PVSio Reference Manual

Version 2.b (DRAFT) • August 2005

César A. Muñoz
munoz@nianet.org
http://research.nianet.org/~munoz/PVSio

National Institute of Aerospace (NIA)
Formal Methods Group • 100 Exploration Way • Hampton VA 23666

PVSio ∈ Packages for Verification Survival

The development of PVSio was supported by the National Aeronautics and Space Administration under NASA Cooperative Agreement NCC-1-02043.

Contents

Chapter 1

Introduction

PVS [4] is a verification system based on a typed classical higher-order logic enriched with predicate subtyping and dependent records [6]. The system is widely known by its expressive specification language and its powerful theorem prover. A considerable large subset of the PVS specification language is executable.

The *ground evaluator* is an experimental feature of PVS that enables the animation of functional specifications. The ground evaluator extracts Common Lisp code from PVS functional specifications to evaluate ground expressions on top of the PVS underlying Common Lisp machine [7].

Semantic attachments [2] are user-defined Common Lisp code that enhances the functionality of the ground evaluator. For instance, uninterpreted PVS functions can be defined in Common Lisp by writing an appropriate semantic attachment. Semantic attachments bring all the computational power and side effect features of Common Lisp to the PVS ground evaluator. However, semantic attachments are a potential source of problems in PVS. Indeed, as semantic attachments are not typechecked by PVS, they can inadvertently break the system.

PVSio [3] is a PVS package[1] that extends the ground evaluator with a predefined library of imperative programming language features such as side effects, unbounded loops, input/output operations, floating point arithmetic, exception handling, pretty printing, and parsing. The PVSio library is implemented via semantic attachments. Furthermore, PVSio provides:

- An alternative (and simplified) Emacs interface to the ground evaluator.
- A user-friendly mechanism to define new semantic attachments.
- A stand alone (Emacs-free) interface to the ground evaluator.
- A set of proof rules that safely integrates the ground evaluator to the PVS theorem prover.

[1]PVS packages are also known as prelude library extensions.

Chapter 2

Installing PVSio

To install PVSio-2.b, remove any previous version of PVSio and check that PVS 3.1 or higher is already installed.[1] For example, type

```
$ pvs-dir/pvs -version
```

where *pvs-dir* is the absolute path to the PVS root directory.

Get the tarball file `PVSio-2.b.tgz` from `http://research.nianet.org/~munoz/PVSio`. If necessary, create a directory to store PVS packages, e.g., *packages*. The directory *packages* may also be *pvs-dir*`/lib`, but note that read and write privileges are required on this directory during the installation procedure. Move the tarball file to *packages* and unpack it:

```
$ tar xfz PVSio-2.b.tgz
```

Change to the distribution directory *packages*`/PVSio-2.b` and type

```
$ make PVS_DIR=pvs-dir
```

Finally, install the package:

```
$ make install PVS_DIR=pvs-dir
```

Detailed installation instructions are provided in the distribution file `INSTALL`.

Before using PVSio, verify that the environment variable `PVS_LIBRARY_PATH` is set such that it includes the directory *packages*. For instance, define `PVS_LIBRARY_PATH` in the startup file of the user shell. In bash, add the line

```
export PVS_LIBRARY_PATH=$PVS_LIBRARY_PATH:packages
```

to the `.bash_profile` file. In csh, add the line

```
setenv PVS_LIBRARY_PATH $PVS_LIBRARY_PATH:packages
```

to the `.cshrc` file. Run `source` on the updated file or restart the login shell.

The first time that PVSio is used in a working context, the package has to be loaded with the Emacs command `M-x load-prelude-library PVSio`. Unless the file `.pvscontext` is removed from the current context, PVS will automatically load PVSio in further PVS sessions.

[1] PVS (`http://pvs.csl.sri.com`) is a software developed and licensed by SRI International.

Things to Remember

- Remove any previous version of PVSio before installing PVSio-2.b, specially if the full NASA PVS library [1] has been previously installed.
- Check that the environment variable `PVS_LIBRARY_PATH` is properly defined.
- Make sure that PVS 3.1 or higher is installed.
- Use `M-x load-prelude-library PVSio` the first time that PVSio is used in a working context.

Chapter 3

PVSio for the Impatient

Launch PVS in a new working context and load the PVSio package with the Emacs command `M-x load-prelude-library PVSio`. Create the file `quadratic.pvs` as shown in Listing 3.1. Typecheck the theory `quadratic` but, for the sake of simplicity, ignore the Type Correctness Conditions (TCCs) generated by the typechecker.[1]

3.1 Read, Eval, Print

The PVSio read-eval-print interface to the ground evaluator is available through the Emacs command `M-x pvsio`. PVSio redefines the standard PVS interface to the ground evaluator. In the new PVSio interface, ground PVS expressions are typed directly after the prompt `<PVSio>` and they are terminated by a semicolon. The result is printed after the string `==>`. For example:

```
<PVSio> root(0,3,3,true);
==>
-1

<PVSio> root(1,1,-6,true);
Hit uninterpreted term quadratic.sqrt_ax during evaluation
```

Not surprisingly, the ground evaluator reports in the latter case that `sqrt_ax` is an uninterpreted function symbol that cannot be evaluated. Quit the read-eval-print interface with the command `exit` and then quit PVS.

