Animation of Functional Specifications with PVSio

Mariano M. Moscato[1]

National Institute of Aerospace
Mariano.Moscato@nianet.org

[1]Based on material by César A. Muñoz.

- Most specifications in PVS are functional, e.g.,

```
sqrt_newton(a:nnreal,n:nat): RECURSIVE posreal =
  IF n=0 THEN a+1
  ELSE LET r=sqrt_newton(a,n-1) IN
    (1/2)*(r+a/r)
  ENDIF
  MEASURE n+1
```

- What is the value of `sqrt_newton(2,10)`?

- Most specifications in PVS are functional, e.g.,

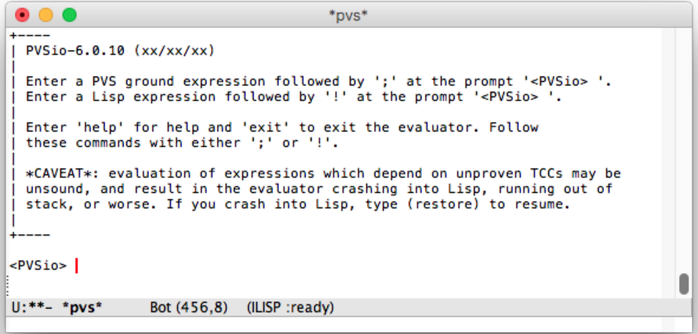
```
sqrt_newton(a:nnreal,n:nat): RECURSIVE posreal =
  IF n=0 THEN a+1
  ELSE LET r=sqrt_newton(a,n-1) IN
    (1/2)*(r+a/r)
  ENDIF
  MEASURE n+1
```

- What is the value of `sqrt_newton(2,10)`?

- Animation is the process of executing a specification to validate its intended semantics.
- Why: It is cheaper, faster, and more fun to test a specification than to prove it.
- How: PVSio.

PVSio is . . .

- an *read-eval-loop* interface to the PVS Ground Evaluator;
- an efficient and sound mechanism to compute within the theorem prover;
- part of the PVS distribution;
- available as the standalone Unix command `pvsio` or through the Emacs command `M-x pvsio`.


pvs
+----
| PVSio-6.0.10 (xx/xx/xx)
|
| Enter a PVS ground expression followed by ';' at the prompt '<PVSio> '.
| Enter a Lisp expression followed by '!' at the prompt '<PVSio> '.
|
| Enter 'help' for help and 'exit' to exit the evaluator. Follow
| these commands with either ';' or '!'.
|
| *CAVEAT*: evaluation of expressions which depend on unproven TCCs may be
| unsound, and result in the evaluator crashing into Lisp, running out of
| stack, or worse. If you crash into Lisp, type (restore) to resume.
|
+----
<PVSio>
U:**- *pvs* Bot (456,8) (ILISP :ready)

```
<PVSio> sqrt_newton(2,10);
==>
1068540411258005424957730996202770251753061700886760050509
2775584086034866316307624567599571273090520553619648095761
8323863188053907381032775618232842813250031327063713965171
4658235752986741761590590866587906685398566655402811587051
1326582300341866167304359343960603343170658488116440998347
6684441998170083079481020253769836865387591260387081397004
4395397342728487283626639303583613156999614503895003382899
3710275557723330463738359457597728824912553479002609510283
8876667217608828542941439658998944011413943276015695324890
7732348479284448531263506286619985710653992842738259074138
7820229684450437162379033859897869490832422027208754929968
4847173162322470343065 7/7555721707723979449648392625272648
5551649317661664229348905711734284588432155133188346299073
5221751911561329826177804181498740064550632899687915531346
8554933480730789681235675239145730235553225685034903270487
8494636528771206171730831540313524856910300210308020618231
5299185952314636972146502157415893978546131789429918177417
2751814809171189000327511471361082396689939198281075027256...
```

```
<PVSio> sqrt_newton(2,10);
==>
1068540411258005424957730996202770251753061700886760050509
2775584086034866316307624567599571273090520553619648095761
8323863188053907381032775618232842813250031327063713965171
4658235752986741761590590866587906685398566655402811587051
1326582300341866167304359343960603343170658488116440998347
6684441998170083079481020253769836865387591260387081397004
4395397342728487283626639303583613156999614503895003382899
3710275557723330463738359457597728824912553479002609510283
8876667217608828542941439658998944011413943276015695324890
7732348479284448531263506286619985710653992842738259074138
7820229684450437162379033859897869490832422027208754929968
4847173162322470343065 7/7555721707723979449648392625272648
5551649317661664229348905711734284588432155133188346299073
5221751911561329826177804181498740064550632899687915531346
8554933480730789681235675239145730235553225685034903270487
8494636528771206171730831540313524856910300210308020618231
5299185952314636972146502157415893978546131789429918177417
2751814809171189000327511471361082396689939198281075027256...
```

PVSio Capabilities

1. A predefined set of PVS functions for input/output operations, side-effects, unbounded-loops, exceptions, string manipulations, and floating point arithmetic
2. A high level interface for extending PVS programming language features.
3. A tool for rapid prototyping.
4. An efficient strategy for evaluating ground expressions.

Contents

- Basic output: print & println

```
print(s:string):     void
print(r:real):       void
print(b:bool):       void

println(s:string):   void
println(r:real):     void
println(b:bool):     void
```

void

- just a rename for bool
- used to mark possible occurence of side effects

PVS source: [prelude] stdio & [prelude] stdlang

- Basic output: print & println

```
print(s:string):     void
print(r:real):       void
print(b:bool):       void

println(s:string):   void
println(r:real):     void
println(b:bool):     void
```

void

- just a rename for bool
- used to mark possible occurence of side effects

PVS source: [prelude] stdio & [prelude] stdlang

```
<PVSio> print("sqrt_newton of 2: " + sqrt_newton(2,10));
sqrt_newton of 2: 1.4142135

<PVSio> print(sq(sqrt_newton(2,10)));
2.0

<PVSio> sq(sqrt_newton(2,10)) = 2.0;
==>
FALSE

<PVSio> sq(sqrt_newton(2,10));
==>
11417786104914273667156938719275144887173049593916168
15411564699377795096902514784133369...
```

```
<PVSio> print("sqrt_newton of 2: " + sqrt_newton(2,10));
sqrt_newton of 2: 1.4142135

<PVSio> print(sq(sqrt_newton(2,10)));
2.0

<PVSio> sq(sqrt_newton(2,10)) = 2.0;
==>
FALSE

<PVSio> sq(sqrt_newton(2,10));
==>
1141778610491427366715693871927514488717304959391616 8
154115646993777950969025147841 3369...
```

