Suchverfahren in Graphen

Graph: $G = [Z,E]$ mit
- Anfangszustand $z_0 \in Z$
- Zielzuständen $Z_f \subseteq V$

Probleme:
- Speicher reicht nicht für vollständigen Zustandsraum
- Aufwand für Erkennen von Wiederholungen

Lösungsmethode:

„Expansion des Zustandsraumes“:

Schrittweise Konstruktion und Untersuchung von Zuständen

„konstruieren – testen – vergessen“

Expansionsstrategien

- **Richtung**
  - Vorwärts, beginnend mit z_0 (forward chaining, data driven, bottom up)
  - Rückwärts, beginnend mit Z_f (backward chaining, goal driven, top down)
  - Bidirektional
- **Ausdehnung**
  - Tiefe zuerst
  - Breite zuerst
- **Zusatzinformation**
  - blinde Suche
  - heuristische Suche

Aufgaben von Suchalgorithmen

- Probleme:
  - Existiert ein Weg von z_0 zu einem $z_f \in Z_f$
  - Konstruiere einen Weg von z_0 zu einem $z_f \in Z$
  - Konstruiere optimalen Weg von z_0 zu einem $z_f \in Z_f$

Güte von Suchalgorithmen

Bezogen auf Konstruktion von Wegen zum Ziel:
- **Korrektheit:**
  - Algorithmus liefert nur korrekte Wege.
- **Vollständigkeit:**
  - Algorithmus liefert (mindestens) alle korrekten Wege.
- **Optimalität:**
  - Algorithmus liefert optimale Wege.

Vollständigkeit kann auch schwächer gefasst werden (vgl. Existenzproblem): Algorithmus liefert einen korrekten Weg, falls eine Lösung existiert.

Komplexität von Suchalgorithmen

- bzgl. Komplexität des Verfahrens:
  - Zahl der Zustände insgesamt
  - Zahl der erreichbaren Zustände
  - Zahl der untersuchten Zustände
  - Suchtiefe
- bzgl. Gefundener Lösung

Graph: $G = [Z,E]$ mit
- Anfangszustand $z_0 \in Z$
- Zielzuständen $Z_f \subseteq V$

Zyklen, Maschen im Suchraum

Prolog:

```
erreichbar(X,Y) :- erreichbar(X,Z), nachbar(Z,Y).
erreichbar(X,X).

symmetrisch(X,Y):- symmetrisch(Y,X).
```

Test auf Wiederholungen:

Zeit-, Speicher- aufwändig

Beschränkung der Suchtiefe

Suche nach einem Weg

Expansion: Schrittweise Konstruktion des Zustandsraums

Datenstrukturen :

• Liste OPEN:

Ein Zustand (Knoten) heißt "offen" , falls er bereits konstruiert, aber noch nicht expandiert wurde (Nachfolger nicht berechnet)

• Liste CLOSED:

Ein Zustand (Knoten) heißt "abgeschlossen" , falls er bereits vollständig expandiert wurde (Nachfolger alle bekannt)

Zusätzliche Informationen:
z.B. Nachfolger/Vorgänger der Knoten (für Rekonstruktion gefundener Wege)

Schema S (Suche nach irgendeinem Weg)

S0: (Start) Falls Anfangszustand z_0 ein Zielzustand: EXIT(„yes:“ z_0).
OPEN := [z_0] , CLOSED := [ ] .
S1: (negative Abbruchbedingung) Falls OPEN = [ ] : EXIT(„no“).
S2: (expandieren)
Sei z der erste Zustand aus OPEN.
OPEN := OPEN – {z} . CLOSED := CLOSED ∪{z} .
Bilde die Menge Succ(z) der Nachfolger von z.
Falls Succ(z) = {} : Goto S1.
S3: (positive Abbruchbedingung)
Falls ein Zustand z_1 aus Succ(z) ein Zielknoten ist: EXIT(„yes:“ z_1).
S4: (Organisation von OPEN)
Reduziere die Menge Succ(z) zu einer Menge NEW(z)
durch Streichen von nicht weiter zu betrachtenden Zuständen.
Bilde neue Liste OPEN durch Einfügen der Elemente aus NEW(z) .
Goto S1.

