Formula Declarations

- `transitive: AXIOM x < y AND y < z IMPLIES x < z`
- `nonzero_fac: THEOREM fac(n) /= 0`
- `poset: ASSUMPTION poset?(T,<=)  % Only in ASSUMINGs`

Expressions

- Equality—(`=`, `/=`)
  Defined for any type; both sides must be the same type. With boolean, `=` is treated as `IFF`.
  - `x * y = 4`
  - `true /= 1  % Illegal`
- Arithmetic—(`+`, `-`, `*`, `/`, `<`, `<=`, `>`, `>=`, `0`, `1`, ...)
  - `((x + 1) * x) / 2 < x * x`
- Logical—(`true`, `false`, `AND`, `&`, `OR`, `IMPLIES`, `=>`, `WHEN`, `NOT`, `IFF`, `<=>`, `FORALL`, `ALL`, `EXISTS`, `SOME`)
  - `(FORALL e: (EXISTS d:`
    `abs(f(x) - f(y)) < d) IMPLIES abs(x - y) < e)`
- `IF-THEN-ELSE` — The `THEN` and `ELSE` parts must have compatible types.
  - `IF x=0 THEN 1 ELSIF y=0 THEN 2 ELSE y/x ENDIF`
- `CASES` — Pattern matching on datatypes.
  - `CASES x OF`
    `cons(x,y): append(reverse(y), cons(x, null))`
    `ELSE null`
    `ENDCASES`
- Function application, lambda-abstraction & function update
  - `f(1,2)(0)`
  - `(lambda x: x + 1)`
  - `f WITH [(0) := 1, (1) := 0]`
- Record construction, field selection & record update
  - `(# pointer := 1, astack := (LAMBDA x: 0) #)`
  - `astack(r)  % r.astack NOT allowed`
  - `r WITH [pointer := 2, (astack)(1) := 1]`
- Tuple construction & projection
  - `(1, true, (LAMBDA (x:int) x + 37))`
  - `proj_3(tup)`
- `LET`, `PLET`, & `WHERE`
  - `LET x = 2, y:nat = x*x IN f(x,y)  % ≡f(2,4)`
  - `PLET x = 2, y:nat = x*x IN f(x,y)`
  - `f(x,y) WHERE x = 2, y:nat = x*x  % Like  LET`
- Coercion — Coercion indicates the expected type to the typechecker to resolve ambiguity.
  - `a + b:natural`
  - `(LAMBDA n -> nat: n - m)  % LAMBDA coercion`
- Names — If `foo` is declared in theory `bar`, then the following are allowable references (the first two may be ambiguous).
  - `foo`
  - `foo[int]`
  - `bar[int].foo`

Type Declarations

- Uninterpreted types and subtypes
  - `foo: TYPE`
  - `some_nums: TYPE FROM number`
  - `nat_to_10: TYPE = {x:nat | x <= 10}`
  - `posint: TYPE = {x:integer | x > 0} CONTAINING 1`
  - `ptype: TYPE = (pred?)  % same as {x | pred?(x)}`
- Function types
  - `intf: TYPE = FUNCTION[int, int -> int]`
  - `altf: TYPE = [int, int -> int]     % same as above`
  - `inta: TYPE = ARRAY[int,int -> int] % same as above`
- Tuple Types
  - `tuptype: TYPE = [int, bool, [int -> int]]`
- Record types
  - ```
    stack: TYPE = [# pointer: nat,
                     astack: [nat -> t] #]
    ```
- Dependent Types
  - `pfun: TYPE = [# dom: predicate[t1], pfn:[(dom)->t2] #]`
  - `date: TYPE = [y,m:nat, {d:nat | d <= days(m,y)}]`
  - `tmod: TYPE = [n,m:int -> {x:nat | x < m}]`
- Enumeration types
  - `color: TYPE = {red, green, blue}`
- Datatypes
  - ```
    list[t:TYPE] : DATATYPE
          BEGIN
            null: null?
            cons (car: t, cdr :list) :cons?
          END list
    ```

Usings, Exportings, and Theory Abbreviations

- `USING orderings[int], set[foo[nat]]`
- `EXPORTING foo, bar WITH set[foo]`
- `pset: THEORY = sets[list[nat]]`

Constants and Recursive Definitions

- `some_int: int`
- `max: int = 10`
- ```
  abs: [int -> nat] =
        (LAMBDA x: IF x < 0 THEN -x ELSE x ENDIF)
  ```
- `abs(x:int): nat = IF x < 0 THEN -x ELSE x ENDIF`
- `sum(f,x,y): int         % f,x,y prev declared VAR`
- `sum(f,(x,y:int)): int       % f prev declared VAR`
- ```
  fac(n): RECURSIVE nat =
       (IF n = 0 THEN 1 ELSE n*fac(n-1) ENDIF)
    MEASURE (LAMBDA n: n)
  ```

