PVS API Reference

Version 3.0 • September 2003

S. Owre
N. Shankar
{Owre,Shankar}@csl.sri.com
http://pvs.csl.sri.com/

SRI International
Computer Science Laboratory • 333 Ravenswood Avenue • Menlo Park CA 94025

The initial development of PVS was funded by SRI International. Subsequent enhancements were partially funded by SRI and by NASA Contracts NAS1-18969 and NAS1-20334, NRL Contract N00014-96-C-2106, NSF Grants CCR-9300044, CCR-9509931, and CCR-9712383, AFOSR contract F49620-95-C0044, and DARPA Orders E276, A721, D431, D855, and E301.

Contents

Chapter 1

Introduction

The Prototype Verification System (PVS) is a comprehensive framework for formal specification and verification that features an expressive specification language based on higher-order logic, and powerful tools for interactive proof construction. Use of PVS is documented in a series of manuals [4, 3, 5]; the present document focuses exclusively on the API used to extend PVS, or to integrate it with other software.

Conceptually, the PVS API consists of three interfaces: the front end or user interface, the middle or strategies interface, and the back end or proof engine interface. The organization of the document follows this three-part structure, beginning with the user interface, which underlies the PVS Emacs interface, and includes, among many functions, those for parsing, typechecking, and proving. The strategies interface consists of built-in strategies such as `..`, as well as user-defined strategies specified via `defstep`. In many cases, strategies are written in Lisp and rely heavily on underlying parser, typechecker, and prover functions; much of the chapter on the strategies interface is devoted to a description of these functions. The chapter on the proof engine interface describes functions used to add new logical engines (e.g., new decision procedures, new model checkers, etc.) to the prover.

The descriptions that follow are based on the code for PVS version 3.2. We view PVS as a work in progress and continually strive to refine and improve the system. To this end, we solicit comments and suggestions on the presentation of the PVS API, as well as proposals for changes to PVS datastructures and code.

Chapter 2

The Front End: User Interface

This chapter describes the functions supporting the user interface. The interaction in PVS is primarily through customized Emacs scripts. The Emacs commands themselves are described in the User Manual. Here we describe the interaction between Emacs and the PVS Lisp image, and the functions in the Lisp image that are invoked by the Emacs commands.

The information provided here should make it easy to add new Emacs commands, or to replace Emacs with a different user interface. Even if you are going to replace Emacs, reading the following sections should make this easier.

It is important to remember that both Emacs and the PVS image are written in Lisp, though they are not the same. In this document, we will refer to Emacs Lisp as Elisp, and Common Lisp as Lisp. We will also refer to the PVS Lisp image as Lisp, context should make it obvious whether it refers to the language or the image.

Note that the Emacs source files are provided with the PVS distribution.

2.1 The Emacs Interface

In this chapter we discuss the Emacs interface, including functions that are useful in defining new commands, various utility functions, and techniques we have found useful in developing and debugging Emacs code. This chapter can be skipped by those wishing to replace Emacs with a different fromt end.

2.1.1 Versions of Emacs

There are many versions of Emacs around. PVS is intended to support GNU Emacs versions 19.34 and later, and all versions of XEmacs. This is not always easy, as there are subtle differences between GNU Emacs and XEmacs, and even between different versions of the same system. For this reason, byte-compiled files go into separate directories, and some parts of the Emacs code test the version.

`pvs-emacs-system` [*Emacs global*]

The version of Emacs in which PVS is running. Set in go-pvs.el. Defined as one of `xemacs21`, `xemacs20`, `xemacs19`, `emacs20`, or `emacs19` and defaults to `emacs20` if the current version cannot be determined. Note that under GNU Emacs version 21 *`pvs-emacs-system`* has value `emacs20`, because the differences between these versions had little impact on PVS.

2.1.2 Defining New PVS Commands

New commands are generally provided to PVS by providing a command in Emacs, optionally giving it a key binding and/or an abbreviated form, defining a corresponding function in Lisp, and adding the command to the PVS menu.

`defpvs` (*`name class arglist docstring`* &REST *`body`*) [*Emacs macro*]

This is similar to Emacs *`defun`*, but includes a *`class`* identifier that is intended to be used for creation of menus and documentation. The *`class`* associated with the *`name`* can be obtained using the Emacs built-in function *`get`* using property name `pvs-command`, e.g., `(get 'prove-theory 'pvs-command)` yields the symbol `prove`.
Otherwise *`defpvs`* simply invokes *`defun`*. Note that to become an Emacs command the first form after the docstring must be an *`interactive`* form; see the Emacs Lisp reference manual for details.

2.2 Emacs/Lisp interaction

The Emacs code of PVS is built on ILISP, which is an interface from Emacs to an inferior Lisp. The basic model is simple: the PVS Lisp image runs as a subprocess of Emacs, lisp forms are sent from Emacs to Lisp, and the results are acted on by Emacs. This simple model has many details that will be described here.

2.2.1 PVS Initialization

PVS initialization starts from the `pvs` shell script, which in turn sets up environment variables and arguments with which to invoke Emacs (unless the `-raw` flag was provided).

Emacs then loads the PVS Emacs files, and runs the *`pvs`* function, which does some Emacs PVS initialization, then starts the interactive Lisp subprocess, and when it is ready, some initialization commands are sent to it.

The interactive buffer associated with the Lisp process is named `*pvs*`. Anything typed into this buffer is sent to the Lisp process when the `Return` key is pressed—unless the input is nufinished, for example because of unbalanced parentheses or a

missing double quote. Output and errors are generally sent to this buffer, though as described later some are filtered out by the output filter.

The `pvs` buffer tracks the low-level interaction between Emacs and Lisp. It is bounded by the Emacs variable `comint-log-size`, initially 15000 (characters). This can be made larger if something of interest scrolls out of the buffer.

Once initialized, both Emacs and Lisp simply wait for input.

pvs () [*Emacs function*]

The *pvs* command starts the pvs lisp process. This is used for PVS initialization, but it is also useful if the Lisp process has died for some reason, and you wish to keep using the same Emacs session.

2.2.2 The Lisp Status

It is important for Emacs (or an alternative interface) to know what Lisp is doing. The main things one is interested in are whether it is busy processing a command or waiting for input, and whether it is in an error state. The Lisp process takes a command, produces output, and then prompts for the next input. Emacs recognizes that Lisp is ready for more input when it sees the prompt. The prompt is not fixed; history indices, break levels, package changes, etc. all change the prompt string, so it is recognized using a regular expression (regexp).

This prompt is specific to Allegro Common Lisp, if and when PVS is ported to other lisps, the prompt regexp will have to be modified accordingly.

`comint-prompt-regexp` [*Emacs global*]

The prompt regexp is one long string (ignore the line breaks)

```
"^[␣]*\\(\\[[0-9]+i?c?\\]␣\\|\\[step\\]␣\\)?
\\(\\(<[-A-Za-z]*␣?[0-9]*>␣\\)\\3?\\|[-A-Za-z0-9]+([0-9]+):␣\\)
\\|Rule\\?␣\\|(Y␣or␣N)\\|(Yes␣or␣No)\\|Please␣enter"
```

This can be understood as follows:

`^[␣]*` - prompt starts with any number (including 0) of spaces.

`\\(\\[[0-9]+i?c?\\]␣\\|\\[step\\]␣\\)?` - optionally followed by a prefix from a break, e.g., `[3]`, where `3` is the break level, and the optional `c` indicates a continuable break, and optional `i` indicates an inspector break. If the stepper is running, the prefix is simply `[step]`.

After this the prompt must match one of six possibilities:

`\\(<[-A-Za-z]*␣?[0-9]*>␣\\)\\3?` - this is for the ground evaluator, which is of the form `<GndEval>`. Not sure why the regexp includes digits and possible repitition (the `\\3?` pattern matches the third occurrence of a `\\(...\\)`, which in this case is the immediately preceding pattern). This is possibly left over from some ground evaluator experiments.

`[-A-Za-z0-9]+([0-9]+):␣` - this is the normal Lisp prompt, usually of the form `pvs(25):␣`.

`Rule\\?␣` - the prover prompt (backslashes are needed for literal `?`).

The other three prompts are obvious, and come up when Lisp asks a question.

Emacs also must recognize when Lisp has been interrupted, because some commands will interrupt any running command. To do this, Emacs sends an interrupt signal to Lisp, and waits for the interrupt to be accepted. Then the interrupting command is sent, and when control returns the original command is allowed to continue.

`comint-interrupt-regexp` [*Emacs global*]

This has the following value.

```
"Error:␣[^\n]*␣interrupt\)"
```

And matches strings such as

```
Error:  Received signal number 2 (Keyboard interrupt)
```

Note that this is not a prompt, though a prompt usually follows shortly thereafter.

2.2.3 Emacs to Lisp

Most PVS commands take input from the user, create a Lisp form, and send it to Lisp using one of the following functions. Most of this process can be seen in the `pvs`

buffer, which records the low-level interactino between Emacs and Lisp.

pvs-send (***string*** &OPTIONAL ***message status***) [*Emacs function*]

This sends the ***string*** to Lisp, but does not expect a result. If Lisp is currently processing another command, this is put in an ILISP queue, and will be executed when all other commands before it are completed. The ***message*** is a string that is displayed in the minibuffer when the command is executed, and the ***status*** string is put on the status bar (it's best to keep this short).

pvs-send-and-wait (***string*** &OPTIONAL ***message status expected***) [*Emacs function*]

This is similar to ***pvs-send***, but waits for the result, which should be of type ***expected***. If another command is currently running it is interrupted and when the interrupt is acknowledged, the ***string*** is passed to Lisp, and the result of evaluating the ***string*** is passed back to the waiting function. Typically this is only used for short functions such as status or display commands. The ***expected*** can be any value, but if it is `'dont-care`, the result is automatically `nil`, otherwise the result is read from the Lisp output string.

pvs-file-send-and-wait (***string*** &OPTIONAL ***message status expected***) [*Emacs function*]

Exactly like ***pvs-send-and-wait***, except that the result is inserted into a temporary file, and the file name is returned. This is useful when the result is large, or if it may contain substrings that could match the prompt or other regexps. You can switch from ***pvs-send-and-wait*** to ***pvs-file-send-and-wait*** without modifying the Lisp code that is invoked.

2.2.4 Lisp to Emacs

The Lisp subprocess of Emacs has an associated *process filter*, ***pvs-process-filter***, that is invoked asynchronously when some output has been generated from lisp. This output is collected until a carriage return is seen, and then sent to the ***pvs-output-filter***, which filters out and acts on specific patterns of output; anything not filtered is then appended to the `*pvs*` buffer, and compared to the prompt regexp. If it matches, the next command in the queue, if any, is sent.

pvs-output-filter (***output***) [*Emacs function*]

The ***pvs-output-filter*** basically looks for patterns in the output, chops them out, acts on them, and returns what is left. There are several possibilities, all having the form

:pvs-***key*** $\texttt{arg}_1$ & ... & $\texttt{arg}_n$:end-pvs-***key***

The possible *key*s, their arguments, and their actions are described below.

Key	Arguments	[Invoking Lisp Functions] Action
msg	*string*	[*pvs-message*, *verbose-msg*] Displays *string* in the minibuffer, and adds it to the PVS Log buffer.
log	*string*	[*pvs-log*] Quietly adds the *string* to the PVS Log buffer.
out	*file*	[*pvs-output*] This is used in Batch Mode to force the *file* contents that usually go into the *pvs* buffer to be output to stdout.
err	*file* *dir* *msg* *err* *place*	[*pvs-error*] Switches to a boffer containing the PVS *file* (in directory *dir* if not nil), and places the cursor at the row and column of *place* (which is simply two nonnegative integers with a space between). Then the contents of file *err* are displayed in the PVS Error buffer, and the *err* file is deleted. Finally, the *msg* string is displayed in the minibuffer.
buf	*bufname* *file* *display* *read-only* *append*	[*pvs-buffer*] Creates a buffer with name *bufname* and contents from *file*. The buffer is displayed if *display* is t, set to read-only if *read-only* is t, and the *file* contents are appended if the optional *append* argument is t, otherwise they replace earlier buffer contents.
yn	*msg* *yesno-p* *timeout-p*	[*pvs-yn*] Uses *msg* as a prompt for a y or n answer from the user in the minibuffer. Both the other args are optional, if *yesno-p* is t, then prompts for yes or no, and if *timeout-p* is t, defaults to y (or yes) after pvs-default-timeout seconds.
bel		[*beep*] Simply beeps.
loc	*dir* *file* *place*	[*pvs-locate*] Adds a message to the PVS Log buffer indicating that this action was taken, and switches to a boffer containing the *file* of directory *dir*, and places the cursor at the row and column of *place* (which is simply two nonnegative integers with a space between).
mod	*dir* *file* *region* *textfile*	[*pvs-modify-buffer*] Replaces the *region* (four nonnegative integers separated by a space) in the given *file* of directory *dir* with the text in file *textfile*.
pmt	*kind* *prompt*	[*pvs-prompt*] Prompts the user for input using the minibuffer. The *prompt* string is displayed, and the *kind* allows Emacs to use normal input techniques (completion, etc.) for the result. Currently, the kind supported is just directory.
wish	*Tcl/Tk-form*	[*pvs-wish*, *pvs-wish-source*] Sends the *Tcl/Tk-form* to the wish subprocess (see Subsection 2.2.9).
eval	*Emacs-form*	[*pvs-emacs-eval*] Evaluates the *Emacs-form* string in Emacs, returning the result to Lisp.

The following functions are used to create the forms described above. However, if they are not invoked from Emacs, they simply produce their results in the `*pvs*` buffer. This is useful for debugging, as it is then easier to tell the sequence of events. This is controlled by the `*to-emacs*` Lisp global variable, which is set to `t` by the Lisp *pvs-errors* macro, which is wrapped around the strings sent by the Emacs *pvs-send*, *pvs-send-and-wait*, and *pvs-file-send-and-wait*.

pvs-message (*ctl* &REST *args*) [*function*]

The *ctl* argument is a *format* control string, and it should correspond to the *args*. If the Lisp global variable `*suppress-msg*` is `t`, nothing is output, otherwise the *format* macro (see CLtL2) is applied to the *args*, for the *string* argument to the `:pvs-msg` form, or simply output if `*to-emacs*` is `nil`.

pvs-log (*ctl* &REST *args*) [*function*]

This is exactly like *pvs-message*, except the results are not printed if `*to-emacs*` is `nil`, and uses the `:pvs-log` form otherwise.

pvs-output (*ctl* &REST *args*) [*function*]

As in *pvs-message*, *format* is applied to *ctl* and *args*. In batch mode (`*noninteractive*` is `t`), the result is put into a temporary file for the *file* argument to the `:pvs-out` form, otherwise it is simply output if `*to-emacs*` is `nil`.

pvs-error (*msg err* &OPTIONAL *ifile iplace*) [*function*]

This function is used to indicate a user error, such as a parse or type error. If `*rerunning-proof*` is not `nil`, `(restore)` is called, so the error is skipped when rerunning a proof. If `*to-emacs*` is set, it builds the `:pvs-err` form arguments as follows:

file - if currently processing a declaration generated for a recursive type (`*adt-decl*` is not `nil`), the *file* is the filename being generated. Otherwise, if `*from-buffer*` is set, it is used, else *ifile*.

dir - unless `*from-buffer*` is set, the `*pvs-context-path*` is used for *dir*.

msg - use the *pvs-error msg* argument.

err - use the *pvs-error err* argument.

place - if `*adt-decl*` is set, use its place, otherwise use *iplace* to extract the line and column.

pvs-buffer (*name contents* &OPTIONAL *display? read-only? append?*) [*function*]

If *to-emacs* is not nil, sends the :pvs-buf form with the given buffer *name*, a temporary *file* containing *contents*, and the three flag values.

pvs-yn (*msg full? timeout?*) [*function*]

This function prompts with *msg* for a yes or no answer. If *noninteractive* is t, this function simply returns t. If *pvs-emacs-interface* and *to-emacs* are both t, the :pvs-yn form is created and output. Then the Lisp *read* is called to read whatever is passed to it. If the value read is the keyword :abort, then *pvs-abort* is called, otherwise the value is returned.
If *pvs-emacs-interface* and *to-emacs* are not both t, then if *full?* is t, *yes-or-no-p-with-prompt* is called, otherwise *y-or-n-p-with-prompt* is called.

beep () [*function*]

If *to-emacs* is not nil, sends the :pvs-bel form.

pvs-locate (*theory obj* &OPTIONAL *loc*) [*function*]

If *to-emacs* is not nil, sends the :pvs-loc form. The *theory* is either a theory or datatype. The *dir* argument is nil unless the theory or datatype is from the prelude or a library, in which case it is the corresponding directory. The *file* is the file containing the theory or datatype, and the *place* is *loc* if non-nil, or the place of *obj*.

pvs-modify-buffer (*dir name place contents*) [*function*]

This function is used to modify a specified pvs file. Given a *dir* string, pvs file *name*, *place* (a place, and *contents* string, the region determined by the *place* is replaced by the *contents*. Note that if *contents* is nil or the empty string, this simply deletes the region.
If *to-emacs* is nil, this simply outputs the *contents*. Otherwise a :pvs-mod form is output, with the given *dir*, and *file* is set to *name*. The *region* is created by invoking *place-list* on the *place* argument. The *contents*, if non-nil, are written to a temporary file (using *write-to-temp-file*) which is used for the *textfile*.

pvs-prompt (*type msg* &REST *args*) [*function*]

This function is used to prompt for a value. It creates the prompt string by invoking `format` on *msg* and *args*. If `*to-emacs*` is `nil`, it outputs the prompt, otherwise it sends the `:pvs-pmt` form to Emacs, with *type* (as the *kind*) and the *prompt* string. In either case, the Common Lisp `read` function is invoked, which reads the result.

pvs-wish (*cmd*) [*function*]

This simply sends the `:pvs-wish` form with the given Tcl/Tk *cmd*.

pvs-wish-source (*file*) [*function*]

This function sends the `:pvs-wish` form with the Tcl/Tk string

```
"catch {source file}; exec rm -f file"
```

This causes the Tcl/Tk process to load the file, and then to delete it.

pvs-emacs-eval (*form*) [*function*]

If `*pvs-emacs-interface*` is not `nil`, this function creates a `:pvs-eval` form, which causes the *form* to be evaluated by Emacs. It then invokes the Common Lisp `read` function to read the result.

2.2.5 Lisp Functions invoked by Emacs Commands

Commands to PVS are normally entered through the Emacs interface, described in Section 2.1. Here we describe the functions invoked by Emacs. This is useful for debugging, and provides the information needed for providing an alternative User Interface.

Most of the functions have obvious arguments, but many of the PVS Emacs commands pass in the current location of the cursor as well. Most of the time this is the line number, with line 1 at the beginning of the file. If a column is given, it starts at 0. This is also true of the PVS output.

In the following, when a function is said to return a buffer with a certain name, it means that it returns a `:pvs-buf` form as described in Subsection 2.2.4.

The following sections correspond to the PVS menu structure, and for each menu entry shows the corresponding Lisp functions that are invoked, if any. Some of these functions are subsidiary, and simply provide, for example, the list of known PVS theories so that completion may be used. Such functions are marked with a daggar (†).

Getting Help

help-pvs	
help-pvs-language	
help-pvs-bnf	
help-pvs-prover	*help-prover*
help-pvs-prover-command	*collect-strategy-names*†, *help-prover*
help-pvs-prover-strategy	*collect-strategy-names*†, *show-strategy*
help-pvs-prover-emacs	
pvs-release-notes	

Most of the help commands simply load a file from the `lib` subdirectory of the PVS installation directory.

help-prover (&OPTIONAL *name*) [*function*]

Returns a buffer with the corresponding prover help. If the *name* is provided, help for that name is returned. If the *name* is not known, the contents will be

```
No such rule, defined rule or strategy.
```

If the *name* is not provided (or is `nil`), help for all the prover commands is returned.

show-strategy (*strat-name*) [*function*]

collect-strategy-names (&OPTIONAL *all?*) [*function*]

Returns a sorted list of the rule and strategy names. If *all?* is not `nil`, the list includes helper strategies as well. The strategies files, if any are loaded first, so user-defined rules and strategies are included.

Editing PVS Files

forward-theory	
backward-theory	
find-unbalanced-pvs	
comment-region	

These functions are entirely in Emacs, and do not invoke any Lisp functions.

