Rechnergestützte Beweissysteme

Prädikatenlogische Spezifikationen

Prädikatenlogik

1. Formeln der Prädikatenlogik
   - offene Formeln, Aussagen
2. Signaturen
   - Sorten und Operationen
   - Beispiele: bool. nat, natStack, Gruppen
   - parametrisierte Signaturen
   - Interpretationen
3. Terme
   - Variablen, Terme,
   - Belegungen, Auswertungen
   - Prädikate
4. Syntax der Prädikatenlogik
   - Ausdrücke, Aussagen
   - Belegungen
   - Erfüllbarkeit, Gültigkeit
5. Spezifikationen
   - PVS-Spezifikation
   - PVS-Beweisaufgabe

Prädikatenlogik

1. Formeln der Prädikatenlogik
   - offene Formeln, Aussagen
2. Signaturen
   - Sorten und Operationen
   - Beispiele: bool. nat, natStack, Gruppen
   - parametrisierte Signaturen
   - Interpretationen
3. Terme
   - Variablen, Terme,
   - Belegungen, Auswertungen
   - Prädikate
4. Syntax der Prädikatenlogik
   - Ausdrücke, Aussagen
   - Belegungen
   - Erfüllbarkeit, Gültigkeit
5. Spezifikationen
   - PVS-Spezifikation
   - PVS-Beweisaufgabe

Prädikatenlogik

- Die Aussagenlogik lehrt uns nur, wie man elementare logische Aussagen mittels $\wedge$, $\vee$, $\neg$, und $\rightarrow$ verknüpft.
  - Die Natur der elementaren Aussagen wird nicht berücksichtigt
  - Was zählt ist, dass sie wahr oder falsch sein können

- In der Prädikatenlogik können wir auch über mathematische Strukturen reden mit
  - Relationen ($\leq$, $<$, $=$, $\in$, ...)
  - Operationen (+, * , succ, sin, ..)
  - Konstanten (0, 1, π, ...)
  - Individuenvariablen (x,y,z, ...)

- Elementare (oder atomare) Ausdrücke sind dann z.B.
  - $x \leq 5$, $x*x + 1 = x-y$, teilt(3,z), $6 \in$ Divisors(z*40)

Offene Formeln

- Durch die Verknüpfung atomarer Ausdrücke mit den aussagelogischen Operatoren erhält man offene Ausdrücke (engl.: open formulas)
  - $x \leq 5 \ \vee \ x*x+1=x-y$,
  - teilt(3,z) $\rightarrow$ $6 \in$ Divisors(z*40)
- Der Wahrheitswert eines offenen Ausdrucks hängt von den Werten der Variablen ab
  - $x \leq 5 \ \vee \ x*x+1=x-y$ ist
    - wahr für x=10 und y = -111
    - falsch für x=11 und y=10
  - teilt(3,z) $\rightarrow$ $6 \in$ Divisors(z*40) ist
    - wahr für alle $z \in$ nat.
- Werden alle Variablen eines offenen Ausdrucks durch Quantoren gebunden, so wird daraus eine Aussage
  - eine solche ist immer wahr oder falsch
    - $\forall(x : \text{int}).\ \exists(y:\text{int}).\ x \leq 5 \ \vee \ x*x+1=x-y$ ist wahr
    - $\forall(x : \text{int}).\ \forall(y:\text{int}).\ x \leq 5 \ \vee \ x*x+1=x-y$ ist falsch

Prädikatenlogik

1. Formeln der Prädikatenlogik
   - offene Formeln, Aussagen
2. Signaturen
   - Sorten und Operationen
   - Beispiele: bool. nat, natStack, Gruppen
   - parametrisierte Signaturen
   - Interpretationen
3. Terme
   - Variablen, Terme,
   - Belegungen, Auswertungen
   - Prädikate
4. Syntax der Prädikatenlogik
   - Ausdrücke, Aussagen
   - Belegungen
   - Erfüllbarkeit, Gültigkeit
5. Spezifikationen
   - PVS-Spezifikation
   - PVS-Beweisaufgabe

Operationen und Relationen

- Eine *n-stellige* ***Operation*** f auf einer Menge A ist eine Abbildung

  $f : A^n \to A$.

- In der Praxis sind die meisten Operationen 2-stellig oder 1-stellig. Eine *0-stellige Operation* ist eine ***Konstante***.

- Eine *k-stellige* ***Relation*** (oder Prädikat) ist eine Teilmenge

  $R \subseteq A^k$.

- Beispiele: Auf den natürlichen Zahlen sind **+**, **succ**, **div**, **mod** Operationen der Stelligkeiten 2, 1, 2, 2 und die Konstante **0** ist Operation der Stelligkeit 0. Relationen sind z.B. $=$, $<$, $\leq$. Eine Relation der Stelligkeit 1 ist einfach eine Teilmenge.

Mehrere Sorten

- In der Praxis hat man auch Operationen und Relationen zwischen verschiedenen Mengen.
  - Beispiele: für mehrsortige Operationen:
    - δ : $S \times \Sigma \rightarrow S$ % bei Automaten
    - length: String $\rightarrow$ int % in Java
    - $\leq$: int $\times$ int $\rightarrow$ bool
  - Beispiel für mehrsortige Relationen
    - hörtBei $\subseteq$ Student $\times$ Professor
    - endZustand $\subseteq S$ % einstellige Relation
    - $\in \subseteq X \times \mathbb{P}(X)$ % für jede Menge X

Prädikatenlogische Signatur

- Eine prädikatenlogische *Signatur* Σ besteht aus
- einer Familie von Typnamen $\{\tau_1, \tau_2, \ldots\}$
  - Vordefiniert sind meist ***bool***, ***nat***, ***real***, etc ..
  - der Benutzer kann weitere definieren
- einer Familie von Funktionsnamen

$$f_i : \tau_{i1} \times \tau_{i2} \ldots \times \tau_{in} \rightarrow \tau_i$$

Dabei ist f_i der Funktionsname, $\tau_{i1}, \tau_{i2}, \ldots, \tau_{in}$ sind die Typen der Argumente und τ_i ist der Ergebnistyp. $\tau_{i1} \times \tau_{i2} \ldots \times \tau_{in} \rightarrow \tau_i$ heißt auch die Signatur von f_i.

- Einer Familie von Relationsnamen
- $$R_j :: \tau_{j1} \times \tau_{j2} \ldots \times \tau_{jn}$$

$\tau_{j1} \times \tau_{j2} \ldots \times \tau_{jn}$ heißt die Signatur von R_j.

Konventionen

- Für n=0 hat man $f : \rightarrow \tau$.
  - Es handelt sich um einen Funktionssymbol für eine Funktion ohne Argumente, also um ein *Konstantensymbol*.
  - Wir schreiben dann kurz
    - $f : \tau$
- Wir gehen davon aus, dass immer vorhanden sind:
  - ein Typ **bool**
  - für jeden Typ τ ein Relationssymbol
  - $=_\tau :: \tau \times \tau$
  - Statt "$=_\tau$"schreibt man abgekürzt nur "=" .

- Es ist Aufgabe des Lesers, herauszufinden welches "$=_\tau$" gemeint ist, wenn er ein "=" liest.
- Funktions*symbol* und Funktions*name* sind das gleiche, ebenso Relations*symbol* und Relations*name*.

Die Signatur bool

Die Signatur bool hat

- Typnamen
  - bool
- Funktionsnamen
  - true, false : bool
  - OR, AND, IFF, IMPLIES : bool × bool → bool
- Relationssymbol
  - = :: bool × bool

Die Signatur der natürlichen Zahlen

Die Signatur der natürlichen Zahlen hat

- Typnamen:
  - nat
  - bool
- Funktionsnamen
  - $+, * : nat \times nat \rightarrow nat$
  - succ: $nat \rightarrow nat$
  - $0 \quad : \quad nat$
- Relationsnamen
  - $<, >, \leq, \geq : nat \times nat$
  - $= :: nat \times nat$

natStack

Die Signatur eines natStack besteht aus

- den Typnamen
  - natStack, nat, bool
- den Operationen
  - push : nat × natStack → natStack
  - empty : natStack
- den Relationen
  - emptyStack? : Stack
  - isTop : nat × natStack
  - isRest : natStack × natStack
- sowie den Operationen und Relationen von nat und von bool.
- Wo sind die Operationen *top* und *pop* geblieben ?
- In PVS sind partielle Operationen nicht erlaubt. Man kann entweder
  - *top* und *pop* aus *isTop*, *isRest* rekonstruieren, oder
  - einen zusätzlichen Typ natStack+ einführen wie auf der folgenden Folie ...

natStack

- In PVS sind partielle Operationen nicht erlaubt. Man führt daher einen neuen Typ natStack+ als Teiltyp von natStack ein, und Operationen
  - push : nat × natStack → natStack+
  - top : natStack+ → nat
  - rest : natStack+ → natStack
  - empty : natStack
- Relationen
  - isEmpty :: natStack
  - $=_{natStack}$ :: natStack × natStack
- sowie die Funktions- und Relationsnamen von nat und bool.

