PVS 3.0 release notes

Sam Owre ‹owre@csl.sri.com›, SRI International

1 Overview

We are still working on updating the documentation, and completion of the ICS (`http://ics.csl.sri.com`) decision procedures. Please let us know of any bugs or suggestions you have by sending them to *pvs-bugs@csl.sri.com* (`mailto:pvs-bugs@csl.sri.com`)

You can download it here (`download.html`).

In addition to the usual bug fixes, there are quite a few changes to this release. Most of these changes are backward compatible, but the new multiple proofs feature makes it difficult to run PVS 3.0 in a given context and then revert back to an earlier version. For this reason we strongly suggest that you copy existing directories (especially the proof files) before running PVS 3.0 on existing specifications.

2 New Features

There are a number of new features in PVS 3.0.

2.1 Allegro 6.2 port

PVS 3.0 has been ported to the case-sensitive version of Allegro version 6.0. This was done in order to be able to use the XML support provided by Allegro 6.0. We plan to both write and read XML abstract syntax for PVS, which should make it easier to interact with other systems.

Note: for the most part, you may continue to define pvs-strategies (and the files they load) as case insensitive, but in general this cannot always be done correctly, and it means that you cannot load such files directly at the lisp prompt. If you suspect that your strategies are not being handled properly, try changing it to all lower case (except in specific cases), and see if that helps. If not, send the strategies file to pvs-bugs (`mailto:pvs-bugs@csl-sri.com`) and we'll fix it as quickly as we can. Because there is no way to handle it robustly, and since case-sensitivity can actually be useful, in the future we may no longer support mixed cases in strategy files.

2.2 Theory Interpretations

Theory interpretations are described fully in Theory Interpretations in PVS (`doc/interpretations.html`)

NOTES:

- This introduces one backward incompatible change; theory abbreviations such as

  ```
  foo: THEORY = bar[int, 3]
  ```

  should be changed to the new form

  ```
  IMPORTING bar[int, 3] AS foo
  ```

  Note that '`AS`' is a new keyword, and may cause parse errors where none existed before.

- The stacks example doesn't work as given; in particular, the mappings for `push`, `top`, and `pop` should be changed to

  ```
  push := LAMBDA (x: t, A: E[cstack, ce]):
            equiv_class[cstack,ce](cpush(x)(rep(A))),
  top := LAMBDA (A: E[cstack, ce] | cnonempty?(rep(A))): ctop(rep(A)),
  pop := LAMBDA (A: E[cstack, ce] | cnonempty?(rep(A))):
            equiv_class[cstack,ce](cpop(rep(A)))
  ```

  Otherwise unprovable TCCs result (e.g., every stack is nonempty).

2.3 Multiple Proofs

PVS now supports multiple proofs for a given formula. When a proof attempt is completed, either by quitting or successfully completing the proof, the proof is checked for changes. If any changes have occured, the user is queried about whether to save the proof, and whether to overwrite the current proof or to create a new proof. If a new proof is created, the user is prompted for a proof identifier and description.

In addition to a proof identifier, description, and proof script, the proof objects contain the status, the date of creation, the date last run, and the run time. Note that this information is kept in the `.prf` files, which therefore look different from those of earlier PVS versions.

Every formula that has proofs has a default proof, which is used for most of the existing commands, such as prove, prove-theory, and status-proofchain. Whenever a proof is saved, it automatically becomes the default.

Three new Emacs commands allow for browsing and manipulating multiple proofs: `display-proofs-formula`, `display-proofs-theory`, and `display-proofs-pvs-file`. These commands all pop up buffers with a table of proofs. The default proof is marked with a '`+`'. Within such buffers, the following keys have the following effects.

Key	Effect
c	Change description: add or change the description for the proof
d	Default proof: set the default to the specified proof
e	Edit proof: bring up a Proof buffer for the specified proof; the proof may then be applied to other formulas
p	Prove: rerun the specified proof (makes it the default)
q	Quit: exit the Proof buffer
r	Rename proof: rename the specified proof
s	Show proof: Show the specified proof in a Proof:*id* buffer
DEL	Delete proof: delete the specified proof from the formula

At the end of a proof a number of questions may be asked:

- Would you like the proof to be saved?
- Would you like to overwrite the current proof?
- Please enter an id
- Please enter a description:

This may be annoying to some users, so the command `M-x pvs-set-proof-prompt-behavior` was added to control this. The possible values are:

`:ask` the default; all four questions are asked

`:overwrite`
similar to earlier PVS versions; asks if the proof should be saved and then simply overwrites the earlier one.

