Formale Techniken der Software-Entwicklung

Matthias Hölzl, Christian Kroiß

26. Mai 2014

„Ontologisches Commitment“

- Aussagenlogik: Es gibt wahre und falsche Aussagen
- Prädikatenlogik: Es gibt Objekte, die in gewissen Relationen zueinander stehen; die Relationen selber sind nicht Objekte der Theorie
- Temporallogik: Es gibt Aussagen, die zu gewissen Zeitpunkten wahr und falsch sein können
- Typtheorie: Es gibt Objekte, die in gewissen Relationen zueinander stehen, Relationen zwischen Relationen, usw.; Relationen sind in der Theorie enthalten

Quantoren

- $\forall$ steht für „für alle"
- $\exists$ steht für „es gibt (mindestens) ein"
- $\exists!$ steht für „es gibt genau ein" und wird durch

$$\exists! x.\phi \simeq \exists x.\phi \wedge \forall x.\forall y.\phi \wedge \phi[x \mapsto y] \Rightarrow x = y$$

- „Für alle ... gilt ..." formalisiert man als

$$\forall x. \cdots \Rightarrow \cdots$$

- „Es gibt ... mit ..." formalisiert man als

$$\exists x. \cdots \wedge \cdots$$

- Es gilt $\neg\forall x.\phi = \exists x.\neg\phi$ und $\neg\exists x.\phi = \forall x.\neg\phi$

Eine Struktur $\mathcal{A}$ zur Signatur $\sigma = (\mathit{Const}, \mathit{Fun}, \mathit{Pred}, |.|)$ (kurz σ-Struktur) besteht aus

- einer nichtleeren Menge A, dem *Träger* (oder der *Grundmenge*) der Struktur
- einem Element $c^{\mathcal{A}} \in A$ für jedes Konstantensymbol $c \in \mathit{Const}$
- einer Funktion $f^{\mathcal{A}} : A^n \to A$ für jedes n-stellige Funktionssymbol $f \in \mathit{Fun}$
- einer Relation $P^{\mathcal{A}} \subseteq A^n$ für jedes n-stellige Prädikatssymbol P

Eine Struktur heißt *endlich* (bzw. *unendlich*), wenn die Trägermenge endlich (bzw. unendlich) ist.

Modelle der Prädikatenlogik

Definition

Ein *Modell* $\mathcal{M}$ einer Sprache $\mathcal{L}$ ist ein Paar $(\mathcal{A}, w)$, bestehend aus einer $\mathcal{L}$-Struktur $\mathcal{A}$ (mit Träger A) und einer *Belegung* $w : Var \to A$.
Wir schreiben $[\![c]\!]_{\mathcal{M}}$ (oder $c^{\mathcal{M}}$, oder $[\![c]\!]$ falls $\mathcal{M}$ klar ist) für $c^{\mathcal{A}}$, entsprechend für Funktions- und Prädikatensymbole.

Modelle einer Sprache $\mathcal{L}$ nennt man auch $\mathcal{L}$-Modelle oder Interpretationen von $\mathcal{L}$. Den Träger von $\mathcal{A}$ nennt man auch Träger von $\mathcal{M}$.

Wir schreiben $M[x \mapsto a]$ für das Modell (A, w') mit

$$w'(y) = \begin{cases} w(y) & \text{für } y \neq x \\ a & \text{für } y = x \end{cases}$$

Semantik von Termen

Durch ein Modell $\mathcal{M}$ wird jedem $\mathcal{L}$-Term ein Element aus A zugeordnet:

$$
\begin{aligned}
[\![x]\!]_{\mathcal{M}} &= w(x) \\
[\![c]\!]_{\mathcal{M}} &= c^{\mathcal{A}} \\
[\![f(t_1, \dots, t_n)]\!]_{\mathcal{M}} &= f^{\mathcal{A}}([\![t_1]\!]_{\mathcal{M}}, \dots, [\![t_n]\!]_{\mathcal{M}})
\end{aligned}
$$

Erfüllungsrelation

Die Semantik von Formeln lässt sich durch die *Erfüllungsrelation* $\mathcal{M} \models \phi$ ($\mathcal{M}$ erfüllt ϕ oder $\mathcal{M}$ ist ein Modell von ϕ) beschreiben:

$$
\begin{aligned}
\mathcal{M} \models R(t_1, \ldots, t_n) &\iff R^{\mathcal{A}}([\![t_1]\!]_{\mathcal{M}}, \ldots, [\![t_n]\!]_{\mathcal{M}}) \\
\mathcal{M} \models t_1 = t_2 &\iff [\![t_1]\!]_{\mathcal{M}} = [\![t_2]\!]_{\mathcal{M}} \\
\mathcal{M} \models \neg\phi &\iff \mathcal{M} \models \phi \text{ ist falsch } (\mathcal{M} \not\models \phi) \\
\mathcal{M} \models \phi \land \psi &\iff \mathcal{M} \models \phi \text{ und } \mathcal{M} \models \psi \\
\mathcal{M} \models \phi \lor \psi &\iff \mathcal{M} \models \phi \text{ oder } \mathcal{M} \models \psi \\
\mathcal{M} \models \phi \Rightarrow \psi &\iff \mathcal{M} \not\models \phi \text{ oder } \mathcal{M} \models \psi \\
\mathcal{M} \models \phi \Leftrightarrow \psi &\iff (\mathcal{M} \models \phi \text{ und } \mathcal{M} \models \psi) \\
&\qquad\quad \text{oder } (\mathcal{M} \not\models \phi \text{ und } \mathcal{M} \not\models \psi) \\
\mathcal{M} \models \forall x.\phi &\iff \mathcal{M}[x \mapsto a] \models \phi \text{ für alle } a \\
\mathcal{M} \models \exists x.\phi &\iff \text{es gibt } a \text{ mit } \mathcal{M}[x \mapsto a] \models \phi
\end{aligned}
$$