3.2 Writing User-defined Semantic Attachments

In the current context, create the file `pvs-attachments`:

```
(defattach quadratic.sqrt_ax(x) (sqrt x))
```

[1]The examples in this section are included in the directory `examples` of the PVSio distribution.

Listing 3.1: Theory `quadratic`

```
quadratic : theory
begin

 % Arbitrary, but unknown, real number
 Undef : real

 % Axiomatic definition of squared root function
 sqrt_ax(x:nnreal): {y:nnreal | y*y = x}

 % Discriminant of a quadratic formula
 discr(a,b,c:real):real=b*b-4*a*c

 % Computes the roots of quadratic equation ax^2+bx+c=0
 root(a,b,c:real,sign:bool):real =
   let d = discr(a,b,c) in
     if (a = 0 ∧ b = 0) ∨ d < 0 then Undef
     elsif a = 0 then -c/b
     elsif sign then
       (-b + sqrt_ax(d))/(2*a)
     else
       (-b - sqrt_ax(d))/(2*a)
     endif

end quadratic
```

The file `pvs-attachments` is a regular Common Lisp file that includes semantic attachment definitions. In PVSio, semantic attachments are defined via the macro `defattach`. In this case the Common Lisp function `sqrt` is associated to the uninterpreted PVS function `sqrt_ax` in the theory `quadratic`.

Restart PVS and load the ground evaluator interface:

```
<PVSio> root(1,1,-6,true);
==>
2.0

<PVSio> root(1,1,-6,false);
==>
-3.0

<PVSio> let x = root(1,1,-6,true) in x*x + x -6;
==>
0.0
```

Listing 3.2: Theory `quadratic_io`

```
quadratic_io : theory
begin

  quio : theory = quadratic{{sqrt_ax := SQRT}}

end quadratic_io
```

Note that `sqrt_ax` computes a floating point number rather than a real number. Indeed:

```
<PVSio> sqrt_ax(2);
==>
1.4142135

<PVSio> sqrt_ax(2)*sqrt_ax(2)=2;
==>
FALSE
```

This example shows that the Common Lisp code associated to the function `sqrt_ax` does not respect its PVS type declaration. Therefore, it would be unsound to enable a ground evaluation of `sqrt_ax`, and for that matter of any expression that uses `sqrt_ax`, in a formal proof.

Finally, try a few additional evaluations:

```
<PVSio> root(1,3,1,false);
==>
-2.618034

<PVSio> root(1,1,1,true);
Hit uninterpreted term quadratic.Undef during evaluation
```

The PVS constant `Undef` is intentionally left uninterpreted to force an exception when the function `root` is undefined.

3.3 Animating a Functional Specification

The simplest way to instrument a functional PVS specification for simulation and testing is through the predefined PVSio library. Quit PVS, delete or rename the file `pvs-attachments`, and restart PVS so that `sqrt` is no longer associated to `sqrt_ax`.

Create the file `quadratic_io.pvs` as shown in Listing 3.2. Theory `quadratic_io` uses *theory interpretations* [5] to substitute the uninterpreted function `sqrt_ax` in `quadratic` by the PVSio function `SQRT`. The PVS typechecker generates the following unprovable TCC:

```
OBLIGATION ∀(x: nnreal): SQRT(x)*SQRT(x) = x;
```

Ignore the TCC as the theory `quadratic_io` will be exclusively used to animate the theory `quadratic` and not as part of a formal development. Of course, if the evaluation of an expression assumes the validity of an unprovable TCC, the ground evaluator may break or not terminate.

Go to the ground evaluator and evaluate `root`:

```
<PVSio> root(1,3,1,false);
==>
-2.618034
```

More interestingly, PVSio provides input/output capabilities for writing interactive interfaces. For instance, add the following lines to the file `quadratic_io.pvs`:

```
 roots : void =
   println("Computes roots (x+,x-) of ax^2+bx+c=0") &
   let a = query_real("Enter a:") in
   let b = query_real("Enter b:") in
   let c = query_real("Enter c:") in
   let (x1,x2) = (root(a,b,c,true),root(a,b,c,false)) in
   println("x+ = "+x1+", x- = "+x2)
```

The type `void` and the functions `println` and `query_real` are, among many others, PVSio predefined functions. Actually, `void` is defined as the PVS type `bool` and `println(...)` is the PVS constant `true`. Sequential statements in PVSio are separated with the operator `&`, which is an alternative syntax for conjunctions in PVS. The expression `query_real(...)` has the PVS type `real`. In PVSio, `println` has the side effect of printing a message in the standard output and `query_real` has the side effects of printing a message in the standard output and reading a `real` value from the standard input.

