Rechnergestützte Beweissysteme

Elementarer Umgang mit PVS

Einführung

1. PVS
   - einführende Beispiele
   - Installation
2. Installation
   - Linux/Knoppix
   - Emacs
   - Umgang mit dem PVS-System
3. Der PVS-Beweiser
   - Sequenzen
   - Elementare Beweiskommandos
     - `split, flatten, skolem, inst,`
     - `replace, prop, lemma, postpone`

Einführung

1. PVS
   - einführende Beispiele
   - Installation

2. Installation
   - Linux/Knoppix
   - Emacs
   - Umgang mit dem PVS-System

3. Der PVS-Beweiser
   - Sequenzen
   - Elementare Beweiskommandos
     - `split, flatten, skolem, inst,`
     - `replace, prop, lemma, postpone`

PVS-Prototype Verification System

- PVS besteht aus
  - einer Spezikationssprache
    - PVS language
  - einem interaktiven Beweiser
    - PVS prover
  - einer Bibliothek von existierenden
    - Spezifikationen und bewiesenen Sätzen
    - PVS prelude.


PVS@Knoppix
PVS File Edit Options Buffers Tools Help
SRI International
PVS
Welcome to the PVS Specification
and Verification System
Type C-c h for a summary of the commands.
Your current working context is
/home/knoppix/
Use M-x change-context to move to a different context.
PVS Version 3.1
Please check our website periodically for news of later versions
at http://pvs.csl.sri.com/
Allegro CL Enterprise Edition
-0:%% PVS Welcome (Text)--L1--Top

PVS-Prototype Verification System

- Mächtige Spezifikationssprache
  - Mengen, Operationen, rekursive Definitionen
- Logik höherer Stufe
  - Induktionsaxiome,
  - Quantifikation über Elemente, Mengen, Relationen, Funktionen
- Mächtiges Beweissystem
  - Sequenzenkalkül für Logik höherer Stufe
  - Viele Spezialbeweismethoden eingebaut, viele automatische Strategien
- Viele eingebaute Theorien
  - set, list, relations, functions, …
  - dazu viele nützliche Sätze und Lemmas
- Support für
  - abstrakte Datentypen, automatische Induktionsaxiome,
  - Codatentypen, Coinduktion

Die PVS Spezifikationssprache

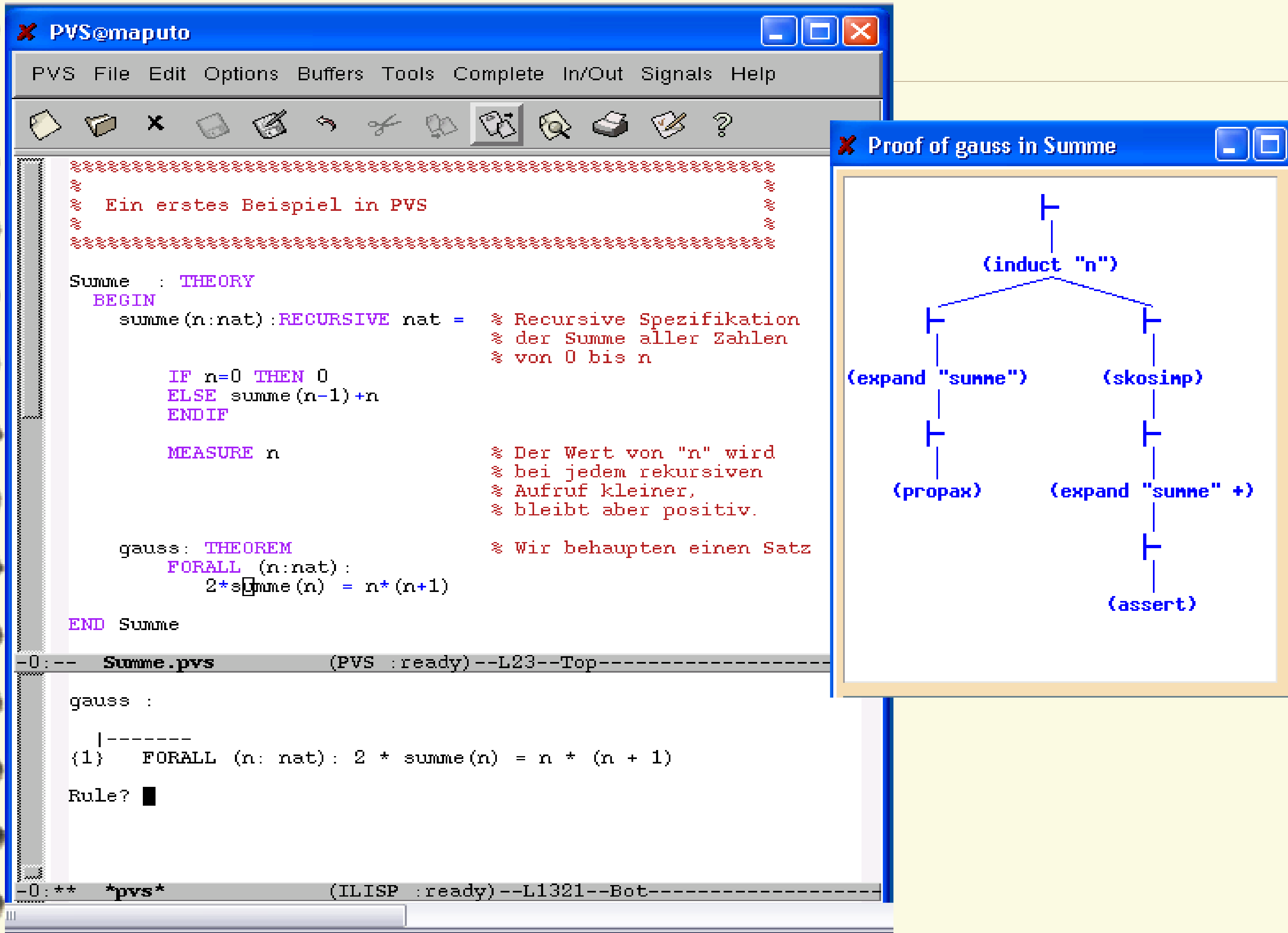
- Ein pvs-file beschreibt eine mathematische Theorie mit
  - Mengen (TYPE)
  - Operationen, Konstanten
  - Definitionen
  - Axiomen
  - Lemmas und Theoremen

- Viele Theorien und darin bewiesene Sätze sind schon vorhanden
  - sie werden als *prelude* geladen
    - *set* Mengenlehre
    - *list[T]* Theorie der Listen
    - *nat* Natürliche Zahlen
    - ...

