

# Kurzanleitung zu PVS

## Teil 1

### Allgemeines

#### Bezugsquellen und Installation

Umfassende und aktuelle Informationen zu PVS gibt es auf der PVS-Homepage unter: <http://pvs.csl.sri.com/>.

Die aktuelle Version von PVS ist 3.2 und erhältlich für Sun Sparc Workstations unter Solaris 2 und Intel x86 Maschinen mit Linux-Betriebssystem.

Installationsdateien von PVS sind lokal in Ulm vorhanden und können über <http://www.informatik.uni-ulm.de/ki/PVS/mirror.html> bezogen werden. Installationshinweise finden sich in der Datei <ftp://ftp.informatik.uni-ulm.de/pub/KI/pvs/INSTALL>.

#### Dokumentation und weitere Informationen

Die ausführliche Dokumentation zu PVS ist nicht ganz aktuell, sondern nur für die Vorgängerversion 2.4 erhältlich, siehe <ftp://ftp.informatik.uni-ulm.de/pub/KI/pvs/pvs2.4/pvs-2.4-doc.tgz>. Neuerungen werden in den Release-Notes zu den Versionen 3.0 bis 3.2 unter <http://pvs.csl.sri.com/announcements/announce3.0.shtml>, <http://pvs.csl.sri.com/announcements/announce3.1.shtml> bzw. <http://pvs.csl.sri.com/announcements/announce3.2.shtml> erklärt.

Die Dokumentation zu PVS ist in drei Dokumente aufgeteilt: das PVS System wird im *PVS System Guide* beschrieben, während sich *PVS Language Reference* und *PVS Prover Guide* der Spezifikationsprache bzw. dem Beweiser von PVS widmen. Die einführenden Kapitel sind jeweils als Einstieg recht nützlich; die weiterführenden Kapitel können als Referenzhandbuch zum Nachschlagen verwendet werden.

Für Interessierte ist als Lektüre insbesondere das WIFT-Tutorial zu empfehlen: <http://www.csl.sri.com/papers/wift-tutorial/>. Dateien zu den dort aufgeführten Beispielen sind auch lokal verfügbar: <ftp://ftp.informatik.uni-ulm.de/pub/KI/pvs/examples/wift-tutorial/>.

## Start von PVS

PVS ist im Linux-Pool unter `/soft/PVS/` installiert. Da dieser Pfad nicht automatisch im Suchpfad enthalten ist, muss man ihn zuerst mit dem Kommando `use pvs` einfügen.

Am besten wechseln Sie vor dem Start in ein separates Verzeichnis. PVS wird mit `pvs` aufgerufen. Es erscheint ein Fenster mit der Willkommens-Seite von PVS. PVS verwendet als Benutzerschnittstelle den EMACS-Editor. Im Linux-Pool wird standardmäßig GNU-EMACS gestartet; wer lieber mit XEMACS arbeitet, kann PVS mit `pvs -emacs xemacs` aufrufen.

## Wichtige PVS Emacs-Kommandos

Die folgende Tabelle beschreibt einige der wichtigsten PVS EMACS-Kommandos. Dabei bedeutet z. B. `C-x` das gleichzeitige Drücken der CONTROL-Taste und `x` und `M-x` analog für die META-Taste (auf PCs ist dies meist die ALT-Taste, auf SUN-Workstations ist sie mit einer Raute `◇` beschriftet).

<code>C-c h</code>	Hilfe zu PVS
<code>M-x nf</code>	anlegen einer neuen PVS-Theorie
<code>C-c C-f</code>	öffnen einer bestehenden PVS-Datei
<code>C-x C-s</code>	speichern einer Datei
<code>C-c C-t</code>	typprüfen einer PVS-Theorie
<code>C-c p</code>	Formel, auf die der Cursor zeigt, beweisen
<code>M-x cc</code>	wechseln des Kontextes (Verzeichnis)
<code>C-x C-c</code>	PVS beenden

Ferner können viele weitere Kommandos und Hilfetexte über das PVS-Menü in der Emacs-Menüleiste erreicht werden.