`:add` asks if the proof should be saved, then creates a new proof with a generated id and empty description.

Note that the id and description may be modified later using the commands described earlier in this section.

2.4 Better Library Support

PVS now uses the `PVS_LIBRARY_PATH` environment variable to look for library pathnames, allowing libraries to be specified as simple (subdirectory) names. This is an extension of the way, for example, the `finite_sets` library is found relative to the PVS installation path - in fact it is implicitly appended to the end the `PVS_LIBRARY_PATH`.

The `.pvscontext` file stores, amongst other things, library dependencies. Any library found as a subdirectory of a path in the `PVS_LIBRARY_PATH` is stored as simply the subdirectory name. Thus if the `.pvscontext` file is included in a tar file, it may be untarred on a different machine as long as the needed libraries may be found in the `PVS_LIBRARY_PATH`. This makes libraries much more portable.

In addition, the `load-prelude-library` command now automatically loads the `pvs-lib.el` file, if it exists, into Emacs and the `pvs-lib.lisp` file, if it exists, into lisp, allowing the library to add new features, e.g., key-bindings. Note that the `pvs-lib.lisp` file is not needed for new strategies, which should go into the `pvs-strategies` file as usual. The difference is that the `pvs-strategies` file is only loaded when a proof is started, and it may be desirable to have some lisp code that is loaded when the library is, i.e., to support some new Emacs key-bindings.

The `PVS_LIBRARY_PATH` is a colon-separated list of paths, and the `lib` subdirectory of the PVS path is added implicitly at the end. Note that the paths given in the `PVS_LIBRARY_PATH` are expected to have subdirectories, e.g., if you have put Ben Di Vito's Manip-package (`http://shemesh.larc.nasa.gov/people/bld/manip.html`) in `~/pvs-libs/Manip-1.0`, then your `PVS_LIBRARY_PATH` should only include `~/pvs-libs`, not `~/pvs-libs/Manip-1.0`.

If the `pvs-libs.lisp` file needs to load other files in other libraries, use `libload`. For example, César Muñoz's Field Package (`http://www.icase.edu./~munoz/Field/field.html`) loads the `Manip-package` using `(libload "Manip-1.0/manip-strategies")`

A new command, `M-x list-prelude-libraries`, has been added that shows the prelude library and supplemental files that have been loaded in the current context.

2.5 Cotuples

PVS now supports cotuple types (also known as coproduct or sum types) directly. The syntax is similar to that for tuple types, but with the '`,`' replaced with a '`+`'. For example,

```
cT: TYPE = [int + bool + [int -> int]]
```

Associated with a cotuple type are injections `IN_`*i*, predicates `IN?_`*i*, and extractions `OUT_`*i* (none of these is case-sensitive). For example, in this case we have

```
IN_1:  [int -> cT]
IN?_1: [cT -> bool]
OUT_1: [(IN?_1) -> int]
```

Thus `IN_2(true)` creates a `cT` element, and an arbitrary `cT` element `c` is processed using `CASES`, e.g.,

```
CASES c OF
  IN_1(i): i + 1,
  IN_2(b): IF b THEN 1 ELSE 0 ENDIF,
  IN_3(f): f(0)
ENDCASES
```

This is very similar to using the `union` datatype defined in the prelude, but allows for any number of arguments, and doesn't generate a datatype theory.

Typechecking expressions such as `IN_1(3)` requires that the context be known. This is similar to the problem of a standalone `PROJ_1`, and both are now supported:

```
F: [cT -> bool]
FF: FORMULA F(IN_1(3))
G: [[int -> [int, bool, [int -> int]]] -> bool]
GG: FORMULA G(PROJ_1)
```

This means it is easy to write terms that are ambiguous:

```
HH: FORMULA IN_1(3) = IN_1(4)
HH: FORMULA PROJ_1 = PROJ_1
```

This can be disambiguated by providing the type explicitly:

```
HH: FORMULA IN_1[cT](3) = IN_1(4)
HH: FORMULA PROJ_1 = PROJ_1[[int, int]]
```

This uses the same syntax as for actual parameters, but doesn't mean the same thing, as the projections, injections, etc., are builtin, and not provided by any theories. Note that coercions don't work in this case, as `PROJ_1::[[int, int] -> int]` is the same as

```
(LAMBDA (x: [[int, int] -> int]): x)(PROJ_1)
```

and not

```
LAMBDA (x: [int, int]): PROJ_1(x)
```

The prover has been updated to handle extensionality and reduction rules as expected.