Erfüllbarkeit, Tautologien

- Eine Formel heißt *erfüllbar*, wenn sie ein Modell besitzt
- Eine Formel $\phi \in \mathcal{L}$ heißt *allgemeingültig*, *logisch gültig* oder *Tautologie*, wenn sie in allen Modellen wahr ist, wenn also für alle $\mathcal{L}$-Modelle gilt $\mathcal{M} \models \phi$
- Formeln ϕ und ψ heißen *logisch äquivalent*, wenn sie von den gleichen $\mathcal{L}$-Modellen erfüllt werden, wenn also $\mathcal{M} \models \phi$ genau dann gilt, wenn $\mathcal{M} \models \psi$ gilt
- Sei Φ eine Menge von Formeln. Wir schreiben $\mathcal{M} \models \Phi$, wenn $\mathcal{M} \models \phi$ für alle $\phi \in \Phi$ gilt
- Wir schreiben $\Phi \models \psi$ (aus Φ folgt ψ) wenn jedes Modell von Φ auch ψ erfüllt, wenn also gilt $\mathcal{M} \models \Phi \implies \mathcal{M} \models \psi$.

Vorsicht: Die Definition von $\mathcal{M} \models \Phi$ ist anders als bei Sequenzen!

Koninzidenzsatz

Theorem (Koninzidenzsatz)

Seien ϕ eine Formel, $V \subseteq Var$ eine Menge von Variablen mit $\mathrm{fv}(\phi) \subseteq V$, $\mathcal{M}$ und $\mathcal{N}$ Modelle über derselben Struktur $\mathcal{A}$ mit $x^{\mathcal{M}} = x^{\mathcal{N}}$ für alle $x \in V$. Dann gilt $\mathcal{M} \models \phi \iff \mathcal{N} \models \phi$.

Der Koinzidenzsatz besagt also, dass nur die Belegung der Variablen, die tatsächlich in einer Formel vorkommen einen Einfluss auf die Modellbeziehung hat.

Substitutionssatz

Sei $\mathcal{M}$ ein $\mathcal{L}$-Modell. Wir defineren $\mathcal{M}\sigma$ als das Modell $(\mathcal{A}, w^\sigma)$, wobei $w^\sigma(x) = (x\sigma)^\mathcal{M}$ für alle $x \in \mathit{Var}$.

Theorem (Substitutionssatz)

Sei $\mathcal{M}$ ein Modell und σ eine Substitution. Für alle Formeln ϕ, die mit σ kollisionsfrei sind, gilt

$$\mathcal{M} \models \phi\sigma \iff \mathcal{M}\sigma \models \phi$$

Der Beweis erfolgt durch Induktion über ϕ.

Folgerungen aus dem Substitutionssatz

Aus dem Substitutionssatz folgt das wichtige Korollar

Korollar (aus dem Substitutionssatz)

Seien ϕ und $[x \mapsto t]$ kollisionsfrei. Dann gelten

- $\forall x.\phi \models \phi[x \mapsto t]$
- $\phi[x \mapsto t] \models \exists x.\phi$
- $\phi[x \mapsto t_1], t_1 = t_2 \models \phi[x \mapsto t_2]$

Gelte $\mathcal{M} \models \forall x.\phi$, also $\mathcal{M}[x \mapsto a] \models \phi$ für alle $a \in A$. Da $t^{\mathcal{M}} \in A$ ist gilt somit auch $\mathcal{M}[x \mapsto t] \models \phi$, nach dem Substitutionssatz also $\mathcal{M} \models \phi[x \mapsto t]$. Die anderen Aussagen zeigt man analog.

Generalisierte Aussagen

Sei ϕ eine Formel mit $\vec{x} = \{x_1, \ldots, x_n\} = \mathrm{fv}(\phi)$. Dann schreiben wir $\phi^\forall$ für die Formel $\forall x_1. \ldots . \forall x_n . \phi = \forall \vec{x}. \phi$ und nennen $\phi^\forall$ die *Generalisierte von* ϕ.

Wir schreiben $\mathcal{A} \models \psi$ wenn für jede Variablenbelegung w gilt $(\mathcal{A}, w) \models \psi$. Mit dieser Notation erhalten wir

$$\mathcal{A} \models \phi \iff \mathcal{A} \models \phi^\forall$$

Sequenzen (Erinnerung)

Eine Sequenz $\Gamma \vdash \Delta$ entspricht einer Implikation: $\bigwedge \Gamma \Rightarrow \bigvee \Delta$

Es ist also $\phi_1, \ldots, \phi_m \vdash \psi_1, \ldots \psi_n$ äquivalent zu

$$\neg\phi_1 \vee \cdots \vee \neg\phi_m \vee \psi_1 \vee \cdots \vee \psi_n$$

Sequenzenkalkül (1)

$$\frac{\Gamma_1 \vdash \Delta_1}{\Gamma_2 \vdash \Delta_2}\mathbf{W} \qquad \text{if } \Gamma_1 \subseteq \Gamma_2 \land \Delta_1 \subseteq \Delta_2$$

$$\frac{}{\Gamma, \phi \vdash \psi, \Delta}\mathbf{Ax} \qquad \text{if } \phi \simeq \psi$$

$$\frac{}{\Gamma, \bot \vdash \Delta}\bot \qquad \frac{}{\Gamma \vdash \top, \Delta}\top$$

$$\frac{\Gamma \vdash \phi, \Delta}{\Gamma, \neg\phi \vdash \Delta}\neg\vdash \qquad \frac{\Gamma, \phi \vdash \Delta}{\Gamma \vdash \neg\phi, \Delta}\vdash\neg$$