Parsing and Typechecking

parse	*pvs-current-directory*†, *parse-file*
typecheck	*pvs-current-directory*†, *typecheck-file*
typecheck-importchain	*pvs-current-directory*†, *typecheck-file*
typecheck-prove	*pvs-current-directory*†, *typecheck-file*
typecheck-prove-importchain	*pvs-current-directory*†, *typecheck-file*

parse-file (*filename* &OPTIONAL *forced?* *no-message?*) [*function*]

This parses the specified *filename*, returning multiple values:

- a list of theories and datatypes
- a flag indicating whether the file was loaded from bin files (`t` if it was)
- the list of theories that changed since the last time the file was parsed

If *forced?* is `t`, then any bin files are ignored, and the file is parsed even if it hasn't changed. Otherwise it is only parsed if it is not up to date, based on the file modification time, or if the prover or ground evaluator are currently running. If binfiles are available and up to date, they are loaded, otherwise the file is parsed.

typecheck-file (*filename* &OPTIONAL *forced?* *prove-tccs?* *importchain?* *nomsg?*) [*function*]

This function is the main function for typechecking PVS files. It starts by calling *parse-file* to get the abstract syntax of the theories and datatypes in the *filename*. If *forced?* is `nil`, and the theories are already typechecked, a corresponding message is output unless *nomsg?* and *restored?* are `nil`. Similarly, if `*in-checker*` or `*in-evaluator*` are `t`, a message is output and the function exits. Finally, if the theories are not typechecked, or *forced?* is `t`, the theories are typechecked. Finally, if *prove-tccs?* is `t`, *prove-unproved-tccs* is invoked to attempt to prove the TCCs.

Prover Invocation

prove	*typecheck-file*, *typechecked?*†, *rerun-proof-at?*†, *prove-file-at*
x-prove	*typecheck-file*, *typechecked?*†, *rerun-proof-at?*†, *prove-file-at*
step-proof	*typecheck-file*, *typechecked?*†, *rerun-proof-at?*†, *prove-file-at*
x-step-proof	*typecheck-file*, *typechecked?*†, *rerun-proof-at?*†, *prove-file-at*
redo-proof	*typecheck-file*, *typechecked?*†, *rerun-proof-at?*†, *prove-file-at*
prove-theory	*collect-theories*†, *prove-theory*
prove-theories	*collect-theories*†, *prove-pvs-theories*
prove-pvs-file	*prove-pvs-file*
prove-importchain	*collect-theories*†, *prove-usingchain*
prove-proofchain	*prove-proofchain*

prove-file-at (*name declname line rerun?* &OPTIONAL *origin buffer prelude-offset background? display? unproved?*) [*function*]

This function is called to prove a specific formula. The *name* is the buffer name, without extension. The *origin* is one of `pvs`, `ppe`, `tccs`, or `prelude`. The *declname* is `nil`, unless this is from a declaration TCCs buffer, in which case it is needed in order to uniquely identify the declaration associated with the TCC (see *decl-to-declname*). The *line* is the line number the cursor was on (the first line is 1). The *rerun?* flag indicates whether the proof is to be rerun. The *buffer* is the name of the buffer, which is needed because files from different directories can have the same name in Emacs, though the buffer names are unique. For example, there might be a `foo.pvs` from two different directories, in which case the buffer name for one of them is likely to be `foo.pvs<2>`. The *background?* flag indicates whether the proof is to be run in the background, *display?* indicates the Tcl/Tk display should be used, and *unproved?* indicates whether to look for the next formula following the cursor position, or the next unproved formula, skipping over any proved ones.

The function works as follows. First the *line* is checked, if it is not an integer, then the arguments are all shifted one to the right, setting the *declname* to `nil`. This is just to handle calls from older versions, as the *declname* is a recent addition.

If `*in-checker*` or `*in-evaluator*` are `t`, the function exits with a message. Otherwise, `*to-emacs*` is bound to the value of *background?*, and the formula declaration is retrieved using *formula-decl-to-prove*. If *rerun?* is `t` and the formula has no associated proof, a message is output and the function exits. Otherwise `*current-context*` and `*current-theory*` are bound to the context and theory of the declaration. Note that the context does not include the declaration itself. The `*start-proof-display*` variable is bound to the value of *display?*, the proof script is extracted using *extract-justification-sexp*, and the relevant strategies files are loaded using *read-strategies-files*.

Then the proof is actually run. If *background?* is `t`, *pvs-prove-decl* is called, otherwise *auto-save-proof-setup* is called, followed by *prove* with the `(rerun)` strategy if *rerun?* is `t`.

When the prover returns control, `*last-proof*` is set to the resulting prooftree. If the proof has changed, it is written to file using *save-all-proofs*, and the proof status associated with the formula in `*pvs-context*` is set. Note that the proof does not change if *background?* is `t`. The *remove-auto-save-proof-file* function is called, and finally, if `*to-emacs*` is `t`, *pvs-locate* is called to make the buffer containing the formula current, and place the cursor at the beginning of it.

prove-theory	(*theoryname* &OPTIONAL *retry? filename use-default-dp?*)	[*function*]
prove-pvs-file	(*file retry?* &OPTIONAL *use-default-dp?*)	[*function*]
prove-usingchain	(*theoryname retry?* &OPTIONAL *exclude use-default-dp?*)	[*function*]
prove-proofchain	(*filename declname line origin retry?* &OPTIONAL *use-default-dp?*)	[*function*]
typechecked?	(*theory*)	[*function*]
rerun-proof-at?	(*name declname line* &OPTIONAL *origin rerun? unproved?*)	[*function*]

Proof Editing

edit-proof	*typecheck-file*, *typechecked?*†, *edit-proof-at*
install-proof	*install-proof*, *prove-proof-at*
display-proofs-formula	*display-proofs-formula-at*
display-proofs-theory	*collect-theories*†, *display-proofs-theory*
display-proofs-pvs-file	*display-proofs-pvs-file*
revert-proof	
remove-proof	*remove-proof-at*
show-proof-file	*show-proof-file*
show-orphaned-proofs	*show-orphaned-proofs*
show-proofs-theory	*show-proofs-theory*
show-proofs-pvs-file	*show-proofs-pvs-file*
show-proofs-importchain	*collect-theories*†, *show-proofs-importchain*
install-pvs-proof-file	*install-pvs-proof-file*
load-pvs-strategies	*read-strategies-files*
toggle-proof-prettyprinting	*toggle-proof-prettyprinting*
set-print-depth	
set-print-length	
set-rewrite-depth	
set-rewrite-length	

edit-proof-at (*filename declname line origin buffer prelude-offset full-label*) [*function*]

install-proof (*tmpfilename name declname line origin buffer prelude-offset*) [*function*]

prove-proof-at (*line step? display?*) [*function*]

display-proofs-formula-at (*name declname origin line*) [*function*]

display-proofs-theory (*theoryname*) [*function*]

display-proofs-pvs-file (*filename*) [*function*]

remove-proof-at (*name declname line origin*) [*function*]

show-proof-file (*context filename*) [*function*]

Given the *context* and PVS *filename*, this invokes *pvs-buffer* to create a `Proofs` buffer containing all the default proof scripts associated with the *filename*.

show-orphaned-proofs () [*function*]

Like

show-proofs-theory (*theoryname*) [*function*]

show-proofs-pvs-file (*file*) [*function*]

show-proofs-importchain (*theoryname*) [*function*]

install-pvs-proof-file (*filename*) [*function*]

read-strategies-files () [*function*]

toggle-proof-prettyprinting () [*function*]

Proof Information

show-current-proof	*call-show-proof*
explain-tcc	*call-explain-tcc*
show-last-proof	*show-last-proof*
ancestry	*call-ancestry*
siblings	*call-siblings*
show-hidden-formulas	*call-show-hidden*
show-auto-rewrites	*show-auto-rewrites*
show-expanded-sequent	*show-expanded-sequent*
show-skolem-constants	*show-skolem-constants*
pvs-set-proof-parens	
pvs-set-proof-prompt-behavior	
pvs-set-proof-default-description	

call-show-proof () [*function*]

call-explain-tcc () [*function*]

show-last-proof (&OPTIONAL *terse?*) [*function*]

call-ancestry () [*function*]

call-siblings () [*function*]

call-show-hidden () [*function*]

show-auto-rewrites () [*function*]

show-expanded-sequent (&OPTIONAL *all?*) [*function*]

show-skolem-constants () [*function*]

Adding and Modifying Declarations

add-declaration	*add-declaration-at*
modify-declaration	*modify-declaration-at*

add-declaration-at (*filename line*) [*function*]

modify-declaration-at (*filename line*) [*function*]

Prettyprint

prettyprint-theory	*collect-theories*†, *prettyprint-theory*
prettyprint-pvs-file	*prettyprint-pvs-file*
prettyprint-declaration	*prettyprint-region*
prettyprint-region	*prettyprint-region*

prettyprint-theory (*theoryname filename*) [*function*]

prettyprint-pvs-file (*filename*) [*function*]

prettyprint-region (*filename pos1* &OPTIONAL *(pos2 pos1)*) [*function*]

Viewing TCCs

show-tccs	*collect-theories*†, *show-tccs*
prettyprint-expanded	*collect-theories*†, *prettyprint-expanded*

show-tccs (*theoryref* &OPTIONAL *unproved-only?*) [*function*]

prettyprint-expanded (*theoryref*) [*function*]

Files and Theories

find-pvs-file	
find-theory	*collect-theories*†
view-prelude-file	
view-prelude-theory	
view-library-file	*library-files*†
view-library-theory	*library-theories*†
new-pvs-file	
new-theory	*collect-theories*†
import-pvs-file	
import-theory	
delete-pvs-file	*delete-pvs-file*
delete-theory	*collect-theories*†, *delete-theory*
save-pvs-file	
save-some-pvs-files	
smail-pvs-files	
rmail-pvs-files	
dump-pvs-files	
undump-pvs-files	
save-pvs-buffer	

library-files () [*function*]

library-theories () [*function*]

delete-pvs-file (*filename* &OPTIONAL *delete-file?*) [*function*]

Printing

pvs-print-buffer	
pvs-print-region	
print-theory	*collect-theories*†
print-pvs-file	
print-importchain	*collect-theories*†
alltt-theory	*collect-theories*†
alltt-pvs-file	
alltt-importchain	*collect-theories*†
alltt-proof	*alltt-proof*
latex-theory	*collect-theories*†, *latex-theory*
latex-pvs-file	*latex-pvs-file*
latex-importchain	*collect-theories*†, *latex-usingchain*
latex-proof	*latex-proof*
latex-theory-view	*collect-theories*†, *latex-theory-view*
latex-set-linelength	

alltt-proof (*file terse?*) [*function*]

latex-theory (*theoryname filename* &OPTIONAL *show-subst*) [*function*]

latex-pvs-file (*filename* &OPTIONAL *show-subst*) [*function*]

latex-usingchain (*theoryname filename* &OPTIONAL *show-subst*) [*function*]

latex-proof (*texfile* &OPTIONAL *terse? show-subst*) [*function*]

latex-theory-view (*theoryname filename viewer* &OPTIONAL *show-subst*) [*function*]

Display Commands

x-theory-hierarchy	*collect-theories*†, *x-module-hierarchy*
x-show-proof	*call-x-show-proof-at*
x-show-current-proof	*call-x-show-proof*
x-prover-commands	*x-prover-commands*

x-module-hierarchy (*theoryname* &OPTIONAL *include-libraries?*) [*function*]

call-x-show-proof-at (*filename declname line origin*) [*function*]

call-x-show-proof () [*function*]

x-prover-commands () [*function*]

Context

list-pvs-files	
list-theories	*context-files-and-theories*
change-context	*pvs-current-directory*†, *change-context*
save-context	*save-context*
pvs-remove-bin-files	
pvs-dont-write-bin-files	
pvs-do-write-bin-files	
context-path	*pvs-current-directory*

context-files-and-theories (&OPTIONAL *context*) [*function*]

change-context (*directory*) [*function*]

save-context () [*function*]

pvs-current-directory () [*function*]

Browsing

show-declaration *find-declaration* *whereis-declaration-used* *list-declarations* *show-expanded-form*	*show-declaration* *find-declaration* *whereis-declaration-used* *collect-theories*†, *list-declarations* *show-expanded-form*

show-declaration (*oname origin pos* &OPTIONAL *x?*) [*function*]

find-declaration (*string*) [*function*]

whereis-declaration-used (*oname origin pos* &OPTIONAL *x?*) [*function*]

list-declarations (*theoryref*) [*function*]

show-expanded-form (*oname origin pos1* &OPTIONAL *(pos2 pos1) all?*) [*function*]

Status

status-theory	*collect-theories*†, *status-theory*
status-pvs-file	*status-pvs-file*
status-importchain	*collect-theories*†, *status-importchain*
status-importbychain	*collect-theories*†, *status-importbychain*
show-theory-warnings	*collect-theories*†, *show-theory-warnings*
show-pvs-file-warnings	*show-pvs-file-warnings*
show-theory-messages	*collect-theories*†, *show-theory-messages*
show-pvs-file-messages	*show-pvs-file-messages*
show-theory-conversions	*collect-theories*†, *show-theory-conversions*
show-pvs-file-conversions	*show-pvs-file-conversions*
status-proof	*proof-status-at*
status-proof-theory	*collect-theories*†, *status-proof-theory*
status-proof-pvs-file	*status-proof-pvs-file*
status-proof-importchain	*collect-theories*†, *status-proof-importchain*
status-proofchain	*proofchain-status-at*
status-proofchain-theory	*collect-theories*†, *status-proofchain-theory*
status-proofchain-pvs-file	*status-proofchain-pvs-file*
status-proofchain-importchain	*collect-theories*†, *status-proofchain-importchain*

status-theory (*theoryref*) [*function*]

status-pvs-file (*filename*) [*function*]

status-importchain (*theory*) [*function*]

status-importbychain (*theory*) [*function*]

show-theory-warnings (*theoryname*) [*function*]

show-pvs-file-warnings (*filename*) [*function*]

show-theory-messages (*theoryname*) [*function*]

show-pvs-file-messages (*filename*) [*function*]

show-theory-conversions (*theoryname*) [*function*]

show-pvs-file-conversions (*filename*) [*function*]

proof-status-at (*filename declname line* &OPTIONAL (*origin* "pvs")) [*function*]

status-proof-theory (*theoryname*) [*function*]

status-proof-pvs-file (*filename*) [*function*]

status-proof-importchain (*theoryname*) [*function*]

proofchain-status-at (*filename declname line* &OPTIONAL (*origin* "pvs")) [*function*]

status-proofchain-theory (*theoryname*) [*function*]

status-proofchain-pvs-file (*filename*) [*function*]

status-proofchain-importchain (*theoryname*) [*function*]

Environment

whereis-pvs *pvs-version* *pvs-log* *pvs-mode* *status-display* *pvs-status* *remove-popup-buffer* *pvs* *pvs-load-patches* *pvs-interrupt-subjob* *reset-pvs*	*pvs-init* *load-pvs-patches*
report-pvs-bug	

pvs-init (&OPTIONAL *dont-load-patches* *dont-load-user-lisp*) [*function*]

load-pvs-patches () [*function*]

Exiting

suspend-pvs	*save-context*
exit-pvs	*exit-pvs*

exit-pvs () [*function*]

2.2.6 Other Lisp Functions invoked by Emacs

Aside from the PVS commands, there are Lisp functions invoked from Emacs indirectly, usually due to buffer-specific key bindings. Here we list such functions, under the associated buffers.

pvs-select-proof (*num*) [*function*]

pvs-view-proof (*num*)	[*function*]
pvs-delete-proof (*num*)	[*function*]
whereis-identifier-used (*string*)	[*function*]
status-proof-theories (*theories*)	[*function*]
theory-status-string (*theoryref*)	[*function*]
full-status-theory (*theoryname*)	[*function*]
show-all-proofs-file (*proofs outstr valid?*)	[*function*]
show-all-proofs-theory (*theory proofs outstr valid?*)	[*function*]
usedby-proofs (*bufname origin line*)	[*function*]
get-decl-at-origin (*bufname origin line*)	[*function*]
set-proofs-default (*line*)	[*function*]
proofs-delete-proof (*line*)	[*function*]
proofs-rename (*line id*)	[*function*]

proofs-show-proof (*line*)	[*function*]
proofs-change-description (*line description*)	[*function*]
proofs-rerun-proof (*line*)	[*function*]
proofs-edit-proof (*line*)	[*function*]
load-prelude-library (*lib* &OPTIONAL *force?*)	[*function*]
list-pvs-libraries ()	[*function*]
remove-prelude-library (*lib*)	[*function*]
list-prelude-libraries ()	[*function*]
latex-theory (*theoryname filename* &OPTIONAL *show-subst*)	[*function*]
latex-pvs-file (*filename* &OPTIONAL *show-subst*)	[*function*]
latex-usingchain (*theoryname filename* &OPTIONAL *show-subst*)	[*function*]
latex-proof (*texfile* &OPTIONAL *terse? show-subst*)	[*function*]
alltt-proof (*file terse?*)	[*function*]

latex-theory-view	(*theoryname filename viewer* &OPTIONAL *show-subst*)	[*function*]
latex-proof-view	(*texfile viewer* &OPTIONAL *terse? show-subst*)	[*function*]

2.2.7 ILISP

2.2.8 Batch Mode

2.2.9 The Tcl/Tk Interface

2.2.10 Debugging Emacs

There are many useful techniques for debugging Emacs code, some of these are listed in the GNU Emacs Lisp Reference Manual [**?**]. We will not discuss those techniques further; here we list some other techniques specific to PVS.

The `pvs` buffer shows the interaction between Emacs and Lisp. By default, it only shows the last 15000 characters; if more are needed set the Emacs global variable `comint-log-size` to a larger value.

2.3 Prover Interaction

The basic interaction of the prover is to print the `Rule?` prompt, read the input, process it, and print a commentary and the new current sequent. This is repeated until the proof is completed, or the user quits.

The input is similar to lisp, and consists of balanced parentheses, keywords (identifiers starting with a colon), identifiers, strings, etc. Note that the input is not processed until a proper form is sent. Thus it is important for parentheses and string quotes to be balanced properly. If running from Emacs with the ILISP interface (the default), the form is not sent to lisp until it is balanced. In the raw interface, it is read as soon as there is a balanced prefix. Hence with Emacs, `(grind))` will not be processed, while in raw mode, `grind` will be processed, followed by the warning

```
Warning: ignoring extra right parenthesis on
         #<TERMINAL-SIMPLE-STREAM [initial terminal io] fd 0/1>
```

When balanced input is received, it is checked to see that it is a valid prover command, and that the arguments are of the correct form.

The command is then processed, comentary is output, followed by the next sequent, if any.

2.3.1 Proof Commentary

Proof commentary consists of command commentary and proofstate commentary. The command commentary is specific to the given rule or strategy. In many cases, it is simply the `format` string applied to the given arguments. But it may consist of more, for example `assert` reports on the auto-rewrites that it does, depending on the settings of `*rewrite-print-depth*` and `*rewrite-print-length*`. Warnings and errors maya also be output. After the command commentary, proofstate commentary is output. This indicates the results of the preceding command, such as if there was no change, how many subgoals were generated, etc. Here is a complete list of such messages.

`which is trivially true.`	the current branch is proved
`this yields ~a subgoals:`	command generated subgoals
`this simplifies to:`	one subgoal was generated
`No change.  False TCCs:`	proofstate is unchanged because of false TCCs
`No change on:  ~s`	proofstate is unchanged
`Q.E.D.`	The formula is proved
`Failed!`	The formula is known to be false (rare)

For simple rules, the commentary is relatively easy to predict, but for complex derived rules and strategies, things get more complicated, especially if they use control based on failure and backtracking. In general, derived rules keep a stack of error messages, and only outputs the last few error messages if the command fails completely.

2.3.2 The Current Sequent

After the commentary, the formula and branch number of the current sequent are presented, and then the sequent itself, followed by the `Rule?` prompt. The formula and branch number is of the form

```
fname.1.2.2
```

where `fname` is the name of the formula being proved, and the numbers following indicate the branch the sequent is on. This may be followed by `(TCC)` if the sequent is a TCC branch.

2.3.3 An Example

The following example illustrates what can happen with `grind`:

```
FF :

  |-------
{1}   F(17, 17)
```

```
Rule? (grind)
Trying repeated skolemization, instantiation, and if-lifting,
this simplifies to:
FF :

  |-------
[1]   F(17, 17)

Rule? (grind)
The following errors occurred within the strategy:

Couldn't find a suitable quantified formula.

No suitable (+ve FORALL/-ve EXISTS) quantified expression found.

No change on: (grind)
FF :

  |-------
[1]   F(17, 17)

Rule?
```

Note that the first invocation of `grind` simply outputs the command commentary, and indicates that a single new subgoal was generated. Note that the sequent looks identical; the change is due to the updated state of the decision procedure, which is not directly visible. The second invocation results in no change, and the commentary attempts to explain why. In this case, `grind` invokes `bash`, whose definition is

```
(then (assert :let-reduce? let-reduce?) (bddsimp)
 (if if-match
     (let ((command
            (generate-instantiator-command if-match polarity?
              instantiator)))
       command)
   (skip))
 (repeat (then (skolem-typepred) (flatten)))
 (lift-if :updates? updates?))
```

The `lift-if` does not give an error, but the preceding `skolem-typepred` does, and it is that error that is made visible. Of course, all errors may be seen by running `grind$`.

