PVS Release Notes

Sam Owre <owre@csl.sri.com>
SRI International
October 5, 2018

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.shtml`.

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 7.1 Release Notes

PVS 7.1 introduces several new features and bug fixes. Some of the highlights include an improved installation (PVS is no longer a "tar bomb"), the concept of workspaces, improvements to the XML-RPC interface, some language changes, an improved tracking for TCC changes, improvements to strategy definitions, improvements to theory interpretations, and inclusion of Yices. There are many other changes not mentioned here; it is worth skimming through the rest of this chapter to see which are useful to you.

Installation Notes

The system is most easily installed by getting a tar file from `http://pvs.csl.sri.com/download.shtml`. Untar it in a directory of your choice (it will create a subdirectory), cd to the subdirectory, and run `sh install.sh`. This sets the path in various scripts, so that PVS knows where it is. It also byte-compiles the emacs source for your Emacs. If you move the PVS directory, or change your Emacs, or forgot whether you ran it already, then simply rerun it.

We strongly suggest getting a pre-built Allegro version, unless you have concerns with the Allegro runtime click-though license, in which case get one of the SBCL Lisp images. It is possible to build from sources, but it can be sensitive to the platform environment. If you decide to try it and run into problems, let us know at `pvs-bugs@csl.sri.com`. PVS 7.1 is built with Allegro CL 10.0 and SBCL 1.0.47. It is available for Linux 64-bit machines, and Mac 64-bit.

Note that the XML-RPC server is not (yet) available in SBCL.

The latest PVS includes a preliminary version of a PVS client, written in wxPython. To use, you will need to have Python 2.7, along with `wx` (`http://www.wxpython.org/`), `pyparsing` (`http://pyparsing.wikispaces.com/`), and optionally `jsonschema` (`https://github.com/Julian/jsonschema`) and `requests` (`http://www.wxpython.org/`). If missing, the latter will give warnings - these can be ignored. These packages may often be gotten through Linux package managers, or using easy_install or pip.

New Features

PVS Contexts, Workspaces, and Libraries

Up to now PVS has had two notions of context, one which relates to directories (called a *PVS context*), and the other the usual list of declarations of type theory. The former are now referred to as workspaces, which are just directories with some sort of PVS file in them.

As with directories in a shell, there is a current workspace; other workspaces may be referenced as libraries. Within PVS names, the library is referred to by an identifier, and this is resolved to a workspace in one of three ways, in order:

- there is a visible library declaration with the given id. Note that this only makes sense in a declaration context, e.g., when typechecking, proving, etc.
- the id is a subdirectory of the current workspace.
- the id appears as a subdirectory in one of the directories of the `PVS_LIBRARY_PATH` environment variable.

Note that the second resolution is sensitive to what the current workspace is; in previous PVS versions this was implemented in such a way that the current workspace was treated in a special way, and changing workspaces meant "throwing away" much of the typechecking, etc. results.

In PVS 7, the information associated with a workspace has been encapsulated. This means that the current workspace is not so important. It's more like the way a shell uses directories; even if you're in a directory, you can run commands on files in other directories, simply by giving a (relative or absolute) pathname. In the same way, you can now typecheck a file from any workspace, even if it has never been seen by PVS before. It temporarily becomes the current workspace, the command is run, and then the current workspace is restored.

PVS can continue to be used exactly as in earlier versions, but there are several advantages to this change:

- Speed: in the past, library theories were treated specially, because references might be relative. Special class instances were created, and changing workspaces meant clearing out typechecked forms, even when they were needed from the new workspace. In PVS 7, changing the current workspace has no such effect; all the work done in the previous workspace will be preserved.
- Convenience: there is no need to change context to typecheck, prove, etc. a PVS spec. These commands now allow the inclusion of a workspace path, and that workspace is temporarily made the current context.

Development details

The current workspace is in the Lisp global `*workspace-session*`. If the `*pvs*` Emacs buffer is not in the prover or ground-evaluator, you can use `(show *workspace-session*)` to see the contents: this is an instance of the `workspace-session` class, with slots

- `path`: the path associated with this workspace. This must be an absolute path, as returned by the Common Lisp function `truename`.
- `pvs-files`: this is a hash-table with key the file name string (without directory or the `.pvs` extension), and returns a list. The first element of the list is the file-date for when it was parsed, and the rest of the list is the theories that make up the file.
- `pvs-theories`: this is a hash-table with key the theory id (symbol), and simply returns the theory instance.
- `prelude-libs`: this is a list of the prelude extensions allowed by PVS. See `load-prelude-library` for details.
- `prelude-context`: this is a context used when prelude-libs is set, as a starting context for typechecking new theories.
- `lisp-files`: workspaces may have a `pvs-lib.lisp` file, that is automatically loaded when the workspace is referenced as a library.
- `subdir-alist`: this is a list of subdirs that may be used as library references from this workspace. Such subdirs must satisfy the rules for a PVS identifier, in particular, names with hyphens are not directly allowed (though a library declaration may reference them).
- `pvs-context`: this is a list that represents the `.pvscontext` file.

- `pvs-context-changed`: this is a flag indicating that the `.pvscontext` has changed, and it will be written at some point.
- `strat-file-dates`: this is a list of dates associated with the `pvs-strategies` files, first from the PVS directory, then from the user's home directory, and finally from the current workspace.
- `all-subst-mod-params-caches`: this is a cache used by the `subst-mod-params` function, and not interesting unless debugging that function.

Note that all of these slots were global variables in earlier versions of PVS; this is what is meant by *encapsulation*.

When PVS is started, it creates a workspace-session instance in `*workspace-session*`, reads the `.pvscontext` file, if it exists, and adds the workspace-session to the `*all-workspace-sessions*` list. When a file is parsed or typechecked in that workspace, the `pvs-files` and `pvs-theories` are updated accordingly.

To parse, typecheck, etc. a PVS file from a different directory, the macros `with-workspace` or `with-pvs-file` are used to temporarily change to the new workspace. If the associated workspace-session is already in `*all-workspace-sessions*` (based on the `path`), then it is simply used, otherwise a new one is created and added to `*all-workspace-sessions*`.

`with-workspace` *path* **&rest** *forms* [Macro]
simply takes a *path* (string or pathname), and temporarily makes that the current workspace and processes the *forms* returning the value of the last one.

`with-pvs-file` *vars pvsfileref body* [Macro]
takes a *pvsfileref*, which in general has the form `dir/file.pvs#theory`, though it can be as simple as a name. It is pulled apart, and if a *dir* is included, `with-workspace` is used, and the rest is split on the `#`, if it exists. *vars* is a list of names, and these are bound in order with the file, theory, and any others if they are provided (the idea is that there could be a `#formula` or `#place` following the `#theory`). The *body* is then executed with the names in *vars* bound accordingly. Note that one must be careful here; if *pvsfileref* is just a name, it may be referring to a file or a theory, depending on the context. Hence there is code such as that for `show-tccs`:

```
(defun show-tccs (theoryref &optional arg)
  (with-pvs-file (fname thname) theoryref
    (let* ((theory (get-typechecked-theory (or thname fname)))
           ...))))
```

Here a theory is expected, and if the `theoryref` is just a name, then `fname` is set to it, and `thname` is `nil`. This is why the `theory` is derived from `(or thname fname)`.