Einführende Beispiele

- Mathematische Formeln: $1^k + 2^k + \ldots + n^k = \ldots$
  - **summe.pvs**
- Sortieralgorithmen
  - **Insertionsort.pvs**
- Algebra
  - **Gruppentheorie.pvs**
- Korrekte Implementierung von Datenstrukturen
  - **Stacks.pvs**
- Spezifikation: Telefonbuch
  - **Telefonbuch.pvs**

Der Satz von Gauss mit Beweis

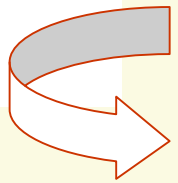


Beispiel: NFAs in PVS

```
nfa[ Zeichen : TYPE                          % Zeichen sei eine beliebige Menge
     Zustand : TYPE
     start   : Zustand                       % start ein beliebiger Zustand
     tau     : Zustand,Zeichen -> set[Zustand]  % eine nondet. Funktion
     final?  : set[Zustand]                  % eine Menge von Zuständen
   ]: THEORY

BEGIN
   Wort : TYPE = list[Zeichen]
   tauStar(z : Zustand, w: Wort):RECURSIVE set[Zustand] =
      CASES w OF
         null     : singleton(z),
        cons(a,v): {s:Zustand|EXISTS(s1:(tauStar(z,v))):tau(s1,a)(s)}
      ENDCASES
    MEASURE length(w)

   nAccept(w: Wort) : bool =
     EXISTS ( z : (final?)):  member(z,tauStar(start,w))

END nfa
```

Behauptung: L(DFA) = L(NFA)

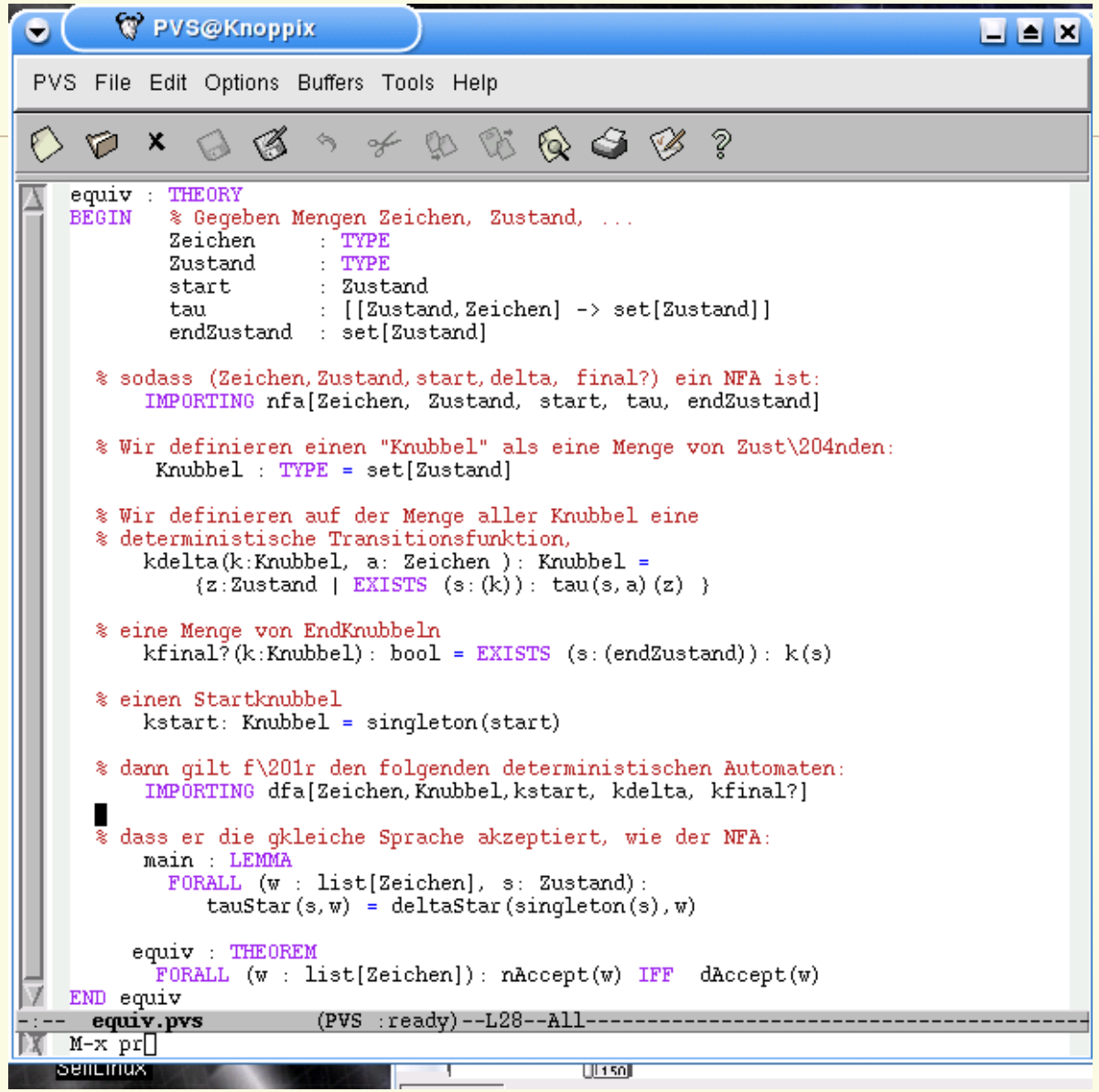
```
equiv : THEORY
BEGIN   % Gegeben Mengen Zeichen, Zustand, ...
        Zeichen      : TYPE
        Zustand      : TYPE
        start        : Zustand
        tau          : [[Zustand,Zeichen] -> set[Zustand]]
        endZustand   : set[Zustand]

  % sodass (Zeichen,Zustand,start,delta, final?) ein NFA ist:
      IMPORTING nfa[Zeichen, Zustand, start, tau, endZustand]

  % Wir definieren einen "Knubbel" als eine Menge von Zuständen:
       Knubbel : TYPE = set[Zustand]

  % Wir definieren auf der Menge aller Knubbel eine
  % deterministische Transitionsfunktion,
      kdelta(k:Knubbel, a: Zeichen ): Knubbel =
          {z:Zustand | EXISTS (s:(k)): tau(s,a)(z) }
```