2.3.4 Prover Interaction without Emacs

Though the prover was designed for interaction, there is nothing in its design or implementation that assumes a human using Emacs is part of the interaction. Thus in principle it should be possible to use the prover as the back end to a symbolic algebra package [1], or as a client to a web based proof tool. In this section we discuss some of the factors involved in using the prover as a back end. Note that this section does not deal with noninteractive back end use of the prover, as supported by the functions *prove-as-black-box* or *prove-formula-decl*, which take a formula and a strategy and simply run it in the backgroud.

The Emacs interface is quite simple; it just waits for the `Rule?` prompt, and sends the next input when it is seen. All other output is simply displayed, without attempting to match it. Unfortunately, external systems may want to separately display the sequent, the commentary, and the input, and determine the prooftree structure.

Recognizing the proof sequent itself would not be too difficult, but there is a much easier way than figuring out the pattern. The proofstate and associated sequent are actually printed using the Common Lisp *print-object* methods. New `:around` methods may easily be defined that make it simple to find the components of the proofstate. For example,

```
(defmethod print-object :around ((ps proofstate) stream)
  (format stream "~%-------Proofstate Start--------")
  (call-next-method)
  (format stream "~%-------Proofstate End--------"))

(defmethod print-object :around ((sequent sequent) stream)
  (format stream "~%-------Sequent Start--------")
  (call-next-method)
  (format stream "~%-------Sequent End--------"))
```

With these definitions, the output looks like

```
Rule? (grind)
Trying repeated skolemization, instantiation, and if-lifting,
this simplifies to:
-------Proofstate Start--------
FF.1 :

-------Sequent Start--------
[-1]  F(17, 16)
  |-------
[1]   F(17, 17)
```

```
-------Sequent End--------
-------Proofstate End--------
Rule?
```

It is now obvious how to find the branching number and the sequent.

2.3.5 Proof Tree Display

Creating a prooftree corresponding to the proofstate is not as straightforward. The branching number helps, but it only changes when more than one subgoal is generated. Also this doesn't help much when `undo` or `postpone` are used.

The current proof tree display is generated by creating Tcl/Tk commands that are interpreted using the `pvs-support.tcl` file loaded into a Tcl/Tk subprocess of Emacs. The lisp functions for supporting this are currently closely integrated with the specific Tcl/Tk functions. Rather than try and describe those here, we will in the future separate the code that determines the prooftree structure from the Tcl/Tk specific code. The generic code will be defined so that it will be easy to provide hooks to alternate interfaces.[1]

`*start-proof-display*` [*global*]

If this is set to `t` at the beginning of a a proof, then the prover will invoke *`call-x-show-proof`* to initialize the proof display. Note that *`call-x-show-proof`* is also invoked by the Emacs commands *`x-show-current-proof`* and *`x-show-proof`*.

`*displaying-proof*` [*global*]

A list of integers, representing the proofs currently being displayed. Multiple proofs may be dislpayed, and each is given a unique integer identifier.

`*current-displayed*` [*global*]

The proofstate currently being displayed. See *`display-proofstate`* for its use.

`*flush-displayed*` [*global*]

When undo is invoked, `*flush-displayed*` is set to the resulting proofstate.

`call-x-show-proof` () [*function*]

[1]The recent releases of Allegro Common Lisp have builtin Gtk functions, which we intend to make use of, as it is a better alternative, since the functions are in Lisp, and do not require the Emacs or Tcl/Tk processes in order to run.

display-proofstate (*proofstate*) [*function*]

This function is called by the *proofstepper* when the proofstate is not a strat-proofstate, and either it is a root or it has an associated rule or input. Essentially, this means that the proofstate is stable, and the prover is not currently working on a state that may potentially fail and disappear. In addition, this function checks that `*in-apply*` is `nil`, and that `*displaying-proof*` and `*current-displayed*` are set.
The given *proofstate* is compared to `*current-displayed*` and if they are different, *dislpay-proof-from* is invoked on `*flush-displayed*` if it is set, and `*current-displayed*` otherwise. `*flush-displayed*` is set to `nil`.

display-proof-from (*ps*) [*function*]

This function creates a file with a series of Tcl/Tk commands that results in updating the prooftree display. See Appendix B for details.

wish-done-proof (*proofstate*) [*function*]

When `*in-apply*` is `nil`, and `*displaying-proof*` is set, this sends the form

```
proof-done fid thid path
```

to indicate that that path of the proof tree is complete.

If you wish to create a different form of display than Tcl/Tk under Emacs, probably the simplest thing to do is to have PVS generate the forms above, and interpret them in a different display engine. To do this, set `*start-proof-display*` to `t`, redefine *pvs-wish-source* to simply print the file name in an easily interceptable form, and read the files to create your own display. A complete example is given in Appendix B

2.4 Replacing Emacs with a Different Interface

Here we discuss what is needed to replace Emacs with another interface. There are several reasons that one might want to do this. For example, to provide PVS as a proof engine within a system that already has a user interface, as was done in the Maple interface [**?**]. Another reason is to provide an interface on an operating system that does not directly support PVS, and use remote procedure calls to interact. PVS could also act as a client on a web server.

2.4.1 Invoking PVS without Emacs

The `pvs` shell script takes a `-raw` flag that starts up the PVS lisp image directly. However, some initialization must be done before the image is ready. In particular,

the package should be set to `:pvs`, and *`pvs-init`* must be invoked.

In this mode, PVS acts as a standard interactive unix process, reading `stdin`, and outputing to `stdout` and `stderr`.

Once the pvs process is started, it prints out the prompt as described above, and waits for input.

2.4.2 Remote PVS Interface

In some cases it may be desirable to use PVS remotely, for example if one wants to use a windows machine and connect to an interface defined there.

Doing this requires setting up an RPC connection, as described in http://www.franz.com/support/documentation/6.2/doc/rpc.htm.

2.4.3 PVS as a Client

Chapter 3

The Middle: Internal Structures and Functions

As described in the PVS prover guide [5], strategies are defined using the Lisp macro `defstep`. Simple strategies are quite easy to write, for instance a strategy that repeatedly tries to `grind` and `apply-extensionality` may be written as

```
(repeat (then (grind) (apply-extensionality)))
```

More complex strategies need to analyze the formulas of the sequent, create new terms, and build proof commands for subsequent prover invocation. The `let` and `if` strategies provide the basic support for this, but the key parts of these strategies are arbitrary Lisp functions. One of the purposes of this document is to describe the functions that are most useful for writing complex, domain-specific strategies.

In addition, we will describe techniques for writing robust strategies, providing useful error messages, and debugging.

A familiarity with Lisp is assumed, including the Common Lisp Object System (CLOS) [**?**], on which most of the PVS data structures are build, including terms, types, formulas, theories, sequents, and proofstates.

3.1 PVS Classes

In CLOS, there are classes and generic functions. Classes form a hierarchy and define slots. Generic functions may be applied to arguments of specific classes. Thus the type of an expression my be obtained by applying the generic function `type` to the expression. If `type` is applied to a type expression or a theory, it results in a Lisp error, as it is not defined on instances of those classes. Note that this approach to objects differs from the usual, in that methods are not viewed as belonging to a class. This has the advantage that binary methods can be easily defined.

defcl (***name classes*** &REST ***args***) [*macro*]

In PVS, classes are defined using the `defcl` macro, which in addition to defining the class, creates a recognizer for the class, updates the `*slot-info*` global variable, and defines ***untc**** methods. Thus

```
(defcl number-expr (expr)
  number)
```

creates the `number-expr` class as a subclass of the `expr` class, with the slot `number` (and whatever slots are defined by `expr` and its superclasses). Slots defined with `defcl` also have accessor, initarg, and initform values defined. In addition, the recognizer `number-expr?` is defined. It returns `t` when applied to an object of class `number-expr` and `nil` otherwise.

`*slot-info*` [*global*]

The descriptions of the classes given below are intensionally incomplete, as it is complex enough without giving all the details.

Similarly, many of the slots used for secondary purposes such as caching information are not described, though they will be seen if the Lisp `describe` function is used.

A *mixin* class is one that is intended to serve as a common superclass for more specialized classes, and not intended to have instances created for it. A mixin serves several purposes:

- it collects common slots into a single class, as with the `declaration` class,
- it provides a general type for the class, for example, the `declaration?` test is often used,
- it allows methods to be no more specialized than they need be, and
- even when methods need to be specialized, it often is the case that a method may be defined for the mixin, and `call-next-method` used for uniform processing of the common aspects of the specialized classes.

Similarly, many subclasses primarily exist for the purpose of prettyprinting—for example, the difference between `x+1` and `+(x,1)` is in the classes each instance is associated with. Both are `application` instances, though the former is actually an instance of the subclass `infix-application`. This makes no difference to the typechecker or the prover, only to the various display tools (e.g., the prettyprinter and LaTeXprinter). A *display* class is thus one that instances may be created for, but rarely should methods be specialized on these classes, unless, of course, they are concerned with prettyprinting or other forms of display.

In the following sections we list all classes, their supertypes, and their slots. For each class, we indicate whether it is a mixin or display class. Slots may be marked with an asterisk (*), indicating they are set during parsing.

3.1.1 Syntax Class

`syntax` $\subset \emptyset$ [*mixin class*]

pvs-sxhash-value a fixnum

This is a mixin class, for objects associated with the abstract syntax. The *pvs-sxhash-value* is the value of calls to `pvs-sxhash`, unless free variables occur in the term.

3.1.2 Specification Classes

Specification ::= { *Theory* | *Datatype* } $^+$

Theory ::= *Id* [*TheoryFormals*] `:` `THEORY`
[*Exporting*]
`BEGIN`
[*AssumingPart*]
[*TheoryPart*]
`END` *Id*

TheoryFormals ::= `[` *TheoryFormal* $^+_{;}$ `]`

TheoryFormal ::= [`(` *Importing* `)`] *TheoryFormalDecl*

TheoryFormalDecl ::= *TheoryFormalType*
| *TheoryFormalConst*
| *TheoryFormalTheory*

TheoryFormalType ::= *Ids* `:` { `TYPE` | `NONEMPTY_TYPE` | `TYPE+` }
[`FROM` *TypeExpr*]

TheoryFormalConst ::= *IdOps* `:` *TypeExpr*

TheoryFormalTheory ::= *IdOps* `:` `THEORY` *TheoryDeclName*

`datatype-or-module` ⊂ `syntax`	[*mixin class*]
*id**	a symbol, the identifier of the top-level recursive type or theory
*formals**	a list of `formal-decl`s and/or `importing`s
formals-sans-usings	a list of `formal-decl`s
*assuming**	a list of `declaration`s and/or `importing`s
filename	a string, the base name of the file (without directory or extension)
status	a list of symbols, `'parsed` and/or `'typechecked`
generated-by	an identifier if this was generated by a theory or recursive type, nil otherwise
tcc-comments	an association list declarations to TCC strings representing trivial or subsumed TCCs
info	a list of strings
warnings	a list of strings
conversion-messages	a list of strings
all-declarations	the list of declarations (no importings) from the formals, assumings, and theory part of the module
all-imported-theories	a list of all the imported theories
all-imported-names	a list of modnames corresponding to the *all-imported-theories*
places	an `eq` hash-table providing the place for each subterm of this module

This class is a superclass of `module` and `recursive-type`. These are the top level entities contained in a PVS file.

Recursive Types

This section describes the recursive types. In PVS 3.0 an implementation of an experimental version of codatatypes (final coalgebras) was introduced, though they were not really announced, since they are going to change. The syntax is currently the same as for datatypes, but using the keyword `CODATATYPE`. This is definitely going to change in the future, though the underlying data structures may be retained.

Datatype ::= *Id* [*TheoryFormals*] : DATATYPE [WITH SUBTYPES *Ids*]
BEGIN
[*Importing* [;]]
[*AssumingPart*]
DatatypePart
END *Id*

InlineDatatype ::= *Id* : DATATYPE [WITH SUBTYPES *Ids*]
BEGIN
[*Importing* [;]]
[*AssumingPart*]
DatatypePart
END *Id*

DatatypePart ::= { *Constructor* : *IdOp* [: *Id*] } $^{+}$

Constructor ::= *IdOp* [({ *IdOps* : *TypeExpr* }$^{+}_{;}$)]

`recursive-type` ⊂ `datatype-or-module` [*mixin class*]

importings*	a list of importings
constructors*	a list of `adt-constructors`
adt-type-name	a type name constructed for the recursive type
adt-theory	the generated (co-)datatype theory
adt-map-theory	the generated map theory (nil if not generated)
adt-reduce-theory	the generated reduce theory
generated-file-date	a date in universal time
positive-types	a list of `type-names` corresponding to the positive type parameters of the generated ***adt-theory***
semi*	flag indicating whether the recursive type was followed by a semi-colon

The `recursive-type` is the mixin for datatypes and co-datatypes.

`recursive-type-with-subtypes` ⊂ `recursive-type` [*mixin class*]

subtypes	a list of identifiers

This is the mixin for `recursive-types` that are declared `WITH SUBTYPES`.

`inline-recursive-type` ⊂ `recursive-type` [*mixin class*]

generated	the list of declarations generated during typechecking. These are the declarations that go in the generated adt-theory of a top level recursive type
typechecked?	a flag indicating whether the recursive type has been typchecked

This is the mixin for recursive types declared inside a theory. In many respects they are like a declaration.

Datatypes

`datatype` ⊂ `recursive-type` [*class*]

This is the datatype subclass of recursive types.

`inline-datatype` ⊂ `inline-recursive-type` `datatype` [*class*]

The inline version of datatypes.

`datatype-with-subtypes` ⊂ `recursive-type-with-subtypes` `datatype` [*class*]

The datatype with subtypes

`inline-datatype-with-subtypes` ⊂ `inline-datatype` `datatype-with-subtypes` [*class*]

`enumtype` ⊂ `inline-datatype` [*class*]

Codatatypes

`codatatype` ⊂ `recursive-type` [*class*]

`inline-codatatype` ⊂ `inline-recursive-type` `codatatype` [*class*]

`codatatype-with-subtypes` ⊂ `codatatype` `recursive-type-with-subtypes` [*class*]

`inline-codatatype-with-subtypes` ⊂ `inline-codatatype` `codatatype-with-subtypes` [*class*]

Recursive Type Constructors

`adt-constructor` ⊂ `syntax` [*class*]

- ***recognizer*** a symbol
- ***ordnum*** .... a number

`constructor-with-subtype` ⊂ `simple-constructor` [*class*]

- ***subtype***

`simple-constructor` ⊂ `adt-constructor` [*class*]

- ***id*** ..... a symbol
- ***arguments*** a list of
- ***con-decl*** . the generated constructor const-decl
- ***rec-decl*** . the generated recognizer const-decl
- ***acc-decls*** the list of generated accessor const-decls

Theories

`module` ⊂ `datatype-or-module` [*class*]

theory	the declarations of the theory part
exporting	an exporting
nonempty-types	a list of type expressions
all-usings	a list of modnames
immediate-usings	a list of modnames
instances-used	a list of modnames
assuming-instances	a list of modnames
used-by	a list of symbols
saved-context	a context
dependent-known-subtypes	a list of type expressions
macro-expressions	a list of expressions
tccs-tried?	a flag
modified-proof?	a flag
tcc-info	a list of four integers
ppe-form	a string
tcc-form	a string
typecheck-time	an integer

Library Theories and Recursive Types

`library-datatype-or-theory` ⊂ `datatype-or-module` [*class*]

lib-ref	a string

`library-datatype` ⊂ `datatype` `library-datatype-or-theory` [*class*]

`library-codatatype` ⊂ `codatatype` `library-datatype-or-theory` [*class*]

`library-theory` ⊂ `module` `library-datatype-or-theory` [*class*]

Interpreted Theories

`theory-interpretation` ⊂ `module` [*class*]

from-theory	a theory
mapping	a mapping

Importings and Exportings

Exporting	::=	`EXPORTING` *ExportingNames* [`WITH` *ExportingTheories*]
ExportingNames	::=	`ALL` [`BUT` *ExportingName*$^{+}_{,}$]
	|	*ExportingName*$^{+}_{,}$
ExportingName	::=	*IdOp* [`:` { *TypeExpr* | `TYPE` | `FORMULA` }]
ExportingTheories	::=	`ALL` | `CLOSURE` | *TheoryNames*
Importing	::=	`IMPORTING` *ImportingItem*$^{+}_{,}$
ImportingItem	::=	*TheoryName* [`AS` *Id*]

`exporting` ⊂ `syntax` [*class*]

names	a list of modnames
but-names	a list of modnames
kind.....	a symbol: nil, all, closure, or default
modules..	a list of modnames
closure..	a list of modnames

`expname` ⊂ `syntax` [*class*]

id..	a symbol
kind	a symbol
type	a type expression

`importing` ⊂ `syntax` [*class*]

theory-name semi chain? refers-to generated tcc-form saved-context

3.1.3 Declaration Classes

TheoryPart	::=	{*TheoryElement* [`;`]}$^{+}$
TheoryElement	::=	*Importing* | *Decl*
Decl	::=	*LibDecl* | *TheoryDecl* | *TypeDecl* | *VarDecl*
	|	*ConstDecl* | *RecursiveDecl* | *MacroDecl* | *InductiveDecl*
	|	*InductiveDecl* | *FormulaDecl* | *Judgement* | *Conversion*
	|	*InlineDatatype* | *AutoRewriteDecl*

LibDecl ::= *Ids* : LIBRARY [=] *String*

TheoryDecl ::= *Ids* : THEORY = *TheoryDeclName*

TypeDecl ::= *Id* [{, *Ids*} | *Bindings*] :
{TYPE | NONEMPTY_TYPE | TYPE+}
[{= | FROM} *TypeExpr* [CONTAINING *Expr*]]

VarDecl ::= *IdOps* : VAR *TypeExpr*

ConstDecl ::= *IdOp* [{, *IdOps* } | *Bindings*$^+$] : *TypeExpr* [= *Expr*]

RecursiveDecl ::= *IdOp* [{, *IdOps* } | *Bindings*$^+$] : RECURSIVE
TypeExpr = *Expr* MEASURE *Expr* [BY *Expr*]

MacroDecl ::= *IdOp* [{, *IdOps* } | *Bindings*$^+$] : MACRO
TypeExpr = *Expr*

InductiveDecl ::= *IdOp* [{, *IdOps* } | *Bindings*$^+$] : INDUCTIVE
TypeExpr = *Expr*

CoInductiveDecl ::= *IdOp* [{, *IdOps* } | *Bindings*$^+$] : COINDUCTIVE
TypeExpr = *Expr*

FormulaDecl ::= *Ids* : *FormulaName Expr*

Judgement ::= *SubtypeJudgement* | *ConstantJudgement*

SubtypeJudgement ::= [*IdOp* :] JUDGEMENT *TypeExpr*$^+_,$ SUBTYPE_OF *TypeExpr*

ConstantJudgement ::= [*IdOp* :] JUDGEMENT *ConstantReference*$^+_,$
HAS_TYPE *TypeExpr*

ConstantReference ::= *Name Bindings**

Conversion ::= {CONVERSION | CONVERSION+ | CONVERSION-} *Expr*$^+_,$

AutoRewriteDecl ::= {AUTO_REWRITE | AUTO_REWRITE+ | AUTO_REWRITE-}
RewriteName$^+_,$

RewriteName ::= *Name* [! [!]] [: {*TypeExpr* | *FormulaName*}]

Bindings ::= (*Binding*$^+_,$)

Binding ::= *TypedId* | { (*TypedIds*) }

TypedIds ::= *IdOps* [: *TypeExpr*] [| *Expr*]

TypedId ::= *IdOp* [: *TypeExpr*] [| *Expr*]

declaration ⊂ syntax [*class*]
 newline-comment
 id.....
 formals.....
 module.....
 refers-to.....
 referred-by....
 chain?.....
 typechecked?...
 visible?.....
 generated.....
 generated-by...
 semi.....
 tcc-form.....
 typecheck-time.

typed-declaration ⊂ declaration [*class*]
 declared-type.....
 declared-type-string
 type.....

