOp for each generated JSON file, and document the PVS file with relevant information about the generated code.

... Claude Generated

```
27 % Exception for JSON parsing errors
28 JSONParseError: SimpleException
29
```

```
...
207 % Parse list of Vect3 waypoints from JSON array
208 json2flight_plan(jv: JValue): FlightPlan =
209     IF NOT jarray?(jv) THEN throw(JSONParseError, "Expected array for flight_plan")
210     ELSE json2flight_plan_rec(jarray(jv))
211     ENDIF
212
213 % Parse AircraftOp from JSON object
214 json2aircraft_op(jv: JValue): AircraftOp =
215     (# uas_class := json2uas_class(jget(jv, "uas_class")),
216      callsign := jstr_val(jget(jv, "callsign")),
217      operator_id := jstr_val(jget(jv, "operator_id")),
218      flight_plan := json2flight_plan(jget(jv, "flight_plan"))
219     #)
```

Demo



The screenshot shows a software interface with a dark background. At the top, there is a horizontal bar containing several tabs. From left to right, the tabs are: 'UTM0.pvs', 'UTM.pvs', 'UTM_io.pvs', 'Evaluating UTM_io' (which is highlighted with a blue border and has a close button 'X' to its right), and a partially visible 'plot_operational_'. Below the tabs, the main area displays the text 'Starting PVSio evaluator session for theory UTM_io'. A green rectangular box highlights a section of text that reads: 'PVSio Evaluator 8.0 (December 31, 2025)' followed by 'Usage:' and two bullet points: '- enter a PVS expression followed by ';', or' and '- enter a Lisp expression followed by '!'. At the bottom left of the main area, the prompt '<PVSio> |' is shown.

```
PVS UTM0.pvs PVS UTM.pvs PVS UTM_io.pvs PVS Evaluating UTM_io X plot_operational_

Starting PVSio evaluator session for theory UTM_io

PVSio Evaluator 8.0 (December 31, 2025)

Usage:
- enter a PVS expression followed by ';', or
- enter a Lisp expression followed by '!'.

<PVSio> |
```

(On Proofs, Proof Rules, and Strategies. . .

- ▶ Proofs are built in a procedural style using proof commands, which interactively construct a **tree of sequents** of the form $\Gamma \vdash \Delta$, where Γ and Δ are lists of formulas.
- ▶ In a sequent $\Gamma \vdash \Delta$, the list of formulas in Γ is called the **antecedent** and the list of formulas in Δ is called the **consequent**.
- ▶ The antecedent represents a conjunction and the consequent represents a disjunction, i.e., if $\Gamma = A_1, \dots, A_n$ and $\Delta = B_1, \dots, B_m$, $\Gamma \vdash \Delta$ represents the formula

$$\bigwedge_{0 \leq i \leq n} A_i \implies \bigvee_{0 \leq i \leq m} B_i$$

Proof Construction

- ▶ The root of the tree is the initial goal.
- ▶ Proof rules, which are applied **bottom up**, transform a sequent $\Gamma \vdash \Delta$ into a list of sub-goal sequents $\Gamma_i \vdash \Delta_i$, i.e.,

$$\frac{\Gamma_1 \vdash \Delta_1 \quad \dots \quad \Gamma_k \vdash \Delta_k}{\Gamma \vdash \Delta} \text{ (proof-rule)}$$

- ▶ The construction ends when all the leaves are discharged, i.e., they have the form $\dots \vdash \dots, \text{TRUE}, \dots$
- ▶ Remark: Rules are **not** bi-directional, arriving to an empty sequent “ \vdash **doesn't** mean the goal is false.
- ▶ Some proof rules may not terminate.

