Rechnergestützte Beweissysteme

Kalküle für die Aussagenlogik

Aussagenlogik

1. Aufbau logischer Sprachen
   – Aussagenlogik
   – Prädikatenlogik (erster Stufe)
   – Logik höherer Stufen
2. Die Sprache der Aussagenlogik
   – Syntax, Semantik
   – Belegung, Wahrheit, Tautologien
   – Kalküle, Beweise
3. Hilbert Kalkül
   – Beweise im Hilbert Kalkül
4. Sequenzenkalkül
   – Korrektheit, Vollständigkeit
   – Der Sequenzenkalkül in PVS
5. Resolventenmethode
   – Korrektheit, Vollständigkeit

Aussagenlogik

1. Aufbau logischer Sprachen
   – Aussagenlogik
   – Prädikatenlogik (erster Stufe)
   – Logik höherer Stufen
2. Die Sprache der Aussagenlogik
   – Syntax, Semantik
   – Belegung, Wahrheit, Tautologien
   – Kalküle, Beweise
3. Hilbert Kalkül
   – Beweise im Hilbert Kalkül
4. Sequenzenkalkül
   – Korrektheit, Vollständigkeit
   – Der Sequenzenkalkül in PVS
5. Resolventenmethode
   – Korrektheit, Vollständigkeit

Aufbau logischer Aussagen(1)

- Logische Aussagen vereinigen mehrere Sprachebenen.

  Ein typischer logischer Ausdruck ist:

  $$\forall n \in \mathbf{N}.\ \forall d \in \mathbf{N}.\ \exists k \in \mathbf{N}.\ \exists r \in \mathbf{N}.\ \ 0 \le r \wedge r < d \ \wedge\ n = k*d+r$$

- **Aussagenlogik** (engl.: *propositional logic*)
  Hier geht es nur um die logischen Operatoren $\wedge$, $\vee$, $\neg$, $\rightarrow$, $\Leftrightarrow$
  und deren Bedeutung. Aussagenlogisch können wir z.B. schließen, dass wir innere Teile umstellen können, z.B.

- $r < d \wedge n=k*d+r \wedge 0 \le r$ oder $\neg\,(0 \le r \rightarrow\ \neg\, r < d\,) \wedge n=k*d+r$

Aufbau logischer Aussagen(2)

- **Prädikatenlogik** (engl.: predicate logic)
  In dieser Sprache redet man über mathematische Strukturen mit
  - Relationen ($\leq$, $<$, $=$, $\in$, ...)
  - Operationen ($+$, $*$, succ, sin, ..)
  - Konstanten (0, 1, π, ...)
- Man hat *Variablen* für **Elemente** der Struktur und *Quantoren* um über alle oder einige etwas auszusagen:
  - x,y,z,a,b, d, n, ...
  - $\forall n \in N.$ $\exists k \in N.$, $\exists x \in \Re.$, ...
- Die Prädikatenlogik umfasst die Aussagenlogik.

Aufbau logischer Aussagen(3)

- In der **Logik höherer Stufe** hat man auch Variablen und Quantoren für Teilmengen：

$$\forall \mathbf{P} \subseteq \mathbb{N}.\ (0 \in P \wedge \forall k \in \mathbb{N}.\ k \in P \to k+1 \in P\ ) \ \to\ P = \mathbb{N}.$$

- Dieses Induktionsaxiom ist ein typischer Ausdruck der Logik zweiter Stufe. Es ist zwingend notwendig um die natürlichen Zahlen eindeutig zu beschreiben.
- Für andere *Datentypen* (Stacks, Listen, Bäume, etc.) hat man analoge Induktionsaxiome
  - Stack-Induktion,
  - Listen-Induktion, etc.

Arithmetik – Logik der Zahlen

Als *Arithmetik* bezeichnet man die Theorie der natürlichen Zahlen mit den Operationen und Relationen

$$0, 1, +, *, <$$

sowie dem *Induktionsaxiom*:

$$\forall P \subseteq \mathbb{N}. \, (0 \in P \land \forall n \in \mathbb{N}. \, n \in P \rightarrow n+1 \in P) \rightarrow P = \mathbb{N}$$

- Das Induktionsaxiom passt nicht in die Prädikatenlogik erster Stufe. Es wird über Teilmengen quantifiziert !
- Gödel: Die wahren Aussagen der Arithmetik sind *nicht aufzählbar* , also auch nicht entscheidbar.
- Eine Aussage Q der Arithmetik ist entweder wahr oder ihre Negation $\neg Q$ ist wahr. Daher würde aus der Aufzählbarkeit schon die Entscheidbarkeit folgen.

Vollautomatisches Beweisen ?

- Allgemein für die Prädikatenlogik unmöglich !

  Gödel: In der Prädikatenlogik erster Stufe ist die Menge aller Folgerungen aus einem System von Axiomen aufzählbar, i.A. aber nicht entscheidbar.

$\Rightarrow$ Es ist möglich, ein Programm zu konstruieren, das alle Aussagen ausdruckt, welche aus den Axiomen folgen. Es ist aber **nicht** möglich, ein Programm zu schreiben welches **entscheidet**, ob eine beliebige prädikatenlogische Aussage aus den Axiomen folgt.

Insbesondere: Die Menge aller **allgemeingültigen** Aussagen der Prädikatenlogik ist *aufzählbar*, aber *nicht entscheidbar*.

Vorsicht: Wenn P eine prädikatenlogische Aussage ist, so kann es sein, dass weder P noch $\neg$ P allgemeingültig ist.

- $(\exists y.\forall x.R(x,y)) \rightarrow \forall x.\exists y.\ R(x,y)$ ist allgemeingültig
- Weder $\exists y.\forall x.R(x,y)$ noch $\forall y.\exists x.\neg R(x,y)$ sind allgemeingültig

Aussagenlogik

1. Aufbau logischer Sprachen
   - Aussagenlogik
   - Prädikatenlogik (erster Stufe)
   - Logik höherer Stufen
2. Die Sprache der Aussagenlogik
   - Syntax, Semantik
   - Belegung, Wahrheit, Tautologien
   - Kalküle, Beweise
3. Hilbert Kalkül
   - Beweise im Hilbert Kalkül
4. Sequenzenkalkül
   - Korrektheit, Vollständigkeit
   - Der Sequenzenkalkül in PVS
5. Resolventenmethode
   - Korrektheit, Vollständigkeit

Die Sprache der Aussagenlogik

- *Variablen* ::= p, q, r, ...
- *Konstante* ::= $\perp$ | T
- $$
\begin{array}{lll}
E & ::= & \mathit{Variable} \\
 & | & \mathit{Konstante} \\
 & | & E \wedge E \\
 & | & E \vee E \\
 & | & E \rightarrow E \\
 & | & E \leftrightarrow E \\
 & | & \neg E \\
 & | & (E)
\end{array}
$$
- Beispiel: $\neg(p \wedge q) \Leftrightarrow ((q \rightarrow \perp) \vee (p \rightarrow \perp))$

Wahrheitswerte

- Auf den Wahrheitswerten $2=\{0,1\}$ sind die bekannten Verknüpfungen festgelegt: $\bot = 0$, $T = 1$, sowie

$\neg$	
0	1
1	0

$\wedge$	0	1
0	0	0
1	0	1

$\vee$	0	1
0	0	1
1	1	1

$\rightarrow$	0	1
0	1	1
1	0	1

$\leftrightarrow$	0	1
0	1	0
1	0	1

- Man prüft nach, dass für beliebige $x, y, z \in \{0,1\}$ gilt:
  - $x \vee y = \neg(\neg x \wedge \neg y)$, $x \rightarrow y = \neg(x \wedge \neg y)$, $x \leftrightarrow y = (x \rightarrow y) \wedge (y \rightarrow x)$
  - $\neg x = x \rightarrow 0$, $x \vee y = (x \rightarrow 0) \rightarrow y$, $(x \wedge y) = (x \rightarrow (y \rightarrow 0)) \rightarrow 0$,
- Man kann also z.B. aus $\wedge$ und $\neg$, oder aus $\rightarrow$ und 0 alle übrigen Verknüpfungen aufbauen.

Belegungen

- Eine *Belegung* einer Menge V von aussagenlogischen Variablen ist eine beliebige Abbildung $\phi : V \rightarrow \{0,1\}$

- Der *Wahrheitswert* $|E|_\phi$ eines aussagenlogischen Ausdrucks E unter der Belegung ϕ wird induktiv definiert:

  - $|p|_\phi = \phi(p)$, falls p aussagenlogische Variable ist.
  - $| T |_\phi = 1$ und $| \perp |_\phi = 0$
  - $| E_1 \wedge E_2 |_\phi = | E_1 |_\phi \wedge | E_2 |_\phi$
  - $| E_1 \vee E_2 |_\phi = | E_1 |_\phi \vee | E_2 |_\phi$
  - $| E_1 \rightarrow E_2 |_\phi = | E_1 |_\phi \rightarrow | E_2 |_\phi$
  - $| E_1 \leftrightarrow E_2 |_\phi = | E_1 |_\phi \leftrightarrow | E_2 |_\phi$
  - $|\neg E |_\phi = \neg | E |_\phi$

Erfüllbarkeit - Tautologie

- Sei *E* ein Ausdruck der Aussagenlogik. V sei die Menge aller Variablen in E. Dann heißt E
  - *erfüllbar*, wenn es eine Belegung $\phi: V \to \{0,1\}$ *gibt* mit $| E |_\phi = 1$
  - *Tautologie*, falls für jede Belegung $\phi: V \to \{0,1\}$ gilt: $| E |_\phi = 1$

- Für jeden Ausdruck E gilt:
  E ist Tautologie, gdw. $\neg$E nicht erfüllbar.

- Für einen Ausdruck mit n aussagelogischen Variablen hat man 2^n viele Belegungen zu testen. Das *Erfüllbarkeitsproblem* (SAT) ist *NP-vollständig* (Cook)

Kalküle

- Unter einem *Kalkül* für eine Logik versteht man eine aufzählbare Menge von *Axiomen* und *Schlussregeln*.
  - Jedes *Axiom* ist eine Formel
  - Jede *Schlussregel* R legt fest, wie man aus einer endlichen Menge $F_1, \dots, F_n$ von Formeln eine neue Formel G gewinnen kann.
  - Es muss entscheidbar sein, ob eine Schlussregel korrekt angewendet wurde.
- Üblicherweise hat man endlich viele Axiome und Regeln.
- Axiome kann man auch als Regeln ohne Prämissen deuten.

Aussagenlogik

1. Aufbau logischer Sprachen
   - Aussagenlogik
   - Prädikatenlogik (erster Stufe)
   - Logik höherer Stufen
2. Die Sprache der Aussagenlogik
   - Syntax, Semantik
   - Belegung, Wahrheit, Tautologien
   - Kalküle, Beweise
3. Hilbert Kalkül
   - Beweise im Hilbert Kalkül
4. Sequenzenkalkül
   - Korrektheit, Vollständigkeit
   - Der Sequenzenkalkül in PVS
5. Resolventenmethode
   - Korrektheit, Vollständigkeit

Axiome des Hilbert Kalküls

Minimalistisch: Alles zurückgeführt auf Operatoren $\rightarrow$ und $\perp$

Konvention: $\rightarrow$ ist rechtsgeklammert.

Axiome:

$$\mathbf{A \rightarrow (B \rightarrow A)}$$

$$\mathbf{(A \rightarrow (B \rightarrow C)) \rightarrow (A \rightarrow B) \rightarrow (A \rightarrow C)}$$

$$\mathbf{\perp \rightarrow A}$$

Schlussregel des Hilbertkalküls:

- Die einzige Schlussregel des Hilbertkalküls ist der

  Modus Ponens:

$$\frac{P, \quad P \rightarrow Q}{Q}$$

P und Q stehen dabei für Instanzen beliebiger Formeln.