Type Declarations

type-decl ⊂ declaration [*class*]
 type-value

nonempty-type-decl ⊂ type-decl [*class*]
 keyword

type-def-decl ⊂ type-decl [*class*]
 type-expr
 contains.

nonempty-type-def-decl ⊂ type-def-decl nonempty-type-decl [*class*]

type-eq-decl ⊂ type-def-decl [*class*]

nonempty-type-eq-decl ⊂ type-eq-decl nonempty-type-def-decl [*class*]

type-from-decl ⊂ type-def-decl [*class*]

nonempty-type-from-decl ⊂ type-from-decl nonempty-type-def-decl [*class*]

Formal Parameter Declarations

formal-decl ⊂ declaration [*class*]
dependent?

formal-type-decl ⊂ formal-decl type-decl typed-declaration [*class*]

formal-nonempty-type-decl ⊂ formal-type-decl nonempty-type-decl [*class*]

formal-subtype-decl ⊂ formal-type-decl type-from-decl [*class*]

formal-nonempty-subtype-decl ⊂ formal-subtype-decl nonempty-type-decl [*class*]

formal-const-decl ⊂ formal-decl typed-declaration [*class*]
possibly-empty-type?

formal-theory-decl ⊂ formal-decl [*class*]
theory-name.....
generated-theory
saved-context...

Library Declarations

lib-decl ⊂ declaration [*class*]
lib-string
lib-ref...

Theory Declarations and Abbreviations

mod-decl ⊂ declaration [*class*]
modname.....
generated-theory
saved-context...

theory-abbreviation-decl ⊂ declaration [*class*]
theory-name.....
generated-theory
saved-context...

Variable Declarations

var-decl ⊂ typed-declaration	[*class*]

Constant Declarations

const-decl ⊂ typed-declaration	[*class*]
definition *def-axiom*. *eval-info*.	
macro-decl ⊂ const-decl	[*class*]
def-decl ⊂ const-decl	[*class*]
declared-measure... *ordering*..... *measure*..... *measure-depth*..... *recursive-signature*	

(Co-)Inductive Definitions

fixpoint-decl ⊂ const-decl	[*class*]
inductive-decl ⊂ fixpoint-decl	[*class*]
corecursive-decl ⊂ const-decl	[*class*]
coinductive-decl ⊂ fixpoint-decl	[*class*]

Formula Declarations

AssumingPart ::= `ASSUMING` {*AssumingElement* [`;`]}$^+$ `ENDASSUMING`

AssumingElement ::= *Importing*
| *Decl*
| *Assumption*

Assumption ::= *Ids* `:` `ASSUMPTION` *Expr*

formula-decl ⊂ declaration [*class*]
spelling.....
definition.....
closed-definition
kind.....
default-proof....
proofs.....

assuming-decl ⊂ formula-decl [*class*]
original-definition

TCC Declarations

tcc-decl ⊂ formula-decl [*class*]
tcc-disjuncts.....
importing-instance

subtype-tcc ⊂ tcc-decl [*class*]

termination-tcc ⊂ tcc-decl [*class*]

judgement-tcc ⊂ subtype-tcc [*class*]

existence-tcc ⊂ tcc-decl [*class*]

assuming-tcc ⊂ tcc-decl [*class*]
theory-instance.....
generating-assumption

mapped-axiom-tcc ⊂ tcc-decl [*class*]
theory-instance.
generating-axiom

cases-tcc ⊂ tcc-decl [*class*]

well-founded-tcc ⊂ tcc-decl [*class*]

same-name-tcc ⊂ tcc-decl [*class*]

cond-disjoint-tcc ⊂ tcc-decl [*class*]

cond-coverage-tcc ⊂ tcc-decl [*class*]

monotonicity-tcc ⊂ tcc-decl [*class*]

Judgement Declarations

judgement ⊂ typed-declaration [*class*]
free-parameters
free-parameter-theories

subtype-judgement ⊂ judgement [*class*]
declared-subtype
subtype

number-judgement ⊂ judgement [*class*]
number-expr

name-judgement ⊂ judgement [*class*]
name

application-judgement ⊂ judgement [*class*]
name
formals
judgement-type

Conversion Declarations

conversion-decl ⊂ declaration [*class*]
k-combinator?
expr
free-parameters
free-parameter-theories

conversionplus-decl ⊂ conversion-decl [*class*]

conversionminus-decl ⊂ conversion-decl [*class*]

Auto-rewrite Declarations

auto-rewrite-decl ⊂ declaration [*class*]
rewrite-names

auto-rewrite-plus-decl ⊂ auto-rewrite-decl [*class*]

auto-rewrite-minus-decl ⊂ auto-rewrite-decl [*class*]

Rewrite Names Rewrite names are used for auto rewrite declarations, as well as many prover commands. The prover allows for distinction between definitions, formula names, and formula sequent numbers, and between lazy, eager, and macro rewrites.

rewrite-elt ⊂ syntax [*mixin class*]

rewrite-name ⊂ rewrite-elt name [*mixin class*]

lazy-rewrite ⊂ rewrite-elt [*mixin class*]

eager-rewrite ⊂ rewrite-elt [*mixin class*]

macro-rewrite ⊂ rewrite-elt [*mixin class*]

constant-rewrite-name ⊂ rewrite-name [*mixin class*]
declared-type
type .....

formula-rewrite-name ⊂ rewrite-name [*mixin class*]
spelling

lazy-rewrite-name ⊂ rewrite-name lazy-rewrite [*class*]

eager-rewrite-name ⊂ rewrite-name eager-rewrite [*class*]

macro-rewrite-name ⊂ rewrite-name macro-rewrite [*class*]

lazy-constant-rewrite-name ⊂ constant-rewrite-name lazy-rewrite-name [*class*]

lazy-formula-rewrite-name ⊂ formula-rewrite-name lazy-rewrite-name [*class*]

eager-constant-rewrite-name ⊂ constant-rewrite-name eager-rewrite-name [*class*]

`eager-formula-rewrite-name` ⊂ `formula-rewrite-name` `eager-rewrite-name` [*class*]

`macro-constant-rewrite-name` ⊂ `constant-rewrite-name` `macro-rewrite-name` [*class*]

`macro-formula-rewrite-name` ⊂ `formula-rewrite-name` `macro-rewrite-name` [*class*]

`fnum-rewrite` ⊂ `rewrite-elt` [*mixin class*]
fnum

`lazy-fnum-rewrite` ⊂ `fnum-rewrite` `lazy-rewrite` [*class*]

`eager-fnum-rewrite` ⊂ `fnum-rewrite` `eager-rewrite` [*class*]

`macro-fnum-rewrite` ⊂ `fnum-rewrite` `macro-rewrite` [*class*]

3.1.4 Type Expression Classes

TypeExpr ::= *Name*
| *EnumerationType*
| *Subtype*
| *TypeApplication*
| *FunctionType*
| *TupleType*
| *CotupleType*
| *RecordType*

EnumerationType ::= `{` *IdOps* `}`

Subtype ::= `{` *SetBindings* `|` *Expr* `}`
| `(` *Expr* `)`

TypeApplication ::= *Name Arguments*

FunctionType ::= [`FUNCTION` | `ARRAY`]
`[` –[*IdOp* `:`] *TypeExpr* $''^{+}_{;}$ `->` *TypeExpr* `]`

TupleType ::= `[` –[*IdOp* `:`] *TypeExpr* $''^{+}_{;}$ `]`

CotupleType ::= `[` –[*IdOp* `:`] *TypeExpr* $''^{+}_{+}$ `]`

RecordType ::= `[#` *FieldDecls*$^{+}_{;}$ `#]`

FieldDecls ::= *Ids* `:` *TypeExpr*

`type-expr` ⊂ `syntax` [*mixin class*]
parens
print-type
from-conversion
free-variables
free-parameters
nonempty?

`type-name` ⊂ `type-expr` `name` [*class*]
adt

`adt-type-name` ⊂ `type-name` [*class*]
adt
recognizer-names
struct-name

`subtype` ⊂ `type-expr` [*class*]
supertype
top-type
predicate

`funtype` ⊂ `type-expr` [*class*]
domain
range

`tupletype` ⊂ `type-expr` [*class*]
types
generated?

`cotupletype` ⊂ `type-expr` [*class*]
types
generated?

`recordtype` ⊂ `type-expr` [*class*]
fields
dependent?

`field-decl` ⊂ `binding` `name` [*class*]
chain? a boolean (`t` or `nil`)

3.1.5 Expression Classes

Expressions are all of class `expr`. The recognizer for this class is `expr?`. In general, expressions may be created using `pc-parse` as described above. All expressions have

a `type` slot and a `types` slot. The `types` slot contains the possible types of the expression during typechecking, and when typechecking is completed, it contains the "best" judgement type. The `type` slot contains the actual type. Thus if the number expression `2` is typechecked, its `type` is `number`, but its `types` is the list containing `posint`.[1]

Expr ::= *Number*
| *String*
| *Name*
| *Id* `!` *Number*
| *Expr Arguments*
| *Expr Binop Expr*
| *Unaryop Expr*
| *Expr* `‘` *–Id* | *Number* `″`
| `(` *Expr*$^{+}_{,}$ `)`
| `(:` *Expr*$^{*}_{,}$ `:)`
| `[|` *Expr*$^{*}_{,}$ `|]`
| `(|` *Expr*$^{*}_{,}$ `|)`
| `{|` *Expr*$^{*}_{,}$ `|}`
| `(#` *Assignment*$^{+}_{,}$ `#)`
| *Expr* `::` *TypeExpr*
| *IfExpr*
| *BindingExpr*
| `{` *SetBindings* `|` *Expr* `}`
| `LET` *LetBinding*$^{+}_{,}$ `IN` *Expr*
| *Expr* `WHERE` *LetBinding*$^{+}_{,}$
| *Expr* `WITH [` *Assignment*$^{+}_{,}$ `]`
| `CASES` *Expr* `OF` *Selection*$^{+}_{,}$ [`ELSE` *Expr*] `ENDCASES`
| `COND` { *Expr* `->` *Expr* }$^{+}_{,}$ [`,` `ELSE` `->` *Expr*] `ENDCOND`
| *TableExpr*

[1] This will change in the near future, as judgement types are going to be handled differently.

IfExpr ::= `IF` *Expr* `THEN` *Expr*
`{` `ELSIF` *Expr* `THEN` *Expr* `}` * `ELSE` *Expr* `ENDIF`

BindingExpr ::= *BindingOp* *LambdaBindings* `:` *Expr*

BindingOp ::= `LAMBDA` | `FORALL` | `EXISTS` | `{` *IdOp* `!` `}`

LambdaBindings ::= *LambdaBinding* [[`,`] *LambdaBindings*]

LambdaBinding ::= *IdOp* | *Bindings*

SetBindings ::= *SetBinding* [[`,`] *SetBindings*]

SetBinding ::= {*IdOp* [`:` *TypeExpr*] } | *Bindings*

Assignment ::= *AssignArgs* { `:=` | `|->` } *Expr*

AssignArgs ::= *Id* [`!` *Number*]
| *Number*
| *AssignArg*$^+$

AssignArg ::= `(` *Expr*$^+_{,}$ `)`
| `‘` *Id*
| `‘` *Number*

Selection ::= *IdOp* [`(` *IdOps* `)`] `:` *Expr*

TableExpr ::= `TABLE` [*Expr*] [`,` *Expr*]
[*ColHeading*]
TableEntry$^+$ `ENDTABLE`

ColHeading ::= `|[` *Expr* { `|` { *Expr* | `ELSE` } } $^+$ `]|`

TableEntry ::= { `|` [*Expr* | `ELSE`] } $^+$ `||`

LetBinding ::= { *LetBind* | `(` *LetBind*$^+_{,}$ `)` } `=` *Expr*

LetBind ::= *IdOp* *Bindings** [`:` *TypeExpr*]

Arguments ::= `(` *Expr*$^+_{,}$ `)`

`expr` ⊂ `syntax` [*mixin class*]
type the type of the expression
. . . . . .
This is a mixin for expressions, as opposed to type expressions.

`name` ⊂ `syntax` [*mixin class*]

id	the identifier
mod-id	a theory identifier
library	a library identifier
actuals	a list of actual parameters
mappings	a list of mappings
resolutions	a list of resolutions, singleton if typechecked

This is a mixin for names, i.e., `name-exprs`, `type-names`, `modnames`. The *id* is always given, and if typechecked, the `resolutions` is set to a single resolution; during typechecking, or in a call to `resolve` `resolutions` may contain many resolutions. All other slots are optional, and will not be filled in by the typechecker. So given the names

```
foo
foo[x]
th!foo
lib@th[x]{{y := 3}}!foo
```

In each case, the associated `name` will have an *id* of `foo`, the second and last one will have *actuals* They are used by the typechecker to help determine a unique resolution. The `id` slot is the identifier for the name. A resolution consists of a declaration and a theory instance, as well as caching the type. A typechecked name expression has only a single resolution, while a partially typechecked one may have many. Note that the `type` slot is redundant in this case, as the type of the resolution is identical to the `type` of the name expression.

`name-expr` ⊂ `name` `expr` [*class*]

Name expressions.

`field-application` ⊂ `expr` [*class*]

id	identifier
actuals	a list of actuals
argument	the argument

A field application is the internal representation for record extraction, e.g., `r‘a` or `a(r)`. In this case, `a` is the `id`, `r` is the `argument`, and the actuals is `nil`.

`projection-expr` ⊂ `name-expr` [*class*]

index	the index number, an integer

A `projection-expr` is of the form `PROJ_`n, where n is an integer greater than 0. The `id` in this case is `PROJ_`n, and the `index` is n. Projection-exprs also allow actuals, e.g., `PROJ_2[[int, boo, int]]` has an `actual` consisting of a list with a tuple type element.

`projection-application` ⊂ `expr` [*class*]

id.....	identifier
index...	the index number, an integer
actuals.	a list of actuals
argument	the argument

This is the class for projections of the form `t‘2` or `PROJ_2(t)`. These are not applications, because there is no declaration associated with, e.g., `PROJ_2`. The form `t‘2` is actually of the subclass `projappl`, but this subclass should only matter for prettyprinting and related purposes.

`adt-name-expr` ⊂ `name-expr` [*class*]

adt-type	a type name

A mixin for the `constructor-name-expr`, `recognizer-name-expr`, and `accessor-name-expr` classes. During typechecking, if a `name-expr` is determined to be a constructor, recognizer, or accessor of a datatype, it is changed to the corresponding class (using `change-class`). The `adt-type` slot is the corresponding type-name, and is fully instantiated. However, it is computed lazily, so do not access it directly, instead use the `adt` generic function.

`constructor-name-expr` ⊂ `adt-name-expr` [*class*]

recognizer-name	associated recognizer
accessor-names.	associated accessors

This is an `adt-name-expr` (see above). The `recognizer-name` is the associated recognizer, fully instantiated, and the `accessor-names` are similarly the associated accessors. Both these slots are computed lazily, use the `recognizer` and `accessors` functions instead of directly referencing these slots.

`recognizer-name-expr` ⊂ `adt-name-expr` [*class*]

constructor-name	associated constructor
unit?.....	a unit recognizer flag

This is an `adt-name-expr` (see above). The `constructor-name` is lazy, as described above, and should be gotten using the `constructor` function, not directly. The `unit?` slot indicates whether this is a unit, i.e., whether the associated constructor has arguments. This is used by the prover to handle enumeration types.

`accessor-name-expr` ⊂ `adt-name-expr` [*class*]

This is an `adt-name-expr` (see above). It is important to note that there is no associated constructor. This is because a given accessor may be "shared" with many constructors. For example, given the datatype

```
prop: DATATYPE
 BEGIN
  p: p?
  and(x1, x2: prop): and?
  or(x1, x2: prop): or?
  not(x1: prop): not?
 END prop
```

There is only one declaration generated for `x1`:

```
x1: [{x: prop | and?(x) OR or?(x) OR not?(x)} -> prop]
```

and this clearly is not associated with any particular constructor.

`injection-expr` ⊂ `constructor-name-expr` [*class*]

index the index number, an integer

This is similar to the projection-expr class, but for cotuples, where it acts more like a constructor (hence is a subclass of `constructor-name-expr`).

`injection?-expr` ⊂ `recognizer-name-expr` [*class*]

index the index number, an integer

This is similar to the projection-expr class, but for cotuples, where it acts more like a recognizer (hence is a subclass of `recognizer-name-expr`).

`extraction-expr` ⊂ `accessor-name-expr` [*class*]

index the index number, an integer

This is similar to the projection-expr class, but for cotuples, where it acts more like an accessor (hence is a subclass of `accessor-name-expr`).

`injection-application` ⊂ `expr` [*class*]

id.....	identifier
index...	the index number, an integer
actuals.	a list of actuals
argument	the argument

This is similar to the `projection-applcation` class, but for cotuple injections. It does not really correspond to an application, because like projections, there is no declaration associated with an injection.

`injection?-expr` ⊂ `recognizer-name-expr` [*class*]

id.....	identifier
index...	the index number, an integer
actuals.	a list of actuals
argument	the argument

This is similar to the `projection-applcation` class, but for cotuple injection recognizers. It does not really correspond to an application, because like projections, there is no declaration associated with an injection recognizer.

`extraction-expr` ⊂ `accessor-name-expr` [*class*]

id.....	identifier
index...	the index number, an integer
actuals.	a list of actuals
argument	the argument

This is similar to the `projection-applcation` class, but for cotuple extractions. It does not really correspond to an application, because like projections, there is no declaration associated with an extraction.

`number-expr` ⊂ `expr` [*class*]

number	a nonnegative integer

This is the class for numbers. Negative numbers are represented as applications, with a `-` operator, so the `number` is always positive. Note that during typechecking, a `number-expr` instance may be changed to a `name-expr`, if it turns out the number has an associated declaration. This is because numbers may be overloaded the same as names, and any number which is overloaded in this way must be treated as a `name-expr`. Typechecking a `number-expr` always either sets the type to `real`, or changes it to a `name-expr`.
Lisp uses a bignum representation for integers they get beyond the word size, so this is accurate even for large integers.

`tuple-expr` ⊂ `expr` [*class*]
exprs a list of expressions

This is the class for tuple expressions, and the elements are in the `expr` list. In PVS, tuple expressions always have at least two elements; 0-tuples are not allowed, and 1-tuples are really just parenthesized expressions. Thus the type of a `tuple-expr` instance is always a `tuptype`, or a subtype of one.

`record-expr` ⊂ `expr` [*class*]
assignments a list of assignments

This corresponds to a record expression of the form `(# a`$_1$`:T`$_1$`, ..., a`$_n$`:T`$_n$ `#)`.

`cases-expr` ⊂ `expr` [*class*]
expression selections else-part

`selection` ⊂ `syntax` [*class*]
constructor args expression

`unpack-expr` ⊂ `cases-expr` [*class*]

`in-selection` ⊂ `selection` [*class*]
index

`application` ⊂ `expr` [*class*]
operator an expr
argument an expr

`string-expr` ⊂ `application` [*class*]
string-value a Lisp string

A string-expr in PVS is really an application of `list2finseq` to a list of characters. But it is often much more efficient to define methods on this class directly.

`propositional-application` ⊂ `application` [*class*]

A mixin.

`negation` ⊂ `propositional-application` [*class*]

When an application has an operator that resolves to the booleans `NOT`, it is changed to this class. This is useful for defining methods. There is a `unary-negation` subclass that is there for prettyprinting and related purposes. Thus `NOT A` is a unary negation, while `NOT(A)` is a negation (assuming `NOT` is from the booleans theory).

`conjunction` ⊂ `propositional-application` [*class*]

When an application has an operator that resolves to the booleans `AND`, it is changed to this class. This is useful for defining methods. There is a `infix-conjunction` subclass that is there for prettyprinting and related purposes. Thus `A AND B` is a unary negation, while `AND(A, B)` is a conjunction (assuming `AND` is from the booleans theory). Note that `&` is an alias for `AND` when it resolves to the boolean `&`, and also leads to a conjunction.
Note that technically if `BB` has type `[bool, bool]`, then `AND(BB)` is a perfectly valid conjunction. In practice, though, everyone writing methods specializing on this class assumed there were two arguments, and getting occasional Lisp errors as a result. Thus when converting to a conjunction class, the typechecker also ensures there are two arguments, by converting this to `AND(BB‘1, BB‘2)`.

`disjunction` ⊂ `propositional-application` [*class*]

Exactly as for `conjunction`s, but with `OR` instead of `AND`, and `infix-disjunction` as the corresponding subclass.

`implication` ⊂ `propositional-application` [*class*]

Exactly as for `conjunction`s, but with `IMPLIES` and `=>` instead of `AND` and `&`, and `infix-implication` as the corresponding subclass.