Proof Rules¹

- ▶ `(assert ...)`: Decision procedures and auto-rewrites.
- ▶ `(grind ...)`: Super-duper strategy.
- ▶ `(ground ...)`: Propositional simplifications plus decision procedures.
- ▶ `(flatten ...)`: Eliminate conjunctions in the antecedent and disjunctions in the consequent
- ▶ `(skeep ...)`: Eliminate FORALL quantifier in the consequent or EXISTS quantifier in the antecedent by introducing skolem variables.
- ▶ `(lemma ...)`: Introduce lemma to the consequent.
- ▶ `(expand ...)`: Expand definition.
- ▶ `(inst ...)`: Eliminate FORALL quantifier in the antecedent or EXISTS quantifier in consequent.

¹See <https://pvs.csl.sri.com/doc/pvs-prover-guide.pdf> for a comprehensive list of proof commands.

Proof Commands

- ▶ **Proof Rule:** Atomic (trusted) prover command, e.g., `split`, `skolem`, `hide`, `metit`, etc.
- ▶ **Strategy:** A proof command that expands into one or more atomic steps, e.g., `grind`, `ground`, etc.
 - ▶ **Black Box:** Proof command that behaves as an atomic step but can be expanded, e.g., `grind`, `ground`, `interval`, etc.
 - ▶ **Glass Box:** A command that is always expanded, e.g., strategy combinators such as `then`, `if`, `try`, ...²
 - ▶ **Tactic:** A strategy defined inside a proof.

²Black box strategies have a “\$”-glass box variant, e.g., `grind$`, `ground$`, `interval$`, ...

User-Defined Strategies

- ▶ User-defined *strategies* conservatively extend the theorem prover capabilities.
- ▶ Strategies are defined in a macro-based language implemented in Lisp.³
- ▶ Strategies do not compromise the soundness of PVS (as long as the proof context is accessed as read-only).

³Lisp is the implementation language of PVS.

Strategy Language: Basic Steps

- ▶ Any proof command, e.g., (`ground`), (`case ...`), etc.
- ▶ (`skip`) does nothing.
- ▶ (`printf` format ...) prints a formatted message.
- ▶ (`comment` message) adds a persistent comment to the proof branch.
- ▶ (`relabel` label fnums) labels formulas fnums with label.
- ▶ (`delabel` fnums) unlabels formulas in fnums.
- ▶ ...and much more!

Strategy Language: Combinators

- ▶ Sequencing: (**then** step1 ...stepn)
- ▶ Branching: (**branch** step (step1 ...stepn))
- ▶ Binding local variables:
(**let** ((var1 lisp1) ... (varn lispn)) step)
- ▶ Conditional: (**if** lisp step1 step2)
- ▶ Loop: (**repeat** step)
- ▶ Backtracking: (**try** step step1 step2)

Remark: if and let are the only combinators where Lisp code can be used.

Creating Fresh Labels

(Advanced Topic)

- ▶ `(with-fresh-labels ((var1 fnum1) ... (varn fnumn)) steps)`:
Create fresh labels for `fnumi` and binds them to `vari`. Then, sequentially apply `steps` to all branches. All created labels are removed before the strategy exits.
- ▶ Example:

```
(with-fresh-labels
  ((l 1) (m -1))
  (inst? l :where m))
```

Creating Fresh Names

(Advanced Topic)

- ▶ (**with-fresh-names** ((var1 e1) ... (varn en)) steps):
Create fresh names for e_i and binds them to $vari$. Then, sequentially apply steps to all branches. All created names are removed before the strategy exits.
- ▶ Example:

```
(with-fresh-names  
  ((e "x+2") (f "sqrt(x)"))  
  (inst 1 e f))
```

Writing your Own Strategies

- ▶ New strategies are defined in a file named `pvs-strategies` in the current context.
- ▶ PVS automatically loads this file every time the theorem prover is invoked.
- ▶ The `IMPORTING` clause automatically loads any file `pvs-strategies` in the importing chain.

