Formale Techniken der Software-Entwicklung

Matthias Hölzl, Christian Kroiß

7. Juli 2014

Formale Techniken

Verwendung von *mathematischen Hilfsmitteln* zur *Entwicklung* von Hardware und Software

- Logik
- Automatentheorie
- Typsysteme
- Formale Sprachen
- Programmsemantik
- Wahrscheinlichkeitstheorie, Statistik

Verschiedene Logiken

- Quantoren
  - Aussagenlogik
  - Logik erster Stufe (Prädikatenlogik)
  - Logik höherer Stufe (Typtheorie)
- Konstruktivität
  - Intuitionistische Logik
  - Klassische Logik
- Modalitäten
  - Klassische Logik (ohne Modalitäten)
  - Temporallogik
  - Modallogik

Aussagenlogik: Syntax

- Menge von Aussagenvariablen $\mathcal{A} = \{A, B, C, \dots\}$
- Logische Symbole (Junktoren): $\neg, \wedge, \vee, \implies, \iff$
- Hilfssymbole: $(,)$
- Ausagenlogische Formeln (Terme) werden rekursiv aus Variablen, Junktoren und Hilfssymbolen gebildet

Aussagenvariablen (propositionale Variablen)

- Stehen für ganze Aussagen
- Können nur die Werte wahr oder falsch annehmen

Keine Individuenvariablen, keine Quantoren, keine modalen/temporalen Operatoren

Aussagenlogische Formeln

$\mathcal{F}$	$=$	$\mathcal{A}$	Variable
	$\mid$	$\top$	Top, Verum, wahr
	$\mid$	$\bot$	Bottom, Falsum, falsch
	$\mid$	$(\neg\mathcal{F})$	Negation, Verneinung, nicht
	$\mid$	$(\mathcal{F} \wedge \mathcal{F})$	Konjunktion, und
	$\mid$	$(\mathcal{F} \vee \mathcal{F})$	Disjunktion, oder
	$\mid$	$(\mathcal{F} \implies \mathcal{F})$	Implikation, wenn-dann
	$\mid$	$(\mathcal{F} \iff \mathcal{F})$	Äquivalenz, genau-dann-wenn

Formeln stehen für Parse-Bäume, Klammern werden weggelassen

Semantik

Durch eine *Semantik* wird einem syntaktischen Objekt eine Bedeutung gegeben. Zum Beispiel wird in der denotationellen Semantik jedem Programm eine mathematische Funktion zugeordnet.

Die Semantik der Aussagenlogik ist im Verhältnis dazu relativ einfach: Jeder aussagenlogischen Formel soll dadurch ein Wahrheitswert zugeordnet werden.

Belegungen

Um Wahrheitswerte von *Formeln* mit aussagenlogischen Variablen angeben zu können, müssen wir den *Variablen* Wahrheitswerte zuweisen. Das erfolgt mit Hilfe von sogenannten *Belegungen* η, die Variablen in Werte aus $\mathbb{B}$ abbilden:

$$\eta : \mathcal{A} \to \mathbb{B}$$

Damit können wir überprüfen, wie die Wahrheitswerte der Variablen den Wahrheitswert der Formel beeinflussen.

Semantik von Formeln

Wir können jetzt die Definition der Semantik von Formeln in mathematischer Notation aufschreiben und auf Variablen erweitern. Dazu definieren wir eine Funktion $[\![\cdot]\!]\eta : \mathcal{F} \to \mathbb{B}$, die von einer Belegung abhängt und Formeln auf Wahrheitswerte abbildet.

$$
\begin{aligned}
[\![\top]\!]\eta &= \text{wahr} \\
[\![\bot]\!]\eta &= \text{falsch} \\
[\![\alpha]\!]\eta &= \eta(\alpha) \quad \text{für } \alpha \in \mathcal{A} \\
[\![\neg\phi]\!]\eta &= !([\![\phi]\!]\eta) \\
[\![\phi \wedge \psi]\!]\eta &= [\![\phi]\!]\eta \,\&\, [\![\psi]\!]\eta \\
[\![\phi \vee \psi]\!]\eta &= [\![\phi]\!]\eta \mid [\![\psi]\!]\eta \\
[\![\phi \implies \psi]\!]\eta &= !([\![\phi]\!]\eta) \mid [\![\psi]\!]\eta \\
[\![\phi \iff \psi]\!]\eta &= [\![\phi]\!]\eta = [\![\psi]\!]\eta
\end{aligned}
$$

Tautologien

Definition

Eine Formel ϕ ist eine Tautologie, wenn sie für alle Belegungen wahr ist, wenn also gilt

$$\forall \eta : \mathcal{A} \to \mathbb{B} : [\![\phi]\!]\eta = \text{wahr}$$

$$A \vee \neg A$$

$$A \implies \neg\neg A$$

$$A \iff \neg\neg A$$

$$(A \implies B) \iff (\neg B \implies \neg A)$$

$$(A \implies B) \wedge (B \implies C) \implies (A \implies C)$$

Erfüllbarkeit

Eine Formel ϕ ist *erfüllbar* (satisfiable), wenn sie für mindestens eine Belegung wahr ist, wenn also gilt

$$\exists \eta : \mathcal{A} \to \mathbb{B} : [\![\phi]\!]\eta = \text{wahr}$$

Eine Formel ist *unerfüllbar* (unsatisfiable), wenn sie nicht erfüllbar ist, wenn also gilt

$$\neg\exists \eta : \mathcal{A} \to \mathbb{B} : [\![\phi]\!]\eta = \text{wahr}$$

oder äquivalent

$$\forall \eta : \mathcal{A} \to \mathbb{B} : [\![\phi]\!]\eta = \text{falsch}$$

Zusammenhänge

- ϕ ist eine Tautologie, genau dann, wenn $\neg\phi$ unerfüllbar ist
- ϕ ist eine Tautologie, genau dann, wenn ϕ äquivalent zu $\top$ ist
- ϕ ist unerfüllbar, genau dann, wenn ϕ äquivalent zu $\bot$ ist
- ϕ ist genau dann äquivalent zu ψ, wenn $\phi \Leftrightarrow \psi$ eine Tautologie ist

Wahrheitstabellen

A	$\neg A$	$A \vee \neg A$	$A \wedge \neg A$	$A \Rightarrow \neg A$
falsch	wahr	wahr	falsch	wahr
wahr	falsch	wahr	falsch	falsch

Der Wahrheitswert einer Formel lässt sich durch eine Wahrheitstabelle für jede Belegung bestimmen.

- Eine Formel ist genau dann eine Tautologie, wenn in der letzten Spalte ihrer Wahrheitstabelle nur der Wert wahr vorkommt.
- Eine Formel ϕ ist genau dann erfüllbar, wenn in der letzten Spalte ihrer Wahrheitstabelle mindestens einmal der Wert wahr vorkommt.
- Eine Formel ϕ ist genau dann unerfüllbar, wenn in der letzten Spalte ihrer Wahrheitstabelle nur der Wert falsch vorkommt.
- Zwei Formeln ϕ und ψ sind genau dann äquivalent, wenn in den ϕ und ψ entsprechenden Spalten einer gemeinsamen Wahrheitstabelle immer der gleiche Wert vorkommt.

Sequenzen

- Eine Sequenz besteht aus zwei Folgen von aussagenlogischen Termen, dem *Antezedens* und dem *Sukzedens* (oder den *Konsequenzen*, *Folgerungen*)
- Eine Sequenz wird in der Form

$$\frac{A_1, A_2, \ldots, A_m}{B_1, B_2, \ldots, B_n}$$

oder

$$A_1, A_2, \ldots, A_m \vdash B_1, B_2, \ldots, B_n$$

geschrieben

- Man schreibt oft A_1, Γ für $A_1, A_2, \ldots, A_m$
- Eine Sequenz $A_1, A_2, \ldots, A_m \vdash B_1, B_2, \ldots, B_n$ bedeutet intuitiv

$$A_1 \wedge A_2 \wedge \cdots \wedge A_m \implies B_1 \vee B_2 \vee \cdots \vee B_n$$

Bedeutung von Sequenzen

Aus der Bedeutung von $A_1, A_2, \ldots, A_m \vdash B_1, B_2, \ldots, B_n$

$$A_1 \wedge A_2 \wedge \cdots \wedge A_m \implies B_1 \vee B_2 \vee \cdots \vee B_n$$

ergibt sich: Eine Sequenz $\dfrac{A_1, A_2, \ldots, A_m}{B_1, B_2, \ldots, B_n}$ ist wahr, wenn

- Ein A_i mit einem B_j identisch ist
- Ein A_i falsch ist
- Ein B_j wahr ist

Inferenzregeln, Formaler Beweis

Regeln zur syntaktischen Umformung von Sequenzen, die die gewünschte Semantik "implementieren"

$$\frac{\Gamma_1 \vdash \Delta_1 \cdots \Gamma_n \vdash \Delta_n}{\Gamma \vdash \Delta}\mathbf{R}$$

Durch mehrmalige Anwendung von Inferenzregeln ergibt sich ein Beweisbaum

$$\dfrac{\dfrac{\dfrac{\dots}{\Gamma_i \vdash \Delta_i} \cdots \dfrac{\dots}{\Gamma_j \vdash \Delta_j}}{\Gamma_k \vdash \Delta_k} \cdots \dfrac{\dfrac{\dots}{\Gamma_l \vdash \Delta_l} \cdots \dfrac{\dots}{\Gamma_m \vdash \Delta_m}}{\Gamma_n \vdash \Delta_n}}{\Gamma \vdash \Delta}$$

Die Wurzel des Beweisbaums ist die zu beweisende Regel, die Blätter sind Axiome oder leer

Sequenzenkalkül (1)

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

$$\frac{}{\Gamma, \phi \vdash \phi, \Delta}\mathbf{Ax}$$

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

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

Sequenzenkalkül (2)

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

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

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

De Morgan'sche Regeln

$$\neg(\phi \wedge \psi) \iff \neg\phi \vee \neg\psi$$
$$\neg(\phi \vee \psi) \iff \neg\phi \wedge \neg\psi$$

Semantischer Folgerungsbegriff

$\psi_1, \ldots, \psi_n$ folgt semantisch aus $\phi_1, \ldots \phi_m$

$$\phi_1, \ldots, \phi_m \models \psi_1, \ldots, \psi_n$$

genau dann, wenn jede Belegung, die alle ϕ_i erfüllt auch (mindestens) ein ψ_j erfüllt, wenn also gilt

$$\bigwedge \phi_1, \ldots, \phi_m \models \bigvee \psi_1, \ldots, \psi_n$$

Deduktionstheorem

Seien Γ eine (endliche) Menge von Formeln, ϕ und ψ Formeln. Dann gilt

$$\Gamma, \phi \models \psi \tag{1}$$

genau dann, wenn

$$\Gamma \models \phi \implies \psi \tag{2}$$

gilt.