Variable Komponenten in Schema S :

(Re-)Organisation von OPEN in S4
– V1. Bildung der Menge NEW(z) aus Succ(z) :
(Auswahl der weiter zu betrachtenden Zustände)
• alle Zustände aus Succ(z)
• einige (aussichtsreiche)
• nur die, die noch nicht in OPEN
• nur die, die nicht in CLOSED
– V2. Sortierung von OPEN
(bestimmt den nächsten zu expandierenden Zustand in S2)
• NEW(z) sortieren
• NEW(z) einfügen, z.B. an Anfang oder Ende,
• OPEN (gesamte Liste) neu sortieren
– V3. Weitere Bedingungen
• Beschränkung der Suchtiefe
• Reduzierte Menge CLOSED

Blinde Suche mit Test auf Wiederholungen:

(1) Tiefe-Zuerst:
• V1: NEW(z) = Succ(z)-(OPEN ∪CLOSED)
• V2: NEW(z) an den Anfang von OPEN

Keller

(2) Breite-Zuerst:
• V1: NEW(z) = Succ(z)-(OPEN ∪CLOSED)
• V2: NEW(z) an das Ende von OPEN

Warteschlange

Für endliche Graphen:
korrekt und vollständig

Hoher Speicheraufwand
speziell für CLOSED

Vollständig im Sinne:
findet (eine) Lösung im Fall der Existenz

Blinde Suche ohne Test auf Wiederholungen:

Graph als „Abgewickelter Baum“

(1) Tiefe-Zuerst:
• V1: NEW(z) = Succ(z)
• V2: NEW(z) an den Anfang von OPEN

(2) Breite-Zuerst:
• V1: NEW(z) = Succ(z)
• V2: NEW(z) an das Ende von OPEN

Für endliche Graphen:
Tiefe-Zuerst: korrekt, aber nicht immer vollständig
Breite-Zuerst: korrekt und vollständig

Blinde Suche ohne Test auf Wiederholungen:

Graph als „Abgewickelter Baum“

Speicheraufwand für OPEN

– Tiefe-Zuerst: linear $d \cdot b$

– Breite-Zuerst: exponentiell b^d
(bei b=fan-out, d=Tiefe)

Hoher Zeitaufwand für Wiederholungen

Backtracking

Implementierung von Tiefe-zuerst-Verfahren

Spezielle Organisation der Liste OPEN:
Referenz auf jeweils nächsten zu expandierenden Zustand in jeder Schicht

Nach Abarbeiten aller Zustände einer Schicht zurücksetzen (backtracking) auf davor liegende Schicht

Möglichkeit für Zyklenvermeidung mit reduzierter Menge CLOSED (nur für aktuellen Zweig):
– Beim Backtracking Rücksetzen von CLOSED auf früheren Stand

Iterative Tiefensuche

Stufenweise begrenzte Tiefensuche
- Stufe 1: begrenzte Tiefensuche bis zur Tiefe 1
- Stufe 2: begrenzte Tiefensuche bis zur Tiefe 2
- Stufe 3: begrenzte Tiefensuche bis zur Tiefe 3
- ...

„Depth-first-iterative deepening (DFID)“

Iterative Tiefensuche

DFID bis Tiefe d bei fan-out b erfordert insgesamt
$b^d + 2 \cdot b^{d-1} + 3 \cdot b^{d-2} + ... + (d-1) \cdot b$ Schritte

Vergleich mit Tiefe-Zuerst/ Breite-Zuerst bis Tiefe d :
$b^d + b^{d-1} + b^{d-2} + ... + b$ Schritte

DFID hat Speicherbedarf für OPEN wie Tiefe-zuerst
DFID findet Lösung wie Breite-zuerst

Heuristische Suche

Schätzfunktion $\sigma(z)$:

geschätzter Konstruktions-Aufwand

für Erreichen eines Zielzustandes von z aus

Heuristik: Zustände mit optimaler Schätzung bevorzugen

Heuristische Suche

(3) Bergsteigen/“hill climbing“ (ohne Test auf Wdh.):
- V1: NEW(z) = Succ(z)
- V2: NEW(z) nach Aufwand sortiert an Anfang von OPEN