PVS Language Changes

- Prime is allowed in names, e.g., `x'`. Allowed anywhere except the first character, which must be alphabetic.
- (Co)Datatypes may now have ASSUMING parts and theory declarations in addition to IMPORTINGs, which may appear interleaved in the constructor declarations.
- More expression judgements are allowed. In earlier versions, the parser distinguished between expression judgements and the other kinds of judgements. This is still possible

for name and number judgements, but the typechecker is now needed in general to distinguish between application and expression judgements. The advantage is that it opens up expression judgements to use most any expression, though it must be kept in mind that the judgement mechanism uses a very simple match, in order to keep the typechecker relatively quick.

- IF can be given as an application, e.g., IF(b,a,c).
- LAMBDA expressions may be given a result type, e.g., `LAMBDA (x: int) -> int: f(x)`. This will assign the type `[int -> int]` to the LAMBDA expression, and generate a TCC on `f(x)` if necessary.
- Binary relations may be chained, e.g., `a < b /= c > 4`. Internally, this becomes `a < b & b /= c & c > 4`. Works for any binary relation between the same or different types.

TCC Formulas and Associated Proofs

TCCs are generated during typechecking for a given declaration, and during a proof for a given branch. During typechecking, the name of a given TCC is derived from the declaration, followed by a number; e.g., if the declaration is named `foo`, then the TCCs will be named `foo_TCC1`, etc. Some declarations have many TCCs, with nontrivial proofs. When such a declaration is modified, it can generate insert, remove, or otherwise renumber the TCCs, and then associate the wrong proofs to them.

The new approach keeps track of where the proof came from, creating a `tcc-origin` instance, with the following slots:

- `root`: the root name for the TCC, before the "_TCC#".
- `kind`: one of `termination-subtype`, `subtype`, `termination`, `well-founded`, `existence`, `assuming`, `mapped-definition-equality`, `mapped-axiom`, `cases`, `actuals`, `disjointness`, or `coverage`.
- `expr`: except for the `existence` TCCs, the expanded form of the expression that was the reason for the TCC.
- `type`: the type associated; e.g., for `subtype` TCCs, the expected type.

The `.prf` file has been extended to include the same information with TCC proofs. When a PVS file is typechecked, the proofs are read from the corresponding `.prf`. TCC proofs are no longer matched by TCC name, but instead the `kind` and `expr` are used. Those that match exactly are directly used, any remaining proofs are installed (in order) in the remaining TCCs. This is obviously not perfect, as the corresponding `expr`s may not match, depending on the changes made to a declaration.

PVS XML-RPC server

Introduction

The PVS GUI is an API for the Prototype Verification System (PVS). In the past, the PVS GUI was based on a modified version of the Emacs Inferior Lisp Mode (`http://ilisp.cons.org/`) interface. This generally works well, but there are some issues:

- Many new users of PVS are inexperienced not only in formal methods, but also in the use of Emacs, which has a steep learning curve
- Many new users have only been exposed to mouse-and-menu interfaces, such as Eclipse

- ILISP is no longer maintained, and has largely been replaced by Slime
- The interface to ILISP is not very flexible, making it difficult to extend with new features

For these reasons, we decided to create a new API for a PVS GUI. We have several constraints we want to satisfy:

- PVS should act as a server, with potentially many clients
- The interface should be "RESTful", in the same way the internet is, i.e., there are no heartbeats, simply requests that are answered by PVS
- There should be no restriction on the language used to implement a client

We started to create an Eclipse plugin for PVS, but found this to be difficult; there is really nothing in Eclipse to support things like proof windows, or the various popup buffers that PVS normally does through Emacs. Note that there is an `eclipse` subdirectory in the PVS Git sources, for anyone who wants to continue this work.

But we took a step back, and started fresh with wxPython, which so far has proved more flexible, and quicker for prototyping.

The basic architecture consists of a PVS server, with any number of clients. A client can make a request to the PVS URI, and PVS will return a response to that client. In addition, a client can start an XML-RPC server and include that URI with the request, which allows PVS to send requests to the client, e.g., to answer questions, provide file names, or simply get notifications.

In the long run, we expect to make Emacs an XML-RPC client as well, but for now, it uses the same ILISP interface. However, as each JSON method is defined (often based on the corresponding Emacs command), the same JSON will be returned to Emacs. This allows testing at the Emacs level, and provides an incremental way to move toward making Emacs an XML-RPC client.

Although PVS allows any number of clients, there is currently only one main PVS thread. This means that all clients would share the same proof session, etc. This may be useful for collaboration, or for switching between clients (i.e., different GUIs that provide different features). In the future we will explore the possibility of having separate threads associated with different clients, allowing different clients to simultaneously run different proofs, possibly in different contexts.

PVS provides an XML-RPC server when started with a `-port` value, e.g., `pvs -port 22334`, normally an unused port between 1024 and 65535. XML-RPC was chosen because it is supported by most modern languages, and we chose to implement the JSON-RPC 2.0 protocol within XML-RPC. Directly using JSON-RPC is possible, but it is not yet widely supported.

There is a single XML-RPC method provided by the PVS server, `pvs.request`, that takes a JSON-RPC request string, and an optional client URI, which is used to send requests to the client, providing a 2-way communication. Note that PVS does not keep the client URI after answering the request, thus clients may be killed and restarted at any time. In like manner, PVS can be restarted without needing to restart any clients, though it may be necessary to change context, retypecheck, etc. At the XML-RPC level, the return value includes the JSON-RPC response, the current PVS context, and the mode (`lisp`, `prover`,

or `evaluator`). Thus if a given client has changed the context and started a proof, that information is included in the next request from a different client.

We chose JSON as the data interchange format over XML since it is more compact, and supported by most languages. In addition, there is a JSON Schema available, which we use to describe the API.

Error handling is done as follows. When an XML-RPC request comes in, PVS sets up a condition handler to catch any errors that may happen as a result of processing the request. If the request is badly formed, for a nonexistent method, or if the JSON-RPC request does not include an id, then a response is returned of the form

```
{"xmlrpc-error": string, "mode": string, "context": string}
```

If the request is well formed and includes an id, the method is invoked under a new condition handler, and the normal JSON-RPC response is given. This means that errors are returned even if the JSON-RPC request is a notification (without an id). Of course, the client is free to ignore such errors.

PVS JSON-RPC methods

There are only a few methods currently supported by PVS; a lot of effort was needed to implement the infrastructure. In particular, the prover was not really designed for a different API, and it was necessary to create hooks for generating a JSON representation of the current goal of a given proof.

The methods currently supported are listed below. Note that details about the possible return values are in the JSON Schema provided with PVS.

`list-methods` [method]
: This method simply lists the currently available methods.

`list-client-methods` [method]
: As described above, PVS may provide information or make requests to the client. This method lists all the JSON-RPC requests that PVS will invoke if it is given a URI at the XML-RPC level. Currently it consists of `info`, `warning`, `debug`, `buffer`, `yes-no`, and `dialog`. The JSON Schema gives details about the format.