Behauptung L(DFA) = L(NFA)

```
% einen Startknubbel
      kstart: Knubbel = singleton(start)

% eine Menge von EndKnubbeln
      kfinal?(k:Knubbel): bool = EXISTS (s:(endZustand)): k(s)


% dann gilt für den folgenden deterministischen Automaten:
      IMPORTING dfa[Zeichen,Knubbel,kstart, kdelta, kfinal?]

% dass er die gkleiche Sprache akzeptiert, wie der NFA:
      main : LEMMA
        FORALL (w : list[Zeichen], s: Zustand):
           tauStar(s,w) = deltaStar(singleton(s),w)

     equiv : THEOREM
       FORALL (w : list[Zeichen]):
           endZustand(start,w) IFF  kfinal?(kstart,w)

END equiv
```

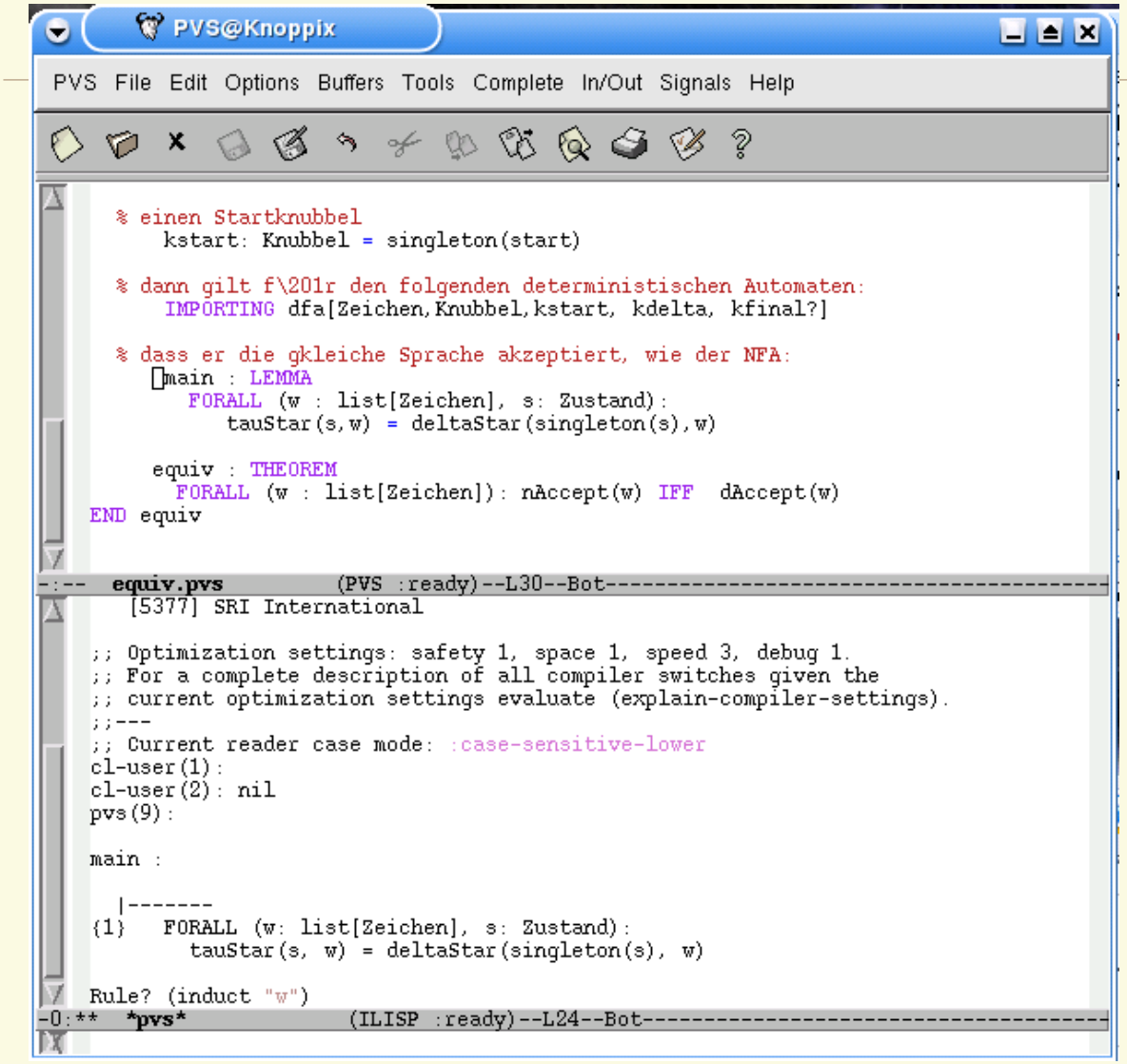
Beginn eines Beweises



Cursor auf die zu beweisende Behauptung setzen und

M-x pr

Erster Beweisschritt



Emacs öffnet ein neues Fenster für die Interaktion mit dem Beweiser.