Beweise

- Ein *Beweis von* P_n ist eine Folge von Formeln

$$P_1, P_2, \dots, P_n.$$

wobei jedes P_i entweder Instanz eines Axioms ist, oder aus den P_k mit $k < i$ durch Anwendung einer Schlussregel entsteht.

<u>Beispiel</u> (Beweis von $A \to A$ im Hilbertkalkül):

(1)	$(A \to ((A \to A) \to A)) \to (A \to (A \to A)) \to (A \to A)$,	(Axiom)
(2)	$A \to ((A \to A) \to A)$,	(Axiom)
(3)	$(A \to (A \to A)) \to (A \to A)$,	(MP)
(4)	$A \to (A \to A)$,	(Axiom)
(5)	$A \to A$.	(MP)

Weitere Operatoren sind Abkürzungen

Definierende Axiome für $\neg$:

$\neg A \to (A \to \bot)$ $\qquad$ $(A \to \bot) \to \neg A$

Definierende Axiome für $\vee$:

$A \to A \vee B$ $\qquad$ $B \to A \vee B$

$(A \to C) \to (B \to C) \to (A \vee B \to C)$

Definierende Axiome für $\wedge$:

$A \wedge B \to A$ $\qquad$ $A \wedge B \to B$

$A \to B \to A \wedge B$

„Klassische Logik“ verwendet zusätzlich „Tertium non datur“ :

$A \vee \neg A$ $\qquad$ (alternativ: $\neg\neg A \to A$)

Zu kompliziert

- Beweise im Hilbert Kalkül sind zu umständlich

- Viele Kalküle versuchen das zu verbessern
  - natürliches Schließen
  - Sequenzenkalküle
  - Resolventenkalkül

- PVS benutzt den Sequenzenkalkül von G. Gentzen

Aussagenlogik

1. Aufbau logischer Sprachen
   – Aussagenlogik
   – Prädikatenlogik (erster Stufe)
   – Logik höherer Stufen
2. Die Sprache der Aussagenlogik
   – Syntax, Semantik
   – Belegung, Wahrheit, Tautologien
   – Kalküle, Beweise
3. Hilbert Kalkül
   – Beweise im Hilbert Kalkül
4. Sequenzenkalkül
   – Korrektheit, Vollständigkeit
   – Der Sequenzenkalkül in PVS
5. Resolventenmethode
   – Korrektheit, Vollständigkeit

Sequenzen: Gentzen-Kalkül

Sequenz ist Paar Γ, Δ endlicher Mengen von Formeln.
Schreibweise: $\Gamma \vdash \Delta$, manchmal:

$$\Gamma \Rightarrow \Delta.$$

Gerhard Gentzen

Notation:

Γ_1, Γ_2	statt $\Gamma_1 \cup \Gamma_2$
Γ, p	statt $\Gamma \cup \{p\}$, etc.

Ist $\Gamma = \{ H_1, \ldots, H_n \}$ und $\Delta = \{ G_1, \ldots, G_k \}$, so soll die Sequenz $\Gamma \vdash \Delta$ ausdrücken:

$$\underbrace{H_1 \wedge \ldots \wedge H_n}_{\text{Antezedenz}} \rightarrow \underbrace{G_1 \vee \ldots \vee G_k}_{\text{Sukkzedenz}}.$$

- H : hypothesis
- G : goal =Ziel

Interpretation einer Sequenz

- Man kann eine Sequenz als Disjunktion von negierten und unnegierten Formeln auffassen, denn

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

  ist logisch äquivalent zu

  $$\neg H_1 \vee \ldots \vee \neg H_n \vee G_1 \vee \ldots \vee G_k \,.$$

- Jede aussagenlogische Formel F kann in konjunktive Normalform äquivalent umgeformt werden. Dabei wird F zu einer Konjunktion von Klauseln, d.h. zu einer Konjunktion von Formeln der Bauart

  $$p_1 \vee \ldots \vee p_n \vee \neg p_{n+1} \vee \ldots \vee \neg p_k$$

  wobei die p_1 .. p_k aussagenlogische Variablen sind.

- Aus dem *Satz über die Disjunktive Normalform* folgt:

  Jede aussagenlogische Formel lässt sich
  als Menge von Sequenzen ausdrücken.

Sequenzenkalkül - Axiome

- Die Axiome des Sequenzenkalküls sind die Sequenzen der Form

  Axiom

$$\Gamma_1, p, \Gamma_2 \;\vdash\; \Delta_1, p, \Delta_2$$

d.h. alle Sequenzen, bei denen eine Ziel-Formel schon als Hypothese auftaucht.

Sequenzenkalkül - Regeln

- Der Sequenzenkalkül besitzt drei Sorten von Regeln:
  - Strukturelle Regeln
    - drücken aus, dass Antezedens und Sukkzedens als Mengen aufzufassen sind
  - Abschwächungsregeln
    - erlauben überflüssige Hypothesen und Konklusionen.
  - Logische Regeln
    - definieren die Konnektoren $\perp$, T, $\vee$, $\wedge$, $\rightarrow$, $\neg$.
- Jede Regel besteht aus zwei Varianten
  - eine bezieht sich auf den Antezedenz (linke Seite),
  - die andere auf den Sukkzedenz (rechte Seite).
- Die Regeln weisen eine Anti-Symmetrie auf
  - Analog zur Antisymmetrie von $\wedge$ und $\vee$ auf der linken und rechten Seite.

Sequenzenkalkül - Strukturregeln

Permutation

$$\frac{\Gamma_1, p, q, \Gamma_2 \;\vdash\; \Delta}{\Gamma_1, q, p, \Gamma_2 \;\vdash\; \Delta} \quad \text{(perm-L)} \qquad \frac{\Gamma \;\vdash\; \Delta_1, p, q, \Delta_2}{\Gamma \;\vdash\; \Delta_1, q, p, \Delta_2} \quad \text{(perm-R)}$$

Kontraktion

$$\frac{\Gamma_1, p, p, \Gamma_2 \;\vdash\; \Delta}{\Gamma_1, p, \Gamma_2 \;\vdash\; \Delta} \quad \text{(con-L)} \qquad \frac{\Gamma \;\vdash\; \Delta_1, p, p, \Delta_2}{\Gamma \;\vdash\; \Delta_1, p, \Delta_2} \quad \text{(con-R)}$$

Diese Regeln besagen, dass Γ und Δ als Mengen aufzufassen sind

Sequenzenkalkül-Abschwächungsregeln

- Überflüssige Hypothesen und alternative Beweisziele vereinfachen die Aufgabe:

Weakening

$$\frac{\Gamma \;\vdash\; \Delta}{\Gamma, p \;\vdash\; \Delta} \quad \text{(weak-L)} \qquad\qquad \frac{\Gamma \;\vdash\; \Delta}{\Gamma \;\vdash\; \Delta, p} \quad \text{(weak-R)}$$

Logische Regeln für $\wedge$, $\vee$

Konjunktion

$$\frac{\Gamma, p, q \vdash \Delta}{\Gamma, p \wedge q \vdash \Delta} \quad (\wedge\text{-L}) \qquad \frac{\Gamma \vdash \Delta, p \quad \Gamma \vdash \Delta, q}{\Gamma \vdash \Delta, p \wedge q} \quad (\wedge\text{-R})$$

Disjunktion

$$\frac{\Gamma, p \vdash \Delta \quad \Gamma, q \vdash \Delta}{\Gamma, p \vee q \vdash \Delta} \quad (\vee\text{-L}) \qquad \frac{\Gamma \vdash \Delta, p, q}{\Gamma \vdash \Delta, p \vee q} \quad (\vee\text{-R})$$

Logische Regeln für ¬, ⊥,T

Negation

$$\frac{\Gamma \vdash p, \Delta}{\Gamma, \neg p \vdash \Delta} \quad (\neg\text{-L})$$

$$\frac{\Gamma, p \vdash \Delta}{\Gamma \vdash \Delta, \neg p} \quad (\neg\text{-R})$$

Falsum

$$\frac{}{\Gamma, \bot \vdash \Delta} \quad (\bot)$$

Verum

$$\frac{}{\Gamma \vdash \top, \Delta} \quad (\top)$$

Logische Regeln für $\rightarrow$

Implikation

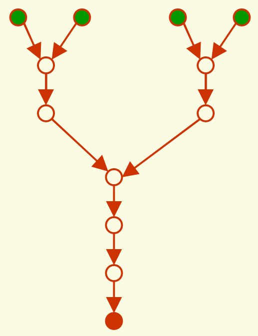
$$\frac{\Gamma, q \vdash \Delta \quad \Gamma \vdash p, \Delta}{\Gamma, p \rightarrow q \vdash \Delta} \; (\rightarrow\text{-L})$$

$$\frac{\Gamma, p \vdash q, \Delta}{\Gamma \vdash p \rightarrow q, \Delta} \; (\rightarrow\text{-R})$$

Beweise im Sequenzenkalkül

Beweis im Sequenzenkalkül als Baum:

- Blätter : Instanzen von Axiomen
- innere Knoten : Anwendungen von Regeln
- Wurzel : bewiesene Formel



$$
\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{A, B \vdash A \qquad A, B \vdash B}{A, B \vdash A \wedge B}\,(\wedge\text{-R})}{A, B \vdash A \wedge B, A \wedge C} \qquad \dfrac{\dfrac{A, C \vdash A \qquad A, C \vdash C}{A, C \vdash A \wedge C}\,(\wedge\text{-R})}{A, C \vdash A \wedge B, A \wedge C}}{A, B \vee C \vdash A \wedge B, A \wedge C}\,(\vee\text{-L})}{A, B \vee C \vdash (A \wedge B) \vee (A \wedge C)}\,(\vee\text{-R})}{A \wedge (B \vee C) \vdash (A \wedge B) \vee (A \wedge C)}\,(\wedge\text{-L})}{\vdash A \wedge (B \vee C) \rightarrow (A \wedge B) \vee (A \wedge C)}\,(\rightarrow\text{-R})
$$

Rückwärtsbeweise

- Einfacher:
  - starte mit vermuteter Formel
  - wende Regeln rückwärts an
  - bis alle blätter Instanzen von Axiomen

$$\vdash A \wedge (B \vee C) \rightarrow (A \wedge B) \vee (A \wedge C)$$

Rückwärtsbeweise

- Einfacher:
  - starte mit vermuteter Formel
  - wende Regeln rückwärts an
  - bis alle blätter Instanzen von Axiomen

$$\frac{A \wedge (B \vee C) \;\vdash\; (A \wedge B) \vee (A \wedge C)}{\vdash\; A \wedge (B \vee C) \rightarrow (A \wedge B) \vee (A \wedge C)} \;(\rightarrow\text{-R})$$

Rückwärtsbeweise

- Einfacher:
  - starte mit vermuteter Formel
  - wende Regeln rückwärts an
  - bis alle blätter Instanzen von Axiomen

$$\dfrac{\dfrac{A, B \vee C \ \vdash \ (A \wedge B) \vee (A \wedge C)}{A \wedge (B \vee C) \ \vdash \ (A \wedge B) \vee (A \wedge C)}\ (\wedge\text{-L})}{\vdash \ A \wedge (B \vee C) \rightarrow (A \wedge B) \vee (A \wedge C)}\ (\rightarrow\text{-R})$$

Rückwärtsbeweise

- Einfacher:
  - starte mit vermuteter Formel
  - wende Regeln rückwärts an
  - bis alle blätter Instanzen von Axiomen

$$\cfrac{\cfrac{\cfrac{A, B \vee C \;\vdash\; A \wedge B, A \wedge C}{A, B \vee C \;\vdash\; (A \wedge B) \vee (A \wedge C)}\;(\vee\text{-R})}{A \wedge (B \vee C) \;\vdash\; (A \wedge B) \vee (A \wedge C)}\;(\wedge\text{-L})}{\vdash\; A \wedge (B \vee C) \rightarrow (A \wedge B) \vee (A \wedge C)}\;(\rightarrow\text{-R})$$

