Kurze Hilfe zu PVS

Allgemeines

Informationen zu PVS sind im WWW erhältlich unter: `http://pvs.csl.sri.com/`. Für Interessierte ist als Lektüre insbesondere das WIFT-Tutorial zu empfehlen: `http://www.csl.sri.com/papers/wift-tutorial/`. Die dort aufgeführten Dateien sind auch lokal verfügbar: `ftp://ftp.informatik.uni-ulm.de/pub/KI/pvs/examples/wift-tutorial/`.

PVS ist im Linux-Pool unter `~hp2/PVS2.4/` installiert. Da dieser Pfad nicht automatisch im Suchpfad enthalten ist, muss man PVS durch die vollständige Angabe dieses Pfads aufrufen: `~hp2/PVS2.4/pvs`

Am besten wechseln Sie vor dem Start in ein separates Verzeichnis. Die Frage nach dem Anlegen eine Kontextes beantworten Sie ggf. mit `yes`. Es erscheint ein EMACS-Fenster mit der Willkommens-Seite von PVS.

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).

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

Ferner können viele weitere Kommandos und Hilfetexte über das PVS-Menü in der Emacs-Menüleiste erreicht werden. Mit dem Befehl `M-x nf` und Angabe eines Theorie-Namens wird eine neue PVS-Theorie erstellt, welche zunächst folgende Gestalt hat:

```
propositions  % [ parameters ]                                          1
              : THEORY
BEGIN

  % ASSUMING
   % assuming declarations
  % ENDASSUMING

END propositions
```

Kommentare werden mit `%` markiert. Bis auf weiteres können Sie die erzeugten Kom-

mentarzeilen ignorieren oder löschen.

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. Zunächst verwenden Sie bitte die auf dem Übungsblatt angegebenen Befehle; wir werden nach und nach weitere Kommandos kennenlernen.

Zu jedem Beweiskommando gibt es mit `(HELP <kommando>)` 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 Spezifikationssprache (Ausschnitt)

Alle Eingaben in PVS werden in *Theorien* strukturiert, welche zusammengehörige Dinge wie Deklarationen, Definitionen, Axiome und Theoreme, etc. zusammenfasst. Eine PVS-Theorie hat folgende Gestalt:

```
examples : THEORY                                                    2
BEGIN

  % --- Theorie-Rumpf: enthaelt Namen, Axiome, Theoreme, etc.

END examples
```

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 `=`.

```
% --- Beispiele fuer Deklarationen:                                3

T : TYPE               % --- T ist ein uninterpretierter Typ.
S : TYPE+              % --- wie oben, jedoch ist S ein nicht-leerer Typ

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

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

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

```
% --- Beispiele fuer Definitionen:                                 4

ten : int = 10                          % --- die Konstante 10

binop : TYPE = [int, int -> int]        % --- Typ der binaeren Operatoren auf
                                        %     ganzen Zahlen
```

```
abs(i:int) : nat =                      % --- Funktionsdefinition      5
  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}
```

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.

Axiome, Lemmata, Theoreme, etc. haben die Form `<Name> : AXIOM <Formel>` bzw. `<Name> : THEOREM <Formel>`. Zur Unterscheidung der Formeln kann statt `THEOREM` auch `LEMMA`, `SUBLEMMA`, `PROPOSITION` u.a. verwendet werden. Axiome unterscheiden sich von Theoremen dahingehend, dass PVS für sie keinen Beweis erwartet.

```
f_definition : AXIOM                                                    6
  FORALL (i,j:int): f(i,j) = abs(i) + abs(j)

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

Durch Variablendeklarationen der Form `<Namen> : VAR <Typ>` können bei Definitionen Typangaben für Parameter entfallen bzw. in Formeln freie Variablen verwendet werden, welche als implizit all-quantifiziert behandelt werden.

```
i,j : VAR int                                                           7

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

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

PVS Beweiskommandos

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-Systax mit Klammern und Schlüsselwortargumenten. Die genaue Signatur einer Regel erhält man mit dem Befehl `(help <Regel>)`.

Alle Beweiserkommandos 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 Kommandos

Beweisregel: (FLATTEN)

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

```
(FLATTEN/$ &REST FNUMS) :                                               8
   Disjunctively simplifies chosen formulas.  It simplifies
   top-level antecedent conjunctions, equivalences, and negations, and
   succedent disjunctions, implications, and negations from the sequent.
```

Beweisregel: (SPLIT)

Spaltet konjunktive Formeln auf und erzeugt entsprechende Unterziele; auch anwendbar auf (Sukzedent-)Formeln der Art `A <=> B` und `IF b THEN T ELSE E ENDIF`.