Wir starten einen Induktionsbeweis Listeninduktion über w:

(induct "w")

Induktionsanfang und -schritt

4 Der (induct) Befehl hat das Induktionsschema für Listen aufgerufen

```
-:--  equiv.pvs          (PVS :ready)--L30--Bot----------------------------------
{1}   FORALL (w: list[Zeichen], s: Zustand):
        tauStar(s, w) = deltaStar(singleton(s), w)

Rule? (induct "w")
Inducting on w on formula 1,
this yields  2 subgoals:
main.1 :

  |-------
{1}   FORALL (s: Zustand): tauStar(s, null) = deltaStar(singleton(s), null)

Rule? (postpone)
Postponing main.1.

main.2 :

  |-------
{1}   FORALL (cons1_var: Zeichen, cons2_var: list[Zeichen]):
        (FORALL (s: Zustand):
           tauStar(s, cons2_var) = deltaStar(singleton(s), cons2_var))
         IMPLIES
         (FORALL (s: Zustand):
            tauStar(s, cons(cons1_var, cons2_var)) =
             deltaStar(singleton(s), cons(cons1_var, cons2_var)))

Rule? (skolem!)
-0:**  *pvs*              (ILISP :ready)--L46--Bot----------------------------
```

Ind.anfang
w=null

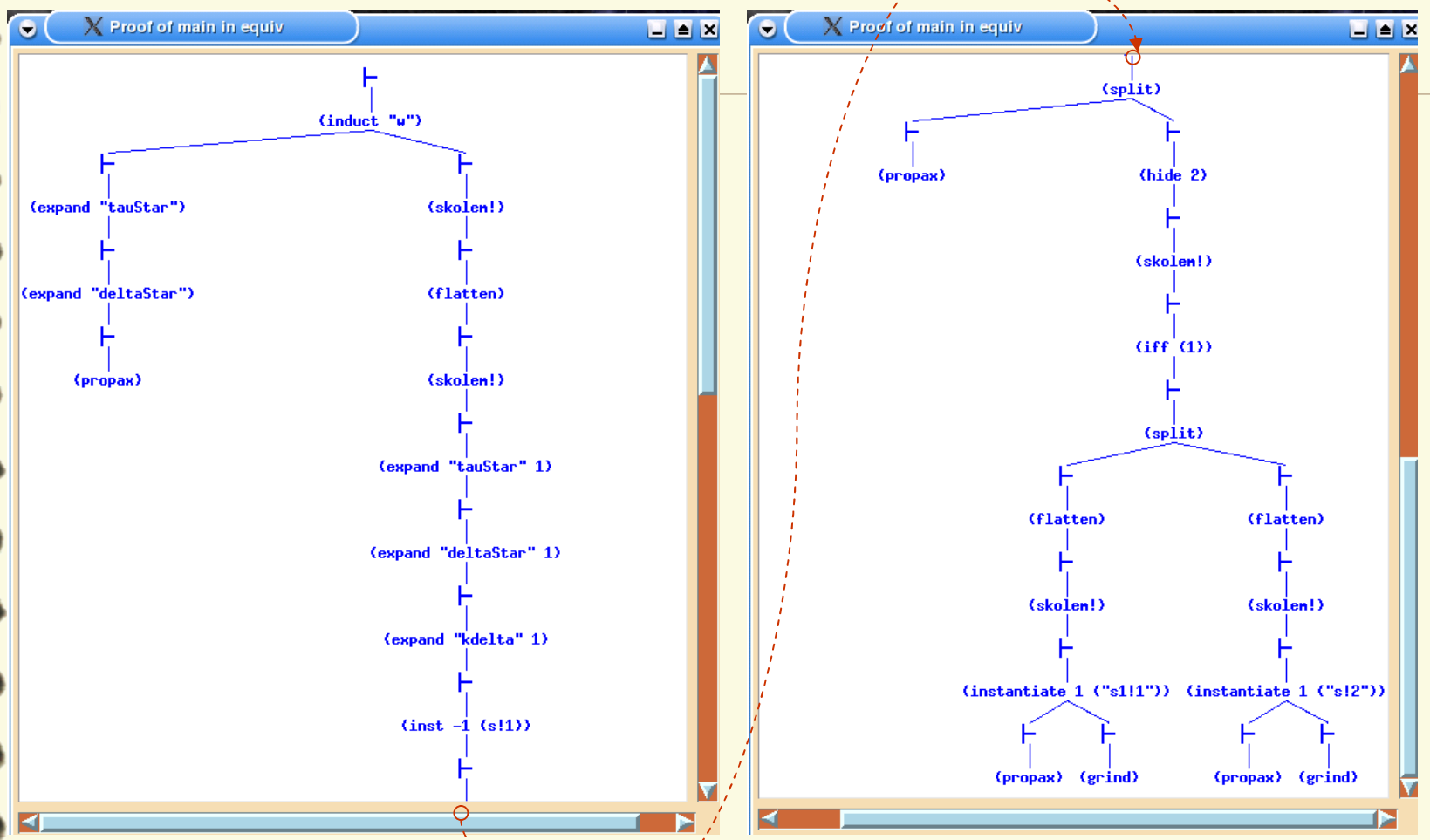
Zurückstellen
(postpone)

Ind.schritt

w=cons(cons1_var,cons2_var)

Als nächster Befehl :
(skolem!)

Baumdarstellung des kompletten Beweises



Der Befehl **`M-x xpr`** stellt den Beweis auch als Baum dar.

Fallstudien in PVS

- Verifikation von Mikroprozessoren
  - pipelining
  - IEEE floating point spezifikation
  - schnelle floating point division (SRT)
- Verifikation von Koordinationsalgorithmen
  - Byzantinische Algorithmen
  - wechselseitiger Ausschluss (bakery algorithm)
- Mathematik
  - Satz von Schröder-Bernstein
  - Ramsey Satz
- Algorithmen
  - Boyer-Moore
  - Abstrakte Programm-Transformationen .... etc.