Ableitung im Sequenzenkalkül

Eine Formel ψ ist im Sequenzenkalkül ableitbar, wenn es eine Herleitung von ψ mit leerem Antezedens gibt:

$$\begin{aligned} \phi_1, \ldots, \phi_m \vdash \psi_1, \ldots, \psi_n &\equiv \neg\phi_1 \vee \cdots \vee \neg\phi_m \vee \psi_1 \vee \ldots, \vee\psi_n \\ \vdash \psi &\equiv \psi \end{aligned}$$

Eine Implikation $\phi \implies \psi$ ist im Sequenzenkalkül genau dann ableitbar, wenn die Sequenz $\phi \vdash \psi$ ableitbar ist.

$$\begin{aligned} \vdash \left(\phi_1 \wedge \cdots \wedge \phi_m \implies \psi_1 \vee \ldots, \vee\psi_n\right) & \\ \equiv \phi_1 \wedge \cdots \wedge \phi_m \implies \psi_1 \vee \ldots, \vee\psi_n & \\ \equiv \neg\phi_1 \vee \cdots \vee \neg\phi_m \vee \psi_1 \vee \ldots, \vee\psi_n & \\ \equiv \phi_1, \ldots, \phi_m \vdash \psi_1, \ldots, \psi_n & \end{aligned}$$

Notation

Eine Sequenz $\Gamma \vdash \Delta$ ist im Sequenzenkalkül ableitbar:

$$\vdash_{\text{seq}} \Gamma \vdash \Delta \quad \text{oder} \quad \Gamma \vdash_{\text{seq}} \Delta$$

Eine Formel ϕ ist im Sequenzenkalkül ableitbar:

$$\vdash_{\text{seq}} \phi$$

Korrektheit und Vollständigkeit

- Korrektheit

$$\Gamma \left|\!\!\frac{}{\text{seq}}\right. \Delta \quad \text{impliziert} \quad \Gamma \models \Delta$$

- Vollständigkeit

$$\Gamma \models \Delta \quad \text{impliziert} \quad \Gamma \left|\!\!\frac{}{\text{seq}}\right. \Delta$$

- Der Sequenzenkalkül für die Aussagenlogik ist korrekt und vollständig

PVS Syntax

```
name : THEORY
  BEGIN

  V1, V2: bool

  P1: PROPOSITION
      V1 OR NOT(V1)
  P2: THEOREM
      (V1 => V2) <=> (NOT(V2) => NOT(V1))

  END name
```

Beispiel

```
lecture_1 : THEORY
  BEGIN

  A, B, C: bool

  tertium_non_datur: PROPOSITION A OR NOT (A)
  double_negation_1: PROPOSITION A => NOT(NOT(A))
  double_negation_2: PROPOSITION A <=> NOT(NOT(A))
  counterpositive:   PROPOSITION
                       (A => B) <=> (NOT(B) => NOT(A))
  transitivity:      PROPOSITION
                       (A => B) AND (B => C) => (A => C)
  END lecture_1
```

Sequenzenkalkül in PVS

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

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

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

`(flatten)` `(split)`

SAT

Gegeben sei eine aussagenlogische Formel ϕ. Das *Erfüllbarkeitsproblem (SAT)* ist die Frage nach einer erfüllenden Belegung: Gibt es ein η mit

$$\models_\eta \phi?$$

SAT war das erste Problem, das als *NP-vollständig* bewiesen wurde.

NP

Ein *Prüfprogramm (Verifier)* für eine Sprache $\mathcal{A}$ ist ein Algorithmus V, für den gilt

$$\mathcal{A} = \{w \mid V \text{ akzeptiert } \langle w, c \rangle \text{ für eine Zeichenkette } c\}$$

c heißt *Zertifikat* oder *Beweis*.

NP ist die Klasse der Sprachen, die in polynomialer Zeit verifiziert werden können. (Das heißt, dass Zertifikate maximal polynomiale Länge haben dürfen.)

NP-Vollständigkeit

Eine Sprache B ist *NP-vollständig*, wenn

- sie in NP enthalten ist und
- jedes A in NP in polynomialer Zeit auf B reduziert werden kann.

Theorem (Cook-Levin)

SAT ist NP-vollständig

Konjunktive Normalform

- Ein *Literal* L ist eine Variable A oder eine negierte Variable $\neg A$

$$L = A \mid \neg A$$

- Eine Klausel K ist eine Disjunktion von Literalen:

$$K = L_1 \vee L_2 \vee \cdots \vee L_m$$

- Eine Formel C ist in konjunktiver Normalform (KNF, CNF), wenn sie eine Konjunktion von Klauseln ist:

$$\begin{aligned} C &= K_1 \wedge K_2 \wedge \cdots \wedge K_n \\ &= (L_{11} \vee \cdots \vee L_{1p}) \wedge \cdots \wedge (L_{k1} \vee \cdots \vee L_{kp}) \end{aligned}$$

Anwendung von CNF

- Überprüfen von Allgemeingültigkeit
- Eine Klausel ist allgemeingültig genau dann, wenn sie Literale L_i, L_j enthält, so dass $L_i = \neg L_j$
- Eine Formel in CNF ist allgemeingültig, wenn jede ihrer Klauseln allgemeingültig ist
- Beispiel:

$$(A \vee B \vee \neg C \vee \neg B) \wedge (B \vee \neg C \vee E \vee F)$$

  ist nicht allgemeingültig, da die zweite Klausel nicht allgemeingültig ist

Anwendung von CNF: Inkonsistenz

- Überprüfen von Inkonsistenz
- Eine Formel in CNF ist genau dann inkonsistent, wenn die leere Klausel ableitbar ist
- Beispiel:

$$\{\{L_1, L_2, \neg L_3, \neg L_2\}, \emptyset, \{L_2, \neg L_3, L_4, L_5\}\}$$

  ist inkonsistent, da eine leere Klausel enthalten ist

Unit-Resolution

In den Übungen zum DPLL-Algorithmus wurde die folgende Ableitungsregel für Klauseln (Unit-Resolution) betrachtet: Wenn M und L_i komplementäre Literale sind, dann gilt

$$\frac{L_1, \ldots, L_m \qquad M}{L_1, \ldots, L_{i-1}, L_{i+1}, \ldots, L_m}$$

(Dass diese Regel korrekt ist überlegt man sich so: Da M und L_i komplementär sind kann man die linke Seite der Prämisse als

$$M \Rightarrow L_1, \ldots, L_{i-1}, L_{i+1}, \ldots, L_m$$

schreiben, und aus M und $M \Rightarrow \phi$ folgt ϕ.)

Resolution

Die Beweisregel zur Unit-Resolution lässt sich folgendermaßen verallgemeinern: Falls L_i und M_j komplementär sind (d.h. $L_i \equiv \neg M_j$ oder $\neg L_i \equiv M_j$) gilt

$$\frac{L_1, \ldots, L_m \qquad M_1, \ldots, M_n}{L_1, \ldots, L_{i-1}, L_{i+1}, \ldots, L_m, M_1 \ldots, M_{j-1}, M_{j+1}, M_n}$$

Diese Form der Ableitung heißt (allgemeine) Resolution.
$L_1, \ldots, L_{i-1}, L_{i+1}, \ldots, L_m, M_1 \ldots, M_{j-1}, M_{j+1}, M_n$ heißt *Resolvente*.

Die hier angegebene Resolutionsregel arbeitet immer auf Klauseln. Falls $L_1, \ldots, L_m$ und $M_1, \ldots, M_n$ keine komplementären Literale haben, so kann die Resolutionsregel nicht angewendet werden.

Resolution: Korrektheit

L_i, M_j komplementär ($L_i \equiv \neg M_j$ oder $\neg L_i \equiv M_j$):

$$\frac{L_1, \ldots, L_m \qquad M_1, \ldots, M_n}{L_1, \ldots, L_{i-1}, L_{i+1}, \ldots, L_m, M_1 \ldots, M_{j-1}, M_{j+1}, M_n}$$

Die Resolvente ist *nicht* äquivalent zu den Ausgansklauseln. Sei z.B. η eine Belegung in der nur L_1 wahr ist. Dann ist die Resolvente wahr aber die Prämisse $M_1, \ldots, M_n$ falsch.

Aber: $L_1, \ldots, L_m$ und $M_1, \ldots, M_n$ sind nur dann beide erfüllbar, wenn die Resolvente erfüllbar ist; wenn also $L_1, \ldots, L_m$ und $M_1, \ldots, M_n$ beide wahr sind, so ist auch die Resolvente wahr.

(Das kann man sich so überlegen: Sei L_i wahr. Dann ist M_j falsch; damit $M_1, \ldots M_n$ gilt, musss also $M_1 \ldots, M_{j-1}, M_{j+1}, M_n$ gelten. Falls L_i falsch ist, kann man entsprechend mit $L_1, \ldots, L_m$ argumentieren.)

Resolution: Korrektheit

Seien C_1, C_2 Formeln in CNF. Dann haben C_1 und C_2 die Form $C_1 = K_1 \wedge \cdots \wedge K_m$, $C_2 = K'_1 \wedge \cdots \wedge K'_n$ mit Klauseln K_i, K'_j.

Wir sagen C_2 ist durch Resolution aus C_1 ableitbar und schreiben

$$C_1 \left|\frac{}{\text{res}}\right. C_2$$

wenn es für jede Klausel K'_j einen Ableitungsbaum gibt, in dem nur die Resolutionsregel verwendet wird, und an dessen Blättern nur die Klauseln $K_1, \ldots, K_m$ stehen.

Die Aussage auf der vorhergehenden Folie bedeutet, dass die Ableitungsbeziehung $C_1 \left|\frac{}{\text{res}}\right. C_2$ korrekt ist: aus wahren Prämissen kann durch Resolution keine falsche Aussage gezeigt werden.

Resolution: Vollständigkeit

Mit dem Sequenzenkalkül haben wir eine korrekte und vollständige Ableitungsregel kennengelernt:

$$\phi \models \psi \quad \text{genau dann, wenn} \quad \phi \left|\overline{\vdash_{\text{seq}}}\right. \psi$$

Jede in einer „Wissensbasis“ ϕ wahre Formel ψ kann also mit dem Sequenzenkalkül aus ϕ abgeleitet werden (und es können keine falschen Formeln abgeleitet werden).

Gilt für den Resolutionskalkül eine ähnliche Eigenschaft? Man kann sich, z.B. fragen, ob jede wahre Formel C_2 in CNF aus einer Wissensbasis C_1, die nur aus Klauseln besteht, abgeleitet werden kann:

$$\text{Frage:} \quad C_1 \models C_2 \quad \text{genau dann, wenn} \quad C_1 \left|\overline{\vdash_{\text{res}}}\right. C_2 \quad ?$$

Resolution

Resolution ist *nicht* vollständig: Sei C eine Formel in CNF (also eine Menge von Klauseln) und K eine Klausel. Aus $C \models K$ folgt nicht $C \vdash_{\text{res}} K$.

Beispiel: Da $A \vee \neg A$ allgemeingültig ist (also $\models A \vee \neg A$ wahr ist), gilt natürlich auch $B, \neg C, D \models A \vee \neg A$. Nachdem in $\{B, \neg C, D\}$ aber keine komplementären Literale vorkommen ist die Resolutionsregel gar nicht anwendbar, man kann also $A \vee \neg A$ nicht ableiten.

Resolution

Resolution ist aber *widerspruchsvollständig*: wenn C inkonsistent ist, dann lässt sich $\bot$ durch Resolution ableiten:

$$\text{Wenn} \quad C \models \bot \quad \text{dann} \quad C \left|\frac{}{\text{res}}\right. \bot$$

Das ist aber schon genug, um jeden aussagenlogischen Schluss (zwischen Formeln in CNF) durch Resolution beweisen zu können:

$$C_1 \models C_2 \quad \text{genau dann, wenn} \quad C_1 \wedge \neg C_2 \models \bot$$

(Die linke Seite sagt, dass in jeder Belegung, in der C_1 wahr ist auch C_2 wahr ist. Ist dass der Fall, so kann es keine Belegung geben, in der C_1 wahr ist und C_2 falsch; das ist genau die Aussage auf der rechten Seite. Offensichtlich gilt auch die umgekehrte Richtung.)

Widerspruchsvollständigkeit der Resolution

Theorem (Grundresolutionstheorem)

Wenn eine Menge von Klauseln C unerfüllbar ist, dann enthält der Resolutionsabschluss von C, $\mathrm{RC}(C)$ die leere Klausel.

Theorembeweisen durch Resolution

Beim Theorembeweisen durch Resolution geht man folgendermaßen vor:

Gezeigt werden soll $\phi \models \psi$.

- Wandle $\phi \wedge \neg\psi$ in CNF um
- Versuche daraus $\bot$ (die leere Klausel) durch Resolution abzuleiten

Gelingt das, ist $\phi \wedge \neg\psi$ widersprüchlich, somit gilt $\phi \models \psi$.

Gelingt das nicht ist $\phi \wedge \neg\psi$ konsistent, $\phi \models \psi$ gilt damit nicht. Aus dem Beweis des Grundresolutionstheorems lässt sich eine Belegung η konstruieren, die ϕ und $\neg\psi$ erfüllt.