```
(SPLIT &OPTIONAL ((FNUM *) DEPTH)):                                    9
   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.
```

Beweisregel: (IFF)

Wandelt eine Gleichung auf Booleschen Ausdrücken `A = B` in die entsprechende Äquivalenz `A <=> B` um.

```
(IFF &REST FNUMS):                                                     10
   Converts top level Boolean equality into an IFF.
   Otherwise, propositional simplification (other than by BDDSIMP)
   is not applied to such equalities.
```

Beweisregel: (LIFT-IF)

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

```
  |-------                                                               11
{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))):                              12
   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(IF A THEN b ELSE c ENDIF) to
   (IF A THEN f(b) ELSE f(c) 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.
```

Kommandos für Quantoren

Beweisregel: (SKOLEM!)

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:

13
```
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:

14
```
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).

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 werden.

```
beispiel :                                                              15

  |-------
{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)                                                     16

(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*.
```

Die Elemente einer Liste werden in Lisp durch runde Klammern eingeschlossen. Wenn wie hier Terme als Argumente verwendet werden, so müssen diese in *Anführungszeichen* gesetzt werden.

Die meisten Beweisregeln erlauben optionale Argumente – so z. B. (SKOLEM!), siehe oben. 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. Mehr dazu bei der Regel (REWRITE).

Beweisregel: (SKOSIMP*)

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

```
(SKOSIMP*/$ &OPTIONAL PREDS?) :                                         17
   Repeatedly Skolemizes (with typepreds on skolem constants when PREDS?
   is T) and disjunctively simplifies.
```

Beweisregeln: (INST) und (INST?)

Instanzieren von Quantoren: anwendbar auf All-Quantoren im Antezedent und Existenz-Quantoren im Sukzedent. Die zweite Form versucht, automatisch geeignete Terme zu finden. Zusätzlich kann eine partielle Instanzierung durch eine Substitutionsliste angegeben werden; diese hat die Form (<variable1> <term1> <variable2> <term2> ...) (vgl. Regel (REWRITE), Anleitung Teil 2):

```
foo :                                                                    18

{-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"))
```

```
(INST/$ FNUM &REST TERMS) :                                              19
   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 occuring 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.
```

Kommandos für Lemmata und Definitionen

Beweisregel: (LEMMA)

Führt ein Lemma in den Antezedenten ein. Optional kann eine partielle Instanzierung angegeben werden, vgl. (INST?).

```
(LEMMA NAME &OPTIONAL (SUBST)):                                          20
   Adds lemma named NAME as the first antecedent formula
   after applying the substitutions in SUBST to it.
   Example:  (LEMMA "assoc" ("x" 1 "y" 2 "z" 3)).
```

Beweisregel: (USE)

Kombiniert (LEMMA) und (INST?).

```
(USE/$ LEMMA &OPTIONAL SUBST (IF-MATCH BEST) (INSTANTIATOR INST?)) :     21
   Introduces lemma LEMMA, then does BETA and INST? (repeatedly) on
   the lemma.  The INSTANTIATOR argument may be used to specify an
   alternative to INST?.
```

Beweisregel: (BETA)

Führt β-Reduktionen durch. Anwendbar nicht nur auf LAMBDA-Ausdrücke, sondern auch auf LET und Projektionen auf Tupel (PROJ_1 etc.).

