

# Beweisen mit PVS

## Anwendung auf Datenbanken

Henning Thielemann

2007-08-28



- 1 Motivation
- 2 Einstieg in PVS
- 3 PVS-Sprachelemente
  - Typen
  - Funktionen
  - Logik
- 4 Datenbanken



## 1 Motivation

2 Einstieg in PVS

3 PVS-Sprachelemente

4 Datenbanken



# Korrektheit von Datenbankabfragen

- `SELECT * FROM Person`  
`WHERE name='Müller' AND name='Meier';`  
Ergebnis immer leer → sinnlos
- `SELECT * FROM Person WHERE 0=0;`  
Zu komplizierte Anfrage  
(TRUE reicht, oder WHERE ganz weglassen)
- `SELECT * FROM Person`  
`WHERE CASE WHEN x > 0 THEN true ELSE false END;`  
Zu komplizierte Anfrage (`x > 0` reicht)



# Korrektheit von Integritätsbedingungen

```
CREATE TABLE item (  
    type          char,  
    title         varchar(40),  
    duration      time,  
    numPages      integer,  
    CONSTRAINT type CHECK (type='b' AND duration IS NULL),  
    CONSTRAINT type CHECK (type='v' AND numPages IS NULL)  
);
```

type='b' und type='v' widersprechen sich



# Beweiser unterstützen semantische Tests

- Beweise, dass Anfrage nie etwas ausgibt.  
Falls beweisbar → Semantischer Fehler liegt vor
- Beweise, dass es Datenbank gibt, bei der Anfrage etwas ausgibt.  
Falls beweisbar → Dieser Fehler wurde nicht gemacht
- Falls keine der beiden Aussagen beweisbar:  
Beweisstrategie könnte verbessert werden.
- Falls beide beweisbar:  
In Theorie stecken unbewiesene Behauptungen  
oder die Übersetzung von SQL in PVS ist schiefgegangen.



# Kann man alle Fehler finden?

- Absolute Sicherheit nicht möglich.  
Ob ein Ausdruck wahr oder falsch ist,  
ist nicht entscheidbar.
- Absolute Sicherheit gar nicht nötig:  
Angenommen Tabelle mit vier Spalten  
positiver natürlicher Zahlen  $x$ ,  $y$ ,  $z$ ,  $n$   
`SELECT * FROM Tripel WHERE n>2 AND x^n + y^n = z^n`  
Gibt nie etwas aus, wegen großen Satzes von Fermat.  
Beweis benötigte mehrere Mathematikerjahrhunderte.



- 1 Motivation
- 2 **Einstieg in PVS**
- 3 PVS-Sprachelemente
- 4 Datenbanken



# Übersicht Beweissysteme

- PVS
- Mizar
- Coq
- Isabelle / HOL



# PVS

PVS = Prototype Verification System  
von SRI International

- Sprache zum Formulieren von Sätzen
- Umgebung zum Führen von Beweisen

Beweisen mit

- eingebauten Entscheidungsprozeduren
- Bibliothek von (bewiesenen) Sätzen
- Beweisstrategien



# Automatisch beweisbare Sätze

1.  $\neg \exists x \in \mathbb{R} \ x < 2 \wedge x > 3$
2.  $\neg \exists x \in \mathbb{Z} \ x > 2 \wedge x < 3$
3.  $1 + \dots + n = \frac{n \cdot (n + 1)}{2}$
4.  $2^n \mid ((n + 1) \cdot (n + 2) \cdot \dots \cdot (2 \cdot n))$



# Nur manuell beweisbare Sätze

1.  $\exists x \in \mathbb{R} \ x > 2 \wedge x < 3$
2.  $\forall x \in \mathbb{R} \ x = 0 \vee x^2 > 0$
3.  $\forall \{y, z\} \subset \mathbb{R} \ (\forall x \in \mathbb{R} \ x < y \iff x < z) \iff y = z$



# Beweisprüfer vs. Computeralgebra

Wolfram bewirbt Mathematica 6 mit Beweiserqualitäten

```
In[1]:= FullSimplify[ $\forall x (x \cdot e == x \wedge a \cdot \bar{a} == e),$   
           $\forall \{x, y, z\} (x \cdot (y \cdot z) == (x \cdot y) \cdot z \wedge e \cdot x == x \wedge \bar{x} \cdot x == e) ]$   
Out[1]= True
```