Sequenzenkalkül (2)

$$\frac{\phi, \psi, \Gamma \vdash \Delta}{\phi \wedge \psi, \Gamma \vdash \Delta} \wedge\vdash \qquad \frac{\Gamma \vdash \phi, \Delta \quad \Gamma \vdash \psi, \Delta}{\Gamma \vdash \phi \wedge \psi, \Delta} \vdash\wedge$$

$$\frac{\phi, \Gamma \vdash \Delta \quad \psi, \Gamma \vdash \Delta}{\phi \vee \psi, \Gamma \vdash \Delta} \vee\vdash \qquad \frac{\Gamma \vdash \phi, \psi, \Delta}{\Gamma \vdash \phi \vee \psi, \Delta} \vdash\vee$$

$$\frac{\psi, \Gamma \vdash \Delta \quad \Gamma \vdash \phi, \Delta}{\phi \Rightarrow \psi, \Gamma \vdash \Delta} \Rightarrow\vdash \qquad \frac{\Gamma, \phi \vdash \psi, \Delta}{\Gamma \vdash \phi \Rightarrow \psi, \Delta} \vdash\Rightarrow$$

Sequenzenkalkül (3)

$$\frac{}{\Gamma \vdash t = t, \Delta}\textbf{Refl} \qquad \frac{t_1 = t_2, \Gamma[x \mapsto t_1] \vdash \Delta[x \mapsto t_1]}{t_1 = t_2, \Gamma[x \mapsto t_2] \vdash \Delta[x \mapsto t_2]}\textbf{Repl}$$

$$\frac{\phi[x \mapsto t], \Gamma \vdash \Delta}{\forall x.\phi, \Gamma \vdash \Delta}\forall\vdash \qquad \frac{\Gamma \vdash \phi[x \mapsto c], \Delta}{\Gamma \vdash \forall x.\phi, \Delta}\vdash\forall \quad c \text{ frisch}$$

$$\frac{\phi[x \mapsto c], \Gamma \vdash \Delta}{\exists x.\phi, \Gamma \vdash \Delta}\exists\vdash \quad c \text{ frisch} \qquad \frac{\Gamma \vdash \phi[x \mapsto t], \Delta}{\Gamma \vdash \exists x.\phi, \Delta}\vdash\exists$$

Endlichkeitssatz

Sei Φ eine (möglicherweise unendliche) Menge von Formeln.

Wir schreiben $\Phi \vdash_{\mathrm{seq}} \phi$, wenn es eine Ableitung im Sequenzenkalkül gibt, deren Antezedens nur Formeln aus Φ enthält und deren Sukzedens ϕ enthält. Wir schreiben $\Phi \models \bot$ wenn Φ unerfüllbar ist.

Theorem (Endlichkeitssatz)

Wenn gilt $\Phi \vdash_{\mathrm{seq}} \phi$, so gibt es eine endliche Teilmenge $\Phi_0 \subseteq \Phi$ für die gilt $\Phi_0 \vdash_{\mathrm{seq}} \phi$.

Das ist klar, da Herleitungen endliche Bäume sind, und in jeder Sequenz nur endlich viele Terme vorkommen.

Korrektheit und Vollständigkeit

Sei Φ eine (möglicherweise unendliche) Menge von Formeln. Dann bedeuten

- Korrektheit eines Beweissystems

$$\Phi \vdash_{\text{seq}} \phi \quad \text{impliziert} \quad \Phi \models \phi$$

- Vollständigkeit eines Beweissystems

$$\Phi \models \phi \quad \text{impliziert} \quad \Phi \vdash_{\text{seq}} \phi$$

Gödelscher Vollständigkeitssatz

Theorem (Gödelscher Vollständigkeitssatz)

Sei $\Phi \subseteq \mathcal{L}$ eine Menge von Formeln und $\phi \in \mathcal{L}$ eine Formel. Dann gilt

$$\Phi \vdash_{\text{seq}} \phi \quad \textit{genau dann, wenn} \quad \Phi \models \phi$$

Wir geben später eine Beweisskizze für die Vollständigkeit des Resolutionskalküls, für den Sequenzenkalkül findet man den Beweis z.B. im Buch „Basic Proof Theory“ von Troelstra und Schwichtenberg.

Aus dem Vollständigkeitssatz und dem Endlichkeitssatz folgt sofort

Theorem (Endlichkeitssatz (für das Folgern))

Wenn $\Phi \models \phi$ gilt, so gibt es eine endliche Teilmenge Φ_0 für die $\Phi_0 \models \phi$ gilt.

Aus dem Endlichkeitssatz für das Folgern ergibt sich

Theorem (Kompaktheitssatz)

Eine Formelmenge Φ ist genau dann erfüllbar, wenn jede endliche Teilmenge von Φ erfüllbar ist.