```
(BETA &OPTIONAL ((FNUMS *) REWRITE-FLAG)):                               22
    Beta-reduces chosen formulas. If REWRITE-FLAG is LR(RL), then
    left(right)-hand-side is left undisturbed for rewriting using
    REWRITE and REWRITE-LEMMA.
    Example reduction steps are:
       (LAMBDA x, y: x + y)(2, 3) to 2 + 3
       (LET x := 1, y := 2 in x + y) to 1 + 2
       b((# a:=1, b:= 2 #)) to 2
       PROJ_2(2, 3) to 3
       cons?(nil) to FALSE
       car(cons(2, nil)) to 2.
```

Beweisregeln: (EXPAND) und (EXPAND*)

Dienen zur Expansion von Definitionen. Die zu expandierenden Vorkommen des Funktionsnamens können auf einzelne Formeln (mit :FNUM) und bestimmte Stellen innerhalb dieser Formeln eingeschränkt werden. Mit der zweiten Form können alle Namen auf einmal expandiert werden.

```
(EXPAND FUNCTION-NAME &OPTIONAL                                              23
        ((FNUM *) OCCURRENCE IF-SIMPLIFIES ASSERT?)):
   Expands (and simplifies) the definition of FUNCTION-NAME at a given
   OCCURRENCE.  If no OCCURRENCE is given, then all instances of the
   definition are expanded.  The OCCURRENCE is given as a number n
   referring to the nth occurrence of the function symbol counting
   from the left, or as a list of such numbers.
   If the IF-SIMPLIFIES flag is T, then any definition expansion occurs only
   if the RHS instance simplifies (using the decision procedures).  Note that
   the  EXPAND step also applies simplification with decision procedures
   (i.e. SIMPLIFY with default options) to any sequent formulas where
   an expansion has occurred.
   ASSERT? can be either NONE (meaning no simplification),
   NIL (meaning simplify using SIMPLIFY), or T (meaning simplify using
   ASSERT).

(EXPAND*/$ &REST NAMES) :
    Expands all the given names and simplifies.
```

Beweisregeln für Termersetzung

Beweisregel: (REPLACE)

Ist die eine Antezedent-Formel von der Gestalt `l = r`, so werden mit `REPLACE` alle in `FNUMS` vorkommenden Teilterme `l` durch `r` ersetzt. Soll die Ersetzung umgekehrt durchgeführt werden, so ist die Schlüsselwortoption `:DIR RL` anzugeben.

```
(REPLACE FNUM &OPTIONAL ((FNUMS *) DIR HIDE? ACTUALS?)):                     24
   Rewrites the given formulas in FNUMS with the formula FNUM.  If
   FNUM is an antecedent equality, then it rewrites left-to-right if DIR is
   LR (the default), and right-to-left if DIR is RL.  If FNUM is not an
   antecedent equality, then any occurrence of the formula FNUM in FNUMS is
   replaced by TRUE if FNUM is an antecedent, FALSE for a succedent.  If
   HIDE? is T, then FNUM is hidden afterward.  When ACTUALS?  is T, the
   replacement is done within actuals of names in addition to the expression
   level replacements.
```

Beweisregel: (REPLACE*)

Mehrfaches `REPLACE`: alle in `FNUMS` angegebenen Gleichheiten werden für die Ersetzung benutzt.

```
(REPLACE*/$ &REST FNUMS) :                                                   25
    Apply left-to-right replacement with formulas in FNUMS.
```

Beweisregel: (REWRITE)

(REWRITE) ist die diejenige Regel, mit der in PVS Termersetzung durchgeführt werden kann. Besitzt eine Voraussetzung die Form einer Gleichheit L = R, so werden durch (REWRITE) alle im aktuellen Beweisziel auftretenden Terme L' durch R' ersetzt, wobei L' aus L bzw. R' aus R durch geeignete Substitution von Variablen hervorgeht. Als Voraussetzungen können dabei entweder Axiome oder Theoreme, aber auch Antezedent-Formeln benutzt werden.

26
```
Rule? (help rewrite)