`help` [method]
: Gives help for any given method returned by `list-methods`.

`lisp` [method]
: Simply sends a string to be evaluated by the PVS lisp interpreter, and returns a string with the result. Certainly an aid to debugging, but may also be useful for other purposes.

`change-context` [method]
: Changes the current context as with the Emacs `change-context` command.

`typecheck` [method]
: Typechecks a specified file. This returns a list of theories, each of which includes the declarations of that theory, as well as their locations.

`names-info` [method]
: This is a new method; given a PVS file, it returns an array of PVS identifiers, their location, the associated declaration (as a string), and the file and location where the declaration can be found. This can be used by the client to provide information about a given identifier when the mouse is hovering over that identifier. Clicking on that identifier could bring up the corresponding file and location.

`reset` [method]
: This interrupts any running process, and resets the system to the state where no proof or ground evaluator sessions are running. This may not clear up low-level server/client problems, as those are on a separate thread and more difficult to reset. We're waiting for a situation where this is an issue.

`prove-formula` [method]
: Given a formula and a theory name, this starts an interactive proof. The result is the current goal consisting of a sequent and other fields, see the JSON schema for details.

`proof-command` [method]
: This sends the specified proof command to PVS, returning the current goal. Currently the proof needs to be started with `prove-formula`, though in principle any client (e.g., Emacs) could start the proof and a different client continue. It's possible for this to allow collaboration on a single proof.

GUI

As described above, the new GUI is built on wxPython. The executable is `pvs-gui` in the top level PVS directory. Starting it with no arguments creates a client with port 22335, and expects the PVS server to be at port 22334. Currently you have to start up both, but they can be started in either order, and if one crashes it can be restarted without directly affecting the other. For starting pvs include `-port 22334` with any other arguments you might want. In principle, pvs can be started with -raw, meaning no Emacs, but it is easier for debugging to have the Emacs ILISP interface available. Run `pvs-gui -h` for details on how to set the ports and debug levels.

The GUI configuration by default is in the file `PVS/python/src/pvside.cfg`, where `PVS` is the pvs installation directory. At startup, this file is read, then the `~/pvside.cfg` file is read, if it exists. This has the same syntax as the default file (with sections and attributes), but should include only those sections and attributes that you wish to overwrite. This includes things like the default ports, fonts, colors, etc.

Once started, it should be fairly easy to explore and find files, change context, typecheck, and start proofs. There is a Help menu that gives more details. This is very much an early prototype, and suggestions are welcome. Note that the sources are available on GitHub with the rest of PVS in the python subdirectory. Please let us know if you would like to get involved in development of the GUI.

Prover Emacs UI

The prover has been significantly modified to generate structures suitable for sending to the GUI. As a means to test this, a new capability was added to PVS Emacs, making use of the same JSON forms as those sent to the GUI. By default, PVS uses the old display,

simply printing the sequent in the `*pvs*` buffer, displaying the `Rule?` prompt, and reading the next prover command.

There are new proof displays available. These are new, and not well tested, please send feedback if you try them out. Keep in mind the distinction Emacs makes between frames, windows, and buffers. A frame is what most systems call a window; each frame can be moved around on the desktop, closed, resized, etc. Frames may be subdivided into windows, and each window displays a buffer. Note that buffers are there, even if they are not currently displayed; there are separate commands for listing buffers, killing buffers, etc.

There are 6 proof display styles available; `no-frame`, `0-frame`, `1-frame`, `2-frame`, `3-frame`, and `4-frame`. As you might guess, the names say how many frames are involved. `no-frame`, the default, works as in the past. The rest create separate windows and frames for different parts of a proof session: the current goal, the command input, the proof commentary, and optionally the proof script.

The `0-frame` uses the same frame as the PVS startup frame, and splits it into separate windows. The `1-frame` creates a new frame for this purpose. The `2-frame` puts the commentary in a separate frame, the `3-frame` puts the commentary and proof script in separate frames, and the `4-frame` puts all four parts in separate frames.

The commentary is used for the running commentary of a proof; information that is part of the proof session, but not really part of a given proof step. The command input is currently just a window into the `*pvs*` buffer, which still has the proof as before, even when displays are active. The sequent buffer has the feature that hovering the mouse over an identifier shows the corresponding declaration; this can be very helpful in proofs.

The different parts of the proof display have associated faces, and can be customized. Do `M-x customize` and search for `pvs` to find all customizable faces.

PVS Identifier Tooltips

A new feature of PVS, developed partly for the new GUI, is the ability to associate tooltips with each PVS identifier of a PVS file or proof sequent. These tooltips are only available in typechecked files. They are automatically available in the GUI after typechecking; in Emacs, run `M-x pvs-add-tooltips` in any typechecked buffer (including the prelude) and then move the mouse over identifiers in the buffer to see their types. Clicking middle takes you to the file with the cursor at the declaration.

Proof Command Definitions

The proof command facility has been revamped, primarily in the argument handling. This section is for those who write strategies.

Just to review, strategy definitions such as `defstep` have required, optional, and rest arguments, e.g.,

```
(defstep foo (a &optional b c &rest d) ...)
```

Invocations of `foo` require the first argument; if there is a second argument it is bound to `b`, a third argument to `c`, and any remaining arguments are bound to `d`. This is similar to Common Lisp, but in PVS the optional and rest arguments may also be given as keywords, so `foo` could be invoked as either of the equivalent forms

```
(foo 3 5 7 11 13)
(foo 3 5 :d (11 13) :c 7)
```

In order to add a new argument to a low-level command, (e.g., the `let-reduce?` flag was added to `assert`), then to make this available to other commands such as `grind` meant adding it and the corresponding documentation to those commands. This is obviously error-prone. Recently we wanted to add the `actuals?` argument of replace to `grind`, in order to allow `grind` to work in type and actual expressions. The problem is that `grind` invokes `replace*`, which has a `&rest fnums` argument; this does not allow new arguments to be added without modifying existing proofs.

To solve this immediate problem we added the `&key` indicator. It is similar to the `&optional` indicator, but the arguments must be provided as keywords. Hence `replace*` could now be rewritten from

```
(defstep replace* (&rest fnums) ...
```

to

```
(defstep replace* (&key actuals? &rest fnums) ...
```

Existing proofs would not break, but new proofs could invoke `replace*` with an `:actuals? t` argument to have replacement happen inside of types and actuals.

But this only solves part of the problem; propagating this argument to strategies such as `grind` is still error-prone. To deal with this, we added another indicator: `&inherit`. With this, `replace*` can be defined as

```
(defstep replace* (&rest fnums &inherit replace) ...)
```

And now replace* automatically inherits all keyword arguments from `replace`. Not only that, but any invocations of `replace` within the body of the defstep automatically include keyword invocations of the `replace` call. In effect, where the body was written simply as `(replace y)`, it is replaced in the actual command by

```
(replace y :dir dir :hide? hide? :actuals? actuals?
           :dont-delete? dont-delete?)
```

Note that this inherited not just the `actuals?` argument, but all the others as well. Note also that if a new argument is added to `replace`, it will be automatically inherited by `replace*`.