Strategy Definitions

- ▶ `defstep` defines a black-box strategy and its glass-box $\$$ -form:

```
(defstep name (parameters &optional parameters)
  step
  help-string  format-string)
```

- ▶ `defhelper` defines a black-box strategy to be used by other strategies.

```
(defhelper name (parameters &optional parameters)
  step
  help-string  format-string)
```

- ▶ `defstrat` defines a glass-box strategy:

```
(defstrat name (parameters &optional parameters)
  step
  help-string)
```

Example: try-grind

In pvs-strategies:

```
(defstep try-grind ()  
  (else (then (grind)(fail)) (skip))  
  "Tries GRIND. UNDO, if it doesn't discharge  
   the sequent"  
  "Trying GRIND")
```

Example: try-grind

In pvs-strategies:

```
(defstep try-grind ()  
  (else (then (grind)(fail)) (skip))  
  "Tries GRIND. UNDO, if it doesn't discharge  
   the sequent"  
  "Trying GRIND")
```

Example: try-grind

In pvs-strategies:

```
(defstep try-grind ()  
  (else (then (grind)(fail)) (skip))  
  "Tries GRIND. UNDO, if it doesn't discharge  
   the sequent"  
  "Trying GRIND")
```

Example: try-grind

In pvs-strategies:

```
(defstep try-grind ()  
  (else (then (grind)(fail)) (skip))  
  "Tries GRIND. UNDO, if it doesn't discharge  
   the sequent"  
  "Trying GRIND")
```

Example: try-grind

In pvs-strategies:

```
(defstep try-grind ()  
  (else (then (grind)(fail)) (skip))  
  "Tries GRIND. UNDO, if it doesn't discharge  
   the sequent"  
  "Trying GRIND")
```

Example: try-grind

In pvs-strategies:

```
(defstep try-grind ()  
  (else (then (grind)(fail)) (skip))  
  "Tries GRIND. UNDO, if it doesn't discharge  
   the sequent"  
  "Trying GRIND")
```

|-----
{1} $x * (1 + x) \geq 0$

Rule? (help try-grind)

(try-grind/\$) :

Tries GRIND. UNDO, if it doesn't discharge the sequent

Rule? (try-grind)

No change on: (try-grind)

test :

|-----
{1} $x * (1 + x) \geq 0$

|-----
{1} $x * (1 + x) \geq 0$

Rule? (help try-grind)

(try-grind/\$) :

Tries GRIND. UNDO, if it doesn't discharge the sequent

Rule? (try-grind)

No change on: (try-grind)

test :

|-----
{1} $x * (1 + x) \geq 0$

|-----
{1} $x * (1 + x) \geq 0$

Rule? (help try-grind)

(try-grind/\$) :

Tries GRIND. UNDO, if it doesn't discharge the sequent

Rule? (try-grind)

No change on: (try-grind)

test :

|-----
{1} $x * (1 + x) \geq 0$

|-----
{1} $x * (1 + x) \geq 0$

Rule? (help try-grind)

(try-grind/\$) :

Tries GRIND. UNDO, if it doesn't discharge the sequent

Rule? (try-grind)

No change on: (try-grind)

test :

|-----
{1} $x * (1 + x) \geq 0$

test.1 :

$\{-1\} \quad x \geq 0$

|-----

[1] $x * (1 + x) \geq 0$

Rule? (try-grind)

Trying GRIND,

This completes the proof of test.1.

test.1 :

$\{-1\} \quad x \geq 0$

|-----

[1] $x * (1 + x) \geq 0$

Rule? (try-grind)

Trying GRIND,

This completes the proof of test.1.

Glass-Box vs Black-Box Strategies

- ▶ `try-grind` was defined as black-box strategy. Therefore, it is saved in the proof even when it skips.
- ▶ When `try-grind` skips, it would be better not to save the command `grind`, which is expensive.
- ▶ Two alternatives:
 - ▶ Use `try-grind$` instead of `try-grind`.
 - ▶ Define `try-grind` as a glass-box strategy:

```
(defstrat try-grind ()  
  (else (then (grind)(fail)) (skip))  
  "Tries GRIND. UNDO, if it doesn't discharge  
  the sequent")
```

Level of Strategy Development

Easy Repetitive tasks, e.g., `(flatten)(assert)(replace -1) ... (flatten)(assert)(replace -1)`.