Parametrisierte Signaturen

- Parametrisierte Signaturen haben einige Typnamen durch Typ-Variablen ersetzt

- Beispiel ist stack[X] mit den Operationen
  - push : $X \times stack[X] \rightarrow stack[X]^+$
  - top : $stack[X]^+ \rightarrow X$
  - rest : $stack[X]^+ \rightarrow stack[X]$
  - empty : stack[X]

- Relationen
  - isEmpty :: stack[X]
  - $=_{natStack}$ :: $stack[X] \times stack[X]$

Aus der Mathematik: Gruppen

- Die Signatur gruppe[G] hat
- Typnamen
  - G,
  - bool
- Funktionsnamen
  - * : $G \times G \to G$
  - inv : $G \to G$
  - e : G
- Relationsnamen
  - $=_G :: G \times G$

Vektorräume über Körper K

- Die Signatur eines Vektorraums V über dem Körper K
  - hat Typnamen
    - V, K, **bool**
  - Operationen
    $+_K, -_K : K \times K \to K$
    $0_K : K$
    $+_V, -_V : V \times V \to V$
    $0_K : V$
    $\bullet_K : K \times V \to V$
  - alle Operationen und Relationen von bool, sowie
  - die Gleichheitsrelationen
    $=_K :: K \times K$
    $=_V :: V \times V$

Interpretation einer Signatur

- Eine *Interpretation* $\mathfrak{I}=((S_k)_{k\in K},(f_i)_{i\in I},(R_j)_{j\in J})$ einer Signatur Σ besteht aus
  - je einer (konkreten) Menge S_k für jeden Typ τ_k ,
  - je einer Abbildung $f_i^{\mathfrak{I}} : S_1 \times \ldots \times S_n \to S_{n+1}$ für jedes Funktionssymbol $f_i : \tau_1 \times \ldots \times \tau_n \to \tau_{n+1}$
  - je einer Relation $R_j^{\mathfrak{I}} \subseteq S_1 \times \ldots \times S_n$ für jedes Relationssymbol $R_j :: \tau_1 \times \ldots \times \tau_n$
- Eine Interpretation der Signatur einer Gruppe, (Stack, Halbordnung, …) ist noch keine Gruppe, (Stack, Halbordnung, …)
  - Wir benötigen noch Axiome … dazu später

Viele Interpretationen

- Interpretationen sind durch die Signatur noch nicht festgelegt.
- Im Falle der Signatur nat haben wir u.a.
  - die Standardinterpretation mit
    - nat:= $\mathbb{N}$ = {0,1,2,3,4,5,...}
    - succ(x) = x+1, etc.
  - eine Nichtstandard-Interpretation mit
    - nat = $\{(x,y) \in \mathbb{Z} \times \mathbb{N} \mid y = 0 \Rightarrow x \geq 0\}$
    - succ(x,y) := (x+1,y)
    - (x,y)+(u,v) := (x+u,max(y,v)),
    - $(x,y) < (u,v) :\Leftrightarrow (x < u \wedge y=v) \vee y < v$
  - eine triviale Interpretation mit
    - nat = {0}
    - succ(0)=0
    - 0+0=0,
    - etc...

$\mathbb{Z}$: ganze Zahlen { ...,-2,-1,0,1,2,...}

$\mathbb{N}$: natürliche Zahlen {0,1,2,...}

Prädikatenlogik

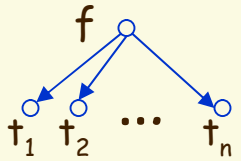
1. Formeln der Prädikatenlogik
   - offene Formeln, Aussagen
2. Signaturen
   - Sorten und Operationen
   - Beispiele: bool. nat, natStack, Gruppen
   - parametrisierte Signaturen
   - Interpretationen
3. Terme
   - Variablen, Terme,
   - Belegungen, Auswertungen
   - Prädikate
4. Syntax der Prädikatenlogik
   - Ausdrücke, Aussagen
   - Belegungen
   - Erfüllbarkeit, Gültigkeit
5. Spezifikationen
   - PVS-Spezifikation
   - PVS-Beweisaufgabe

Terme zu einer Signatur

- Gegeben
  - eine Signatur S
  - zu jeder Sorte τ von S eine Menge V_τ von Variablen
- Wir definieren induktiv Terme vom Typ τ

1. Jedes $v \in V_\tau$ ist ein Term vom Typ τ
2. Ist $f : \tau_1 \times \tau_2 \ldots \times \tau_n \to \tau$ ein Operationszeichen und sind
   - $t_1, .., t_n$ Terme der Typen $\tau_1, \tau_2, \ldots, \tau_n$, dann ist
   - $f(t_1, .. , t_n)$ ein Term vom Typ τ.

Terme sind als syntaktische Gebilde. Man könnte sie 2-dimensional als Bäume auffassen, oder eindimensional als Strings. Dann braucht man allerdings Klammern.



Terme - Beispiele

- Sind m,n,k Variablen vom Typ nat, dann sind Terme vom Typ nat:
  - succ(m) + n*k
  - succ(0+m)*(k+succ(0))
- Sind s_1 und s_2 Variablen vom Typ stack[X]+ und a,b Variablen vom Typ X, dann sind Terme
  - vom Typ stack[X]+ :
    - s_1, push(a,s_1), push(a,push(b,push(a,s_2)))
  - vom Typ stack[X] :
    - jeder Term vom Typ stack[X]+
    - pop(s_1), empty
- pop(pop(s_1)) ist nur dann ein legaler Term, falls gezeigt werden kann, dass pop(s_1) sogar vom Typ stack[X]+ ist !!!

Variablenbelegungen

- Sei Σ eine Signatur
  - Sei $\mathfrak{I}=((S_k)_{k\in K},(f_i)_{i\in I},(R_j)_{j\in J})$ die Interpretation einer Signatur Σ.
  - Eine Variablenbelegung ρ ist eine typgerechte Zuordnung von Variablen zu Werten, d.h.
    - einer Variablen v vom Typ τ wird ein Wert aus S_τ zugeordnet:

      $\rho: V_\tau \rightarrow S_\tau$ für jedes τ

Termauswertung

- Gegeben sei eine Interpretation $\mathfrak{I}$ und eine Belegung ρ der Variablen.
  Wir definieren den Wert $[\![t]\!]_{\mathfrak{I},\rho}$ eines Terms :
  - $[\![x]\!]_{\mathfrak{I},\rho} := \rho(x)$
  - $[\![f_i(t_1,\ldots,t_n)]\!]_{\mathfrak{I},\rho} := f_i^{\mathfrak{I}}([\![t_1]\!]_{\mathfrak{I},\rho}, \ldots, [\![t_n]\!]_{\mathfrak{I},\rho})$
- Man ersetzt also die Variablen durch ihre ρ-Werte und *rechnet den Term* in der gewählten Interpretation *aus*.
  - Beispiel: Im vorigen Beispiel mit der Standardinterpretation von nat und der Belegung ρ mit $\rho(m) = 2$, $\rho(k) = 1$, $\rho(n)=17$ gilt:
    - $[\![\text{succ}(0+m)*(k+\text{succ}(0))]\!] = [\![\text{succ}(0+m)]\!] * [\![k+\text{succ}(0)]\!]$
- $= \text{succ}([\![0+m]\!])*[\![k]\!] + [\![\text{succ}(0)]\!]$
- $= \ldots =$
- $= \text{succ}(0+2)*(1+1) = 6$

Atomare Ausdrücke

- Sei Σ eine Signatur mit
  - Typnamen
  - Funktionszeichen und
  - Relationszeichen $R_k :: \tau_{k1} \times \ldots \times \tau_{kn}$
- Jeder Ausdruck der Form
  - $R_k(t_1,\ldots,t_n)$ wobei t_i ein Term vom Typ τ_{ki} ist, heißt atomarer Ausdruck.
  - Englische Bezeichnung: atomic formula

Bedeutung atomarer Ausdrücke

- Atomare Ausdrücke sind die einfachsten Bestandteile der Prädikatenlogik. Ein atomarer Ausdruck
- $R(t_1,\ldots, t_n)$
- besagt, dass die durch die Terme $t_1, \ldots, t_n$ bezeichneten Elemente in der Relation R stehen.
- Ist ρ eine Variablenbelegung, dann sagen wir
  - $R(t_1,\ldots, t_n)$ ist wahr unter der Belegung ρ, falls

$$([\![t_1]\!]_{\mathfrak{I}\rho}, \ldots, [\![t_1]\!]_{\mathfrak{I},\rho}) \in R^{\mathfrak{I}}$$

Wir schreiben dann auch:

$$\mathfrak{I},\rho \models R(t_1,\ldots, t_n)$$

Prädikatenlogik

1. Formeln der Prädikatenlogik
   - offene Formeln, Aussagen
2. Signaturen
   - Sorten und Operationen
   - Beispiele: bool. nat, natStack, Gruppen
   - parametrisierte Signaturen
   - Interpretationen
3. Terme
   - Variablen, Terme,
   - Belegungen, Auswertungen
   - Prädikate
4. Syntax der Prädikatenlogik
   - Ausdrücke, Aussagen
   - Belegungen
   - Erfüllbarkeit, Gültigkeit
5. Spezifikationen
   - PVS-Spezifikation
   - PVS-Beweisaufgabe