Future Work

There is still work to be done; currently optional and key arguments allow a default, but we want to in addition allow `:documentation` and `:kind` keywords, even for required arguments. The documentation will be used to document the arguments, rather than have them in the main documentation of the proof command. For optional and key arguments, this documentation will then propagate, so that, e.g., the documentation for `replace*` directly explains the `actuals?` argument, without having to look up `replace`.

The `:kind` will be used to support refactoring (among other possibilities). One problem with refactoring currently is that proofs are kept as proof scripts, and any types, expressions, etc. are given as strings. Thus, for example, a command such as `(expand "foo")` will resolve the name `foo`, and expand occurrences of it within the current sequent. This is the case even if `foo` is overloaded, and has three definitions in the sequent. Note that `foo` is resolved by the prover, and the resolutions are used in the subsequent expansions, but then discarded. If now the user decides that overloaded `foo` is confusing, and wants to name them apart, there is no way to know which ones to name apart in proof scripts without rerunning them.

The `:kind` keyword will be used to associate a kind with each argument, which in cases such as above would invoke functions that generate the resolutions and cache the resolution information with the proof, in a way that it may be used subsequently for refactoring, etc.

Detailed Description

The basic idea and motivation are above, the rest of this section goes into more details for those wanting to write new strategies.

The formal arguments list for a new prover command is in a specific order: required, optional, key, rest, and optional. The actual syntax is

```
prover-args ::= {var}*
                [&optional {var | (var initform)}*]
                [&key {var | (var initform)}*]
                [&rest var]
                [&inherit {cmd | (cmd :except var+)}*]
```

Required, optional, and rest arguments work exactly as detailed in the prover guide. Key arguments are similar to optional arguments, but may only be specified by keyword, not by position.

The inherit argument is fundamentally different. A proof command inherits arguments from other proof commands. This can only be done for proof commands that are directly referenced in the body; for example, `grind` inherits from `replace*`, not `replace`, because it does not directly call the latter. There are two aspects to inheriting arguments from a command. The first is that the command being defined takes the union of all the arguments of its own and inherited commands. The second, is that these inherited arguments are propagated to any calls of inherited commands.

The inherited arguments are always either optional or key arguments; they are always treated as key. Hence the order of inherited arguments is not an issue, though there is a possible issue if the names of arguments clash with different meanings. This can be controlled to some extent by using the `:except` form, specifying the arguments to be ignored of an inherited command. If there are more than one unignored arguments with the same name and different default values, the first is taken as default. Again, this can easily be controlled, for example, if we have the forms

```
(defstep foo (x &optional (a 3) &key (b 5) (c 7)) ...)
(defstep bar (y &optional (b 7) &key (a 11) (c 13)) ...)
(defstep baz (z &key (a 13) &inherit (foo :except c) bar) ...)
```

Then `baz` gives its own default to `a`, and takes `foo`s default for `b` and `bar`s default for `c`.

Propagating the arguments to calls is relatively straightforward. Using the above as examples, if the body of `baz` has an occurrence of `(bar m)`, it is simply replace by `(bar m :b b :a a :c c)` and `(foo n)` is replaced by `(foo n :a a :b b)`. Note that multiple invocations may be made to, e.g., `foo`, and all of them are replaced. Note also that, e.g., one could be as above, while the second invocation is `(foo :c 31 :a 37)`, which gets expanded to `(foo :c 31 :a 37 :b b)`.

The PVS Emacs command `M-x help-pvs-prover-strategy` (`C-c C-h s`) now includes the expanded argument list and definitions, as well as the original forms. This can be helpful in understanding how the prover arguments work.

This change has little impact on existing proofs, though in the regression tests it was found that a couple of strategies defined in the NASA libraries were not quite correct, but the old strategy mechanism simply ignored extra arguments. Now those generate an error.

Positive Type Parameters

PVS treats positive type parameters specially in datatypes, so that, e.g., `cons[int](1, null) = cons[nat](1, null)`, but this did not extend beyond constructors and accessors. Now PVS treats all definitions accordingly. The basic idea is that if a given definition does not depend directly on the type, and only on the values, then it is safe to ignore the type parameter - though typechecking may still generate a TCC.

Thus, for example, `length[T]((: 2, 3, 5 :))` is 3, regardless of which numeric subtype T may be, though unprovable TCCs may result (e.g., if `T` is `even`). Similarly, `nth` and `every` depend only on the arguments, not on the types. An example of a definition that depends on the types, not merely the arguments, is

```
th[T: type from int]: theory
 ...
 foo(x: T): int = if (exists (y: T): y > x) then x else 0 endif
 ...
end th
```

This change can have an impact on existing proofs, though mostly it makes them more direct - some proofs involving recursive functions, e.g., `length[int](x) = length[nat](x)` require convoluted proofs.

Typepred Extension

The `typepred` prover command was extended to include functional typepreds. Thus if `f` has type `[D -> {x: R | p(x)}]`, then the proof command `(typepred "f")` would generate a hypothesis of the form `FORALL (x: D): p(f(x))`. Note that some commands such as `skolem`, take a flag that causes typepreds to be generated - this would also include these functional typepreds.

TCC Ordering

TCCs that depended on conjunctive forms were generated in some cases in reverse. This has no bearing on soundness or correctness, but some meta-analysis of PVS was made more difficult because of this, so it was fixed.

Yices

The yices prover commands have been fully integrated into PVS, and Yices versions 1 and 2 are included in the distribution.

Development Notes

with-context, cam

Incompatibilities

There are three primary sources of incompatibilities with this release. This first is due to more rigorous checking of arguments in proof commands. In the past, if there were left

over arguments after pairing command arguments with their invocation, they were simply ignored. Now an error is invoked. Generally these are easy to debug, and they usually indicate a programming error to begin with.

TCC ordering can affect formula numbering (e.g. `foo_TCC1` and `foo_TCC2` could be swapped, and within a proof, the branches may be swapped. In the regression tests, this was fairly rare.

The addition of more typepred information in proofs leads to additional hypotheses, and this can cause formula numbers to be shifted.

In the past, the typechecker was a little strict in creating new variables when a formula change was made (e.g., expanding a definition). It tends to keep the existing variable name more often. Occasionally, this means a reference to, e.g., "i_1" should be changed to "i".

PVS 6.0 Release Notes

PVS 6.0 is a significant new release of PVS. The highlights include declaration parameters, better numeric simplification, Unicode character support, and full integration of NASA packages.

Installation Notes

The system is installed as usual; see the download pages at

```
http://pvs.csl.sri.com/download.shtml
```

We strongly suggest getting a pre-built Allegro version, unless you have concerns with the Allegro runtime click-though license, in which case get one of the SBCL Lisp images. It is possible to build from sources, but it can be sensitive to the platform environment. If you decide to try it and run into problems, let us know at `pvs-bugs@csl.sri.com`. PVS 6.0 is built with Allegro CL 9.0 and SBCL 1.0.47. It is available for Linux 32- and 64-bit machines, and Mac 64-bit. Note that because CMU Lisp lacks support for both 64-bit Linux and Unicode, we no longer provide that image.

