PVS Release Notes

Sam Owre <owre@csl.sri.com>
SRI International
December 15, 2005

Introduction

The PVS release notes are given here, for each version, going back to version 3.0.

You can always download the latest version of PVS from
`http://pvs.csl.sri.com/download.html`.

Note that the release notes are now written in texinfo, and are thus available in Emacs info, HTML, Postscript, and PDF forms. `M-x pvs-release-notes` brings up the info files while in PVS. The others are available in the `doc/release-notes` subdirectory of the PVS distribution.

PVS 4.0 Release Notes

Release notes for PVS version 4.0.[1] The major difference from earlier versions of PVS is that this release is under the GPL license. In addition, there is now a PVS WIKI page at `http://pvs-wiki.csl.sri.com`.

Installation Notes

Generally, installation is the same as usual; but now there are more choices. To start with, the

However, if you have received patches from SRI that you have put into your `~/.pvs.lisp` file, they should be removed. If you anticipate wanting to try the newer and older versions together, you can do this by using `#-pvs4.0` in front of the patches. This is a directive to the Lisp reader, and causes the following s-expression to be ignored unless it is an earlier version of PVS.

New Features

Record and Tuple Type Extensions

Record and tuple types may now be extended using the `WITH` keyword. Thus, one may create colored points and moving points from points as follows.

```
point: TYPE = [# x, y: real #]
colored_point: TYPE = point WITH [# color: Color #]
moving_point: TYPE = point WITH [# vx, vy: real #]
```

Similarly, tuples may be extended:

```
R3: TYPE = [real, real, real]
R5: TYPE = R3 WITH [real, real]
```

For record types, it is an error to extend with new field names that match any field names in the base record type. The extensions may not be dependent on the base type, though they may introduce dependencies within themselves.

```
dep_bad: TYPE = point WITH [# z: {r: real | x*x + y*y < 1} #]
dep_ok: TYPE = point WITH [# a: int, b: below(a) #]
```

Note that the extension is a type expression, and may appear anywhere that a type is allowed.

Structural Subtypes

PVS now has support for structural subtyping for record and tuple types. A record type `S` is a structural subtype of `R` if every field of `R` occurs in `S`, and similarly, a tuple type `T` is a structural subtype of a tuple formed from some prefix of `T`. Section [Record and Tuple Type Extensions], page 3 gives examples, as `colored_point` is a structural subtype of `point`, and `R5` is a structural subtype of `R3`. Structural subtypes are akin to the class hierarchy of object-oriented systems, where the fields of a record can be viewed as the slots

[1] These started as the release notes for PVS 3.3, but this was changed to a major release when we made PVS open source.

of a class instance. The PVS equivalent of setting a slot value is the override expression (sometimes called update), and this has been modified to work with structural subtypes, allowing the equivalent of generic methods to be defined. Here is an example.

```
points: THEORY
BEGIN
 point: TYPE+ = [# x, y: real #]
END points

genpoints[(IMPORTING points) gpoint: TYPE <: point]: THEORY
BEGIN
 move(p: gpoint)(dx, dy: real): gpoint =
  p WITH [`x := p`x + dx, `y := p`y + dy]
END genpoints

colored_points: THEORY
BEGIN
 IMPORTING points
 Color: TYPE = {red, green, blue}
 colored_point: TYPE = point WITH [# color: Color #]
 IMPORTING genpoints[colored_point]
 p: colored_point
 move0: LEMMA move(p)(0, 0) = p
END colored_points
```

The declaration for `gpoint` uses the structural subtype operator `<:`. This is analogous to the `FROM` keyword, which introduces a (predicate) subtype. This example also serves to explain why we chose to separate structural and predicate subtyping. If they were treated uniformly, then `gpoint` could be instantiated with the unit disk; but in that case the `move` operator would not necessarily return a `gpoint`. The TCC could not be generated for the `move` declaration, but would have to be generated when the `move` was referenced. This both complicates the typechecking, and makes TCCs and error messages more inscrutable. If you really want to deal with both, simply include a structural subtype followed by a (predicate) subtype, for example:

```
genpoints[(IMPORTING points) gpoint: TYPE <: point,
          spoint: TYPE FROM gpoint]: THEORY
```

Empty and Singleton Record and Tuple Types

Empty and singleton record and tuple types are now allowed in PVS. Thus the following are valid declarations:

```
Tup0: TYPE = [ ]
Tup1: TYPE = [int]
Rec0: TYPE = [# #]
```

Note that the space is important in the empty tuple type, as otherwise it is taken to be an operator (the box operator).

PVSio

César Muñoz has kindly provided lisp code for PVSio, which has been incorporated into PVS. See the PVSio web page `http://research.nianet.org/~munoz/PVSio/` for more information.

Random Testing

We have developed a capability for random test generation in PVS, based, in part, on work done in Haskell and Isabelle. Random tests may be generated for universally quantified formulas in the ground evaluator or in the prover. In each case, the purpose is to try and find a counter example to the given formula, by evaluating a number of instances until one of them returns `FALSE`. The falsifying instance is then displayed.

This is a good way to test a specification before attempting a proof. Unlike model checking, it is inherently incomplete; on the other hand, there is no requirement for all types to be finite, only that all involved types and constants have interpretations.