Alternativen zu PVS

- Coq, Isabelle, HOL
  - saubere Implementierung
  - überschaubarer Kern
  - fast beliebige Logiken kodierbar
  - mit Taktiken erweiterbar
  - Beweisobjekt verfügbar

  aber

  - Beweise relativ low level

- PVS
  - parametrisierte Typen
  - Subtypen, dependent types
  - Strukturierung(Theorien, Module)
  - Mächtige Entscheidungsalgorithmen
  - Sehr viel läuft vollautomatisch
  - praktisch gut einsetzbar
  - viele Fallstudien
  - aber
  - Beweis nicht als Objekt verfügbar
  - hat (noch) bugs
  - Implementierung undokumentiert

Einführung

1. PVS
   - einführende Beispiele
   - Installation

2. Installation
   - Linux/Knoppix
   - Emacs
   - Umgang mit dem PVS-System

3. Der PVS-Beweiser
   - Sequenzen
   - Elementare Beweiskommandos
     - `split, flatten, skolem, inst,`
     - `replace, prop, lemma, postpone`

Verfügbarkeit von PVS

- Plattform: Solaris/Linux
- User interface: emacs
- Erhältlich von: SRI international (http://www.sri.com)
- Preis: kostenlos
- Dokumentation: (WIFT-Tutorial) Crow, Owre, Rushby, Shankar, Srivas: *A tutorial Introduction to PVS* www.csl.sri.com/papers/wift-tutorial

- Für die Vorlesung sollte dies ausreichen.
- Weitere Dokumente sind im Semesterapparat und über die Vorlesungsseite erreichbar.

Knoppix – a taste of Linux

- Linux auf einer CD
  - braucht nicht installiert werden
  - von CD booten – Fertig
  - kann NTFS Partitionen lesen
  - kann FAT32 Partitionen lesen und schreiben
- CD als ISO-Image kostenlos verfügbar:
  - www.knoppix.de
- HomeVerzeichnis normalerweise auf Ram-Disk
  - Kann aber permanent gemacht werden
    - auf FAT32 Partition
    - auf ZIP-Diskette
    - auf USB-Speicherstift

Installation von PVS

- Linux-Verzeichnis (z.B.: **PVS**) anlegen
  - **md PVS**
- Herunterladen der folgenden Dateien:
  - **pvs-3.2-linux.tar,**
  - **pvs-3.2-system.tar,**
  - **pvs-3.2-libraries.tar**
- Entpacken mit
  - **tar –xvf pvs-3.2-*.tar**
- Pfad anpassen
  - **./bin/relocate**
- PVS starten mit
  - **pvs**

Die Benutzeroberfläche emacs

- Programmierbarer Texteditor mit Shell unter Unix/Linux
  - Bedient mehrere „buffer" gleichzeitig
    - Buffer entspricht dem Windows „Fenster"
  - ein Buffer kann enthalten
    - eine Datei – zum Editieren
    - eine Dateiauswahl - um eine Datei daraus zu laden
    - ein Befehlsfenster - oft nur eine Zeile
- Befehle beginnen meist mit Ctrl- oder Alt/ESC Sequenzen.
  - Auf traditionellen Unix Keyboards gab es
    - Control (Ctr) und Meta (M)-Key
  - Auf dem PC-Keyboard:
    - Ctr = Strg
    - M = ESC oder Alt

Grundlegende emacs-Befehle

- Generell
  - Ctr-x d — zeige directory
  - Ctr-x f — lade file
  - Ctr-x s — schreibe file
  - Ctr-g — reset/escape von einem Fehler im Befehlsfenster
- In einem Editorfenster
  - Ctr-<Space> — setze Marke. Definiert Bereich von Marke bis Cursor
  - Ctr-w — auschneiden
  - M-w — kopieren
  - Ctr-y — einfügen
  - Cursortasten, <BS>, etc. funktionieren wie gewohnt
- **`xemacs`** stellt die meisten Befehle über die Menüleiste zur Verfügung

Wichtige Emacs Befehle für PVS

- M-x cc — change Context
  (setze Arbeitsverzeichnis)
- M-x tc — typecheck
  (Syntax- und Typprüfung)
- M-x pr — prove
  (starte Beweis der Aussage unter dem Cursor)
- M-p — previous (zeige letztes Kommando nochmal)

Benutzung von PVS

- PVS starten
  - **pvs** ↵ *startet pvs in einer emacs shell*
- Kontext setzen
  - **M-x cc** ↵
  - System fragt nach gewünschtem Pfad:
  - **./PVS/Examples** ↵
- Datei laden
  - **Ctr-x f** ↵
  - System fragt nach gewünschte Datei:
  - **sum.pvs** ↵
- Neue PVS-Datei erstellen
  - **Ctr-x nf** ↵
- Wenn Sie sich vertippt haben
  - **Ctr-g** ↵

Einführung

1. PVS
   - einführende Beispiele
   - Installation

2. Installation
   - Linux/Knoppix
   - Emacs
   - Umgang mit dem PVS-System

3. Der PVS-Beweiser
   - Sequenzen
   - Elementare Beweiskommandos
     - `split, flatten, skolem, inst,`
     - `replace, prop, lemma, postpone`

Beginn eines Beweises

- Prüfen, ob Datei syntaktische oder Typfehler hat
  - **M-x tc ↵**
- Beweis einer Behautung (LEMMA, THEOREM, ..)
  - Cursor irgendwo in die Behauptung setzen
  - **M-x pr ↵**
  - ein neues Fenster öffnet sich mit der zu beweisenden Behauptung und einem prompt.
  - das System erwartet jetzt Beweis-Kommandos
- Im Beispiel von sum.pvs führen zwei Kommandos zum Ziel – einem erfolgreichen Beweis:
  - **(induct "w")**
  - **(assert)**
- Damit ist dieser Beweis schon erfolgreich.
  - Im Allgemeinen geht es nicht ganz so automatisch.