## Erste Schritte mit PVS

Die Arbeitsschritte mit PVS können in folgende 4 Punkte unterteilt werden:

**1. Theorien spezifizieren** Alle Eingaben in PVS werden in *Theorien* strukturiert, welche zusammengehörige Dinge wie Deklarationen, Definitionen, Axiome und Theoreme, etc. zusammenfasst. Mit dem Befehl `M-x nf` und Angabe eines Theorie-Namens wird eine neue PVS-Theorie erstellt, welche zunächst folgende Gestalt hat:

```
propositions % [ parameters ]  
              : THEORY  
BEGIN  
  
  % ASSUMING  
  % assuming declarations  
  % ENDASSUMING  
  
END propositions
```

Kommentare werden mit % markiert. Bis auf weiteres können Sie die erzeugten Kommentarzeilen ignorieren oder löschen.

**2. Deklarationen und Definitionen** Mit Deklarationen werden neue Namen (z. B. uninterpretierte Typen oder Variablen) eingeführt, denen ein Typ zugewiesen wird. Eine Deklaration hat die Form:

`<Name> : <TypAusdruck>.`

Definitionen (z. B. von Konstanten) haben in PVS im allgemeinen die Form:

`<Name> : <TypAusdruck> = <Ausdruck>.`

Formeln haben die Form:

`<Name> : <FormelArt> <Formel>.`

Bei der `<FormelArt>` wird zwischen *Axiomen* und *zu beweisenden Formeln* unterschieden. Erstere werden als gültig vorausgesetzt und brauchen deshalb – im Unterschied zu letzteren – nicht bewiesen werden. Zu beweisende Formeln können von der Art THEOREM, LEMMA, SUBLEMMA, CHALLENGE, CLAIM, CONJECTURE, COROLLARY, FACT, FORMULA, LAW, PROPOSITION und POSTULATE sein. Allerdings dient diese Unterscheidung nur Darstellungszwecken – intern werden alle diese Formelarten von PVS gleich behandelt.

**3. Typprüfung** Alle verwendeten Ausdrücke müssen typkorrekt sein. Mit dem Kommando `M-x tc` wird eine Theorie auf Typkorrektheit überprüft; der Ausdruck `5 = true` z. B. ist nicht typkorrekt und erzeugt bei der Typprüfung eine Fehlermeldung.

**4. Beweisen** Der PVS-Beweiser wird aufgerufen, indem der Cursor auf die zu beweisende Formel gesetzt und der Befehl `C-c p` oder `M-x pr` benutzt wird. Es wird ein neuer Emacs-Buffer geöffnet, in dem der Beweiser abläuft. Auf der Kommandozeile (mit dem Prompt `Rule?`) können nun die PVS-Beweisbefehle eingegeben werden.

Zu jeder Beweisregel gibt es mit (`HELP <Regel>`) eine kurze Beschreibung. Kann ein Beweis nicht zu Ende geführt werden, so kann der Beweiser mit (`QUIT`) verlassen werden. Ferner kann man mit (`POSTPONE`) ein Beweisziel zurückstellen und zum nächsten wechseln und mit (`UNDO <n>`) die letzten `n` Beweisschritte zurücknehmen.

Ein PVS-Beweisziel ist schematisch folgendermaßen aufgebaut:

```
impl_trans :  
  
{-1}  valid(-B)  
 |-----  
[1]   valid(-A)  
[2]   valid(C)  
  
Rule?
```

Es handelt sich hierbei um eine sogenannte *Sequenz*, wobei die zu beweisenden Formeln unter dem symbolisierten Ableitungszeichen |----- stehen und mit positiven Nummern gekennzeichnet sind, während die Voraussetzungen mit negativen Nummern über dem Ableitungszeichen stehen. Formeln, die sich im letzten Beweisschritt geändert haben oder neu hinzugekommen sind, werden in geschweifte Klammern {} eingeschlossen; unveränderte Formeln stehen in eckigen Klammern []. Beendete und abgebrochene Beweise werden von PVS in Dateien mit der Endung .prf gespeichert. Dateien mit der Endung .bin dienen nur dem schnelleren Laden und können ohne Schaden gelöscht werden (z.B. bei Platzmangel).

## Syntax der PVS Spezifikationsprache (Ausschnitt)

**Theorien** Eine PVS-Theorie hat im allgemeinen folgende Gestalt:

```
examples : THEORY  
BEGIN  
  
% --- Theorie-Rumpf: enthaelt Deklarationen von  
%                   Namen, Axiomen, Theoremen, etc.  
  
END examples
```

**Deklaration und Definitionen** Mit Deklarationen werden neue Namen eingeführt, denen ein Typ zugewiesen wird. Eine Deklaration hat die Form <Name> : <Typ>. Als Basistypen stehen in PVS unter anderem Boolesche Werte `bool`, natürliche, ganze, rationale und reelle Zahlen `nat`, `int`, `rat`, `real` zur Verfügung. Ferner existiert für jeden Typ eine vordefinierte Gleichheit =.