(4) Bestensuche (ohne Test auf Wdh.):
- V1: NEW(z) = Succ(z)
- V2: OPEN ∪ NEW(z) nach Aufwand sortieren

Für endliche Graphen:
korrekt, aber nicht immer vollständig

Bei beiden Verfahren oft nicht alle Zustände aus NEW(z) weiter betrachten (Speicher sparen, „Pruning“)

Typische Probleme lokaler Optimierung

Vorgebirgsproblem:
steilster Anstieg führt auf lokales Optimum ("Nebengipfel")

Plateau-Problem:
keine Unterschiede in der Bewertung

Grat-Problem:
vorgegebene Richtungen erlauben keinen Anstieg

Konsequenz: zwischenzeitliche Verschlechterungen zulassen

Suche nach "bestem Weg“

Bester/optimaler Weg:

Minimale Kosten bzgl. einer Kostenfunktion.

Unterschied zu Schätzfunktion σ

Suche nach "bestem Weg“

Kosten für Zustandsübergang (Kante)

$c: E \rightarrow \mathcal{R}^+$ (Kosten stets positiv!)

mit $c(e)$ = Kosten der Kante $e \in E$

bzw. $c(z,z')$ = Kosten der Kante $e=[z,z']$

Weg-Kosten als Summe von Kosten der Kanten.

Kosten eines Weges $s = e_1 \ldots e_n \in E^*$:

$$c(e_1 \ldots e_n) = \Sigma_{i=1,\ldots,n}\, c(e_i)$$

Kosten eines Weges $s = z_0\, z_1 \ldots z_n \in Z^*$

$$c(z_0\, z_1 \ldots z_n) = \Sigma_{i=1,\ldots,n}\, c(z_{i-1}, z_i)$$

Suche nach "bestem Weg“

Kosten für Erreichen des Zustandes z' von z aus:

– Falls z' von z erreichbar:

$g(z,z') := \text{Min}\{ c(s) / s$ Weg von z nach $z' \}$,

– Andernfalls: $g(z,z') := \infty$

Vorläufigkeit der Kostenberechnung während Expansion:

$G' = [Z', E']$ sei (bekannter) Teilgraph von G

$g'(z, z', G') := \text{Min}\ \{ c(s) / s$ Weg in G' von z nach $z' \}$

$g'(z,z',G') \geq g(z,z')$

Suche nach "bestem Weg“

Verfahren “Generate and Test”:
Alle Wege im Graphen untersuchen.

$L(z_0)$
= Menge der in z_0 beginnenden Wege $p= v_0 \ldots v_n$

$L(z_0,Z_f)$
= Menge der in z_0 beginnenden Wege $p= v_0 \ldots v_n$ mit $v_n \in Z_f$

Kürzesten Weg in $L(z_0,Z_f)$ bestimmen.

Suche nach bestem Weg

Schema S (Suche nach irgendeinem Weg)
findet eventuell zuerst teure Wege

S0: (Start) Fall
OPEN := [z
S1: (negative Abbruchbedingung) Falls OPEN = [ ] : EXIT(„no“).
S2: (expandieren)
Sei z der erste Zustand aus OPEN.
OPEN := OPEN – {z} . CLOSED := CLOSED ∪{z} .
Bilde die Menge Succ(z) der Nachfolger von z.
Falls Succ(z) = {} : Goto S1.
S3: (positive Abbruchbedingung)
Falls ein Zustand z_1 aus Succ(z) ein Zielknoten ist: EXIT(„yes:“ z_1).
S4: (Organisation von OPEN)
Reduziere die Menge Succ(z) zu einer Menge NEW(z)
durch Streichen von nicht weiter zu betrachtenden Zuständen.
Bilde neue Liste OPEN durch Einfügen der Elemente aus NEW(z) .
Goto S1.