# Beweisprüfer vs. Computeralgebra

## Voraussetzung

Assoziativität  $\forall x \forall y \forall z (x \circ y) \circ z = x \circ (y \circ z)$

Linksneutrales Element  $\forall x e \circ x = x$

Linksinverses Element  $\forall x \bar{x} \circ x = e$

## Behauptung

Rechtsneutrales Element  $\forall x x \circ e = x$

Rechtsinverses Element  $\forall x x \circ \bar{x} = e$



# Fähigkeiten von PVS

PVS kann

- Quantoren mit erratenem Wert belegen (Typen helfen bei Auswahl),
- logisch vereinfachen,
- konstante Ausdrücke zusammenrechnen,
- ausmultiplizieren.

PVS muss gar nicht übermäßig intelligent sein,  
Beweise müssen reproduzierbar sein



# Beweisschritte in PVS

- Quantor eliminieren (skolem oder instantiate)
- Satz anwenden
- Vereinfachen
- Induktion
- Extensionalität



- 1 Motivation
- 2 Einstieg in PVS
- 3 PVS-Sprachelemente**
- 4 Datenbanken



- 1 Motivation
- 2 Einstieg in PVS
- 3 PVS-Sprachelemente
  - Typen
  - Funktionen
  - Logik
- 4 Datenbanken



# Theorien mit Typparametern

```
sets [T: TYPE]: THEORY
BEGIN

  set: TYPE = setof [T]

  ...

END sets
```

- vergleichbar mit C++-Templates
- automatisierte Ermittlung einer sinnvollen Belegung von Typparametern (type inference)



# Untertypen

$X$  ist Untertyp von  $Y$  wenn  $X$  nur Werte von  $Y$  enthält

Beispiel:

Natürliche Zahlen bilden Untertyp der ganzen Zahlen

```
naturalnumber: TYPE = nonneg_int
```

```
nonneg_int: NONEMPTY_TYPE =  
  {i: int | i >= 0} CONTAINING 0
```



# Abhängige Typen

Abhängiger Typ: Typ hängt von Werten ab  
(nicht nur von anderen Typen)

```
below(i): TYPE = {s: nat | s < i}
```

```
is_finite(s): bool =  
  EXISTS N, (f: [(s) -> below[N]]): injective?(f)
```

```
finite_set: TYPE = (is_finite) CONTAINING emptyset[T]
```

Gegenbeispiel:

set[T] wird nicht als abhängiger Typ bezeichnet.

Erlaubt sehr elegante Formalisierung von Mathematik.



# Behandlung von Zahlen

- Zahlentypen sind Untertypen von `number`
- Untertypen `number`  $\supset$  `real`  $\supset$  `rat`  $\supset$  `int`  $\supset$  `nat`
- Zahlenliterale haben Typ `number`
- nicht konstruktiv, sehr formal



- 1 Motivation
- 2 Einstieg in PVS
- 3 PVS-Sprachelemente
  - Typen
  - Funktionen
  - Logik
- 4 Datenbanken



# Funktionen

- Grundlegung durch Funktionen
- Grundlegung nicht durch Mengen  
(etwa Axiomatik von ZERMELO und FRAENKEL)
- Menge als Funktion  
 $\text{set: TYPE} = [T \rightarrow \text{bool}]$   
zwei Sichten auf gleichen Typ: Prädikat vs. Menge



# Syntaktischer Zucker für Funktionen

$\{(x:T) \mid p(x)\}$	-	LAMBDA (x:T): p(x)
nicht erlaubt: $\{2*n \mid n\}$	→	$\{m \mid \text{EXISTS } n: m=2*n\}$
FORALL (x:T): p(x)	-	forall (LAMBDA (x:T): p(x))
EXISTS (x:T): p(x)	-	exists (LAMBDA (x:T): p(x))

## funktionale Interpretation

- wird bei Anwendung von Quantor-Sätzen benötigt
- klärt automatisch, wie mit undefinierten Werten umgegangen werden soll



# Totale Funktionen

- Es gibt kein `undefined`!
- Stattdessen: Erteilen von Beweispflichten (proof obligations)
- Handhabbar, weil Definitionsbereiche von Funktionen abhängige Untertypen sein können
- Bedeutung für Datenbanken:  
NULL muss ausdrücklich in Datentyp untergebracht werden