Restart the ground evaluator and test the functional specification of `root` via `roots`:

```
<PVSio> roots;
Computes roots (x+,x-) of ax^2+bx+c=0
Enter a:
1.2
Enter b:
4.5
Enter c:
3.8
x+ = -1.284273, x- = -2.4657269
```

Note that the evaluation of `roots` does not print any result of the form `==> ....` This is because PVSio does not print the result of expressions of type `void`, such as `roots`. This is the only operational difference between the types `void` and `bool`.

3.4 Stand Alone Evaluations

PVSio provides a stand alone Emacs-free interface to the ground evaluator. The command

```
$ packages/PVSio/pvsio quadratic_io
```

executes the PVSio read-eval-loop with the theory `quadractic_io`:

```
+----
| PVSio-2.b (08/12/05)
|
| Enter a PVS ground expression followed by a symbol ';' at the <PVSio> prompt.
| Enter a Lisp expression followed by a symbol '!' at the <PVSio> prompt.
|
| Enter help! for a list of commands and quit! to exit the evaluator.
|
| *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>
```

The command

```
$ packages/PVSio/pvsio quadratic_io:roots
```

evaluates the PVS function `roots` in the theory `quadractic_io`:

```
Computes roots (x+,x-) of ax^2+bx+c=0
Enter a:
1
Enter b:
2
Enter c:
1
x+ = -1.0, x- = -1.0
==>
TRUE
```

3.5 Proving Ground Properties

PVSio safely integrates the ground evaluator to the PVS theorem prover. Indeed, PVSio provides proof rules `eval-expr` and `eval-formula` that use the Common Lisp code extracted by the ground evaluator to simplify ground expressions appearing in sequent formulas.

Create the file `sqrt_newton.pvs` as shown in Listing 3.3. The theory `sqrt_newton` defines `sqrt_approach`, which computes an approximation of the square root function using the Newton method. Upper and lower bounds are defined using `sqrt_approx`.[2] Typechek the

[2]A formal proof of the general statement $\forall$ `(x:nnreal): sqrt_lb(x)` $\leq$ `sqrt(x)` $<$ `sqrt_ub(x)` appears in the theory `sqrt_approx` of the NASA PVS reals library [1].

Listing 3.3: Theory `sqrt_newton`

```
sqrt_newton : theory
begin

  % Newton approximation of sqrt
  sqrt_approx(a:nnreal,n:nat): recursive posreal =
    if n=0 then a+1
    else let r=sqrt_approx(a,n-1) in
      (1/2)*(r+a/r)
    endif
    measure n+1

  % sqrt upper bound
  sqrt_ub(a:nnreal):posreal =
    sqrt_approx(a,10)

  % sqrt lower bound
  sqrt_lb(a:nnreal):nnreal =
    a/sqrt_approx(a,10)

  sqrt2 : lemma
    sqrt_lb(2) < sqrt_ub(2)

  sqrt_lb2 : lemma
   2/3 ≤ 1/sqrt_lb(2)

  foo : lemma
    1 = 0

end sqrt_newton
```

theory `sqrt_newton` and prove the TCCs.

Go to the lemma `sqrt2` and prove the lemma with the proof rule `grind`. It takes a few moments. Once the lemma has been discharged retry the lemma, but this time use the proof rule `eval-formula`:

```
sqrt2 :

  |-------
{1}  sqrt_lb(2) < sqrt_ub(2)

Rule? (eval-formula)
Evaluating formula 1 in the current sequent,
Q.E.D.
```

PVS proof rules such as `grind` and `assert` perform *symbolic* simplifications. In contrast, the PVSio proof rule `eval-formula` uses the ground evaluator to simplify ground sequent formulas via the underlying PVS Common Lisp machine.

PVSio also provides a proof rule `eval-expr` that evaluates a ground expression and equates the original expression to its ground evaluation. For example, go to the lemma `sqrt_lb2` and start the theorem prover:

```
sqrt_lb2 :

  |-------
{1}  2 / 3 ≤ 1 / sqrt_lb(2)

Rule? (eval-expr "sqrt_lb(2)")
Evaluating expression sqrt_lb(2) in the current sequent,
this simplifies to:
sqrt_lb2 :

{-1} sqrt_lb(2) =
      151114434154479588992967852505452971103298635323328458697811423468569176 86...
      ...
        /
      106854041125800542495773099620277025175306170088676005050927755840860348 66...
      ...
  |-------
[1]  2 / 3 ≤ 1 / sqrt_lb(2)
```

The goal is discharged with `(assert)`.

Unless the option `:safe?` is set to `nil`, the proof rule `eval-expr` refuses to evaluate an expression that generates TCCs:

```
Rule? (eval-expr "1/sqrt_lb(2)")
Typechecking "1/sqrt_lb(2)" produced TCCs:

subtype TCC for sqrt_lb(2): sqrt_lb(2) /= 0

Use option :safe? nil if TCCs are provable.

No change on: (eval-expr "1/sqrt_lb(2)")
```

The proof rule `eval-expr` disables ground evaluation of expressions that depend on semantic attachments. This way, potential soundness problems due to side-effects occurring in semantic attachments are avoided.