PVSio Library: Input/Output Operations

Output to screen – Example

```
<PVSio> print("sqrt_newton of 2: " + sqrt_newton(2,10));
sqrt_newton of 2: 1.4142135

<PVSio> print(sq(sqrt_newton(2,10)));
2.0

<PVSio> sq(sqrt_newton(2,10)) = 2.0;
==>
FALSE

<PVSio> sq(sqrt_newton(2,10));
==>
114177861049142736671569387192751448871730495939168
15411564699377795096902514784l3369...
```

```
<PVSio> print("sqrt_newton of 2: " + sqrt_newton(2,10));
sqrt_newton of 2: 1.4142135

<PVSio> print(sq(sqrt_newton(2,10)));
2.0

<PVSio> sq(sqrt_newton(2,10)) = 2.0;
==>
FALSE

<PVSio> sq(sqrt_newton(2,10));
==>
11417786104914273667156938719275144887173049593916168
15411564699377795096902514784133369...
```

```
<PVSio> print("sqrt_newton of 2: " + sqrt_newton(2,10));
sqrt_newton of 2: 1.4142135

<PVSio> print(sq(sqrt_newton(2,10)));
2.0

<PVSio> sq(sqrt_newton(2,10)) = 2.0;
==>
FALSE

<PVSio> sq(sqrt_newton(2,10));
==>
1141778610491427366715693871927514488717304959391616 8
1541156469937779509690251478413369...
```

PVSio Library: Input/Output Operations

Output to screen – Example

```
<PVSio> print("sqrt_newton of 2: " + sqrt_newton(2,10));
sqrt_newton of 2: 1.4142135

<PVSio> print(sq(sqrt_newton(2,10)));
2.0

<PVSio> sq(sqrt_newton(2,10)) = 2.0;
==>
FALSE

<PVSio> sq(sqrt_newton(2,10));
==>
11417786104914273667156938719275144887173049593916168
1541156469937779509690251478413369...
```

```
<PVSio> print("sqrt_newton of 2: " + sqrt_newton(2,10));
sqrt_newton of 2: 1.4142135

<PVSio> print(sq(sqrt_newton(2,10)));
2.0

<PVSio> sq(sqrt_newton(2,10)) = 2.0;
==>
FALSE

<PVSio> sq(sqrt_newton(2,10));
==>
11417786104914273667156938719275144887173049593916168
1541156469937779509690251478413369...
```

```
<PVSio> print("sqrt_newton of 2: " + sqrt_newton(2,10));
sqrt_newton of 2: 1.4142135

<PVSio> print(sq(sqrt_newton(2,10)));
2.0

<PVSio> sq(sqrt_newton(2,10)) = 2.0;
==>
FALSE

<PVSio> sq(sqrt_newton(2,10));
==>
11417786104914273667156938719275144887173049593916168
1541156469937779509690251478413369...
```

```
format(s:string,t:T):string
```

- Similar to Lisp's format function ≈ `(format nil s t)`
- `s` is the control string
  - a *program* in a syntax-based language
  - optimized for compactness rather than easy comprehension
- `t` are the values to print
  - given as a single value or as a tuple of values
- Additionally, the function

  ```
  printf(s:string,t:T):void
  ```

  prints the result of `format(s,t)`

```
format(s:string,t:T):string
```

- Similar to Lisp's format function ≈ `(format nil s t)`
- `s` is the control string
  - a *program* in a syntax-based language
  - optimized for compactness rather than easy comprehension
- `t` are the values to print
  - given as a single value or as a tuple of values
- Additionally, the function

  ```
  printf(s:string,t:T):void
  ```

  prints the result of `format(s,t)`

```
format(s:string,t:T):string
```

- Similar to Lisp's format function ≈ `(format nil s t)`
- `s` is the control string
  - a *program* in a syntax-based language
  - optimized for compactness rather than easy comprehension
- `t` are the values to print
  - given as a single value or as a tuple of values
- Additionally, the function

  ```
  printf(s:string,t:T):void
  ```

  prints the result of `format(s,t)`

```
format(s:string,t:T):string
```

- Similar to Lisp's format function ≈ `(format nil s t)`
- `s` is the control string
  - a *program* in a syntax-based language
  - optimized for compactness rather than easy comprehension
- `t` are the values to print
  - given as a single value or as a tuple of values
- Additionally, the function

  ```
  printf(s:string,t:T):void
  ```

  prints the result of `format(s,t)`

```
<PVSio> format("The half of four is ~a.",4/2);
==>
"The half of four is 2."

<PVSio> format("The half of ~a is ~a.",("four",4/2));
==>
"The half of four is 2."

<PVSio> format("sqrt_newton of 2: ~a",sqrt_newton(2,10));
==>
"sqrt_newton of 2: 10685404112580054249577309962027702517
530617008867600505092775584086034866316307624567599571273
090520553619648095761832386318805390738103277561823284281
325003132706371396517146582357529867417615905908665879066
853985666554028115870511326582300341866167304359343960...
```

```
<PVSio> format("The half of four is ~a.",4/2);
==>
"The half of four is 2."

<PVSio> format("The half of ~a is ~a.",("four",4/2));
==>
"The half of four is 2."

<PVSio> format("sqrt_newton of 2: ~a",sqrt_newton(2,10));
==>
"sqrt_newton of 2: 10685404112580054249577309962027702517
5306170088676005050927755840860348663163076245675995712730
0905205536196480957618323863188053907381032775618232842813
2500313270637139651714658235752986741761590590866587906685
398566655402811587051132658230034186616730435934396 0...
```

```
<PVSio> format("The half of four is ~a.",4/2);
==>
"The half of four is 2."

<PVSio> format("The half of ~a is ~a.",("four",4/2));
==>
"The half of four is 2."

<PVSio> format("sqrt_newton of 2: ~a",sqrt_newton(2,10));
==>
"sqrt_newton of 2: 10685404112580054249577309962027702517
530617008867600505092775584086034866316307624567599571273
090520553619648095761832386318805390738103277561823284281
325003132706371396517146582357529867417615905908665879066
853985666554028115870511326582300341866167304359343960...
```

```
<PVSio> format("The half of four is ~a.",4/2);
==>
"The half of four is 2."

<PVSio> format("The half of ~a is ~a.",("four",4/2));
==>
"The half of four is 2."

<PVSio> format("sqrt_newton of 2: ~a",sqrt_newton(2,10));
==>
"sqrt_newton of 2: 10685404112580054249577309962027702517
5306170088676005050927755840860348663163076245675995712 73
0905205536196480957618323863188053907381032775618232842 81
3250031327063713965171465823575298674176159059086658790 66
853985666554028115870511326582300341866167304359343960...
```

```
<PVSio> format("The half of four is ~a.",4/2);
==>
"The half of four is 2."

<PVSio> format("The half of ~a is ~a.",("four",4/2));
==>
"The half of four is 2."

<PVSio> format("sqrt_newton of 2: ~a",sqrt_newton(2,10));
==>
"sqrt_newton of 2: 106854041125800542495773099620277702517
530617008867600505092775584086034866316307624567599571273
090520553619648095761832386318805390738103277561823284281
3250031327063713965171465823575298674176159059086658790066
8539856665540281158705113265823003418661673043593439 60...
```

```
<PVSio> format("The half of four is ~a.",4/2);
==>
"The half of four is 2."

<PVSio> format("The half of ~a is ~a.",("four",4/2));
==>
"The half of four is 2."

<PVSio> format("sqrt_newton of 2: ~a",sqrt_newton(2,10));
==>
"sqrt_newton of 2: 10685404112580054249577309962027702517
5306170088676005050927755840860348663163076245675995712 73
0905205536196480957618323863188053907381032775618232842 81
3250031327063713965171465823575298674176159059086658790 66
8539856665540281158705113265823003418661673043593439 60...
```