In the ground evaluator, we added the `test` command:[2]

```
(test expr &optional (count 10) (size 100) (dtsize 10)
             all? verbose? instance)
```

For the prover, there is a similar rule:

```
(random-test &optional (fnum *) (count 10) (size 100)
             (dtsize 10) all? verbose? instance
             (subtype-gen-bound 1000))
```

The `count` argument controls how many random tests to try. The `size` and `dtsize` control the possible ranges of random values, as described below. Normally the tests stop when a counter example is found; setting the `all?` flag to `t` causes further tests to be run until `count` is reached. The `verbose?` flag indicates that all random test values should be displayed. This is often useful to understand why a given test seems to always be true. The `instance` argument allows formals and uninterpreted types and constants to be given as a theory instance with actuals and mappings. The current theory may also be instantiated this way. For example, `th[int, 0]{{T := bool, c := true}}` may be a theory instance, providing actuals and mappings for the terms involved in the given formula. The `subtype-gen-bound` is used to control how many random values to generate in attempting to satisfy a subtype predicate, as described below.

In the prover, the universal formula is generated from the formulas specified by the `fnum` argument, first creating an implication from the conjunction of antecedents to the disjunction of consequents. Any Skolem constants are then universally quantified and the result passed to the random tester. This is useful for checking if the given sequent is worth proving; if it comes back with a counter example, then it may not be worth trying to prove. Of course, it may just be that a lemma is needed, or relevant formulas were hidden, and that it isn't a real counter example.

The random values are generated per type. For numeric types, the builtin Lisp `random` function is used:

- `nat` uses random(0..`size`)

---
[2] This is not a prover command; the optional arguments are as in Lisp, i.e., positional, not keyword arguments.

- `int` uses random(`-size`..`size`)
- `rat` creates two random `int`s, the second nonzero, and returns the quotient
- `real` and above just use `rat` values

All other subtypes create a random value for the supertype, and then check if it satisfies the subtype predicate. It stops after `subtype-gen-bound` attempts. Higher-order subtypes such as `surjective?` are not currently supported. Function types generate a lazy function, so that, e.g.,

```
FORALL (f: [int -> int], x, y, z: int):
  f(x) + f(f(y)) > f(f(f(z)))
```

creates a function that memoizes its values. Other types (e.g., record and tuple types) are built up recursively from their component types.

Datatypes are controlled by `dtsize`. For example, with `size` and `dtsize` set to their defaults (`100` and `10`, respectively), a variable of type `list[int]` will generate lists of length between 0 and 10, with integer values between -100 and 100.

Yices

New prover commands are available that invoke the Yices SMT solver. See `http://yices.csl.sri.com` for details on Yices and its capabilities. You must download Yices from there and include it in your `PATH`, as it is not included with PVS. You will get a warning on starting PVS if Yices is not found in your path, but this can be ignored if you will not be using Yices.

The `yices` rule is an endgame solver; if it does not prove the (given formulas of) the sequent, it acts as a `skip`. In addition to the primitive `yices` rule, the strategies `yices-with-rewrites` and `ygrind` have been added. Use `help` (e.g., `(help ygrind)`) for details.

Recursive Judgements TCCs

Judgements on recursive functions often lead to difficult proofs, as one generally has to prove the resulting obligation using tedious induction. For example, here is a definition of append on lists of integer, and a judgement that it is closed on lists of natural numbers (note that this example is artificial; append is defined polymorphically in the prelude):

```
append_int(l1, l2: list[int]): RECURSIVE list[int] =
   CASES l1 OF
     null: l2,
     cons(x, y): cons(x, append_int(y, l2))
   ENDCASES
   MEASURE length(l1)

append_nat: JUDGEMENT append_int(a, b: list[nat]) HAS_TYPE list[nat]
```

This yields the TCC

```
append_nat: OBLIGATION
  FORALL (a, b: list[nat]): every[int]({i: int | i >= 0})(append_int(a, b));
```

Which is difficult to prove automatically (or even manually).

By adding the keyword `RECURSIVE` to the judgement, the TCCs are generated by

- creating the predicate on the top-level call to the function, in this case `every({i: int | i >= 0})(append_int(a, b))`.
- substituting the variables into the body of the recursive definition
- typechecking the substituted body against the expected result type (`list[nat]`), with the predicate as a condition.

With these changes, the TCC becomes

```
append_nat_TCC1: OBLIGATION
  FORALL (a, b: list[nat], x: int, y: list[int]):
    every({i: int | i >= 0})(append_int(a, b)) AND a = cons(x, y) IMPLIES
     every[int]({i: int | i >= 0})(cons[int](x, append_int(y, b)));
```

and this is easily discharged automatically (e.g., with `grind`).

Note that recursive judgements are used in exactly the same way as the non-recursive form; the only difference is in the generated TCCs.

Recursive judgements are only allowed on recursive functions, and they are only for closure conditions (i.e., arguments must be provided). If a non-recursive judgement is given where a recursive judgement would apply, then a warning is output. In general, recursive judgements are preferred. In fact, we considered making it the default behavior, but found that it was not backward compatible.