Suche nach "bestem Weg“

S0: (Start) Falls Anfangszustand z_0 ein Zielzustand: EXIT(„yes:“ z_0).
OPEN := $[z_0]$, CLOSED := [ ] .
S1: (negative Abbruchbedingung) Falls OPEN = [ ] : EXIT(„no“).
S2: (expandieren)
Sei z der erste Zustand aus OPEN.
OPEN := OPEN - {z} . CLOSED := CLOSED ∪{z} .
…enge Succ(z) der Nachfolger von z.
…z) = {} : Goto S1.

Lösungsidee jetzt:

Abbrechen, wenn alle offenen Wege teurer sind als aktuell gefundene Lösung

…rch Streichen von nicht weiter zu betrachtenden Zuständen.
…de neue Liste OPEN durch Einfügen der Elemente aus NEW(z) .

dafür:

- Positive Abbruchbedingung von Schema S verändern
- Umstellung der Schritte in Schema S

Schema S‘ für Suche nach "bestem Weg“

S‘0: (Start) Falls Anfangszustand z_0 ein Zielzustand: EXIT(„yes:“ z_0).
OPEN := [z_0] , CLOSED := [ ] .
S‘1: (negative Abbruchbedingung) Falls OPEN = [ ] : EXIT(„no“).
S‘2: **(positive Abbruchbedingung)**
Sei z der erste Zustand aus OPEN.
Falls z ein Zielknoten ist: EXIT(„yes:“ z).
S‘3: (expandieren)
OPEN := OPEN - {z} . CLOSED := CLOSED ∪{z} .
Bilde die Menge Succ(z) der Nachfolger von z.
Falls Succ(z) = {} : Goto S‘1.
S‘4: (Organisation von OPEN)
– $g'(z_0, z', G')$ für alle $z' \in Succ(z)$ berechnen (im aktuellen G´) .
– Neue Liste OPEN durch Einfügen der Elemente aus Succ(z):
Sortieren von OPEN ∪ Succ(z) nach aufsteigendem $g'(z_0, z', G')$
Goto S‘1.

Schema S‘ für Suche nach "bestem Weg“

Satz :
Vor.: Es existiert $\delta > 0$ mit $c(z,z') > \delta$ für alle Kanten in G
Beh.: Falls Lösung existiert, so findet S‘ einen optimalen Weg

• „Verzweigen und Begrenzen“ (Branch and bound)
• „Dijkstra's Algorithmus“ (1959)

Verbesserungen möglich:
Streichen aus OPEN (bzw. Succ(z)):
– Zustände aus CLOSED
– mehrmaliges Auftreten von Zuständen
• Prinzip der Dynamischen Optimierung/Programmierung

Heuristische Suche nach „bestem Weg“

Problem:
Gleichzeitig **Kostenfunktion** $g'(z, z', G')$ *(bisheriger Weg)*
und **Schätzfunktion** $\sigma(z)$ *(zukünftiger Weg)* berücksichtigen

Vorläufigkeit der Kostenberechnung während Expansion:
$G' = [Z', E']$ sei (bekannter) Teilgraph von G
$g'(z, z', G') :=$ Min { c(s) / s Weg in G´ von z nach z´ }

Schätzfunktion $\sigma(z)$: geschätzter Konstruktions-Aufwand für Erreichen eines Zielzustandes von z aus
Heuristik: Zustände mit optimaler Schätzung bevorzugen

Schema S‘‘ für heurist. Suche nach "bestem Weg“

S‘‘0: (Sta… …0).
OPE…
S‘‘1: (ne…
S‘‘2: **(positive Abbruchbedingung)**
Sei z d…
Falls z e…
S‘‘3: (expandieren)
OPEN := OPEN - {z} . CLOSED := CLOSED ∪{z} .
Bilde die Menge Succ(z) der Nachfolger von z.
Falls Succ(z) = {} : Goto S‘1.
S‘‘4: (Organisation von OPEN)
– $g'(z_0, z', G') + \sigma(z)$ für alle $z' \in Succ(z)$ berechnen (im aktuellen G´)
– Neue Liste OPEN durch Einfügen der Elemente aus Succ(z):
– Sortieren von OPEN ∪ Succ(z) nach aufsteigendem $g'(z_0, z', G') + \sigma(z)$
Goto S‘‘1.