2.6 Coinduction

Coinductive definitions are now supported. They are like inductive definitions, but introduced with the keyword '`COINDUCTIVE`', and generate the greatest fixed point.

2.7 Datatype Updates

Update expressions now work on datatypes, in much the same way they work on records. For example, if `lst: list[nat]`, then `lst WITH [‘car := 0]` returns the list with first element

0, and the rest the same as the cdr of `lst`. In this case there is also a TCC of the form `cons?(lst)`, as it makes no sense to set the car of `null`.

Complex datatypes with overloaded accessors and dependencies are also handled. For example,

```
dt: DATATYPE
BEGIN
 c0: c0?
 c1(a: (even?), b: int): c1?
 c2(a: nat, c: int): c2?
END dt

datatype_update: THEORY
BEGIN
 IMPORTING dt
 x: dt
 y: int
 f: dt = x WITH [a := y]
END datatype_update
```

This generates the TCC

```
f_TCC1: OBLIGATION
  (c1?(x) AND IF c1?(x) THEN even?(y) ELSE y >= 0 ENDIF) OR
   (c2?(x) AND IF c1?(x) THEN even?(y) ELSE y >= 0 ENDIF);
```

2.8 Datatype Additions

There are two additions to the theory generated from a datatype: a new ord function, and an every relation. Both of these can be seen by examining the generated theories.

The new ord function is given as a constant followed by an ordinal axiom. The reason for this is that the disjointness axiom is not generated, and providing interpretations for datatype theories without it is not sound. However, for large numbers of constructors, the disjointness axiom gets unwieldy, and can significantly slow down typechecking. The ord axiom simply maps each constructor to a natural number, thus using the builtin disjointness of the natural numbers. For lists, the new ord function and axiom are

```
list_ord: [list -> upto(1)]

list_ord_defaxiom: AXIOM
  list_ord(null) = 0 AND
   (FORALL (car: T, cdr: list): list_ord(cons(car, cdr)) = 1);
```

This means that to fully interpret the list datatype, `list_ord` must be given a mapping and shown to satisfy the axiom.

If a top level datatype generates a map theory, the theory also contains an `every` relation. For lists, for example, it is defined as

```
every(R: [[T, T1] -> boolean])(x: list[T], y: list[T1]):  boolean =
    null?(x) AND null?(y) OR
     cons?(x) AND
```

```
        cons?(y) AND R(car(x), car(y)) AND every(R)(cdr(x), cdr(y));
```

Thus, `every(<)(x, y: list[nat])` returns true if the lists x and y are of the same length, and each element of `x` is less than the corresponding element of `y`.

2.9 Conversion Extensions

Conversions are now applied to the components of tuple, record, and function types. For example, if `c1` is a conversion from `nat` to `bool`, and `c2` from `nat` to `list[bool]`, the tuple `(1, 2, 3)` will be converted to `(c1(1), 2, c2(3))` if the expected type is `[bool, nat, list[bool]]`. Records are treated the same way, but functions are contravariant in the domain; if `f` is a function of type `[bool -> list[bool]]`, and the expected type is `[nat -> bool]`, then the conversion applied is `LAMBDA (x: nat): c2(f(c1(x)))`.

Conversions now apply pointwise where possible. In the past, if `x` and `y` were state variables, and `K_conversions` enabled, then `x < y` would be converted to `LAMBDA (s: state): x(s) < y(s)`, but `x = y` would be converted to `LAMBDA (s: state): x = y`, since the equality typechecks without applying the conversion pointwise. Of course, this is rarely what is intended; it says that the two state variables are the same, i.e., aliases. The conversion mechanism has been modified to deal with this properly.

2.10 Conversion Messages

Messages related to conversions have been separated out, so that if any are generated a message is produced such as

```
po_lems typechecked in 9.56s: 10 TCCs, 0 proved, 3 subsumed,
                      7 unproved; 4 conversions; 2 warnings; 3 msgs
```

In addition, the commands `M-x show-theory-conversions` and `M-x show-pvs-file-conversions` have been added to view the conversions.