- Basic directives

  `~%` new line, `~&` *fresh* line, `~~` a tilde (no data consumption)

  `~a` outputs next data in human-readable form

  `~d ~x ~o ~b` allow to format integer values

  `~r ~:r` print numbers as English words

  `~@r ~:@r` print numbers as Roman numerals

- Conditional Formatting
  - `~[` ··· `~;` ··· `~]` uses next datum to index such list

```
format("~[zero~;um~;dois~]", 0) → "zero"
format("~[zero~;um~;dois~]", 1) → "um"
format("~[zero~;um~;dois~]", 2) → "dois"
```

- PVSio provides the outfix operator {| |} to use format directives on PVS lists and PVS boolean values
- Lists

```
<PVSio> LET numbers = (:1,2,3:) IN
        format("~{~a~^, ~}",{|numbers|});
==>
"1, 2, 3"
```

- Boolean values

```
<PVSio> LET b = true IN
        format("~:[falso~;verdade~]",{|b|});
==>
"verdade"
```

- PVSio provides the outfix operator {| |} to use format directives on PVS lists and PVS boolean values
- Lists

```
<PVSio> LET numbers = (:1,2,3:) IN
        format("~{~a~^, ~}",{|numbers|});
==>
"1, 2, 3"
```

- Boolean values

```
<PVSio> LET b = true IN
        format("~:[falso~;verdade~]",{|b|});
==>
"verdade"
```

- PVSio provides the outfix operator {| |} to use format directives on PVS lists and PVS boolean values
- Lists

```
<PVSio> LET numbers = (:1,2,3:) IN
        format("~{~a~^, ~}",{|numbers|});
==>
"1, 2, 3"
```

- Boolean values

```
<PVSio> LET b = true IN
        format("~:[falso~;verdade~]",{|b|});
==>
"verdade"
```

- PVSio supports a special directive to print numbers in decimal form
- `~-n/pvs:d/`
  - where n is the amount of fractional digits
- Example

```
<PVSio> printf("~-70/pvs:d/~%",1/3);
0.3333333333333333333333333333333333333333333333333333333333333333333333
```

- PVSio supports a special directive to print numbers in decimal form
- `~-`n`/pvs:d/`
  - where n is the amount of fractional digits
- Example

```
<PVSio> printf("~-70/pvs:d/~%",1/3);
0.3333333333333333333333333333333333333333333333333333333333333333333333
```

- PVSio supports a special directive to print numbers in decimal form
- `~-`n`/pvs:d/`
  - where n is the amount of fractional digits
- Example

```
<PVSio> printf("~-70/pvs:d/~%",1/3);
0.333333333333333333333333333333333333333333333333333333333333333333333333
```

no prompt	prompt	Description
`read_real`	`query_real(msg)`	Reads a real number
`read_int`	`query_int(msg)`	Reads an integer
`read_word`	`query_word(msg)`	Reads a word
`read_bool(ans)`	`query_bool(msg,ans)`	Checks if the entered word is equal to *ans*
`read_line`	`query_line(msg)`	Reads the whole line
`read_token(s)`	`query_token(msg,s)`	Returns the smaller prefix that ends in any of the chars in *s*

```
msg,s,ans:  VAR string
```

PVSio Library: Input/Output Operations
Input from *stdin* - Examples

```
<PVSio> let x = read_real in
        print("sqrt("+x+") = "+sqrt_newton(x,10));
10
sqrt(10) = 3.1622777

<PVSio> let x = query_real("Enter a real number:") in
        print("sqrt("+x+") = "+sqrt_newton(x,10));
Enter a real number:
10
sqrt(10) = 3.1622777
```

PVSio Library: Input/Output Operations

Input from *stdin* - Examples

```
<PVSio> let x = read_real in
        print("sqrt("+x+") = "+sqrt_newton(x,10));
10
sqrt(10) = 3.1622777

<PVSio> let x = query_real("Enter a real number:") in
        print("sqrt("+x+") = "+sqrt_newton(x,10));
Enter a real number:
10
sqrt(10) = 3.1622777
```

```
<PVSio> let x = read_real in
        print("sqrt("+x+") = "+sqrt_newton(x,10));
10
sqrt(10) = 3.1622777

<PVSio> let x = query_real("Enter a real number:") in
        print("sqrt("+x+") = "+sqrt_newton(x,10));
Enter a real number:
10
sqrt(10) = 3.1622777
```

```
<PVSio> let x = read_real in
        print("sqrt("+x+") = "+sqrt_newton(x,10));
10
sqrt(10) = 3.1622777
```

```
<PVSio> let x = query_real("Enter a real number:") in
        print("sqrt("+x+") = "+sqrt_newton(x,10));
Enter a real number:
10
sqrt(10) = 3.1622777
```

```
<PVSio> let x = read_real in
        print("sqrt("+x+") = "+sqrt_newton(x,10));
10
sqrt(10) = 3.1622777

<PVSio> let x = query_real("Enter a real number:") in
        print("sqrt("+x+") = "+sqrt_newton(x,10));
Enter a real number:
10
sqrt(10) = 3.1622777
```

```
<PVSio> let x = read_real in
        print("sqrt("+x+") = "+sqrt_newton(x,10));
10
sqrt(10) = 3.1622777

<PVSio> let x = query_real("Enter a real number:") in
        print("sqrt("+x+") = "+sqrt_newton(x,10));
Enter a real number:
10
sqrt(10) = 3.1622777
```