`iff-or-boolean-equation` ⊂ `application` [*class*]

This class is a mixin. If `A` and `B` are booleans, then `A IFF B` and `A = B` are logically equivalent, and it is sometimes useful to have methods that don't make a distinction. As described below, these do belong to distinct subclasses, so distinctions can be made if it is desirable. For example, the prover `prop` command will case-split on `IFF`, but not on `=`.
Note that this is not a subclass of `propositional-application`.

`iff` ⊂ `iff-or-boolean-equation` [*class*]

Exactly as for `conjunction`s, but with `IFF` instead of `AND`, and `infix-iff` as the corresponding subclass.

`equation` ⊂ `application` [*class*]

When an application has an operator that resolves to the `equalities` theory declaration `=`, it is changed to this class. There is an `infix-equation` subclass that is there for prettyprinting and related purposes.

`boolean-equation` ⊂ `iff-or-boolean-equation` `equation` [*class*]

This class is for the special case where the application is an `equation` as above, and the arguments are boolean. The infix subclass is `infix-boolean-equation`.

`disequation` ⊂ `application` [*class*]

When an application has an operator that resolves to the `notequal` theory declaration `/=`, it is changed to this class. There is an `infix-disequation` subclass that is there for prettyprinting and related purposes.

The following classes relate to IF expressions. The following table should explain the differences. Given a declaration `IF: [int, int, int -> int]`, in addition to the usual `IF` declared in the prelude, the following shows which class each kind of expression belongs to. The `if-expr` and `mixfix-branch` classes are useful only for prettyprinting and related methods.

Expression	Class
`IF(1, 3, 7)`	`application`
`IF 1 THEN 3 ELSE 7 ENDIF`	`if-expr`
`IF(TRUE, 3, 7)`	`branch`
`IF TRUE THEN 3 ELSE 7 ENDIF`	`mixfix-branch`

`branch` ⊂ `application` [*class*]

When an application has an operator that resolves to the `if_def` theory declaration `IF`, it is changed to this class. There are `mixfix-branch` and `chained-branch` subclasses associated with this. The `mixfix-branch` class simply means it had the `IF-THEN-ELSE-ENDIF` form, and the `chained-branch` is exactly like the `chained-if-expr`. Both these subclasses are there primarily for prettyprinting and related purposes.

`let-expr` ⊂ `application` [*class*]

When a LET expression is parsed, it is converted to an application. This is because `LET` $\mathtt{x}_1$ `=` e_1`, ...,` $\mathtt{x}_n$ `=` e_n `IN` e`(`$\mathtt{x}_1$`,...,`$\mathtt{x}_n$`)` is equivalent to `(LAMBDA (`$\mathtt{x}_1$`,...,`$\mathtt{x}_n$`):` e`(`$\mathtt{x}_1$`,...,`$\mathtt{x}_n$`))(`e_1`,...,`e_n`)`. Ocassionally it is useful to have methods on this class, the `let-reduce?` flag argument to many of the prover commands makes the distinction when beta-reducing terms. In addition, the typechecker treats `let-exprs` specially, because the variables on the left-hand side do not have to have a type declared for them, it will be determined from the right-hand side.
Remember that the LET expression with multiple bindings is treated as a sequential LET, i.e., `LET a = b, c = d IN e` is equivalent to `LET a = b IN LET c = d IN e`. In the former case, the transformation is done, and the inner `let-expr`(s) are converted to the `chained-let-expr` subclass.

`binding-expr` ⊂ `expr` [*class*]

bindings	a list of bind-decls
expression	an expr
commas?	whether the bindings have commas
chain?	whether the binding-expr was chained

This is a mixin for quantified-exprs and lambda-exprs. Note that binding expressions of the form `F ! (x: T): p(x)` are transformed by the parser to `binding-application` instances, which are actually a subclass of `application`. The `commas?` flag indicate the difference between `LAMBDA (x: int), (y: int): x + y` and `LAMBDA (x: int)(y: int): x + y`. The former is the uncurried form, and has a `bindings` list with two elements `commas?` set to `t`, whereas the latter curried form yields a nested lambda expression and `commas?` is set to `nil`. The `chain?` flag of the inner lambda expression(s) are set to `t` in this case.

`lambda-expr` ⊂ `binding-expr` [*class*]

This is the subclass of `binding-expr` used for LAMBDA expressions.

`quant-expr` ⊂ `binding-expr` [*class*]

This mixin is the subclass of `binding-expr` used for quantified expressions.

`forall-expr` ⊂ `quant-expr` [*class*]

This is the subclass of `quant-expr` used for FORALL expressions.

`exists-expr` ⊂ `quant-expr` [*class*]

This is the subclass of `quant-expr` used for EXISTS expressions.

`update-expr` ⊂ `expr` [*class*]

expression	an expr
assignments	a list of assignments

An update expression of the form `e WITH [x := 1, y := 2]`, maps to an `update-expr` instance, where the `expression` is `e`, and the `assignments` slot is set to the list of generated `assigment` instances. Note that these are very succinct representations, but correspondingly difficult to typecheck or to translate to other systems (e.g., decision procedures). See the description of the function ***translate-update-to-if!*** for more details.

Assignment Classes

`assignment` ⊂ `syntax` [*class*]

arguments.	the list of arguments
expression	the value expression

Assignments occur in both record-exprs and update-exprs. The `arguments` form is a list of lists. For example, given the assignment `` `a(x, y)`1 := 0 ``, the `arguments` are `((a) (x y) (1))` and the `expression` is `0`.

`maplet` ⊂ `assignment` [*class*]

A `maplet` is an assignment that may extend the update (it is illeagal in record expressions). When an assignment of the form `(x) |-> y` is parsed, it becomes a maplet. Most likely only the typechecker needs methods for this.

`field-assignment-arg` ⊂ `field-name-expr` [*class*]

A field assignment argument has three possible forms, e.g., `x := 3`, `(x) := 3`, and `` `x := 3 ``. The first two forms are ambiguous, and the typechecker must determine whether `x` is a field name of a record assignment, or an argument to a function. Initially `x` will be a `name-expr`. If it is determined to be a record assignment, then its class is changed to `field-assignment-arg`. Only useful for prettyprinting.

3.1.6 Names

TheoryNames ::= *TheoryName*$^{+}_{;}$

TheoryName ::= [*Id* `@`] *Id* [*Actuals*] [*Mappings*]

TheoryDeclName ::= [*Id* `@`] *Id* [*Actuals*] [*TheoryMaps*]

Names ::= *Name*$^{+}_{;}$

Name ::= [*Id* `@`] *IdOp* [*Actuals*] [*Mappings*] [`.` *IdOp*]

Actuals ::= `[` *Actual*$^{+}_{;}$ `]`

Actual ::= *Expr* | *TypeExpr*

Mappings ::= `{{` *Mapping*$^{+}_{;}$ `}}`

mapping ::= *Mapping*

MappingLhs MappingRhs MappingLhs ::= *IdOp Bindings** [`:` { `TYPE` | `THEORY` | *TypeExpr* }]

MappingRhs ::= `:=` { *Expr* | *TypeExpr* }

TheoryMaps ::= `{{` *TheoryMap*$^{+}_{;}$ `}}`

TheoryMap ::= *MappingLhs TheoryMapRhs*

TheoryMapRhs ::= *MapSubst* | *MapDef* | *MapRename*

MapSubst ::= `:=` { *Expr* | *TypeExpr* }

MapDef ::= `=` { *Expr* | *TypeExpr* }

MapRename ::= `::=` { *IdOp* | *Number* }

IdOps ::= *IdOp*$^{+}_{;}$

IdOp ::= *Id* | *Opsym* | *Number*

Opsym ::= *Binop* | *Unaryop*
| `IF` | `TRUE` | `FALSE` | `[||]` | `(||)` | `{||}`

Binop ::= `o` | `IFF` | `<=>` | `IMPLIES` | `=>` | `WHEN` | `OR` | `\/` | `AND`
| `/\` | `&` | `XOR` | `ANDTHEN` | `ORELSE` | `^` | `+` | `-` | `*` | `/`
| `++` | `~` | `**` | `//` | `^^` | `|-` | `|=` | `<|` | `|>` | `=`
| `/=` | `==` | `<` | `<=` | `>` | `>=` | `<<`
| `>>` | `<<=` | `>>=` | `#` | `@@` | `##`

Unaryop ::= `NOT` | `~` | `[]` | `<>` | `-`

FormulaName ::= `AXIOM` | `CHALLENGE` | `CLAIM` | `CONJECTURE` | `COROLLARY`
| `FACT` | `FORMULA` | `LAW` | `LEMMA` | `OBLIGATION`
| `POSTULATE` | `PROPOSITION` | `SUBLEMMA` | `THEOREM`

Ids ::= *Id*$^{+}_{,}$

Id ::= *Letter IdChar*$^{+}$

Number ::= *Digit*$^{+}$

String ::= `"` *ASCII-character** `"`

IdChar ::= *Letter* | *Digit* | `_` | `?`

Letter ::= `A` | ... | `Z` | `a` | ... | `z`

Digit ::= `0` | ... | `9`

3.1.7 Binding Declarations

`simple-decl` ⊂ `syntax` [*mixin class*]

id	a symbol
declared-type	a type expr
type	a type expr

`binding` ⊂ `simple-decl` [*mixin class*]

`dep-binding` ⊂ `binding` `name` [*class*]

parens	a nonnegative integer

`bind-decl` ⊂ `binding` `name-expr` [*class*]

chain?	a boolean (`t` or `nil`)

3.1.8 Theory Names

`modname` ⊂ `name` [*class*]

Actuals and Mappings

`actual` ⊂ `syntax` [*class*]

expr	a type expression or expression
type-value	a type expression or `nil`

`mapping` ⊂ `syntax` [*class*]
- ***lhs*** ..... an expression
- ***rhs*** ..... a `mapping-rhs` instance
- ***kind*** ..... a symbol
- ***declared-type*** a type expression
- ***type*** ..... a type expression

`mapping-def` ⊂ `mapping` [*class*]
- ***mapped-decl*** a declaration

`mapping-subst` ⊂ `mapping` [*class*]

`mapping-rename` ⊂ `mapping` [*class*]
- ***mapped-decl*** a declaration

`mapping-with-formals` ⊂ `mapping` [*class*]
- ***formals*** a list of bind-decls

`mapping-def-with-formals` ⊂ `mapping-with-formals` `mapping-def` [*class*]

`mapping-subst-with-formals` ⊂ `mapping-with-formals` `mapping-subst` [*class*]

`mapping-rename-with-formals` ⊂ `mapping-with-formals` `mapping-rename` [*class*]

`mapping-rhs` ⊂ `actual` [*class*]

3.1.9 Resolutions

`resolution` ⊂ ∅ [*class*]
- ***declaration*** .... a declaration
- ***module-instance*** a modname
- ***type*** ..... a type expression

3.1.10 The Context

The context is the set of declarations available while typechecking or proving. In simple type theory, this is usually denoted as Γ, and is simply a sequence of type, constant, and variable declarations. PVS contexts are more complex, and include the notion of a *current theory*, a *current declaration*, a symbol table

`context` $\subset \emptyset$		[*class*]
theory	a theory	
theory-name	a modname	
declaration	a declaration	
declarations-hash	a hash table	
using-hash	a hash table	
named-theories	a list of modnames	
library-alist	an association list	
judgements	a `judgements` instance	
known-subtypes	an association list	
conversions	a list of conversion declarations	
disabled-conversions	a list of conversion declarations	
auto-rewrites	a list of auto-rewrite-names	
disabled-auto-rewrites	a list of auto-rewrite-names	

`judgements` $\subset \emptyset$		[*class*]
judgement-types-hash	a hash table	
number-judgements-alist	an association list	
name-judgements-alist	an association list	
application-judgements-alist	an association list	

`name-judgements` $\subset \emptyset$		[*class*]
minimal-judgements	a list of judgements	
generic-judgements	a list of judgements	

`application-judgements` $\subset \emptyset$		[*class*]
generic-judgements	a list of judgements	
judgements-graph	a list of lists of justments	

`proof-info` $\subset \emptyset$	[*class*]
id	
description	
create-date	
run-date	
script	
status	
refers-to	
real-time	
run-time	
interactive?	
decision-procedure-used	

decl-reference $\subset \emptyset$	[*class*]
id	
class	
type	
theory-id	
library	

3.1.11 Prover Classes

s-formula $\subset \emptyset$	[*class*]
formula	
label	
new?	
asserted?	

sequent $\subset \emptyset$	[*class*]
s-forms	
p-sforms	
n-sforms	
hidden-s-forms	
info	

dpinfo $\subset \emptyset$	[*class*]
dpinfo-sigalist	
dpinfo-findalist	
dpinfo-usealist	

`proofstate` $\subset \emptyset$ [*class*]

label	The string representing the branch of the proofstate; a formula id followed by integers separated by periods, e.g., `"fml.1.2.1"`
current-goal	the sequent associated with this proofstate.
parent-proofstate	the parent of this proofstate.
current-subgoal	see Control section below.
pending-subgoals	see Control section below.
remaining-subgoals	see Control section below.
done-subgoals	see Control section below.
current-rule	
dp-state	
status-flag	
subgoalnum	
justification	In top-proofstate, set to justification of declaration. Set in proofstepper for propax. Set in success-step from current-rule, current-xrule, label, comment, and done-subgoals. Set in make-update when there is a justification field. Set in rule-apply for a successful goal. Set in undo-proof.
current-input	Set by query*, used to control display and find proofstates associated with undo forms
parsed-input	
printout	
comment	
strategy	
context	
proof-dependent-decls	
dependent-decls	
current-auto-rewrites	
tcc-hash	
subtype-hash	
rewrite-hash	
current-xrule	

This is a node of the proof tree with label `label` and sequent `current-goal`. It is a child of `parent-proofstate`, and has children `current-subgoal`, `pending-subgoals`, `remaining-subgoals`, and `done-subgoals`.

Control

The proof starts in the top proofstate, with a `current-goal` sequent consisting solely of the formula to be proved. The `parent-proofstate`, `current-subgoal`, `pending-subgoals`, `remaining-subgoals`, and `done-subgoals` are all initially `nil`. , a `label` string consisting of the formula identifier, a strategy which is either provided or `(query*-step)`.

tcc-sequent ⊂ sequent	[*class*]
tcc...	
expr..	
type..	
reason	
kind..	

tcc-proofstate ⊂ proofstate	[*class*]

rewrite ⊂ ∅	[*class*]
lhs	
rhs	
hyp	
res	

auto-rewrites-info ⊂ ∅	[*class*]
rewrites.....	
all-rewrites-names..	
auto-rewrites-names.	
auto-rewrites!-names	
macro-names.....	

top-proofstate ⊂ proofstate	[*class*]
in-justification	
declaration.....	

strat-proofstate ⊂ proofstate	[*class*]

strategy ⊂ ∅	[*class*]
topstep.....	
subgoal-strategy	
failure-strategy	

rulemacro ⊂ ∅	[*class*]
rule-list	

Rule and Strategy Entries

Note that rules and strategies may be introduced by the user as well as the system. The parts of the rules and strategies may be accessed through the global lisp variables `*rulebase*`, `*rules*`, or `*steps*`, each of which is a hashtable indexed by the name of the corresponding rule or strategy, and providing a rule-entry, defrule-entry, defstep-entry, or defhelper-entry.

Thus, for example, all format strings may be obtained using the following function

```
(defun collect-format-strings ()
  (with-open-file (*standard-output* "/dev/null" :direction :output
                                     :if-exists :overwrite)
    (read-strategies-files))
  (let ((format-strings nil))
    (maphash #'(lambda (n s)
                 (unless (string= (format-string s) "")
                   (push (format-string s) format-strings)))
             *rulebase*)
    (maphash #'(lambda (n s)
                 (unless (string= (format-string s) "")
                   (push (format-string s) format-strings)))
             *rules*)
    (maphash #'(lambda (n s)
                 (unless (string= (format-string s) "")
                   (push (format-string s) format-strings)))
             *steps*)
    format-strings))
```

`entry` ⊂ ∅ [*class*]

name
required-args
optional-args
docstring

`rule-entry` ⊂ `entry` [*class*]

rule-function
format-string

Primitive rule entries in the `*rulebase*` hash table.

`rule-instance` ⊂ ∅ [*class*]

rule
rule-input
rule-format

`defrule-entry` ⊂ ∅ [*class*]

name	the rule identifier
formals	the list of arguments
defn	the rule body
docstring	the helper documentation
format-string	the commentary string, a `format` control string

Derived rule entries in the `*rules*` hash table.

`defstep-entry` ⊂ `defrule-entry` [*class*]

Strategy entries in the `*steps*` hash table.

`defhelper-entry` ⊂ `defstep-entry` [*class*]

Helper strategy entries in the `*steps*` hash table.

`strategy-instance` ⊂ ∅ [*class*]
- *strategy-fun*
- *strategy-input*

`rulefun-entry` ⊂ `entry` [*class*]
- *rulefun*

`strategy-entry` ⊂ `entry` [*class*]
- *strategy-fun*

`justification` ⊂ ∅ [*class*]
- *label*
- *rule*
- *subgoals*
- *xrule*
- *comment*

`skolem-const-decl` ⊂ `const-decl` [*class*]

3.1.12 Ground Evaluator Classes

`eval-defn-info` ⊂ ∅ [*class*]
- *unary*
- *multiary*
- *destructive*

`eval-defn` ⊂ ∅ [*class*]
- *name*
- *definition*
- *output-vars*

`eval-info` ⊂ ∅ [*class*]
- *internal*
- *external*

`destructive-eval-defn` ⊂ `eval-defn` [*class*]
side-effects

3.2 Defining Methods

3.2.1 A Template for Defining Methods

3.3 Global Variables

Global variables are used extensively in PVS. In some cases they simply are a way to optimize, keeping functions from having large numbers of arguments, and providing hash tables. One of the advantages of CLOS is that it is very easy to define a set of methods that produce a result, then at a later date add an around method that uses a global hash table to memoize the function.

`*pvs-library-path*` [*global*]

Set from the `PVS_LIBRARY_PATH` environment variable + *pvs-path*

`*pvs-emacs-interface*` [*global*]

Set to `t` by Emacs in pvs-load - affects how pvs-emacs functions work

`*pvs-directories*` [*global*]

This is set to a list of directories to search for using the `lf` function.

`*pvs-version*` [*global*]

The version number, currently the string `"3.2"`.

`*binfile-version*` [*global*]

`*pvs-context*` [*global*]

`*pvs-context-path*` [*global*]

The current context path - this can change when loading libraries.

`*pvs-current-context-path*` [*global*]

The current context path - this one only changes with change-context

`*pvs-files*` [*global*]

`*pvs-verbose*` [*global*]

Flag indicating level of messages to print when noninteractive

`*suppress-msg*` [*global*]

Flag indicating whether to suppress messages output with pvs-message

`*show-conversions*` [*global*]

Flag indicating whether conversions are to be displayed when unparsing.

`*untypecheck-hook*` [*global*]

Functions (with no args) to be called whenever untypecheck is called

`*prelude-context*` [*global*]

Provides the context associated with the prelude

`*prelude*` [*global*]

The hash-table of prelude

`*prelude-theories*` [*global*]

A list of the prelude theories; more useful than *prelude* when the order is important

`*prelude-library-context*` [*global*]

Provides the context associated with the current prelude libraries

`*prelude-libraries*` [*global*]

The pathnames of the prelude libraries that have been loaded. Given a pathname, returns a hash-table which can then be put into *visible-libraries*

`*imported-libraries*` [*global*]

`*pvs-modules*` [*global*]

The hash-table of modules known to the system in this session

`*in-evaluator*`	[*global*]
`*noninteractive*`	[*global*]
`*disable-gc-printout*`	[*global*]
`*boolean*`	[*global*]
`*true*`	[*global*]
`*false*`	[*global*]
`*number*`	[*global*]
`*number_field*`	[*global*]
`*real*`	[*global*]
`*rational*`	[*global*]
`*integer*`	[*global*]
`*naturalnumber*`	[*global*]
`*posint*`	[*global*]
`*even_int*`	[*global*]

`*odd_int*` [*global*]

`*ordinal*` [*global*]

`*tcdebug*` [*global*]

`*generate-tccs*` [*global*]

`*tccs*` [*global*]

`*collecting-tccs*` [*global*]

Controls whether TCCs are inserted in the current theory or simply collected in *tccforms*. Note that if *in-checker* is `t`, TCCs will be added to *tccforms* regardless.
Used by the typechecker in order to generate proper TCCs for datatype updates, which may be disjunctions.