Syntax der Prädikatenlogik

$$
\begin{aligned}
\text{form} \;::=\; & R(t_1, \ldots, t_n) \\
\mid\; & \text{form} \wedge \text{form} \\
\mid\; & \text{form} \vee \text{form} \\
\mid\; & \text{form} \Rightarrow \text{form} \\
\mid\; & \text{form} \Leftrightarrow \text{form} \\
\mid\; & \neg\, \text{form} \\
\mid\; & \forall (x{:}\tau) : \text{form} \\
\mid\; & \exists (x{:}\tau) : \text{form}
\end{aligned}
$$

atomarer Ausdruck

offene Formeln

falls x Variable vom Typ τ

Syntax in PVS

form ::= $R(t_1,\ldots, t_n)$

| form **AND** form
| form **OR** form
| form **IMPLIES** form
| form **IFF** form
| **NOT** form

| **FORALL** $(x_1,\ldots,x_n:\tau)$: form
| **EXISTS** $(x_1,\ldots,x_n:\tau)$: form

atomarer Ausdruck

offene Formeln

falls $x_1,\ldots,x_n$ Variablen vom Typ τ

Modifikation einer Belegung

- Gegeben eine Belegung ρ ordnet jeder Variablen x aus V_τ ein Element aus S_τ zu.
- Zu einer Variablen x aus V_τ und einem Element $a \in S_\tau$ definieren wir eine neue Belegung $\rho[x:=a]$. Dies ist die Abbildung die für alle $v \in V_\tau$ durch

$$\rho[x:=a]\,(v) \;:=\; \text{IF } v \equiv x \text{ THEN } a \text{ ELSE } \rho(v)$$

- definiert ist.
- Andere Notation:
  - $\rho + [x \mapsto a]$

Wahrheitswert eines Ausdrucks

- Der Wahrheitswert eines Ausdrucks wird definiert:
  - $[\![R_i(t_1,\ldots,t_m)]\!]_{\mathfrak{I},\rho} := R_i^{\mathfrak{A}} ([\![t_1]\!]_{\mathfrak{I},\rho}, \ldots , [\![t_m]\!]_{\mathfrak{I},\rho})$
  - $[\![E_1 \bullet E_2]\!]_{\mathfrak{I},\rho} := [\![E_1]\!]_{\mathfrak{I},\rho} \bullet [\![E_2]\!]_{\mathfrak{I},\rho}$ für $\bullet \in \{\wedge,\vee,\rightarrow,\leftrightarrow\}$
  - $[\![\neg E]\!]_{\mathfrak{I},\rho} := \neg [\![E]\!]_{\mathfrak{I},\rho}$
  - $[\![\exists x{:}\tau.\ E]\!]_{\mathfrak{I},\rho} =$ { true, falls es ein $a \in S_\tau$ gibt mit $[\![E]\!]_{\mathfrak{I},\rho[x:=a]}$; false, sonst
  - $[\![\forall x{:}\tau.\ E]\!]_{\mathfrak{I},\rho} =$ { true, falls für jedes $a \in S_\tau$ gilt $[\![E]\!]_{\mathfrak{I},\rho[x:=a]}$; false, sonst

Konkrete Berechnung

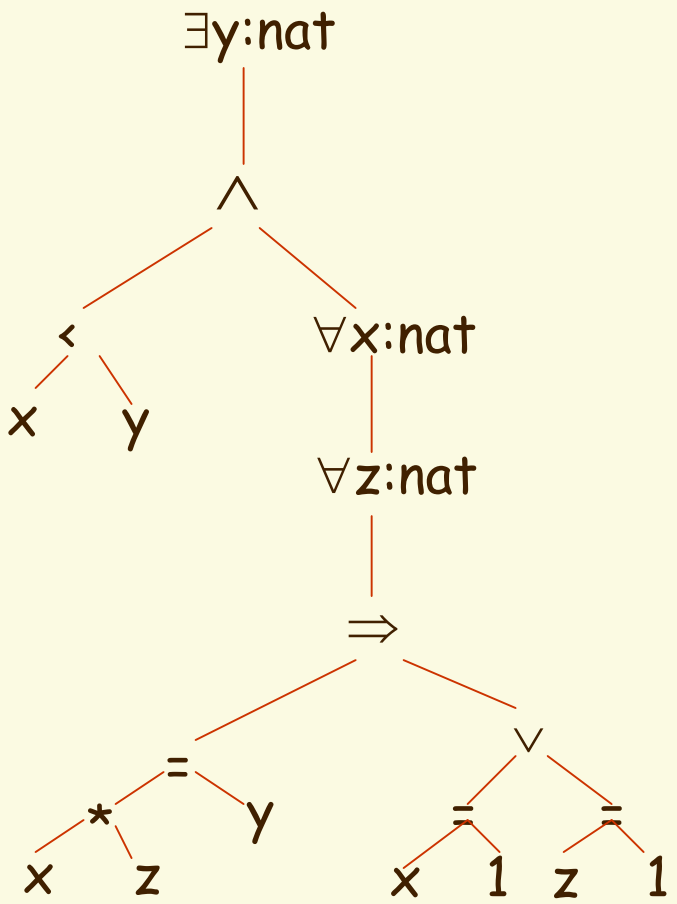
- Signatur: $\Sigma = ((P,S), (+ : P \times P \to P, \quad e : \to P, \quad s : P \to S), (< :: P))$
- Interpretationen
  - $S_n = ($ $(S_n, \{-1,1\})$
  - $(+(f,g) = gof, \quad e = id_n, \quad s(f) = sign(f))$
  - $(< (f,g) := \#\{fix(f) < \#fix(g))$
  - nat = ((nat, nat)
    (+(x,y) := x+y, e := 0, s(x) := x+1)
    (<(x,y) := x < y)
- Wir berechnen den Wert der Formel $\phi := \forall p \in P.\ s(+(p,p)) = s(e)$
- In der ersten Interpretation:
  - Sei σ eine beliebige Belegung.
    Es gilt $[\![s(+(p,p))]\!]_\sigma = sign([\![+(p,p)]\!]_\sigma) = sign([\![p]\!]_\sigma \circ [\![p]\!]_\sigma) = 1$
    und $[\![s(e)]\!]_\sigma = sign([\![e]\!]_\sigma) = sign(id_n) = 1$
  - Folglich gilt die Formel
- In der zweiten Interpretation
  - Wähle die Belegung $\sigma(p) = 1$. Es folgt
  - $[\![s(+(p,p))]\!]_\sigma = [\![+(p,p)]\!]_\sigma + 1 = [\![p]\!]_\sigma + [\![p]\!]_\sigma + 1 = 1 + 1 + 1 = 3$
  - $[\![s(e)]\!]_\sigma = [\![e]\!]_\sigma + 1 = 0 + 1 = 1$
    Folglich ist die Formel falsch.

Freie/gebundene Variablen

- Intuitiv:
  - Eine freie Variable ist eine solche, die nicht durch einen Quantor gebunden ist.
  - Vorsicht:
    - Eine Variable kann sowohl frei als auch gebunden vorkommen

Freie/gebundene Vorkommen

- Formal:
  - Repräsentiere Formeln durch ihren Syntaxbaum
  - Blätter sind Variablen oder Konstanten
  - Knoten sind
    - Funktionszeichen
    - Relationszeichen
    - logische Konnektoren
    - Quantoren mit ihren Variablen
  - Variable v ist frei, gdw. auf dem Weg zur Wurzel befindet sich kein Quantor der Form $\forall v$ bzw. $\exists v$.
  - Ist v nicht frei, so wird sie durch den auf dem Weg zur Wurzel zuerst angetroffenen Quantor $\forall v$ bzw. $\exists v$ gebunden.



Aussagen

- Gegeben eine Signatur
  - Eine Aussage ist eine Formel ohne freie Variablen
  - In jeder Interpretation $\mathfrak{I}$ ist eine Aussage wahr oder falsch
  - Da eine Aussage keine freien Variablen hat, ist die Wahrheit oder Falschheit nicht von einer Variablenbelegung ρ abhängig

Beispiel: In der Signatur nat ist die Aussage

$$\forall x{:}nat.\ \neg succ(x)=x$$

- wahr in der Standardinterpretation
- falsch in der trivialen Interpretation

Erfüllbarkeit - Gültigkeit

- Sei eine feste Interpretation $\mathfrak{I}$ vorausgesetzt.
  - Ein logischer Ausdruck *E* heißt erfüllbar, wenn es eine Belegung ρ der Variablen gibt, so dass $[\![E]\!]_{I,\rho}$ = true ist.
  - *E* heißt gültig, falls es unter jeder Belegung ρ wahr ist.
  - E heißt widersprüchlich, falls er unter keiner Belegung wahr wird.
- Seien $x_1, \ldots, x_n$ die einzigen freien Variablen in *E*. Dann gilt:
  - E erfüllbar gdw. $\exists x_1{:}\tau_1. \ \ldots \ \exists x_n{:}\tau_n.$ E gültig
  - E widersprüchlich gdw. $\forall x_1{:}\tau_1. \ \ldots \ \forall x_n{:}\tau_n. \neg$ E gültig.
- Beispiel:

  $\exists x{:}\text{nat}.\ x = y*x \wedge x < y$

  ist erfüllbar (wähle ρ mit $\rho(y) \neq 0$), aber nicht gültig.

Allgemeingültigkeit

- Ein logischer Ausdruck heißt allgemeingültig, falls er unter jeder Interpretation $\mathfrak{I}$ gültig ist.