Dokumentation von PVS

- Auf Deutsch:
  - Kurze Hilfe zu PVS
  - www.informatik.uni-ulm.de/ki/Edu/Vorlesungen/Inferenzsysteme/WS0203/pvs-hilfe.pdf
- Auf Englisch:
  - A Tutorial Introduction to PVS
    - das sollten Sie auf jeden Fall durchlesen
    - enthält im Anhang eine kompakte Referenz
      - (auf dem Stand von Version 2.0!)
  - PVS Quick Reference
    - die wichtigsten Kommandos kompakt
- Zum Nachschlagen – zu umfangreich zum ausdrucken!
  - PVS Systems Guide
    - enthält im Anhang eine kurze „Introduction to Emacs"
  - PVS Prover Guide
  - PVS Language Reference
- Alles unter http://pvs.csl.sri.com/documentation.shtml

Elementare PVS-Beweisbefehle

- Sequenzen
- flatten, split, skolem
- Aussagenlogik,
- Prädikatenlogik,
- Logik höherer Stufe
- Vollautomatisches Beweisen ist i.A. unmöglich

Sequenzen

Darstellungsweise von Beweisaufgaben:

$$H_1, H_2, \ldots, H_n \vdash G_1, G_2, \ldots, G_k$$

Dabei sind $H_1, \ldots, H_n$ sowie $G_1, \ldots, G_k$ logische Formeln.
Die Folge $H_1, \ldots, H_n$ heißt *„Antezedenz"*, die Folge $G_1, \ldots, G_k$ *Sukkzedenz.*

Die Beweisaufgabe besteht darin, aus der
Konjunktion der *Hypothesen* $H_1, \ldots, H_n$
die **Disjunktion** der *Zielformeln (goal formulae)* $G_1, \ldots, G_k$
zu beweisen.

Die **Semantik** (Bedeutung) der obigen Sequenz ist also:

$$H_1 \wedge H_2 \wedge \ldots \wedge H_n \rightarrow G_1 \vee G_2 \vee \ldots \vee G_k$$

Sequenzen in PVS

- Sequenzen werden in PVS folgendermaßen dargestellt

```
  [-1]   H1
  [-2]   H2
     ...
  [-n]   Hn
|--------
   [1]   G1
     ...
   [k]   Gk
```

- Durch die vorangestellten Nummern -1,...,n, 1,...,k können sich Beweisbefehle auf einzelne Formeln der Sequenz beziehen.
- Mit „+" bezieht man sich auf alle Formeln im Sukkzedens, mit „-" auf alle Formeln im Antezedenz.

Beweisschritte

- Jeder Beweisschritt dient dazu, eine Sequenz auf einfachere Sequenzen zurückzuführen. Dabei kann die Sequenz vereinfacht werden **(flatten)**, sie kann aber auch durch mehrere neue - einfachere Sequenzen ersetzt werden **(split)**.

- Ist die Sequenz einfach genug, so daß PVS ihre Gültigkeit sofort erkennt, so wird sie als bewiesen gemeldet.

- Wenn alle Sequenzen bewiesen sind, ist die ursprüngliche Aussage gezeigt.

Flatten

$\ldots, p \wedge q \vdash r, \ldots$ **flatten** $\ldots, p, q \vdash r, \ldots$

$\ldots, p \vdash r \vee s, \ldots$ **flatten** $\ldots, p \vdash r, s, \ldots$

$\ldots, p \vdash r \rightarrow s, \ldots$ **flatten** $\ldots, p, r \vdash s, \ldots$

`Flatten` vereinfacht ***geeignete*** Sequenzen, wobei ein logischer Operator verschwindet.
(Die „ ... " stehen immer für Bestandteile einer Sequenz, die sich nicht verändern.)

Flatten in Aktion

```
left_cancellation :
  |------
{1} FORALL (a,b,c:G):
       a o b = a o c IMPLIES b = c

Rule? (skolem!)
Skolemizing, this simplifies to:
left_cancellation :

  |------
{1}    a!1 o b!1 = a!1 o c!1
          IMPLIES b!1 = c!1

Rule? (flatten)

Applying disjunctive simplification
to flatten sequent, this simplifies to:
left_cancellation :

{-1}   a!1 o b!1 = a!1 o c!1
  |------
{1}    b!1 = c!1
```

Wir akzeptieren einstweilen:

Mit `skolem!` entfernen wir einen *Allquantor* im Sukzedenz.

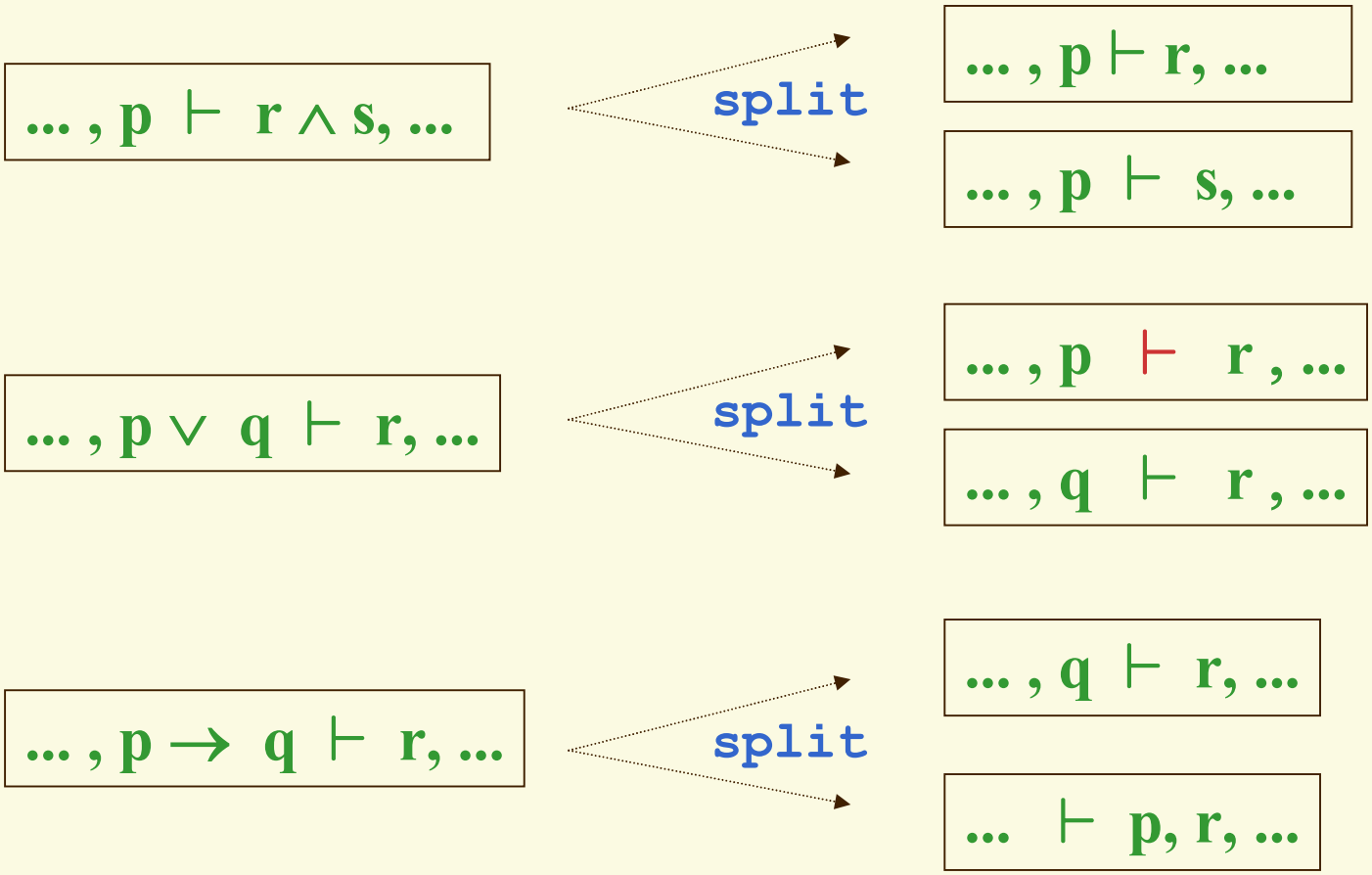
"Seien `a!1, b!1,` *und* `c!1` *fest aber beliebig ...."*