For example, assume that a malicious user tries to prove `foo` by evaluating an expression such as `RANDOM /= RANDOM`. `RANDOM` is a predefined PVSio function that produces a pseudo-random number in the interval $[0, 1]$ each time that the function is called. It is implemented as a semantic attachment via the the Common Lisp function `random`. However, the proof rules `eval-expr` and `eval-formula` refuse to evaluate ground expressions that depend on `RANDOM`:

```
foo :

  |-------
{1}  1 = 0

Rule? (eval-expr "RANDOM /= RANDOM")
Function stdmath.RANDOM is defined as a semantic attachment.
It cannot be evaluated in a formal proof.

No change on: (eval-expr "RANDOM /= RANDOM")
```

On the other hand, the proof rule `eval` evaluates arbitrary ground expressions; even if they are defined using semantic attachments. However, in contrast to `eval-expr` and `eval-formula`, `eval` does not actually modify the proof context. For this reason, it is logically safe to evaluate user defined semantic attachments using `eval`:

```
Rule? (eval "RANDOM /= RANDOM")
RANDOM /= RANDOM = TRUE

No change on: (eval "RANDOM /= RANDOM")
foo :

  |-------
{1}  1 = 0
Rule? (eval "RANDOM")
RANDOM = 0.96723104

No change on: (eval "RANDOM")
foo :

  |-------
{1}  1 = 0

Rule? (eval "RANDOM")
RANDOM = 0.93110865

No change on: (eval "RANDOM")
```

Chapter 4

Read-Eval-Print Interface

PVSio provides an alternative read-eval-print interface to the ground evaluator via the Emacs command `M-x pvsio`. It displays the following message in the `*pvs*` buffer:

```
+----
| PVSio-2.b (08/12/05)
|
| Enter a PVS ground expression followed by a symbol ';' at the <PVSio> prompt.
| Enter a Lisp expression followed by a symbol '!' at the <PVSio> prompt.
|
| Enter help! for a list of commands and quit! to exit the evaluator.
|
| *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>
```

The standard PVS interface to the ground evaluator is still available with the Emacs command `M-x pvs-ground-evaluator`. The main differences between both interfaces are:

- The PVSio interface is quiet and compilation messages are always turned off.
- In the PVSio interface, ground expressions are typed directly after the prompt and they are terminated by a semicolon ';'. In the standard PVS interface, expressions are surrounded by double quotes and, therefore, string values have to be escaped.
- The PVSio interface also doubles as a Common Lisp evaluator: Lisp expressions are typed directly after the prompt and they are terminated by an exclamation mark '!'. This feature is specially useful when debugging semantic attachments.

The examples in this document are illustrated using the PVSio interface. However, both interfaces provides roughly the same functionality with respect to the predefined PVSio library of semantic attachments.

4.1 PVSio Special Commands

The following commands can be followed by either a ';' or a '!'.

- `help`: Prints a help message.
- `quit`: Exits the evaluator with confirmation.
- `exit`: Exits the evaluator without confirmation.
- `timing`: Prints timing information for each evaluation.
- `notiming`: Turns off printing of timing information.
- `reload_pvsio`: Restarts PVSio, for example, after the ground evaluator has been abnormally interrupted.
- `load_pvs_attachments`: Forces a reload of semantic attachments from the current and imported contexts.
- `pvsio_version`: Shows the current version of PVSio.
- `list_attachments`: Lists the semantic attachments loaded in the current context.

Help for semantic attachments are available through the following commands:

- `(help_pvs_attachment` ***`attachment`***`)!` and `help_pvs_attachment(`***`attachment`***`);` display help for ***`attachment`***.
- `(help_pvs_theory_attachments` ***`theory`***`)!` and `help_pvs_theory_attachments(`***`theory`***`)` display help for attachments in ***`theory`***.

4.2 PVSio Emacs Commands

PVSio provides the following commands to be used in the PVS Emacs interface. They are the Emacs counterpart of some of the above special commands.

- `M-x reload-pvsio`: Restarts PVSio.
- `M-x load-pvs-attachments`: Forces a reload of semantic attachments.
- `M-x pvsio-version`: Shows the current version of PVSio.
- `M-x list-attachments`: Lists the semantic attachments loaded in the current context.
- `M-x help-pvs-attachment` (`C-c C-h a`): Displays help for a given semantic attachment.
- `M-x help-pvs-theory-attachments` (`C-c C-h t`): Displays help for semantic attachments of a given theory.

Things to Remember

- `<PVSio>` *`expr`*`;` to evaluate PVS expression *`expr`*.
- `<PVSio>` *`expr`*`!` to evaluate Common Lisp expression *`expr`*.
- `<PVSio> help!` to get a help message.
- `<PVSio> quit!` to exit the ground evaluator.
- `M-x list-attachments` to list all loaded attachments.
- `C-c C-h a` to display help for a given semantic attachment.
- `C-c C-h` *`t`* to display help for semantic attachments in a given theory.
- `reload_pvsio!` to restart PVSio if necessary.