Variable Declarations

- `x, y, z: VAR int`
- `f: VAR [int -> [int -> int]]`

PVS Language Examples

Theories

```
orderings[t: TYPE] : THEORY
  BEGIN
    x, y, z: VAR t
    pp, qq: VAR predicate[t]
    <= : VAR predicate[[t,t]]

    reflexive?(<=) : bool = (FORALL x: x <= x)

    antisymmetric?(<=) : bool =
      (FORALL x, y: (x <= y & y <= x) => x = y)

    transitive?(<=) : bool =
      (FORALL x, y, z: (x <= y & y <= z) => x <= z)

    partialorder?(<=) : bool =
      reflexive?(<=) & antisymmetric?(<=)
                     & transitive?(<=)

    linear?(<=) : bool =
      (FORALL x, y : x <= y OR y <= x)

    total_order?(<=) : bool =
      partialorder?(<=) & linear?(<=)
  END orderings

sort [domain, range: TYPE,
      (USING orderings[range], orderings[domain])
      d_order: (total_order?[domain]),
      r_order: (partial_order?[range])] : THEORY
  BEGIN
    A, B, C: VAR ARRAY[domain->range]
    sorted?(A): bool =
      (FORALL (x, y: domain): (d_order(x, y) & x /= y)
                     => NOT r_order(A(y), A(x)))
  END sort
```

Lexical Rules

Comments start with % and go to the end of the line

Identifiers are composed of letters, digits, question mark,and underscores; they must begin with a letter.

Numbers are composed of digits; floating point numbers are not supported.

Reserved Words

ALL	COROLLARY	FALSE	LAW	RECURSIVE
AND	DATATYPE	FORALL	LEMMA	SOME
ARRAY	ELSE	FORMULA	LET	SUBLEMMA
ASSUMING	ELSIF	FROM	MEASURE	THEN
ASSUMPTION	END	FUNCTION	NOT	THEOREM
AXIOM	ENDASSUMING	IF	O	THEORY
BEGIN	ENDCASES	IFF	OBLIGATION	TRUE
BUT	ENDIF	IMPLIES	OF	TYPE
CASES	EPSILON	IMPORTING	OR	VAR
CLAIM	EXISTS	IN	PLET	WHEN
CONJECTURE	EXPORTING	LAM	POSTULATE	WHERE
CONTAINING	FACT	LAMBDA	PROPOSITION	WITH

Special Symbols

&	(	)	*	+	,	-	->	.	/	/=	:	:=
;	<	<=	<=>	=	=>	>	>=	|	[	]	[#	#]
{	}	(#	#)	\/	/\	~	==	[]	<>	^		

(auto-rewrite &REST *names*) .... Add automatic rewrite rules
(auto-rewrite-theory &REST *names*) .....
Add automatic rewrite rules from a theory
(do-rewrite &OPT *fnums[*]*) ..... Apply auto-rewrites
(stop-rewrite &REST *names*) .....
Turn off automatic rewrite rules
(stop-rewrite-theory &REST *names*) .....
Turn off automatic rewrite rules from a theory

Proof Strategies

(quote *strat*) ..... Identity strategy
(try *strat strat strat*) ..... Subgoaling and backtracking
(if *condition strat strat*) ..... Conditional selection of strategies
(let ((*var*$_1$ *expr*$_1$)$\cdots$(*var*$_n$ *expr*$_n$)) *strat*) .....
Using Lisp in strategies

Defined Rules

(prop) ..... Propositional simplification
(ground) ... Decision procedures and propositional simplification
(rewrite-lemma *lemma subst* &OPT *fnums[*] dir[*LR*]) .....
Rewrite using a lemma
(rewrite *lname* &OPT *fnums[*] subst target-fnums[*] dir[*LR*] *order[*IN*])
..... Matching and rewriting with lemmas
(inst &OPT *fnum[*] &REST *subst*) .....
Instantiation of universal strength without copying
(inst-cp &OPT *fnum[*] &REST *subst*) .....
Instantiation of universal strength with copying
(quant? &OPT *fnum[*] subst where[*] copy? if-match*) .....
Instantiation of universal strength quantifiers
(inst? &OPT *fnum[*] subst where[*] copy? if-match*) .....
Instantiation of universal strength quantifiers
(quit) ..... Terminate the proof attempt
(replace-extensionality *expr expr* &OPT *expected*) .....
Show equality using extensionality
(apply-extensionality &OPT *fnum[+]*) .....
Show equality using extensionality on formula
(induct *var* &OPT *fnum[*1*] name*) ... Employ induction scheme
(forward-chain *name*) ..... Employ induction scheme