DPLL

- Lösungsverfahren für SAT Probleme
- Kann fast alle in der Praxis vorkommenden SAT Probleme effizient lösen
- Basis: Backtracking-Algorithmus
- Effizientere Implementierung: „klassisches“ DPLL
  - Iterativer Algorithmus
  - Frühzeitiger Abbruch
  - Elimination von reinen Literalen (pure symbol heuristic, pure literal elimination)
  - Unit-Propagation (unit clause heuristic)
- Noch Effizientere Implementierung: „modernes“ DPLL
  - Heuristik zur Variablenauswahl
  - Conflict-directed Backjumping
  - Lernen von Klauseln
  - Beobachtete Literale

Backtracking-Suche

- Zustandsraum: alle möglichen (partiellen) Belegungen der Variablen
- Rekursive Aufzählung des Zustandsraumes
- Faktorisierte Darstellung der Zustände: $\{A_1 = \eta_1, \ldots, A_k = \eta_k\}$
- Suchoperatoren: Hinzufügen einer neuen Variablenbelegung

$$\{A_1 = \eta_1, \ldots, A_k = \eta_k\} \rightarrow \{A_1 = \eta_1, \ldots, A_k = \eta_k, A_{k+1} = \eta_{k+1}\}$$

- Feste Ordnung der Variablen

Verbesserung: Frühzeitiger Abbruch der Suche

Die erste Implementierung der Backtracking-Suche (`ttTautology` für Allgemeingültigkeit und $\mathrm{ttSatisfiable}$ für Erfüllbarkeit) erzeugt immer eine vollständige Belegung der Variablen und wertet dann den Term aus.

In vielen Fällen kann man die Suche schon früher abbrechen:

$$A \vee \neg A \vee (B \wedge C) \vee (A \wedge B)$$
$$(A \vee B) \wedge (\neg A \wedge C \wedge D) \wedge \neg B$$

Beim ersten Term ist schon nach der Belegung von A durch einen beliebigen Wert klar, dass der Term wahr ist; beim zweiten Term kann man nach der Wahl von A und B schon feststellen, dass kein Wert von C oder D den Term noch erfüllen kann.

Auswertung bei partieller Belegung

Um mit partiellen Variablenbelegungen umgehen zu können kann man die Wahrheitstabellen folgendermaßen erweitern:

	$\neg$
falsch	wahr
wahr	falsch
undef	undef

$\wedge$	falsch	wahr	undef
falsch	falsch	falsch	falsch
wahr	falsch	wahr	undef
undef	falsch	undef	undef

$\vee$	falsch	wahr	undef
falsch	falsch	wahr	undef
wahr	wahr	wahr	wahr
undef	undef	wahr	undef

DPLL und CNF

- Der grundlegende DPLL Algorithmus ist für beliebige Formeln verwendbar
- Die Auswertung ist nicht effizient
- Für weitere Verbesserungen benötigen wir mehr Information über die Struktur der Terme
- Diese Struktur bekommen wir z.B. durch CNF
- Keine Änderung am Algorithmus *nötig*, sofern wir in CNF das Interface implementieren, das zum Backtracking benötigt wird
- Viele Verbesserungen *möglich*, wenn wir auf CNF umsteigen

Propagation in DPLL

Beispiel: Wenn eine Klausel nur ein Literal L enthält, so muss die entsprechende Variable A `true` oder `false` sein, je nachdem ob L positiv (nicht negiert, d.h. $L \equiv A$) oder negativ (negiert, d.h. $L \equiv \neg A$) ist.

Ein solches Literal kann man dann aus anderen Klauseln entfernen; das entspricht genau der Anwendung der Unit-Propagationsregel.

Propagation von Werten ist auch auf Probleme anwendbar, die allgemeine Relationen zur Beschreibung eines Sachverhalts verwenden (sog. Constraint-Probleme).

Relationen/Constraints

Definition

Sei $X = \{x_1, \ldots, x_n\}$ eine Menge von Variablen mit Domänen $D = \{D_1, \ldots, D_n\}$.
Eine Relation R über $S \subseteq X$ ist eine Teilmenge von $\prod_{x_s \in S} D_s$; S heißt *Scope* von R (scope(R)).
Ein Constraint ist eine Relation.

Grundlegende Theorien:

- Relationale Algebra
- Graphentheorie

Constraint-Netzwerke

Definition

Ein Constraint-Netzwerk $\mathfrak{N} = (X, D, C)$ besteht aus einer endlichen Menge $X = \{x_1, \ldots, x_n\}$ von Variablen mit Domänen $D = \{D_1, \ldots, D_n\}$ und einer Menge von Constraints $C = \{C_1, \ldots, C_k\}$. Die Menge aller Scopes von Constraints heißt das *Schema* des Constraint-Netzwerks.

Ein Constraint beschreibt, welche Werte für die Variablen aus seinem Scope gleichzeitig zulässig sind.

Beispiel: Karte von Australien

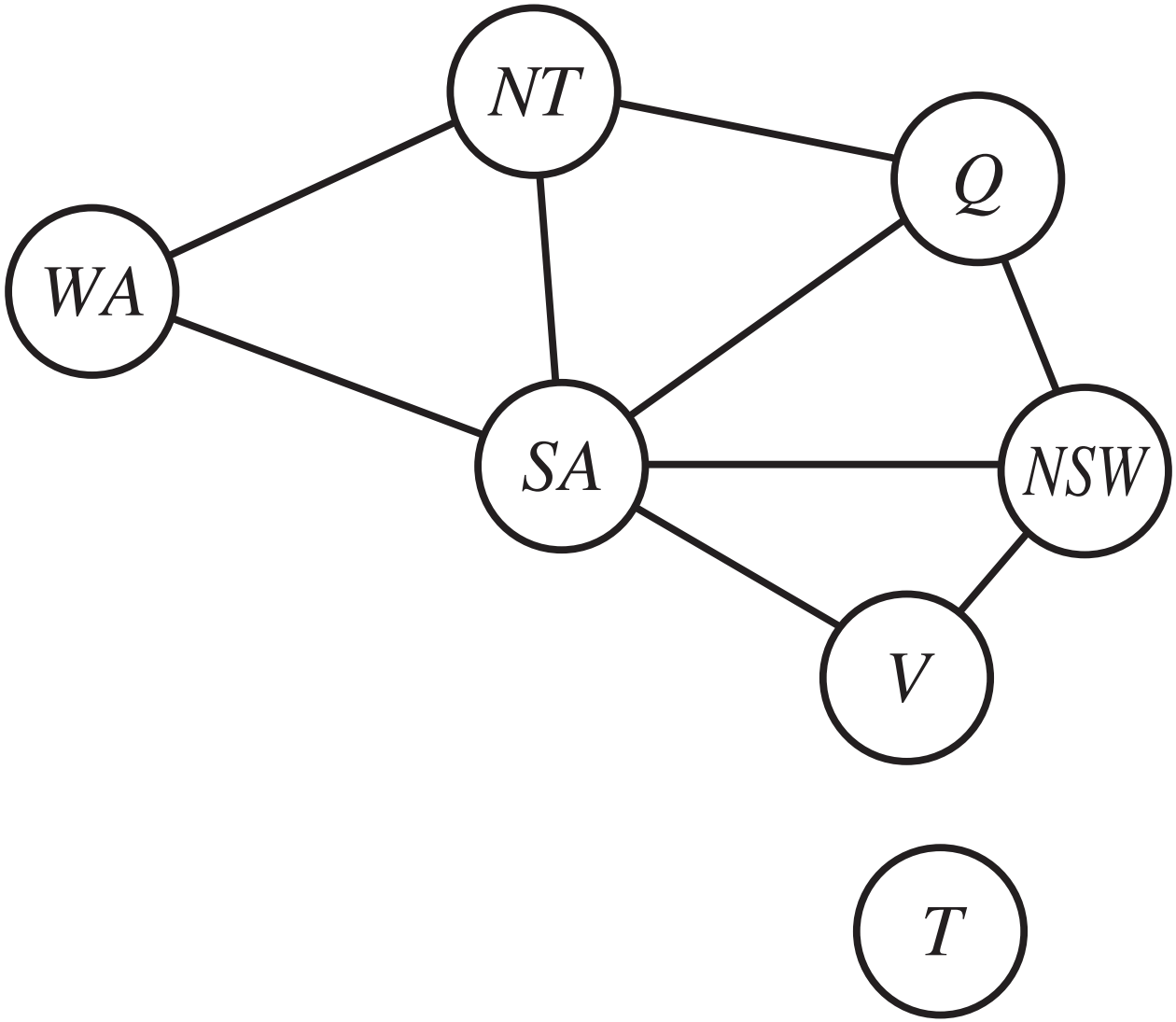
- $X = \{WA, NT, Q, NSW, V, SA, T\}$
- $D_i = \{R, G, B\}$
- $C = \{WA \neq NT, WA \neq SA, SA \neq V, SA \neq NSW,$
  $SA \neq Q, SA \neq NT, NT \neq Q, Q \neq NSW, NSW \neq V\}$
- Alle Constraints haben 2 Variablen

Graph eines Constraint Netzwerks

Ein Constraint-Netzwerk wird häufig durch seinen Graphen veranschaulicht.

- Die Knoten des Graphen sind die Variablen
- Zwei Knoten sind durch eine Kante verbunden, wenn es einen Constraint gibt, in dessen Scope beide Variablen vorkommen.

Beispiel: Karte von Australien



Instanziierungen

Definition

Eine Instanziierung einer Teilmenge $Y \subseteq X$ von Variablen ist eine Zuweisung, die jeder Variable aus Y ein Element ihrer Domäne zuweist. Man kann eine Instanziierung also als eine Relation mit Scope Y auffassen, die nur ein einziges Tupel enthält. Eine Instanziierung I erfüllt einen Constraint C_k, wenn $\pi_{\text{scope}(C)}(I) \in C_k$ ist, wenn also $\text{scope}(Y) \supseteq \text{scope}(X)$ ist und alle Werte von I zusammen in C_k vorkommen dürfen.

Eine partielle Instanziierung mit Scope Y ist *konsistent*, wenn sie alle Constraints erfüllt, deren Scopes Teilmengen von Y sind.

Eine Lösung eines Constraint-Netzwerks (X, DC) ist eine konsistente Instanziierung mit Scope X.

Lösung von CSPs: Backtracking

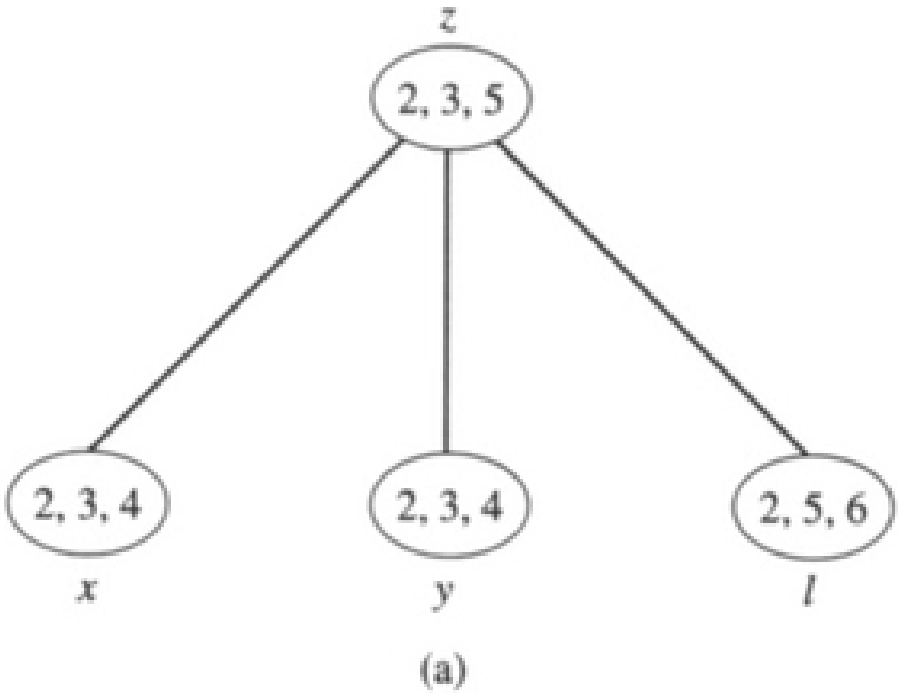
function BACKTRACKING-SUCHE(X, D, C)
  $i \leftarrow 1,\ D'_1 \leftarrow_C D_1$
  while $1 \leq i \leq n$ **do**
    $\langle a_i, D'_i \rangle \leftarrow$ WÄHLE-WERT($\vec{a}_{i-1}, x_i, X, D', C$)
    if $a_i = null$ **then** *[Kein Wert für x_i]*
      $i \leftarrow i - 1$ *[Backtracking]*
    else
      $i \leftarrow i + 1,\ D'_i \leftarrow_C D_i$
    end if
  end while
  if $i = 0$ **then**
    return inkonsistent
  else
    return $\vec{a}_n$
  end if
end function