Rückwärtsbeweise

- Einfacher:
  - starte mit vermuteter Formel
  - wende Regeln rückwärts an
  - bis alle blätter Instanzen von Axiomen

$$
\cfrac{\cfrac{\cfrac{\cfrac{A, B \vdash A \wedge B, A \wedge C \qquad\qquad A, C \vdash A \wedge B, A \wedge C}{A, B \vee C \vdash A \wedge B, A \wedge C}\;(\vee\text{-L})}{A, B \vee C \vdash (A \wedge B) \vee (A \wedge C)}\;(\vee\text{-R})}{A \wedge (B \vee C) \vdash (A \wedge B) \vee (A \wedge C)}\;(\wedge\text{-L})}{\vdash A \wedge (B \vee C) \rightarrow (A \wedge B) \vee (A \wedge C)}\;(\rightarrow\text{-R})
$$

Rückwärtsbeweise

- Einfacher:
  - starte mit vermuteter Formel
  - wende Regeln rückwärts an
  - bis alle blätter Instanzen von Axiomen

$$
\dfrac{\dfrac{\dfrac{\dfrac{(\text{weak-R})\ \dfrac{A, B \vdash A \wedge B}{A, B \vdash A \wedge B, A \wedge C} \qquad A, C \vdash A \wedge B, A \wedge C}{A, B \vee C \vdash A \wedge B, A \wedge C}\ (\vee\text{-L})}{A, B \vee C \vdash (A \wedge B) \vee (A \wedge C)}\ (\vee\text{-R})}{A \wedge (B \vee C) \vdash (A \wedge B) \vee (A \wedge C)}\ (\wedge\text{-L})}{\vdash A \wedge (B \vee C) \rightarrow (A \wedge B) \vee (A \wedge C)}\ (\rightarrow\text{-R})
$$

Rückwärtsbeweise

- Einfacher:
  - starte mit vermuteter Formel
  - wende Regeln rückwärts an
  - bis alle blätter Instanzen von Axiomen

$$
\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{A, B \vdash A \qquad A, B \vdash B}{A, B \vdash A \wedge B}\,(\wedge\text{-R})}{A, B \vdash A \wedge B, A \wedge C}\,(\text{weak-R}) \qquad A, C \vdash A \wedge B, A \wedge C}{A, B \vee C \vdash A \wedge B, A \wedge C}\,(\vee\text{-L})}{A, B \vee C \vdash (A \wedge B) \vee (A \wedge C)}\,(\vee\text{-R})}{A \wedge (B \vee C) \vdash (A \wedge B) \vee (A \wedge C)}\,(\wedge\text{-L})}{\vdash A \wedge (B \vee C) \rightarrow (A \wedge B) \vee (A \wedge C)}\,(\rightarrow\text{-R})
$$

Rückwärtsbeweise

- Einfacher:
  - starte mit vermuteter Formel
  - wende Regeln rückwärts an
  - bis alle blätter Instanzen von Axiomen

$$
\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{A, B \vdash A}\,(\text{ax}) \qquad \dfrac{}{A, B \vdash B}\,(\text{ax})}{A, B \vdash A \wedge B}\,(\wedge\text{-R})}{A, B \vdash A \wedge B, A \wedge C}\,(\text{weak-R}) \qquad A, C \vdash A \wedge B, A \wedge C}{A, B \vee C \vdash A \wedge B, A \wedge C}\,(\vee\text{-L})}{A, B \vee C \vdash (A \wedge B) \vee (A \wedge C)}\,(\vee\text{-R})}{A \wedge (B \vee C) \vdash (A \wedge B) \vee (A \wedge C)}\,(\wedge\text{-L})}{\vdash A \wedge (B \vee C) \to (A \wedge B) \vee (A \wedge C)}\,(\to\text{-R})
$$

Rückwärtsbeweise

- Einfacher:
  - starte mit vermuteter Formel
  - wende Regeln rückwärts an
  - bis alle blätter Instanzen von Axiomen

$$
\dfrac{
  \dfrac{
    \dfrac{
      \dfrac{
        \dfrac{
          \dfrac{
            \dfrac{}{A, B \vdash A}\,(\text{ax}) \quad \dfrac{}{A, B \vdash B}\,(\text{ax})
          }{A, B \vdash A \wedge B}\,(\wedge\text{-R})
        }{A, B \vdash A \wedge B, A \wedge C}\,(\text{weak-R})
        \qquad
        \dfrac{A, C \vdash A \wedge C}{A, C \vdash A \wedge B, A \wedge C}\,(\text{weak-R})
      }{A, B \vee C \vdash A \wedge B, A \wedge C}\,(\vee\text{-L})
    }{A, B \vee C \vdash (A \wedge B) \vee (A \wedge C)}\,(\vee\text{-R})
  }{A \wedge (B \vee C) \vdash (A \wedge B) \vee (A \wedge C)}\,(\wedge\text{-L})
}{\vdash A \wedge (B \vee C) \rightarrow (A \wedge B) \vee (A \wedge C)}\,(\rightarrow\text{-R})
$$

Rückwärtsbeweise

- Einfacher:
  - starte mit vermuteter Formel
  - wende Regeln rückwärts an
  - bis alle blätter Instanzen von Axiomen

$$
\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{A, B \vdash A}\,(\text{ax}) \quad \dfrac{}{A, B \vdash B}\,(\text{ax})}{A, B \vdash A \wedge B}\,(\wedge\text{-R})}{A, B \vdash A \wedge B, A \wedge C}\,(\text{weak-R}) \qquad \dfrac{\dfrac{A, C \vdash A \quad A, C \vdash C}{A, C \vdash A \wedge C}\,(\wedge\text{-R})}{A, C \vdash A \wedge B, A \wedge C}\,(\text{weak-R})}{A, B \vee C \vdash A \wedge B, A \wedge C}\,(\vee\text{-L})}{A, B \vee C \vdash (A \wedge B) \vee (A \wedge C)}\,(\vee\text{-R})}{A \wedge (B \vee C) \vdash (A \wedge B) \vee (A \wedge C)}\,(\wedge\text{-L})}{\vdash A \wedge (B \vee C) \to (A \wedge B) \vee (A \wedge C)}\,(\to\text{-R})
$$

Rückwärtsbeweise

- Einfacher:
  - starte mit vermuteter Formel
  - wende Regeln rückwärts an
  - bis alle blätter Instanzen von Axiomen

$$
\dfrac{
  \dfrac{
    \dfrac{
      \dfrac{\dfrac{}{A, B \vdash A}\,(ax) \quad \dfrac{}{A, B \vdash B}\,(ax)}{A, B \vdash A \wedge B}\,(\wedge\text{-R})
    }{A, B \vdash A \wedge B, A \wedge C}\,(\text{weak-R})
    \qquad
    \dfrac{
      \dfrac{\dfrac{}{A, C \vdash A}\,(ax) \quad \dfrac{}{A, C \vdash C}\,(ax)}{A, C \vdash A \wedge C}\,(\wedge\text{-R})
    }{A, C \vdash A \wedge B, A \wedge C}\,(\text{weak-R})
  }{
    \dfrac{A, B \vee C \vdash A \wedge B, A \wedge C}{A, B \vee C \vdash (A \wedge B) \vee (A \wedge C)}\,(\vee\text{-R})
  }\,(\vee\text{-L})
}{
  \dfrac{A \wedge (B \vee C) \vdash (A \wedge B) \vee (A \wedge C)}{\vdash A \wedge (B \vee C) \rightarrow (A \wedge B) \vee (A \wedge C)}\,(\rightarrow\text{-R})
}\,(\wedge\text{-L})
$$

Korrektheit

- Der Sequenzenkalkül ist korrekt. Das bedeutet:

  Jede im Sequenzenkalkül herleitbare Formel ist eine Tautologie

  Allgemeiner gilt für jede Regel* :

$$\frac{\Gamma_1 \vdash \Delta_1, \ldots, \Gamma_n \vdash \Delta_n}{\Gamma \vdash \Delta}$$

  Ist ϕ eine Belegung mit $|\wedge\Gamma_i \to \vee\Delta_i|_\phi = 1$ für alle $i = 1 \ldots n$, so gilt auch $|\wedge\Gamma \to \vee\Delta|_\phi = 1$.

*Offensichtlich haben wir im Sequenzenkalkül immer $n = 0, 1$ oder 2.

Abschwächungsregeln überflüssig

- Man sieht am letzten Beweis:
  - Abschwächungsregeln braucht man nicht
  - Aber ohne sie
    - schleppt man überflüssige Formeln mit
      - im Antezedenz bzw. im Sukkzedenz
    - Im vorigen Fall würde der Beweis enden mit

$$
\dfrac{
\dfrac{
\dfrac{(ax)}{A, B \vdash A, A \wedge C}
\quad
\dfrac{(ax)}{A, B \vdash B, A \wedge C}
}{A, B \vdash A \wedge B, A \wedge C}
\qquad
\dfrac{
\dfrac{(ax)}{A, C \vdash A \wedge B, A}
\quad
\dfrac{(ax)}{A, C \vdash A \wedge B, C}
}{A, C \vdash A \wedge B, A \wedge C}
}{\ldots}
$$

Invertierbarkeit

- Alle Regeln außer den Abschwächungsregeln sind invertierbar, d.h.
  - Für jede Belegung ϕ gilt
    Die Konklusion ist wahr unter ϕ gdw.
    alle Prämissen sind wahr unter ϕ
- Aus dieser Beobachtung folgt die Vollständigkeit des Sequenzenkalküls für die Aussagenlogik.

Vollständigkeit

- Zu jeder Tautologie der Aussagenlogik lässt sich ein Beweisbaum im Sequenzenkalkül konstruieren.

q sei aussagenlogische Formel. Die Sequenz

$$\{\} \vdash q$$

wird Wurzel eines Herleitungsbaumes.
Führe Rückwärtsbeweis und verwende alleine die logischen Regeln:
Solange in einem Blatt noch ein logischer Operator vorkommt, wende die entsprechende logische Regel rückwärts an.
Es entsteht ein Baum, in dem jedes Blatt die Form

$$x_1, \dots x_n \vdash y_1, \dots, y_k$$

hat, wobei die x_i, y_j aussagenlogische Variablen sind.

Wenn an einem Blatt $\{x_1, \dots x_n\} \cap \{y_1, \dots y_n\} = \emptyset$ gilt, so finden wir eine Belegung, die diesen Sequenten falsifiziert. $(x_i \mapsto T, y_i \mapsto \perp)$

Da alle angewendeten Regeln invertierbar waren wird mit dieser Belegung auch die Ausgangsformel falsch. Sie war also keine Tautologie. Andernfalls ist jedes Blatt ein Axiom und der Baum ein Beweisbaum.

Der Sequenzenkalkül in PVS

- In PVS führt man Rückwärtsbeweise.
  - Aus der zu beweisenden Formel p wird die Sequenz $\{\,\} \vdash p$
  - **`(flatten)`** führt einen oder mehrere der folgenden Regeln aus:
    $(\vee\text{–R}),\ (\wedge\text{–L}),\ (\to\text{–R})$.
    Dabei verändert sich die aktuelle Sequenz - es entsteht keine neue.
  - **`(split)`** führt einen oder mehrere der folgenden Regeln aus:
    $(\vee\text{–L}),\ (\wedge\text{–R}),\ (\to\text{–L})$.
    Aus der aktuellen Sequenz *S* entstehen zwei neue: *S.1* und *S.2*.
    PVS fährt mit *S.1* fort.
  - Die Regeln $(\neg\text{-L})$ und $(\neg\text{ -R})$, $(\perp)$ und (T) führt PVS immer ungefragt durch.
    Ebenso erkennt PVS automatisch, ob die aktuelle Sequenz eine Instanz eines Axioms ist.