Definitionen in PVS haben im allgemeinen die Form <Name> : <Typ> = <Ausdruck>.

**Typen** Ein paar Beispiele:

```
% --- Beispiele fuer Deklarationen:

T : TYPE                                % --- ein uninterpretierter Typ,
                                        %   darf auch leer sein
S : TYPE+                               % --- wie oben, zusaetzliche Annahme: S ist
                                        %   nicht leer
                                        % --- Beispiele fuer Defintionen:

Z : TYPE = [int -> int]                 % --- Definition eines Typ einer Funktion
binop : TYPE = [int, int -> int]        % --- Typ der binaeren Operatoren auf
                                        %   ganzen Zahlen
```

**Konstanten** sind Typen mit festem Wert.

```
% --- Beispiele fuer Definitionen:
ten : int = 10                          % --- die Konstante 10

wahr : bool= TRUE                        % --- die Konstante True
```

**Funktionen und Prädikate** Ein paar Beispiele:

```
f : [int, int -> int] % --- zweistellige Funktion auf ganzen Zahlen.

P : [nat,nat -> bool] % --- Praedikate sind bool-wertige Funktionen
Q : pred[[nat,nat]]  %   Deklarationen von P und Q sind aequivalent

abs(i:int) : nat =                          % --- Funktionsdefinition
  IF i < 0 THEN -i ELSE i ENDIF

injective?(f:[S -> T]) : bool =              % --- Praedikat auf Funktionsraum [S -> T]
  FORALL (a,b:S): f(a) = f(b) => a = b

                                        % --- aequivalente Schreibweise mit
inj? : pred[[S->T]] =                        %   Mengennotation
  {f : [S->T] | FORALL (a,b:S): f(a) = f(b) => a = b}
```

**Variablen** Mit einer Variablendeklaration `<Namen> : VAR <Typ>` können bei Definitionen Typangaben für Parameter entfallen bzw. in Formeln freie Variablen verwendet werden, welche als implizit universell quantifiziert behandelt werden.

```
i,j : VAR int

abs(i) : nat =                % --- keine Typangabe bei i noetig
  IF i < 0 THEN -i ELSE i ENDIF

f_definition : AXIOM          % --- i und j sind implizit
  f(i,j) = abs(i) + abs(j)    %      universell quantifiziert
```

**Formeln** Beispiele für die Deklaration eines Axioms und eines Theorems:

```
f_definition : AXIOM
  FORALL (i,j:int): f(i,j) = abs(i) + abs(j)

abs_nonneg : THEOREM
  FORALL (i:int): abs(i) >= 0
```

## PVS Beweisregeln

Der PVS-Beweiser ist in Lisp implementiert und die Anwendung einer Beweiserregel ist nichts anderes als der Aufruf einer Lisp-Funktion. Aus diesem Grund besitzen PVS-Beweiserregeln die typische Lisp-Syntax mit Klammern und Schlüsselwortargumenten. Die genaue Signatur einer Regel erhält man mit dem Befehl (**help** <Regel>).

Die Elemente einer Liste werden in Lisp durch runde Klammern eingeschlossen. Wenn Terme als Argumente verwendet werden, so müssen diese in *Anführungszeichen* gesetzt werden, z. B. (**INST** -1 "A").

Die meisten Beweisregeln erlauben optionale Argumente. Hat eine Regel mehrere optionale Argumente, so können diese in der angegebenen Reihenfolge übergeben werden, oder in einer beliebigen Reihenfolge, wobei ihnen dann aber Schlüsselworte der Form :<Argumentname> vorangestellt werden müssen.

Alle Beweiserregeln können mit dem Befehl **M-x x-prover-commands** in einem Tcl/Tk-Fenster betrachtet werden. Ein Klick mit der mittleren Maustaste lässt den Hilfetext des entsprechenden Befehls in einem Emacs-Buffer erscheinen.

## Aussagenlogische Regeln

**Beweisregel: (FLATTEN)**  $\frac{\cancel{A}}{\cancel{A}}$

Eliminiert disjunktive Formeln: Disjunktionen im Sukzedent und Konjunktionen im Antezedent werden in ihre Teile zerlegt, welche jeweils eine eigene Formel im neuen Beweisziel darstellen. Siehe auch FLATTEN-DISJUNCT.