Prelude Additions

To support the Yices interface, several operators from the bitvector library have been moved to the prelude. These are in the new theories `floor_div_props`, `mod`, `bv_arith_nat_defs`, `bv_int_defs`, `bv_arithmetic_defs`, and `bv_extend_defs`. The `floor_div_props` and `mod` theories have been moved completely, the rest have only had the operators added to the prelude - the rest of the theory, along with lemmas and other useful declarations, is still in the bitvector library - just drop the `_def` for the corresponding theory.

Note that this can have some side effects. For example, the WIFT tutorial `adder` example expects conversions to be used in a certain way because there were no arithmetic operators on bit vectors. Now that there are such operators, conversions no longer are needed, and proofs obviously fail.

Decimal Representation for Numbers

PVS now has support for decimal representation of numbers, for example, `3.1416`. Internally, this is treated as a fraction, in this case `31416/10000`. So there is no floating point arithmetic involved, and the results are exact, since Common Lisp represents fractions exactly. The decimal representation must start with an integer, i.e., `0.007` rather than `.007`.

Unary +

The `+` operator may now be used as a unary operator. Note that there is no definition for unary `+`, for example, `+1` will lead to a type error. This was added primarily for user declarations.

Bug Fixes

This version fixes many (though not all) bugs. Generally those marked as `analyzed` in the PVS bugs list have been fixed, and most have been incorporated into our validation suite.

Incompatibilities

There were some improvements made to judgements and TCC generation, that in some cases lead to different forms of TCCs. In the validation suite, these were all easily detected and the proofs were not difficult to repair.

It was noted in bug number 920 that the instantiator only looks for matches within the sequent, though often there are matches from the Skolem constants that are not visible. The `inst?` command was modified to look in the Skolem constants as a last resort, so earlier proofs would still work. Unfortunately, `grind` and similar strategies use `inst?` eagerly, and may now find a Skolem constant match that is incorrect, rather than waiting for a better match after further processing. This is exactly the problem that `lazy-grind` was created for. In our validation suite only a few formulas needed to be repaired, and those generally could be fixed simply by replacing `grind` by `lazy-grind`. Since hidden Skolem constants are difficult for a new user to deal with, we feel that this is a worthwhile change.

PVS 3.2 Release Notes

PVS 3.2 contains a number of enhancements and bug fixes.

Installation Notes

Installation is the same as usual. However, if you have received patches from SRI that you have put into your `~/.pvs.lisp` file, they should be removed. If you anticipate wanting to try the newer and older versions together, you can do this by using `#-pvs3.2` in front of the patches. This is a directive to the Lisp reader, and causes the following s-expression to be ignored unless it is an earlier version of PVS.

New Features

Startup Script Update

The PVS startup script `pvs` has been made to work with later versions of Linux (i.e., RedHat 9 and Enterprise).

Theory Interpretation Enhancements

There are a number of changes related to theory interpretations, as well as many bug fixes.

There is now a new form of mapping that makes it simpler to systematically interpret theories. This is the *Theory View*, and it allows names to be associated without having to directly list them. For example, given a theory of timed automaton:

```
automaton:THEORY
BEGIN
  actions: TYPE+;
  visible(a:actions):bool;
  states: TYPE+;
  enabled(a:actions, s:states): bool;
  trans(a:actions, s:states):states;
  equivalent(a1, s2:states):bool;
  reachable(s:states):bool
  start(s:states):bool;
END automaton
```

One can create a `machine` with definitions for `actions`, etc., and create the corresponding interpretation simply by typing

```
IMPORTING automaton :-> machine
```

This is read as a *machine viewed as an automaton*, and is equivalent to

```
IMPORTING machine
IMPORTING automaton {{ actions := machine.actions, ... }}
```

Here the theory view was in an importing, but it is really a theory name, and hence may be used as part of any name. However, the implicit importing of the target is done only for theory declarations and importings. In all other cases, the instance needed must already be imported. Thus it is an error to reference

```
automaton :-> machine.start(s)
```

unless `machine` has already been imported. This is not very readable,[1] so it is best to introduce a theory abbreviation:

```
IMPORTING automaton :-> machine AS M1a
```

or a theory declaration:

```
M1t: THEORY = automaton :-> machine
```

The difference is that `M1a` is just an abbreviation for an instance of an existing theory, whereas `M1t` is a new copy of that theory, that introduces new entities. Thus consider

```
IMPORTING automaton :-> machine AS M2a
M2t: THEORY = automaton :-> machine
```

The formula `M1a.actions = M2a.actions` is type correct, and trivially true, whereas `M1t.actions = M2t.actions` is not even type correct, as there are two separate `actions` declarations involved, and each of those is distinct from `machine.actions`.

The grammar for *Name* and *TheoryName* has been changed to reflect the new syntax:

```
TheoryName := [Id ’]́ Id [Actuals] [Mappings] [’:->’ TheoryName]

Name := [Id ’]́ IdOp [Actuals] [Mappings] [’:->’ TheoryName] [’.’ IdOp]
```

The left side of `:->` is called the *source*, and the right side is called the *target*. Note that in this case the target provides a *refinement* for the source.

For a given theory view, names are matched as follows. The uninterpreted types and constants of the target are collected, and matched to the types and constants of the source. Partial matching is allowed, though it is an error if nothing matches. After finding the matches, the mapping is created and typechecked.