(REWRITE/$ LEMMA-OR-FNUM &OPTIONAL (FNUMS *) SUBST (TARGET-FNUMS *) (DIR LR)
                                    (ORDER IN)) :
    Rewrites using LEMMA-OR-FNUM (lemma name or fnum) of the form
    H IMPLIES L = R by finding match L' for L and replacing L' by R' with
    subgoal proof obligations for H'.
    A lemma H IMPLIES L is treated as H IMPLIES L = TRUE, and also H can be
    empty.
FNUMS constrains where to search for a match,
SUBST takes a partial substitution and tries to find a match extending this,
TARGET-FNUMS constrains where the rewriting occurs,
DIR is left-to-right(LR) or right-to-left(RL),
ORDER is inside-out(IN) or outside-in (OUT).
```

Die Anwendungsmöglichkeiten von (REWRITE) sind vielfältig. Zunächst reicht es für unsere Zwecke aber aus, die Bedeutung einiger weniger Varianten zu kennen. In der einfachsten Form wird (REWRITE) nur der Name des anzuwendenden Axioms oder Theorems übergeben:

27
```
beispiel :

  |-------
{1}   (i(a) * a) * b = b

Rule? (rewrite "assoziativ")
Found matching substitution:
z: G gets b,
y gets a,
x gets i(a),
Rewriting using assoziativ, matching in *,
this simplifies to:
beispiel :

  |-------
{1}   i(a) * (a * b) = b

Rule?
```

Im Gegensatz zu gerichteten Termersetzungsregeln können Gleichungen natürlich auch von rechts nach links angewendet werden. Dazu dient das Schlüsselwortargument `:DIR`. Wird als Argument hier `rl` angegeben, wird die entsprechende Gleichung in umgekehrter Richtung angewendet:

```
beispiel :                                                              28

  |-------
{1}   i(a) * (a * b) = b

Rule? (rewrite "assoziativ" :dir rl)
Found matching substitution:
z: G gets b,
y gets a,
x gets i(a),
Rewriting using assoziativ, matching in *,
this simplifies to:
beispiel :

  |-------
{1}   (i(a) * a) * b = b

Rule?
```

Für gewöhnlich sucht PVS automatisch nach einer geeigneten Variablensubstitution, die die linke Seite der Gleichung mit dem Beweisziel unifiziert. 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 (<var$_1$> <term$_1$> <var$_2$> <term$_2$> ... <var$_n$> <term$_n$>) übergeben wird. Dabei sind <var$_i$> die Variablennamen, die in der anzuwendenden Gleichung auftreten (nicht notwendigerweise alle), die dann mit den jeweiligen Termen <term$_i$> belegt werden.

```
beispiel :                                                              29

  |-------
{1}   (i(a) * a) * b = b

Rule? (rewrite "assoziativ" :subst ("x" "i(a)" "y" "a"))
Found matching substitution:
z: G gets b,
x gets i(a),
y gets a,
Rewriting using assoziativ, matching in * where
  x gets i(a),
  y gets a,
this simplifies to:
beispiel :

  |-------
{1}   i(a) * (a * b) = b

Rule?
```

Beweisregel: (CASE-REPLACE)

Die (REWRITE)-Regel ersetzt grundsätzlich alle im Beweisziel vorkommenden passenden Teilterme. Dies lässt sich zwar noch durch Angabe des Schlüsselwortarguments :TARGET-FNUMS auf eine bestimmte Formel in der Sequenz einschränken. Tritt aber wie im folgenden Beispiel der zu ersetzende Teilterm – hier x!1 – mehrfach in einer Formel auf und sollen aber nur bestimmte Vorkommen ersetzt werden, hier z. B. das erste Vorkommen von x!1 durch e * x!1, so reicht (leider) (REWRITE) nicht aus.