Beispiele:

$\exists\, y{:}\sigma.\ \forall x{:}\tau.\ P(x,y) \rightarrow \forall x{:}\tau.\ \exists y{:}\sigma.\ P(x,y)$

$\forall\, x{:}\tau.\ P(x) \rightarrow \exists\, y{:}\tau.\ P(y)$

$(\forall\, x{:}\tau_1.\ R(x,f(x))) \rightarrow\ \forall\, x{:}\tau_1.\ \exists\, y{:}\tau_2.\ R(x,y)$

$(\forall\, x{:}\tau_1.\ \forall\, y{:}\tau_2.\ P(x) \vee Q(y)) \rightarrow (\forall\, x{:}\tau_1.\ P(x)) \vee (\forall\, y{:}\tau_2.\ Q(y))$

Falls τ nicht leer ist:

$\exists\, x{:}\tau.\ (\ P(x) \rightarrow \forall y{:}\tau.\ P(y)\ )$ (Trinker Paradoxon)

$(Q \text{ OR } \exists\, x{:}\tau_1.P(x)) \rightarrow \exists\, x{:}\tau_1.\ Q \vee P(x)$

Ein Satz der Logik in PVS

```
PredLog   : THEORY
 BEGIN

   S,T : TYPE
   P:[S,T -> boolean]

   test:THEOREM
      (EXISTS (y:T):
         FORALL (x:S):
                  P(x,y) )
     IMPLIES
       FORALL (x:S):
        EXISTS (y:T):
                  P(x,y)
  END PredLog
```

P sei Relation zwischen beliebigen Mengen S, T.

Dann gilt :
$(\exists y{:}T.\forall x{:}S.\ P(x,y))$
$\Rightarrow \forall x{:}S.\exists y{:}T.P(x,y)$

Quantoren haben geringste Bindungs-stärke, sie erstrecken sich also so weit nach rechts wie möglich.
Daher ist hier die Klammer notwendig

Prädikatenlogik

1. Formeln der Prädikatenlogik
   - offene Formeln, Aussagen
2. Signaturen
   - Sorten und Operationen
   - Beispiele: bool. nat, natStack, Gruppen
   - parametrisierte Signaturen
   - Interpretationen
3. Terme
   - Variablen, Terme,
   - Belegungen, Auswertungen
   - Prädikate
4. Syntax der Prädikatenlogik
   - Ausdrücke, Aussagen
   - Belegungen
   - Erfüllbarkeit, Gültigkeit
5. Spezifikationen
   - PVS-Spezifikation
   - PVS-Beweisaufgabe

Spezifikationen

- Eine Spezifikation erster Stufe besteht aus
  - einer Signatur
    - Typen
    - Funktionen und Konstanten
  - einer Folge von Aussagen der Prädikatenlogik zu dieser Signatur
    - Assumptions (Annnahmen)
    - Axiome

- Spezifikationen können
    - Typen
    - Funktionen
    - Konstante
  - als Parameter haben

- Ein Modell einer Spezifikation ist
  - eine Interpretation, in der alle Assumptions wahr sind
  -

PVS-Spezifikationen

- In PVS heißen Spezifikationen: **`Theory`**
  - **`G : Type`** heißt: *G* sei eine Sorte/Menge/Typ
  - **`G : Type+`** heißt: *G* sei Sorte mit dem Axiom
    `EXISTS (g:G) TRUE`
    m.a.W.: *G* ist nichtleerer Typ

- Eine Konstante zu einem Typ erklärt man wie folgt
  - **`e : G`** **`% e ist Konstantes Element`**

- Eine Funktion `f`: `A` → `B` ist ein Element des Raumes `[A -> B]` aller Funktionen von `A` nach `B` , konkret:
  - **`inv:[G -> G]`** **`% inv sei Abbildung von G nach G`**
  - **`o :[ G,G -> G ]`** **`% o sei Abbildung von G×G nach G`**

Axiome

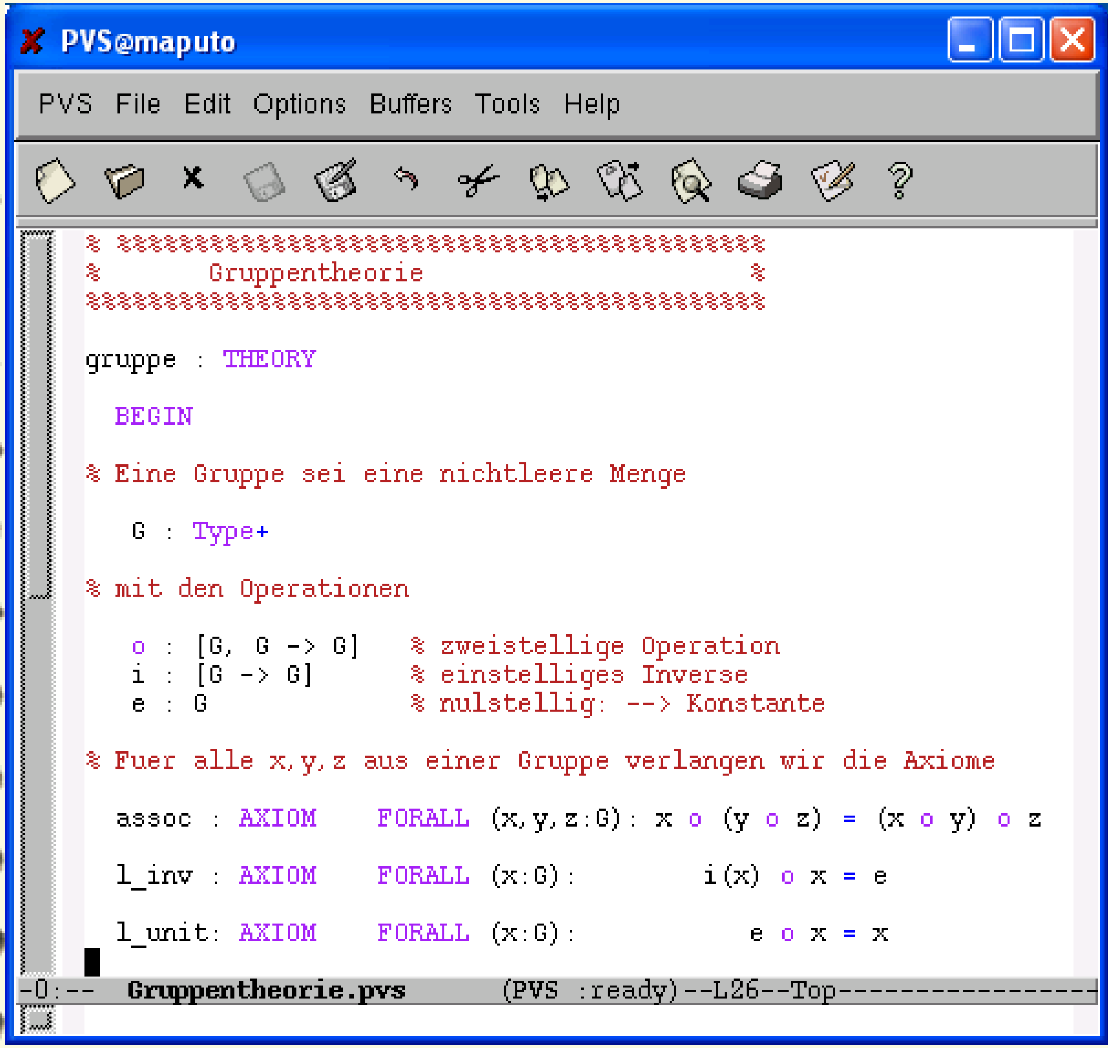
- Axiome beschreiben Annahmen über eine Signatur
  - Ein Axiom ist ein beliebige geschlossene Formel der Signatur
- Jedes Axiom muss benannt werden.
  - Mit dem Befehl

    ```
    (lemma <name>)
    ```

    kann der Beweiser eines der Axiome benutzen.
- Axiome müssen nicht bewiesen werden
  - Beispiel:

    ```
    assoziativ : FORALL (x,y,z : G): (x o y) o z = x o (y o z)
    ```
- Vorsicht: Wenn die Axiome widersprüchlich sind, kann man aus ihnen alles beweisen, z.B.:
  - 2=1
  - Ich bin der Papst
  - das Programm ist korrekt

Eine Spezifikation in PVS



Type+ heisst: nichtleere Sorte

Die Signatur

Die Axiome

Lemma, Proposition, Theorem,

- Sätze sind beliebige Formeln
- Wie Axiome benötigen sie einen Namen
- Sätze, Axiome, Theoreme müssen bewiesen werden
- Beispiel:
  - loesung : PROPOSITION
    FORALL (a,b : G): EXISTS (x:G): a o x = b
  -
- Die Namen
  - LEMMA
  - PROPOSITION und
  - THEOREM
  - haben alle die gleiche Bedeutung

Definierte Typen

- PVS erlaubt, neue Typen aus vorhandenen zu definieren
- Beispiele:
  - `Punkte : TYPE = [nat,nat]`
  - `posNat : TYPE = { n:nat | n > 0}`
  - `Zahlfunktion : TYPE = [nat -> nat]`
  - `Torsion : TYPE = {g:G|EXISTS (n:nat):power(g,n)= e}`