References to Mapped Entities

Mapping an entity typically means that it is not accessible in the context. For example, one may have

```
IMPORTING T{{x := e}} AS T1
```

where the *e* is an expression of the current context. The `x`, having been mapped, is not available, but it is easy to forget this and one is often tempted to refer to `T1.x`. One possible work-around is to use theory declarations with `=` in place of `:=`, but then a new copy of `T` will be created, which may not be desirable (or in some cases even possible - see the Theory Interpretations Report).

To make mappings more convenient, such references are now allowed. Thus in a name of the form `T1.x`, `x` is first looked for in `T1` in the usual way, but if a compatible `x` cannot be found, and `T1` has mappings, then `x` is searched for in the left sides, and treated as a macro for the right side if found. Note that `x` by itself cannot be referenced in this way; the theory name must be included.

[1] Parentheses seem like they would help, but it is difficult to do this with the current parser.

Cleaning up Specifications

Developing specifications and proofs often leads to the creation of definitions and lemmas that turn out not to be necessary for the proof of the properties of interest. This results in specifications that are difficult to read. Removing the unneeded declarations is not easy, as it is difficult to know whether they are actually used or not.

The new commands `unusedby-proof-of-formula` and `unusedby-proofs-of-formulas` facilitate this. The `unusedby-proof-of-formula` command creates a '`Browse`' buffer listing all the declarations that are unused in the proof of the given formula. Removing all these declarations and those that follow the given formula should give a theory that typechecks and for which the proofchain is still complete, if it was in the full theory. This could be done automatically in the future.

Binary Files

PVS specifications are saved as binary (`.bin`) files, in order to make restarting the system faster. Unfortunately, it often turned out that loading them caused problems. This was handled by simply catching any errors, and simply retypechecking. Thus in many cases the binary files actually made things slower.

Until PVS version 3.2, binary files corresponded to the specification files. This means that if there is a circularity in the files (i.e., theories `A` and `C` are in one file, `B` in another, with `A` importing `B` importing `C`) then there is no way to load these files. In 3.2, bin files correspond to theories. These are kept in a `pvsbin` subdirectory of the current context.

However, there was a more serious problem with the binary files. It turns out that loading a binary file took more space, and the proofs took longer to run. The reason for this is that the shared structure that is created when typechecking sources is mostly lost when loading binary files. Only the structure shared within a given specification file was actually shared. In particular, types are kept in canonical form, and when shared, testing if two types are equal or compatible is much faster.

The binary files are now saved in a way that allows the shared structure to be regained. In fact, there is now more sharing than obtained by typechecking. This is one of the main reasons that this release took so long, as this forced many new invariants on the typechecker.

The payoff is that, in general, binary files load around five times faster than typechecking them, and proofs run a little faster because of the increased sharing. This is based on only a few samples, in the future we plan on systematically timing the specifications in our validation suite.

Generating HTML

The commands `html-pvs-file` and `html-pvs-files` generate HTML for PVS specification files. These can be generated in place, or in a specified web location. This is driven by setting a Lisp variable `*pvs-url-mapping*`, as described below.

The in place version creates a `pvshtml` subdirectory for each context and writes HTML files there. This is done by copying the PVS file, and adding link information so that comments and whitespace are preserved. Note that there is no `html-theory` command. This is not an oversight; in creating the HTML file links are created to point to the declarations of external HTML files. Hence if there was as way to generate HTML corresponding to both theory and PVS file, it would be difficult to decide which a link should refer to.

HTML files can be generated in any order, and may point to library files and the prelude. Of course, if these files do not exist then following these links will produce a browser error. The `html-pvs-files` command will attempt to create all files that are linked to, failure is generally due to write permission problems.

Usually it is desirable to put the HTML files someplace where anybody on the web can see them, in which case you should set the `*pvs-url-mapping*` variable. It's probably best to put this in your `~/.pvs.lisp` file in your home directory so that it is consistently used. This should be set to a list value, as in the following example.

```
(setq *pvs-url-mapping*
      '("http://www.csl.sri.com/~owre/"
        "/homes/owre/public_html/"
        ("/homes/owre/pvs-specs" "pvs-specs" "pvs-specs")
        ("/homes/owre/pvs3.2" "pvs-specs/pvs3.2" "pvs-specs/pvs3.2")
        ("/homes/owre/pvs-validation/3.2/libraries/LaRC/lib"
         "pvs-specs/validation/nasa"
         "pvs-specs/validation/nasa")))
```

The first element of this list forms the base URL, and is used to create a `<base>` element in each file. The second element is the actual directory associated with this URL, and is where the `html-pvs-file` commands put the generated files. The rest of the list is composed of lists of three elements: a specification directory, a (possibly relative) URL, and a (possibly relative) HTML directory. In the above example, the base URL is `http://www.csl.sri.com/~owre/`, which the server associates with `/homes/owre/public_html`. The next entry says that specs found in (a subdirectory of) `/homes/owre/pvs-specs` are to have relative URLs corresponding to `pvs-specs`, and relative subdirectories similarly. Thus a specification in `/homes/owre/pvs-specs/tests/conversions/` will have a corresponding HTML file in `/homes/owre/public_html/pvs-specs/test/conversions/` and correspond to the URL `http://www.csl.sri.com/~owre/pvs-specs/test/conversions/`. In this case, PVS is installed in `/homes/owre/pvs3.2`, and thus references to the prelude and distributed libraries (such as finite sets), will be mapped as well. Note that in this example, all the relative structures are the same, but it doesn't have to be that way.