Strukturregeln in PVS

- Die Strukturregeln sind eigentlich nicht notwendig, wenn man Antezedenz und Sukzedens als Mengen implementiert. PVS verfügt dennoch über entsprechende Befehle:
  - **(hide)** implementiert (weak-L) und (weak-R). Als Argument erhält dieser Befehl die Nummer der Formel.
  - **(copy)** implementiert die Kontraktionen (con-L) bzw. (con-R).

Navigation und Befehlsmodi

- Man kann jederzeit einen Teilbeweis zurückstellen.
  - **(postpone)** macht die nächste Sequenz zur aktuellen.
  - **(undo)** macht einen oder mehrere Beweisschritte rückgängig.
- Alle genannten Befehle haben Parameter, die in den Tutorials, dem Prover-Manual und in der Hilfe (erreichbar über das Menü im PVS-Fenster) erklärt werden.
- Für die gängigsten Befehle gibt es auch Abkürzungen mittels vorgestellter Tab-Taste:
- **TAB-u**, **TAB-s**, **TAB-i,TAB-I,** etc.
- Der Vorteil dieser Art der Eingabe ist, dass auch die Argumente abgefragt werden.

Zusätzliche Regeln - cut

- Es spricht nichts dagegen, zusätzliche Regeln einzuführen, solange sie korrekt sind und einen Beweis einfacher oder verständlicher machen können.
  - Oft ist eine Fallunterscheidung sinnvoll. Dabei wird ein beliebiger Ausdruck E erfunden und nacheinander angenommen, dass E wahr ist, danach, dass E falsch ist. Die Regel ist:

Schnittregel :
$$\frac{\Gamma \vdash \Delta, p \qquad \Gamma, p \vdash \Delta}{\Gamma \vdash \Delta} \quad \text{(cut)}$$

- PVS hat einen entsprechenden Befehl:

  (case "E") splittet die aktuelle Sequenz

  $H \vdash G$

  in die Sequenzen

  $H, E \vdash G$ und $H \vdash E, G$

Anwendungen der Schnittregel

```
drei.2.1.2 :
{-1}  3 * a + 5 * b = k
[-2]  k > 8
  |-------
[1]   EXISTS (x, y: nat): 3 * x + 5 * y = k + 1

Rule? (then (case "b>=1") (inst 1 "a+2" "b-1")(assert))
-0:**  *pvs*          (ILISP :ready)--L582--87%----------------
```

```
drei.2.1.2.2 :
[-1]  3 * a + 5 * b = k
[-2]  k > 8
  |-------
[1]   b >= 1
{2}   EXISTS (x, y: nat): 3 * x + 5 * y = 1 + k

Rule? (case "b=0")
Case splitting on
   b = 0,
this yields  2 subgoals:
drei.2.1.2.2.1 :

{-1}  b = 0
[-2]  3 * a + 5 * b = k
[-3]  k > 8
  |-------
[1]   b >= 1
[2]   EXISTS (x, y: nat): 3 * x + 5 * y = 1 + k

-0:**  *pvs*          (ILISP :ready)--L1023--91%----------------
```

Zwischenbehauptungen

- Die Schnittregel kann man für die kurzfristige Einführung von Zwischenbehauptungen verwenden
- Dabei wird die Formel p als Lemma gesehen
- Die Beweisaufgabe

$$\Gamma \vdash \Delta$$

wird durch den Schnitt zu den Teilaufgaben reduziert

$\Gamma \vdash p, \Delta$ `% beweise das Lemma p`

und

$\Gamma, p \vdash \Delta$ `% benutze das Lemma p`

CASE

```
PVS@maputo
PVS  File  Edit  Options  Buffers  Tools  Complete  In/Out  Signals  Help

drei.2 :

{-1}  (k > 8 IMPLIES (EXISTS (x, y: nat): 3 * x + 5 * y = k))
{-2}  k + 1 > 8
  |-------
{1}   EXISTS (x, y: nat): 3 * x + 5 * y = k + 1

Rule? (case "k=8 or k>8")
Case splitting on
   k = 8 OR k > 8,
this yields  2 subgoals:
drei.2.1 :

{-1}  k = 8 OR k > 8
[-2]  (k > 8 IMPLIES (EXISTS (x, y: nat): 3 * x + 5 * y = k))
[-3]  k + 1 > 8
  |-------
[1]   EXISTS (x, y: nat): 3 * x + 5 * y = k + 1

Rule? (postpone)
Postponing drei.2.1.

drei.2.2 :

[-1]  (k > 8 IMPLIES (EXISTS (x, y: nat): 3 * x + 5 * y = k))
[-2]  k + 1 > 8
  |-------
{1}   k = 8 OR k > 8
[2]   EXISTS (x, y: nat): 3 * x + 5 * y = k + 1

Rule?
-0:**  *pvs*          (ILISP :ready)--L264--Bot-----------------
```

Einführung des Lemmas

Hier steht es zur Verfügung:

Hier ist das Lemma zu zeigen

Zusätzliche Regeln - IF

- PVS hat auch einen IF-Operator. IF(b,q,r) ist eine Abkürzung für $(b \wedge q) \vee (\neg b \wedge r)$.

IF-Regeln :

$$\frac{\Gamma, b, q \vdash \Delta \qquad \Gamma, r \vdash \Delta, b}{\Gamma, IF(b,q,r) \vdash \Delta} \quad \text{(IF-links)}$$

$$\frac{\Gamma, b \vdash \Delta, q \qquad \Gamma \vdash \Delta, b, r}{\Gamma \vdash \Delta, IF(b,q,r)} \quad \text{(IF-rechts)}$$

Taktiken/Strategien - *prop*

- Mehrere Beweisschritte kann man zu sogenannten *Taktiken* oder *Strategien* zusammenfassen.

- Taktiken sind Programme, deren Aktionen aus PVS-Befehlen bestehen. Sie werden wie Beweisbefehle aufgerufen.

- Man kann Beweisbefehle hintereinander ausführen - **then** - in eine Schleife packen - **repeat** - versuchsweise ausführen – **try** - und vieles mehr.

- Eine eingebaute Strategie ist **(prop)**, mit der man eine beliebige aussagenlogische Tautologie auf einen Schlag beweisen kann.

- Der Benutzer kann sich beliebige Strategien programmieren.

Aussagenlogik

1. Aufbau logischer Sprachen
   – Aussagenlogik
   – Prädikatenlogik (erster Stufe)
   – Logik höherer Stufen
2. Die Sprache der Aussagenlogik
   – Syntax, Semantik
   – Belegung, Wahrheit, Tautologien
   – Kalküle, Beweise
3. Hilbert Kalkül
   – Beweise im Hilbert Kalkül
4. Sequenzenkalkül
   – Korrektheit, Vollständigkeit
   – Der Sequenzenkalkül in PVS
5. Resolventenmethode
   – Korrektheit, Vollständigkeit

Der Resolventenkalkül

- Eine automatische Beweismethode
  - für die Aussagenlogik : vollständig
  - für die Prädikatenlogik : vollständig, aber nicht immer terminierend.

- Beweis im Batch-Modus
  - viele Parameter zur effizienten Beweisfindung
  - Effizienteste Implementierung:
    Otter (**http://www-unix.mcs.anl.gov/AR/otter/**)
    Unter Linux/Unix und unter Windows frei erhältlich

- Erfolgreich beim Lösen offener mathematische Probleme z.B. in
  - Algebraischer Geometrie
  - Verbandstheorie
  - Quasigruppentheorie
  - Logik
  - Kombinatorik
  - ... etc. ....

Normalformen

- Jede aussagenlogische Formel hat eine Normalform:
  - Satz von der disjunktive Normalform:
    „Jede aussagenlogische Formel ist äquivalent zu einer Konjunktion von Disjunktionen von Variablen und negierten Variablen.“

- Analog gibt es eine konjunktive Normalform (KNF):
  - *„Jede aussagenlogische Formel ist äquivalent zu einer Disjunktion von Konjunktionen von Variablen und negierten Variablen.“*

- Jede Formel lässt sich auf einfache Weise in eine DNF oder KNF verwandeln. Dazu benötigt man u.a.

  - Idempotenz $(x \vee x) = x = (x \wedge x)$
  - Kommutativität $(x \vee y) = (y \vee x)$
    $(x \wedge y) = (y \wedge x)$
  - deMorgansche Gesetze $\neg(x \wedge y) = (\neg x \vee \neg y)$,
    $\neg(x \vee y) = (\neg x \wedge \neg y)$
  - Distributivgesetze $x \wedge (y \vee z) = (x \wedge y) \vee (x \wedge z)$
    $x \vee (y \wedge z) = (x \vee y) \wedge (x \vee z)$
  - Doppelte Negation $\neg\neg x = x$

Klauseln

- Ein Literal ist eine Variable oder eine negierte Variable
  - $p, \neg q, \neg x, y, q_{12}, \neg A_{12}, \ldots$
  - Ein Literal heißt *negativ* bzw. *positiv*, je nachdem ob es negiert ist oder nicht
- Eine Klausel ist eine Disjunktion von Literalen
  - z.B.: $p \lor \neg q \lor \neg x \lor q_{12} \lor \neg A_{12}$
  - Wegen Idempotenz und Kommutativität kann man annehmen,
    - daß jedes Literal höchstens einmal vorkommt
    - daß zuerst alle negativen, dann alle positiven Literale erscheinen:
      $\neg x \lor \neg q \lor \neg A_{12} \lor p \lor q_{12}$
- Jede Formel ist äquivalent zu einer Konjunktion von Klauseln

Klauseln und Sequenten

- Eine Klausel

  $\{\neg p_1, \ldots, \neg p_k, q_1, \ldots, q_k\}$

  wobei die p_i und die q_i Atome sind, repräsentiert

  $\neg p_1 \vee \ldots \vee \neg p_k, q_1 \vee \ldots \vee q_k$

- diese ist äquivalent zu der Implikation

  $p_1 \wedge \ldots \wedge p_k \Rightarrow q_1 \vee \ldots \vee q_k$

- also zu dem Sequenten

  - $p_1, \ldots, p_k \vdash q_1, \ldots, q_k$

Repräsentation von Klauseln

- Man repräsentiert
  - Literale als Variablen mit Vorzeichen
  - Klauseln als Mengen von Literalen
    $\{p, \neg q, \neg x, q_{12}, \neg A_{12}\}$, bzw. $\{ \neg p, q, \neg x, y, q_{12}, \neg A_{12} \}$
  - Formeln als Mengen von Klauseln
    $\{ \{p, \neg q, \neg x, q_{12}, \neg A_{12}\},$
    $\{ \neg p, q, \neg x, y, q_{12}, \neg A_{12} \} \}$
  - Mengen von Formeln ebenfalls als Mengen von Klauseln
- Mengenrepräsentation implementiert automatisch
  - Idempotenz, Kommutativität, Assoziativität
- Disjunktion zweier Klauseln entspricht ihrer Mengenvereinigung.
- Leere Klausel repräsentiert die Konstante $\perp$.

Von Formeln zu Klauseln

- Sei eine Formel φ gegeben. Wir wandeln diese in eine Menge $C(\varphi)$ von Klauseln um:

1. Alle Negationen nach innen bringen. (*deMorgansche Regeln*)
2. Doppelte Negationen entfernen
3. Ausdistribuieren $x \vee (y \wedge z) = (x \vee y) \wedge (x \vee z)$

   Danach hat die Formel die Gestalt

   $$(x_{11} \vee \dots \vee x_{1k_1}) \wedge \dots \wedge (x_{n1} \vee \dots \vee x_{nk_n})$$

4. Setze $C(\varphi) := \{ \; \{ x_{11}, \dots, x_{1k_1} \}, \dots, \{ x_{n1}, \dots, x_{nk_n} \} \; \}$
5. Klauseln, die eine Variable sowohl positiv als auch negativ enthalten, kann man entfernen

Allgemeingültigkeit $\leftrightarrow$ Unerfüllbarkeit

- Eine Formel φ ist genau dann eine Tautologie (d.h. allgemeingültig), wenn ihre Negation $\neg\varphi$ unerfüllbar ist.
- Um zu zeigen, dass φ allgemeingültig ist, schreibt man $\neg\varphi$ als Menge von Klauseln $C(\neg\varphi)$ und zeigt, dass diese unerfüllbar ist.
- Frage: Wie kann man feststellen, ob eine Menge von Klauseln unerfüllbar ist ?