Lösung von CSPs: Auswahl der Werte

function WÄHLE-WERT($\vec{a}_{i-1}, x_i, X, D', C$)
    while $D'_i \neq \emptyset$ **do**
        Wähle $a_i \in D'_i$
        $D'_i \leftarrow D'_i \setminus \{a_i\}$
        if KONSISTENT?($\vec{a}_i, C$) **then**
            return $\langle a_i, D'_i \rangle$
        end if
    end while
    return *null*
end function

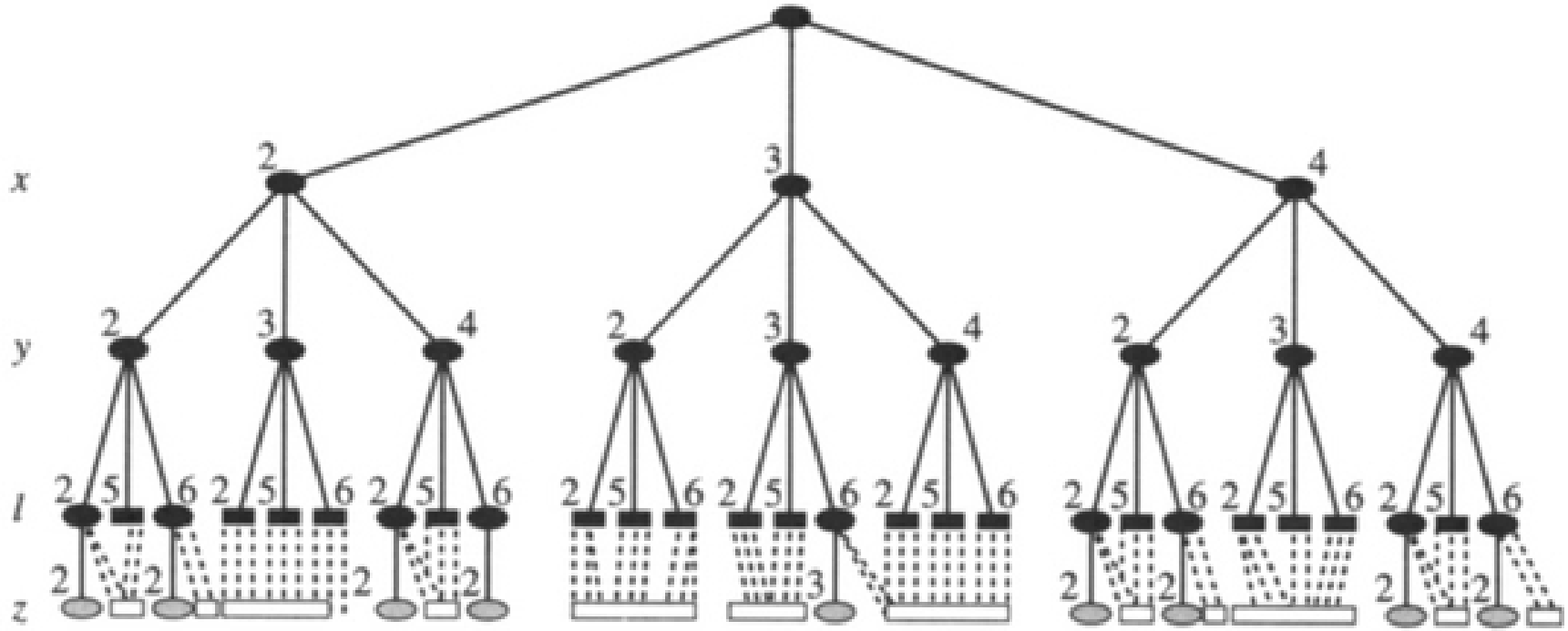
Backtracking: Einfluss der Variablenordnung

$X = \{x, y, l, z\}$
$D_x = \{2, 3, 4\}, D_y = \{2, 3, 4\}, D_l = \{2, 5, 6\}, D_z = \{2, 3, 5\}$
$C_{xz} = (x = 0 \mod z), C_{yz} = (y = 0 \mod z),$
$C_{lz} = (l = 0 \mod z)$



Backtracking: Einfluss der Variablenordnung

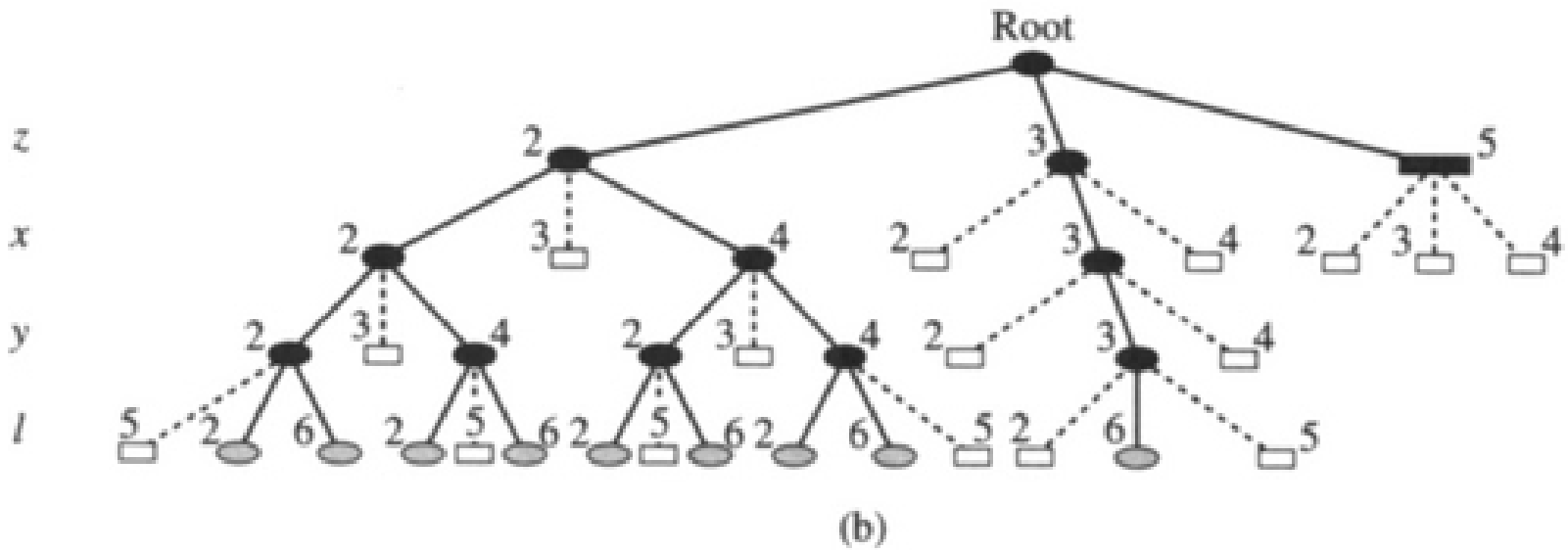
Ordnung der Variablen: x, y, l, z: 48 Suchzustände, 18 Sackgassen



(c)

Backtracking: Einfluss der Variablenordnung

Ordnung der Variablen: z, x, y, l: 20 Suchzustände, 1 Sackgasse



Heuristik: Ordnung der Variablen, Werte

- Fail first: Wähle die Variable, die den Suchraum am meisten einschränkt (z.B. dynamisch die Variable mit der kleinsten Domäne, *min-domain*, *minimum remaining values (MRV)*)
- Wähle die Variable, die in den meisten Constraints vorkommt (*degree heuristic*)
- Suche für diese Variable den Wert aus, der die zukünftige Auswahl am wenigsten beschränkt (*least constraining value*)
- Randomisiertes Backtracking: Wähle Variablen zufällig aus; wenn die Suche nach einer gewissen Zeit kein Ergebnis liefert breche ab und starte mit einer anderen (zufälligen) Variablenordnung neu

Verbesserung des Backtrackings

- Geschickte Auswahl der Variablenordnung
- Reduktion des Verzweigungsfaktors: Propagation als Vorbereitungsschritt
- Vermeiden von „thrashing,“ dem wiederholten Durchsuchen der gleichen partiellen Lösungen
  - Look-Ahead Schemata
  - Look-Back Schemata
- Look-Ahead: Stelle (durch begrenzte Propagation) fest, wie aktuelle Entscheidungen die zukünftige Suche beeinflussen
  - Welche Variable als nächstes instanziiert werden soll
  - Welcher Wert dieser Variablen zugewiesen werden soll
- Look-Back: Steuert das Backtracking-Verhalten:
  - Bis zu welcher Stelle soll beim Backtracking zurückgesprungen werden
  - Lerne Constraints, die vermeiden, dass der aktuelle Konflikt in zukünftigen Suchpfaden wieder auftritt

Propagation

Propagation oder „lokale Konsistenz“ bezeichnet das Entfernen von Werten, die mit den Constraints im Widerspruch stehen, aus den Domänen der Variablen.

Es gibt verschiedene Formen von lokaler Konsistenz:

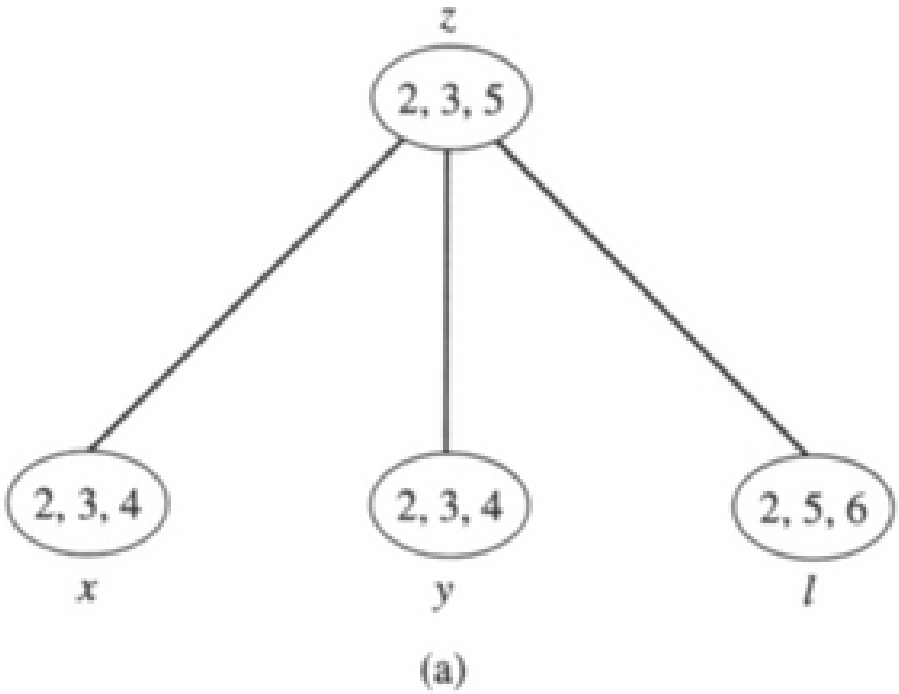
- Knotenkonsistenz, 1-Konsistenz (node consistency): nur Werte, die mit den unären Constraints vereinbar sind werden in den Domains behalten
- Bogenkonsistenz, 2-Konsistenz (arc consistency): Sei x_i eine Variable und x_j eine benachbarte Variable (d.h., es gibt einen Constraint mit Scope $\{x_i, x_j\}$). Zu jedem Wert von x_i gibt es einen Wert im Bereich von x_j, der konsistent ist.
- Pfadkonsistenz, 3-Konsistenz (path consistency): Zu jeder konsistenten Instanziierung von x_i, x_j und jedem zu x_i, x_j benachbarten Knotens x_k gibt es eine konsistente Instanziierung von x_i, x_j, x_k.