# Totale Funktionen: Division

```
mo460941: THEORY
```

```
BEGIN
```

```
x, y, z: VAR nreal
```

```
solution: THEOREM
```

```
x + y + z = 1 AND 1/x + 1/y + 1/z = 1 IFF
```

```
(x = 1 AND y = -z) OR
```

```
(y = 1 AND z = -x) OR
```

```
(z = 1 AND x = -y)
```

```
END mo460941
```



- 1 Motivation
- 2 Einstieg in PVS
- 3 PVS-Sprachelemente
  - Typen
  - Funktionen
  - Logik
- 4 Datenbanken



# Logik

- PVS verwendet klassische Logik, keine konstruktivistische Logik
- Voraussetzungen Und-verknüpft  
Behauptungen Oder-verknüpft  
Form  $A_1 \wedge A_2 \wedge A_3 \Rightarrow B_1 \vee B_2 \vee B_3$
- Dualität: Voraussetzungen und Behauptungen
- Aussagen werden immer ohne Negation dargestellt

$$\begin{aligned} A \text{ AND } B &\Rightarrow C \\ &\iff \\ A &\Rightarrow \text{NOT } B \text{ OR } C \end{aligned}$$



# Fallunterscheidung

$$\begin{array}{c} A \Rightarrow B \\ \iff \\ (A \text{ AND } C \Rightarrow B) \text{ AND } (A \Rightarrow C \text{ OR } B) \end{array}$$

Kann man auch verwenden,  
um Sätze lokal einzuführen und zu beweisen



# Induktion

- formaler Satz, der Induktionsprinzip begründet
- Prelude.naturalnumbers:

nat\_induction: LEMMA

(p(0) AND (FORALL j: p(j) IMPLIES p(j+1)))  
IMPLIES (FORALL i: p(i))

- Induktionsbeweise häufig nicht so elegant  
aber automatisch durchführbar



# Beispiel: Ungleichungen

- $a^2+b^2 \geq 2*a*b$
- $a^2+b^2+c^2 \geq a*b+b*c+c*a$



# Beispiel: eindeutige Existenz

verschiedene Definitionen

- Quantoren

```
unique?(p): bool =  
  FORALL x, y: p(x) AND p(y) IMPLIES x = y
```

```
exists1(p): bool = (EXISTS x: p(x)) AND unique?(p)
```

- Mengen

```
singleton?(a): bool =  
  EXISTS (x:(a)): (FORALL (y:(a))): x = y
```

- Satz der Äquivalenz

```
singleton_exists1: LEMMA  
  sets[T].singleton? = exists1
```



- 1 Motivation
- 2 Einstieg in PVS
- 3 PVS-Sprachelemente
- 4 Datenbanken**



# relationale Algebra: Multimenge

- `multiset`: `TYPE = [T -> nat]`
- `s: multiset[T]`, `x: T`, dann gibt `s(x)` an, wie oft `x` in der Multimenge `s` vorkommt.



# Tabellen vs. Multimengen

Tabelle

Vorname	Nachname
Hans	Meier
Max	Müller
Opa	Hoppenstedt
Hans	Meier

Multimenge

["Hans", "Meier"] → 2  
["Max", "Müller"] → 1  
["Opa", "Hoppenstedt"] → 1  
["Klaus", "Hülsensack"] → 0



# relationale Algebra: Operationen

- Auswahl (filter)

```
filter (p: pred[T]) (a: multiset[T]): multiset[T]
```

- kartesisches Produkt

```
cartesian_product(a: multiset[T], b: multiset[S]):  
  multiset[[T, S]]
```

- Projektion (map) (Schwierigkeit: Endlichkeit!)

```
project(f: [T -> S])(a: finite_multiset[T]):  
  finite_multiset[S]
```

- Gruppierung (uncurry)

```
group(ab: multiset[[T,S]]): [T -> multiset[S]]
```

- keine sinnvolle Definition für Komplementmenge



# Beispiel

- Filter auf Multimengen
- `contradictory_filter2`: LEMMA  
`empty_filter? (filter  
  (LAMBDA s: s = "Müller" AND s = "Meier"))`
- `wellformed_filter2`: LEMMA  
`NOT empty_filter? (filter  
  (LAMBDA s: s = "Müller" OR s = "Meier"))`