The `*pvs-url-mapping*` is checked to see that the directories all exist, though currently no URLs are checked (if anybody knows a nice way to do this from Lisp, please let us know). If a subdirectory is missing, the system will prompt you for each subdirectory before creating it. A `n` or `q` answer terminates processing without creating the directory, a `y` creates the directory and continues, and a `!` causes it to just create any needed directories without further questions.

If a `*pvs-url-mapping*` is given, it must be complete for the file specified in the `html-pvs-file` command. In practice, this means that your PVS distribution must be mapped as well. PVS will complain if it is not complete; in which case simply add more information to the `*pvs-url-mapping*` list.

No matter which version is used, the generated HTML (actually XHTML) file contains a number of `<span>` elements. These simply provide a way to add `class` attributes, which can then be used in Cascading Style Sheet (CSS) files to define fonts, colors, etc. The classes currently supported are:

```
span.comment
span.theory
span.datatype
span.codatatype
span.type-declaration
span.formal-declaration
span.library-declaration
span.theory-declaration
span.theory-abbreviation-declaration
span.variable-declaration
span.macro-declaration
span.recursive-declaration
span.inductive-declaration
span.coinductive-declaration
span.constant-declaration
span.assuming-declaration
span.tcc-declaration
span.formula-declaration
span.judgement-declaration
span.conversion-declaration
span.auto-rewrite-declaration
```

See the `<PVS>/lib/pvs-style.css` file for examples. This file is automatically copied to the base directory if it doesn't already exist, and it is referenced in the generated HTML files. Most browsers underline links, which can make some operators difficult to read, so this file also suppresses underlines. This file may be edited to suit your own taste or conventions.

Both the `html-pvs-file` commands take an optional argument. Without it, many of the common prelude operators are not linked to. With the argument all operators get a link. Overloaded operators not from the prelude still get links.

Default Strategies

There is now a `default-strategy` that is used by the prover for the prove-using-default commands, and may be used as a parameter in pvs-strategies files. For example, the `pvs-strategies` file in the home directory may reference this, which is set to different values in different contexts.

Better handling of TCCs in Proofs

While in the prover, the typechecker now checks the sequent to see if the given expression needs to have a TCC generated. It does this by examining the formulas of the sequent, to see if the given expression occurs at the top level, or in a position from which an unguarded TCC would be generated. Thus if `1/x` appears in the sequent in an equation `y = 1/x`, the TCC `x /= 0` will not be generated. But if the expression only appears in a guarded formula, for example, `x = 0 IMPLIES y = 1/x`, then the TCC will still be generated.

This is sound, because for the expression to appear in the sequent necessary TCCs must already have been generated. This greatly simplifies proofs where annoying TCCs pop up over and over, and where the judgment mechanism is too restrictive (for example, judgements cannot currently state that `x * x >= 0` for any real `x`).

Obviously, this could affect existing proofs, though it generally makes them much simpler.

`typepred!` rule and `all-typepreds` strategy

Any given term in the sequent may have associated *implicit type constraints*. When a term is first introduced to a sequent there may be TCCs associated, either on the formula itself, or as new branches in the proof. The term may subsequently be rewritten, but there is still associated with the term an implicit TCC. For example, the term `1/f(x)` may be introduced, and later simplified to `1/(x * x - 1)`. Since `f(x)` was known to be nonzero, it follows that `x * x - 1` is also nonzero (in this context), though this is not reflected in the types or judgements.

The `typepred!` rule has been modified to take a `:implicit-typepreds?` argument, which looks for occurrences of the given expression in the sequent, and creates the implicit type constraint (if any) as a hypothesis. It does this only for occurrences that are *unguarded*, i.e., occur positively. This is stricter than the way TCCs are actually generated. This is needed because, for example, conjunction is commutative, and can be rewritten in the prover. Thus the hypothesis `x /= 0 => 1/x /= x` could be rewritten to `1/x = x => x = 0`, and the left-to-right reading will generate `x /= 0`, which is obviously unsound. Note that this does not mean that TCC generation or applying the rewrite is unsound, as the TCC simply says that a type can be assigned to the term. Technically, a TCC for a term of the form `A => B` could be a disjunction `(A => TCC(B)) OR (NOT B => TCC(A))`, but this is more costly in many ways, and rarely useful in practice.

Thus the command `(typepred! "x * x - 1" :implicit-typepreds? t)` generates the hypothesis `x * x - 1 /= 0` assuming that the term occurs positively in a denominator.

A generally more useful strategy is `all-typepreds`. This collects the implicit type constraints for each subexpression of the specified formula numbers. This can be especially handy for automating proofs, though there is the potential of creating a lot of irrelevant hypotheses.

grind-with-ext and reduce-with-ext

There are two new prover commands: `grind-with-ext` and `reduce-with-ext`. These are essentially the same as `grind` and `reduce`, but also perform extensionality. This is especially useful when reasoning about sets.