New Features

Declaration Parameters

PVS has theory level parameters, which allow generic theories to be defined. They are very useful, and are used extensively, but there are situations where they are not so convenient. In particular, because of the way that Why3 (`http://why3.lri.fr/`) generates theories, it was becoming increasingly difficult to support PVS. Declaration parameters solve these problems.

Declaration Parameter Examples

For example, the theory of groups may be introduced as

```
groups[T: TYPE, *: [T,T -> T], one: T]: THEORY
```

Basic facts may be given, after which it is natural to define homomorphisms. But to define these over two (possibly) distinct groups, the theory requires two sets of parameters, hence another theory:

```
homomorphism_lemmas[T1: TYPE, *: [T1,T1 -> T1], one1: T1,
                    T2: TYPE, o: [T2,T2 -> T2], one2: T2]: THEORY
```

This seemingly minor inconvenience is made much worse when stating that homomorphisms are associative, which now requires yet another theory with four sets of parameters. At this point getting the parameters and importings right, while at the same time trying to follow a standard mathematical presentation of group theory becomes extremely difficult.

PVS 6.0 solves this by providing declaration parameters. These are similar to theory parameters in form, but their scope is just the associated declaration. The following exemplifies this with a somewhat different formulation of groups. Note that everything is within a single theory. This is mostly for illustrative purposes, there are many possible representations of groups in PVS; choosing one depends on the use to be made of it.

```
groups: theory
```

```
groups: theory
begin
 G[t: type+]: type+ from t
 assocG[t: type+]: type = (associative?[G[t]])
 idG[t: type+](op: assocG[t]): type = (identity?[G[t]](op))
 inverseG?[t: type+](op: assocG[t], id: idG[t](op))
                    (inv: [G[t] -> G[t]]): bool
    = inverses?[G[t]](op)(inv)(id)
 inverseG[t: type+](op: assocG[t], id: idG[t](op)): type
    = (inverseG?(op, id))
 +[t: type+]: assocG[t]
 0[t: type+]: idG[t](+[t])
 -[t: type+]: inverseG[t](+[t], 0[t])

 hom?[t1, t2: type+](h: [G[t1] -> G[t2]]): bool =
   h(0) = 0 and
   forall (a, b: G[t1]): h(a + b) = h(a) + h(b) and
   forall (a: G[t1]): h(-a) = -h(a)

 hom_is_assoc[t1, t2, t3, t4: type+]: lemma
   forall (f: (hom?[t1, t2]), g: (hom?[t2, t3]), h: (hom?[t3, t4])):
     h o (g o f) = (h o g) o f
end groups
```

Currently declaration parameters are restricted to types, this will likely be extended in the future. One of the Why3 examples is

```
whyex: theory
begin

 ilist[t: type]: datatype
  begin
   inull: inull?
   icons(icar: t, icdr: ilist): icons?
  end ilist

 length[t: type](l: ilist[t]): RECURSIVE nat =
    CASES l OF
      inull: 0,
      icons(x, y): length(y) + 1
    ENDCASES
   MEASURE reduce_nat(0, (LAMBDA (x: t), (n: nat): n + 1))

  inth[t: type](l: ilist[t], (n:below[length(l)])): RECURSIVE t =
    IF n = 0 THEN icar(l) ELSE inth(icdr(l), n-1) ENDIF
   MEASURE length(l)

 mem[t: type](x: t, l: ilist[t]): recursive bool =
   cases l of
```

```
      inull: false,
      icons(y, ll): x = y OR mem(x, ll)
    endcases
  measure length(l)

 mem_inth[t: type]: lemma
   forall (x: t, l: ilist[t]):
     mem(x, l) iff (exists (n: below(length(l))): x = inth(l, n))

 sorted(l: ilist[int]): bool =
   forall (n, m: below(length(l))): n < m => inth(l, n) <= inth(l, m)
 sorted_mem: lemma
   forall (x: int, l: ilist[int]):
       (forall (y: int): mem[int](y, l) => x <= y)
     & sorted(l) <=> sorted(icons(x, l))

end whyex
```

Note that this defines a list datatype. In the past, inline datatypes could not generate the `map` or `reduce` functions, as they needed extra theory parameters and had to generate external theories. With declaration parameters, this is not a problem, so these are generated inline.

Declaration Parameter Details

The parser has been modified to allow declarations to have an optional argument of exactly the form of theory parameters, except that (for now) importings, theory declarations, and constant declarations are not allowed.

As declaration parameters are types, the PVS type checker can usually infer the types, as seen in the examples above. Where it becomes ambiguous, names can include the parameters. For example,

```
th[t: TYPE]: THEORY
BEGIN
 f[s: TYPE](x: t): s
END th
```

A reference to `f` may be unambiguous, if not, `f[int]` may work, but PVS will try the actuals as both theory and declaration parameters; if that doesn't work, then `f[int][int]` is allowed. In this case, that would be equivalent to `th[int].f[int]`. Empty brackets are allowed syntactically, but this hasn't been thoroughly tested. The intention is that `f[][int]` means `f` must come from a theory with no parameters, while `f[int][]` means that `f` must have one parameter in its theory, and no declaration parameters.

Most declarations allow these parameters, except for library declarations. However, most judgements will fail to match if they have declaration parameters, as the judgement mechanism uses a fairly simple matching algorithm to ensure it is fast.

In mappings for theory interpretations, uninterpreted types and constants with declaration parameters must also include theory parameters, as shown in the following example

```
monad: THEORY
```

```
 BEGIN

 m[a: TYPE+]: TYPE+

 return[a: TYPE+]: [a -> m[a]]

 >>=[a, b: TYPE+](x: m[a], f: [a -> m[b]]): m[b] % infix
 >>=[a, b: TYPE+](x: m[a])(f: [a -> m[b]]): m[b] = x >>= f; % Curried

 >>[a, b: TYPE+](x: m[a])(y: m[b]): m[b] = x >>= (lambda (z: a): y);

 join[a: TYPE+](x: m[m[a]]): m[a] = x >>= id[m[a]]

 bind_return[a, b: TYPE+]: AXIOM
   FORALL (x: a, f: [a -> m[b]]): (return[a](x) >>= f) = f(x)

 bind_ret2[a: TYPE+]: AXIOM
   FORALL (x: m[a]): (x >>= return[a]) = x

END monad

Maybe[a: type]: datatype
begin
  Nothing: Nothing?
  Just(Val: a): Just?
end Maybe

maybe: THEORY
BEGIN
 importing Maybe

 bind[a, b: type](x:Maybe[a])(f: [a -> Maybe[b]]): Maybe[b]
   = cases x of Nothing: Nothing, Just(y): f(y) endcases

 mm: theory =
      monad{{m[a: type] := Maybe[a],
             return[a: type] := Just[a],
             >>=[a, b: type](x:Maybe[a], f: [a -> Maybe[b]])
                    := cases x of Nothing: Nothing,
                                  Just(y): f(y) endcases
            }}

  f(x: int): Maybe[int] =
      if rem(2)(x) = 0 then Nothing else Just(2 * x) endif
  g(x: int): Maybe[int] =
      if rem(3)(x) = 0 then Nothing else Just(3 * x) endif
  h(x: int): Maybe[int] =
```

```
        if rem(5)(x) = 0 then Nothing else Just(5 * x) endif
    k(x: int): Maybe[int] = f(x) >>= g >>= h
    k7: formula k(7) = Just(210)
    k25: formula k(25) = Nothing

end maybe
```

Better Numeric Simplification

PVS 6.0 now includes better simplification as part of the prover assert command for all the four arithmetic operators (`+`, `-`, `*`, and `*`). A new numeric expression class was introduced to handle results that are not natural numbers, in particular negatives and rationals. What this means is that subterms such as `(5 / 13 - 7 * 3)` get simplified to `-268/13`. This can have a dramatic effect in speed and readability.