PVSio Library: Input/Output Operations

Input from *stdin* - Examples

```
<PVSio> let x = read_real in
          print("sqrt("+x+") = "+sqrt_newton(x,10));
10
sqrt(10) = 3.1622777

<PVSio> let x = query_real("Enter a real number:") in
          print("sqrt("+x+") = "+sqrt_newton(x,10));
Enter a real number:
10
sqrt(10) = 3.1622777
```

```
<PVSio> let x = read_real in
        print("sqrt("+x+") = "+sqrt_newton(x,10));
10
sqrt(10) = 3.1622777

<PVSio> let x = query_real("Enter a real number:") in
        print("sqrt("+x+") = "+sqrt_newton(x,10));
Enter a real number:
10
sqrt(10) = 3.1622777
```

- As usual, PVSio Streams have *kind & direction*
  - Standard, File, String
  - Input, Ouput
- Ad-Hoc Datatypes & Constants

```
Stream :   TYPE+                 stdin :   IStream
IStream:   TYPE+ FROM Stream     stdout :  OStream
OStream:   TYPE+ FROM Stream     stderr :  OStream
```

- Functions, being `f:VAR Stream`

`fopen?(f):bool`	Checks if the stream is *open*

`strstream?(f):bool`
`filestream?(f):bool`
`sdtstream?(f):bool`
	Check the kind of stream

`finput?(f):bool` `fouput?(f):bool`	Check the direction of the stream

- Kind and direction are represented by the enumerated type `Mode`
  - `Mode : TYPE = {input,output,create,append,overwrite,rename,str}`
- More Functions

`fopenin(m:Mode,s:string)`	: `IStream`	Opens an input stream in mode `m`
`fopenin(s:string)`	: `string`	Opens an input stream from file `s`
`fopenout(m:Mode,s:string)`	: `OStream`	Opens an output stream in mode `m`
`eof?(f:IStream)`	: `bool`	Checks if the stream has been completely consumed
`flength(f:Stream)`	: `nat`	Returns the length of the stream

- Read

`fread_line(f:IStream)`	: `string`	Reads a line from f
`fread_word(f:IStream)`	: `string`	Reads a word from f
`fread_real(f:IStream)`	: `rat`	Reads a real number from f
`fread_int(f:IStream)`	: `int`	Reads an integer from f
`fread_bool(f:IStream,answer:string)`	: `bool`	Reads a boolean from f

- Write

`fprint(f:OStream,s:string)`	: `void`	Writes the string *s* to the stream *f*
`fprint(f:OStream,r:real)`	: `void`	Writes the real number *r* to the stream *f*
`fprint(f:OStream,b:bool)`	: `void`	Writes the boolean value *b* to the stream *f*
`fprintln(f:OStream,s:string)`	: `void`	Writes the string *s* on a new line in *f*
`fprintln(f:OStream,r:real)`	: `void`	Writes the real number *r* on a new line in *f*
`fprintln(f:OStream,b:bool)`	: `void`	Writes the boolean value *b* on a new line in *f*

If the content of the file "dez.txt" are

line 1 10

```
<PVSio> let f = fopenin("dez.txt"),
            x = fread_int(f)
        in print("sqrt("+x+") = "+sqrt_newton(x,10))
           & fclose(f);
sqrt(10) = 3.1622777
```

If the content of the file "dez.txt" are

line 1 10

```
<PVSio> let f = fopenin("dez.txt"),
            x = fread_int(f)
        in print("sqrt("+x+") = "+sqrt_newton(x,10))
           & fclose(f);
sqrt(10) = 3.1622777
```

If the content of the file "dez.txt" are

line 1 10

```
<PVSio> let fin   = fopenin("dez.txt"),
            fout  = fopenout(create,"sqdez.txt"),
            x     = fread_int(fin)
        in fprintln(fout,"sqrt("+x+") = "+sqrt_newton(x,10))
           & fclose(fout)
           & fclose(fin)
           & print("file saved.")
file saved.
```

The contents of the file "sqdez.txt" will be

line 1 3.1622777

- Even more functions

`fcheck(f:IStream)`	: `bool`	Checks if the stream is open and did not reach *eof*
`fname(f:Stream)`	: `string`	Returns the full name of the file stream `f`
`fgetpos(f:Stream,n:nat)`	: `nat`	Returns current position of the file stream `f`
`fsetpos(f:Stream,n:nat)`	: `void`	Set current position of file stream `f`
`echo(f:OStream,s:string)`	: `void`	Prints `f` to `s` and echoes to stdout
`echo(f:OStream,r:real)`	: `void`	Prints `r` to `s` and echoes to stdout
`echo(f:OStream,b:bool)`	: `void`	Prints `b` to `s` and echoes to stdout
`echoln(f:OStream,s:string)`	: `void`	Prints `f` to `s` in a new line and echoes to stdout
`echoln(f:OStream,r:real)`	: `void`	Prints `r` to `s` in a new line and echoes to stdout
`echoln(f:OStream,b:bool)`	: `void`	Prints `b` to `s` in a new line and echoes to stdout

- Bounded loops
  - for i = n to m do <statement>
  - Support for proofs of correctness
- Unbounded loops
  - Pragmatic approach
  - while(true) do <statement>

- Functional version ($m \leq n$)

$$f(n, f(...f(m+1, f(m, a))...))$$

- Imperative version

```
local a : T := init;
local i : int;
for (i := m; i <= n; i++) {
  a := f(i,a);
}
return a;
```

- PVS implementation

```
for[T:TYPE](m,n:int,init:T,f:[subrange(m,n),T]->T) : T
```

```
%% a = 1;
%% for (i=1; i <= n; i++) {
%%    a = a*x;
%% }

expit(x:real,n:nat): real =
  for[real](1,n,1,LAMBDA(i:subrange(1,n),a:real):a*x)

<PVSio> expit(2,10);
==>
1024
```