`*tccforms*` [*global*]

The TCCs generated while typechecking an expression when *in-checker*, *in-evaluator*, or *collecting-tccs* is `t`. Otherwise the TCCs are added directly to the theory.

`*bound-variables*` [*global*]

`*valid-id-check*` [*global*]

`*subtype-of-hash*` [*global*]

`*keep-unbound*` [*global*]

`*last-proof*` [*global*]

pvs-operators [*global*]

default-char-width [*global*]

3.3.1 Prover Globals

in-checker [*global*]

rulebase [*global*]

The hash table of primitive rules, i.e., rules defined using ***addrule***.

rules [*global*]

The hash table of derived rules, i.e., rules defined using ***defrule***, ***defstep***, or ***defhelper***.

steps [*global*]

The hash table of strategies defined using ***defstep***, or ***defhelper***.

top-proofstate [*global*]

new-fmla-nums [*global*]

current-decision-procedure [*global*]

default-decision-procedure [*global*]

decision-procedures [*global*]

decision-procedure-descriptions [*global*]

`*subgoals*` [*global*]

`*multiple-proof-default-behavior*` [*global*]

`*dump-sequents-to-file*` [*global*]

`*show-parens-in-proof*` [*global*]

`*pvs-context-writable*` [*global*]

`*save-proofs-pretty*` [*global*]

`*number-of-proof-backups*` [*global*]

`*debugging-print-object*` [*global*]

`*current-theory*` [*global*]

is the theory instance that the formula being proved belongs to.

`*current-context*` [*global*]

is the context of the current formula

`*top-proofstate*` [*global*]

`*ps*` [*global*]

`*number-of-proof-backups*` [*global*]

save-proofs-pretty	[*global*]

3.4 Functions

There are several way to create types and terms in PVS. The most direct way is to use the Common Lisp `make-instance` function, but it is not the most convenient, as it is difficult to know how to set the slots in a consistent manner. For example, the *type* and *declared-type* slots, should always be such that `(pc-typecheck (declared-type` *instance*`))` is `tc-eq` to `(type` *instance*`)`. Still, it is often convenient to use `make-instance`.

There are three usual alternatives for creating instances. The safest, albeit slowest, way is to build strings and to parse and typecheck them. Creating terms that are just a minor difference from an existing term can be done by copying them. Finally, terms can be created bottom up using the *make* functions.

3.4.1 Parsing and Typechecking

Parsing and typechecking a PVS file can be done using *parse-file* and `typecheck-file`, as described in Sections 3.4.1 and 3.4.2.

Parsing

Parsing a term can is usually done using `pc-parse`. The parser is built using the Ergo parser generator [2], similar to YACC or bison, but for Lisp. Parsing in PVS means invoking the Ergo generated parser function `pvs-parse`, and converting the Ergo term abstract syntax into PVS terms using functions with names of the form `xt-`*nonterminal*. The `pc-parse` function calls *parse*, but with a simplified interface that makes it easier to use.

parse-importchain (theories)

pc-parse (*input nonterminal*) [*function*]

If the input is of type `syntax`, it is simply returned (without checking that it satisfies the *nonterminal*). Otherwise if *input* is not a string, `pc-parse` is called on the string generated by `(format nil "~a" input)`. Finally, if it is a string, then it is parsed and the resulting PVS term is returned. The `nonterminal` is one of the nonterminals defined in the grammar input for the Ergo parser generator. The most useful nonterminals are:

`expr` - Expressions

`name-expr` - Name expressions

`type-expr` - Type expressions

`modname` - Theory names

`name` - Name

If the string is not parsable as a *nonterminal*, `parse-error` is called. See Section 3.6.
The `pc-parse` function can be called from anywhere, regardless of the value of `*current-context*`.

parse (&REST *keys*) [*function*]

pvs-parse (&KEY (*nt* 'adt-or-theories) *error-threshold* *ask-about-bad-tokens* *return-errors* *stream* *string* *file* *exhaust-stream*) [*function*]

3.4.2 Typecheck

pc-typecheck (*ex* &KEY *expected* (*fnums* '*) (*uniquely?* t)) [*function*]

takes an expression or type-expression and typechecks it. The result depends on whether *ex* is an expression or type expression.
Fo a type expression, `pc-typecheck` either returns an error message or the canonical form of the type. The canonical form is essentially the type with all defined types expanded. Thus `int` becomes {`x: {x: {x: number | real_pred(x)} | rational_pred(x)} | integer_pred(x)`}.
For an expression, `pc-typecheck` either returns an error message or decorates the given expression with type information. If none of the keyword arguments are given, then if the expression has a unique type, or can be found in the current sequent, then that determines the type given to the expression, otherwise an error is reported. If the *expected* type is given, then the typechecker will use that as the type.
If *expected* is `NIL`, *uniquely?* is `NIL`, and the expression cannot be found in the current sequent, then if there is no type error the expression is not yet fully typechecked. In this case, the `types` slot is set to reflect the possible types of the expression, and the `type` slot is not yet set. This is useful only in special cases, for example it may be desirable to partially typecheck the expression and then look through the possible types for record types, without knowing beforehand which recordtype is the expected type. Note that such partially typechecked expressions should not be passed in to other functions without first fully typechecking them, which may be accomplished by invoking `pc-typecheck` with an *expected* type.
There are a number of types that are used frequently enough that global variables have been assigned their values: `*boolean*`, `*naturalnumber*`, `*integer*`, `*rational*`, `*real*`, and `*number*`. These may be used as the *expected* types.

Here is an example of the use of pc-parse and pc-typecheck:

```
(pc-typecheck (pc-parse "LAMBDA (x:int): x + 2" 'expr)
              :expected (pc-typecheck (pc-parse "[int -> int]" 'type-expr)))
```

resolve (*name kind args*) [*function*]

returns a list of the possible resolutions for the *name*. A resolution consists of a declaration and a theory instance, and (for name expressions) a type. The *kind* is a symbol representing the kind of name to expect: `expr`, `type`, `module`, or `formula`. The *args* are the arguments provided if it is an operator; this can help to filter the possible resolutions and provide the theory instance. For example, `car` normally has only one declaration in the `list` datatype, but without more information the instance cannot be determined. If `l` is a name expression of type `list[int]`, then `(resolve '|cons| 'expr (list l))` will return a fully instantiated resolution.

typecheck (*obj* &KEY *expected context tccs*) [*function*]

*typecheck** (*obj expected kind arguments*) [*function*]

resolve (*name kind args* &OPTIONAL (*context* `*current-context*`)) [*function*]

formula-or-definition-resolutions (*name*) [*function*]

definition-resolutions (*name*) [*function*]

formula-resolutions (*name*) [*function*]

resolve-theory-name (*modname*) [*function*]

resolve-theory-name (*modname*) [*function*]

make-new-variable-name-expr (*id type*) [*function*]

make-new-variable (*base expr* &OPTIONAL *num*) [*function*]

3.4.3 Constructing Types and Expressions

Constructing Names and Resolutions

The most common names to construct are `name-expr`s, `type-name`s, and `modname`s. To construct a typechecked `name-expr` or `type-name`, a `resolution` must be provided.

def-pvs-term (*name term theory* &KEY *(nt 'expr) expected*) [*macro*]

This macro is used to create a function that returns a specific term from a specific theory instance. This is an optimization, for example

```
(def-pvs-term not-operator "NOT" "booleans")
```

creates the function *not-operator*, which returns a `name-expr` that resolves to the the `NOT` declared in the `booleans` theory of the prelude. This may then be used to check if a given term is *tc-eq* to it, or used to build up a negation.
The *name* is used to create a function. The first time this function is called, it builds the given *term* and typechecks it. For each call after that the computed term is returned.
If the term is not an expression, the *nt* may be used to specify a diffferent nonterminal, e.g., `type-expr`.
The *expected* not often needed, but it is useful when the term may otherwise be ambiguous, for example, the minus and difference operators.

```
(def-pvs-term difference-operator "-" "number_fields" :expected "[number_field,
(def-pvs-term minus-operator "-" "number_fields" :expected "[number_field -> nur
```

Note that the given theory instance should not be from a theory that has theory parameters

mk-formula-decl (*id expr* &OPTIONAL *(spelling 'formula) kind*) [*function*]

mk-name-expr (*id* &OPTIONAL *actuals mod resolution*) [*function*]

Creates a `name-expr` instance

make-bind-decl () [*function*]

make-variable-expr () [*function*]

mk-resolution (*decl modinst type*) [*function*]

make-resolution (*decl modinst* &OPTIONAL *type*) [*function*]

Constructing Types

mk-type-name (*id* &OPTIONAL *actuals mod-id resolution*) [*function*]

mk-dep-binding (*id* &OPTIONAL *type dtype*) [*function*]

mk-subtype (*supertype predicate*) [*function*]

mk-setsubtype (*supertype predicate*) [*function*]

mk-expr-as-type (*expr*) [*function*]

mk-funtype (*domain range* &OPTIONAL (*class* 'funtype)) [*function*]

mk-predtype (*type*) [*function*]

mk-tupletype (*types*) [*function*]

mk-cotupletype (*types*) [*function*]

mk-recordtype (*field-decls dependent?*)	[*function*]
mk-field-decl (*id dtype* &OPTIONAL *type*)	[*function*]
make-recordtype (*fields*)	[*function*]
make-tupletype (*types*)	[*function*]
make-cotupletype (*types*)	[*function*]
make-domain-type-from-bindings (*vars*)	[*function*]
make-tupletype-from-bindings (*vars* &OPTIONAL *result*)	[*function*]
make-funtype (*domain range*)	[*function*]
make-predtype (*type*)	[*function*]
make-declared-type (*te*)	[*function*]

Constructing Expressions

Aside from parsing and *make-instance*, there are three basic ways to construct expressions. These are characterized by the function prefix, as follows.

mk- These functions generally just invoke *make-instance*, but are easier to use. The result is generally not typechecked.

make- These functions generally include an expected type argument, and typecheck the resulting term.

make!- These functions require the subterms to be typechecked, and create a new typechecked term.

Deciding which of these to use depends on what is needed. The ***make!-*** forms are the fastest, but TCCs will not be generated, so it is best to use these only in circumstances in which any potential TCCs have already been dealt with. For example, it is safe to take the two arguments of a sum, and create a product using ***make!-times***. But unless it is already known that the second argument is nonzero, or that TCC checks will be made later, it is not safe to use ***make!-divides***.

One technique often used is to build up the subterms using ***mk-*** functions, then use a ***make-*** function at the top level to get the typechecked form. The same thing can be accomplished using ***mk-*** forms throughout, then calling ***typecheck*** on the result.

For the most part, a function of the form ***mk-foo*** (or ***make-foo*** or ***make!-foo***) creates an instance of (a subclass of) ***foo***.

mk- Functions

mk-number-expr (***num***) [*function*]

Simply creates a `number-expr` from ***num***, which must be a nonnegative integer.

mk-record-expr (***assignments***) [*function*]

Creates a `record-expr` instance from the list of ***assignments***. See ***mk-assignment*** for creating assignment instances.

mk-tuple-expr (***exprs***) [*function*]

Creates a `tuple-expr` instance from ***exprs***, which is a list of expressions.

mk-cases-expr (***expr selections else***) [*function*]

Creates a `cases-expr` instance of the form

```
CASES expr OF
  selections
  ELSE else
ENDCASES
```

mk-selection (***name-expr args expr***) [*function*]

mk-application* (***op arguments***) [*function*]

mk-application (*op* &REST *args*) [*function*]

mk-if-expr (*cond then else*) [*function*]

mk-chained-if-expr (*cond then else*) [*function*]

*mk-if-expr** (*class cond then else*) [*function*]

mk-implication (*ante succ*) [*function*]

mk-iff (*ante succ*) [*function*]

mk-conjunction (*args*) [*function*]

mk-disjunction (*args*) [*function*]

mk-negation (*arg*) [*function*]

mk-rec-application (*op base args*) [*function*]

mk-rec-application-left (*op base args*) [*function*]

mk-lambda-expr (*vars expr*) [*function*]

mk-let-expr (*bindings expr*) [*function*]

mk-coercion (*expr type*) [*function*]

mk-forall-expr (*vars expr*) [*function*]

mk-exists-expr (*vars expr*) [*function*]

mk-equation (*lhs rhs*) [*function*]

mk-update-expr (*expr assignments*) [*function*]

mk-update-expr-1 (*expr index value*) [*function*]

mk-greatereq (*a1 a2*) [*function*]

mk-greater (*a1 a2*) [*function*]

mk-lesseq (*a1 a2*) [*function*]

mk-less (*a1 a2*) [*function*]

mk-floor (*a1*) [*function*]

mk-null-expr () [*function*]

mk-list-expr (*exprs*) [*function*]

*mk-list-expr** (*exprs result*) [*function*]

mk-bindings (*vars*) [*function*]

*mk-bindings** (*vars* &OPTIONAL *result*) [*function*]

mk-chained-bindings (*bindings*) [*function*]

mk-bind-decl (*id dtype* &OPTIONAL *type*) [*function*]

mk-arg-bind-decl (*id dtype* &OPTIONAL *type*) [*function*]

mk-assignment (*flag arguments expression*) [*function*]

mk-maplet (*flag arguments expression*) [*function*]

mk-actual (*arg*) [*function*]

mk-mapping (*lhs rhs*) [*function*]

mk-mapping-rhs (*ex*) [*function*]

mk-field-name-expr (*id res*) [*function*]

*mk-arg-tuple-expr** (*args*) [*function*]

mk-arg-tuple-expr (&REST *args*) [*function*]

mk-sum (*a1 a2*) [*function*]

mk-difference (*a1 a2*) [*function*]

mk-product (*a1 a2*) [*function*]

mk-division (*a1 a2*) [*function*]

mk-implies-operator () [*function*]

make- Functions

make-tuple-expr (*exprs* &OPTIONAL *expected*) [*function*]

make-record-expr (*assignments expected*) [*function*]

make-cases-expr (*expr selections else*) [*function*]

make-arg-tuple-expr (*args*) [*function*]

*make-application** (*op arguments*) [*function*]

make-application (*op* &REST *arguments*) [*function*]

make-projection-application (*index arg*) [*function*]

make-field-application (*field-name arg*) [*function*]

make-projections (*expr* &OPTIONAL *type*) [*function*]

projection-application-type (*projapp type*) [*function*]

field-application-types (*types expr*) [*function*]

field-application-type (*field type arg*) [*function*]

make-if-expr (*cond then else*) [*function*]

make-chained-if-expr (*cond then else*) [*function*]

make-equation (*lhs rhs*) [*function*]

make-implication (*ante succ*) [*function*]

make-iff (*ante succ*) [*function*]

make-conjunction (*args*) [*function*]

make-disjunction (*args*) [*function*]

make-negation (*arg*) [*function*]

make-lambda-expr (*vars expr*) [*function*]

make-forall-expr (*vars expr*) [*function*]

make-exists-expr (*vars expr*) [*function*]

make-null-expr (*type*) [*function*]

make-list-expr (*exprs* &OPTIONAL *type*) [*function*]

make-number-expr (*number*) [*function*]

make-difference (*a1 a2 type*) [*function*]

make-assignment (*arg expression*) [*function*]

make-update-expr (*expression assignments* &OPTIONAL *expected*) [*function*]

make-greatereq (*x y*) [*function*]

make-greater (*x y*) [*function*]

make-lesseq (*x y*) [*function*]

make-less (*x y*)	[*function*]
make-floor (*x*)	[*function*]
make-implication ()	[*function*]
make-conjunction ()	[*function*]
make-disjunction ()	[*function*]
make-lambda-expr ()	[*function*]
make-equality ()	[*function*]
negate ()	[*function*]
universal-closure (*form*)	[*function*]
existential-closure (*form*)	[*function*]
mk-everywhere-true-function (*type*)	[*function*]
mk-everywhere-false-function (*type*)	[*function*]
mk-identity-fun (*te*)	[*function*]

make!- Functions

make!-applications (*make"!-applications*)	[*function*]
*make!-application** (*make"!-application**)	[*function*]
make!-application (*make"!-application*)	[*function*]
make!-reduced-application (*make"!-reduced-application*)	[*function*]
make!-number-expr (*make"!-number-expr*)	[*function*]
make!-name-expr (*make"!-name-expr*)	[*function*]
make!-equation (*make"!-equation*)	[*function*]
make!-disequation (*make"!-disequation*)	[*function*]
make!-if-expr (*make"!-if-expr*)	[*function*]
make!-chained-if-expr (*make"!-chained-if-expr*)	[*function*]
*make!-if-expr** (*make"!-if-expr**)	[*function*]
make!-arg-tuple-expr (*make"!-arg-tuple-expr*)	[*function*]
*make!-arg-tuple-expr** (*make"!-arg-tuple-expr**)	[*function*]
make!-projected-arg-tuple-expr (*make"!-projected-arg-tuple-expr*)	[*function*]
*make!-projected-arg-tuple-expr** (*make"!-projected-arg-tuple-expr**)	[*function*]
make!-tuple-expr (*make"!-tuple-expr*)	[*function*]
*make!-tuple-expr** (*make"!-tuple-expr**)	[*function*]
make!-projections (*make"!-projections*)	[*function*]
*make!-projections** (*make"!-projections**)	[*function*]
make!-projection-application (*make"!-projection-application*)	[*function*]
*make!-projection-type** (*make"!-projection-type**)	[*function*]
make!-injection-application (*make"!-injection-application*)	[*function*]
make!-injection?-application (*make"!-injection?-application*)	[*function*]
make!-extraction-application (*make"!-extraction-application*)	[*function*]
make!-field-application (*make"!-field-application*)	[*function*]
make!-field-application-type (*make"!-field-application-type*)	[*function*]

*make!-field-application-type** (*make"!-field-application-type**)	[*function*]
make!-update-expr (*make"!-update-expr*)	[*function*]
make!-negation (*make"!-negation*)	[*function*]
make!-conjunction (*make"!-conjunction*)	[*function*]
*make!-conjunction** (*make"!-conjunction**)	[*function*]
*make!-conjunction*** (*make"!-conjunction***)	[*function*]
make!-disjunction (*make"!-disjunction*)	[*function*]
*make!-disjunction** (*make"!-disjunction**)	[*function*]
*make!-disjunction*** (*make"!-disjunction***)	[*function*]
make!-implication (*make"!-implication*)	[*function*]
make!-iff (*make"!-iff*)	[*function*]
make!-plus (*make"!-plus*)	[*function*]
make!-difference (*make"!-difference*)	[*function*]
make!-minus (*make"!-minus*)	[*function*]
make!-times (*make"!-times*)	[*function*]
make!-divides (*make"!-divides*)	[*function*]
make!-forall-expr (*make"!-forall-expr*)	[*function*]
make!-exists-expr (*make"!-exists-expr*)	[*function*]
make!-lambda-expr (*make"!-lambda-expr*)	[*function*]
make!-set-expr (*make"!-set-expr*)	[*function*]
make!-bind-decl (*make"!-bind-decl*)	[*function*]
make!-floor (*make"!-floor*)	[*function*]
make!-succ (*make"!-succ*)	[*function*]
make!-pred (*make"!-pred*)	[*function*]
make!-expr-as-type (*make"!-expr-as-type*)	[*function*]
make!-unpack-expr (*make"!-unpack-expr*)	[*function*]
make!-unary-minus (*make"!-unary-minus*)	[*function*]

make!-less (*make"!-less*) [*function*]

make!-lesseq (*make"!-lesseq*) [*function*]

3.4.4 Comparison Functions

tc-eq (*x y*) [*function*]

tc-eq-with-bindings (*x y bindings*) [*function*]

strong-tc-eq (*x y*) [*function*]

compatible? (*atype etype*) [*function*]

strict-compatible? (*(atype etype*) [*function*]

compatible-type (*t1 t2*) [*function*]

compatible-types (*list-of-types*) [*function*]

compatible-preds (*atype etype aexpr*) [*function*]

subtype-of? (*(t1 t2*) [*function*]

tc-match (*t1 t2 bindings* &OPTIONAL *strict-matches*) [*function*]

simple-match (*ex inst*) [*function*]

match (*expr instance bind-alist subst*) [*function*]

3.4.5 Substitution Functions

copy () [*function*]

lcopy (*obj* &REST *initargs*) [*function*]

substit () [*function*]

gensubst (***obj substfn testfn***) [*function*]

gensubst walks down the term ***obj***, applying the ***testfn*** function to each subterm recursively. If it returns a non-nil value, then the ***substfn*** function is applied. If it results in a term that is not `eq` to the original term, it causes a creation of copies of the branch leading to the substitution term. Note that there is no check for type correctness; the substitution is made blindly. If you are simply substituting terms for variables, use substit instead, as this is faster and guaranteed to be type correct if the bindings are. An example using `gensubst` is `expose-binding-types`, which is applied to TCCs so that the types are made visible.

```
(defun expose-binding-types (expr)
  (let ((*visible-only* t))
    (gensubst expr #'expose-binding-types! #'expose-binding-types?)))

(defmethod expose-binding-types? (ex)
  (declare (ignore ex))
  nil)

(defmethod expose-binding-types? ((ex type-application))
  t)

(defmethod expose-binding-types? ((ex untyped-bind-decl))
  t)

(defmethod expose-binding-types! ((ex type-application))
  ex)

(defmethod expose-binding-types! ((ex untyped-bind-decl))
  (let ((dtype (or (and (type ex) (print-type (type ex)))
   (declared-type ex)
   (type ex))))
    (if dtype
(change-class (copy ex 'declared-type dtype) 'bind-decl)
ex)))
```

There are two global variables that affect ***gensubst***. `*parsing-or-unparsing*` says whether the term has been typechecked, and `*visible-only*` will only go down the visible subterms; in particular, it will only affect declared types (reflected in the `print-type` slot), not the expanded canonical form of the type.

subst-theory-params (*term alist*) [*function*]

subst-mod-params (*obj modinst* &OPTIONAL *theory*) [*function*]

3.4.6 Prover Functions

Defining Rules and Strategies

The following macros allow rules and strategies to be defined. Every rule and strategy has associated with it a list of arguments, a body, a documentation string, and a format string. The ***addrule*** macro has separate lists for the required and optional arguments, for the other macros they are separated by &OPTIONAL , as in Lisp. The docstring simply describes the rule or strategy, and is used by the ***help*** rule. The format string is used to provide commentary, and has the form of a Lisp ***format*** string, with arguments corresponding to the arguments of the rule or strategy. A simple example is the format string for ***simple-induct***:

```
(defstep simple-induct (var fmla &optional name)
  ...
  "Inducting on ~ a with formula ~ a")
```

Here the first `~a` is matched to `var`, and the second to `fmla`. A more complicated example is ***install-rewrites***:

```
(defstep install-rewrites (&optional defs theories rewrites
                                     exclude-theories exclude)
  ...
  "Installing rewrite rules from~
~@[~% definitions (~ a) in the sequent~]~
~@[,~% theories: ~ a~]~
~@[,~% rewrites: ~ a~]~
~@[,~%and excluding theories: ~ a~]~
~@[,~%and excluding rewrites: ~ a~]")
```

In this case, the `~@[ ... ~]` control forms are displayed only if the corresponding argument is not `nil`, which is appropriate for optional arguments. See the Common Lisp Reference for full details on `format` control strings.