Ist eine endliche Teilmenge $\Phi_o \subseteq \Phi$ unerfüllbar, so ist offensichtlich auch Φ unerfüllbar. Ist umgekehrt Φ unerfüllbar, d.h., $\Phi \models \bot$, dann gibt es eine endliche Teilmenge $\Phi_0 \subseteq \Phi$ für die gilt $\Phi_0 \models \bot$ und die Aussage ist gezeigt.

Überabzählbarkeit

Eine Menge M heißt abzählbar, wenn es eine surjektive Abbildung der natürlichen Zahlen in N, $f : \mathbb{N} \to N$, gibt, wenn also die Elemente von M in der Form $M = \{a_i \mid 0 \leq i \leq n\}$ oder $M = \{a_i \mid i \in \mathbb{N}\}$ geschrieben werden können.

Es gibt Mengen, die nicht abzählbar sind. In der Analysis zeigt man z.B., dass die Menge der reeleen Zahlen $\mathbb{R}$ überabzählbar ist. Anschaulich gesprochen bedeutet das, dass es mehr reelle Zahlen als natürliche Zahlen gibt.

Eine Theorie T ist eine deduktiv abgeschlossene Menge von Formeln, d.h., gilt $T \vdash_{\text{seq}} \phi$, so ist $\phi \in T$. Eine Theorie ist vollständig, wenn es keine konsistente Erweiterung von T gibt, d.h., wenn für jede Formel ϕ gilt $T \vdash_{\text{seq}} \phi$ oder $T \vdash_{\text{seq}} \neg\phi$.

Theorem (Satz von Löwenheim-Skolem)

Eine abzählbar konsistente Theorie hat immer auch ein abzählbares Modell.

Der Beweis dieses Satzes ergibt sich aus der im Beweis des Vollständigkeitssatzes verwendeten Konstruktion: man konstruiert dabei ein Termmodell, d.h., ein Modell dessen Elemente nur aus einer Menge von Konstanten und der Anwendung von Funktionssymbolen auf diese Konstanten bestehen. Da die Menge der dabei eingeführten Konstantensymbole abzählbar ist, ergibt sich die Aussage.

Das Paradoxon von Löwenheim-Skolem

In der Prädikatenlogik lässt sich die Mengenlehre axiomatisieren, z.B. durch das Axiomensystem von Zermelo-Fraenkel mit Auswahlaxiom, ZFC. In diser Theorie lässt sich die Existenz von überabzählbaren Mengen leicht beweisen.

Wie passt das mit dem Satz von Löwenheim-Skolem zusammen?

Die Erklärung ist, dass Begriffe wie „Abzählbarkeit“ innerhalb und außerhalb der Theorie unterschiedlich interpretiert werden: Ein abzählbares Modell $\mathcal{M}$ von ZFC enthält nicht alle Funktionen, sondern nur so viele, dass die Axiome von ZFC erfüllt werden. Die von außen sichtbaren surjektiven Abbildungen zwischen $\mathbb{N}^{\mathcal{M}}$ nach $\mathbb{R}^{\mathcal{M}}$ sind nicht im Modell $\mathcal{M}$ enthalten.

Aufzählbarkeit, Entscheidbarkeit

Im Gegensatz zur Aussagenlogik ist die Prädikatenlogik nicht entscheidbar. Es gilt:

- Die Sätze einer axiomatisierbaren Theorie T sind effektiv aufzählbar, d.h., es gibt einen Algorithmus der alle Sätze von T der Reihe nach erzeugt
- Eine vollständige axiomatisierbare Theorie T ist entscheidbar, d.h., es gibt einen Algorithmus, der für jede Formel ϕ aus $\mathcal{L}(T)$ in endlicher Zeit bestimmen kann ob $\phi \in T$ oder $\phi \notin T$ gilt
- Es gibt unendscheidbare axiomatisierbare Theorien T, d.h., für manche Theorien gibt es keinen Algorithmus der zu einer vorgegebenen Formel ϕ in endlicher Zeit bestimmen kann ob $\phi \in T$ oder $\phi \notin T$ gilt

Alternative Syntax: PVS

Terme:

$t \in \mathcal{T}$	$t' \in$ PVS
x	`x`
c	`c`
$f(t_1, \ldots, t_n)$	`f`$(t'_1, \ldots, t'_n)$

Formeln:

$\xi \in \mathcal{L}$	$\xi' \in$ PVS
$P(t_1, \ldots, t_n)$	`P`$(t'_1, \ldots, t'_n)$
$t_1 = t_2$	t'_1 `=` t'_2
$\neg\phi$	`not`(ϕ')
$\phi \wedge \psi$	ϕ' `and` ψ', ϕ' `&` ψ'
$\phi \vee \psi$	ϕ' `or` ψ'
$\phi \Rightarrow \psi$	ϕ' `implies` ψ', ϕ' `=>` ψ'
$\phi \Leftrightarrow \psi$	ϕ' `iff` ψ', ϕ' `<=>` ψ'
$\forall x.\phi$	`forall (x:A):` ϕ'
$\exists x.\phi$	`exists (x:A):` ϕ'

Beispiel

$\xi \in \mathcal{L}$	$\xi' \in$ Snark
$M(\text{Plato})$	`M(plato)`
$M(x)$	`M(x)`
$\forall x.M(x) \Rightarrow S(x)$	`forall (x:A): M(x) => S(x)`
$\forall x.M(x) \Rightarrow M(\text{mutter}(x))$	`forall (x:A): M(x) => S(mutter(x))`

PVS verwendet eine mehrsortige (typisierte) Logik. Daher muss bei jeder gebundenen Variable der Typ der Variable angegeben werden. Man kann eine einsortige (ungetypte) Logik simulieren, indem man einen Typ, z.B. `A`, für alle Variablen verwendet.