Chapter 5

Semantic Attachments

Semantic attachments in PVSio should be written in a file `pvs-attachments` in the current context or in the file `~`*`user`*`/.pvs-attachments`. PVSio automatically loads these two files at startup. If needed, the Emacs command `M-x pvs-load-attachments` reloads the file `pvs-attachments` in the current context. This may be useful if the file has been modified during the PVS session.

The files `pvs-attachments` and `.pvs-attachments` are Common Lisp files. They may contain arbitrary Lisp forms. Semantic attachment definitions have the form:

```
(defattach theory.attachment (parameters)
  ⟦ docstr ⟧
  body)
```

The macro `attachments` can be used to define multiple semantic attachments for the same theory:

```
(attachments theory

 (defattach attachment (parameters)
   ⟦ docstr ⟧
   body)

 ...

)
```

The macro `defattach` attaches the Lisp code *`body`* to *`attachment`* with parameters *`parameters`* in theory *`theory`*. In contrast to PVS, Common Lisp is by default case insensitive. Therefore, PVS identifiers that contain uppercase letters have to be surrounded by bars in Common Lisp. The list of names in *`parameters`* is empty when *`attachment`* is the semantic attachment of a PVS constant. The documentation string *`docstr`* is optional; it is used to produce the help messages displayed by the Emacs commands `M-x help-pvs-attachment` (`C-c C-h a`) and `M-x help-pvs-theory-attachments` (`C-c C-h t`). The *`body`* is a regular Common Lisp expression. For instance, the semantic attachments of the constant `PI` and the function `SIN` in the PVSio theory `stdmath` are defined:

```
(attachments stdmath

 (defattach |PI| ()
  "Number Pi"
  pi)

 (defattach |SIN| (x)
  "Sine of X"
  (sin x))
)
```

Semantic attachment definitions may call other semantic attachments, including themselves in recursive calls. The function name of a semantic attachment ***theory*.*attachment*** is `pvsio_`*`theory`*`_`*`attachment`*`_`*`n`*, where ***n*** is the number of parameters in the attachment. For example, the attachment definition of `fibonacci(n,f1,f2:nat):nat` in a theory `myfibo` could be written:

```
(defattach myfibo.fibonacci (n f1 f2)
 (if (= n 0) f1
   (pvsio_myfibo_fibonacci_3 (- n 1) f2 (+ f1 f2))))
```

5.1 Overloading

Overloading of semantic attachments is partially supported in PVSio. That is, semantic attachments with the same name can be defined if they are in different theories or if they are in the same theory but they have different number of parameters. On the other hand, the ground evaluator fully supports PVS overloading. For example, the PVSio theory `stdstr` contains the following declarations of two functions `tostr` that convert, respectively, real numbers to string and Boolean values to string:

```
tostr(r:real): string = real2str(r)                               [Function]
tostr(b:bool): string = bool2str(b)                               [Function]
```

The functions `real2str` and `bool2str` are appropriately defined via semantic attachments. Thus,

```
<PVSio> tostr(SQRT(2));
==>
"1.4142135"

<PVSio> tostr(exists (x:subrange(1,10)): x=5);
==>
"TRUE"
```

5.2 From PVS to Common Lisp and Back Again

All data structures in PVS are defined as classes in the Common Lisp Object System (CLOS) [8]. However, the formal parameters in ***parameters*** and the result returned by ***body*** are type-free Common Lisp representations of PVS CLOS data structures. Basic PVS types such as `real`, `bool`, and `string` are the built-in Common Lisp types rationals, Booleans, and strings, respectively. PVS enumerate types are represented by integers, PVS records are represented by arrays, and PVS functional closures are represented by Common Lisp `lambda` expressions. Since access to complex PVS data structures may be difficult in the type-free Common Lisp representation, it is highly recommended to use only basic types in the domain and range of semantic attachments.

If absolutely necessary, back and forth translations between PVS CLOS and type-free Common Lisp representations are possible through the predefined Common Lisp functions `pvs2cl` and `cl2pvs`:

- `(pvs2cl` ***clos-expr***`)`: Returns the type-free Common Lisp representation of the PVS CLOS expression ***clos-expr***.

- `(cl2pvs` ***lisp-expr clos-type***`)`: Returns the PVS CLOS representation of the type-free Common Lisp expression ***lisp-expr*** of PVS CLOS type ***clos-type***.

The macro `defattach` provides a hidden parameter `*the-pvs-type*` that is instantiated with the string representation of the PVS type of the attachment. For example, the attachment `typeof`, in the PVSio theory `stdpvs`, that returns the string value of the PVS type of its argument is defined as follows:

```
(defattach typeof(e)
 "Returns the string value of the type of E"
 (let* ((the-type (pc-parse *the-pvs-type* 'type-expr))
        (domain (domain the-type)))
   (format nil "~a" (or (print-type domain) domain))))
```

Thus,

```
<PVSio> typeof(SQRT(2));
==>
"nnreal"

<PVSio> typeof(∃(x:subrange(1,10)):x=5);
==>
"boolean"
```

Things to Remember

- Semantic attachments are defined in the file `pvs-attachments` in the current context or in the file `~`***user***`/.pvs-attachments`.