Various adjustments were made to rewriting, matching, etc., in order to, for example, match the variables in `x / y` to the single rational number `-268/13`. This is not perfect, and some proofs will likely need adjustment, depending on how much arithmetic is involved.

Controlling Assert Post-processing

The prover `assert` command by default checks all of the type predicates of any formula being asserted, as very occasionally a contradiction is found - e.g., `even?(3)`. In rare cases (e.g., proofs within the Bernstein package of the NASA library), this check can take a significant amount of time, even though there are no contradictions. The `assert` command now includes an `ignore-typepreds?` flag to address this. By default `assert` works as before but if it is taking an inordinate amount of time, it may be worth setting this flag to `t`, in which case it will forgo the checks. Note that this is not unsound as any contradiction will be detected in later processing, but it may delay detection and thereby make it more difficult to pinpoint the cause.

The `ignore-typepreds?` flag has been included in all PVS prover commands that invoke `assert`.

Unicode Support

PVS 6.0 supports Unicode. As the release notes are written in Texinfo, which does not support Unicode, the main documentation is in the PVS Unicode help file, `M-x help-pvs-unicode` (`C-c C-h u`). Note that the help describes the Emacs input methods, but the point is PVS specifications may include Unicode, however they are edited.

Loading Patches

Patches are now loaded from files in `pvs-patches` subdirectories located in the `PVS_LIBRARY_PATH` and the PVS distribution `lib` (`M-x whereis-pvs`). The files should have the form `patch-*.lisp`, where the `*` is usually a number (not required). The `pvs-patches` subdirectories are searched in reverse order: thus the PVS distribution `pvs-patches` will be loaded first, and the patches in the first library appearing in `PVS_LIBRARY_PATH` will be loaded last. This only matters in case of conflicting patches, and generally means that the patches in a given library override all following patches.

From within a `pvs-patch` subdirectory the files are loaded in order of the numbers, if given, or the names.

Note that this replaces the older patch mechanism - the `patchlevel` arguments are still allowed, but `none` or 0 mean load no patches and everything else is treated the same as the default and loads all patches.

PVSio, ProofLite, Field, and Manip

NASA developed these packages, and PVSio was earlier integrated into PVS. Thanks to NASA, ProofLite, Field, and Manip packages are now integrated into PVS, and no longer need to be separately obtained and installed. The documentation is included in the `doc` subdirectory of the PVS distribution: `PVSio-2.d.pdf`, `manip-guide.pdf`, `extrategies.pdf` (for Field), and `ProofLite-4.2.pdf`. Note that these may not reflect the integration, so ignore anything that mentions obtaining and installing the package.

Theory Interpretation Changes

There are a significant number of changes to theory interpretations, mostly bug fixes and changes to handle declaration parameters.

Recursive types and `finite_sets`

The (co)datatype mechanism has been modified to allow `finite_sets` in recursive types, i.e.,

```
tree[t: TYPE]: DATATYPE
BEGIN
 leaf: leaf?
 node(children: finite_set[tree]): node?
END
```

Note that allowing `set[tree]` would cause problems, as the cardinality of the type of tree cannot be determined so there cannot be a set-theoretic semantics. Finite sets cause no such problems.

Datatype subterms

The `<<` subterm relation generated for datatypes was declared to be well-founded, but it is actually also strict, i.e., irreflexive and transitive. This property is often useful, and proving it for each instance of a datatype is inconvenient, so it was added to both the declaration and the axiom.

Incompatibilities

The changes in 6.0 lead to a number of incompatibilities, the impact is primarily in the proofs. As usual, the best way to deal with possible incompatibilities is to make a copy of your specification directory, run PVS 6.0 on it, and for any proof that fails in ways that are not obvious, run the earlier version of PVS on the original directory in parallel. Start the proofs with `M-x step-proof` and use `TAB 1` to step through the two proofs, and look for differences.

- The handling of rationals tends to lead to smaller forms for arithmetic expressions (see the Bernstein library at `http://shemesh.larc.nasa.gov/fm/ftp/larc/PVS-library/` for examples where this is an issue). In general, this makes proofs easier, but in some cases, a rewrite that worked in earlier versions no longer matches.

As the simplification only happens with an assert (or a strategy that invokes assert), it may be possible to modify the proof by delaying the assert. Otherwise, it is always possible to use the lemma command to instantiate the formula directly, followed by replace - the `use` command may be useful here.

- The declaration parameters allows names to have up to two sets of parameters, i.e., `f[int][real]`, which would resolve to an `f` from a theory with a single type parameter, and it would itself have a single type parameter. PVS also allows `f[real]`, and treats this ambiguously. To be less ambiguous, PVS should allow, e.g., `f[][real]`, which requires that the theory has no parameters. To support this, empty square brackets can no longer be an operator. This is somewhat mitigated by having Unicode available, so `\Box` and `\Diamond` can be used in place of `[]` and `<>`.

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`.

PVS 5.0 Release Notes

PVS 5.0 is a significant new release of PVS. The highlights include an SBCL port, the incorporation of the PVSio, Manip, Field, and ProofLite packages from NASA, a new theory interpretation implementation, new judgement forms, new support for Yices, and several bug fixes.

Installation Notes

The system is installed as usual; see the download pages at

`http://pvs.csl.sri.com/download.shtml`

We strongly suggest getting a pre-built Allegro version, unless you have concerns with the Allegro runtime click-though license, in which case get one of the pre-built CMU or SBCL Lisp images.

Note that we are no longer providing PVS for Solaris or PowerPC on the download page, as we have limited funds and the vast majority of our users have Intel Macs or Linux machines. If you do need either of these, let us know at `pvs-sri@csl.sri.com`. We will be keeping earlier versions available for the foreseeable future.

It's important to note that the `.prf` format has changed. While PVS 5.0 can easily read earlier proof files, the new format will confuse earlier versions of PVS. Please make copies of any directories that you want to also use earlier versions of PVS on. This is especially helpful when trying to repair proofs that work in earlier versions.

New Features

Available Lisp/Platforms

There are now 64-bit and 32-bit versions available for Intel Mac and Linux for Allegro (version 8.2) and SBCL Common Lisp. 32-bit versions are also available for CMU Common Lisp.