(Starke) k-Konsistenz

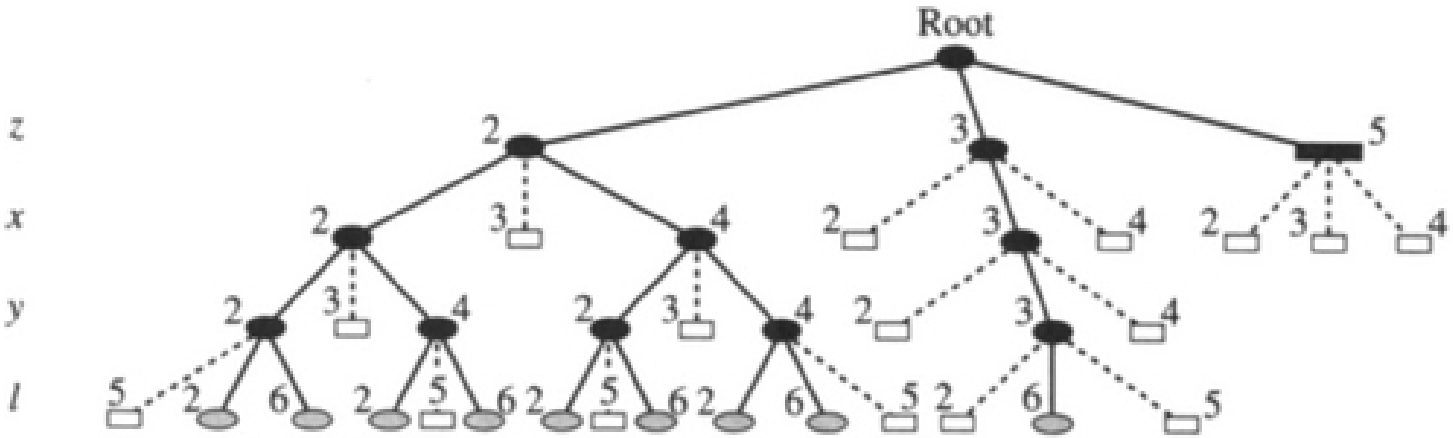
- Ein CSP ist k-konsistent, wenn sich jede konsistente Instanziierung von $k-1$ Variablen zu jeder anderen Variablen fortsetzen lässt.
- Für binäre Netzwerke entspricht 2-Konsistenz der Bogenkonsistenz, 3-Konsistenz der Pfadkonsistenz.
- Ein CSP ist stark k-konsistent, wenn es $1, 2, \ldots, k$-konsistent ist.

Backtracking: Einfluss von Propagation

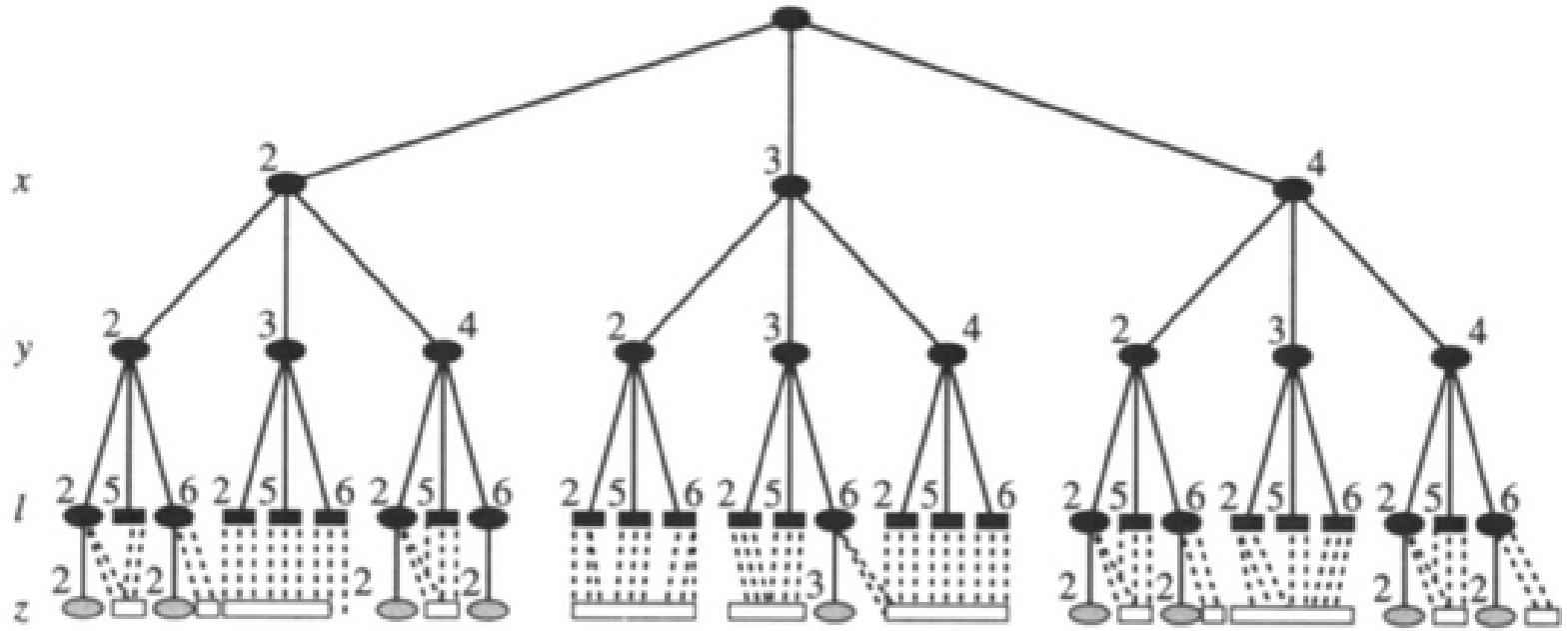
$X = \{x, y, l, z\}$
$D_x = \{2, 3, 4\}, D_y = \{2, 3, 4\}, D_l = \{2, 5, 6\}, D_z = \{2, 3, 5\}$
$C_{xz} = (x = 0 \mod z), C_{yz} = (y = 0 \mod z),$
$C_{lz} = (l = 0 \mod z)$



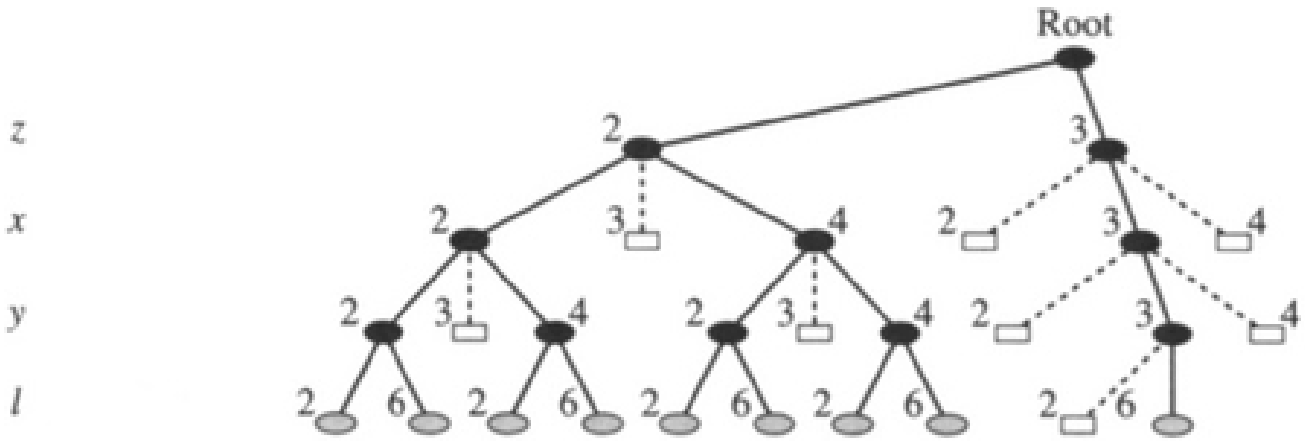
Backtracking: Einfluss von Propagation



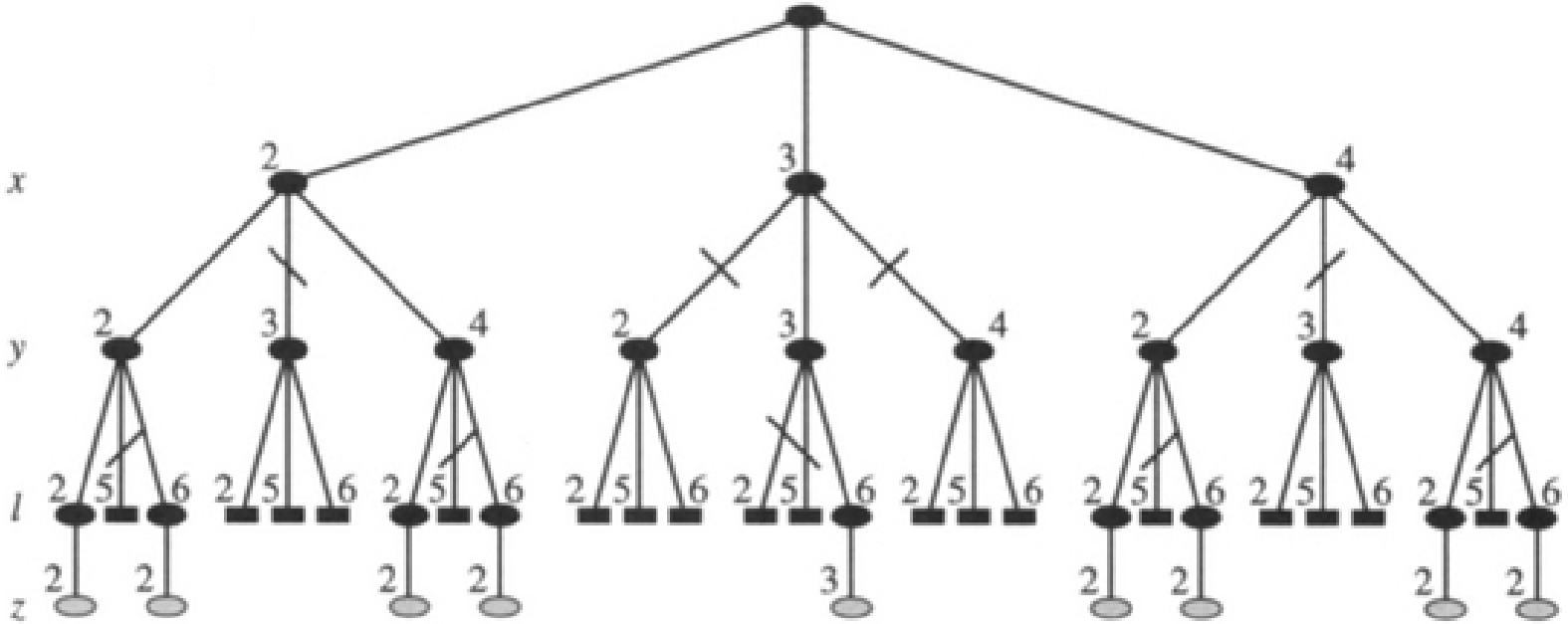
(b)



Backtracking: Einfluss von Propagation



(a)



Backtrack-Freie Netzwerke

Das obere Netzwerk auf der vorhergehenden Folie enthält keine Sackgassen mehr; eine Lösung kann daher immer ohne Backtracking gefunden werden.

Definition

Ein Netzwerk R heißt *backtrack-frei* entlang einer Variablenordnung d wenn jedes Blatt im Backtracking-Suchbaum eine Lösung von R ist.