Bei einer Implikation im Sukzedens schiebt `flatten` die Prämisse in den Antezedens ...

"Wir nehmen also `a!1 o b!1 = a!1 o c!1` *an und zeigen* `b!1 = c!1`*"*

Split

- `Split` zerlegt Sequenzen in mehrere einfache



Split in Aktion

```
{-1}  j!1 >= 8 IMPLIES (EXISTS (d: nat, f: nat):
                          j!1 = d * 3 + f * 5)
{-2}  j!1 + 1 >= 8
  |-------
{1}   EXISTS (d: nat, f: nat):
                 j!1 + 1 = d * 3 + f * 5
Rule? (split)
Splitting conjunctions,this yields  2 subgoals:
three_five.2.1 :
{-1}  EXISTS (d: nat, f: nat):
                j!1 = d * 3 + f * 5
[-2]  j!1 + 1 >= 8
  |-------
[1]   EXISTS (d: nat, f: nat):
        j!1 + 1 = d * 3 + f * 5
Rule? (postpone)
Postponing three_five.2.1.
three_five.2.2 :
[-1]  j!1 + 1 >= 8
  |-------
{1}   j!1 >= 8
[2]   EXISTS (d: nat, f: nat):
         j!1 + 1 = d * 3 + f * 5
```

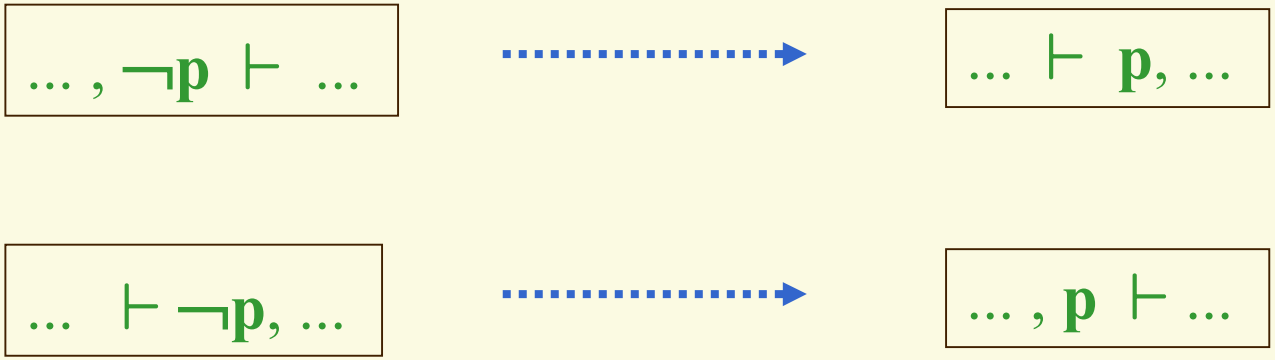
Ein „→“ in einer Hypothese kann gesplittet werden

Der erste Sequent, der entstanden ist wird zur nächsten Beweisaufgabe.

Man kann die Aufgabe verschieben (engl. „to **postpone**“ und zuerst die folgende erledigen

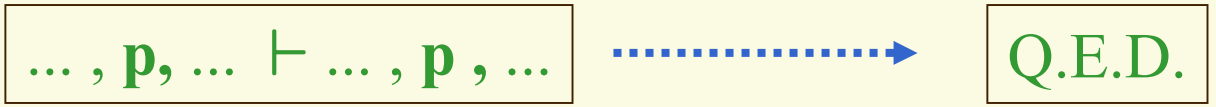
Negation

- Die Entfernung von Negationen geschieht bei PVS automatisch.



Axiome

- Eine Sequenz ist offensichtlich richtig, falls eines der Beweisziele in den Hypothesen auftaucht. Dies wird von PVS immer automatisch überprüft und erkannt.



Distributivität – Teil des Beweises

```
dist :
  |-------
{1} A!1 AND (B!1 OR C!1) IMPLIES
     (A!1 AND B!1) OR (A!1 AND C!1)
Rule? (flatten)
Applying disjunctive simplification
to flatten sequent, this simplifies to:
dist :
{-1}  A!1
{-2}  (B!1 OR C!1)
  |-------
{1}   (A!1 AND B!1)
{2}   (A!1 AND C!1)
Rule? (split -2)
Splitting conjunctions,
this yields 2 subgoals:
dist.1 :
{-1}  B!1
[-2]  A!1
  |-------
[1]   (A!1 AND B!1)
[2]   (A!1 AND C!1)
```

Eine Sequenz ohne Hypothese: Eine Richtung eines der Distributivgesetze

Hier wurde **flatten** automatisch mehrfach ausgeführt. *Wie oft ?*

Hier wurde die zweite Hypothese gesplittet . Es entstehen zwei neue Sequenzen.
Wie sieht dist.2 *aus ?*

Distributivität – Teil des Beweises

```
{-1}  B!1
[-2]  A!1
  |-------
[1]   (A!1 AND B!1)
[2]   (A!1 AND C!1)
Rule? (hide 2)
Hiding formulas: 2, simplifies to:
dist.1 :
[-1]  B!1
[-2]  A!1
  |-------
[1]   (A!1 AND B!1)
Rule? (split)
Splitting conjunctions, This
yields 2 subgoals:
dist.1.1 :
[-1]  B!1
[-2]  A!1
  |-------
{1}   A!1
which is trivially true. This
completes the proof of dist.1.1.
```

Offensichtlich ist ein Beweis des ersten Goals möglich. Das zweite ist überflüssig und wird versteckt.