PVS

PVS (Prototype Verification System) enthält einen interaktiven Theorembeweiser, der den besprochenen Sequenzenkalkül implementiert. Das PVS-System ist Open Source und kann von der URL `http://pvs.csl.sri.com/` heruntergeladen werden.

In PVS werden Axiome und zu beweisende Aussagen in einer *Theorie* zusammengefasst. Eine Theorie enthält

- Deklarationen von Sorten (Typen)
- Deklarationen von Variablen und Konstanten
- Deklarationen und Definitionen von Funktionen
- Axiome
- Zu beweisende Aussagen

Beispiel

```
family: THEORY
 BEGIN

  person: TYPE+

  betty, carol, joe: person

  c, f, m, p, q, r, x, y, z: VAR person

  father(p): person
  mother(p): person

  ...
 END family
```

Beispiel

```
parent(p, c): bool
is_male(p): bool
is_female(p): bool

all_names_are_different: AXIOM betty /= carol
  & betty /= joe & carol /= joe

betty_is_parent_of_carol: AXIOM parent(betty, carol)
joe_is_father_of_carol: AXIOM father(carol) = joe
```

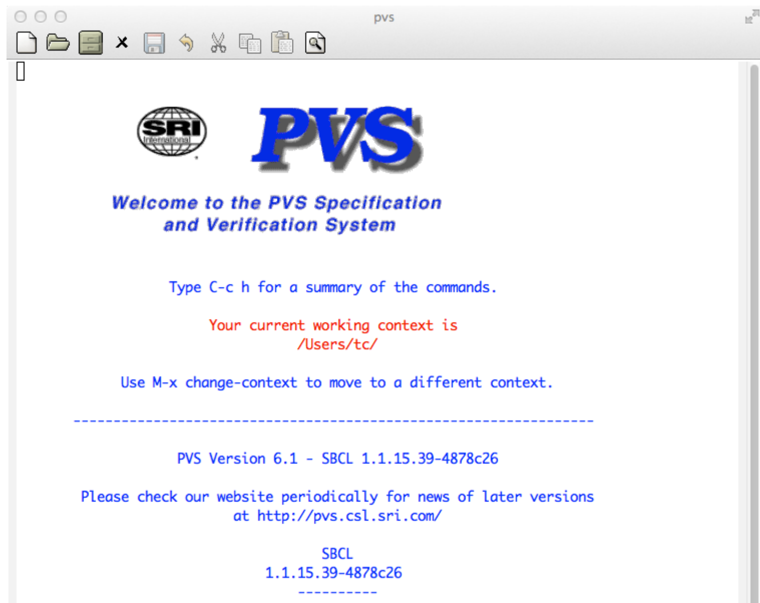
Beispiel

```
father_is_parent: AXIOM parent(father(c), c)
mother_is_parent: AXIOM parent(mother(c), c)

father_is_male: AXIOM
  (exists c: f = father(c)) => is_male(f)
mother_is_female: AXIOM
  (exists c: m = mother(c)) => is_female(m)
male_is_not_female: AXIOM
  is_male(x) <=> not(is_female(x))

parent_is_either_father_or_mother: AXIOM
  parent(p, c) <=> (p = father(c) OR p = mother(c))
father_is_not_mother: AXIOM
  father(c) /= mother(c)
```

Beispiel

```
joe_is_male: PROPOSITION
  is_male(joe)

betty_is_mother_of_carol: PROPOSITION
  mother(carol) = betty
```

PVS Emacs



PVS Context

- Alle Interaktionen mit PVS erfolgen innerhalb eines Kontexts
- Der Kontext enthält (Hilfs-)Theorien, angefangene Beweise, etc.
- Am Anfang jeder PVS-Session muss mit `M-x change-context` in den richtigen Kontext gewechselt werden

Interaktives Beweisen

- Nach dem Laden einer PVS-Theorie kann man sie auf syntaktische und Typfehler überprüfen und die darin enthaltenen Aussagen interaktiv beweisen
- Der Theorembeweiser wird mit dem Kommando `M-x prove-theory` für die komplette Theorie aufgerufen
- Falls im Kontext schon vorhergehende Beweise oder Beweisversuche existieren können diese nochmals ausgeführt werden
- Mit `M-x prove` oder `C-c p` kann der interaktive Theorembeweiser aufgerufen werden.
- Die zu beweisende Sequenz wird in einem Fenster angezeigt, Beweiskommandos werden im gleichen Fenster eingegeben:

Beweis in PVS

```
Installing rewrite rule sets.singleton_rew (all instances)
joe_is_male :

  |-------
{1}   is_male(joe)

Rule? (lemma "joe_is_father_of_carol")
Applying joe_is_father_of_carol
this simplifies to:
joe_is_male :

{-1}  father(carol) = joe
  |-------
[1]   is_male(joe)
```

Beweis in PVS

```
joe_is_male :

{-1}  father(carol) = joe
  |-------
[1]   is_male(joe)

Rule? (lemma "father_is_male")
Applying father_is_male
this simplifies to:
joe_is_male :

{-1}  FORALL (f: person):
          (EXISTS c: f = father(c)) => is_male(f)
[-2]  father(carol) = joe
  |-------
[1]   is_male(joe)
```