(FLATTEN/\$ &REST FNUMS) :  
 Disjunctively simplifies chosen formulas. It simplifies top-level antecedent conjunctions, equivalences, and negations, and succedent disjunctions, implications, and negations from the sequent.

```
{-1} A AND B
  |-----
{1}  E OR (C => D)
Rule? (flatten)
Applying disjunctive simplification to flatten sequent, this simplifies to:
{-1} A
{-2} B
{-3} C
  |-----
{1}  E
{2}  D
```

**Beweisregel: (SPLIT)**  $\frac{\cancel{A}}{\cancel{A}}$   $\frac{\cancel{A}}{\cancel{A}}$   $\frac{\cancel{A}}{\cancel{A}}$

Spaltet konjunktive Formeln im Sukzedent bzw. disjunktive Formeln im Antezedent auf und erzeugt entsprechende Unterziele; auch anwendbar auf (Sukzedent-)Formeln der Art  $A \Leftrightarrow B$  und  $\text{IF } b \text{ THEN } T \text{ ELSE } E \text{ ENDIF}$ .

(SPLIT &OPTIONAL ((FNUM \*) DEPTH)):  
 Conjunctively splits formula FNUM. If FNUM is -, + or \*, then the first conjunctive sequent formula is chosen from the antecedent, succedent, or the entire sequent. Splitting eliminates any top-level conjunction, i.e., positive AND, IFF, or IF-THEN-ELSE, and negative OR, IMPLIES, or IF-THEN-ELSE.

beispiel :

```
{-1} A OR B
|-----
{1}  C AND D
```

Rule? (split)

Splitting conjunctions,  
this yields 2 subgoals:

beispiel.1 :

```
{-1} A
|-----
[1]  C AND D
```

Rule? (postpone)

Postponing beispiel.1.

beispiel.2 :

```
{-1} B
|-----
[1]  C AND D
```

Rule?

**Beweisregel: (IFF)**  $\equiv$

Wandelt eine Gleichung auf Booleschen Ausdrücken  $A = B$  in die entsprechende Äquivalenz  $A \Leftrightarrow B$  um.

(IFF &REST FNUMS):

Converts top level Boolean equality into an IFF.

Otherwise, propositional simplification (other than by BDDSIMP)

is not applied to such equalities.

beispiel :

```
|-----  
{1}  A = B
```

Rule? (iff)

Converting top level boolean equality into IFF form,  
Converting equality to IFF,  
this simplifies to:

beispiel :

```
|-----  
{1}  A IFF B
```

## Beweisregel: (LIFT-IF)

IF-THEN-ELSE-Ausdrücke werden auf die „oberste“ Ebene angehoben:

```
|-----  
{1}  (IF i!1 < 0 THEN -i!1 ELSE i!1 ENDIF) >= 0
```

Rule? (lift-if)

```
|-----  
{1}  IF i!1 < 0 THEN (-i!1 >= 0) ELSE (i!1 >= 0) ENDIF
```

(LIFT-IF &OPTIONAL (FNUMS (UPDATES? T))):

Lifts IF occurrences to the top of chosen formulas.

CASES, COND, and WITH applications, are treated as IF occurrences.

IF-lifting is the transformation that takes  $f(\text{IF } A \text{ THEN } b \text{ ELSE } c \text{ ENDIF})$  to  $(\text{IF } A \text{ THEN } f(b) \text{ ELSE } f(c) \text{ ENDIF})$ .

LIFT-IF only lifts the leftmost-innermost contiguous block of conditionals so it might have to be applied repeatedly to lift all the conditionals.

The UPDATE? flag controls whether update-applications are converted into IF-THEN-ELSE form and lifted.

E.g., (lift-if) : applies IF-lifting to every sequent formula.

(lift-if - :updates? nil) : lifts only antecedent IF that are not applications of updates.