PVS Invocation

The `pvs` shell script has been modified to support SBCL Common Lisp. In addition, the load and eval capabilities have been simplified, where `-l` and `-e` are for Emacs files and expressions, and `-L` and `-E` are for Common Lisp files. There was an earlier `-L` flag, but it loaded the file *before* PVS initialization, which is more difficult to use.

PVSio Integration

César Muñoz has provided further improvements for PVSio, which is now more fully integrated into PVS. PVSio provides

- an alternative interface to the ground evaluator, making it easier to interact
- Several semantic attachments that facilitate programming in PVS
- A `defattach` macro, allowing users to add their own semantic attachments
- A `pvsio` script, allowing PVS specifications to be run as scripts
- Adds the `eval`, `eval-expr`, and `eval-formula` rules for use in the prover

PVSio now starts up without the need for loading extra library files. The sematic attachments are pre-installed, and the supporting theories are in the prelude. For more details on how to use PVSio, see the manual in `doc/PVSio-2.d.pdf`, (but ignore the installation instructions).

Manip and Field

The NASA Manip and Field packages have similarly been integrated, and improved by their authors. The Manip package provides many features, including Emacs extensions, many new proof strategies, and a pattern matching facility that allows reference to subterms during proof. For more details, see `doc/manip-guide.pdf` (but ignore the install instructions).

The Field package builds on Manip, and adds several prover commands making it easier to reason about nonlinear formulas. See `doc/extrategies.pdf` for details.

ProofLite

César Muñoz has also provided his ProofLite extension as part of PVS. This allows proofs to be included directly in PVS specifications and run from them, allowing for a literate programming style of specification. See `doc/ProofLite-4.2.pdf` for details.

Theory Interpretations

Theory interpretations have been significantly modified. Previously a theory declaration would generate a separate theory. This caused all kinds of problems, as it couldn't reference any declarations from the referencing theory. In some cases this could be handled by splitting theories into pieces, but even this doesn't always work.

In the new treatment, theory declarations are simply expanded in place, with the theory declaration id prepended to the included theory declarations. This solves the problem with declaration references, but introduces new issues. First, since the expanded theory declaration may itself have theory declarations, names have been extended to include any number of periods, e.g., `th1.th2.th3.d`. In general, these are only needed to disambiguate, and even then a suffix of the full name usually suffices.

The `prettyprint-theory-instance` is no longer meaningful, simply use `M-x prettyprint-expanded` to see the included declarations.

Expression Judgements

A new Judgement form is available. This is still in the experimental stage, but we welcome any feedback. The judgement has the form of a subtype judgement, but with a preceding `FORALL` that gives the types of the variables, as well as making the parsing unambiguous. Within the forall, any expression is allowed. For example,

```
judgement forall (x: real) = x*x has_type nnreal
f: [nnreal -> real]
foo: formula forall (y: real):
  f(f((y - 100) * (y - 100)) * f((y - 100) * (y - 100))) = 2
```

Without the judgement, `foo` generates 2 TCCs; with the judgement, none are generated for `foo`, just the one for the judgement itself, which is much simpler to understand. Note that the judgement mechanism has to be fast, and not itself generate proof obligations, so the matches are purely syntactic. Thus '(x - 1) * (-1 + x)' will still generate a TCC.

Yices Enhancements

There are improvements to the `yices` and `yices-with-rewrites` rules. Note that to use these commands you must install Yices from `yices.csl.sri.com`, and make certain the directory containing the `yices` executable is in your `PATH` environment variable before invoking PVS.

PVS Libraries Speedbar

A simple Emacs speedbar extension is now available. This makes browsing libraries easier, as it lists all directories on the `PVS_LIBRARY_PATH` as well as the built-in libraries in the PVS `lib` subdirectory. This is invoked with `M-x pvs-speedbar-browser`, and clicking on a given library shows the specification files within; clicking on one of them brings it up in an Emacs buffer. In the future we plan to include declaration lists within the speedbar display.

Incompatibilities

The changes lead to some incompatibilities, primarily in the proofs. This is due to a number of factors:

- A bug was fixed in the `lift-if` rule that could potentially lead to unsoundness. The fix means that the rule is no longer as aggressive, so proofs may need to be repaired. This is especially true for proofs with sequences of `lift-ifs`.
- Some bugs were fixed in which TCCs were missing.
- In general, judgement processing has been improved. Though this is almost always better for new proofs, it does tend to cause older proofs to fail, because the proof trees differ.
- PVS 5.0 uses a new format for .prf files. The reason for this is to remove some information (such as the time the last run took) that causes version control systems to think there is a change, making it difficult to keep developments in sync.
- Any reliance on NASA libraries is obviously different now that the NASA extensions are integrated into PVS. In practice, this has been relatively painless.
- In addition to adding theories from the NASA extensions, a number of minor changes have been made to the prelude. Two notable ones are that `choose` no longer has a definition, and `singleton` is no longer a conversion by default. If you really want `choose` to be `epsilon`, there is an axiom for that. The `singleton` conversion is often used unexpectedly, especially when bitvectors are involved. If you want it, simply add a `conversion+ singleton` where needed. Remember it's imported, so it can be placed in a theory low in the hierarchy; it doesn't have to be added to every theory. The rest of the changes are only likely to be a problem if they accidentally introduce ambiguities. These are usually easily resolved.

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`.

PVS 4.2 Release Notes

PVS 4.2 is primarily a bug fix release; there are few new features. Some of the changes do affect proofs, though our experience is that only a few proofs need adjustment, and most of these are quite easy to recognize and fix.

Installation Notes

The system is installed as usual; see the download pages at

`http://pvs.csl.sri.com/download.shtml`

We strongly suggest getting a pre-built Allegro version, unless you have concerns with the Allegro runtime click-though license, in which case get the pre-built CMU Lisp image.

Changes

– The prelude has been modified. First, the definition of `the` has been removed. This is to keep automatic rewrites from expanding `the` to epsilon, which is generally not useful. Proofs that used this will need to be modified, usually by bringing in the `the_lem` lemma. In addition, a number of new lemmas have been added. Here is the complete list of changes:

- From theory `sets`:

```
the(p: (singleton?)): (p)

the_lem: LEMMA FORALL (p: (singleton?)): the(p) = epsilon(p)

the_prop: LEMMA FORALL (p: (singleton?)): p(the(p))

is_singleton: LEMMA
  FORALL a: (nonempty?(a) AND
              FORALL x, y: a(x) AND a(y) IMPLIES (x=y))
    IMPLIES singleton?(a)

singleton_elt_lem: LEMMA
   singleton?(a) and a(x) IMPLIES singleton_elt(a) = x

singleton_elt_def: LEMMA
   singleton?(a) IMPLIES singleton_elt(a) = choose(a)

singleton_rew: LEMMA singleton_elt(singleton(x)) = x

AUTO_REWRITE+ singleton_rew
```

- From theory `list_props`:

```
every_nth: LEMMA
    every(P)(l) IFF FORALL (i:below(length(l))): P(nth(l,i))
```

- From theory `more_map_props`:

```
map_nth_rw: LEMMA
```

```
FORALL (i: below(length(l))):
  nth(map(f)(l), i) = f(nth(l, i))
```

- César Muñoz has provided improvements for pvsio. See `doc/PVSio-2.d.pdf` for details.
- Judgements over dependent types have been fixed - in most cases this meant the judgement was not used where it should have been, in a couple of cases it left free variables uninstantiated, causing breaks.
- Recursive judgements were recently introduced (see the 4.1 release notes), and several bugs have been fixed. In addition, now when recursive judgement has a name, the corresponding formula is generated as an axiom.
- Auto-rewrites now find the proper instances; prior to this, the auto-rewrites were kept in generic form, and never properly instantiated.
- Theory interpretations have had a number of bugs fixed.
- TCC subsumption tests have been improved, leading to fewer TCCs.
- Batch mode now saves the context; before this, Emacs was exiting without giving lisp a chance to save.
- Libraries are more robust; in particular, relative library paths now work properly when used recursively.