```
rechtsneutral :                                                         30

  |-------
{1}   x!1 * e = x!1

Rule? (rewrite "linksneutral" :dir rl)
Found matching substitution:
x: G gets x!1,
Rewriting using linksneutral, matching in *,
this simplifies to:
rechtsneutral :

  |-------
{1}   e * x!1 * e = e * x!1

Rule?
```

In diesem Fall muss man sich eines kleinen Umwegs bedienen und die Beweiser-Regel CASE-REPLACE anwenden, welcher als Argument eine Gleichung der Form L = R über-

geben wird. Diese Formel wird als neue Annahme in das Beweisziel aufgenommen und gleichzeitig alle im Beweisziel auftretenden Terme `L` durch die rechte Seite `R` ersetzt. Im obigen Beispiel verwenden wir als Argument die Gleichung `x!1*e = (e*x!1)*e`, welches gerade angibt, wie wir die linke Seite im aktuellen Beweisziel verändern möchten:

```
rechtsneutral :                                                          31

  |-------
{1}   x!1 * e = x!1

Rule? (CASE-REPLACE "x!1*e = (e*x!1)*e")
Assuming and applying x!1*e = (e*x!1)*e,
this yields  2 subgoals:
rechtsneutral.1 :

{-1}  x!1 * e = (e * x!1) * e
  |-------
{1}   (e * x!1) * e = x!1

Rule?
```

Hier kann der Beweis nun wie gewohnt fortgesetzt werden. Die eingeführte zusätzliche Annahme ist in diesem Fall nicht weiter nötig; wen sie stört, der kann sie mit der Regel `(HIDE -1)` verstecken, bzw. mit `(DELETE -1)` ganz löschen.

Selbstverständlich muss die neu aufgenommene Annahme aber auch gültig sein. Um dies sicherzustellen, erzeugt die Regel `(CASE-REPLACE)` noch ein zweites Teilziel, wo wir exakt die als Argument verwendete Gleichung beweisen müssen:

```
rechtsneutral.1 :                                                        32

{-1}  x!1 * e = (e * x!1) * e
  |-------
{1}   (e * x!1) * e = x!1

Rule? (postpone)
Postponing rechtsneutral.1.

rechtsneutral.2 :

  |-------
{1}   x!1 * e = (e * x!1) * e
[2]   x!1 * e = x!1

Rule?
```

Um die Formel `{1}` zu beweisen, kann nun die Regel `REWRITE` wie ursprünglich geplant angewendet werden.

```
rechtsneutral.2 :                                                    33

  |-------
{1}   x!1 * e = (e * x!1) * e
[2]   x!1 * e = x!1

Rule? (rewrite "linksneutral")
Found matching substitution:
x: G gets x!1,
Rewriting using linksneutral, matching in *,

This completes the proof of rechtsneutral.2.

rechtsneutral.1 :

{-1}  x!1 * e = (e * x!1) * e
  |-------
{1}   (e * x!1) * e = x!1

Rule?
```

Entscheidungsprozeduren

Beweisregel: (PROP)

Entscheidungsprozedur für Aussagenlogik: löst aussagenlogische Teilziele.

```
(PROP/$) :                                                           34
    A black-box rule for propositional simplification.
```

Beweisregel: (ASSERT)

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

```
(ASSERT/$ &OPTIONAL (FNUMS *) REWRITE-FLAG FLUSH? LINEAR?                    35
                    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.
```

Subtypen

Alle in PVS definierten Funktionen müssen total sein. Eine Möglichkeit, auch partielle Funktionen (wie z. B. die Division) in PVS zu behandeln ist, die Funktion durch eine Einschränkung des Definitionsbereichs mit einem Subtyp zu „totalisieren“.

Beispiel: Subtyp der reellen Zahlen ungleich 0.

```
nzreal : NONEMPTY_TYPE = {r: real | r /= 0} CONTAINING 1                    36

/: [real, nzreal -> real]
```

Die Anwendung einer Funktion, deren Argumente auf einen Subtyp eingeschränkt wurden, führt bei der Typüberprüfung meist zu Typkorrektheitsbedingungen, siehe unten.

Prädikate und Subtypen: Für ein Prädikat `P` über einem Elementtyp `T` bezeichnet `(P)` den zugehörigen Subtyp von `T` all derjenigen Elemente, die `P` erfüllen:

```
P : pred[nat] = LAMBDA (n:nat): n >= 100000                                  37

BigNumbers : TYPE = (P)  % -- Typ der natürlichen Zahlen ab 100000
```

Beweisregeln: `(TYPEPRED)` und `(SKOSIMP* :PREDS? T)`

`(TYPEPRED)` führt die Subtypeinschränkungen für die Ausdrücke in `EXPRS` als Antezedentformeln ein.