- Functional version $(m \leq n)$

$$f(m, f(...f(n-1, f(n,a))...))$$

- Imperative version

```
local a : T := init;
local i : int;
for (i := n; i >= m; i--) {
  a := f(i,a);
}
return a;
```

- PVS implementation

```
for_down[T:TYPE](n,m:int,init:T,f:[subrange(m,n),T]->T) : T
```

```
%% a = 1;
%% for (i=n; i >= 1; i-) {
%%   a = a*i;
%% }

factit(n:nat) : nat =
  for_down[nat](n,1,1,LAMBDA(i:subrange(1,n),a:nat):a*i)

<PVSio> factit(10);
==>
3628800
```

- Functional version $(m \leq n)$

$$(\cdots((f(m) \circ f(m+1)) \circ f(m+2)) \circ \cdots f(n))$$

- Imperative version

```
local a : T = f(m);
local i : int;
for (i := m+1; i <= n; i++) {
  a := a o f(i)
}
return a;
```

- PVS implementation

```
iterate_left[T:TYPE](m,n:int,f:subrange(upfrom,upto)->T,o:[[T,T]->T]) : T
```

```
%% a = nth(l,0);
%% for (i=1;i<=length(l)-1;i++) {
%%   a = max(a,nth(l,i))
%% }

maxit(l:(cons?[real])) : real =
  iterate_left(0,length(l)-1,
               LAMBDA(i:below(length(l))):nth(l,i),max)

<PVSio> maxit((:2,3,4,1,2:));
==>
4
```

- Functional version ($m \leq n$)

$$f(m) \circ (\cdots (f(n-2) \circ (f(n-1) \circ f(n))) \cdots)$$

- Imperative version

```
local a : T = f(n);
local i : int;
for (i := n-1; i >= m; i-) {
  a := f(i) o a
}
return a;
```

- PVS implementation

```
iterate_right[T:TYPE](m,n:int,f:subrange(upfrom,upto)->T,o:[[T,T]->T]) : T
```

```
%% a = nth(l,0);
%% for (i=1;i<=length(l)-1;i++) {
%%   a = min(nth(l,i),a)
%% }

minit(l:(cons?[real])) : real =
  iterate_right( 0,
                 length(l)-1,
                 LAMBDA(i:below(length(l))):nth(l,i),
                 min )

<PVSio> minit((:2,3,4,1,2:));
==>
1
```

- Previous definitions are not suitable for unbounded calculations

```
while(b:bool,s:void) : void
```

- Example: reads a file one line at a time
  - As in the *cat* unix command

```
cat : void =
  let f=fopenin("pvsio_examples.pvs") in
  while(not eof?(f),println(fread_line(f)))
  & fclose(f)
```

- PVSio also provides support for *exception handling*
- Mechanism to respond to the occurrence of exceptional events
  - often changing the normal flow of program execution
- Usually used in input/output operations

PVS source: [prelude] stdexc

```
int_aux : int =
  let i = query_int("Enter a number less than 10") in
  if i > 10 then throw("GreaterThan10")
  else i endif
```

```
readupto10 : int =
  catch[int]((:NotAnInteger,"GreaterThan10":),
       int_aux,0)
```

```
<PVSio> readupto10;
Enter a number less than 10
15
==>
0
```

```
int_aux : int =
  let i = query_int("Enter a number less than 10") in
  if i > 10 then throw("GreaterThan10")
  else i endif

readupto10 : int =
  catch[int]((:NotAnInteger,"GreaterThan10":),
       int_aux,0)
```

```
<PVSio> readupto10;
Enter a number less than 10
15
==>
0
```

```
int_aux : int =
  let i = query_int("Enter a number less than 10") in
  if i > 10 then throw("GreaterThan10")
  else i endif

readupto10 : int =
  catch[int]((:NotAnInteger,"GreaterThan10":),
       int_aux,0)
```

```
<PVSio> readupto10;
Enter a number less than 10
15
==>
0
```

```
int_aux : int =
  let i = query_int("Enter a number less than 10") in
  if i > 10 then throw("GreaterThan10")
  else i endif

readupto10 : int =
  catch[int]((:NotAnInteger,"GreaterThan10":),
       int_aux,0)
```

```
<PVSio> readupto10;
Enter a number less than 10
15
==>
0
```

```
int_aux : int =
  let i = query_int("Enter a number less than 10") in
  if i > 10 then throw("GreaterThan10")
  else i endif

readupto10 : int =
  catch[int]((:NotAnInteger,"GreaterThan10":),
       int_aux,0)
```

```
<PVSio> readupto10;
Enter a number less than 10
15
==>
0
```

```
int_aux : int =
  let i = query_int("Enter a number less than 10") in
  if i > 10 then throw("GreaterThan10")
  else i endif

readupto10 : int =
  catch[int]((:NotAnInteger,"GreaterThan10":),
       int_aux,0)
```

```
<PVSio> readupto10;
Enter a number less than 10
15
==>
0
```

- Throw

  `throw[T:TYPE](tag:ExceptionTag): T`

  where `ExceptionTag : TYPE = string`

```
int_aux : int =
  let i = query_int("Enter a number less than 10") in
  if i > 10 then throw("GreaterThan10")
  else i endif
```

- Catch

  `catch[T:TYPE](tag:ExceptionTag,program,valueOnException:T): T`

  `catch[T:TYPE](tags:list[ExceptionTag],program,valOnExcep:T): T`

```
readupto10 : int =
  catch[int]((:NotAnInteger,"GreaterThan10":),
             int_aux,0)
```

PVS source: [prelude] stdprog + [prelude] stdexc

Imperative-like variables

- `Mutable : TYPE+`
- `ref(value:T) : Mutable`
  - defines a local variable storing the value `value`
- `val(var:Mutable): T`
  - returns the value stored in the variable `var`
  - if `var` stores no value, `UndefinedMutableVariable` is thrown
- `undef(var:Mutable) : bool`
  - indicates if the variable `var` stores any value or not

PVS source: [prelude] stdprog

```
woow(x:int) : void =
    let lvar = ref[int](x) in
    println("The value of lvar is: "+val(lvar)) &
    set[int](lvar,x+1) &
    print("The new value of lvar is: "+val(lvar))
```

```
<PVSio> woow(23);
The value of lvar is: 23
The new value of lvar is: 24
```