Strategies and Defined Rules

addrule (***name required-args optional-args body docstring*** [*ma*
&OPTIONAL ***format-string***)

This macro is used to add primitive rules to PVS. ***name*** is an iden
required-args is a list of identifiers, and ***optional-args*** is a list of id
fiers, optionally with one `&rest` keyword anywhere before the last identifier.
is a Lisp expression, that should refer to the identifiers in ***required-args***
optional-args. ***docstring*** is a string that describes the rule (this is disp
by the ***help*** command), and ***format-string*** is used to generate the ru
commentary. ***format-string*** is a format control string, and is applied t
argument list obtained by concatenating ***required-args*** and ***optional-a***
Here is an example.

```
(addrule 'foo (req) ((opt 3) &REST rest)
  (foo-fun req opt rest)
  "Fooify the sequent"
  "Fooifying ~a as ~d using ~{~a, ~}")
```

Primitive Rules

The `foo` rule has the required argument `req`, an optional argument `opt`, w
if not provided, defaults to the value `3`, and a argument `rest` that is a list
remaining arguments. The body simply invokes a Lisp function named `foo`
passing along the arguments. This is typical, and makes debugging easier.
rule might be invoked as

```
(foo ẍ + 1¨ 6 a b c)
```

in which case `foo-fun` will be called with

```
(foo ẍ + 1¨ 6 (a b c))
```

addrule creates a `rule-entry`, adds it to the `*rulebase*`, and adds the ***na***
`*rulenames*`. It then creates an entry and adds it to `*prover-keywords*`
entry is a list consisting of the ***name***, a flag (`t` if there is an `&rest` argum
and a list of keywords created from the arguments. In the case above, the
is

```
(foo t :req :opt :rest)
```

Prover Errors and Messages Various errors and messages are possible in executing commands. It is important to get these right, otherwise the user of the command will not understand what happened. Note that it is not enough to get it right at the top level, one must always take into account that the command may be invoked from deep within another strategy, so the printout should either be suppressed or continue to make sense in this context.

The main printout is from the *format-string* which is a format string that is applied to the actual arguments of the command. There are several other ways to write out information, and these are often necessary when it is desired to display intermediate results, for example, the instantiations actually found in an `inst?` invocation.

One way of producing printout is with the `skip-msg` rule. This takes a string as an argument, prints out the string, and acts as a `skip` otherwise. By default, if the command is not at the top level, the message is not printed, but the optional argument *force-printing?* causes it to be printed no matter where it occurs. Note that the message must be a string, not a form that evaluates to a string. A common pattern of use is

```
(defstep foo ()
  ...
  (let ((msg (format nil "The argument is: ~a" argument)))
    (skip-msg msg))
  ...
)
```

It's important to remember that

defrule (*name args body doc format*)	[*macro*]
defstrat (*name args body doc* &OPTIONAL *format*)	[*macro*]
defstep (*name args body doc format*)	[*macro*]
defhelper (*name args body doc format*)	[*macro*]
format-if (*string* &REST *args*)	[*macro*]
error-format-if (*string* &REST *args*)	[*macro*]
restore ()	[*macro*]

prove (*name* &KEY *strategy*) [*function*]

simplify-expression (*expr module-name strategy* &OPTIONAL *display?* (*id* 'simplify-expr)) [*function*]

simplify-expr (*expr module-name strategy* &OPTIONAL *display?* (*id* 'simplify-expr)) [*function*]

prove-decl (*decl* &KEY *strategy context*) [*function*]

check-command-arguments (*cmd keywords arguments has-rest?* &OPTIONAL *expect-key?*) [*function*]

select-seq (*seq nums*) [*function*]

delete-seq (*seq nums*) [*function*]

gather-seq (*seq yesnums nonums* &OPTIONAL (*pred* #'always-true)) [*function*]

gather-fnums (*sforms yesnums nonums* &OPTIONAL (*pred* #'always-true)) [*function*]

find-all-sformnums (*sforms sformnums pred*) [*function*]

find-sform (*sforms sformnum* &OPTIONAL (*pred* #'always-true)) [*function*]

pos-s-forms (*s-forms*) [*function*]

neg-s-forms (*s-forms*) [*function*]

seq-formula (*sform*) [*function*]

count-proofstates (*ps* &OPTIONAL (*num* 0)) [*function*]

select-seq () [*function*]

s-forms () [*function*]

current-goal () [*function*]

formula () [*function*]

new-sko-symbol () [*function*]

find-all-sformnums () [*function*]

delete-seq () [*function*]

find-quant-terms () [*function*]

substitutable-vars () [*function*]

match () [*function*]

create-formulas () [*function*]

collect-subterms () [*function*]

3.4.7 Predicates

recognizer? () [*function*]

datatype-subtype? () [*function*]

not-expr? () [*function*]

typed? (*expr*) [*function*]

fully-typed? (*obj*) [*function*]

fully-instantiated? (*obj*) [*function*]

ground-arithmetic-term? (*expr*) [*function*]

constant? (*expr*) [*function*]

variable? (*expr*) [*function*]

from-prelude? (*obj*) [*function*]

quant-occurs? (*expr*) [*function*]

connective-occurs? (*expr*) [*function*]

update-or-connective-occurs? (*expr*) [*function*]

record-redex? (*expr*) [*function*]

function-update-redex? (*expr*) [*function*]

accessor-update-redex? (*expr*) [*function*]

integer-expr? (*(ex application)*) [*function*]

record-update-redex? (*expr*) [*function*]

is-predicate? (*expr*) [*function*]

is-plus? (*op*) [*function*]

is-minus? (*op*) [*function*]

is-sub-minus? (*op*) [*function*]

is-mult? (*op*) [*function*]

is-div? (*op*) [*function*]

is-division? (*expr*) [*function*]

is-addition? (*expr*) [*function*]

is-subtraction? (*expr*) [*function*]

is-unary-minus? (*expr*) [*function*]

is-multiplication? (*expr*) [*function*]

negative-number? (*expr*) [*function*]

arithop-decl? (*x*) [*function*]

occurs-in (*x y*) [*function*]

id-occurs-in (*id y*) [*function*]

proved? ((*fdecl*) [*function*]

unproved? ((*formula-decl*) [*function*]

parsed-file? ((*filename*) [*function*]

parsed? (*mod*) [*function*]

possibly-empty-type? (*(te*) [*function*]

special-variable-p (*obj*) [*function*]

3.4.8 Utility Functions

get-theory () [*function*]

freevars () [*function*]

find-supertype () [*function*]

mapobject () [*function*]

file-older (*file1 file2*) [*function*]

add-decl (*decl* &OPTIONAL (*insert?* t) (*generated?* t) *assuming?*) [*function*]

compare (*old new*) [*function*]

get-pvs-file-dependencies (*filename*) [*function*]

get-theory-dependencies (*theoryid*) [*function*]

collect-theories () [*function*]

find-all-usedbys (*theoryref*) [*function*]

adt-generated-theories (*adt*) [*function*]

collect-conjuncts ((*ex conjunction*)) [*function*]

collect-disjuncts ((*ex disjunction*)) [*function*]

type-canon (*te*) [*function*]

free-params (*obj*) [*function*]

mapobject (*fn obj*) [*function*]

judgement-types+ (*expr*) [*function*]

judgement-types (*ex*) [*function*]

type-constraints (*ex* &OPTIONAL *all?*) [*function*]

show-expanded-form (*oname origin pos1* &OPTIONAL (*pos2 pos1*) *all?*) [*function*]

makesym (*ctl* &REST *args*) [*macro*]

with-no-type-errors (&REST *forms*) [*macro*]

with-no-parse-errors (&REST *forms*) [*macro*]

valid-pvs-id (*symbol*) [*function*]

unparse (*obj* &KEY *string stream file char-width length level lines* (*pretty* t)) [*function*]

unpindent (*inst indent* &KEY (*width* *default-char-width*) *length level lines string comment?*) [*function*]

clear-theories (&OPTIONAL *all?*) [*function*]

get-pvs-version-information () [*function*]

all-importings (*theory* &OPTIONAL *lib*) [*function*]

prove-unproved-tccs (*theories* &OPTIONAL *importchain?*) [*function*]

quit (&OPTIONAL *status*) [*function*]

get-parsed-theory (*theoryref*) [*function*]

get-typechecked-theory (*theoryref* &OPTIONAL *theories*) [*function*]

get-decl-at (*line class theories*) [*function*]

get-decls (*ref*) [*function*]

locality (*ex*) [*function*]

set-type (*ex expected*) [*function*]

expand1 (*ex*) [*function*]

lambda-binding-number (*ex*) [*function*]

get-arithmetic-value (*expr*) [*function*]

cartesian-product (*list-of-lists* &OPTIONAL (*result* (list nil))) [*function*]

load-prelude () [*function*]

merge-proofs-into-updated-prelude (*file*) [*function*]

save-prelude-proofs () [*function*]

prove-prelude (&OPTIONAL *retry?* *use-default-dp?*) [*function*]

trace-methods (*funsym*) [*function*]

;; Discuss metering

types (*ex*) [*function*]

typecheck-uniquely (*expr* &KEY (*tccs* 'all given)) [*function*]

get-immediate-usings (*theory*) [*function*]

lf (*file* &OPTIONAL *force*) [*function*]

show (*obj*) [*function*]

run-program (*command* &KEY *arguments*) [*function*]

environment-variable (*string*) [*function*]

chmod (*prot file*) [*function*]

get-theory (*name*) [*function*]

context (*obj*) [*function*]

make-new-context (*theory*) [*function*]

translate-update-to-if! (*translate-update-to-if"!*) [*function*]

find-supertype ((*te subtype*)) [*function*]

pseudo-normalize (*expr* &OPTIONAL *include-typepreds?*) [*function*]

*operator** (*expr*) [*function*]

argument* (***expr*** &OPTIONAL ***args***) [*function*]

current-theory () [*function*]

current-theory-name () [*function*]

current-declaration () [*function*]

lift-predicates-in-quantifier (***ex*** &OPTIONAL ***exclude***) [*function*]

collect-references (***ex***) [*function*]

collect-type-constraints (***expr***) [*function*]

arity (***expr***) [*function*]

beta-reduce (***obj*** &OPTIONAL (***let-reduce?*** t)) [*function*]

or+ (***forms***) [*function*]

and+ (***form*** &OPTIONAL ***depth***) [*function*]

freevars (***obj***) [*function*]

conjuncts (***fmla***) [*function*]

disjuncts (*fmla*) [*function*]

3.5 Prettyprinting

3.6 Error Handling

3.7 PVS Development Hints

3.7.1 Lisp Name Conflicts

PVS defines a lot of functions, macros, and variables, and if you define a new function without first checking, you may find that you have redefined an existing function. This can often be difficult to debug, so it is better to check beforehand, using `describe` on the name first. If it indicates that it already has a definition, change the name. Thus you might create a function that builds a quantified formula, and a subfunction that creates new bindings for it. It would seem that `make-new-bindings` is an ideal name for it, until you check.

```
pvs(22): (describe 'make-new-bindings)
make-new-bindings is a symbol.
  It is unbound.
  It is internal in the pvs package.
  Its function binding is #<Function make-new-bindings>
    The function takes arguments (old-bindings alist expr)
  Its property list has these indicator/value pairs:
excl::dynamic-extent-arg-template  nil
pvs(23):
```

So you try a different name. Since the formula being constructed is really specific to foos, you try again:

```
pvs(23): (describe 'make-new-bindings-for-foos)
  make-new-bindings-for-foos is a symbol.
    It is unbound.
    It is internal in the pvs package.
  pvs(24):
```

Another technique is to define a file with functions, not worrying about the names used. Then load the file manually, and watch carefully for any warnings that indicate you are overwriting an existing entity. This technique is also useful when two or more

people have developed their own lisp files, and don't know whether they have any naming problems.

There are approaches to avoiding this problem. You can use packages, as long as your package name is unique, and you don't import names into the PVS package without checking, there is no possible way for a conflict to be introduced. This is a good technique, though the Common Lisp package mechanism can be confusing, and at times inconvenient.

A simpler approach is to use long, specific names for your functions. Lisp has no restriction on the length of it's names, though if you use other outside tools, like grep, you may find that it is best to use fewer than 255 character names, as some shells have this restriction.

For one thing, I often restart just the lisp image, rather than all of PVS. I do this by going to the *pvs* buffer, and typing "(quit)y" or by clicking on "KILL" in the "Signals" button on the menu bar (which only shows when the *pvs* buffer is current). Then I restart pvs using M-x pvs. This way all my Emacs buffers, as well as proof command history (e.g., for M-p, M-s), etc. are still available.

Another undocumented command is (clear-theories). If you type this in at the prompt in the *pvs* buffer, the current context is cleared. If you invoke (clear-theories t), then all libraries are cleared out as well. This does not affect bin files. In debugging I was often just running 'rm */*.bin' and (clear-theories t) to get a fresh typecheck. So in your instructions "in the attached sources typecheck sc_prelude/label.pvs and quit pvs to create the bin files. Typechecking label.pvs again yields an error" I instead would typecheck label, run M-x sc (which creates the bin files), then (clear-theories t) and typecheck again.

3.7.2 Debugging Hints

initialize-instance :around
    (setf foo) :around
    trace-methods

Chapter 4

The Back End: Proof Engine Interface

In this chapter we describe how to interface new rules and decision procedures. Much of the difficulty here is that these are often implemented in a language other than lisp, so efficiency considerations often require a foreign function interface, with special care needed so that the garbage collector does not have any memory leaks, or worse, leave dangling pointers. This is exacerbated if the other language has its own garbage collector, e.g., ocaml or JAVA.

The general model here is that one has an existing engine, and wants to invoke it from the PVS theorem prover. To do this, there are several steps involved:

1. Define a new proof rule that invokes the engine.
2. Translate the current sequent to the term language of the engine
3. Translate the result(s) to PVS
4. Update the proofstate

4.1 Adding New Rules

4.1.1 Defining New Rules: `addrule`

4.2 Adding New Decision Procedures

4.3 Interfacing with Lisp

4.3.1 The Subprocess Interface

The simplest approach is to use subprocesses. In this case, the engine acts as a black box, input is provided for it, and the output read in. Even here, there are many

possibilities to consider:

- Is this a terminating or nonterminating subprocess?
- How will it obtain its input and report its results?
- How are error conditions handled?

4.3.2 The Foreign Function Interface

4.4 Translation of PVS Expressions

4.4.1 Translating Input from PVS Expressions

4.4.2 Translating Output to PVS Expressions

4.5 Updating the Proofstate

Appendix A

Secondary Classes

This Appendix contains secondary classes for completeness. These classes should rarely appear as method specializers, unless there is a good reason, such as writing new prettyprint methods.

A.1 Secondary Specification Classes

`modules` $\subset \emptyset$ [*class*]

modules list of theories and top-level recursive types

This class exists so in order to print out PVS files containing multiple theories and top level datatypes.

`adtdecl` $\subset$ `typed-declaration` [*class*]

bind-decl

`adt-constructor-decl` $\subset$ `const-decl` [*class*]

ordnum

`adt-recognizer-decl` $\subset$ `const-decl` [*class*]

ordnum

`adt-accessor-decl` $\subset$ `const-decl` [*class*]

`adt-def-decl` $\subset$ `def-decl` [*class*]

A.2 Secondary Declaration Classes

lib-eq-decl ⊂ lib-decl [*class*]

uninterpreted-type-name ⊂ type-name [*class*]

type-application ⊂ type-expr [*class*]
type.....
parameters

datatype-subtype ⊂ subtype [*class*]
declared-type

A.3 Secondary Type Expression Classes

setsubtype ⊂ subtype [*class*]
formals
formula

nsetsubtype ⊂ setsubtype [*class*]

simple-expr-as-type ⊂ type-expr [*class*]
expr

expr-as-type ⊂ subtype simple-expr-as-type [*class*]

functiontype ⊂ funtype [*class*]

arraytype ⊂ funtype [*class*]

domain-tupletype ⊂ tupletype [*class*]

dep-domain-tupletype ⊂ domain-tupletype [*class*]
var-bindings

Type Variables (sort of)

type-var ⊂ type-name [*class*]

`type-variable` ⊂ `type-var` [*class*]

`tup-type-variable` ⊂ `type-var` [*class*]

`cotup-type-variable` ⊂ `type-var` [*class*]

A.4 Secondary Expression Classes

`unary-negation` ⊂ `negation` `unary-application` [*class*]

`unary-application` ⊂ `application` [*class*]

`mixfix-branch` ⊂ `if-expr` `branch` [*class*]

`field-name-expr` ⊂ `name-expr` [*class*]

`let-lambda-expr` ⊂ `lambda-expr` [*class*]

LET expressions have been extended to make it more convenient to give local function definitions, for example, `LET f(x: int) = 7*x IN f(f(f(x)))`. These are transformed by the parser into simple lets, e.g., `LET f = LAMBDA (x: int): 7*x IN f(f(f(x)))`. The lambda expression created in this case is an instance of the `let-lambda-expr` class. This is just for prettyprinting purposes.

`when-expr` ⊂ `implication` [*class*]

The `when-expr` class is a little different, when an application has the booleans `WHEN` operator, the typechecker changes its class to `when-expr`, makes sure there are two arguments as discussed in the conjunction class, and reverses its arguments. Thus, again, only the prettyprinter and related functions need to have methods for this class. The `infix-when-expr` subclass is for the infix version of a `WHEN` application.

`if-expr` ⊂ `application` [*class*]

This class is really just for prettyprinting; an expression of the form `IF a THEN b ELSE c ENDIF` is parsed into an instance of this class.
This has a subclass `chained-if-expr`. When an expression of the form `IF a THEN b ELSIF c THEN d ELSE e ENDIF` is parsed, it is treated the same as `IF a THEN b ELSE IF c THEN d ELSE e ENDIF ENDIF`, but the inner `if-expr` is converted to a `chained-if-expr`, so that the prettyprinter and related methods can handle it properly.
IMPORTANT: you probably don't want to define methods over this class, use `branch` or `application` instead.