Unerfüllbare Klauselmengen

- Eine einzelne Klausel κ ist erfüllbar,
  außer es handelt sich um die leere Klausel {}.

- Eine Klausel, die das gleiche Literal sowohl positiv als auch negativ enthält, wird von jeder Belegung erfüllt, sie kann also weggeworfen werden.

- $\{ \ldots , L_i. , \ldots , \neg L_i , \ldots \} \sim \ldots \vee L_i \vee \ldots \vee \neg L_i \vee \ldots = T$

- Eine Klauselmenge ist erfüllbar, falls es eine Belegung ϕ gibt, die in jeder Klausel mindestens ein Literal erfüllt.

- Spitzfindigkeit: Die Klausel**menge** { } repräsentiert T die Klauselmenge {{}} repräsentiert $\perp$.

Belegung einer Variablen

- $C = \{ \kappa_1, \ldots, \kappa_i \}$ Menge von Klauseln, p eine Variable

- Definiere
  - $C[p:=T] := \{ \kappa - \{\neg p\} \mid \kappa \in C \} - \{ \kappa \in C \mid p \in \kappa\}$
  - $C[p:=\bot] := \{ \kappa - \{p\} \mid \kappa \in C \} - \{ \kappa \in C \mid \neg p \in \kappa\}$

  - Variable p kommt in $C[p:=T]$ bzw. $C[p:=\bot]$ nicht mehr vor !

- Für jede Belegung ϕ: Var $\rightarrow$ 2 gilt:
  - ϕ erfüllt $C[p:=T]$ gdw. $\phi_{[p:=T]}$ erfüllt C
  - ϕ erfüllt $C[p:=\bot]$ gdw. $\phi_{[p:=\bot]}$ erfüllt C

Rekursive Konstruktion einer Belegung

- Gegeben: Menge C von Klauseln.

- Gesucht: Belegung ϕ, die alle Klauseln in C wahr macht
  - d.h. mit $[|\ C\ |]_\phi$
- Ein solches ϕ existiert gdw. C erfüllbar

- Algorithmus: Wähle eine Variable p :
  - Zwei Möglichkeiten: $\phi(p) = T$ oder $\phi(p) = \cdot \perp$
  - If Belegung ψ: Var-$\{p\} \to 2$ existiert, mit $[|\ C[p{:=}T]\ |]_\psi$

    return $\phi = \psi + [p \mapsto T]$

    else if ψ : Var-$\{p\} \to 2$ existiert, mit $[|\ C[p{:=}\perp]\ |]_\psi$

    return $\phi = \psi + [p \mapsto \perp]$

    else return *unerfüllbar*

Resolventenregel

- Seien κ_1, κ_2 Klauseln und l ein Literal:

- $$\frac{\kappa_1 \cup \{\, l \,\}, \quad \kappa_2 \cup \{\, \neg l \,\}}{\kappa_1 \cup \kappa_2}$$

  $\kappa_1 \cup \kappa_2$ heißt Resolvente oder Schnitt von κ_1 und κ_2.

- <u>Beispiele</u>:
  - Aus $\{\neg p, r, s\}$ und $\{\neg p, \neg r, \neg t, u\}$ erhalten wir die neue Klausel $\{\neg p, s, \neg t, u\}$.
- - Aus $\{p\}$ und $\{\neg p\}$ erhalten wir die leere Klausel $\{\}$
- - Aus $\{p,q\}$ und $\{\neg p\}$ erhalten wir $\{ q \}$.

Die Resolventenmethode

- Eine Menge C von Klauseln heißt *unter Resolution abgeschlossen*, falls mit $\kappa_1, \kappa_2 \in C$ auch jede Resolvente von κ_1, κ_2 zu C gehört.

- Algorithmus zur Berechnung des Abschlusses:

  WHILE $\{\} \notin C$ **DO**
      CHOOSE $\kappa_1, \kappa_2 \in C$:
      IF ex. $l \in \kappa_1$: $\neg l \in \kappa_2$ **THEN**
          $C := C \cup \{\kappa_1 - \{l\} \cup \kappa_2 - \{\neg l\}\}$.
- Dieser Algorithmus vergrößert C. Wir stoppen ihn, sobald C unter Resolution abgeschlossen ist.
- C ist genau dann unerfüllbar, wenn $\{\} \in C$.

- <u>Vorsicht</u>: Die Ausgangsklauseln κ_1 und κ_2 muss man i.A. beibehalten.
  - Resolvente oft länger als Ausgangsklauseln

Korrektheit und Terminierung

- Die Resolutionsregel ist korrekt:
- Ist ϕ eine Belegung, die $\kappa_1 \cup \{l\}$ und $\kappa_2 \cup \{\neg l\}$ erfüllt, dann erfüllt sie auch κ_1 oder κ_2, folglich auch $\kappa_1 \cup \kappa_2$.

- Die Resolventenmethode ist korrekt.
- Wenn die Klauselmenge C erfüllbar ist, dann gibt es eine Belegung ϕ, die gleichzeitig alle Klauseln in C erfüllt. Dann erfüllt ϕ auch jede daraus entstandene Resolvente.

- Die Resolventenmethode terminiert.
  Seien $x_1, ..., x_n$ alle Variablen der ursprünglichen Formel, dann kommen in jeder entstehenden Klausel höchstens n der 2n möglichen Literale vor. C bleibt auf jeden Fall endlich.

Vollständigkeit

- Die Resolventenmethode ist vollständig.
- Noch zu zeigen: *Ist eine Menge C von Klauseln gegen Resolution abgeschlossen und nicht erfüllbar, dann gilt { }$\in$C.*

Induktion über die Anzahl n der aussagenlogischen Variablen:

- n=0: $C_0 = \{\}$ oder $C_1 = \{ \{\} \}$. C_0 ist erfüllbar, C_1 nicht. C_1 enthält tatsächlich die leere Klausel.

- n=k+1: Für eine beliebige Variable p betrachten wir

$$C[p{:=}T] := \{ \kappa - \{\neg p\} \mid \kappa \in C \} - \{ \kappa \in C \mid p \in \kappa \in C \} \text{ und}$$

$$C[p{:=}\bot] := \{ \kappa - \{p\} \mid \kappa \in C \} - \{ \kappa \in C \mid \neg p \in \kappa \in C \}.$$

Beide sind gegen Resolution abgeschlossen und haben nur k viele Variablen. Ist eine dieser Klauselmengen erfüllbar, dann auch C. Sind sowohl $C[p{:=}T]$ als auch $C[p{:=}\bot]$ unerfüllbar, so enthalten beide nach Ind.Vor. die leere Klausel {}. Es folgt $\{\neg p\} \in$C und $\{p\} \in$C,

somit auch $\{\} \in$C.

Implementierung der Resolution

- Frühzeitiger Abbruch
  - sobald die leere Klausel erzeugt ist, kann man stoppen
- Es bestehen noch sehr viele Freiheiten
  - wie wählt man das nächste Paar κ_1 und κ_2 von Klauseln
  - welche Variable wählt man zur Bildung der Resolventen
- Bei *Otter* lassen sich bestimmte Strategien durch Parameter einstellen.

Komplexität

- Die Methode funktioniert in der Praxis recht gut
  - Es sind schon sehr komplexe Beweise gefunden worden.
  - Bekannte mathematische Probleme wurden gelöst´.

  aber ..
- worst case $O(2^N)$ mit N Anzahl der Variablen
  - exponential blowup bei Klauselproduktion (DNF)
    - dies kann repariert werden
  - Anzahl der Resolutionsschritte kann exponentiell wachsen.
    - Schlimmes Beispiel: „Pidgeon Hole Principle“.
    - Haken: Kein Ausweg

Effizientere Klauselproduktion

- Gegeben Aussagenlogische Formel ϕ mit Operatoren $\neg, \wedge, \vee, \rightarrow, \ldots$ und Variablen $x_1,\ldots,x_n$.
- Stelle die Formel als Baum dar und führe neue Variablen $v_1,\ldots,v_k$ für alle Baumknoten ψ_{i}, $_{i=1..k}$ ein mit
  - $v_i \leftrightarrow \neg v_r$ falls $\psi_i = \neg\, \psi_r$
  - $v_i \leftrightarrow v_r \wedge v_s$ falls $\psi_i = \psi_r \wedge \psi_s$
  - andere Operatoren analog

- Ersetze die Äquivalenzen durch Klauseln
  - $v_i \leftrightarrow \neg v_r$ durch $\{\{\neg\, v_i, \neg\, v_r\}, \{v_i, v_r\}\}$
  - $v_i \leftrightarrow v_r \wedge v_s$ durch $\{\{\neg\, v_i, v_r\}, \{\neg\, v_i, v_s\}, \{\neg\, v_s, \neg\, v_r, v_i\}\}$
  - andere Operatoren analog

- Ursprüngliche Formel erfüllbar gdw. neue Klauselmenge erfüllbar.

Subsumption

- Eine Klausel κ_1 subsumiert eine Klausel κ_2, falls jedes Literal von κ_1 in κ_2 vorkommt, d.h. $\kappa_1 \subseteq \kappa_2$

- $\{ p_1, \ldots, p_n \}$ subsumiert $\{ q_1,\ldots,q_m \} \Leftrightarrow (p_1 \vee \ldots \vee p_n \rightarrow q_1 \vee \ldots \vee q_m)$

- Sei C eine Klauselmenge, $\kappa_1, \kappa_2 \in C$ mit $\kappa_1 \subseteq \kappa_2$ dann gilt:

  C ist erfüllbar gdw. $C-\{\kappa_2\}$ erfüllbar

- Also: Subsumierte Klauseln können im Resolutionsprozess weggelassen werden

Unit Klauseln

- Eine Klausel, die nur aus einem Literal besteht heißt Unit-Klausel
  - $U = \{ p \}$ oder $U = \{ \neg p \}$
- Unit-resolution
  - Schnitt von Klausel K mit Unit-Klausel $\{ l \}$
    - liefert kürzere Klausel $K - \{ \neg l \}$
    - Ausgangsklausel K wird subsumiert
      - K kann entfernt werden
      - (Unit subsumption)
- Fallunterscheidung
  - Klauselmenge C ist erfüllbar gdw. für jedes Literal p
    - $C \cup \{ \{p\} \}$ erfüllbar oder $C \cup \{ \{\neg p\} \}$ erfüllbar
  - diese Methode der Fallunterscheidung liefert Unit-Klauseln

DPLL (Davis, Putnam, Logeman, Loveland)

- Sei C eine Klauselmenge.

- REPEAT
  - Unit Resolution
  - Unit subsumption
  - Fallunterscheidung
- UNTIL (leere Klausel {} oder nur noch Unit-Klauseln)

- IF leere Klausel entstanden
  - nicht erfüllbar

  ELSE
  - Unit-Klauseln liefern Belegung

- Algorithmus ist korrekt und vollständig.

- Siehe Stroetmann: http://www.ba-stuttgart.de/~stroetma/Skripten/logik.pdf

  - detaillierte Implementierung in SETL
  - Vorbildliche Java-Implementierung
  - Anwendung: 8-Damen Problem

Hakens negatives Beispiel:

- P_n besagt: „Keine n+1 Tauben passen in n Verschläge."
- Wir führen n*(n+1) viele Variablen T_{ik} ein mit $i \in \{1,\ldots,n+1\}$ und $k \in \{1,\ldots,n\}$. Intuitiv soll T_{ik} ausdrücken:
  „*Taube i sitzt im Verschlag k*"
- <u>Beispiel:</u> (n=2)
  $P_2 = \{\{T_{11},T_{12}\}, \{T_{21},T_{22}\}, \{T_{31},T_{32}\},$ % jede Taube in einem Verschlag
  $\{\neg T_{11}, \neg T_{21}\}, \{\neg T_{11}, \neg T_{31}\}, \{\neg T_{21}, \neg T_{31}\},$ % keine zwei in Nr. 1
  $\{\neg T_{12}, \neg T_{22}\}, \{\neg T_{12}, \neg T_{32}\}, \{\neg T_{22}, \neg T_{32}\}$ % keine zwei in Nr. 2
  $\}$
  ist eine widersprüchliche Klauselmenge.
- <u>Satz(Haken)</u>: Es gibt eine Zahl c > 1 so daß jeder Resolutionsbeweis von P_n mindestens c^n viele Schritte benötigt.