```
(TYPEPRED &REST EXPRS):                                                    38
   Extract subtype constraints for EXPRS and add as antecedents.
   Note that subtype constraints are also automatically recorded by
   the decision procedures.
```

Dieser Befehl ist nützlich, um Typbedingungen von Skolemvariablen sichtbar zu machen:

```
  |-------                                                                39
{1}    FORALL (x: {y: nat | y > 2}): P(x)

Rule? (skosimp*)
  |-------
{1}    P(x!1)

Rule? (typepred "x!1")
{-1}    x!1 > 2
  |-------
[1]    P(x!1)
```

Um Typinformation von quantifizierten Variablen direkt bei der Skolemisierung einzuführen, kann bei `(SKOSIMP*)` die Schlüsselwortoption `:PREDS? T` benutzt werden.

Abstrakte Datentypen und Induktion

Im Gegensatz zu gewöhnlichen Deklarationen können Datentypdeklarationen auch außerhalb von Theorien auftreten *(warum?)*. In diesem Fall wird die entsprechende ADT-Theorie auch in eine eigene PVS-Datei geschrieben. Wird ein Datentyp innerhalb einer Theorie deklariert, so kann die ADT-Theorie mit `M-x ppe` (*pretty-print-expanded*) betrachtet werden.

Beweisregeln: `(INDUCT)` und `(INDUCT-AND-SIMPLIFY)`

`(INDUCT)` initiiert einen Induktionsbeweis mit der Induktionsvariablen `VAR` durch Instanzierung des zum Typ von `VAR` gehörenden Induktionsaxioms. Bei Bedarf kann letzteres auch explizit durch das Schlüsselwort `NAME` angegeben werden.

```
(INDUCT/$ VAR &OPTIONAL (FNUM 1) NAME) :                                    40
  Selects an induction scheme according to the type of VAR in FNUM and uses
  formula FNUM to formulate an induction predicate, then simplifies yielding
  base and induction cases.  The induction scheme can be explicitly supplied as
  the optional NAME argument.
  (induct "i"):
  If i has type nat and occurs outermost universally quantified in formula
  FNUM, the nat_induction scheme is instantiated with a predicate constructed
  from formula FNUM, and beta-reduced and simplified to yield base and
  induction subcases.  If i has type that is a datatype, then the induction
  scheme for that datatype is used by default.
  (induct "x" :fnum 2 :name "below_induction[N]"):
  The below_induction scheme is instantiated with an induction predicate
  constructed from fnum 2.
```

Beispiel: Induktion bei natürlichen Zahlen. Mit `(INDUCT "n"` *fnum*`)` kann Induktion über die Variable `n` geführt werden, wenn die Formel *fnum* die Form `(FORALL (n:nat) P(n))` hat. Es werden dabei die zwei Unterziele `P(0)` und `(FORALL (n:nat): P(n) IMPLIES P(n+1))` generiert.

Einfache Induktionsbeweise können oft mit der vordefinierten Strategie `(INDUCT-AND-SIMPLIFY "n")` „erledigt“ werden.

Typkorrektheitsbedingungen (TCCs)

Vor dem eigentlichen Beweisen muss jede Theorie auf Typkorrektheit hin überprüft werden. Kommandos: `M-x tc` oder `C-c C-t`. Typüberprüfung in PVS ist (wegen der erwähnten Subtypen) nicht entscheidbar. Es werden deshalb Bedingungen erzeugt, unter denen die Theorie typkorrekt ist, sogenannte *Type Correctness Conditions, TCCs*. Ein Theorem einer Theorie gilt erst dann als (vollständig) bewiesen, wenn alle TCCs der Theorie gezeigt sind.

Es gibt u.a. folgende Arten von TCCs: Subtyp-TCCs, Terminations-TCCs und Existenz-TCCs. Die Befehle `M-x show-tccs` bzw. `C-c C-q s` zeigen die für eine Theorie erzeugten TCCs in einem eigenen Buffer an. TCCs werden genau wie andere Lemmata oder Theoreme behandelt und können auf die gewohnte Art bewiesen werden.

Viele TCCs sind sehr simpel und können von PVS automatisch bewiesen werden. Der Befehl `M-x tcp` bzw. `M-x typecheck-prove` veranlasst PVS, soviele der erzeugten TCCs wie möglich mit Hilfe von Standard-Strategien zu beweisen.

Den „Status“ aller Formeln einer Theorie – also TCCs und deklarierte Lemmata und Theoreme – kann man mit Befehl `M-x spt` bzw. `M-x status-proof-theory` ermitteln.

Subtyp-TCCs

Subtyp-TCCs werden dann erzeugt, wenn ein Ausdruck vom Typ `T` an einer Stelle verwendet wird, wo ein Ausdruck von einem Subtyp `S` von `T` erwartet wird.