PVS source: [NASAlib] examples@pvsio_examples

```
woow(x:int) : void =
    let lvar = ref[int](x) in
    println("The value of lvar is: "+val(lvar)) &
    set[int](lvar,x+1) &
    print("The new value of lvar is: "+val(lvar))
```

```
<PVSio> woow(23);
The value of lvar is: 23
The new value of lvar is: 24
```

PVS source: [NASAlib] examples@pvsio_examples

```
woow(x:int) : void =
    let lvar = ref[int](x) in
    println("The value of lvar is: "+val(lvar)) &
    set[int](lvar,x+1) &
    print("The new value of lvar is: "+val(lvar))
```

```
<PVSio> woow(23);
The value of lvar is: 23
The new value of lvar is: 24
```

PVS source: [NASAlib] examples@pvsio_examples

- Provided as PVS constants of type `Global`

  `Global[T:TYPE+,initial_value:T]: TYPE+ = Mutable[T]`

- Example

```
gvar : Global[int,0]

WOOW(x:int) : void =
  println("The original of gvar is: "+val(gvar)) &
  set(gvar,x) &
  print("The value of gvar is: "+val(gvar))
```

```
<PVSio> WOOW(23);
The original of gvar is: 0
The value of gvar is: 23
```

PVS source: [prelude] stdglobal

- PVS parsing features are accesible through the function str2pvs

```
str2pvs[T:TYPE+](s:string):T
```

- Example

```
Point : TYPE = [# x : real, y: real #]
zero : Point = str2pvs("(# x := 0, y:= 0 #)")
```

```
<PVSio> zero;
==>
(# x := 0, y := 0 #)
```

PVS source: [prelude] stdpvs & [NASAlib] examples@pvsio_examples

- PVS parsing features are accesible through the function `str2pvs`

```
str2pvs[T:TYPE+](s:string):T
```

- Example

```
Point : TYPE = [# x : real, y: real #]
zero : Point = str2pvs("(# x := 0, y:= 0 #)")
```

```
<PVSio> zero;
==>
(# x := 0, y := 0 #)
```

PVS source: [prelude] stdpvs & [NASAlib] examples@pvsio_examples

- PVS parsing features are accesible through the function `str2pvs`

```
str2pvs[T:TYPE+](s:string):T
```

- Example

```
Point : TYPE = [# x : real, y: real #]
zero : Point = str2pvs("(# x := 0, y:= 0 #)")
```

```
<PVSio> zero;
==>
(# x := 0, y := 0 #)
```

PVS source: [prelude] stdpvs & [NASAlib] examples@pvsio_examples

- `pvs2str` returns a string representation of a PVS element
  `pvs2str[T:TYPE+](t:T) : string`
- Example

```
<PVSio> print((:1,2,3:));
first argument to print has the wrong type
     Found: (list_adt[real].cons?)
  Expected: booleans.bool
Try again.

<PVSio> pvs2str((:1,2,3:));
==>
"(: 1, 2, 3 :)"
```

PVS source: [prelude] stdpvs & [NASAlib] examples@pvsio_examples

- `pvs2str` returns a string representation of a PVS element

```
pvs2str[T:TYPE+](t:T) : string
```

- Example

```
<PVSio> print((:1,2,3:));
first argument to print has the wrong type
     Found: (list_adt[real].cons?)
  Expected: booleans.bool
Try again.

<PVSio> pvs2str((:1,2,3:));
==>
"(: 1, 2, 3 :)"
```

PVS source: [prelude] stdpvs & [NASAlib] examples@pvsio_examples

- pvs2str returns a string representation of a PVS element

  `pvs2str[T:TYPE+](t:T) : string`

- Example

```
<PVSio> print((:1,2,3:));
first argument to print has the wrong type
     Found: (list_adt[real].cons?)
  Expected: booleans.bool
Try again.

<PVSio> pvs2str((:1,2,3:));
==>
"(: 1, 2, 3 :)"
```

PVS source: [prelude] stdpvs & [NASAlib] examples@pvsio_examples

- `pvs2str` returns a string representation of a PVS element
  `pvs2str[T:TYPE+](t:T) : string`
- Example

```
<PVSio> print((:1,2,3:));
first argument to print has the wrong type
     Found: (list_adt[real].cons?)
  Expected: booleans.bool
Try again.

<PVSio> pvs2str((:1,2,3:));
==>
"(: 1, 2, 3 :)"
```

PVS source: [prelude] stdpvs & [NASAlib] examples@pvsio_examples

- PVSio provides a "user-friendly" mechanism for extending the ground evaluator.
- Semantic attachements: Lisp functions attached to uninterpreted PVS functions.

- Every uninterpreted function symbol f_i in a PVS theory Th
- Can be *semantically attached* to Lisp code
  - using the macro `defattach`
  - the name *must be* $|Th.f_i|$
  - as many parameters as the PVS function
  - in a file named "pvs-attachments"
  - located in the context directory
- PVSio executes the attachment code when the symbol is evaluated

```
Th : THEORY
BEGIN
 ...
 f_i(p_0:T_0,···,p_n:T_n) : T
 ...
END Th
```

```
(defattach |Th.f_i| (p'_0 ··· p'_n)
 ⟨Documentation string⟩
 ⟨Lisp code⟩)
```

```
cubic_root : THEORY
BEGIN
 ...
 cubic(x:real) : real
 ...
END cubic_root
```

Create the file pvs-attachments in context directory:

```
;; File: pvs-attachments
(defattach cubic_root.cubic (x)
   "Cubic root of x"
   (expt x (/ 1 3))) ;;; <==== THIS IS LISP
```

In PVSio:

```
<PVSio> cubic(10);
==>
2.1544347
```

```
cubic_root : THEORY
BEGIN
 ...
 cubic(x:real) : real
 ...
END cubic_root
```

Create the file pvs-attachments in context directory:

```
;; File: pvs-attachments
(defattach cubic_root.cubic (x)
   "Cubic root of x"
   (expt x (/ 1 3))) ;;; <==== THIS IS LISP
```

In PVSio:

```
<PVSio> cubic(10);
==>
2.1544347
```

```
cubic_root : THEORY
BEGIN
 ...
 cubic(x:real) : real
 ...
END cubic_root
```

Create the file pvs-attachments in context directory:

```
;; File: pvs-attachments
(defattach cubic_root.cubic (x)
   "Cubic root of x"
   (expt x (/ 1 3))) ;;; <==== THIS IS LISP
```