Modifikation von Schema S‘ in Schritt 4:
$g'(z_0, z', G') + \sigma(z)$ anstelle von $g'(z_0, z', G')$

Verfälschung des Ergebnisses durch $\sigma(z)$ möglich.

Heuristische Suche nach „bestem Weg“

Schätzfunktion σ heisst *optimistisch* oder *Unterschätzung*,
falls $\sigma(z) \leq g(z, Z_f)$ für alle $z \in Z$.

Satz :
Vor.: Es existiert $\delta > 0$ mit $c(z,z') > \delta$ für alle Kanten in G
σ sei eine optimistische Schätzfunktion
Beh.: Falls Lösung existiert, so findet S‘ einen optimalen Weg

Schema S‘‘ für heurist. Suche nach "bestem Weg“

Streichen von CLOSED-Zuständen aus OPEN kann bei S‘‘ zu Problemen führen.

Benötigen schärfere Bedingungen an σ:
„konsistente Schätzfunktion“
Algorithmus A*

Heuristik vs. Kosten

	ohne Kosten c	mit Kosten c
ohne Heuristik σ	*Blinde Suche, irgendein Weg, S*	*Bester Weg, S'*
mit Heuristik σ	*Heurist. Suche, irgendein Weg, S*	*S'' (A*)*

Gierige Suche (greedy search)

Allgemein: Eine aktuell beste Bewertung bevorzugen

In Suchverfahren:
Expansion bei aktuell besten Werten für g' und/oder σ

Varianten:
- OPEN vollständig sortieren (vgl. Bestensuche)
- NEW(z) sortiert an Anfang von OPEN (vgl. Bergsteigen)
- Nur den besten Zustand weiter verfolgen
  L**ocal greedy:** Variante von Bergsteigen

„Lokale Optimierung"

Probleme bzgl. Vollständigkeit/Korrektheit

Theoretische Grundlagen von PROLOG

Probleme:

Automatische Beweisverfahren
Was leistet der PROLOG-Interpreter ?

„Program = Logic + Control"

Prozedurale/imperative Sprachen:
• Abläufe formulieren
• Computer führt aus

von-Neumann-Maschine

Idee von deklarativen/logischen/funktionalen Programmiersprachen:
- Zusammenhänge formulieren
- Computer erschließt weitere Zusammenhänge

Built-in Prädikate

Programm:
Nutzerdefinierte Prädikate

Interpreter:
Laufzeitsystem, Beweis-Maschine

„Program = Logic + Control"

Idee von deklarativen/logischen/funktionalen Programmiersprachen:
- Zusammenhänge formulieren
- Computer erschließt weitere Zusammenhänge

Deklarative Semantik

Built-in Prädikate

Programm:
Nutzerdefinierte Prädikate

Interpreter:
Laufzeitsystem, Beweis-Maschine

Ablauf der Beweismaschine:

Prozedurale Semantik

Ziele der Software-Technologie:

... Der zweite Aspekt erfordert eine Sprache,
die im Idealfall ,,nahe am Problem" ist,
so daß die Konzepte der Problemlösung direkt und schlüssig formuliert werden können. ...
Die Verbindung zwischen der Sprache,
in der wir programmieren/denken,
und den Problemen und Lösungen,
die wir uns vorstellen können, ist sehr eng. ...

Bjarne Stroustrup:
Die C++ Programmiersprache, S. 10
(,,Philosophische Bemerkung")

Probleme mit Software:

Softwaresysteme gehören zu den komplexesten Gebilden,
die je von Menschenhand geschaffen wurden.
Strukturen und Abläufe in großen Systemen sind
im einzelnen oft nicht mehr überschaubar.
Man kann sie weder im vorhinein, beim Entwurf,
noch im nachhinein, beim Testen, beim Betrieb
und in der Wartung vollständig verstehen.