## Regeln für Quantoren

### Beweisregel: (SKOLEM!) EXISTS FORALL

Die (SKOLEM!)-Regel ist auf universell quantifizierte Sukzedent-Formeln (Formeln „unter“ dem Ableitungszeichen) und auf existenziell quantifizierte Antezedent-Formeln (Formeln „über“ dem Ableitungszeichen) anwendbar. Die Quantoren werden dabei eliminiert und für die gebundenen Variablen sogenannte *Skolem-Konstanten* eingesetzt. Mit der Regel (SKOLEM!) werden Namen für Skolem-Konstanten automatisch generiert. Dabei wird an den jeweiligen Variablennamen die Nummerierung !1, !2, usw. angehängt:

```
beispiel :
  |-----
{1}  FORALL (x, y: G): (i(x) * x) * y = y

Rule? (skolem!)
Skolemizing,
this simplifies to:
beispiel :
  |-----
{1}  (i(x!1) * x!1) * y!1 = y!1

Rule?
```

Die (SKOLEM!)-Regel hat folgende Signatur:

```
Rule? (help skolem!)

(SKOLEM!/$ &OPTIONAL (FNUM *) KEEP-UNDERSCORE?) :
  Skolemizes by automatically generating skolem constants.
  When KEEP-UNDERSCORE? is T, a bound variable x_1 is replaced by
  skolem constant x_1!n rather than x!n, for some number n.
```

Hier ist die Angabe von FNUM also optional. Wird der Parameter weggelassen, wird als Defaultwert \*, d.h. alle Formeln, angenommen. Für FNUM können positive oder negative Zahlen angegeben werden, oder auch nur - (nur Formeln im Antezedent) oder + (nur Sukzedent-Formeln).

### Beweisregel: (SKOLEM!)

Wem die Art der Namensgebung von (SKOLEM!) nicht gefällt, kann mit der Regel (SKOLEM) die Namen auch von Hand vergeben. Die Namen für Skolem-Konstanten sind

im wesentlichen frei wählbar; um Konflikte zu vermeiden dürfen sie jedoch nicht bereits anderweitig benutzt sein.

```
beispiel :  
  
  |-----  
{1}  FORALL (x, y: G): (i(x) * x) * y = y  
  
Rule? (skolem 1 ("a" "b"))  
For the top quantifier in 1, we introduce Skolem constants: (a b),  
this simplifies to:  
beispiel :  
  
  |-----  
{1}  (i(a) * a) * b = b  
  
Rule?
```

Die (SKOLEM)-Regel erwartet zwei Argumente: die Nummer der Formel, die skolemisiert werden soll, sowie eine Liste von Skolemkonstanten:

```
Rule? (help skolem)  
  
(SKOLEM FNUM CONSTANTS):  
  Replaces the universally quantified variables in FNUM with new skolem  
  constants in CONSTANTS.  
Example: (skolem 1 ("A" "B"))  
See also SKOLEM!, SKOSIMP, SKOSIMP*.
```

## Beweisregel: (SKOSIMP\*)

Wendet wiederholt (SKOLEM!) und (FLATTEN) an.

```
(SKOSIMP*/$ &OPTIONAL PREDS?) :  
  Repeatedly Skolemizes (with typepreds on skolem constants when PREDS?  
  is T) and disjunctively simplifies.
```

**Beweisregeln: (INST) und (INST?)** ~~FORALL~~  
~~EXISTS~~

Instanziieren von Quantoren: anwendbar auf All-Quantoren im Antezedent und Existenz-Quantoren im Sukzedent.

```
(inst/$ fnum &rest terms) :  
  Instantiates the top quantifier in formula FNUM. See INST-CP for copying  
  quantified formula.
```

(INST) verlangt als Argumente die Nummer der Formel, die instanziiert werden soll und die Terme, die für die Variablen jeweils eingesetzt werden sollen.

```
beispiel :  
  
{-1} (FORALL x, y, z: x => y = z)  
  |-----  
{1}  A = B  
  
Rule? (inst -1 "TRUE" "A" "B")  
Instantiating the top quantifier in -1 with the terms:  
TRUE, A, B,  
Q.E.D.
```