Definierte Funktionen

- Weitere Funktionen können in der Notation einer funktionalen Programmiersprache definiert werden
- Wichtigste Sprachkonzepte
  - Fallunterscheidung
    - `IF <bedingung> THEN <wert1> ELSE <wert2> ENDIF`
    - ```
      IF   <bed1>   THEN <wert1>
      ELSEIF <bed2> THEN <wert2>
      ...
      ENDIF
      ```
    - ```
      CASE <expr> OF
            <pattern1> : <wert1>
            <pattern2> : <wert2>
      ```
  - Rekursion
    - Rekursive Funktionen benötigen `MEASURE`
- KEINE Zuweisung
- KEIN `while`, `for`, `repeat`

Beispiel von Funktionen

- `teile(g1, g2 : G) : G = o(g1,inv(g2))`
  - Offensichtlich gilt
    `teile : [ G,G -> G]`
- `istGerade( n: nat) : bool =`
  `IF n mod 2 = 0 THEN true ELSE false ENDIF`
  - Damit ist
    `istGerade : [ nat -> bool]`
- `istUngerade (n:nat) = not (n mod 2=0)`
  - Offensichtlich
    `istUngerade : [ nat -> bool]`
- `absolut (n:nat) = IF n < 0 THEN -n ELSE n ENDIF`
  - Somit
    `absolut : [nat -> nat]`

Rekursiv definierte Funktionen

- ```
  power(g:G, n:nat) : RECURSIVE G  =
    IF (n=0) THEN e
    ELSE o(power(g,n-1),g)
    ENDIF
  ```
  - Gilt hier auch
    `power : [ G, nat -> G ]` ?

- ```
  threen(n:nat) : RECURSIVE nat  =
    IF (n=1) THEN 1
    ELSIF n mod 2 = 0 THEN threen(n/2)
    ELSE threen(3*n+1)
    ENDIF
  ```
  - Gilt hier auch
    `threen : [ nat -> nat ]` ?

Ist eine rekursiv definierte Funktion eine Funktion?

- PVS akzeptiert eine Funktionsdefinition

  $f(x_1 : T_1, \ldots, x_n : T_n)$: RECURSIVE T = ...

  erst dann als Funktion

  $f : [T_1, \ldots, T_n \rightarrow T]$

  wenn bewiesen ist:

  f ist für jedes Argumente-Tupel

  $(x_1, \ldots, x_n) \in [T_1, \ldots, T_n]$

  definiert und liefert einen Wert aus T.

- Insbesondere muss bewiesen werden, dass die Rekursion stoppt.

- Einen solchen Beweis kann mit mit einem Terminierungsmaß führen

Measures

```
sum(n:nat):RECURSIVE nat =
  IF (n=0) THEN 0
     ELSE n+sum(n-1) ENDIF

MEASURE n
```

- Jede Funktion `f:[T1-> T2]` muss *total* sein, d.h. sie muss für alle Argumente terminieren.
  - PVS verlangt für rekursive Funktionen eine Schranke
    `MEASURE` <expr>
    für die Anzahl der rekursiven Aufrufe.
- Die Behauptung, dass die Maßfunktion tatsächlich die Anzahl der Aufrufe von f beschränkt, wird Bestandteil der zu beweisenden TCCs.
  - Ist das bewiesen, dann ist die Funktion total
- Genauer muss bewiesen werden:
  - <expr> ist immer ganzzahlig und ≥ 0
  - in jedem inneren Aufruf wird <expr> kleiner
- Diese Beweisverpflichtungen werden als TCC (type-check-conditions) zurückgestellt. Sie müssen später bewiesen werden.

Beispiel

- Wir definieren den ggT rekursiv
  - ```
    ggT(x,y:posNat):RECURSIVE nat =
    IF (x=y) THEN x
    ELSEIF x>y THEN ggT(x-y,y)
    ELSE ggT(x,y-x)
    ENDIF

    MEASURE x+y
    ```
- Es muss gezeigt werden:
  - Korrekte Aufrufe:
    - $\forall x, y \in \text{posNat}.\ x > y \rightarrow x-y \in \text{posNat}$
    - $\forall x, y \in \text{posNat}.\ \text{not}(x>y) \rightarrow y-x \in \text{posNat}$
    - $\forall x, y \in \text{posNat}.\ x+y \geq 0$
    - $\forall x, y \in \text{posNat}.\ x > y \rightarrow x+y > (x-y)+y$
    - $\forall x, y \in \text{posNat}.\ \text{not}(x < y) \rightarrow x+y > x+(y-x)$
- Diese Beweisaufgaben werden als sogenannte TCCs gespeichert.
    ```

Relationen sind Prädikate

- Eine beliebige Relation
  - $R \subseteq A_1 \times A_2 \times \ldots \times A_n$
  - kann man auch durch die charakteristische Funktion ersetzen.
  - Dies ist eine mehrsortige Operation:
  - $\chi_R : A_1 \times A_2 \times \ldots \times A_n \rightarrow$ bool
  - mit
  - $(x_1,\ldots,x_n) \in R \Leftrightarrow \chi_R(x_1,\ldots,x_n) = \text{true}$
- PVS fasst Relationen immer als charakteristische Funktionen auf. Ist S eine Menge, dann sind z.B gleichwertig:
  - member(x,S)
  - S(x) = true
  - S(x)

Ein Beispiel in PVS

- Eine *Halbordnung* ist eine Menge H mit einer zweistelligen Relation $\leq$, so dass für alle $x,y,z \in H$ gilt:
  - $x \leq x$ (reflexiv)
  - $x \leq y , y \leq z \Rightarrow x \leq z$ (transitiv)
  - $x \leq y , y \leq x \Rightarrow x = y$ (anti-symmetrisch)
- Eine strikte Ordnung ist eine Menge H mit einer zweistelligen Relation $<$, so dass für alle $x,y,z \in H$ gilt:
  - $\text{not } x < x$ (irreflexiv)
  - $x < y, y < z \Rightarrow x < z$ (transitiv)
- Man zeige:
  - Ist $(H, \leq)$ eine Halbordnung, so wird durch
    - $x < y \ :\Leftrightarrow x \leq y \wedge \text{not } x = y$
  - eine strikte Ordnung definiert.
  - Ist $(H,<)$ eine strikte Ordnung, so wird durch
    - $x \leq y :\Leftrightarrow x < y \vee x = y$
    - eine Halbordnung definiert.
  - Geht man von einer Halbordnung $\leq$ zur strikten Ordnung $<$, und von dort wieder zur Halbordnung, so gelangt man wieder zum Ausgangspunkt.

Axiomatisierung in PVS

```
Halbordnung  : THEORY
  BEGIN
   H : TYPE                  % H eine Menge
   <= : [H,H -> bool]        % zweistellige Relation


% Axiome einer Halbordnung                          %
    reflexiv : AXIOM
         FORALL (x:H): x <= x ;

    antisymmetrisch: AXIOM
      FORALL (x,y:H):
         x <= y AND y <= x IMPLIES x = y;

    transitiv: AXIOM
      FORALL (x,y,z:H):
         x <= y AND y <= z IMPLIES x <= z;

  % Ein kleines Lemma zum Warmwerden
    zirkulaer : LEMMA
      FORALL (x,y,z:H):
        x <= y AND y <= z AND z <= x
                  IMPLIES x=y AND y = z;
```

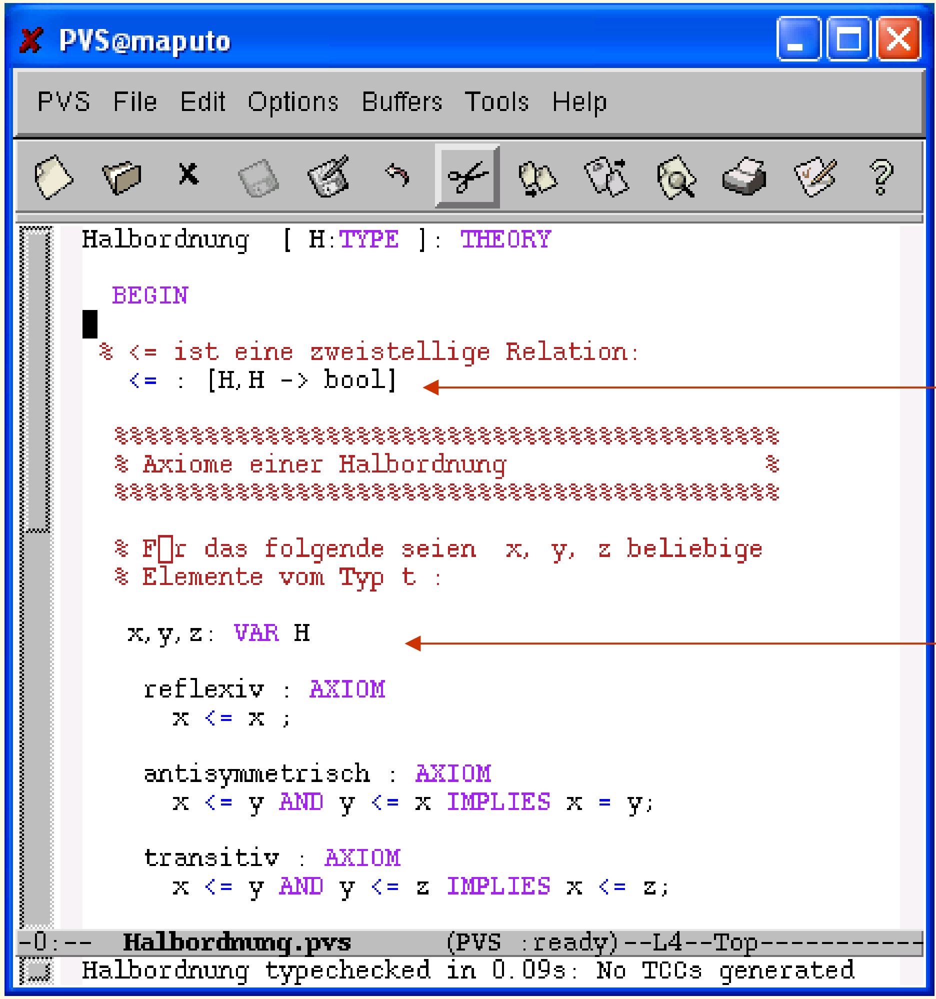
Spezifikation

Alternativ `<=` : `[H,H]`

Axiome für Halbordnungen

Eine (behauptete) Folgerung

Parametrisierte PVS-Spezifikation