In PVSio:

```
<PVSio> cubic(10);
==>
2.1544347
```

```
cubic_root : THEORY
BEGIN
 ...
 cubic(x:real) : real
 ...
END cubic_root
```

Create the file pvs-attachments in context directory:

```
;; File: pvs-attachments
(defattach cubic_root.cubic (x)
   "Cubic root of x"
   (expt x (/ 1 3))) ;;; <==== THIS IS LISP
```

In PVSio:

```
<PVSio> cubic(10);
==>
2.1544347
```

- Name of the attachment and number of parameters
  - Given by the PVS definition of the function
- Data types
  - Parameters and return value
  - Only basic types have an automatic translation to Lisp
    - string $\leftrightarrow$ string (`simple-array character`)
    - nat, int, bool $\rightarrow$ immediate fixnum
    - bool $\leftarrow$ bool

- PVS theories and attachments do not share namespaces
- PVS global variables can be accessed through ad-hoc macros
  - `(pvsio_get_gvar_by_name` ⟨*var name*⟩`)`
  - `(pvsio_set_gvar_by_name` ⟨*var name*⟩ ⟨*value*⟩`)`
- For more general cases, PVSio provides macro "`using`"
  - It allows to refer PVS declarations in attachments
  - Similar in structure to Lisp's `let` macro

```
(using
 ((⟨name_0⟩ "⟨pvs decl name⟩")
  ...
  (⟨name_n⟩ "⟨pvs decl name⟩"))
 ⟨body⟩)
```

  - to use $name_i$ in *body*, Lisp's function `funcall` must be used

- PVS theories and attachments do not share namespaces
- PVS global variables can be accessed through ad-hoc macros
  - (pvsio_get_gvar_by_name ⟨*var name*⟩)
  - (pvsio_set_gvar_by_name ⟨*var name*⟩ ⟨*value*⟩)
- For more general cases, PVSio provides macro "using"
  - It allows to refer PVS declarations in attachments
  - Similar in structure to Lisp's `let` macro

```
(using
 ((⟨name_0⟩ "⟨pvs decl name⟩")
  ...
  (⟨name_n⟩ "⟨pvs decl name⟩"))
 ⟨body⟩)
```

  - to use $name_i$ in *body*, Lisp's function `funcall` must be used

- PVS theories and attachments do not share namespaces
- PVS global variables can be accessed through ad-hoc macros
  - `(pvsio_get_gvar_by_name` ⟨*var name*⟩`)`
  - `(pvsio_set_gvar_by_name` ⟨*var name*⟩ ⟨*value*⟩`)`
- For more general cases, PVSio provides macro "`using`"
  - It allows to refer PVS declarations in attachments
  - Similar in structure to Lisp's `let` macro

```
(using
 ((⟨name_0⟩ "⟨pvs decl name⟩")
  ...
  (⟨name_n⟩ "⟨pvs decl name⟩"))
 ⟨body⟩)
```

  - to use $name_i$ in *body*, Lisp's function `funcall` must be used

- PVS theory att_test

```
ct0: real = 13
add_fun(x,y: nat): nat = x + y
addtoct0(x: nat): nat
```

- In pvs-attachment file:

```
(defattach |att_test.addtoct0| (x)
  "Example of 'using' macro"
  (using
   ((ct "ct0")
    (add "add_fun"))
   (funcall add (funcall ct) x)))
```

- In PVSio

```
<PVSio> addtoct0(3);
  ==>
  16
```

- PVS theory att_test

```
ct0: real = 13
add_fun(x,y: nat): nat = x + y
addtoct0(x: nat): nat
```

- In pvs-attachment file:

```
(defattach |att_test.addtoct0| (x)
  "Example of 'using' macro"
  (using
   ((ct "ct0")
    (add "add_fun"))
   (funcall add (funcall ct) x)))
```

- In PVSio

```
<PVSio> addtoct0(3);
  ==>
  16
```

- PVS theory `att_test`

```
ct0: real = 13
add_fun(x,y: nat): nat = x + y
addtoct0(x: nat): nat
```

- In `pvs-attachment` file:

```
(defattach |att_test.addtoct0| (x)
  "Example of 'using' macro"
  (using
   ((ct "ct0")
    (add "add_fun"))
   (funcall add (funcall ct) x)))
```

- In `PVSio`

```
<PVSio> addtoct0(3);
  ==>
  16
```

- Trigonometric constants and operations are defined in `NASAlib/trig`
  - pi, sin, cos, tan, atan, asin, acos
  - Example

```
<PVSio> printf("~-70/pvs:d/~%",pi_def.pi);
3.141592653589793115997963468544185161590576171875
```

(48 digits)

- By default, they are attached to Lisp's implementations
- `NASAlib/fast_approx` provides more accurate implementations
  - Example
    - Adding `IMPORTING fast_approx@top` in the PVS theory

```
<PVSio> printf("~-70/pvs:d/~%",pi_def.pi);
3.1415926525148811252603186491743123303931396337610363930264040161439269
```

(70 digits)

- Trigonometric constants and operations are defined in `NASAlib/trig`
  - pi, sin, cos, tan, atan, asin, acos
  - Example

```
<PVSio> printf("~-70/pvs:d/~%",pi_def.pi);
3.141592653589793115997963468544185161590576171875
```

(48 digits)

- By default, they are attached to Lisp's implementations
- `NASAlib/fast_approx` provides more accurate implementations
  - Example
    - Adding `IMPORTING fast_approx@top` in the PVS theory

```
<PVSio> printf("~-70/pvs:d/~%",pi_def.pi);
3.1415926525148811252603186491743123303931396337610363930264040161439269
```

(70 digits)

- Trigonometric constants and operations are defined in `NASAlib/trig`
  - pi, sin, cos, tan, atan, asin, acos
  - Example

```
<PVSio> printf("~-70/pvs:d/~%",pi_def.pi);
3.141592653589793115997963468544185161590576171875
```

(48 digits)

- By default, they are attached to Lisp's implementations
- `NASAlib/fast_approx` provides more accurate implementations
  - Example
    - Adding `IMPORTING fast_approx@top` in the PVS theory

```
<PVSio> printf("~-70/pvs:d/~%",pi_def.pi);
3.1415926525148811252603186491743123303931396337610363930264040161439269
```

(70 digits)