`first-cond-expr` ⊂ `mixfix-branch` [*class*]

`single-cond-expr` ⊂ `mixfix-branch` [*class*]

`cond-expr` ⊂ `mixfix-branch` [*class*]

`last-cond-expr` ⊂ `mixfix-branch` [*class*]

`else-condition` ⊂ `unary-negation` [*class*]

`where-expr` ⊂ `let-expr` [*class*]

A `where-expr` is just syntactic sugar for a `let-expr`. This class exists primarily for prettyprinting and related purposes. As with the `let-expr`, there is a `chained-where-expr` subclass.

`coercion` ⊂ `application` [*class*]

A coercion of the form `e :: T` is converted into an application `(LAMBDA (x: T): x)(e)`. Thus this class is for prettyprinting and related functions. Note: in the future this may be different for function types, e.g., `f :: [nat -> nat]` will be translated to `(LAMBDA (g: [nat -> nat]): g)(LAMBDA (x: nat): x)(f(x))`. In other words, coercions on functions will be given the more natural closure interpretation that when applied to an element of the domain, it returns an element of the range.

`table-expr` ⊂ `expr` [*class*]

row-expr	the optional expression associated with rows
col-expr	the optional expression associated with columns
row-headings	a list of exprs associated with the rows
col-headings	a list of exprs associated with the columns
table-entries	a list of lists of exprs

Table expressions are translated into `cond-expr`s or `cases-expr`s. The slots are used to do this. If `row-expr` and `col-expr` are empty, then the `row-heading`s and `col-heading`s become conditions in the generated `cond-expr`(s), which are nested if there are both `row-heading`s and `col-heading`s. Details may be found in the language manual. Note that the slots are used to create the cond expressions, after which they are only used for prettyprinting. Thus functions that modify a table-expr (for example, substitution functions) must also modify the `table-expr` slots for the `table-expr` to be displayed consistently.
Generally it is unnecessary to create methods for `table-expr`s. When they were introduced, the parser and typechecker needed to be updated, but except for some substitution functions the prover was untouched.

`cases-table-expr` ⊂ `cases-expr table-expr` [*class*]

This is a subclass of `table-expr` created when the corresponding row or column is determined to be a `cases-expr`.

`cond-table-expr` ⊂ `first-cond-expr table-expr` [*class*]

This is a subclass of `table-expr` created when the corresponding row or column is determined to be a `cond-expr`.

`single-cond-table-expr` ⊂ `single-cond-expr table-expr` [*class*]

This is similar to the `single-cond-expr` class, created when there is a single row or column in a table.

`set-expr` ⊂ `lambda-expr` [*class*]

Recall that `{x: T | p(x)}` is just alternative syntax for `LAMBDA (x: T): p(x)`. This is a subclass of `lambda-expr`, so that set-exprs are prettyprinted correctly.

`implicit-conversion` ⊂ `application` [*class*]

This is for the application of conversions other than lambda-conversions. It is much simpler, in that it only involves the application of the found conversion to the argument. This class exists so that untypechecking can remove the application, and so that prettyprinting can print properly when `*show-conversions*` is `nil`.

The `lambda-conversion` and `argument-conversion` classes are associated with

lambda-conversions. For example, suppose we have the following declarations:

```
state: TYPE
k: [int -> [state -> int]] = (LAMBDA i: (LAMBDA s: i))
CONVERSION k
f: [[state -> int] -> bool]
ss: [state -> int]
```

Then `f(ss + 1)` is converted by the typechecker to `(LAMBDA s: f(ss(s) + 1))`, where the outer `lambda-expr` is a `lambda-conversion` instance and `ss(s)` is an `argument-conversion`. These class exist so that untypechecking can remove the conversions, and so that prettyprinting can print properly when `*show-conversions*` is `nil`.

`lambda-conversion` ⊂ `lambda-expr` [*class*]

`argument-conversion` ⊂ `application` [*class*]

`funtype-conversion` ⊂ `lambda-expr` [*class*]

`domain-conversion` the conversion applied to the domain

`range-conversion`. the conversion applied to the range

As described in the language manual, when looking for conversions where the given and expected types are function types, conversions are searched for the ranges and the domains (contravariantly). If successful, a `funtype-conversion` is created, and the slots set accordingly.

`rectype-conversion` ⊂ `lambda-expr` [*class*]

`conversions` the list of conversions found

As described in the language manual, when looking for conversions where the given and expected types are record types, conversions are searched for on the corresponding component types. If successful, a `rectype-conversion` is created, and the `conversions` slot set accordingly.

`tuptype-conversion` ⊂ `lambda-expr` [*class*]

`conversions` the list of conversions found

As described in the language manual, when looking for conversions where the given and expected types are tuple types, conversions are searched for on the corresponding component types. If successful, a `tuptype-conversion` is created, and the `conversions` slot set accordingly.

`uni-assignment` ⊂ `assignment` [*class*]

PVS supports two forms of assignment arguments, e.g., `x := 3` and `(x) := 3`. The former generates a `uni-assignment`, but in all other respects these are the same. Only the prettyprinter should care.

`uni-maplet` ⊂ `maplet uni-assignment` [*class*]

Same as `uni-assignment`, but for maplets.

`quoted-assign` ⊂ `syntax` [*class*]

This is a mixin for assignment arguments that are quoted. For example, the `a` in `` `a := 0 `` or the `1` in `` `1 := 0 ``.

`id-assign` ⊂ `name-expr quoted-assign` [*class*]

This is for assignemnt argument identifiers that are quoted, i.e., the `a` in `` `a := 0 ``. It exists primarily for prettyprinting purposes.

`field-assign` ⊂ `field-assignment-arg id-assign` [*class*]

`proj-assign` ⊂ `number-expr quoted-assign` [*class*]

`accessor-assignment-arg` ⊂ `accessor-name-expr` [*class*]

`accessor-assign` ⊂ `accessor-assignment-arg id-assign` [*class*]

A.5 Secondary Binding Declaration Classes

`arg-bind-decl` ⊂ `bind-decl` [*class*]

`pred-bind-decl` ⊂ `bind-decl` [*class*]

`untyped-bind-decl` ⊂ `bind-decl` [*class*]

`full-modname` ⊂ `modname` [*class*]

`datatype-modname` ⊂ `modname` [*class*]

`modname-no-tccs` ⊂ `modname` [*class*]

`conversion-resolution` ⊂ `resolution` [*class*]
conversion an expression

`lambda-conversion-resolution` ⊂ `resolution` [*class*]
k-conv-type a type expression

`recursive-function-resolution` ⊂ `resolution` [*class*]
conversion an expression

`store-print-type` ⊂ ∅ [*class*]
print-type
type.....

Appendix B

A Prooftree Display Example

Here we give an example of how a different prooftree display could be created using PVS without the Emacs and Tcl/Tk interfaces. The `prooftree.lisp` file for this example contains the following.[1]

```
(in-package :pvs)

(pvs-init)
(setq *pvs-context-path* (shortpath (working-directory)))

(defun pvs-wish-source (file)
  (format t "~%PT:~a" file))

(prove-file-at "prelude" nil 4820 t "prelude" "prelude.pvs" 0 nil t)
```

Running `pvs -raw -L prooftree.lisp` results in the following output.

```
% pvs -raw -L prooftree.lisp

Loading compiled patch file ~/.pvs.lfasl
;   Fast loading /homes/owre/.pvs.lfasl
### Caml startup
### successful
;   Fast loading from bundle code/ffcompat.fasl.
Warning: pvs-wish-source, :operator was defined in
         /project/pvs/pvs3.0/src/prover/wish.lisp and is now being
         defined in /export/u1/homes/owre/pvs3.0/doc/api/prooftree.lisp

PT:/tmp/pvs-47030.p1
PT:/tmp/pvs-47030.p2
PT:/tmp/pvs-47030.p3
nat2bv_rew :
```

[1] Note that this example is actually for PVS version 3.1, it will be simpler in 3.2.

```
  |-------
{1}   FORALL (bv: bvec[N], val: below(exp2(N))):
        nat2bv(val) = bv IFF bv2nat(bv) = val

Rerunning step: (skosimp*)
Repeatedly Skolemizing and flattening,
this simplifies to:
nat2bv_rew :

  |-------
{1}   nat2bv(val!1) = bv!1 IFF bv2nat(bv!1) = val!1

PT:/tmp/pvs-47030.p4
PT:/tmp/pvs-47030.p5
Rerunning step: (typepred "nat2bv(val!1)")
Adding type constraints for  nat2bv(val!1),
this simplifies to:
nat2bv_rew :

{-1}  bv2nat(nat2bv(val!1)) = val!1
  |-------
[1]   nat2bv(val!1) = bv!1 IFF bv2nat(bv!1) = val!1

PT:/tmp/pvs-47030.p6
PT:/tmp/pvs-47030.p7
Rerunning step: (prop)
Applying propositional simplification,
this yields  2 subgoals:
nat2bv_rew.1 :

{-1}  nat2bv(val!1) = bv!1
[-2]  bv2nat(nat2bv(val!1)) = val!1
  |-------
{1}   bv2nat(bv!1) = val!1

PT:/tmp/pvs-47030.p8
PT:/tmp/pvs-47030.p9
Rerunning step: (assert)
Simplifying, rewriting, and recording with decision procedures,
PT:/tmp/pvs-47030.p10

This completes the proof of nat2bv_rew.1.

nat2bv_rew.2 :

{-1}  bv2nat(bv!1) = val!1
[-2]  bv2nat(nat2bv(val!1)) = val!1
  |-------
{1}   nat2bv(val!1) = bv!1
```

```
PT:/tmp/pvs-47030.p11
PT:/tmp/pvs-47030.p12
Rerunning step: (rewrite "bv2nat_inj")
Found matching substitution:
y: bvec[N] gets bv!1,
x gets nat2bv(val!1),
Rewriting using bv2nat_inj, matching in *,
PT:/tmp/pvs-47030.p13

This completes the proof of nat2bv_rew.2.

PT:/tmp/pvs-47030.p14
PT:/tmp/pvs-47030.p15
PT:/tmp/pvs-47030.p16
PT:/tmp/pvs-47030.p17
Q.E.D.

PT:/tmp/pvs-47030.p18
PT:/tmp/pvs-47030.p19

Run time  = 0.20 secs.
Real time = 0.37 secs.
:pvs-loc nil&prelude.pvs&(4820 2 4820 64) :end-pvs-loc
Allegro CL Enterprise Edition
6.2 [Linux (x86)] (Feb 14, 2003 18:42)
Copyright (C) 1985-2002, Franz Inc., Berkeley, CA, USA.  All Rights Reserved.

This development copy of Allegro CL is licensed to:
   [5377] SRI International

;; Optimization settings: safety 1, space 1, speed 3, debug 1.
;; For a complete description of all compiler switches given the
;; current optimization settings evaluate (explain-compiler-settings).
;;---
;; Current reader case mode: :case-sensitive-lower
cl-user(1):
```

The contents of the tmp files are as follows:

```
/tmp/pvs-47030.p1:
setup-proof nat2bv_rew bv_nat /export/u1/homes/owre/pvs3.0/doc/api/ 1 1

/tmp/pvs-47030.p2:
delete-proof-subtree nat2bv_rew bv_nat top
proof-num-children nat2bv_rew bv_nat top 0
proof-sequent nat2bv_rew bv_nat top {nat2bv_rew} {
nat2bv_rew :

  |-------
{1}   FORALL (bv: bvec[N], val: below(exp2(N))):
        nat2bv(val) = bv IFF bv2nat(bv) = val
```

```
}
proof-show nat2bv_rew bv_nat top 1
layout-proof nat2bv_rew bv_nat 1

/tmp/pvs-47030.p3:
proof-current nat2bv_rew bv_nat top
/tmp/pvs-47030.p4:
delete-proof-subtree nat2bv_rew bv_nat top
proof-num-children nat2bv_rew bv_nat top 1
proof-rule nat2bv_rew bv_nat top {(skosimp*)}
proof-sequent nat2bv_rew bv_nat top {nat2bv_rew} {
nat2bv_rew :

  |-------
[1]   FORALL (bv: bvec[N], val: below(exp2(N))):
        nat2bv(val) = bv IFF bv2nat(bv) = val
}
proof-show nat2bv_rew bv_nat top 1
proof-num-children nat2bv_rew bv_nat top.0 0
proof-sequent nat2bv_rew bv_nat top.0 {nat2bv_rew} {
nat2bv_rew :

  |-------
{1}   nat2bv(val!1) = bv!1 IFF bv2nat(bv!1) = val!1
}
proof-show nat2bv_rew bv_nat top.0 1
layout-proof nat2bv_rew bv_nat 1

/tmp/pvs-47030.p5:
proof-current nat2bv_rew bv_nat top.0
/tmp/pvs-47030.p6:
delete-proof-subtree nat2bv_rew bv_nat top.0
proof-num-children nat2bv_rew bv_nat top.0 1
proof-rule nat2bv_rew bv_nat top.0 {(typepred "nat2bv(val!1)")}
proof-sequent nat2bv_rew bv_nat top.0 {nat2bv_rew} {
nat2bv_rew :

  |-------
{1}   nat2bv(val!1) = bv!1 IFF bv2nat(bv!1) = val!1
}
proof-show nat2bv_rew bv_nat top.0 1
proof-num-children nat2bv_rew bv_nat top.0.0 0
proof-sequent nat2bv_rew bv_nat top.0.0 {nat2bv_rew} {
nat2bv_rew :

{-1}  bv2nat(nat2bv(val!1)) = val!1
  |-------
[1]   nat2bv(val!1) = bv!1 IFF bv2nat(bv!1) = val!1
}
proof-show nat2bv_rew bv_nat top.0.0 1
layout-proof nat2bv_rew bv_nat 1
```

```
/tmp/pvs-47030.p7:
proof-current nat2bv_rew bv_nat top.0.0
/tmp/pvs-47030.p8:
delete-proof-subtree nat2bv_rew bv_nat top.0.0
proof-num-children nat2bv_rew bv_nat top.0.0 2
proof-rule nat2bv_rew bv_nat top.0.0 {(prop)}
proof-sequent nat2bv_rew bv_nat top.0.0 {nat2bv_rew} {
nat2bv_rew :

{-1}  bv2nat(nat2bv(val!1)) = val!1
  |-------
[1]   nat2bv(val!1) = bv!1 IFF bv2nat(bv!1) = val!1
}
proof-show nat2bv_rew bv_nat top.0.0 1
proof-num-children nat2bv_rew bv_nat top.0.0.0 0
proof-sequent nat2bv_rew bv_nat top.0.0.0 {nat2bv_rew.1} {
nat2bv_rew.1 :

{-1}  nat2bv(val!1) = bv!1
[-2]  bv2nat(nat2bv(val!1)) = val!1
  |-------
{1}   bv2nat(bv!1) = val!1
}
proof-show nat2bv_rew bv_nat top.0.0.0 1
proof-num-children nat2bv_rew bv_nat top.0.0.1 0
proof-sequent nat2bv_rew bv_nat top.0.0.1 {nat2bv_rew.2} {
nat2bv_rew.2 :

{-1}  bv2nat(bv!1) = val!1
[-2]  bv2nat(nat2bv(val!1)) = val!1
  |-------
{1}   nat2bv(val!1) = bv!1
}
proof-show nat2bv_rew bv_nat top.0.0.1 1
layout-proof nat2bv_rew bv_nat 1

/tmp/pvs-47030.p9:
proof-current nat2bv_rew bv_nat top.0.0.0
/tmp/pvs-47030.p10:
proof-done nat2bv_rew bv_nat top.0.0.0 1
/tmp/pvs-47030.p11:
delete-proof-subtree nat2bv_rew bv_nat top.0.0.0
proof-num-children nat2bv_rew bv_nat top.0.0.0 0
proof-rule nat2bv_rew bv_nat top.0.0.0 {(assert)}
proof-sequent nat2bv_rew bv_nat top.0.0.0 {nat2bv_rew.1} {
nat2bv_rew.1 :

{-1}  nat2bv(val!1) = bv!1
[-2]  bv2nat(nat2bv(val!1)) = val!1
  |-------
```

```
{1}   bv2nat(bv!1) = val!1
}
proof-done nat2bv_rew bv_nat top.0.0.0 1
proof-show nat2bv_rew bv_nat top.0.0.0 1
layout-proof nat2bv_rew bv_nat 1

/tmp/pvs-47030.p12:
proof-current nat2bv_rew bv_nat top.0.0.1
/tmp/pvs-47030.p13:
proof-done nat2bv_rew bv_nat top.0.0.1 1
/tmp/pvs-47030.p14:
proof-done nat2bv_rew bv_nat top.0.0 1
/tmp/pvs-47030.p15:
proof-done nat2bv_rew bv_nat top.0 1
/tmp/pvs-47030.p16:
proof-done nat2bv_rew bv_nat top 1
/tmp/pvs-47030.p17:
proof-done nat2bv_rew bv_nat top 1
/tmp/pvs-47030.p18:
delete-proof-subtree nat2bv_rew bv_nat top.0.0.1
proof-num-children nat2bv_rew bv_nat top.0.0.1 0
proof-rule nat2bv_rew bv_nat top.0.0.1 {(rewrite "bv2nat_inj")}
proof-sequent nat2bv_rew bv_nat top.0.0.1 {nat2bv_rew.2} {
nat2bv_rew.2 :

{-1}  bv2nat(bv!1) = val!1
[-2]  bv2nat(nat2bv(val!1)) = val!1
  |-------
{1}   nat2bv(val!1) = bv!1
}
proof-done nat2bv_rew bv_nat top.0.0.1 1
proof-show nat2bv_rew bv_nat top.0.0.1 1
layout-proof nat2bv_rew bv_nat 1

/tmp/pvs-47030.p19:
proof-current nat2bv_rew bv_nat {}
```

The contents of the files are calls to Tcl/Tk functions defined in the `wish/pvs-support.tcl` file included in the PVS distribution. The following briefly describes the intent of the functions. It is important to keep in mind that these are intended to support the interactive prover, so the prooftree display is dynamic, and commands are send as the proof develops.

`setup-proof`	*fid*	used to initialize a proof display
	thid	
	dir ctr	
`delete-proof-subtree`	*fid*	deletes the subtree at the node specified by
	thid	the path; usually done in order to replace it
	path	with a new node
`proof-show`	*fid*	recomputes the internal Tcl/Tk structures
	thid	associated with the proof
	path 1	
`layout-proof`	*fid*	recomputes the layout of the proof window
	thid 1	
`proof-current`	*fid*	indicates that the specified path is the
	thid	current one. If the path is empty ({}),
	path	there is no current path (i.e., the proof of the formula is finished)
`proof-done`	*fid*	indicates the proof on this branch is
	thid	complete
	path 1	
`proof-num-children`	*fid*	gives the number of children of the specified
	thid	path
	path	
	num_children	
`proof-rule`	*fid*	the rule associated with the specified path
	thid	
	path	
	{*rule*}	
`proof-sequent`	*fid*	the sequent associated with the specified
	thid	path
	path	
	{*label*}	
	{*sequent*}	
`proof-tcc`	*path*	indicates that the given path is a TCC branch

Here *fid* is the formula identifier, the *thid* is the theory containing the formula, the *path* is the path from the root to the given sequent, of the form `top.0.1.3`.

Bibliography

[1] Andrew Adams, Martin Dunstan, Hanne Gottliebsen, Tom Kelsey, Ursula Martin, and Sam Owre. Computer algebra meets automated theorem proving: Integrating Maple and PVS. In Richard J. Boulton and Paul B. Jackson, editors, *Theorem Proving in Higher Order Logics, TPHOLs 2001*, volume 2152 of *Lecture Notes in Computer Science*, pages 27–42, Edinburgh, Scotland, September 2001. Springer-Verlag. 34

[2] P. Lee, F. Pfenning, J. Reynolds, G. Rollins, and D. Scott. Research on semantically based program-design environments: The ergo project in 1988. Technical Report CMU-CS-88-118, Department of Computer Science, Carnegie Mellon University, 1988. 82

[3] S. Owre, N. Shankar, J. M. Rushby, and D. W. J. Stringer-Calvert. *PVS Language Reference*. Computer Science Laboratory, SRI International, Menlo Park, CA, September 1999. 1

[4] S. Owre, N. Shankar, J. M. Rushby, and D. W. J. Stringer-Calvert. *PVS System Guide*. Computer Science Laboratory, SRI International, Menlo Park, CA, September 1999. 1

[5] N. Shankar, S. Owre, J. M. Rushby, and D. W. J. Stringer-Calvert. *PVS Prover Guide*. Computer Science Laboratory, SRI International, Menlo Park, CA, September 1999. 1, 39

Index