Beweis in PVS

```
joe_is_male :

{-1}  FORALL (f: person):
          (EXISTS c: f = father(c)) => is_male(f)
[-2]  father(carol) = joe
  |-------
[1]   is_male(joe)

Rule? (inst?)
Found substitution:
f: person gets joe,
Using template: is_male(f)
Instantiating quantified variables,
this simplifies to:
```

Beweis in PVS

```
joe_is_male :

{-1}  FORALL (f: person):
          (EXISTS c: f = father(c)) => is_male(f)
[-2]  father(carol) = joe
  |-------
[1]   is_male(joe)

Rule? (inst?)
...
joe_is_male :

{-1}  (EXISTS c: joe = father(c)) => is_male(joe)
[-2]  father(carol) = joe
  |-------
[1]   is_male(joe)
```

Beweis in PVS

```
joe_is_male :

{-1}  (EXISTS c: joe = father(c)) => is_male(joe)
[-2]  father(carol) = joe
  |-------
[1]   is_male(joe)

Rule? (split)
Splitting conjunctions,
this yields  2 subgoals:
```

Beweis in PVS

```
joe_is_male.1 :

{-1}  is_male(joe)
[-2]  father(carol) = joe
  |-------
[1]   is_male(joe)

which is trivially true.

This completes the proof of joe_is_male.1.
```

Beweis in PVS

```
joe_is_male.2 :

[-1]  father(carol) = joe
  |-------
{1}   EXISTS c: joe = father(c)
[2]   is_male(joe)

Rule? (inst?)
Found substitution:
c gets carol,
Using template: father(c)
Instantiating quantified variables,
this simplifies to:
```

Beweis in PVS

```
joe_is_male.2 :

[-1]  father(carol) = joe
  |-------
{1}   EXISTS c: joe = father(c)
[2]   is_male(joe)

Rule? (inst?)
...
joe_is_male.2 :

[-1]  father(carol) = joe
  |-------
{1}   joe = father(carol)
[2]   is_male(joe)
```

Beweis in PVS

```
joe_is_male.2 :

[-1]  father(carol) = joe
  |-------
{1}   joe = father(carol)
[2]   is_male(joe)

Rule? (grind)
Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of joe_is_male.2.
Q.E.D.

Run time  = 0.44 secs.
Real time = 104.163 secs.
NIL
*
```

Beweis in PVS

```
Installing rewrite rule sets.singleton_rew (all instances)
betty_is_mother_of_carol :

  |-------
{1}   mother(carol) = betty

Rule? (auto-rewrite-theory "family")
Rewriting relative to the theory: family,
this simplifies to:
```

Beweis in PVS

```
betty_is_mother_of_carol :

  |-------
[1]   mother(carol) = betty

Rule? (lemma "betty_is_parent_of_carol")
Applying betty_is_parent_of_carol
this simplifies to:
betty_is_mother_of_carol :

{-1}  parent(betty, carol)
  |-------
[1]   mother(carol) = betty
```

Beweis in PVS

```
betty_is_mother_of_carol :

{-1}  parent(betty, carol)
  |-------
[1]   mother(carol) = betty

Rule? (lemma "parent_is_either_father_or_mother")
Applying parent_is_either_father_or_mother
this simplifies to:
betty_is_mother_of_carol :

{-1}  FORALL (c, p: person):
        parent(p, c) <=> (p = father(c) OR p = mother(c))
[-2]  parent(betty, carol)
  |-------
[1]   mother(carol) = betty
```

Beweis in PVS

```
betty_is_mother_of_carol :

{-1}  FORALL (c, p: person):
        parent(p, c) <=> (p = father(c) OR p = mother(c))
[-2]  parent(betty, carol)
  |-------
[1]   mother(carol) = betty

Rule? (inst?)
Found substitution:
c: person gets carol,
p: person gets betty,
Using template: parent(p, c)
Instantiating quantified variables,
this simplifies to:
```

Beweis in PVS

```
betty_is_mother_of_carol :

{-1}  FORALL (c, p: person):
        parent(p, c) <=> (p = father(c) OR p = mother(c))
[-2]  parent(betty, carol)
  |-------
[1]   mother(carol) = betty

Rule? (inst?) ...
betty_is_mother_of_carol :

{-1}  parent(betty, carol) <=>
       (betty = father(carol) OR betty = mother(carol))
[-2]  parent(betty, carol)
  |-------
[1]   mother(carol) = betty
```

Beweis in PVS

```
betty_is_mother_of_carol :
{-1}  parent(betty, carol) <=>
        (betty = father(carol) OR betty = mother(carol))
[-2]  parent(betty, carol)
  |-------
[1]   mother(carol) = betty

Rule? (lemma "father_is_not_mother") ...

betty_is_mother_of_carol :
{-1}  FORALL (c: person): father(c) /= mother(c)
[-2]  parent(betty, carol) <=>
        (betty = father(carol) OR betty = mother(carol))
[-3]  parent(betty, carol)
  |-------
[1]   mother(carol) = betty
```

Beweis in PVS

```
{-1}  FORALL (c: person): father(c) /= mother(c)
[-2]  parent(betty, carol) <=>
       (betty = father(carol) OR betty = mother(carol))
[-3]  parent(betty, carol)
  |-------
[1]   mother(carol) = betty

Rule? (inst?) ...

{-1}  father(carol) /= mother(carol)
[-2]  parent(betty, carol) <=>
       (betty = father(carol) OR betty = mother(carol))
[-3]  parent(betty, carol)
  |-------
[1]   mother(carol) = betty
```