New forward chain commands

There are new forward chain commands available: `forward-chain@`, `forward-chain*`, and `forward-chain-theory`. `forward-chain@` takes a list of forward-chaining lemmas (of the form `A1 & ... & An => B`, where free variables in `B` occur among the free variables in the `Ai`), and attempts the forward-chain rule until the first one succeeds. `forward-chain*` takes a list, and repeatedly forward-chains until there is no change; when successful it starts back at the beginning of the list. `forward-chain-theory` creates a list of the applicable lemmas of the given theory and invokes `forward-chain*`.

TeX Substitutions

TeX substitutions have been improved, allowing substitutions to be made for various delimiters, as shown below. The TeX commands are defined in the `pvs.sty` file at the top level of the PVS directory. They consist of the prefix, followed by 'l' or 'r' to indicate the left or right delimiter.

Name	Symbols	TeX Command Prefix	TeX
parentheses	`( )`	`\pvsparen`	( )
brackets	`[ ]`	`\pvsbracket`	[ ]
record type constructors	`[# #]`	`\pvsrectype`	[# #]
bracket bar	`[| |]`	`\pvsbrackvbar`	⟦ ⟧
parenthesis bar	`(| |)`	`\pvsparenvbar`	⦇ ⦈
brace bar	`{| |}`	`\pvsbracevbar`	{[ ]}
list constructor	`(: :)`	`\pvslist`	⟨ ⟩
record constructor	`(# #)`	`\pvsrecexpr`	(# #)

These can be customized either by including new mappings for the symbols in a `pvs-tex.sub` file, or by overriding the TeX commands in your LaTeX file. It may be useful to look at the default `pvs.sty` and `pvs-tex.sub` files; both are located in the top level of the PVS installation (provided by `M-x whereis-pvs`).

add-declaration and IMPORTINGs

The `add-declaration` command now allows IMPORTINGs. This is most useful during a proof when a desired lemma is in a theory that has not been imported. Note that it is possible for the file to no longer typecheck due to ambiguities after this, even though the proof will go through just fine. Such errors are typically very easy to repair.

Prelude additions

Although no new theories have been added, there are a number of new declarations, mostly lemmas. These are in the theories `sets`, `function_inverse`, `relation_defs`, `naturalnumbers`, `reals`, `floor_ceil`, `exponentiation`, and `finite_sets`.

The `bv_cnv` theory was removed, as the conversion can sometimes hide real type errors. To enable it, just add the following line to your specification.

```
CONVERSION fill[1]
```

Bug Fixes

The PVS Bugs List (`http://www.csl.sri.com/cgi-bin/pvs/pvs-bug-list/`) shows the status of reported bugs. Not all of these have been fixed as of PVS version 3.2. Those marked `feedback` or `closed` are the ones that have been fixed. The more significant bug fixes are described in the following subsections.

Retypechecking

PVS specifications often span many files, with complex dependencies. The typechecker is lazy, so that only those theories affected by a change will need to be retypechecked. In addition, not all changes require retypechecking. In particular, adding comments or whitespace will cause the typechecker to reparse and compare the theories to see if there was a real change. If not, then the place information is updated and nothing needs to be retypechecked. Otherwise, any theory that depends on the changed theory must be untypechecked. This means that the typechecker cannot decide if something needs to be untypechecked until it actually reparses the file that was modified.

Thus when a file is retypechecked, it essentially skips typechecking declarations until it reaches an importing, at which point it retypechecks that theory. When it reaches a theory that has actually changed, untypechecking is triggered for all theories that import the changed theory. The bug was that only the top level theory was untypechecked correctly; any others would be fully untypechecked, but since they were already in the process of being typechecked, earlier declarations would no longer be valid.

The fix is to keep a stack of the theories being typechecked and the importing they are processing, and when a change is needed, the theories are only untypechecked after the importing.

Quantifier Simplification

In PVS 3.1, a form of quantifier simplification was added, so that forms such as `FORALL x: x = n IMPLIES p(x)` were automatically simplified to `p(n)`. In most cases, this is very useful, but there are situations where the quantified form is preferable, either to trigger forms of auto-rewriting or to allow induction to be used.

Many proof commands now include a `:quant-simp?` flag to control this behavior. By default, quantifier simplification is not done; setting the flag to `t` allows the simplification.

`simplify`, `assert`, `bash`, `reduce`, `smash`, `grind`, `ground`, `lazy-grind`, `crush`, and `reduce-ext` all have this flag.

Incompatibilities

Ground Decision Procedure Completeness

The decision procedures have been made more complete, which means that some proofs may finish sooner. Unfortunately, some proofs may also loop that didn't before[2]. This is usually due to division, and a workaround is to use the `name-replace` command to replace the term involving division with a new name, and then using the decision procedure (e.g., `assert`). If you find that the prover is taking too long, you can interrupt it with `C-c C-c`, and run `:bt` to see the backtrace. If it shows something like the following, then you know you are in the ground decision procedure.

```
find1 <-
  pr-find <- chainineqs <- transclosure <- addineq <- process1 <-
  ineqsolve <- arithsolve <- solve <- pr-merge <- process1 <-
  ineqsolve <- arithsolve <- solve
```

At this point, you can either run `(restore)` to try a different command (like `name-replace`), or `:cont` in the hope that it will terminate with a little more time. And yes, there are situations where the bug is not a problem, it just takes a long time to finish.