Medium Programatic tasks, e.g., `(case "0 <= n AND n < 1"), ... , (case "1023 <= n AND n < 1024")`.

Advanced Control flow, e.g., implementation of a new proof combinator.

Expert Proof search, e.g., implementation of a decision procedure or an heuristic method.

Using Lisp to Access PVS Proof Context

- ▶ Arbitrary Lisp expressions (functions, global variables, etc.) can be included in a strategy file.
- ▶ PVS's data structures are based on various Common Lisp Object System (CLOS) classes. They are available to the strategy programmer through global variables and accessory functions.

Strategy Development Pitfalls

- ▶ Strategies may not terminate, e.g.,
Rule? (repeat (case "0=0"))
- ▶ Strategies may be non-deterministic, e.g.,
Rule? (let ((n (freshname "n"))) (name n "10"))
- ▶ Avoid non-deterministic black-box strategies; they may fail when the proof is rerun.
- ▶ If non-determinism is unavoidable, use glass-box strategies.
- ▶ If fresh identifiers are needed, use the robust glass-box strategies `with-fresh-labels` and `with-fresh-names`.

... Strategy Writing References)

- ▶ Documentation: PVS Prover Guide, N. Shankar, S. Owre, J. Rushby, D. Stringer-Calvert, SRI International:
<http://www.csl.sri.com/pvs.html>.
- ▶ Proceedings of STRATA 2003:
<http://hdl.handle.net/2060/20030067561>.
- ▶ Examples: Manip⁴, Field⁵, Extrategies⁶.
- ▶ Programming: Lisp The Language, G. L. Steele Jr., Digital Press⁷.

⁴<https://shemesh.larc.nasa.gov/fm/pvs/Manip/>

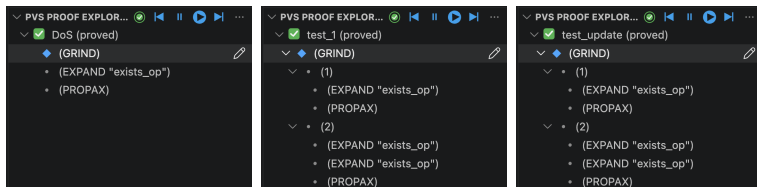
⁵<https://shemesh.larc.nasa.gov/fm/pvs/Field/>

⁶<https://shemesh.larc.nasa.gov/fm/pvs/Extrategies/>

⁷<https://www.cs.cmu.edu/Groups/AI/html/cltl/clm/clm.html>

Exercise

Write a strategy called `uss-tcc` that discharges the proofs below and use it to prove `subscribe_j`, `test_0`, `test_1`, `DoS`, and `test_update`.



Hint: The strategy does the following: first apply `grind`, then to each generated subgoal, expand `exists_op` as many times as needed.

Questions?

- ▶ Acknowledgements: FM Team@LaRC & Friends – <https://shemesh.larc.nasa.gov/fm/fm-main-team.html>:
Ricky Butler, Esther Conrad, Aaron Dutle, Marco Feliu, Alwyn Goodloe, George Hagen, Laura Humphrey, Jeff Maddalon, Paolo Masci, Paul Miner, Mariano Moscato, Natasha Neogi, John Siratt, Tanner Slagel, Lauren White, Michael Holloway, Víctor Carreño, Ben Di Vito, Kelly Hayhurst, Anthony Narkawicz, Lee Pike, Kristin Rozier, Laura Titolo, Jason Upchurch, SRI, INRIA, CEA, CNRS, Entalus, University of Brasilia, University of La Coruña, interns, visitors, ...
- ▶ Resources: <https://shemesh.larc.nasa.gov/fm/fm-main-research.html>
- ▶ PVS: <https://github.com/SRI-CSL/PVS>
- ▶ NASALib: <https://github.com/nasa/pvslib>