Denert: Software-Engeneering, S. 4

Probleme mit Software:

Das entscheidende Charakteristikum der industriell
einsetzbaren Software ist, dass es für den einzelnen
Entwickler sehr schwierig, wenn nicht gar unmöglich ist,
alle Feinheiten des Designs zu verstehen.
Einfach ausgedrückt, überschreitet die Komplexität
solcher Systeme die Kapazität der menschlichen Intelligenz.

Booch: Objektorientierte Analyse und Design

Prolog basiert auf Prädikatenlogik

Vorbild für „Formalismus“:
– exakt, präzise, (theoretisch) beherrscht

Aufbau:
–Zeichen
–Ausdrücke (rekursive Definition)
–Sätze („Theorie“) **Th**
•syntaktisch bestimmt: **Th** = Abl(**Ax**)
•semantisch bestimmt: **Th** = allgemeingültige Sätze einer Struktur

Nach speziellen formalen Regeln erzeugbare Formeln

Beweise

„Irreflexive, transitive Relationen sind asymmetrisch“

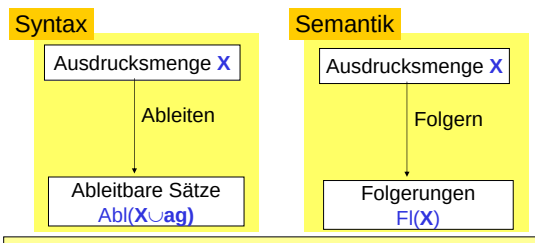
Inhaltlicher Beweis:
– Argumentation

Formaler (syntaktischer) Beweis:
– Umformung von Ausdrücken („Kalkül“)

Theorembeweiser:
– (Syntaktisches) Verfahren zur Entscheidung, ob ein Ausdruck zu einer Satzmenge (Theorie) gehört:

$H \in Th$?

Axiomatische Behandlung der Logik



Korrektheit von Abl : $Abl(X \cup ag) \subseteq Fl(X)$
Vollständigkeit von Abl : $Abl(X \cup ag) \supseteq Fl(X)$
Äquivalenz von Abl : $Abl(X \cup ag) = Fl(X)$

Formales Ableiten (Resolutionsregel)

Für Klauseln („Disjunktion von Literalen“)

Voraussetzung:
K1 und K2 unifizierbar mittels Unifikator σ

K =Res(K1,K2,σ) entsteht durch
– „Vereinigung“ von σ(K1) und σ(K2)
– Streichen komplementärer Literale

Formales Ableiten (Resolution)

KA1: $\neg R(x,x)$

KA2: $\neg R(u,y) \vee \neg R(y,z) \vee R(u,z))$

K1: $R(c,f(c))$

K2: $R(f(c),c)$

K3 =Res(KA1,KA2,σ): $\neg R(w,y) \vee \neg R(y,w)$
mit $\sigma(u)= \sigma(z)= \sigma(x)= w$

K4 = Res(K1,K3,σ): $\neg R(f(c),c)$
mit $\sigma(w)=c \quad \sigma(y)=f(c)$

K5 = Res(K2,K4,σ):

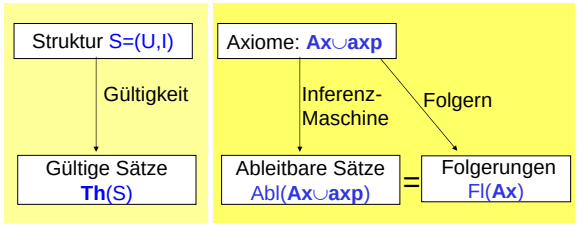
Enscheidbarkeit in der Logik

(rein logisch) allgemeingültige Sätze:
ag = Abl(**axp**) = Fl(∅)

$H \in$ **ag** ?