```
distance(x,y:int) : nat =                                                    41
  IF x > y THEN x - y ELSE y - x ENDIF
```

Der Abstand zweier ganzen Zahlen ist offensichtlich nicht-negativ, die Funktion `distance` hat also den Resulttattyp `nat`. Der Typ der Subtraktionsfunktion ist jedoch `[real -> real]` und damit sind die Ausdrücke `x-y` und `y-x` zunächst vom Typ `real` und nicht vom deklarierten Subtyp `nat`. Es werden folglich zwei entsprechende Subtyp-TCCs erzeugt:

```
% Subtype TCC generated (at line 6, column 18) for  x - y                   42
  % untried
distance_TCC1: OBLIGATION FORALL (x, y: int): x > y IMPLIES x - y >= 0;

% Subtype TCC generated (at line 6, column 29) for  y - x
  % untried
distance_TCC2: OBLIGATION FORALL (x, y: int): NOT x > y IMPLIES y - x >= 0;
```

Terminations-TCCs

Definitionen rekursiver Funktionen: Neben der eigentlichen Definition muss zum Nachweis der Terminierung noch eine `MEASURE`-Funktion angegeben werden, die die selben Parameter wie die zu definierende hat, diese in eine wohlfundierten Menge abbildet und dort strikt abnehmend ist.

Beispiel:

```
fac(n:nat): RECURSIVE nat =                                                  43
  IF n = 0 THEN 1 ELSE n * fac(n-1) ENDIF
    MEASURE (LAMBDA (n:nat): n)
```

Die Typprüfung erzeugt folgende Terminations-TCC:

```
% Termination TCC generated (at line ...) for  fac                          44
  % untried
fac_TCC1: OBLIGATION
  (FORALL (n:nat): NOT (n=0) IMPLIES n-1 < n)
```

Existenz-TCCs

Existenz-TCCs werden immer dann erzeugt, wenn Elemente eines Typs *deklariert* werden, von dem PVS nicht ohne weiteres feststellen kann, dass er überhaupt Elemente besitzt (also nicht-leer oder „bewohnt“ ist).

Beispiel: Deklaration einen beliebigen natürlichen Zahl kleiner oder gleich 5.

```
n : {x:nat | x <= 5}                                                    45
```

Die Typprüfung erzeugt folgende Existenz-TCC:

```
% Existence TCC generated (at line 8, column 2) for  n: {x: nat | x <= 5}   46
  % untried
n_TCC1: OBLIGATION EXISTS (x1: {x: nat | x <= 5}): TRUE;
```

Die Wichtigkeit solcher TCCs wird an folgendem Beispiel klar:

```
n : {x:nat | x < 0}                                                     47

quatsch : THEOREM
  FORALL (P:pred[nat]): P(n)

absoluter_unsinn : THEOREM
  TRUE = FALSE
```

Hier wird eine Konstante deklariert, die es offensichtlich nicht geben kann. Diesen Widerspruch kann man dazu ausnutzen, die beiden „Theoreme“ zu „beweisen“. Die (nicht beweisebare) Existenz-TCC verhindert, dass solche Beweise als zulässig angesehen werden.