- Semantic attachment definitions have the form:

```
(defattach theory.attachment (parameters) ⟦ docstr ⟧ body)
```

- Semantic attachment definitions for the same theory can be grouped using the macro `attachments`:

```
(attachments theory (defattach attachment ... ) ... )
```

- Semantic attachments with the same name can be defined if they are in different theories or if they are in the same theory but they have different number of parameters
- Parameters of semantic attachments and the returned values should be basic objects such as numbers, Booleans, and strings.

Chapter 6

Library

PVSio predefined theories are implicit imported in a working context where PVSio has been loaded. The PVS library consists of the following theories:

- `stdlang`: Basic definitions.
- `stdstr`: String operations.
- `stdio`: Input/output operations.
- `stdfmap`: File iterations.
- `stdmath`: Floating point arithmetic.
- `stdpvs`: PVS parsing and printing.
- `stdindent`: Pretty printing via indentations.
- `stdtokenizer`: Lexing and parsing via tokenizers.
- `stdexc`: Definition of exceptions.
- `stdcatch`: Exception handling.
- `stdprog`: Imperative programming features.
- `stdglobal`: Global variables.
- `stdpvsio`: PVSio interface.
- `stdsys`: System utilities.

The logical structure of the the library gives a good overview of the functionality provided by PVSio. However, it should be noted that for technical reasons some functions are defined in unexpected theories.

6.1 Statements

The type `void` is intended to be used as the type of procedures and statements, i.e., functions and expressions with side effects. This usage is a convention rather than a policy enforced by the typechecker. Indeed, `void` is just an alias to `bool`. Constants `skip` and `fail` are empty statements that represent success (`true`) and failure (`false`), respectively.

```
void : TYPE = bool                                                   [Type]
skip : void = true                                               [Constant]
fail : void = false                                              [Constant]
```

6.2 Sequence and Bounded Loop

The type encoding of statements enables the use of the PVS operator `&` (logical *and*) as separator of sequential statements and the universal quantifier `forall` as the bounded loop operator. For instance, the statement `forall (i:subrange(n,m)):s` iterates the statement *s* for `i` varying sequentially from *n* to *m*.

Statements and expressions can be composed using the operator `prog`:

```
prog(s:void,t:T): T = t                                          [Function]
```

For instance, `prog[T](s,expr)` evaluates statement *s* and then returns expression *expr* of type *T*. If *T* is not provided, it will be inferred by the type checker.

6.3 Conditionals

In addition to standard PVS conditional operators, i.e., `if`, `cases`, `cond`, `table`, PVSio provides:

```
try(s1:void,s2:void) : MACRO void = s1 or s2                         [Macro]
try(s:void) : MACRO void = try(s,skip)                               [Macro]
ifthen(b:bool,s:void) : MACRO void = if b then s else skip endif     [Macro]
ifelse(b:bool,s:void) : MACRO void = if b then skip else s endif     [Macro]
```

Note the use of `MACRO` in these definitions. In the presence of side effects, this is necessary to simulate lazy evaluation of parameters.

6.4 Unbounded Loops

6.5 Exceptions

6.6 Mutable and Global Variables

6.7 Strings

PVSio complements the basic definition of the type `string` in the PVS prelude library with constants, operators, and functions on string values. The theory `stdstr` defines the following self-explained string constants:

```
emptystr : string                                    [Constant]
space : string                                       [Constant]
newline : string                                     [Constant]
tab : string                                         [Constant]
doublequote : string                                 [Constant]
singlequote : string                                 [Constant]
backquote : string                                   [Constant]
```

The table of standard Common Lisp symbols is printed by the procedure `chartable`. Symbol codes are converted to string values with the function `charcode`.

```
chartable : void                                     [Function]
charcode(n:nat) : string                             [Function]
```

For example:

```
<PVSio> chartable;
 32 :    33 : !  34 : "  35 : #  36 : $  37 : %  38 : &  39 : '  40 : (  41 : )
 42 : *  43 : +  44 : ,  45 : -  46 : .  47 : /  48 : 0  49 : 1  50 : 2  51 : 3
 52 : 4  53 : 5  54 : 6  55 : 7  56 : 8  57 : 9  58 : :  59 : ;  60 : <  61 : =
 62 : >  63 : ?  64 : @  65 : A  66 : B  67 : C  68 : D  69 : E  70 : F  71 : G
 72 : H  73 : I  74 : J  75 : K  76 : L  77 : M  78 : N  79 : O  80 : P  81 : Q
 82 : R  83 : S  84 : T  85 : U  86 : V  87 : W  88 : X  89 : Y  90 : Z  91 : [
 92 : \  93 : ]  94 : ^  95 : _  96 : `  97 : a  98 : b  99 : c 100 : d 101 : e
102 : f 103 : g 104 : h 105 : i 106 : j 107 : k 108 : l 109 : m 110 : n 111 : o
112 : p 113 : q 114 : r 115 : s 116 : t 117 : u 118 : v 119 : w 120 : x 121 : y
122 : z 123 : { 124 : | 125 : } 126 : ~
==>
TRUE

<PVSio> charcode(35);
==>
"#"
```