Strategies

(query*) ..... Basic interaction strategy
(skolem! &OPT *fnums[*]*) .....
Automatic introduction of Skolem constants
(skosimp &OPT *fnums[*]*) ..... Skolemize and flatten
(skosimp*) ..... Repeatedly Skolemize and flatten
(else *strat strat*) ..... Simple backtracking
(repeat *strat*) ..... Iterate along main proof branch
(repeat* *strat*) ..... Iterate along all branches
(then *strat strat*) ..... A sequencing strategy
(then* &REST *strats*) ..... Apply strategies in sequence
(then@ &REST *strats*) .....
Apply strategies in sequence along main branch
(spread *strat strats*) ..... Assign strategies to subgoals
(branch *strat strats*) ..... Assign strategies to subgoals
(try-branch *strat strats strat*) ..... Branch or backtrack
(prop$) ..... Propositional simplification
(ground$) . Decision procedures and propositional simplification
(rerun &OPT *proof*) ..... Rerun a proof or partial proof
(tcc-strategy) Strategy for proving type correctness conditions

PVS Prover Commands

Prover commands are entered in the `*pvs*` buffer at the `Rule?` prompt. Commands are interpreted by lisp, so must be surrounded by parentheses, and arguments are separated by whitespace (`Space`, `Tab`, or `Return`). Some commands require PVS names or expressions; these must be surrounded by double quotes (`"`). `Return` enters the command, unless parentheses or strings are unbalanced.

The arguments are shown in *emphasized* font. Optional arguments follow the keyword &OPT and may be omitted. They may be provided in the order listed, or followed by a keyword whose name is derived form the argument name preceded by a colon, *e.g.,* `(expand "foo" :beta-reduce t)`. An &REST keyword indicates that one or more of the following argument may be provided.

Basic Rules

`(help` &OPT *name[*]*`)` ..... Describe prover commands
`(skip)` ..... Do nothing
`(postpone)` ..... Go to next remaining goal
`(fail)` ..... Propagate failure to the parent
`(undo` &OPT *to[1]*`)` ..... Undo proof to an ancestor

Structural Rules

`(delete` &REST *fnums*`)` ..... Delete selected formulas
`(hide` &REST *fnums*`)` ..... Hide selected formulas
`(reveal` &REST *fnums*`)` ..... Reveal some hidden formulas

Propositional Rules

`(propax)` ..... Propositional axioms
`(flatten` &REST *fnums[*]*`)` ..... Disjunctive simplification
`(split` &OPT *fnum[*]*`)` ..... Conjunctive splitting
`(lift-if` &REST *fnums[*]*`)` ..... Lift embedded `IF` connectives
`(case` &REST *formulas*`)` ..... Case analysis on formulas
`(iff` &REST *fnums*`)` ..... Convert Boolean equality to `IFF`

Quantifier Rules

`(skolem` *fnum constants*`)` .....
Replace universally quantified variables with new constants
`(quant` *fnum[*] terms* &OPT *copy?*`)` .....
Instantiate existentially quantified variables

Equality Rules

`(lemma` *lname* &OPT *subst*`)` ..... Employing instances of lemmas
`(expand` *dname* &OPT *fnum[*] occurrence if-simplifies beta-reduce[T]*`)`
..... Expand (and simplify) function definitions
`(replace` *fnum* &OPT *fnums[*] direction*`)` .....
Rewrite using equalities
`(beta` &OPT *fnums rewrite-flag*`)` ..... Beta reduction
`(extensionality` *type*`)` .....
Add an extensionality premise derived from the type
`(name` *name expr*`)` ..... Introduce a *expr=name* antecedent

Other Rules

`(typepred` &REST *exprs*`)` Make implicit type constraints explicit
`(apply` *strategy*`)` ..... Make proof strategy *strat* atomic
`(record` &OPT *fnums[*]*`)` .....
Record assumptions for the decision procedures
`(simplify` &OPT *fnums[*]*`)` .....
Simplify using the decision procedures
`(assert` &OPT *fnums[*] rewrite-flag flush? linear?*`)` .....
Simplify using the decision procedures

Browsing Commands

show-declaration	`M-.`
find-declaration	`M-,`
whereis-declaration-used	`M-;`
list-declarations	`M-:`

Status Commands

status-theory	`M-x stt, C-c C-s t`
status-pvs-file	`M-x stf, C-c C-s f`
status-importchain	`M-x sti, C-c C-s i`
importchain	`M-x ic`
importbychain	`M-x ibc`
status-proof	`M-x sp`
status-proof-theory	`M-x spt`
status-proof-importchain	`M-x spi`
status-proofchain	`M-x spc`

Environment Commands

help-pvs	
help-pvs-language	
help-pvs-prover	
whereis-pvs	
pvs-version	
pvs-mode	
remove-popup-buffer	`C-z 1`
pvs-status	
keyboard-quit	`C-g`
reset-pvs	`C-z C-g`

LaTeX Substitution Files

LaTeX Substitutions for file `foo.pvs` may come from any of the following files.