Actuals not allowed for Current Theory

In the past, a name could reference the actuals of the current theory. This is actually a mistake, as the actuals were generally ignored in this case. Though this rarely caused problems, there were a few reported bugs that were directly due to this, so now the system will report that the actuals are not allowed. To fix this, simply remove the actual parameters. Note that this can affect both specifications and proofs.

[2] There are some outstanding bugs reported on decision procedure loops that have not yet been resolved

Referencing Library Theories

In earlier versions of PVS, once a library theory was typechecked, it could be referenced without including the library id. This is no longer valid. First of all, if the given theory appears in two different libraries, it is ambiguous. Worse, if it also appears in the current context, there is no way to disambiguate. Finally, even if there is no ambiguity at all, there can still be a problem. Consider the following:

```
A: THEORY ... IMPORTING B, C ... END A

B: THEORY ... IMPORTING lib@D ... END B

C: THEORY ... IMPORTING D ... END C
```

This typechecks fine in earlier versions of PVS, but if in the next session the user decides to typecheck `C` first, a type error is produced.

Renaming of Bound Variables

This has been improved, so that variables are generally named apart. In some cases, this leads to proofs failing for obvious reasons (an inst variable does not exist, or a skolem constant has a different name).

`bddsimp` and Enumeration Types

Fixed bddsimp to return nicer formulas when enumeration types are involved. These are translated when input to the BDD package, but the output was untranslated. For example, if the enumeration type is `{a, b, c}`, the resulting sequents could have the form

```
a?(x)      b?(x)
|----      |----      |----
           a?(x)      b?(x)
                      a?(x)
```

With this change, instead one gets

```
a?(x)      b?(x)      c?(x)
|----      |----      |----
```

Which is nicer, and matches what is returned by prop. This makes certain proofs faster, because they can use the positive information, rather than the long and irrelevant negative information. Of course, the different formula numbering can affect existing proofs.

Prettyprinting Theory Instances

The `prettyprint-theory-instance` command was introduced along with theory interpretations, but it was restricted to theory instances that came from theory declarations, and would simply prettyprint these. Unfortunately, such theories are very restricted, as they may not refer to any local declarations. The `prettyprint-theory-instance` now allows any theory instance to be given, and displays the theory with actuals and mappings performed. This is not a real theory, just a convenient way of looking at all the parts of the theory instance.

Assuming and Mapped Axiom TCC Visibility Rules

The visibility rules for assumings and mapped axioms has been modified. Most TCCs are generated so that the entity that generated them is not visible in a proof. This is done simply by inserting the TCCs before the generating declaration. Assuming and Mapped Axiom TCCs are a little different, in that they may legitimately refer to declarations that precede them in the imported theory. To handle this, these TCCs are treated specially when creating the context. All declarations preceding the assuming or axiom that generated the TCC are visible in the proof of the TCC.

Replacing actuals including types

The `replace` prover command now does the replacement in types as well as expressions when the `:actuals?` flag is set. It is possible, though unlikely, that this could cause current proofs to fail. It is more likely that branches will be proved sooner.

`expand` Rule uses Full Name

When the `expand` rule was given a full name it would ignore everything but the id. This has been fixed, so that other information is also used. For this command, the name is treated as a pattern, and any unspecified part of the name is treated as matching anything. Thus `th.foo` will match `foo` only if it is from theory `th`, but will match any instance or mapping of `th`. `foo[int]` will match any occurrence of `foo` of any theory, as long as it has a single parameter matching `int`. The `occurrence` number counts only the matching instances.

This change is only going to affect proofs in which more than just an identifier is given to `expand`.

finite_sets min and max renamed

In theory `finite_sets_minmax` the functions `min` and `max` defined on the type parameter have been renamed to `fsmin` and `fsmax`, respectively. This was done because they are only used in the definitions of `min` and `max` over finite sets, and can cause ambiguities elsewhere.

induct no longer beta-reduces everything

There was a bug reported where `induct` was generating a large number of subgoals; this turned out to be due to the indiscriminate use of `beta`, which was intended to simplify newly added formulas but could also affect the conclusion and subsequent processing. To fix this, `beta` is now only applied to newly generated formulas. This may make some proofs fail, though generally they will be fixed simply by using `beta` after `induct`.

PVS 3.1 Release Notes

PVS 3.1 is primarily a bug fix release, there are no new features, although the prelude has been augmented. Some of the changes do affect proofs, though our experience is that only a few proofs need adjustment, and most of these were quite easy to recognize and fix.

The bugs that have been fixed in 3.1 are mostly those reported since December 2002. Some of these fixes are to the judgement and TCC mechanism, so may have an impact on existing proofs. As usual, if it is not obvious why a proof is failing, it is often easiest to run it in parallel on an earlier version to see where it differs.

Some of the differences can be quite subtle, for example, one of the proofs that quit working used `induct-and-simplify`. There were two possible instantiations found in an underlying `inst?` command, and in version 3.0 one of these led to a nontrivial TCC, so the other was chosen. In version 3.1, a fix to the judgement mechanism meant that the TCC was no longer generated, resulting in a different instantiation. In this case the proof was repaired using `:if-match all`.

Most of the other incompatibilities are more obvious, and the proofs are easily repaired. If you have difficulties understanding why a proof has failed, or want help fixing it, send it to PVS bugs (`pvs-bugs@csl.sri.com`).