```
Halbordnung  [ H:TYPE ]: THEORY

 BEGIN

 % <= ist eine zweistellige Relation:
   <= : [H,H -> bool]

  %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
  % Axiome einer Halbordnung                %
  %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

  % F□r das folgende seien  x, y, z beliebige
  % Elemente vom Typ t :

   x,y,z: VAR H

    reflexiv : AXIOM
      x <= x ;

    antisymmetrisch : AXIOM
      x <= y AND y <= x IMPLIES x = y;

    transitiv : AXIOM
      x <= y AND y <= z IMPLIES x <= z;
```

H wird als Typ-parameter verwendet

Eine Relation wird als Prädikat definiert

Seien x,y,z beliebige Elemente aus H, dann sollen die folgenden Axiome erfüllt sein ...

Axiomatisierung in PVS – (Forts.)

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Strikte Ordnung aus einer Halbordnung %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
  < (x,y:H): bool  =  x <= y  AND NOT x=y

 strict_reflexiv : Lemma NOT x < x

 strict_transitiv: Lemma
    x < y AND y < z IMPLIES x < z;

% Wir definieren aus der strikten
% Ordnung eine Halbordnung:

  <<(x,y:H): bool = x < y OR x=y

% und zeigen, dass die so entstehende
% identisch zur ursprünglichen ist:

  hinUndZurueck : Lemma
   x << y iff x <= y

END Halbordnung
```

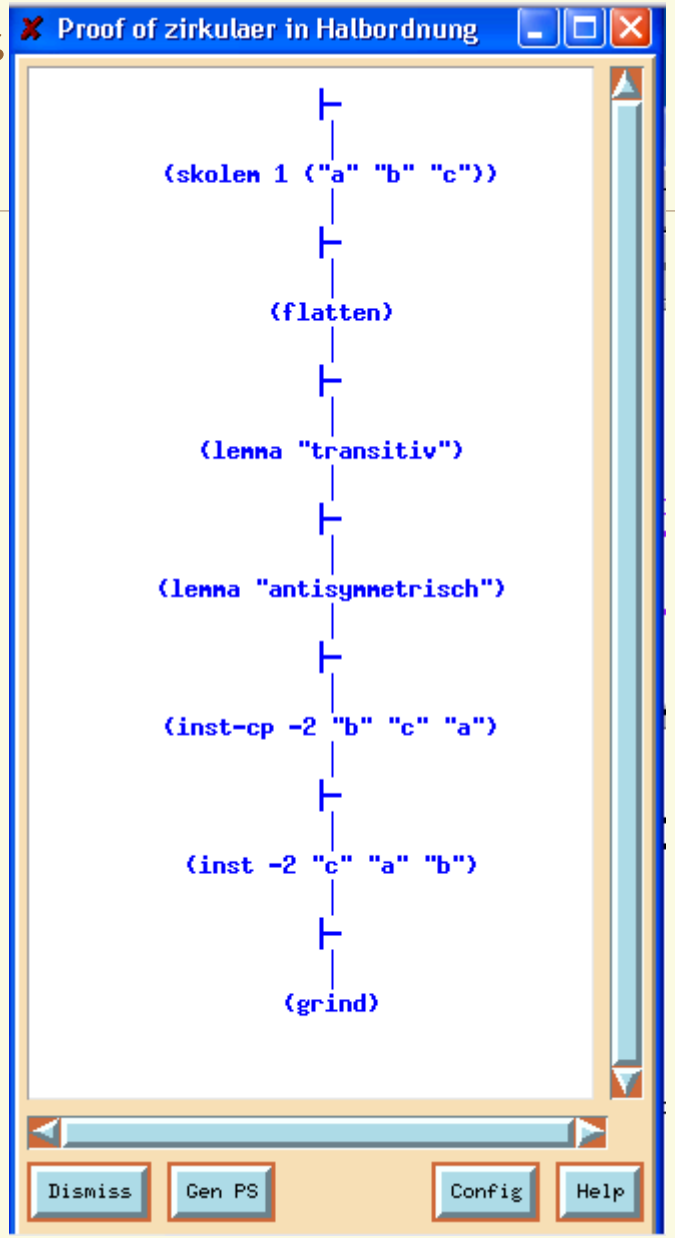
Definition einer Relation „<“ als Funktion

$<: H \times H \rightarrow \text{Bool}$

Semikolon „ ; “ hier nötig, weil ein Sonderzeichen „<„ folgt.

Beweis von „zirkulär“

- Mit Mx- xpr wird der Beweisbaum in einem X-Fenster angezeigt
- (grind) ist eine mächtige Strategie, die oft viele Trivialitäten erledigt



Logische Folgerung

- Sei (Σ, Ax) eine Spezifikation, bestehend aus einer
  - Signatur Σ, und einer
  - Menge Ax von Axiomen

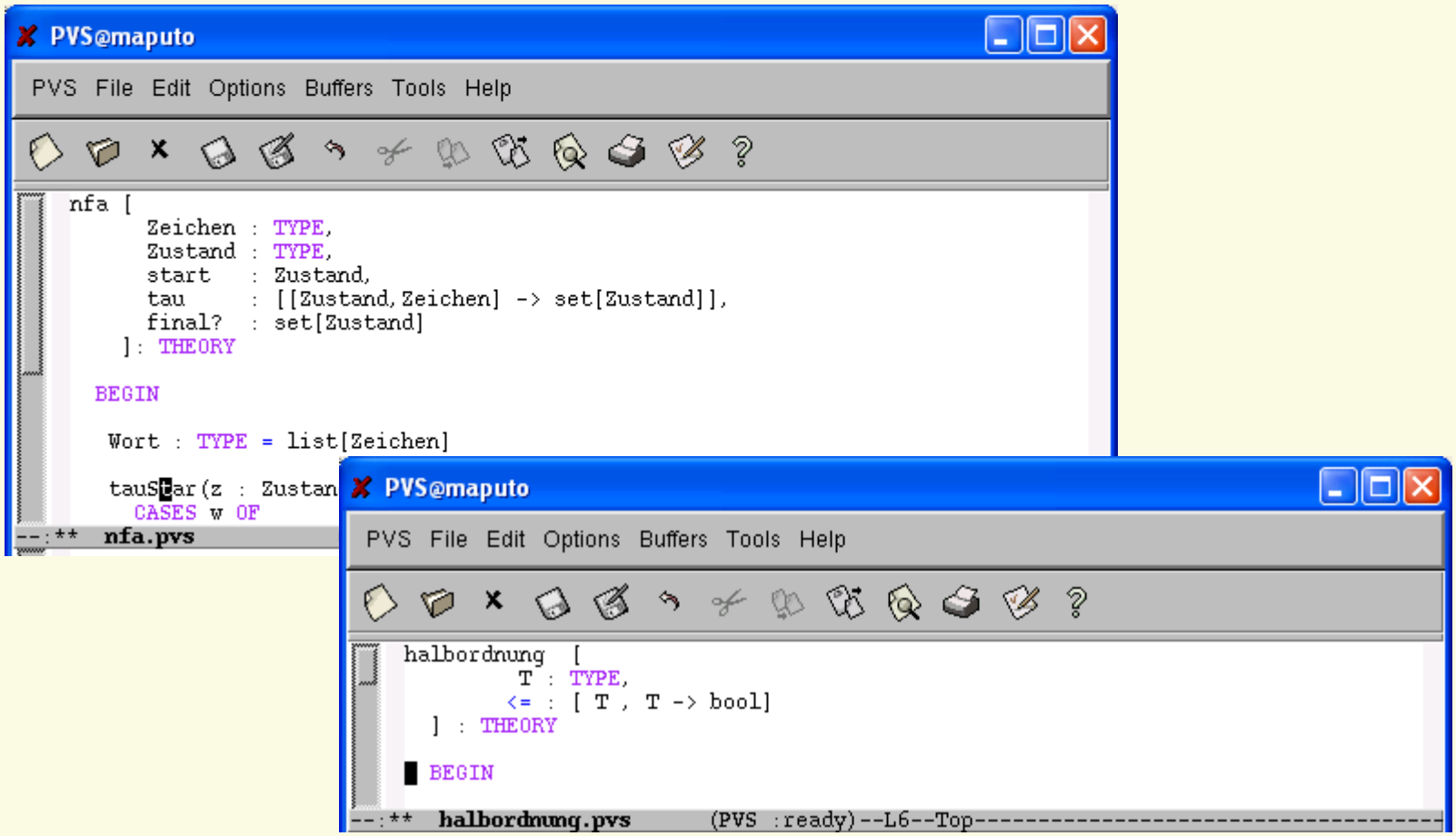
- Sei p eine Aussage in der Sprache von S. Wir sagen

$$(\Sigma, Ax) \models p$$

falls in jedem Modell von (Σ, Ax) auch p wahr ist.