2.11 More TCC Information

Trivial TCCs of the form `x /= 0 IMPLIES x /= 0` and `45 < 256` used to quietly be suppressed. Now they are added to the messages associated with a theory, along with subsumed TCCs. In addition, both trivial and subsumed TCCs are now displayed in commented form in the show-tccs buffer.

2.12 Show Declaration TCCs

The command `M-x show-declaration-tccs` has been added. It shows the TCCs associated with the declaration at the cursor, including the trivial and subsumed TCCs as described above.

2.13 Numbers as Constants

Numbers may now be declared as constants, e.g.,

```
42: [int -> int] = LAMBDA (x: int): 42
```

This is most useful in defining algebraic structures (groups, rings, etc.), where overloading 0 and 1 is common mathematical practice. It's usually a bad idea to declare a constant to be of a number type, e.g.,

```
42: int = 57
```

Even if the typechecker doesn't get confused, most users would.

2.14 Theory Search

When the parser encounters an importing for a theory `foo` that has not yet been typechecked, it looks first in the `.pvscontext` file, then looks for `foo.pvs`. In previous versions, if the theory wasn't found at this point an error would result. The problem is that file names often don't match the theory names, either because a given file may have multiple theories, or a naming convention (e.g., the file is lower case, but theories are capitalized)

Now the system will parse every `.pvs` file in the current context, and if there is only one file with that theory id in it, it will be used. If multiple files are found, a message is produced indicating which files contain a theory of that name, so that one of those may be selected and typechecked.

NOTES:

- Once a file has been typechecked, the `.pvscontext` is updated accordingly, and this check is no longer needed.
- `.pvs` files that contain parse errors will be ignored.

2.15 Improved Decision Procedures

The existing (named Shostak, for the original author) decision procedures have been made more complete. Note that this sometimes breaks existing proofs, though they are generally easy to repair, especially if the proof is rerun in parallel with the older PVS version. If you have difficulties repairing your proofs, please let us know.

2.16 ICS Integration

PVS 3.0 now has an alpha test integration of the ICS decision procedure (`http://ics.csl.sri.com"`). Use `M-x set-decision-procedure ics` to try it out. Note that this is subject to change, so don't count on proofs created using ICS to work in future releases. Please let us know of any bugs encountered.

2.17 LET Reduce

The `BETA` and `SIMPLIFY` rules, and the `ASSERT`, `BASH`, `REDUCE`, `SMASH`, `GRIND`, `GROUND`, `USE`, and `LAZY-GRIND` strategies now all take an optional `LET-REDUCE?` flag. It defaults to `t`, and if set to `nil` keeps `LET` expressions from being reduced.

2.18 Prelude Changes

2.18.1 New Theories

`restrict_props, extend_props`
: Provides lemmas that `restrict` and `extend` are identities when the subtype equals the supertype.

`indexed_sets`
: Provides indexed union and intersection operations and lemmas.

`number_fields`
: The `real` theory was split into two, with `number_fields` providing the field axioms and the subtype `reals` providing the ordering axioms. This allows for theories such as complex numbers to be inserted in between, thus allowing reals to be a subtype of complex numbers without having to encode them.

`nat_fun_props`
: Defines special properties of injective/surjective functions over nats, provided by Bruno Dutertre.

`finite_sets`
: combination of `finite_sets_def` (which was in the 2.4 prelude), `card_def`, and `finite_sets` (from the finite_sets library)

`bitvectors:`
: To provide support for the bitvector theory built in to ICS, the following theories were moved from the bitvectors library to the prelude: `bit`, `bv`, `exp2`, `bv_cnv`, `bv_concat_def`, `bv_bitwise`, `bv_nat`, `empty_bv`, and `bv_caret`.

`finite_sets_of_sets`
: Proves that the powerset of a finite set is finite, and provides the corresponding judgement.

`equivalence classes`
: The following theories were derived from those provided by Bart Jacobs: `EquivalenceClosure`, `QuotientDefinition`, `KernelDefinition`, `QuotientKernelPropertie`
`QuotientSubDefinition`, `QuotientExtensionProperties`, `QuotientDistributive`,
and `QuotientIteration`.

`Partial Functions`
: Bart Jacobs also provided definitions for partial functions: `PartialFunctionDefinitions`
and `PartialFunctionComposition`.

2.18.2 New Declarations

The following declarations have been added to the prelude: `relations.equivalence type`, `sets.setofsets`, `sets.powerset`, `sets.Union`, `sets.Intersection`, `sets_lemmas.subset_powerset`, `sets_lemmas.empty_powerset`, `sets_lemmas.nonempty_powerset`, `real_props.div_cancel4`, and `rational_props.rational_pred_ax2`.