PVS does not support decimal notation of real numbers. PVSio alleviates this limitation with the function `str2real`. It takes the string representation of a number in decimal notation and returns its value as a rational number. The function `str2int` returns the integer value represented by a string. These functions raise exceptions if the string does not represent a numerical value of the given type. The functions `number?` and `int?` check if a given string represents a numerical value and an integer number, respectively. The function `tostr` returns the string representation of a basic value. The operator `+` concatenates two strings. Functions `tostr` and `+` are overloaded for basic PVS types, i.e., `real`, `bool`, and `string`. Indeed, the operator `+` behaves pretty much as `+` in Java.

```
str2real(s:string) : rat                                      [Function]
str2int(s:string) : int                                       [Function]
number?(s:string) : bool                                      [Function]
int?(s:string) : bool                                         [Function]
tostr(x) : string                                             [Function]
+(x,y) : string                                               [Function]
```

For example:

```
<PVSio> str2real("1343.45");
==>
26869/20

<PVSio> int?("1345.45");
==>
FALSE

<PVSio> tostr(PI*PI);
==>
"9.869604"

<PVSio> (1+1)+"1";
==>
"21"

<PVSio> true+tab+"/="+newline+tab+false;
==>
"TRUE /=
      FALSE"
```

String search, substrings and string comparison

The function `strfind` returns the index of leftmost occurrence of `s1` in `s2`, starting from 0, or -1 if `s1` does not occur in `s2`. The function `substr` extracts the substring of `s` from `i` to `j` if `i<=j`, from `j` to `i` if `j<=i`, or `emptystr` if indices are out of range. The function `strcmp` returns -1 if `s1` $<$ `s2`, 1 if `s1` $>$ `s2`, 0 if strings are equal. Comparison is case sensitive when the parameter `sensitive` is `TRUE`.

```
strfind(s1,s2:string) : int                                  [Function]
substr(s:string,i,j:nat) : string                            [Function]
strcmp(s1,s2:string,sensitive:bool) : int                    [Function]
strcmp(s1,s2:string): int = strcmp(s1,s2,true)               [Function]
```

Upcase, downcase, capitalize

```
upcase(s:string) : string                                    [Function]
downcase(s:string) : string                                  [Function]
capitalize(s:string) : string                                [Function]
```

Trimming

The functions `strtrim`, `strtrim_left`, and `strtrim_right` trim a string `s2` with respect to characters in string `s1`. The functions `trim`, `trim_left`, and `trim_right` trim string `s` with respect to `newline`, `space`, and `tab`.

```
strtrim(s1,s2:string) : string                               [Function]
strtrim_left(s1,s2:string) : string                          [Function]
strtrim_right(s1,s2:string) : string                         [Function]
trim(s:string) : string                                      [Function]
trim_left(s:string) : string                                 [Function]
trim_right(s:string) : string                                [Function]
```

Paths and file names

```
filename(s:string) : string                                  [Function]
directory(s:string) : string                                 [Function]
```

Formatting and padding

Function `pad` concatenates `n` times a given string `s`, function `spaces` returns a string with `n` spaces, functions `center`, `flushleft`, and `flushright` justify a string with respect to a number of columns, and function `format` mimics in PVSio the behavior of the Common Lisp function `format`.

```
pad(n:nat,s:string) : string
spaces(n:nat) : string
center(col:nat,s:string) : string
flushleft(col:nat,s:string) : string
flushright(col:nat,s:string) : string
format(s:string,x) : string
```

More examples

```
<PVSio> strcmp("hola","HOLA",false);
==>
0

<PVSio> strcmp("hola","HOLA",true);
==>
1

<PVSio> strtrim("01","0001ABCD110");
==>
"ABCD"

<PVSio> trim_right("Hola    ");
==>
"Hola"

<PVSio> pad(50,"-")+newline+center(50,"center")+newline+flushleft(50,"left")+
newline+flushright(50,"right");
==>
"--------------------------------------------------
                      center
left
                                             right"
```

```
<PVSio> format("~%~6a:~5d~%~6a:~5d~%~6a:~5d~%",("day",1,"month",1,"year",2000));
==>
"
day   :    1
month :    1
year  : 2000
"
```

6.8 Input/Output

(see `lib/stdio.pvs` and [3])

6.9 Floating Point Arithmetic

(see `lib/stdmath.pvs` and [3])

6.10 Pretty Printing

(see `lib/stdindent.pvs` and [3])

6.11 Parsing

(see `lib/stdtokenizer.pvs` and [3])

6.12 PVS Lexer and Parser

(see `lib/stdpvs.pvs` and [3])

Chapter 7

Stand Alone Interface

PVSio provides the Unix command `lib/pvsio` for stand alone (Emacs-free) evaluations:

```
pvsio [-options ...] [<file>@]<th>[:<main>]                          [Command]
```

where `options` include:

- `-e|-expr <expr>`: Evaluates <expr> after startup.
- `-h|-help`: Prints a help message.
- `-p|-packages <p1>,..,<pn>`: Loads packages `<p1>`...`<pn>`.
- `-tccs`: Generates TCCs.
- `-timing`: Prints timing information for each evaluation.
- `-verbose`: Prints typechecking information.
- `-version`: Prints PVSio version.