Bogenkonsistenz

Definition

Sei x_i eine Variable und x_j eine benachbarte Variable (d.h., es gibt einen Constraint C_{ij} mit Scope $\{x_i, x_j\}$). x_i ist bogenkonsistent zu x_j, wenn es zu jedem Wert von x_i einen Wert im Bereich von x_j gibt, der C_{ij} erfüllt.
x_i und x_j sind bogenkonsistent, wenn x_i bogenkonsistent zu x_j und x_j bogenkonsistent zu x_i ist.
Ein binäres Constraint-Netzwerk ist bogenkonsistent, wenn jedes Paar von Variablen bogenkonsistent ist.

Ein binäres Constraint-Netzwerk, das nicht bogenkonsistent ist, kann durch Änderung der Domänen in ein äquivalentes bogenkonsistentes Netzwerk überführt werden.

Bogenkonsistenz: Revision von Domänen

function REVISE(x_i, x_j, D, C)
  $change \leftarrow$ falsch
  for all $a_i \in D_i$ **do**
    if Es gibt kein $a_j \in D_j$ mit $\langle a_i, a_j \rangle \in C_{ij}$ **then**
      $D_i \leftarrow D_i \setminus \{a_i\}$
      $change \leftarrow$ wahr
    end if
  end for
  return *change*
end function

Revise berechnet $D_i \leftarrow D_i \cap (C_{ij} \bowtie D_j)$ und gibt wahr zurück, wenn D_i verändert wurde.

Sei k eine Obergrenze für die Größe der D_i. Die Komplexität von REVISE ist $O(k^2)$.

Bogenkonsistenz: AC-1

function AC-1(X, D, C)
  repeat
    dg $\leftarrow$ falsch
    for $\langle x_i, x_j \rangle$ für die C_{ij} existiert **do**
      dg $\leftarrow$ dg $\vee$ Revise(x_i, x_j, D, C)
      dg $\leftarrow$ dg $\vee$ Revise(x_j, x_i, D, C)
    end for
  until dg $=$ falsch
end function

AC-3

function $\text{AC-3}(X, D, C)$
  for $\langle x_i, x_j \rangle$ für die C_{ij} existiert **do**
    $queue \leftarrow queue \cup \{(x_i, x_j), (x_j, x_i)\}$
  end for
  while $queue \neq \emptyset$ **do**
    Wähle (x_i, x_j) aus queue
    $queue \leftarrow queue \setminus \{(x_i, x_j)\}$
    if $\text{Revise}(x_i, x_j, C)$ **then**
      $queue \leftarrow queue \cup \{(x_k, x_i) \mid k \neq i, k \neq j\}$
    end if
  end while
end function

Gerichtete Bogenkonsistenz

Definition

Seien $R = (X, D, C)$ ein Constraint-Netzwerk und d eine Ordnung auf X. R heißt gerichtet bogenkonsistent (*directed arc consistent, DAC)*, wenn jedes x_i bogenkonsistent mit jedem $x_j, j > i$ ist.

Ein zyklenfreies, gerichtet bogenkonsistentes Constraint-Netzwerk ist backtrack-frei.

Look-Ahead

Propagation kann während der Backtracking-Suche eingesetzt werden

- Dadurch wird der Suchraum verkleinert, aber der Aufwand für jede Variableninstanziierung steigt
- Der Aufwand für Propagation sollte daher beschränkt werden; stärkere Konsistenz als Bogenkonsistenz wird während der Suche nur in wenigen Fällen verwendet
- Starke Propagation (Pfadkonsistenz, i-Konsistenz) vor der Suche macht Look-Ahead (manchmal) überflüssig, hat aber selber hohe Komplexität
- Look-Ahead kann dynamisch existierende Abhängigkeiten in Betracht ziehen

Die verschiedenen Look-Ahead-Strategien unterscheiden sich durch die verwendete Wähle-Wert Funktion

Verallgemeinertes Look-Ahead

function LOOK-AHEAD(X, D, C, WÄHLE-WERT)
  $D' \leftarrow_C D$, $i \leftarrow 1$
  while $1 \leq i \leq n$ **do**
    $D^i \leftarrow_C D'$ *[Werte von D' vor Instanziierung von x_i]*
    $a_i \leftarrow$ WÄHLE-WERT$(\vec{a}_{i-1}, x_i, X, D', C)$
    if $a_i = null$ **then**
      $i \leftarrow i - 1$, $D'_k \leftarrow D^i_k$ für alle $k > i$
    else
      $i \leftarrow i + 1$
    end if
  end while
  if $i = 0$ **then**
    return inkonsistent
  else
    return $\vec{a}_n$
  end if
end function

Forward Checking

- Forward Checking ist eine Variante des Look-Aheads, die nur relativ wenig Propagation durchführt
- Die Konsistenz jeder uninstanziierten Variable wird als gesondertes Problem betrachtet: Bei der Instanziierung von x_i werden alle Constraints, deren Scope Variablen aus $\vec{x_i}$ und ein x_k mit $k > i$ enthält, gesondert betrachtet
- Constraints zwischen x_j, x_k mit $j, k > i$ werden nicht propagiert
- Die Propagation wird nur einmal durchgeführt, nicht bis zu einem Fixpunkt

Forward Checking

function WÄHLE-WERT-FC($\vec{a}_{i-1}, x_i, X, D', C$)
  $D'' \leftarrow_C D'$
  while $D'_i \neq \emptyset$ **do**
    Wähle $a_i \in D'_i$, $D'_i \leftarrow D'_i \setminus \{a_i\}$
    for all $k, i < k \leq n$ **do**
      for all $a_k \in D'_k$ **do**
        if $\neg$KONSISTENT?$(\langle \vec{a}_i, x_k = a_k \rangle, C)$ **then**
          $D'_k \leftarrow D'_k \setminus \{a_k\}$
        end if
      end for
      if $D'_k = \emptyset$ **then**
        $D'_k \leftarrow D''_k$ für alle $k > i$
      else
        return a_i
      end if
    end for
  end while
  return *null*
end function

AC-Suche

function WÄHLE-WERT-AC($\vec{a}_{i-1}, x_i, X, D', C$)
  while $D'_i \neq \emptyset$ **do**
    Wähle $a_i \in D'_i$, $D'_i \leftarrow D'_i \setminus \{a_i\}$
    repeat
      wert-entfernt? $\leftarrow$ falsch
      for all j, k mit $i < j \leq n, i < k \leq n, k \neq j$ **do**
        for all $a_j \in D'_j$ **do**
          if $\neg\exists a_k \in D'_k$ mit KONSISTENT?$(\langle \vec{a}_i, a_j, a_k \rangle, C)$ **then**
            $D'_j \leftarrow D'_j \setminus \{a_j\}$
            wert-entfernt? $\leftarrow$ wahr
          end if
        end for
      end for
    until wert-entfernt? $=$ falsch
    if irgendein $D'_k = \emptyset$ **then** $D'_k \leftarrow D''_k$ für alle $k > i$
    else return a_i
    end if
  end while
  return *null*
end function

Full Look-Ahead

function WÄHLE-WERT-FLA($\vec{a}_{i-1}, x_i, X, D', C$)
  while $D'_i \neq \emptyset$ **do**
    Wähle $a_i \in D'_i$, $D'_i \leftarrow D'_i \setminus \{a_i\}$
    for all j, k mit $i < j \leq n, i < k \leq n, k \neq j$ **do**
      for all $a_j \in D'_j$ **do**
        if $\neg\exists a_k \in D'_k$ mit KONSISTENT?$(\langle \vec{a}_i, a_j, a_k \rangle, C)$ **then**
          $D'_j \leftarrow D'_j \setminus \{a_j\}$
        end if
      end for
    end for
    if irgendein $D'_k = \emptyset$ **then** $D'_k \leftarrow D''_k$ für alle $k > i$
    else return a_i
    end if
  end while
  return *null*
end function

Partial Look-Ahead

function WÄHLE-WERT-PLA($\vec{a}_{i-1}, x_i, X, D', C$)
  while $D'_i \neq \emptyset$ **do**
    Wähle $a_i \in D'_i$, $D'_i \leftarrow D'_i \setminus \{a_i\}$
    for all j, k mit $i < j \leq n, j < k \leq n$ **do**
      for all $a_j \in D'_j$ **do**
        if $\neg\exists a_k \in D'_k$ mit KONSISTENT?$(\langle \vec{a}_i, a_j, a_k \rangle, C)$ **then**
          $D'_j \leftarrow D'_j \setminus \{a_j\}$
        end if
      end for
    end for
    if irgendein $D'_k = \emptyset$ **then** $D'_k \leftarrow D''_k$ für alle $k > i$
    else return a_i
    end if
  end while
  return *null*
end function

Look Ahead: Variablenordnung

Die durch Look Ahead gewonnene Information kann zur dynamischen Auswahl von Variablenordnungen oder zur Verbesserung der Werteauswahl-Heuristiken verwendet werden: z.B. Dynamic Variable Forward Checking: Ordne die uninstanziierten Variablen nach jedem Look-Ahead-Schritt so, dass die Variable mit der kleinsten Domäne als nächstes instanziiert wird.

Backjumping

- Der Backtracking-Algorithmus geht in der Rückwärtsrichtung immer einen Schritt zurück
- Das ist oft nicht der Schritt in dem das Problem aufgetreten ist
- Idee: Finde den zuletzt zugewiesenen Wert einer Variable, der für die Inkonsistenz verantwortlich ist und springe direkt zu dieser Stelle zurück.

Backjumping

Eine Instanziierung $\vec{a}_i$ heißt *Sackgasse* (*dead end state*) (auf Ebene i), wenn sie mit jedem möglichen Wert von x_{i+1} inkonsistent ist. x_{i+1} heißt dann *dead-end Variable.*

Sei $\vec{a} = a_{i_1}, \ldots, a_{i_k}$ eine konsistente Instanziierung, x eine uninstanziierte Variable. Wenn es keinen Wert $b \in D_x$ gibt, der mit $\vec{a}$ konsistent ist, dann heißt $\vec{a}$ eine *Konfliktmenge* (*conflict set*) zu x; man sagt $\vec{a}$ steht im Konflikt mit x.

Steht kein Subtupel von $\vec{a}$ im Konflikt mit x so heißt $\vec{a}$ eine *minimale Konfliktmenge* zu x.

No-Good

Definition

Sei (X, D, C) ein Constraint-Netzwerk. Eine partielle Instanziierung $\vec{a}$, die in keiner Lösung von (X, D, C) vorkommt heißt *No-Good* oder *Nogood*.

Eine Konfliktmenge ist immer ein Nogood, aber es gibt auch Nogoods, die keine Konfliktmengen sind.

Backjumping

Backjumping versucht immer, wenn bei der Suche eine Sackgasse auftritt so weit wie möglich zurückzuspringen ohne dabei Lösungen zu verlieren.

Definition

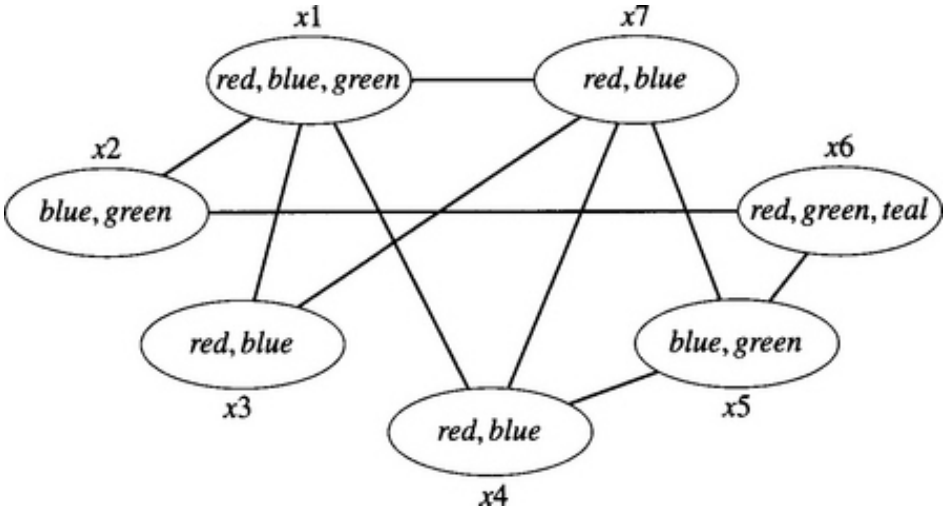
Sei $\vec{a}_i$ ein Blatt des Suchbaumes, das eine Sackgasse ist. a_j mit $j \leq i$ heißt *sicher* (*safe*), wenn $\vec{a}_j$ ein Nogood ist, d.h., wenn $\vec{a}_j$ nicht zu einer Lösung erweitert werden kann.