- Entscheidbar im AK
- Unentscheidbar im PK1
  (aber aufzählbar, da axiomatisierbar)

Formalisierung einer Domäne

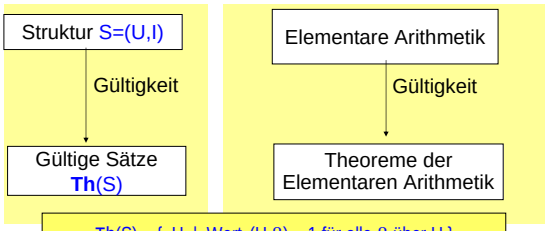


Korrektheit der Formalisierung : $Abl(\mathbf{Ax} \cup \mathbf{axp}) \subseteq \mathbf{Th}(S)$

Vollständigkeit der Formalisierung: $Abl(\mathbf{Ax} \cup \mathbf{axp}) \supseteq \mathbf{Th}(S)$

Äquivalenz der Formalisierung : $Abl(\mathbf{Ax} \cup \mathbf{axp}) = \mathbf{Th}(S)$

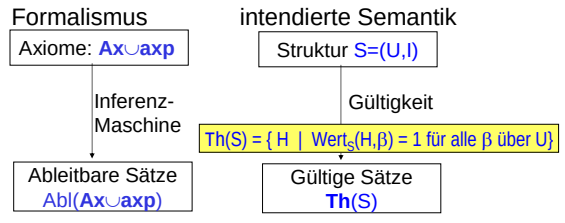
Beispiel: Formalisierung der Arithmetik



$Th(S) = \{ H \mid Wert_S(H,\beta) = 1 \text{ für alle } \beta \text{ über } U \}$

β : Belegung der Variablen
mit Individuen aus dem Universum U

Formalisierung einer Domäne

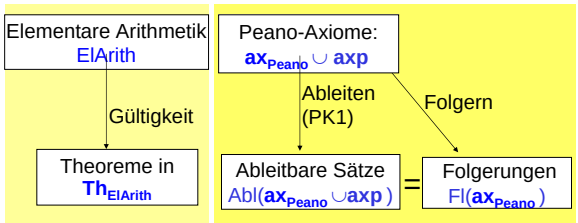


Korrektheit der Formalisierung : $Abl(\mathbf{Ax} \cup \mathbf{axp}) \subseteq \mathbf{Th}(S)$

Vollständigkeit der Formalisierung: $Abl(\mathbf{Ax} \cup \mathbf{axp}) \supseteq \mathbf{Th}(S)$

Äquivalenz der Formalisierung : $Abl(\mathbf{Ax} \cup \mathbf{axp}) = \mathbf{Th}(S)$

Beispiel: Formalisierung der Arithmetik



Korrektheit der Formalisierung :
$Abl(\mathbf{Ax}_{Peano} \cup \mathbf{axp}) \subseteq \mathbf{Th}_{ElArith}$

Unvollständigkeit der Formalisierung:
$Abl(\mathbf{Ax}_{Peano} \cup \mathbf{axp}) \neq \mathbf{Th}_{ElArith}$

Adäquatheit der Formalisierung

Adäquatheit:
Die wesentlichen Aspekte werden in der Struktur S bzw. den Axiomen **Ax** korrekt erfaßt.

–Parallelen-Axiom der Geometrie

–Stetigkeitsdefinition in der Analysis
–Modellierung eines Staubsaugers

Syntax des PK1

Terme: Individuen (Konstante, Variable, Funktionen)
Prädikate (atomare Formeln):
Relationen $R(x_1,...,x_n)$ (wahr/falsch)
Ausdrücke:
logische Beziehungen zwischen Prädikaten mittels
– Aussagenlogischen Operatoren $\neg \wedge \vee \leftrightarrow \rightarrow$
– Quantifikation von Variablen $\forall \exists$
z.B. $\forall x \exists y\, R(x, x_1,...,x_n) \wedge \neg R(y, x_1,...,x_n)$

Positives Literal: nicht negierte atomare Formel
Negatives Literal: negierte atomare Formel

Syntax des PK1

Ableiten: Ausdrücke umformen mit Ableitungsregeln

z.B. Abtrennungsregel (modus ponens)

$$\frac{H_1 \ , \ H_1 \rightarrow H_2}{H_2}$$

H ableitbar aus **X ,**
falls Ableitungsfolge für H aus **X** existiert
X |- H oder: H ∈ **X**|- oder: H ∈ Abl(**X**)