The command loads theory `<th>`, from file `<file>.pvs`, evaluates the expression `<main>` in PVSio, and exits. If `<file>` is not provided, `<th>` is assumed to be the name of the file too. If `<main>` is not provided, `pvsio` starts the PVSio read-eval-loop with the theory `<th>`.

For instance, given this PVS theory `hello` in Listing 7.1, we have

```
$ pvsio hello:main
Hello, World!
```

There is nothing special about the name `main`, it can be any PVS ground expression, e.g.,

```
$ pvsio hello:'you(\"Tom\")'
Hello, Tom!
```

Or even,

```
$ pvsio hello:'1+2+3+4+5+6+7+8+9'
==>
45
```

Note that string values have to be escaped. Furthermore, to prevent Unix for preprocessing the command line, if may be necessary to surround the PVS expression with single quotes.

Listing 7.1: Theory `hello`

```
hello : THEORY
BEGIN

 main : void =
  println("Hello, World!");

 you(name:string):void =
  println("Hello, "+name+"!");

END hello
```

Chapter 8

Proof Rules

PVSio uses a conservative approach to integrate the ground evaluator to the theorem prover. It considers that all user-defined semantic attachments are unsafe and, consequently, disables their ground evaluation in the theorem prover.

8.1 eval-formula

`(eval-formula &optional (fnum 1) safe?)` *[Strategy]*

Evaluates the formula `fnum` in Common Lisp and adds the result to the antecedent of the current goal. If `safe?` is `t` and `fnum` generates TCCs, the expression is not evaluated. The strategy is safe in the sense that user-defined semantic attachments are not evaluated. However, the strategy may fail in the presence of unproven TCCs.

8.2 eval-expr

`(eval-expr expr &optional (safe? t) (auto? t))` *[Strategy]*

Adds the antecedent `expr=`*eval* to the current goal, where *eval* is the Common Lisp evaluation of `expr`. If `safe?` is `t` and `expr` generates TCCs, the expression is not evaluated. Otherwise, TCCs are added as subgoals and the expression is evaluated. If `auto?` is `t`, TCCs are ground evaluated. The strategy is safe in the sense that user-defined semantic attachments are not evaluated. However, the strategy may fail or loop in the presence of unproven TCCs.

8.3 eval

`(eval expr &optional (safe? t))` *[Strategy]*

Evaluates expression `expr`. If `safe?` is `t` and `expr` generates TCCs, the expression is not evaluated.

PVSio Index

Bibliography

[1] Formal Methods Groups at NASA Langley and National Institute of Aerospace. NASA langley PVS libraries. Available at `http://shemesh.larc.nasa.gov/fm/ftp/larc/PVS2-library/pvslib.html`.

[2] J. Crow, S. Owre, J. Rushby, N. Shankar, and D. Stringer-Calvert. Evaluating, testing, and animating PVS specifications. Technical report, Computer Science Laboratory, SRI International, Menlo Park, CA, March 2001. Available at `http://www.csl.sri.com/users/rushby/abstracts/attachments`.

[3] C. Muñoz. Rapid prototyping in pvs. Report NIA Report No. 2003-03 and NASA/CR-2003-212418, NIA-NASA Langley, National Institute of Aerospace, Hampton, VA, May 2003.

[4] S. Owre, J. M. Rushby, and N. Shankar. PVS: A prototype verification system. In Deepak Kapur, editor, *11th International Conference on Automated Deduction (CADE)*, volume 607 of *Lecture Notes in Artificial Intelligence*, pages 748–752, Saratoga, NY, June 1992. Springer-Verlag.

[5] S. Owre and N. Shankar. Theory interpretations in pvs. Technical Report SRI-CSL-01-01, Computer Science Laboratory, SRI International, Menlo Park, CA, April 2001.

[6] Sam Owre and Natarajan Shankar. The formal semantics of PVS. Technical Report SRI-CSL-97-2, Computer Science Laboratory, SRI International, Menlo Park, CA, August 1997.

[7] N. Shankar. Efficiently executing PVS. Project report, Computer Science Laboratory, SRI International, Menlo Park, CA, November 1999. Available at `http://www.csl.sri.com/shankar/PVSeval.ps.gz`.

[8] N. Shankar, S. Owre, J. M. Rushby, and D. W. J. Stringer-Calvert. *PVS Prover Guide*. Computer Science Laboratory, SRI International, Menlo Park, CA, September 1999.