Beweis in PVS

```
{-1}  father(carol) /= mother(carol)
[-2]  parent(betty, carol) <=>
       (betty = father(carol) OR betty = mother(carol))
[-3]  parent(betty, carol)
  |-------
[1]   mother(carol) = betty

Rule? (lemma "all_names_are_different") ...

{-1}  betty /= carol & betty /= joe & carol /= joe
[-2]  father(carol) /= mother(carol)
[-3]  parent(betty, carol) <=>
       (betty = father(carol) OR betty = mother(carol))
[-4]  parent(betty, carol)
  |-------
[1]   mother(carol) = betty
```

Beweis in PVS

```
betty_is_mother_of_carol :

{-1}  betty /= carol & betty /= joe & carol /= joe
[-2]  father(carol) /= mother(carol)
[-3]  parent(betty, carol) <=>
       (betty = father(carol) OR betty = mother(carol))
[-4]  parent(betty, carol)
  |-------
[1]   mother(carol) = betty

Rule? (grind)
Trying repeated skolemization, instantiation, and if-lifting,
Q.E.D.
```

Wichtige Beweiskommandos

- `(flatten)`: Führt Regeln aus, die den Beweisbaum nicht weiter aufspalten
- `(split)`: Führt Regeln aus, die den Beweis in mehrere Äste aufspalten
- `(prop)`: Führt propositionale Vereinfachung durch
- `(grind)`: Leistungsfähigste Strategie, kann auch viele nicht-propositionale Theoreme beweisen
- `(lemma)`: Führt ein Lemma im Antezendens ein
- `(inst?)`: Versucht einen Existenzquantor zu Instanziieren (Allquantor im Antezendens, genauer Quantor mit existenzieller Stärke)
- `(skolem!)`: Einführung von Skolem-Konstanten
- `(skosimp)`: `(skolem!)` gefolgt von `(flatten)`

PVS

Wie man schon an diesem kleinen Beispiel sieht, sind Beweise mit PVS relativ aufwändig. Man kann die manuelle Arbeit manchmal durch die Definition geeigneter Beweisstrategien reduzieren. Für manche Theorien können Beweise auch durch die in PVS eingebauten Entscheidungsprozeduren automatisiert werden.

Im Folgenden wollen wir einen Theorembeweiser betrachten, der ohne jegliche Benutzerinteraktion arbeitet.

Alternative Syntax: Snark/Poem/KIF

Terme:

$t \in \mathcal{T}$	$t' \in$ Snark
x	`?x`, `x`
c	`c`
$f(t_1, \ldots, t_n)$	(f $t'_1 \ldots t'_n$)

Formeln:

$\xi \in \mathcal{L}$	$\xi' \in$ Snark
$P(t_1, \ldots, t_n)$	(p $t'_1 \ldots t'_n$)
$t_1 = t_2$	(= t'_1 t'_2)
$\neg\phi$	(not ϕ')
$\phi \wedge \psi$	(and ϕ' ψ')
$\phi \vee \psi$	(or ϕ' ψ')
$\phi \Rightarrow \psi$	(implies ϕ' ψ')
$\phi \Leftrightarrow \psi$	(iff ϕ' ψ')
$\forall x.\phi$	(forall (?x) ϕ'), (forall (x) ϕ')
$\exists x.\phi$	(exists (?x) ϕ'), (exists (x) ϕ')

Beispiel

$\xi \in \mathcal{L}$	$\xi' \in$ Snark
$M(\text{Plato})$	`(m plato)`
$M(x)$	`(m ?x)`

$\forall x.M(x) \Rightarrow S(x)$	`(forall (?x)`
	`  (implies (m ?x) (s ?x)))`
	oder
	`(forall (x)`
	`  (implies (m x) (s x)))`

$\forall x.M(x) \Rightarrow M(\text{mutter}(x))$	`(forall (x)`
	`  (implies (m x)`
	`           (m (mutter x))))`

Variablen können immer in der Form `?x` geschrieben werden. Bei quantifizierten Variablen kann das führende Fragezeichen auch weggelassen werden.

Snark

Snark (SRI's New Automated Reasoning Kit) ist ein automatischer Theorembeweiser für Prädikatenlogik. Im Gegensatz zu PVS versucht Snark einen Beweis ohne Interaktion mit dem Benutzer zu finden.

Snark hat keinen „Theoriebegriff"; man führt Axiome durch `assert`-Anweisungen ein und startet den Beweis durch Eingabe von `(prove)` oder `(new-prove)`.

Die Homepage von Snark ist `http://www.ai.sri.com/~stickel/snark.html`, ein (leider etwas veraltetes) Tutorial findet man unter `http://www.ai.sri.com/snark/tutorial/tutorial.html`. Eine Version, die etwas leichter zu Installieren ist, ist unter `https://github.com/hoelzl/Snark` zu finden.

Beweis mit Snark

```
(assert '(and (/= betty carol)
              (/= betty joe)
              (/= carol joe)))

(assert '(parent betty carol)
        :name 'betty-is-parent-of-carol)

(assert '(= (father carol) joe)
        :name 'joe-is-father-of-carol)

(assert '(parent (father ?c) ?c)
        :name 'father-is-parent)

(assert '(parent (mother ?c) ?c)
        :name 'mother-is-parent)
```

Beweis mit Snark

```
(assert '(implies (exists (?c) (= ?m (mother ?c)))
                  (is-female ?m))
        :name 'mother-is-female)

(assert '(iff (is-male ?x) (not (is-female ?x))))

(assert '(iff (parent ?p ?c)
              (or (= ?p (father ?c)) (= ?p (mother ?c)))))

(assert '(/= (father ?c) (mother ?c))
        :name 'father-is-not-mother)
```

Interaktion mit Snark

Snark ist in eine Common Lisp Programmierumgebung eingebettet und bietet verschiedene Reasoner an. Vor der Verwendung von Snark muss man angeben, welche Reasoner man verwenden will. Wir verwenden hier Resolution (zum rein logischen Schließen) und Paramodulation (zur Behandlung von Gleichheiten).