Ein letztes Aufsplitten

Offensichtlich gültig, da ein Goal, A!1, auch im Antezedens vorkommt.

Wie viele weitere Sequenzen sind noch zu beweisen ?

Geht es auch vollautomatisch ?

- Ist eine Sequenz *Instanz* einer *aussagenlogischen Tautologie*, so kann PVS sie mit dem Befehl
  (prop)
  automatisch beweisen.

Beispiel:

..., $((A \to B) \to A) \vdash A$, ...
ist eine *tautologische Sequenz*.

Eine *Instanz* davon ist z.B
..., $(x < 5 \to \exists w.w+x = 3) \to x < 5), \ldots \vdash x < 5, \ldots$
Statt mit den Befehlen
`(split)` und `(flatten)`
kann man sie auch auf einen Schlag mit
`(prop)`
beweisen.

Wieviele Anwendungen von **`split`** sind in diesem Beispiel nötig?

```
¬(  (A11 ∨ A12)
    ∧ ( A21 ∨ A22 )
    ∧ ( A31 ∨ A32 )
    ∧ ¬( A11 ∧ A21 )
    ∧ ¬( A11 ∧ A31 )
    ∧ ¬( A21 ∧ A31 )
    ∧ ¬( A12 ∧ A22 )
    ∧ ¬( A12 ∧ A32 )
    ∧ ¬( A22 ∧ A32 ))
```

Axiome und (Hilfs)sätze anwenden

- Axiome, Theoreme und Lemmata kann man benutzen, indem man sie in die Hypothese aufnimmt:

```
(lemma "Hilfssatz")
```

```
zirkulaer :
{-1}  a <= b
{-2}  b <= c
{-3}  c <= a
  |-------
{1}   a = b AND b = c

Rule? (lemma "transitiv")
Applying transitiv this simplifies to:

zirkulaer :
{-1}  FORALL (x, y, z: t): x <= y AND y <= z IMPLIES x <= z
[-2]  a <= b
[-3]  b <= c
[-4]  c <= a
  |-------
[1]   a = b AND b = c
```

Gezielte Befehlsanwendung

- `(split k)` - wende *split* auf k-te Zielformel an
- `(split -k)` - wende *split* auf k-te Hypothese an
- `(split +)` - *splitte* alle Formeln im Sukkzedenz
- `(split -)` - analog auf alle Formeln im Antezedenz
- `(split)` - versuche *split* überall anzuwenden
- `(split (k1 k2 ... kn))` - wende *split* auf die angegebenen Formeln an

Analoges gilt für die meisten PVS Befehle

Beispiele: `(split 5)`, `(flatten -)`, `(split (2 -3))`, `(flatten (1 3 4 -1))`, `(flatten)`.

Skolemisieren

- Um einen Allquantor $\forall x.P(x)$ zu beweisen, zeigt man P(c) für eine *„feste aber beliebige Konstante c"*.

```
(skolem n ("c1" ... "cn"))
```

n : Nummer der Formel,
"c_1" ... "c_n" : Namen für die Konstanten

```
 zirkulaer :
 |------
{1}   FORALL (x, y, z: t):x <= y AND y <= z AND z <= x IMPLIES x = y
AND y = z

Rule? (skolem 1 ("a" "b" "c"))
For the top quantifier in 1, we introduce Skolem constants: (a b c),
this simplifies to:

zirkulaer :
  |-----
{1}   a <= b AND b <= c AND c <= a IMPLIES a = b AND b = c
```

Instanziieren

- Einen All-Quantor in der Prämisse kann man mit beliebigen Werten des passenden Typs instanziieren:

  (inst n "t_1" ... "t_n")

- Man beachte hier die fehlenden Klammern um die Terme t_1 ... t_n !!

```
zirkulaer :
{-1}  FORALL (x, y, z: t): x <= y AND y <= z IMPLIES x <= z
[-2]  a <= b
[-3]  b <= c
[-4]  c <= a
  |-------
[1]   a = b AND b = c

Rule? (inst -1 "b" "c" "a")
Instantiating top quantifier in -1 with terms: b,c,a, this simplifies to:

zirkulaer :
{-1}  b <= c AND c <= a IMPLIES b <= a
[-2]  a <= b
[-3]  b <= c
[-4]  c <= a
  |-------
[1]   a = b AND b = c
```

Gleichheiten einsetzen

- Eine Gleichung x=t in der Hypothese erlaubt es, gezielt oder überall „x" durch „t" zu ersetzen. Der Befehl lautet
- **(replace n wo)**

```
zirkulaer.1.1.2 :
[-1]  a = c
{-2}  c <= c
{-3}  c <= b
[-4]  b <= c
  |-------
{1}   c = b
Rule? (replace -1 (-2 1) :dir RL)

Replacing using formula -1,this
simplifies to:
zirkulaer.1.1.2 :
[-1]  a = c
{-2}  a <= a
[-3]  c <= b
[-4]  b <= c
  |-------
{1}   a = b
```

n : Nummer der Formel x=t
wo : Nummer oder Nummern der Formel(n) in denen ersetzt werden soll

Optionale Argumente können in LISP selektiv angegeben werden:

Hier hat das optionale Argument **:dir** den Default-Wert LR
Als Werte sind hier zugelassen:
RL oder LR.