Horn-Klauseln

- Klauseln mit höchstens einem positiven Literal
- „Programmklausel:

  $p \vee \neg q_1 \vee \ldots \vee \neg q_n$
- Spezialfall: Fakt:

  p
- Zielklausel
- $\neg g_1 \vee \ldots \vee \neg g_n$

- Prolog Programme sind Listen von Horn Klauseln.

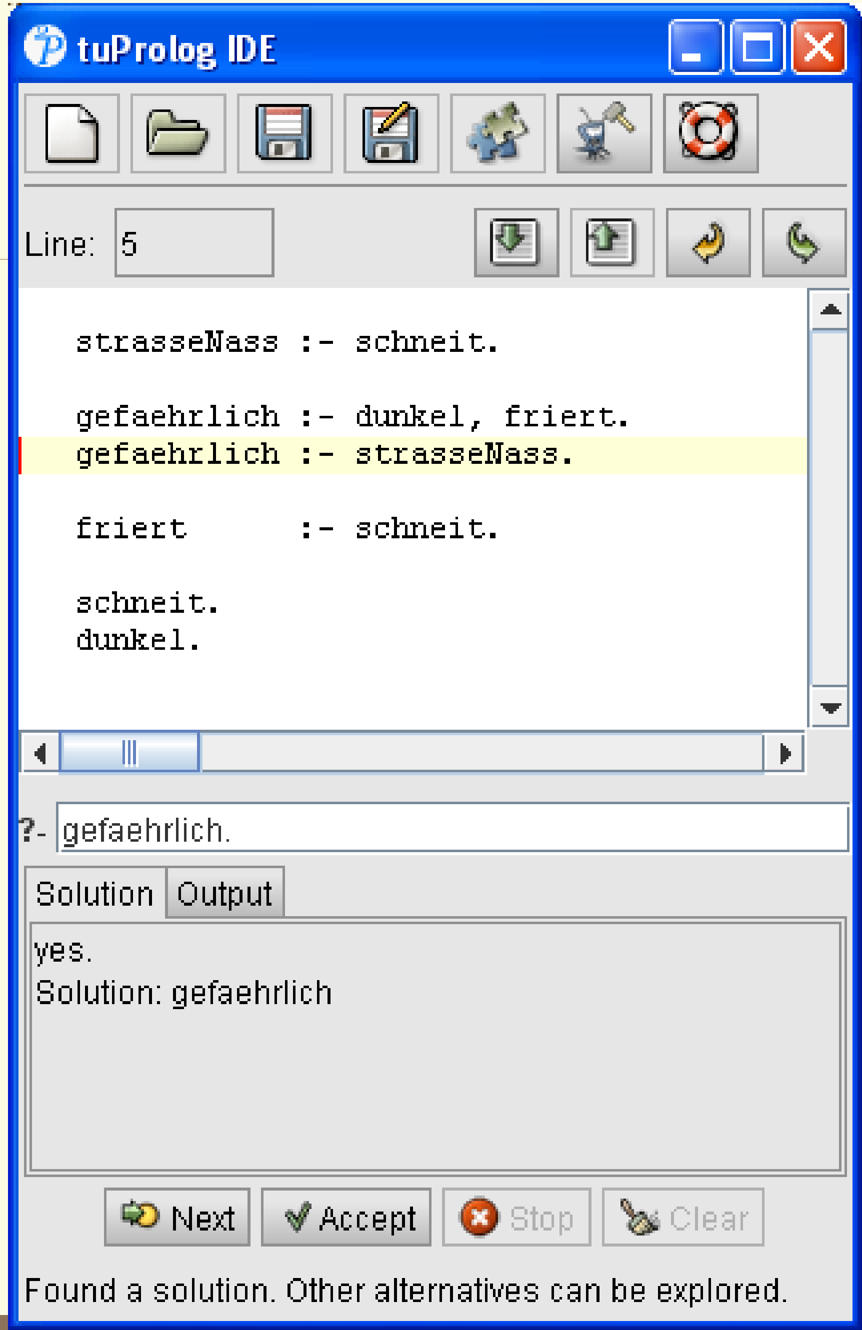
- In Prolog geschrieben als:

  p :- $q_1, \ldots, q_n$.
  - Fakt:

    p.
  - Goal:
    - :- $g_1, \ldots, g_n$.

- Lies p :- $q_1, \ldots, q_n$.

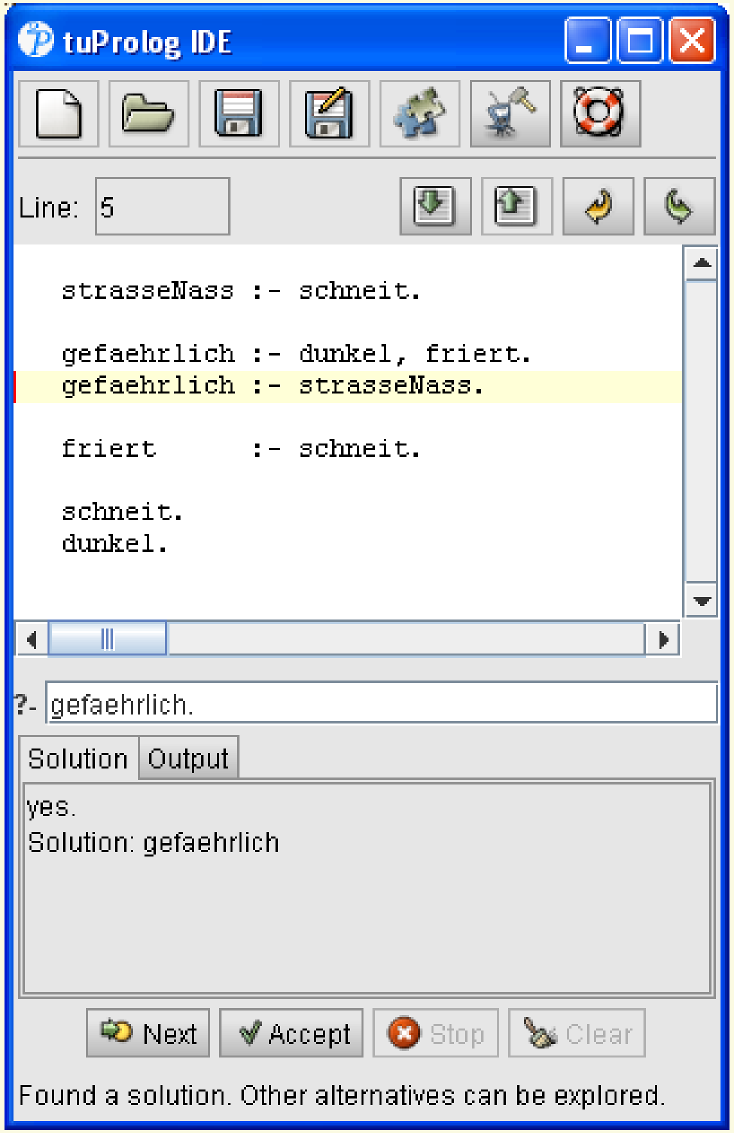
  als
  - „p falls q_1 und … und q_n“.



Resolventen in Prolog

- Leere Klausel nur mit Hilfe einer goal-Klausel möglich.

- Resolvente von goal-Klausel mit Programmklausel ergibt neue goal-Klausel:
  - Goal
    `:- gefaehrlich.`
  - mit Programmklausel
    `gefaehrlich :- dunkel, friert.`
  - ergibt neues goal
    `:- dunkel, friert.`
  - mit Fakt
    `dunkel.`
  - ergibt
    `:- friert`
  - mit Programmklausel
    `friert :- schneit`
  - ergibt
    `:- schneit`
  - mit Fakt
    `schneit.`
  - ergibt leeres goal : yes.

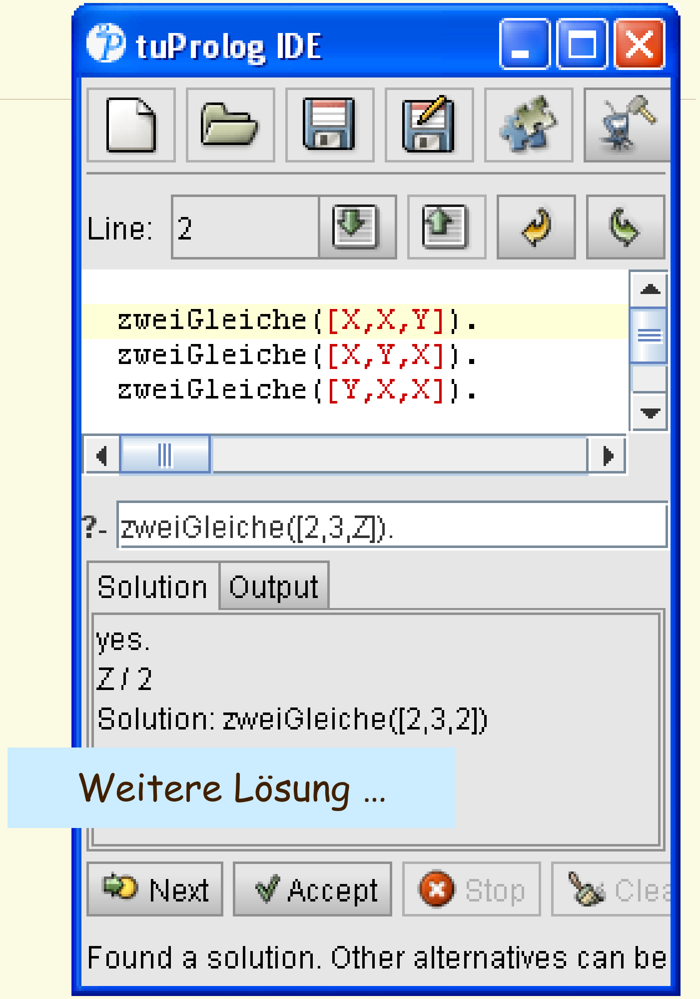


Werte in Prolog

- Terme
  - Atome (beginnen mit Kleinbuchstaben)
    - a, b, otto, eva, r2D2
  - Zahlen
    - 1, 2, -5, 3.14
  - Strings
    - “Hallo Welt“
    - ‘Hallo Welt‘
  - Listen
    - [1,2,otto,“Grüss Gott“]
    - [[a,b],[], [1,2,[3]],[]]

Variablen in Prolog

- Syntaktisch:
  - Beginnen mit Grossbuchstaben
    - X, Y, T1, Clausel1, TestWert
- Semantisch
  - matchen (genauer: unifizieren) mit
    - Werten
    - anderen Variablen
    - (gemischten) Termen
- Beispiele
  - X matcht mit 17.
    - Ergebnis: { X $\mapsto$ 17 }
  - [X, Y, Z] matcht mit [1, 3, [2,4]]
    - Ergebnis: { X $\mapsto$ 1, Y $\mapsto$ 3, Z $\mapsto$ [2,4] }
  - [X, 17, 3, 42] matcht mit [3, Y, X, 42]
    - Ergebnis: { X $\mapsto$ 3, Y $\mapsto$ 17}
  - [X, 17, 3] matcht nicht mit [4, Y, X]
    - Konflikt: X $\mapsto$ 4, X $\mapsto$ 3