Parametrisierte Theorien

- PVS-Theorien können beliebige Typen und Funktionen als Parameter haben. Beispiele haben wir bereits gesehen:

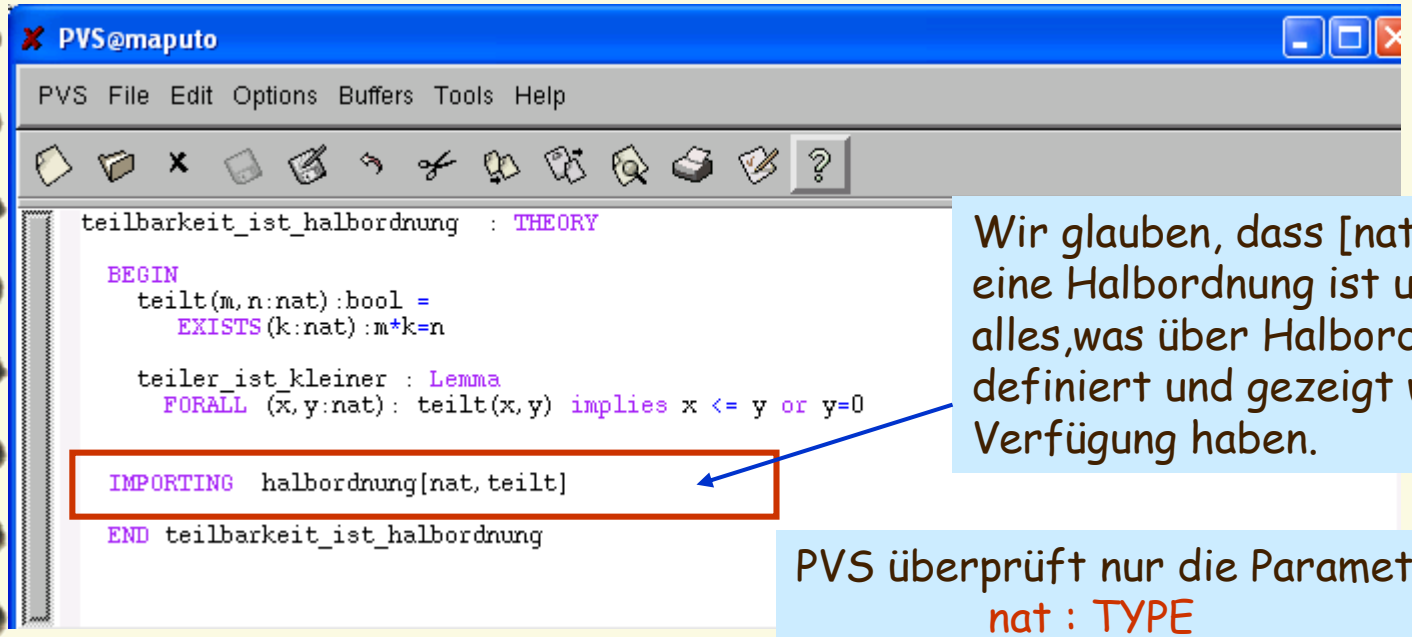


Theorien und Instanziierungen

- Allgemein definiert man in der Mathematik:
  - Sei H eine Menge und
  - $\leq : H \times H \rightarrow$ bool eine Abbildung.
  - dann heisst $(H, \leq)$ eine Halbordnung, falls
    - $\forall x \in H. \quad x \leq x$ (Reflexivität)
    - $\forall x,y \in H. \quad x \leq y \ \wedge y \leq x \Rightarrow x=y$ (Antisymmetrie)
    - $\forall x,y,z \in H. \ x \leq y \ \wedge y \leq z \Rightarrow x \leq z$ (Transitivität)
- Anschliessend kann man beweisen, dass zum Beispiel
  - $H = \mathbb{N}$ mit „|" als Teilbarkeitsordnung
    - $a \mid b \ :\Leftrightarrow \ \exists k \in$ Nat. $a*k=b$
  - Σ^* mit Präfixordnung
    - $u \angle v \ :\Leftrightarrow$ u ist Anfangsstück von v
  - Halbordnungen sind.
- Folglich kann man alle für Halbordnungen bewiesenen Theoreme auch für $(\mathbb{N}, \mid)$ und für $(\Sigma^*, \angle)$ ohne erneuten Beweis verwenden.

IMPORTING

- Auf einer Menge (hier Nat) definieren wir eine zweistellige Relation (hier „|“) und behaupten, dass wir eine Halbordnung haben.
- Importing macht alle Begriffe, AXIOMe, THEOREMe etc. für die importierte Theorie verfügbar.



Wir glauben, dass [nat,teilt] eine Halbordnung ist und wollen alles,was über Halbordnungen definiert und gezeigt wurde, zur Verfügung haben.

PVS überprüft nur die Parameter:

nat : TYPE √

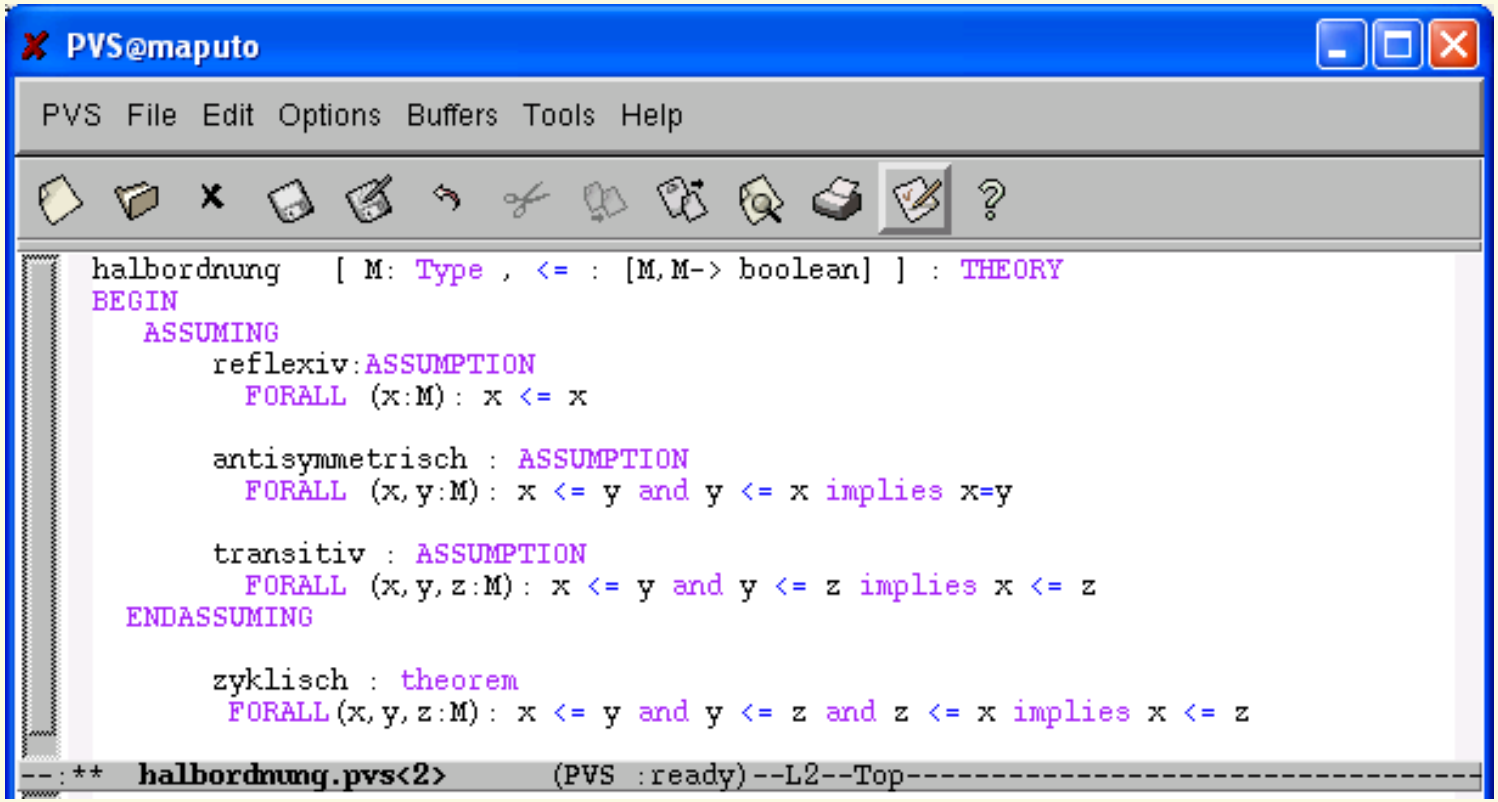
teilt : [nat, nat -> bool] √

Erfüllung einer Theorie

- Wir wollen, dass die Importierte Theorie verpflichtet wird, die notwendigen Axiome zu überprüfen
- Dazu dienen ASSUMPTIONS
- Die Axiome der Theorie werden in einem ASSUMING ... ENDASSUMING Teil als ASSUMPTIONS (Annahmen) formuliert
- Das System soll keine Instanziierung zulassen, in der eine der Annahmen nicht erfüllt ist.
- Beispielsweise:
  - Es muss nachgeprüft werden, dass in der Tat die Axiome der Halbordnung für $(\mathbb{N}, |\ )$ und für $(\Sigma^*, \sphericalangle)$ gelten.