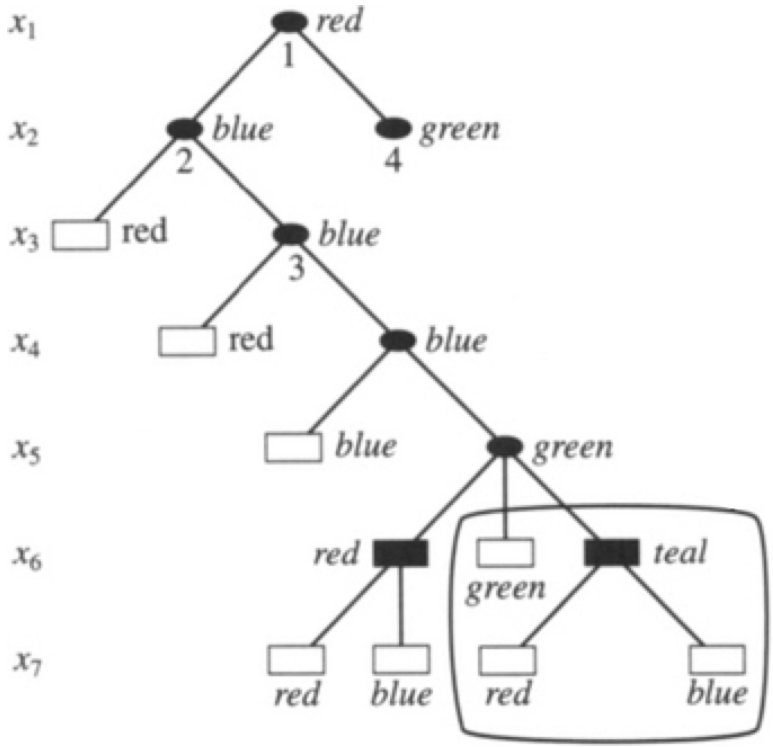
Was „so weit wie möglich“ bedeutet hängt von der Information ab, die der Algorithmus mitführt

Arten des Backjumpings

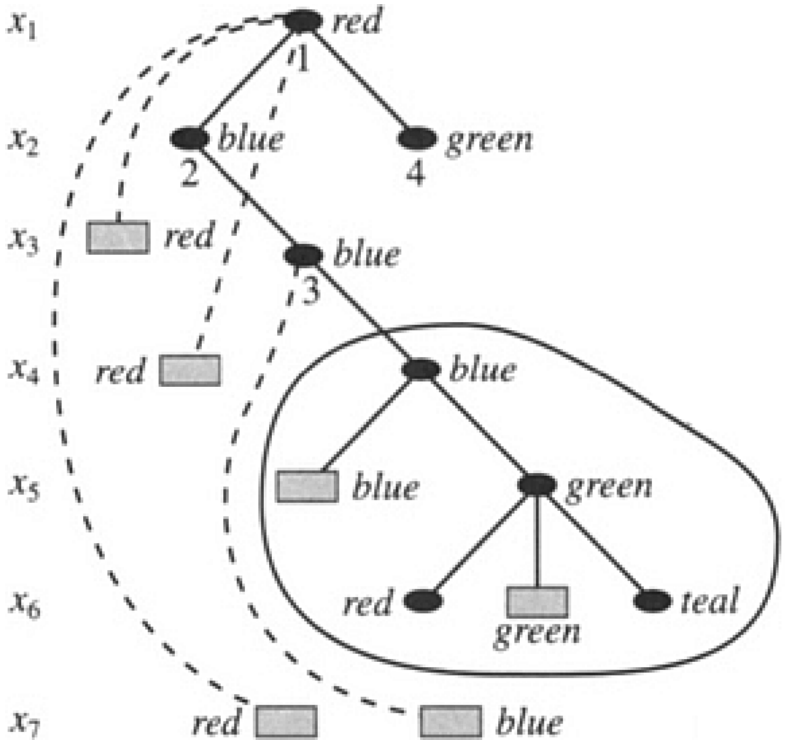
- Gaschnigs Backjumping: Verwendet Information aus der aktuellen Belegung um zurückzuspringen, springt aber nur an Blättern des Suchbaumes
- Graphenbasiertes Backjumping: Verwendet die Struktur des Constraintgraphen, um auch an inneren Sackgassen zurückzuspringen, beachtet aber die aktuelle Belegung der Variablen nicht
- Konfliktgesteuertes Backjumping: Verbindet Gaschnigs und graphenbasiertes Backjumping; springt an Blättern und inneren Knoten zurück und verwendet die aktuelle Belegung der Variablen um Konflikte zu identifizieren

Gaschnigs Backjumping

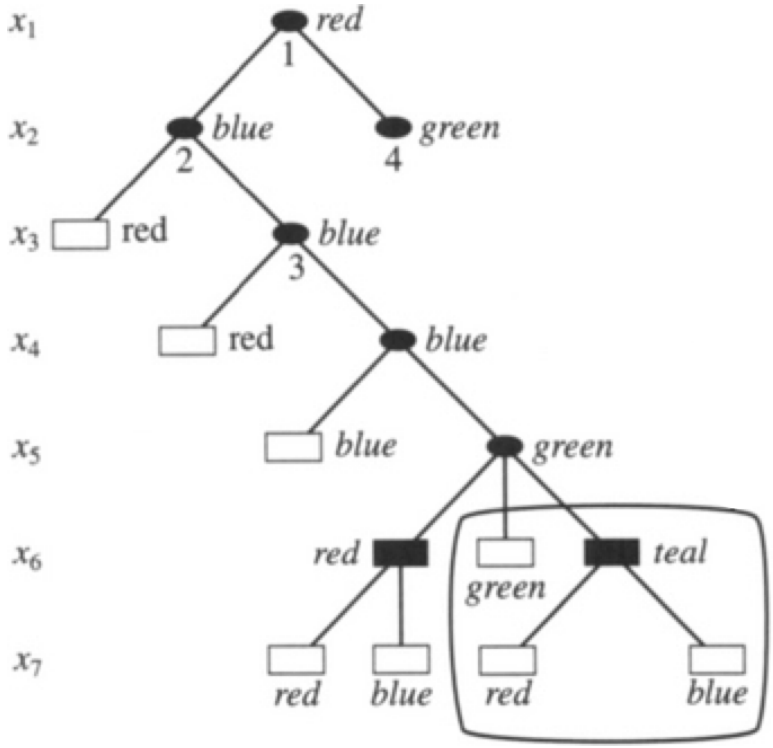




FC versus Gaschnigs Backjumping



Forward Checking



Gaschnigs Backjumping

MiniZinc

- Sprache zur Beschreibung von (reinen) Constraintproblemen
- Kann mit vielen Constraint-Solvern verwendet werden
- Damit sowohl Einbettung als Bibliothek als auch Integration in CLP-Systeme wie Eclipse möglich

Beispiel: Karte von Australien

$$X = \{WA, NT, Q, NSW, V, SA, T\}$$
$$D_i = \{1, 2, 3\}$$

```
% Anzahl der Farben
int: nc = 3;

% Constraintvariablen
var 1..nc: wa;   var 1..nc: nt;   var 1..nc: q;
var 1..nc: nsw;  var 1..nc: v;    var 1..nc: sa;
var 1..nc: t;
```

Beispiel: Karte von Australien

$$C = \{WA \neq NT, WA \neq SA, SA \neq V, SA \neq NSW, \\ SA \neq Q, SA \neq NT, NT \neq Q, Q \neq NSW, NSW \neq V\}$$

```
constraint wa  != nt;
constraint wa  != sa;
constraint sa  != v;
constraint sa  != nsw;
constraint sa  != q;
constraint sa  != nt;
constraint nt  != q;
constraint q   != nsw;
constraint nsw != v;
```

Beispiel: Karte von Australien

```
solve satisfy;

output ["wa=", show(wa), "\t nt=", show(nt),
        "\t sa=", show(sa), "\n",
        "q=", show(q),  "\t nsw=", show(nsw),
        "\t v=", show(v), "\n",
        "t=", show(t),  "\n"];
```

Syntax der Prädikatenlogik: Terme

Die Menge der prädikatenlogischen Terme $\mathcal{T}$ (über *Var*, *Fun* und *Const*) ist folgendermaßen rekursiv definiert.

- Jede Variable $x \in \mathit{Var}$ ist in $\mathcal{T}$ (d.h. $\mathit{Var} \subseteq \mathcal{T}$)
- Jede Konstante $c \in \mathit{Const}$ ist in $\mathcal{T}$ (d.h. $\mathit{Const} \subseteq \mathcal{T}$)
- Sei f ein n-stelliges Funktionssymbol und seien $t_1, \ldots, t_n$ Terme, dann ist $f(t_1, \ldots, t_n) \in \mathcal{T}$

Variablen und Konstanten nennt man auch *Primterme* oder *atomare Terme* oder *Atome*. Alle anderen Terme heißen *Funktionsterme*.
Terme in denen keine Variablen vorkommen heißen *Grundterme*.
Grundterme kann es nur geben, wenn *Const* nichtleer ist.

Terminduktion

Seien $t = f(t_1, \ldots, t_n)$ und $s = g(s_1, \ldots, s_m)$ syntaktisch gleiche Terme, $t \equiv s$. Dann ist $m = n$, $f \equiv g$ und für alle i mit $1 \leq i \leq n$ gilt $t_i \equiv s_i$.

Das folgende Beweisprinzip wird häufig verwendet, um syntaktische Eigenschaften zu beweisen.

Proposition (Terminduktion)

Sei $\mathcal{E}$ eine Eigenschaft von Termen, für die folgendes gilt:

- *$\mathcal{E}(t)$ gilt für alle Primterme t*
- *Sei f ein n-stelliges Funktionssymbol und gelte $\mathcal{E}(t_i)$ für $t_1, \ldots, t_n$. Dann gilt auch $\mathcal{E}(f(t_1, \ldots, t_n))$*

Dann gilt $\mathcal{E}$ für alle Terme.

Entsprechend kann man Eigenschaften von Termen durch *Rekursion über die Termstruktur* definieren.