- RANDOM is a constant defined in the prelude (stdmath theory)
- attached to a Lisp implementation of a random number generator

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

- but...

```
<PVSio> let r=RANDOM in r = r;
==>
TRUE
```

- and the following lemma is trivially true

```
thefact : LEMMA
  RANDOM = RANDOM
```

- RANDOM is a constant defined in the prelude (stdmath theory)
- attached to a Lisp implementation of a random number generator

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

- but...

```
<PVSio> let r=RANDOM in r = r;
==>
TRUE
```

- and the following lemma is trivially true

```
thefact : LEMMA
  RANDOM = RANDOM
```

- RANDOM is a constant defined in the prelude (stdmath theory)
- attached to a Lisp implementation of a random number generator

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

- but...

```
<PVSio> let r=RANDOM in r = r;
==>
TRUE
```

- and the following lemma is trivially true

```
thefact : LEMMA
  RANDOM = RANDOM
```

```
maxl_th : THEORY
BEGIN
 IMPORTING list[real]

 maxl(l:list) : RECURSIVE real =
   cases l of
     null : 0,
     cons(a,r) : max(a,maxl(r))
   endcases
   MEASURE l by <<

END maxl_th
```

```
maxl_io : THEORY
BEGIN

  IMPORTING maxl_th

  test : void =
    println("Testing the function maxl") &
    LET s = query_line("Enter a list of real numbers: "),
        l = str2pvs[list[real]](s),
        m =  maxl(l) IN
      println("The max of "+s+" is "+m)

END maxl_io
```

```
<PVSio> test;
Testing the function maxl
Enter a list of real numbers:
(: -1, -2, 5, 3, 2 :)
The max of (: -1, -2, 5, 3, 2 :) is 5

<PVSio> test;
Testing the function maxl
Enter a list of real numbers:
(: -1, -2, -3, -4 :)
The max of (: -1, -2, -3, -4 :) is 0
```

```
<PVSio> test;
Testing the function maxl
Enter a list of real numbers:
(: -1, -2, 5, 3, 2 :)
The max of (: -1, -2, 5, 3, 2 :) is 5

<PVSio> test;
Testing the function maxl
Enter a list of real numbers:
(: -1, -2, -3, -4 :)
The max of (: -1, -2, -3, -4 :) is 0
```

```
<PVSio> test;
Testing the function maxl
Enter a list of real numbers:
(: -1, -2, 5, 3, 2 :)
The max of (: -1, -2, 5, 3, 2 :) is 5

<PVSio> test;
Testing the function maxl
Enter a list of real numbers:
(: -1, -2, -3, -4 :)
The max of (: -1, -2, -3, -4 :) is 0
```

```
<PVSio> test;
Testing the function maxl
Enter a list of real numbers:
(: -1, -2, 5, 3, 2 :)
The max of (: -1, -2, 5, 3, 2 :) is 5
```

```
<PVSio> test;
Testing the function maxl
Enter a list of real numbers:
(: -1, -2, -3, -4 :)
The max of (: -1, -2, -3, -4 :) is 0
```

```
<PVSio> test;
Testing the function maxl
Enter a list of real numbers:
(: -1, -2, 5, 3, 2 :)
The max of (: -1, -2, 5, 3, 2 :) is 5

<PVSio> test;
Testing the function maxl
Enter a list of real numbers:
(: -1, -2, -3, -4 :)
The max of (: -1, -2, -3, -4 :) is 0
```

```
<PVSio> test;
Testing the function maxl
Enter a list of real numbers:
(: -1, -2, 5, 3, 2 :)
The max of (: -1, -2, 5, 3, 2 :) is 5

<PVSio> test;
Testing the function maxl
Enter a list of real numbers:
(: -1, -2, -3, -4 :)
The max of (: -1, -2, -3, -4 :) is 0
```

```
$ pvsio maxl_io:test
Testing the function maxl
Enter a list of real numbers:
(: 5, 4 ,3 ,2 :)
The max of (: 5, 4 ,3 ,2 :) is 5
```

```
$ pvsio maxl_io:test
Testing the function maxl
Enter a list of real numbers:
(: 5, 4 ,3 ,2 :)
The max of (: 5, 4 ,3 ,2 :) is 5
```

PVSio and the PVS Theorem Prover

- PVSio safely enables the ground evaluator in the theorem prover.
- Ground expressions are translated into Lisp and evaluated in the PVS Lisp engine.
- The theorem prover only trusts the Lisp code automatically generated from PVS functional specifications.
- Semantic attachments are always considered harmful for the theorem prover.

Evaluation of ground expressions via the ground evaluator:

```
  |-----
{1}   2 < sqrt_newton(2, 10) * sqrt_newton(2, 10)

Rule? (eval-formula 1)
Q.E.D.
```

Evaluation of ground expressions via the ground evaluator:

```
  |-----
{1}   2 < sqrt_newton(2, 10) * sqrt_newton(2, 10)

Rule? (eval-formula 1)
Q.E.D.
```

```
|-----
{1}   RANDOM /= RANDOM
Rule? (eval-formula 1)

Function stdmath.RANDOM is defined as a semantic
attachment.  It cannot be evaluated in a formal
proof.

No change on: (eval-formula 1)
```

```
|-----
{1}   RANDOM /= RANDOM
Rule? (eval-formula 1)

Function stdmath.RANDOM is defined as a semantic
attachment.  It cannot be evaluated in a formal
proof.

No change on: (eval-formula 1)
```

```
  |-----
{1}   2 < sqrt_newton(2, 10) * sqrt_newton(2, 10)

Rule? (grind)
sqrt_newton rewrites sqrt_newton(2, 10)
  to (1/2) * (2 / ((1/2) * (2 / (3 * ((1/2) * (1/2))
  + (1/2) * (2/(3 * (1/2) + (1/2) * (2/3)))
  + (1/2) * (1/2) * (2/3)))
  + 3 * ((1/2) * (1/2) * (1/2))))
  + ...
```

```
  |-----
{1}   2 < sqrt_newton(2, 10) * sqrt_newton(2, 10)

Rule? (grind)
sqrt_newton rewrites sqrt_newton(2, 10)
  to (1/2) * (2 / ((1/2) * (2 / (3 * ((1/2) * (1/2))
  + (1/2) * (2/(3 * (1/2) + (1/2) * (2/3)))
  + (1/2) * (1/2) * (2/3)))
  + 3 * ((1/2) * (1/2) * (1/2))))
  + ...
```

References

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