2.18.3 Modified Declarations

The following declarations have been modified. `finite_sets.is_finite_surj` was turned into an IFF and extended from posnat to nat.

The fixpoint declarations of the `mucalculus` theory have been restricted to monotonic predicates. This affects the declarations `fixpoint?`, `lfp`, `mu`, `lfp?`, `gfp`, `nu`, and `gfp?`.

2.19 Conversion Expressions

Conversions may now be any function valued expression, for example,

```
CONVERSION+ EquivClass(ce), lift(ce), rep(ce)
```

This introduces a possible incompatibility if the following declaration is for an infix operator. In that case the conversion must be followed with a semi-colon ';'.

2.20 Judgement TCC proofs

Judgement TCCs may now be proved directly, without having to show the TCCs using `M-x show-tccs` or `M-x prettyprint-expanded`. Simple place the cursor on the judgement, and run one of the proof commands. Note that there may be several TCCs associated with the judgement, but only one of them is the judgement TCC. To prove the others you still need to show the TCCs first.

2.21 PVS Startup Change

On startup, PVS no longer asks whether to create a context file if none exists, and if you simply change to another directory no `.pvscontext` file is created. This fixes a subtle bug in which typing input before the question is asked caused PVS to get into a bad state.

2.22 Dump File Change

The `M-x dump-pvs-files` command now includes PVS version information, Allegro build information, and prelude library dependencies. Note that since the proof files have changed, the dumps may look quite different. See the Multiple Proofs section for details.

2.23 Bitvector Library

Bart Jacobs kindly provided some additional theories for the bitvector library. These were used as an aid to Java code verification, but are generally useful. The new files are `BitvectorUtil`, `BitvectorMultiplication`, `BitvectorMultiplicationWidenNarrow`, `DivisionUtil`, `BitvectorOneComplementDivision`, `BitvectorTwoComplementDivision`, and `BitvectorTwoComplementDivisionWidenNarrow`, and are included in the libraries tar file.

3 Bug Fixes

Although there are still a number of bugs still outstanding, a large number of bugs have been fixed in this release. All those in the pvs-bugs list (`http://pvs.csl.sri.com/cgi-bin/pvs/pvs-bug-list/` that are marked as analyzed have been fixed, at least for the specific specs that caused the bugs.

4 Incompatibilities

Most of these are covered elsewhere, they are collected here for easy reference.

4.1 Improved Decision Procedures

The decision procedures are more complete. Though this is usually a good thing, some existing proofs may fail. For example, a given auto-rewrite may have worked in the past, but now the key term has been simplified and the rewrite no longer matches.

4.2 Prelude Changes

These are given in See Section 2.18 [Prelude Changes], page 10. Theory identifiers used in the prelude may not be used for library or user theories, some existing theories may need to be adjusted.

The theories `finite_sets`, `finite_sets_def`, and `card_def` were once a part of the `finite_sets` library, but have been merged into a single `finite_sets` theory and moved to the prelude. This means that the library references such as

```
IMPORTING finite_sets@finite_sets
IMPORTING fsets@card_def
```

must be changed. In the first case just drop the prefix, drop the prefix and change `card_def` to `finite_sets` in the second.

The `reals` theory was split in two, separating out the field axioms into the `number_fields` theory. There is the possibility that proofs could fail because of adjustments related to this, though this did not show up in our validations.

4.3 Theory Abbreviations

Theory abbreviations such as

```
foo: THEORY = bar[int, 3]
```

should be changed to the new form

```
IMPORTING bar[int, 3] AS foo
```

Note that '`AS`' is a new keyword, and may cause parse errors where none existed before.

4.4 Conversion Expressions

Since conversions may now be arbitrary function-valued expressions, if the declaration following is an infix operator it leads to ambiguity. In that case the conversion must be followed with a semi-colon '`;`'.

4.5 Occurrence numbers in `expand` proof command

Defined infix operators were difficult to expand in the past, as the left to right count was not generally correct; the arguments were looked at before the operator, which meant that the parser tree had to be envisioned in order to get the occurrence number correct. This bug has been fixed, but it does mean that proofs may need to be adjusted. This is another case where it helps to run an earlier PVS version in parallel to find out which occurrence is actually intended.

Short Contents

Table of Contents