Thanks to Jerry James (`jamesj@acm.org`), a number of new theories and declarations have been added to the prelude. Several judgments have been added. Remember that these generally result in fewer TCCs, and could affect proofs as noted above.

PVS 3.0 Release Notes

The PVS 3.0 release notes contain the features, bug fixes, and incompatibilities of PVS version 3.0 over version 2.4.

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 (`pvs-bugs@csl.sri.com`).

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.

New Features

There are a number of new features in PVS 3.0.

Allegro 6.2 port

PVS 3.0 has been ported to the case-sensitive version of Allegro version 6.2. This was done in order to be able to use the XML support provided by Allegro 6.2. 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 instances), and see if that helps. If not, send the strategies file to PVS Bugs (`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 not support mixed cases in strategy files.

Theory Interpretations

Theory interpretations are described fully in Theory Interpretations in PVS (`http://pvs.csl.sri.com/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, and there is an improved version that will be available shortly, built on the new equivalence class definition in the prelude.

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

Multiple Proofs

PVS now supports multiple proofs for a given formula. When a proof attempt is ended, 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 new proof contains 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.

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 with the library, for example, 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.

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 by 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 of its use 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.

Coinduction

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

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: int, b: {z: (even?) | z > a}, c: int): c1?
 c2(a: int, b: {n: nat | n > a}, c: int): c2?
END dt

datatype_update: THEORY
BEGIN
```

```
  IMPORTING dt
  x: dt
  y: int
  f: dt = x WITH [b := y]
 END datatype_update
```

This generates the TCC

```
f_TCC1: OBLIGATION
     (c1?(x) AND even?(y) AND y > a(x))
  OR (c2?(x) AND y >= 0 AND y > a(x));
```

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`.

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.

Conversion Messages

Messages related to conversions have been separated out from the warnings, 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.

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.

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.

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 didn't get confused, most readers would.

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.

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.

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.

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.

Prelude Changes in 3.0

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`,
  `QuotientKernelProperties`,
  `QuotientSubDefinition`,
  `QuotientExtensionProperties`,
  `QuotientDistributive`, and
  `QuotientIteration`.

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

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`.

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?`.

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 '`;`'.

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.

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.

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.

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.

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/?bugs=open&bugs=analyzed) that are marked as analyzed have been fixed, at least for the specific specs that caused the bugs.

Incompatibilities

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

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.

Prelude Incompatibilities

These are given in Prelude Changes in 3.0. 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.

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.

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 ';'.

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

Introduction 1
PVS 4.0 Release Notes 3
PVS 3.2 Release Notes 9
PVS 3.1 Release Notes 19
PVS 3.0 Release Notes 21

Table of Contents

Introduction **1**

PVS 4.0 Release Notes **3**

Installation Notes 3
New Features 3
Record and Tuple Type Extensions 3
Structural Subtypes 3
Empty and Singleton Record and Tuple Types 4
PVSio 4
Random Testing 5
Yices 6
Recursive Judgements TCCs 6
Prelude Additions 7
Decimal Representation for Numbers 7
Unary `+` 7
Bug Fixes 8
Incompatibilities 8

PVS 3.2 Release Notes **9**

Installation Notes 9
New Features 9
Startup Script Update 9
Theory Interpretation Enhancements 9
References to Mapped Entities 10
Cleaning up Specifications 11
Binary Files 11
Generating HTML 11
Default Strategies 13
Better handling of TCCs in Proofs 13
`typepred!` rule and `all-typepreds` strategy 14
grind-with-ext and reduce-with-ext 14
New forward chain commands 14
TeX Substitutions 14
add-declaration and IMPORTINGs 15
Prelude additions 15
Bug Fixes 15
Retypechecking 15
Quantifier Simplification 16
Incompatibilities 16
Ground Decision Procedure Completeness 16
Actuals not allowed for Current Theory 16
Referencing Library Theories 17

Renaming of Bound Variables 17
bddsimp and Enumeration Types 17
Prettyprinting Theory Instances 17
Assuming and Mapped Axiom TCC Visibility Rules 18
Replacing actuals including types 18
expand Rule uses Full Name 18
finite_sets min and max renamed 18
induct no longer beta-reduces everything 18

PVS 3.1 Release Notes 19

PVS 3.0 Release Notes 21

Overview 21
New Features 21
Allegro 6.2 port 21
Theory Interpretations 21
Multiple Proofs 22
Better Library Support 23
Cotuples 23
Coinduction 24
Datatype Updates 24
Datatype Additions 25
Conversion Extensions 25
Conversion Messages 26
More TCC Information 26
Show Declaration TCCs 26
Numbers as Constants 26
Theory Search 26
Improved Decision Procedures 27
ICS Integration 27
LET Reduce 27
Prelude Changes in 3.0 27
New Theories 27
New Declarations 28
Modified Declarations 28
Conversion Expressions 28
Judgement TCC proofs 28
PVS Startup Change 28
Dump File Change 29
Bitvector Library 29
Bug Fixes 29
Incompatibilities 29
Improved Decision Procedures 29
Prelude Incompatibilities 29
Theory Abbreviations 29
Conversion Expressions 30
Occurrence numbers in expand proof command 30