Relationen in Prolog

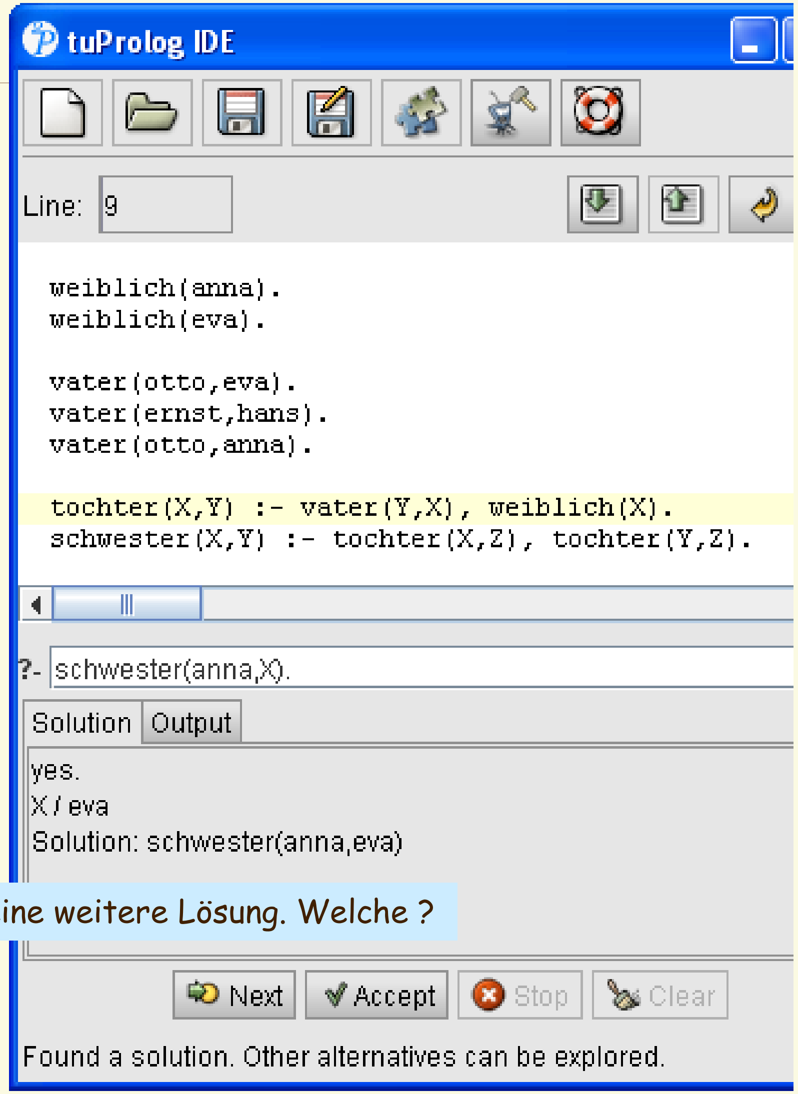
- Statt einfachen Aussagen kann Prolog auch Relationen benutzen

  - Fakten mit einstelligen Relationen:
    ```
    weiblich(anna).
    weiblich(eva).
    ```

  - mit zweistelligen Relationen
    ```
    vater(otto,eva).
    vater(ernst,hans).
    vater(otto,anna).
    ```

  - Programmklauseln mit Relationen
    ```
    tochter(X,Y) :- vater(Y,X),
       weiblich(X).
    schwester(X,Y) :-
       tochter(X,Z), tochter(Y,Z).
    ```

  - Goals
    ```
    :- schwester(anna,X).
    ```



Es gibt eine weitere Lösung. Welche ?

Variablen in Relationen

- **sommer** unterrichtet **graphik**
- …
- jeder (**X**) unterrichtet **praktische**

- **einstein** unterrichtet alles (**Y**)

- **frageZu(G,Nr)**
  - falls es ein **Prof** gibt,
  - mit **unterrichtet(Prof,G),**
  - und **telefon(P,Nr).**

```
telefon(sommer,23416).
telefon(gumm,21516).
telefon(seeger,21526).
telefon(einstein,42).

unterrichtet(sommer,graphik).
unterrichtet(sommer,c).
unterrichtet(seeger,datenbanken).
unterrichtet(gumm,theoretische).

unterrichtet(X,praktische).
unterrichtet(einstein,X).

frageZu(Gebiet,Nummer) :-
        unterrichtet(Prof,Gebiet),
        telefon(Prof,Nummer).
```

```
?- frageZu(praktische,Y).
```

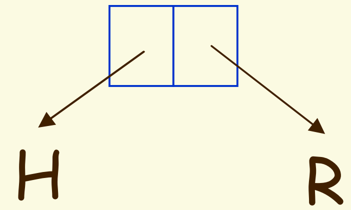
Solution | Output

```
yes.
Y / 23416
Solution: frageZu(praktische,23416)
```

Listen in Prolog

- [] leere Liste

- [H | R] Liste mit Kopf H und Rest R

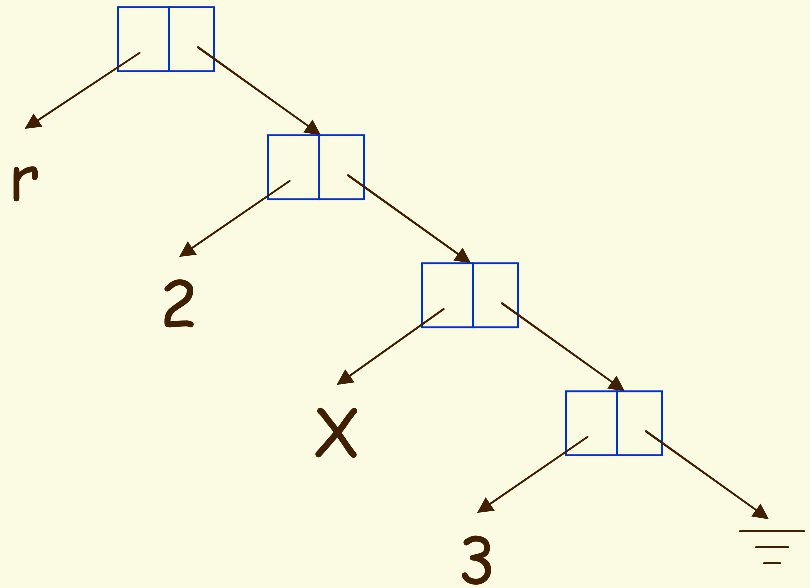


- Kurzschreibweisen:

  [3]   kurz für [3 | []]

  [2, 3] kurz für
  [2 | [3]] = [2| [3 | []]]

  [r, 2, X, 3] kurz für
  [r | [2,X,3]] =
  = …
  = [r | [2 | [X | [3 | []]]]]



-

Funktionen

- n-stellige Funktionen durch (n+1)-stellige Prädikate repräsentiert

  p_f(x_1,…,x_n,y)

  statt

  f(x_1,…,x_n) = y

- Beispiele:
  - conc([r,2],[d,2],[r,2,d,2])
    statt conc([r,2],[d,2])= [r,2,d,2]
  - rev([r,3,“otto“],X)
    statt X=rev[r,3,“otto“]

- rekursive Aufrufe erlaubt

- Abarbeiten der Programm-klauseln der Reihe nach

- Bearbeitung der Goal-teile von links nach rechts

```
% Ist X Element der Liste
  elem(X,[X|Rest]) .
  elem(X,[H|Rest]) :- elem(X,Rest).

% Konkateniere zwei Listen
  conc([],Y,Y).
  conc([X|R],Y,[X|Z]) :- conc(R, Y, Z).

% drehe Liste um
% imperative Leseweise:
% die leere Liste ergibt umgedreht die leere Liste
  rev([],[]).
% um eine nichtleere Liste [H|R] umzudrehen,
% -- drehe den Rest um
% -- konkateniere Ergebnis mit Einerliste des Heads
  rev([H|T],R) :- rev(T,T1),conc(T1,[H],R).

?- rev([3,5,7,anna],R).

Solution  Output
yes.
R / [anna,7,5,3]
Solution: rev([3,5,7,anna],[anna,7,5,3])
```

Arithmetische Operationen

t is e

linkes Argument:
- ein Term t
- meistens eine Variable

rechtes Argument:
- ein arithmetischer Ausdruck e

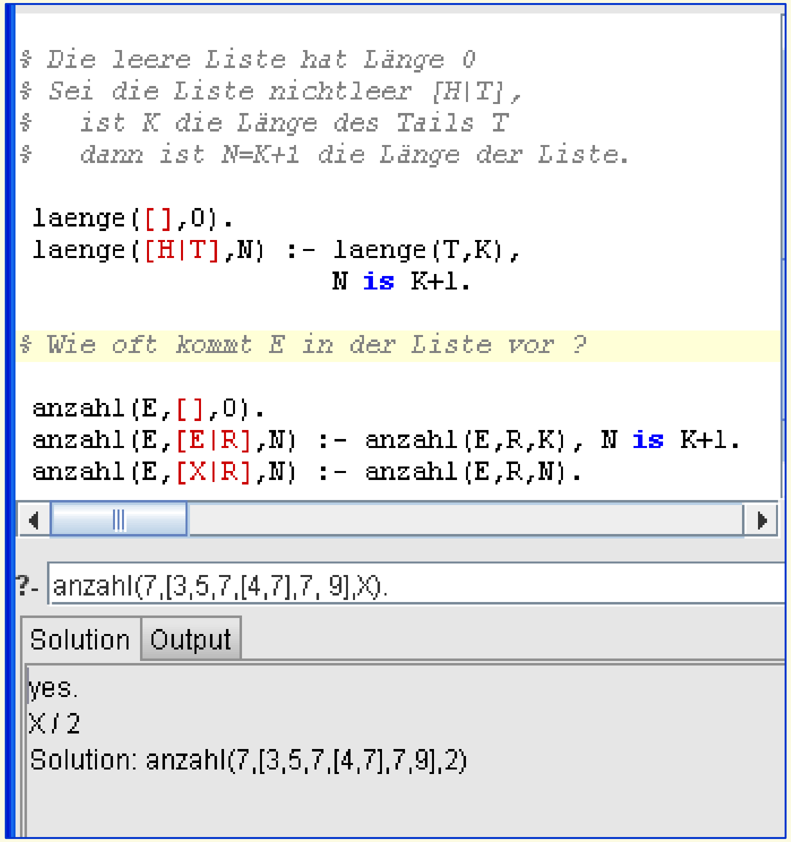
Semantik
- berechnet arithmetischen Ausdruck
- versucht, Ergebnis mit dem Term zu matchen

Beispiele:

X is 2+3
- berechnet 2+3 und matcht mit X

0 is 56088 mod 123
- wahr, falls 56088 mod 123 = 0, false sonst

Y = X*X
- wahr falls zum Zeitpunkt der Berechnung X einen Zahlenwert hat dessen Quadrat Y matcht.