Die Regel (INST?) versucht, automatisch geeignete Terme zu finden. Manchmal wird jedoch keine oder eine ungünstige Belegung gefunden, so dass man gezwungen ist, eine Substitution von Hand anzugeben. Dies geschieht mit Hilfe des Schlüsselwortarguments :SUBST, dem eine Liste der Form ( $\langle \text{var}_1 \rangle \langle \text{term}_1 \rangle \langle \text{var}_2 \rangle \langle \text{term}_2 \rangle \dots \langle \text{var}_n \rangle \langle \text{term}_n \rangle$ ) übergeben wird. Dabei sind  $\langle \text{var}_i \rangle$  die Variablennamen, die in der anzuwendenden Gleichung auftreten (nicht notwendigerweise alle), die dann mit den jeweiligen Termen  $\langle \text{term}_i \rangle$  belegt werden.

```
beispiel :  
  
{-1} (FORALL (i:int, j:int, k:int): i < j AND j < k IMPLIES i < k)  
[-2]  a!1 > 0  
[-3]  b!1 > a!1  
  |-----  
[1]  b!1 > 0  
  
Rule? (inst? :subst ("k" "b!1" "i" "0"))
```

```
Found substitution:
j: int gets a!1,
i: int gets 0,
k: int gets b!1,
Using template: j
Instantiating quantified variables,
this simplifies to:
beispiel :

{-1}  0 < a!1 AND a!1 < b!1 IMPLIES 0 < b!1
[-2]  a!1 > 0
[-3]  b!1 > a!1
      |-----
[1]   b!1 > 0
```

```
(INST/$ FNUM &REST TERMS) :
    Instantiates the top quantifier in formula FNUM. See INST-CP for copying
    quantified formula.

(INST?/$ &OPTIONAL (FNUMS *) SUBST (WHERE *) COPY?
    IF-MATCH POLARITY? (TCC? T)) :
    Tries to automatically instantiate a quantifier:
    FNUMS indicates which quantified formula: *,-, +, or (n1, n2,...)
    SUBST is a partial substitution for the bound variable names.
    WHERE indicates which fnums to search for a match.
    COPY? if T, the quantified formula is copied.
    IF-MATCH if all, all possible instantiations of a chosen template
        subexpression containing all the instantiable variables
        of the chosen quantified formula are found, and if
        this fails, then it tries INST? with IF-MATCH set to NIL,
        if best, the instantiation from the all case that generates
        the fewest TCCs is chosen,
        if T, the instantiation only occurs if the match succeeds,
        otherwise the given partial substitution is used.
        if first*, all possible instantiations of the
            first successful template are chosen.
    POLARITY? if T, a positively occurring template is only matched against
        negatively occurring subexpressions, and less-than
        term occurrences are matched against greater-than
        occurrences.
    TCC? if NIL only selects instantiations that do not generate any TCCs.
        The default value is T. There is no check to see if the TCCs are
        true in the given context.
```

## Entscheidungsprozeduren

### Beweisregel: (PROP)

Entscheidungsprozedur für Aussagenlogik: löst aussagenlogische Teilziele.

```
(PROP/$) :  
  A black-box rule for propositional simplification.
```

Beispiel:

```
beispiel :  
  
  |-----  
{1} (A OR B => C) => NOT A OR C  
  
Rule? (prop)  
Applying propositional simplification,  
Q.E.D.
```

### Beweisregel: (ASSERT)

Benutzt Entscheidungsprozeduren zum Lösen von aussagenlogischen und linear-arithmetischen Beweiszielen.

```
beispiel :  
  
{-1} 0 < a!1 AND a!1 < b!1 IMPLIES 0 < b!1  
[-2] a!1 > 0  
[-3] b!1 > a!1  
  |-----  
[1] b!1 > 0  
  
Rule? (assert)  
Simplifying, rewriting, and recording with decision procedures,  
Q.E.D.
```

```
(ASSERT/$ &OPTIONAL (FNUMS *) REWRITE-FLAG FLUSH? LINEAR?  
CASES-REWRITE? (TYPE-CONSTRAINTS? T)  
IGNORE-PROVER-OUTPUT?) :
```

Simplifies/rewrites/records formulas in FNUMS using decision procedures. Variant of SIMPLIFY with RECORD? and REWRITE? flags set to T. If REWRITE-FLAG is RL(LR) then only lhs(rhs) of equality is simplified. If FLUSH? is T then the current asserted facts are deleted for efficiency. If LINEAR? is T, then multiplication and division are uninterpreted. If CASES-REWRITE? is T, then the selections and else parts of a CASES expression are simplified, otherwise, they are only simplified when simplification selects a case. See also SIMPLIFY, RECORD, DO-REWRITE.

Examples:

(assert): Simplifies, rewrites, and records all formulas.

(assert -1 :rewrite-flag RL): Apply assert to formula -1 leaving RHS untouched if the formula is an equality.

(assert :flush? T :linear? T): Apply assert with fully uninterpreted nonlinear arithmetic after flushing existing decision procedure database.