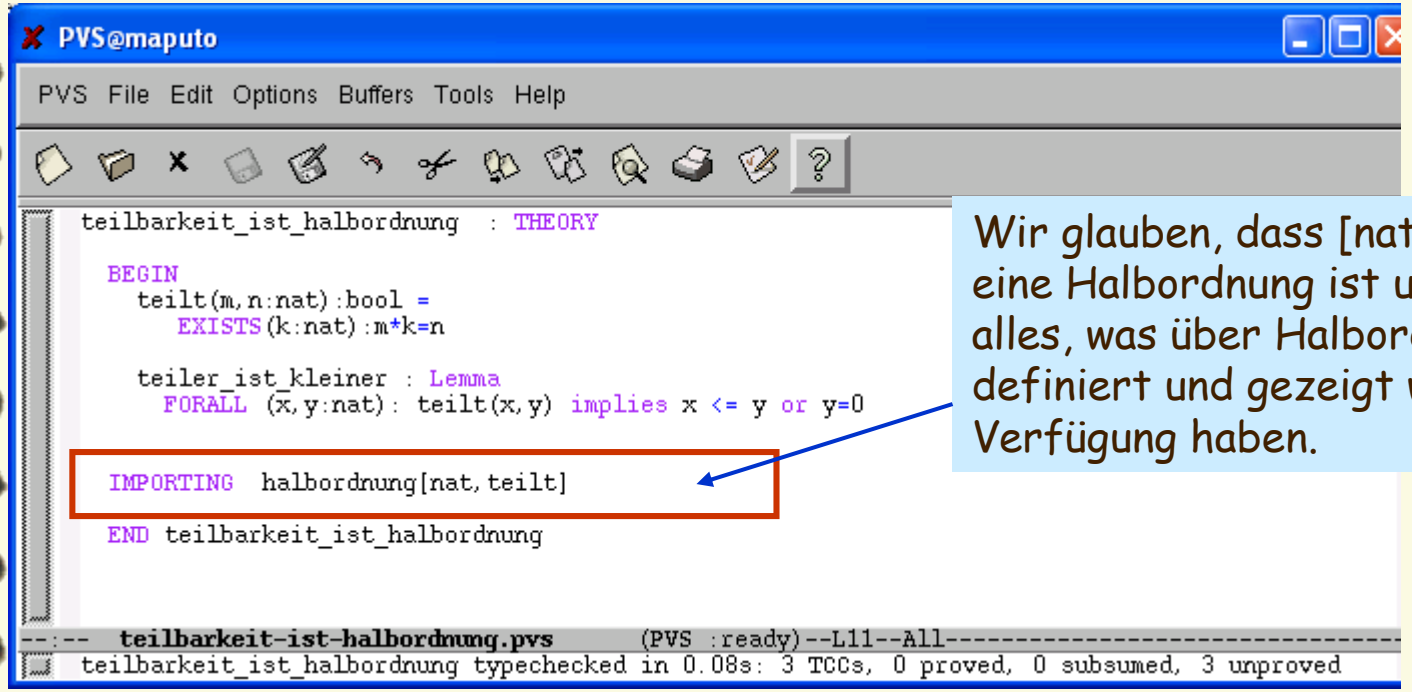
ASSUMING

- Zwischen ASSUMING und ENDASSUMING formuliert man die Assumptions, die für eine Halbordnung erfüllt sein sollen



IMPORTING

- Auf einer Menge (hier $\mathbb{N}$) definieren wir eine zweistellige Relation (hier „|“) und behaupten, dass wir eine Halbordnung haben.

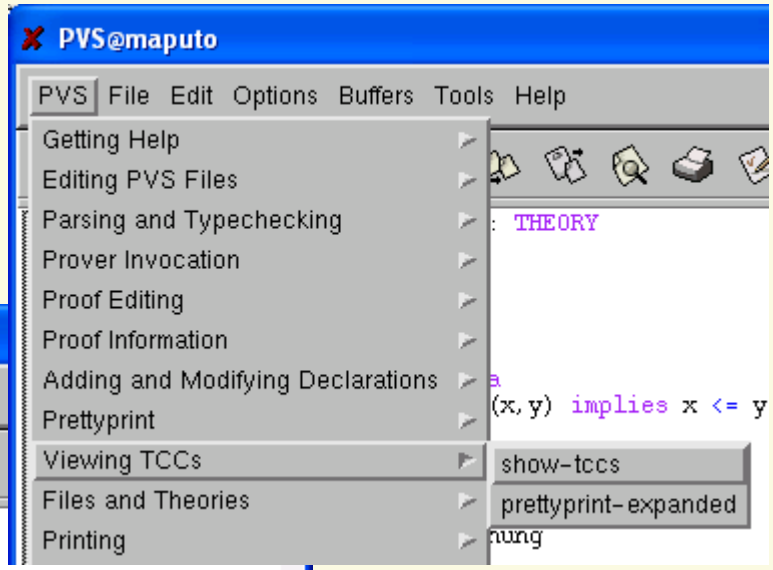


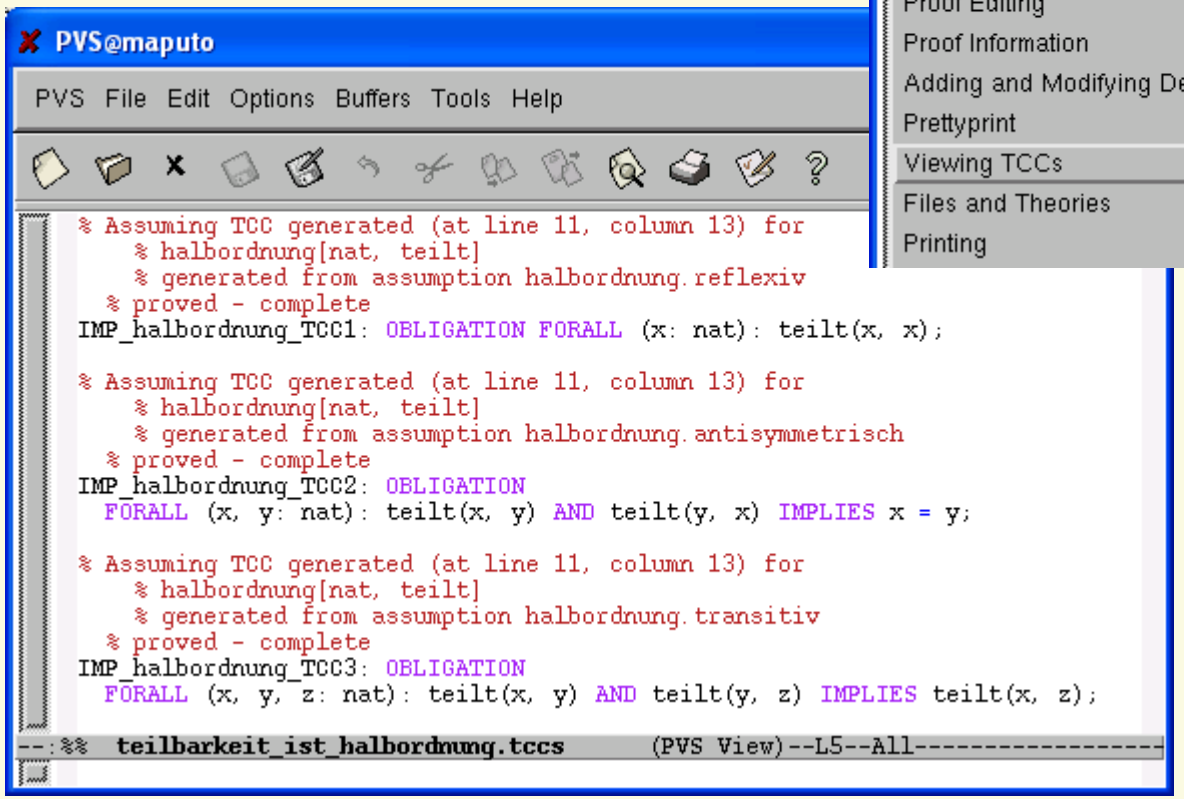
Wir glauben, dass [nat,teilt] eine Halbordnung ist und wollen alles, was über Halbordnungen definiert und gezeigt wurde, zur Verfügung haben.

PVS teilt uns mit, dass 3 Obligationen entstanden sind, die es noch nicht selber beweisen konnte.

TCCs - Obligationen

- Die Obligationen entsprechen den Axiomen für Halbordnungen, spezialisiert für die Funktion „teilt“
- Mit show-tccs lädt man sie in einen Buffer





Alle Obligationen müssen natürlich bewiesen werden

M-x xpr