File name	Location
`foo.sub`	the directory of the current context
`pvs-tex.sub`	the directory of the current context
`pvs-tex.sub`	user's home directory
`pvs-tex.sub`	the main PVS directory

Examples of substitution entries—numbers refer to the number of arguments; thus the third entry translates `f2[3,G]` (to G_3^f) but not `f2[int]`, and the last entry translates, e.g., `f4(G)(1,n)` (to $\sum_{i=1}^{n} G(i,1)$). Length is an estimation of the size of the translation, ignoring the size of the arguments.

Identifier	Type	Length	Substitution
`THEORY`	`key`	9	`{\large\bf Theory}`
`f1`	`id`	3	`{\rm bar}`
`f2`	`id[2]`	2	`{#2_{#1}^{f}}`
`f3`	`2`	2	`{#1^#2}`
`f4`	`(1 2)`	3	`{\sum_{i=#2}^{#3}#1(i,#2)}`

PVS Files

foo.pvs	Specification file (contains theories)
foo.prf	Saved proofs for *foo*.pvs
.pvscontext	Context information
foo-alltt.tex	Alltt-printed version of *foo*
foo.tex	LaTeX-printed version of *foo*
pvs-files.tex	LaTeX file generated for testing Alltt and LaTeX-printed files

Adding and Modifying Declarations

add-declaration
modify-declaration

Prettyprinting Commands

prettyprint-theory	`M-x ppt, C-c C-q t`
prettyprint-file	`M-x ppf, C-c C-q f`
prettyprint-declaration	`M-x ppd, C-M-q`
prettyprint-region	`M-x ppr, C-M-\`

Viewing TCCs

prettyprint-expanded	`M-x ppe, C-c C-q e`
show-tccs	

PVS File and Theory Commands

find-pvs-file	`M-x ff, C-c C-f`
find-theory	`M-x ft`
view-prelude-file	`M-x vpf`
view-prelude-theory	`M-x vpt`
new-pvs-file	`M-x nf`
new-theory	`M-x nt`
import-pvs-file	`M-x imf`
import-theory	`M-x imt`
delete-pvs-file	`M-x df`
delete-theory	`M-x dt`
save-pvs-file	`C-x C-s`
save-some-pvs-files	`M-x ssf`
dump-pvs-files	
undump-pvs-files	

Printing Commands

pvs-print-buffer	
pvs-print-region	
print-theory	`M-x ptt`
print-pvs-file	`M-x ptf`
print-importchain	`M-x pti`
alltt-theory	`M-x alt, C-c C-a t`
alltt-pvs-file	`M-x alf, C-c C-a f`
alltt-importchain	`M-x ali, C-c C-a i`
latex-theory	`M-x ltt, C-c C-l t`
latex-pvs-file	`M-x ltf, C-c C-l f`
latex-importchain	`M-x lti, C-c C-l i`
latex-proof	`M-x ltp, C-c C-l p`
latex-theory-view	`M-x ltv, C-c C-l v`

Context Commands

list-pvs-files	`M-x lf`
list-theories	`M-x lt`
change-context	`M-x cc`
save-context	`M-x sc`
context-path	`M-x cp`

PVS Reference Card

DRAFT (for version 1.0β)

The commands which appear below are given with their abbreviations and keybindings, if any. For example, the command Prove with aliases `pr` and `C-c p` indicates that the parse command can be invoked by the Emacs extended commands `M-x prove` or `M-x pr`, or the key binding `C-c p`.

Entering and Exiting PVS

To enter PVS just `cd` to a working directory and type `pvs`.

suspend-pvs	`C-x C-z`
exit-pvs	`C-x C-c`

Getting Help

help-pvs	`C-h p`
help-pvs-language	
help-pvs-prover	

Editing PVS Files

forward-theory	`M-]`
backward-theory	`M-[`
find-unbalanced-pvs	`C-c ]`
comment-region	`C-c ;`

Basic PVS Commands

parse	`M-x pa`
typecheck	`M-x tc, C-c t`
typecheck-prove	`M-x tcp`

Prove Commands

prove	`M-x pr, C-c p`
redo-proof	`C-c C-p r`
prove-theory	`M-x prt, C-c C-p t`
prove-pvs-file	`M-x prf, C-c C-p f`
prove-importchain	`M-x pri, C-c C-p i`
prove-proofchain	`M-x prp, C-c C-p p`

Proof Editing Commands

edit-proof	
install-proof	`C-c C-i`
revert-proof	
show-proof-file	
show-orphaned-proofs	

Proof Information Commands

show-proof
show-last-proof
ancestry
siblings
show-hidden-formulas
show-auto-rewrites