Benutzerdefinierte Terme

Datenwerte bisher

- Atome, Strings, Listen

Eigene Datenwerte einführbar

- müssen nicht definiert werden.
- Syntaktisch: wie Relationen
- Semantisch: Konstruktoren

- $f(t_1,\ldots,t_n)$
  - f Funktionszeichen
  - t_i Terme
- stehen aber dort wo Wert verlangt ist

Beispiele:

- Paare
  - paar(2,3), paar(7,-19), paar(2,paar(X,3))

- Bäume
  - baum('*'baum('+',blatt(5),
  - blatt(7)),
  - blatt(3))

- Selbstgebaute Zahlen
  - nix, s(nix), s(s(nix)), s(s(X)), …

```
/* Hilfsfunktion max */
 max(X,Y,X) :- Y =< X.
 max(X,Y,Y) :- X =< Y.

% Syntaxbäume als baum(Operator,Baum,Baum) oder als Wert
tiefe(baum(Op,X,Y),T) :-  tiefe(X,T1),
                          tiefe(Y,T2),
                          max(T1,T2,Ta),
                          T is Ta + 1.
tiefe(blatt(X),0).    % ansonsten.

% Syntaxbaum auswerten
eval(baum(Op,L,R),W) :-  eval(L,WL),
                          eval(R,WR),
                          apply(Op,WL,WR,W).
eval(blatt(X),X).

apply('+',W1,W2,W) :- W is W1+W2.
apply('*',W1,W2,W) :- W is W1*W2.

beispiel(baum('*',baum('+',blatt(5),blatt(7)),blatt(3))).
```

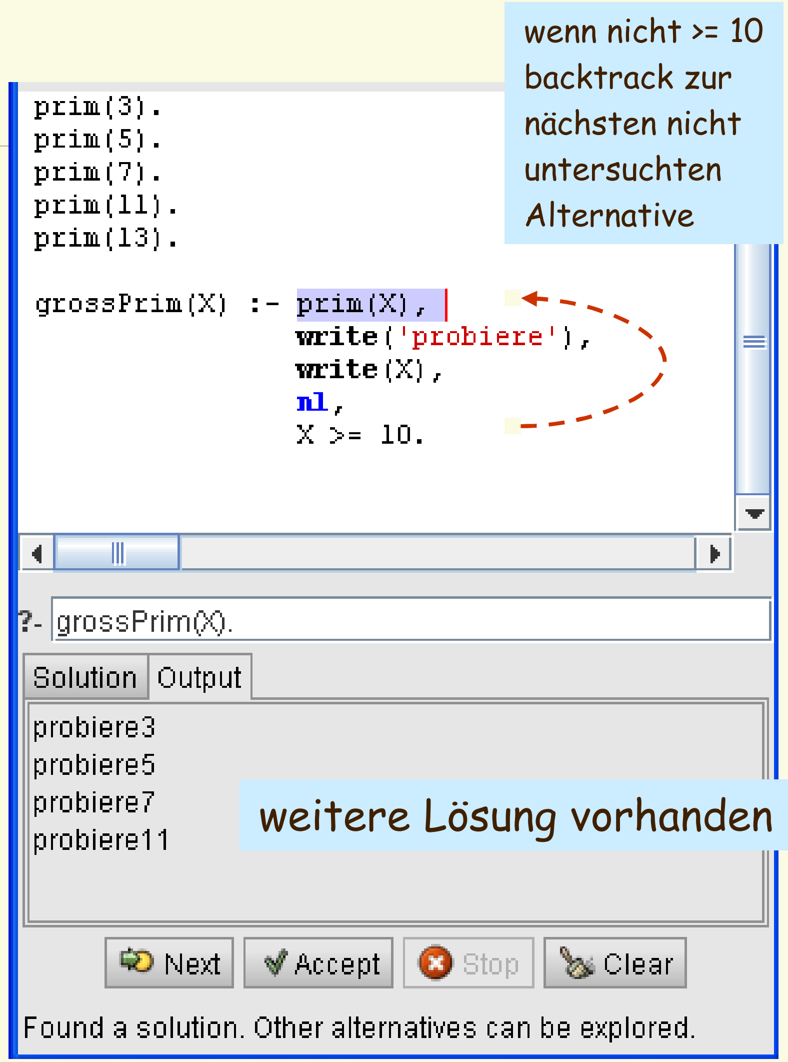
```
?- beispiel(X), tiefe(X,Y).
```

Solution | Output

```
yes.
X / baum('*',baum('+',blatt(5),blatt(7)),blatt(3))  Y / 2
Solution: ','(beispiel(baum('*',baum('+',blatt(5),blatt(7)),blatt(3))),tiefe(baum('*',b
```

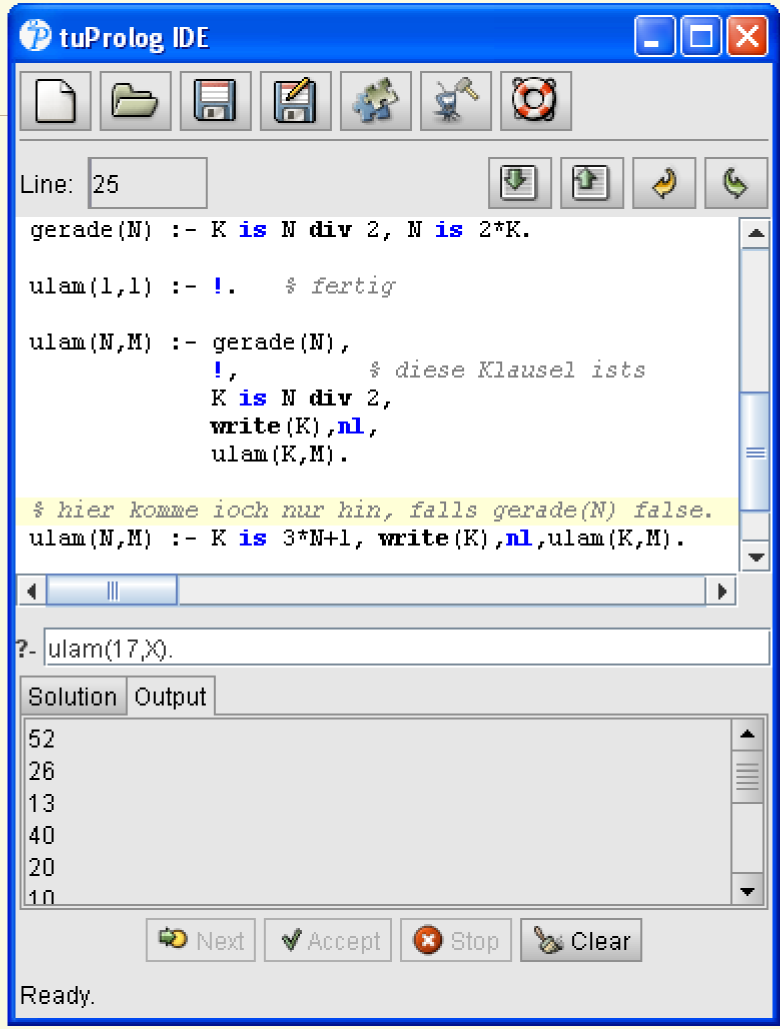
Seiteneffekte

- Bearbeitung der Goal-Bestandteile von links nach rechts
  - :- write(‘Hallo‘), write(‘ Welt‘).
- Programmklauseln werden der Reihe nach ausprobiert
  - Wenn notwendig: Backtracking



Der Cut

- Dient dazu, Backtracking abzuschneiden
  - kann man verwenden, um Negation zu implementieren.
- $p \text{ :- } \underbrace{q_1,\ldots,q_k}_{\text{test}},\ \underbrace{!}_{\text{commit}},\ q_{k+1},\ldots,q_n.$
- Wenn bei der Berechnung von p die Prämissen $q_1,\ldots,q_k$ erfolgreich getestet wurden, wird keine weitere Klausel für p mehr berücksichtigt.



Beispiel: sat in Prolog

sat(Cls,Part,Bel)

ist wahr, falls die partielle Belegung Part zu einer Belegung Bel ergänzt werden kann, die ale Klauseln in Cls wahr macht.

```
/* Erfüllbarkeit in Prolog
   Atomare Aussagen repräsentiert durch Integer != 0
   -L ist Negation von L */

   %% sat(+Klauselmenge,+Belegung,-Belegung)

   sat([],X,X)              :- !,write(X).            % gefunden

   sat([[]|_],_,_)          :- !,fail.                % leere Klausel

   sat([[L|_]|Rest],X,Y)    :- member(L,X),           % L schon wahr
                                !,sat(Rest,X,Y).

   sat([[L|Ls]|Rest],X,Y)   :- NL is -L, member(NL,X), %-L schon wahr
                                !, sat([Ls|Rest],X,Y).

   sat([[L|_]|Rest],X,Y)    :- sat(Rest,[L|X],Y),!.   % probiere L=true

   sat([[L|Ls]|Rest],X,Y)   :- NL is -L,
                               sat([Ls|Rest],[NL|X],Y),!.   % probiere L=false

/* Testaufruf */

test(X) :- sat([[1,4],[1,3,-8],[1,8,11],[2,11],[-7,-3,9],
                [-7,8,-9],[7,8,-10],[7,10,-11],[-3,-7,8],
                [3,7,-1],[-3,-4,7],[3,-7],[-3,7]
               ],
               [],X).
```

?- test(X).

Solution | Output

yes.
X / [-4,8,9,3,7,2,1]
Solution: test([-4,8,9,3,7,2,1])

Beispiel: DPLL in Prolog(1)

```
/* Davis Putnam Resolution, H.P.Gumm, 2007.

   Literale : a, ~a, b, ~b, ... oder auch 1,2,3,~1,~2,~3, ...
   Klauseln : Listen von Literalen [a,~b,c]
   Klauselmengen : Listen von Klauseln.  [[a,~b,c],[~c,b],[~a,b]]
   Invariante: in keiner Klausel kommt ein Literal negiert und unnegiert vor.   */

 /* Hilfsfunktionen */
 :- op(100,fx,'~').         % ~ als präfix-Operator erklären

 /* Negiere ein Literal */
 negate(~L,L) :- !.
 negate(L,~L).

 /* Versuche Element aus Liste zu entfernen - always succeed */
 delmem(_,[],[]).
 delmem(A,[A|As],As) :- !.
 delmem(A,[B|Bs],[B|Rs]) :- delmem(A,Bs,Rs).

 /* Unter der Voraussetzung, dass Lit wahr ist, entferne ~Lit aus Klausel */

 simpClause(Lit,Clause,Clausel) :- negate(Lit,NL), delmem(NL,Clause,Clausel).

 /* Entsprechend: Vereinfache Klauselmenge:
     - unit subsumption : entferne alle Klauseln mit Lit
     - unit resolution :  entferne ~Lit aus allen übrigen Klauseln  */

 simpAll(L,[],[]) :- !.
 simpAll(L,[C|Cs],Csl) :- member(L,C), !, simpAll(L,Cs,Csl).
 simpAll(L,[C|Cs],[Cl|Csl]) :- simpClause(L,C,Cl),simpAll(L,Cs,Csl).

/* Berechne erfüllende Belegung Env für Klauselmenge oder berichte: "unerfüllbar"
```

Beispiel: DPLL in Prolog (2)

```
/* Berechne erfüllende Belegung Env für Klauselmenge oder berichte: "unerfüllbar"
   Env ist Liste von Literalen, die die Klauseln wahr machen
   resolve(Klauselmenge,partielle Belegung,fertige Belegung)
        Anfangs: partielle Belegung = []
        Zum Schluss: Fertige Belegung = Partielle Belegung
   Die dritte Regel (Unit-Klausel) ist nicht notwendig, aber sehr effektiv
   count dient nur zum Profiling. */

   resolve([],Env,Env)    :- !.   /* Env gefunden */

   resolve(Cs,_,_)        :- member([],Cs),!,fail.

   resolve(Cs,Env,Envl)   :- member([Lit],Cs),                          % es git eine UnitKlausel
                             !,
                             simpAll(Lit,Cs,Csl),                       % Unit-Resolution, Unit subsumption
                             resolve(Csl,[Lit|Env],Envl).               % erweitere partielle Belegung

   resolve([Clause|Cs],Env,Envl) :-
                             member(Lit,Clause),                        % wähle ein Literal der 1. Klausel
                             simpAll(Lit,Cs,Csl),                       % wie oben
                             resolve(Csl,[Lit|Env],Envl).


 /*  ============================ Die Hauptroutine: solve  ================== */
     solve(Cs,Env) :- resolve(Cs,[],Env),!,write(Env).
     solve(Cs,_)   :- write("unerfüllbar ").
```

?- solve([[b,c],[c,d,~b],[~d,c],[b]],X).

Solution | Output

yes.
X / [c,b]
Solution: solve([[b,c],[c,d,'~'(b)],['~'(d),c],[b]],[c,b])

Anwendung: Taubenproblem in Prolog