Man kann die Theorie direkt in die Kommandozeile eingeben oder von einer Datei laden. Besser ist es aber, Funktionen zu schreiben, die die Initialisierung von Snark und das Laden der Theorie übernehmen:

Interaktion mit Snark

```
(defun init ()
  (initialize)
  ;; Optionen für die Ausgabe der Lösung...
  (use-resolution)
  (use-paramodulation))

(defun set-up-family-theory ()
  (init)
  ...
  (assert '(/= (father ?c) (mother ?c))
          :name 'father-is-not-mother))
```

Interaktion mit Snark

```
(defun prove-joe-is-male ()
  (set-up-family-theory)
  (prove '(is-male joe)))
```

Interaktion mit Snark

```
CL-USER> (require :snark)
NIL
CL-USER> (in-package :snark-user)
#<PACKAGE "SNARK-USER">
SNARK-USER> (load "family.lisp")
T
SNARK-USER> (prove-joe-is-male)
```

Interaktion mit Snark

```
SNARK-USER> (prove-joe-is-male)
; Running SNARK from /Users/tc/Prog/Lisp/Hacking/Iliad/Libraries/Snark/ in
(Refutation
(Row JOE-IS-FATHER-OF-CAROL
   (= (FATHER CAROL) JOE)
   ASSERTION)
(Row FATHER-IS-MALE
   (OR (NOT (= ?X (FATHER ?Y))) (IS-MALE ?X))
   ASSERTION)
(Row 16
   (NOT (IS-MALE JOE))
   NEGATED_CONJECTURE)
(Row 20
   (IS-MALE JOE)
   (RESOLVE FATHER-IS-MALE JOE-IS-FATHER-OF-CAROL))
(Row 21
   FALSE
   (REWRITE 16 20)))
:PROOF-FOUND
SNARK-USER>
```

Mehrsortige Logik

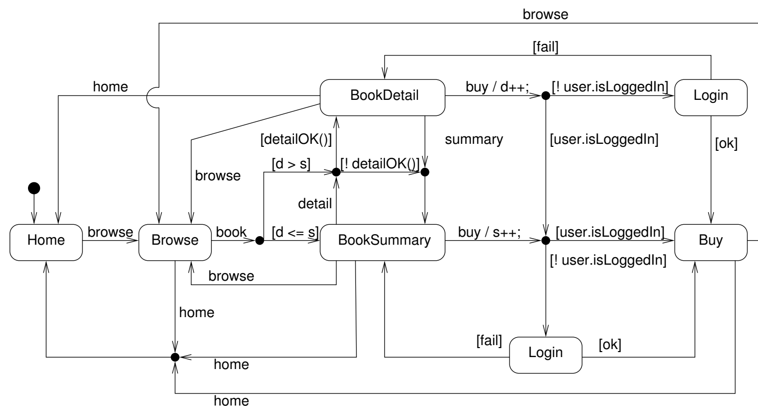
Ähnlich wie bei Programmiersprachen kann man in der Prädikatenlogik verschiedene Typen von Objekten unterscheiden. In der Logik wird statt Typen meist der Begriff *Sorten* verwendet.

Mehrsortige Logik

```
(declare-sort 'person)
(declare-sort 'animal)

(declare-sorts-incompatible 'person 'animal)

(declare-constant 'betty :sort 'person)
(declare-constant 'carol :sort 'person)
(declare-constant 'fido :sort 'animal)

(declare-function 'father 1 :sort '(person person))
(declare-function 'mother 1 :sort '(person person))
(declare-relation 'parent 2 :sort '(person person))

(declare-relation 'human 1)
(declare-relation 'animal 1)
```

```
SNARK-USER> (new-prove '(parent carol fido))
; Evaluation aborted on #<SIMPLE-ERROR
       "Atomic formula ~A is not well sorted." {1006CC7233}>.

Atomic formula (PARENT CAROL FIDO) is not well sorted.
   [Condition of type SIMPLE-ERROR]

Restarts:
 0: [RETRY] Retry SLIME REPL evaluation request.
 1: [*ABORT] Return to SLIME's top level.
 2: [ABORT] Abort thread (#<THREAD "repl-thread" RUNNING {1002
```

Beispiel: Konflikt in State Machines



Beispiel: Konflikt in State Machines

```
;;; M1
;;; -->  a -----> b ----> goto w
;;; |       |----> c ----> goto x
;;; |       |-> ~b & ~c -> goto y
;;; |-------------------> goto z

;;; M2
;;; -->  d -----> e ----> goto w
;;; |       |----> f ----> goto x
;;; |       |-> ~e & ~f -> goto y
;;; |-------------------> goto z
```

Beispiel: Konflikt in State Machines

```
...
(declare-relation 'm1 2)
(declare-relation 'm2 2)
(declare-relation 'a 1)

...

(assert '(implies (and (a ?state) (b ?state)) (m1 ?state w))
        :name :ab->w)

(assert '(implies (and (a ?state) (c ?state)) (m1 ?state x))
        :name :ac->x)
...
```