Incompatibilities

The changes lead to some incompatibilities. The improved judgements and TCC subsumption lead to fewer TCCs, hence may cause TCC renumbering and proofs may have to be shifted (`M-x show-orphaned-proofs` may be useful here).

In addition to these, the improved auto-rewrites also affect proofs, as some branches of a proof may no longer be generated, or may have a different form. It is usually easy to repair, though it often helps to run an older version of PVS in parallel to figure out where the proof deviates. Finally, any proof that relies on the expansion of `the` will need to use `the_lem` instead.

PVS 4.1 Release Notes

PVS 4.1 is primarily a bug fix release; there are few new features. 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.

Installation Notes

The system is installed as usual; see the download pages at

`http://pvs.csl.sri.com/download.shtml`

We strongly suggest getting a pre-built Allegro version, unless you have concerns with the Allegro runtime click-though license, in which case get the pre-built CMU Lisp image.

The build process is largely untested outside of SRI. The process has been somewhat improved with this release, but please let us know of your experiences, and suggestions for improvement. Problems and solutions may be sent to `pvs-bugs@csl.sri.com`. If you are more ambitious, feel free to expand on the build description in the PVS Wiki (`pvs-wiki.csl.sri.com`).

PVS was recently moved from CVS to Subversion, and you can now access the system using `svn`:

```
svn checkout https://spartan.csl.sri.com/svn/public/pvs/trunk pvs
```

For now, this is read-only outside of SRI. If you wish to contribute code to PVS, please send it to `pvs-sri@csl.sri.com`, and we will assess and incorporate it.

Upgrades

- PVS Now uses Allegro 8.1.
- ILISP has been updated to version 5.12.0 - we include slightly modified sources of the ILISP files in `pvs/emacs/emacs-src/ilisp/`.
- The `pvs` startup script has been modified to work with both Allegro and CMU Lisp. In particular, the '`-L`' flag may be used to indicate a lisp file to be loaded *after* PVS starts.
- Thanks to Jerry James (`jamesj@acm.org`, the BDD and WS1S source code have been brought up to date, and no longer generate warnings.
- We have made minor additions to the prelude: A `nonempty_set` type, and the following lemmas: `Union_member lemma`, `Union_emptyset_rew`, `Union_union_rew`, `Intersection_member`, `Intersection_intersection_rew`, `mod_wrap2`, `mod_inj1`, `mod_inj2`, `mod_wrap_inj_eq`, `mod_neg_limited`, `odd_mod`, `even_mod`, and `finite_Union_finite`.
- The prover `instantiate` command now allows "_" in the same way as the `skolem` command, allowing partial instantiation.
- The prover `copy` command was a derived rule, and is now a primitive rule. This is to keep it from generating spurious TCCs.
- The `let-reduce?` flag has been added to various strategies.
- Various improvements have been made to the `yices` interface.

– After struggling with making the bin files work with both case sensitive and case insensitive Lisps, we decided the simplest solution is to create separate bin files for each. Thus there is now a `pvsbin` subdirectory created for case sensitive Lisps (Allegro in this case), and `PVSBIN` for case-insensitive ones (CMU Lisp).

Incompatibilities

The bugs that have been fixed in 4.1 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 of PVS 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 4.1, 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`.

PVS 4.0 Release Notes

PVS 4.0 is available at `http://pvs.csl.sri.com/download.shtml`.

Release notes for PVS version 4.0.[1] The major difference from earlier versions of PVS is that this release is open source, under the GPL license (`http://www.gnu.org/copyleft/gpl.html`). In addition, there is now a PVS Wiki page (`http://pvs-wiki.csl.sri.com`).

Installation Notes

Installation of binaries is the same as before; the only difference is that only one file needs to be downloaded. This leads to slightly more overhead when downloading for multiple platforms, but simplifies the overall process. Simply create a directory, untar the downloaded file(s) there, and run `bin/relocate`.

If you have received patches from SRI that you have put into your `~/.pvs.lisp` file, they should be (re)moved. If you anticipate wanting to try the newer and older versions together, you can do this by using `#-pvs4.0` in front of forms in your 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

Open Source

PVS is now open source, under the under the GPL license (`http://www.gnu.org/copyleft/gpl.html`). It currently builds with Allegro and CMU Common Lisps, and we are working on porting it to SBCL. Feel free to join in if your favorite Lisp or platform is not yet supported. See the PVS Wiki page (`http://pvs-wiki.csl.sri.com`) for details.

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 simple 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.

[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.

Structural Subtypes

PVS now has support for structural subtyping for record and tuple types. A record type `S` is a structural subtype of record type `R` if every field of `R` occurs in `S`, and similarly, a tuple type `T` is a structural subtype of a tuple type forming a prefix of `T`. Section [Record and Tuple Type Extensions], page 31, 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 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 typechecking, and makes TCCs and error messages more inscrutable. If both are desired, simply include a structural subtype followed by a predicate subtype, for example:

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

Now `move` may be applied to `gpoint`s, but if applied to a `spoint` an unprovable TCC will result.

Structural subtypes are a work in progress. In particular, structural subtyping could be extended to function and datatypes. And to have real object-oriented PVS, we must be able to support a form of method invocation.

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 fully incorporated into PVS. Thus for PVS 4.0 there is no need to download the package. See the `doc/PVSio-2.d.pdf` manual for details, and the PVSio web page `http://research.nianet.org/~munoz/PVSio/` for updates.

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.

For the prover, random testing is invoked with the `random-test` rule:

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

In the ground evaluator, we added the `test` command:

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

Note one important difference: the optional arguments in the `test` command are **not** keywords. To set the `all?` flag you would need to invoke `test` as

```
(test "foo" 10 100 10 t)
```

In general, random testing is most easily used in the prover. Note that you can get an arbitrary expression into the sequent by using the `case` command.

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`)
- `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.

More details may be found in the paper Random Testing in PVS (`http://fm.csl.sri.com/AFM06/papers/5-Owre.pdf`), which was presented at AFM 2006 (`http://fm.csl.sri.com/AFM06/`).

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 safely be ignored if you will not be using Yices.

The `yices` rule is an endgame solver; if it does not prove (the specified 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 for judgements on recursive functions, but this would make existing proofs fail.

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.

Thanks to Jerry James, 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.

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 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`.

These 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

Table of Contents