Syntax der Prädikatenlogik: Formeln

Die *atomaren Formeln* (*Primformeln*) $\mathcal{A}$ (über *Var*, *Fun*, *Const* und *Pred*) sind folgendermaßen rekursiv definiert:

- Seien P ein n-stelliges Prädikatensymbol und $t_1, \ldots, t_n \in T$. Dann ist $P(t_1, \ldots, t_n) \in \mathcal{A}$
- Für $t_1, t_2 \in \mathcal{T}$ ist $t_1 = t_2 \in \mathcal{A}$

Die Menge der prädikatenlogischen Formeln $\mathcal{L}$ (über *Var*, *Fun*, *Const* und *Pred*) ist folgendermaßen rekursiv definiert:

- Ist $\phi \in \mathcal{A}$ so ist $\phi \in \mathcal{L}$ (d.h. $\mathcal{A} \subseteq \mathcal{L}$)
- Sind $\phi, \psi \in \mathcal{L}$, so sind auch $\neg\phi$, $(\phi \wedge \psi)$, $(\phi \vee \psi)$, $(\phi \Rightarrow \psi)$ und $(\phi \Leftrightarrow \psi)$ in $\mathcal{L}$ enthalten
- Sind $x \in \mathit{Var}$ und $\phi \in \mathcal{L}$, so sind auch $\forall x.\phi$ und $\exists x.\phi$ in $\mathcal{L}$ enthalten

Strukturen

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

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:

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

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:

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

Semantik von Formeln

In einer an die Semantik der Aussagenlogik angelehnten Schreibweise kann man die Erfüllbarkeitsrelation auch folgendermaßen aufschreiben:

$$
\begin{aligned}
[\![P(t_1, \ldots, t_n)]\!]_{\mathcal{M}} &= P^{\mathcal{A}}([\![t_1]\!]_{\mathcal{M}}, \ldots, [\![t_n]\!]_{\mathcal{M}}) \\
[\![t_1 = t_2]\!]_{\mathcal{M}} &= [\![t_1]\!]_{\mathcal{M}} = [\![t_2]\!]_{\mathcal{M}} \\
[\![\neg\phi]\!]_{\mathcal{M}} &= ![\![\phi]\!]_{\mathcal{M}} \\
[\![\phi \wedge \psi]\!]_{\mathcal{M}} &= [\![\phi]\!]_{\mathcal{M}} \,\&\, [\![\psi]\!]_{\mathcal{M}} \\
[\![\phi \vee \psi]\!]_{\mathcal{M}} &= [\![\phi]\!]_{\mathcal{M}} \mid [\![\psi]\!]_{\mathcal{M}} \\
[\![\phi \Rightarrow \psi]\!]_{\mathcal{M}} &= (![\![\phi]\!]_{\mathcal{M}}) \mid [\![\psi]\!]_{\mathcal{M}} \\
[\![\phi \Leftrightarrow \psi]\!]_{\mathcal{M}} &= [\![\phi]\!]_{\mathcal{M}} = [\![\psi]\!]_{\mathcal{M}} \\
[\![\forall x.\phi]\!]_{\mathcal{M}} &= \text{wahr gdw. } [\![\phi]\!]_{\mathcal{M}[x \mapsto a]} = \text{wahr für alle } a \\
[\![\exists x.\phi]\!]_{\mathcal{M}} &= \text{wahr gdw. es ein } a \text{ gibt mit } [\![\phi]\!]_{\mathcal{M}[x \mapsto a]} = \text{wahr}
\end{aligned}
$$

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!

Definition der Substitution

Seien alle vorkommenden Terme und Formeln mit $[x \mapsto t]$ kollisionsfrei.

$$
\begin{aligned}
x[x \mapsto t] &= t \\
y[x \mapsto t] &= y \quad \text{falls } y \neq x \\
c[x \mapsto t] &= c \\
f(t_1, \ldots, t_n)[x \mapsto t] &= f(x_1[x \mapsto t], \ldots, x_n[x \mapsto t]) \\
\\
R(t_1, \ldots, t_n)[x \mapsto t] &= R(x_1[x \mapsto t], \ldots, x_n[x \mapsto t]) \\
(t_1 = t_2)[x \mapsto t] &= t_1[x \mapsto t] = t_2[x \mapsto t] \\
(\neg\phi)[x \mapsto t] &= \neg(\phi[x \mapsto t]) \\
(\phi \operatorname{op} \psi)[x \mapsto t] &= (\phi[x \mapsto t]) \operatorname{op} (\psi[x \mapsto t]) \quad \operatorname{op} \in \{\wedge, \vee, \Rightarrow, \Leftrightarrow\} \\
(\mathrm{Q}\, x.\phi)[x \mapsto t] &= (\mathrm{Q}\, x.\phi) \qquad \mathrm{Q} \in \{\forall, \exists\} \\
(\mathrm{Q}\, y.\phi)[x \mapsto t] &= (\mathrm{Q}\, y.\phi[x \mapsto t]) \quad \mathrm{Q} \in \{\forall, \exists\}, y \neq x
\end{aligned}
$$

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.

Sequenzenkalkül (1)

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

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

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

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

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)

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

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

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

Koninzidenzsatz

Theorem (Koninzidenzsatz)

Seien ϕ eine Formel, $V \subseteq Var$ eine Menge von Variablen mit $\text{fv}(\phi) \subseteq V$, *$\mathcal{M}$ 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.

Endlichkeitssatz

Sei Φ eine (möglicherweise unendliche) Menge von Formeln.

Wir schreiben $\Phi \left|\frac{}{\text{seq}}\right. \phi$, 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 $\Phi \left|\frac{}{\text{seq}}\right. \phi$*, so gibt es eine endliche Teilmenge* $\Phi_0 \subseteq \Phi$ *für die gilt* $\Phi_0 \left|\frac{}{\text{seq}}\right. \phi$*.*

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

$$\Phi \left|\frac{}{\text{seq}}\right. \phi \quad \text{impliziert} \quad \Phi \models \phi$$

- Vollständigkeit eines Beweissystems

$$\Phi \models \phi \quad \text{impliziert} \quad \Phi \left|\frac{}{\text{seq}}\right. \phi$$

Gödelscher Vollständigkeitssatz

Theorem (Gödelscher Vollständigkeitssatz)

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

$$\Phi \left|\overline{\vdash_{\text{seq}}}\ \phi \quad \textit{genau dann, wenn} \quad \Phi \models \phi\right.$$

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.

Eigenschaften der Prädikatenlogik

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.

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

Prädikatenlogische Resolution

- Pränex-Normalform: Alle Quantoren werden an den Anfang der Formel gezogen. Aus $(\forall x.P(x)) \wedge (\exists y.Q(y))$ wird $\forall x.\exists y.P(x) \wedge Q(y)$ oder $\exists y.\forall x.P(x) \wedge Q(y)$
- Umwandlung des quantorenfreien Teils der Formel in CNF
- Skolemisierung: Existenzquantoren werden durch neue Konstanten oder Funktionssymbole ersetzt. Aus $\exists y.\forall x.P(x) \wedge Q(y)$ wird, $\forall x.P(x) \wedge Q(c^s)$, aus $\forall x.\exists y.P(x) \wedge Q(y)$ wird $\forall x.P(x) \wedge Q(f^s(x))$
- Allquantoren am Anfang der Formel werden weggelassen. Aus $\forall x.P(x) \wedge Q(c^s)$ wird $P(x) \wedge Q(c^s)$
- Unifikation stellt fest wann $P(t_1, \ldots, t_n)$ und $P(t'_1, \ldots, t'_n)$ komplementär sind und ermöglicht die Berechnung der Resolvente zweier Klauseln

Normalformen der Prädikatenlogik

Definition

Eine prädikatenlogische Formel ϕ ist genau dann in *Pränex-Normalform*, wenn sie die Form

$$Q_1\, x_1 \ldots . Q_n\, x_n . \mu$$

hat, wobei $Q_i \in \{\forall, \exists\}$ und μ eine quantorenfreie Formel ist.
$Q_1\, x_1 \ldots . Q_n\, x_n$ heißt der *Präfix*, μ die *Matrix* der Formel.

Zu jeder Formel gibt es (mindestens) eine Pränex-Normalform.

Wenn eine Formel ϕ in Pränex-Form ist, ihr Präfix nur aus Allquantoren besteht, und ihre Matrix in CNF ist, dann sagt man ϕ sei in *Skolem-Normalform* oder *Standardform*. Üblicherweise lässt man den Präfix dann weg.

Skolem-Normalform erhält Inkonsistenz

Proposition

Sei ϕ eine Formel und S die nach dem oben beschriebenen Verfahren erhaltene Skolem-Normalform zu ϕ. S ist genau dann inkonsistent, wenn ϕ inkonsistent ist.

Die Skolem-Normalform ϕ_S einer Formel ϕ ist *nicht* äquivalent zu ϕ. Sei, zum Beispiel

$$\phi = \exists x.P(x)$$

und $\mathcal{A}$ die folgende Struktur

Träger	$\{1, 2\}$
Konstanten	$c^{\mathcal{A}} = 1$
Prädikate	$P(1)^{\mathcal{A}} = \text{falsch}$, $P(2)^{\mathcal{A}} = \text{wahr}$

Dann gilt $\mathcal{A} \models \phi$ aber $\mathcal{A} \not\models \phi_S$

Unifikatoren

Definition (Komposition von Substitutionen)

Seien σ und θ Substitutionen. Die Komposition von θ und σ, $\theta \circ \sigma$ schreibt man auch $\sigma\theta$.

Bei der Komposition von Substitutionen wird also die links stehende Substitution zuerst ausgeführt. Das ist sinnvoll, weil die a Substitution von rechts angewendet wird: $(\phi\sigma)\theta = \phi(\sigma\theta) = \phi(\theta \circ \sigma)$.

Definition

Sei $E = \{E_1, \ldots E_k\}$ eine Menge von Ausdrücken (Termen oder Formeln). Eine Substitution σ heißt *Unifikator* von E wenn $E_1\sigma \equiv E_2\sigma \equiv \ldots \equiv E_k\sigma$ ist. E heißt *unifizierbar* wenn es einen Unifikator von E gibt. Ein Unifikator σ heißt *allgemeinster Unifikator* (*most general Unifyer*, *MGU*) von E, wenn sich jeder andere Unifikator θ in der Form $\theta = \sigma\eta$ schreiben lässt.

Binäre Resolution

Seien $L = \{L_1, \ldots, L_m\}$ und $M = \{M_1, \ldots, M_n\}$ so, dass L und M keine gemeinsamen Variablen haben. Falls L_i und M_j mit MGU σ unifizierbar sind und $L_i\sigma$ und $M_j\sigma$ komplementär sind (d.h. $L_i\sigma \equiv \neg M_j\sigma$ oder $\neg L_i\sigma \equiv M_j\sigma$) gilt

$$\frac{L_1, \ldots, L_m \qquad M_1, \ldots, M_n}{(L_1, \ldots, L_{i-1}, L_{i+1}, \ldots, L_m, M_1 \ldots, M_{j-1}, M_{j+1}, M_n)\sigma}$$

Diese Form der Ableitung heißt binäre Resolution.
$(L_1, \ldots, L_{i-1}, L_{i+1}, \ldots, L_m, M_1 \ldots, M_{j-1}, M_{j+1}, \ldots, M_n)\sigma$ heißt *binäre Resolvente* von $L_1, \ldots, L_m$ und $M_1, \ldots, M_n$.

Binäre Resolution

Binäre Resolution für Prädikatenlogik ist keine vollständige Ableitungsregel. Man muss binäre Resolution entweder durch Resolution von Mengen unifizierbarer komplementärer Literale erweitern oder sog. Faktorisierung einführen.

Definition

Sei $L = \{L_1, \ldots L_m\}$ eine Menge von Literalen. Sind zwei oder mehrere L_i mit MGU σ unifizierbar, so heißt $L\sigma$ ein *Faktor* von L.

Resolution

Seien L und M Mengen von Literalen. N ist eine Resolvente von L und M, wenn die Resolvente einer Anwendung binärer Resolution auf

1. L und M,
2. L und einen Faktor von M,
3. einen Faktor von L und M oder
4. einen Faktor von L und einen Faktor von M

ist. Die Ableitung einer solchen Resolvente nennt man *(prädikatenlogische) Resolution*. Wir schreiben dann $L, M \left|\frac{}{\text{res}}\right. M$.

Vollständigkeit der Resolution

Wir schreiben $S \vdash_{\text{res}} M$, wenn durch prädikatenlogische Resolution aus $S, \neg M$ die leere Klausel abgeleitet werden kann. Dann gilt:

Satz

Prädikatenlogische Resolution ist vollständig und korrekt bezüglich der genannten Ableitungsrelation, d.h., es gilt

$$S \vdash_{\text{res}} M \quad \textit{genau dann, wenn} \quad S \models M$$

Der Beweis erfolgt durch den Satz von Herbrand bzw. die im Satz von Herbrand verwendete Konstruktion.

(Wie im aussagenlogischen Fall kann durch den prädikatenlogischen Resolutionskalkül nicht direkt M aus S abgeleitet werden, wenn $S \models M$ gilt. Prädikatenlogische Resolution ist *widerspruchsvollständig* und wir führen damit immer Widerspruchsbeweise.)

Satz von Herbrand

Satz (Herbrand)

Eine Menge von Klauseln S ist genau dann unerfüllbar, wenn es eine endliche, unerfüllbare Menge S' gibt, die aus Grundinstanzen von S besteht.

Der Satz von Herbrand ist die Rechtfertigung für automatisierte Beweisverfahren wie den DPLL-Algorithmus und die Grundlage für den Korrektheitsbeweis des Resolutionskalküls.

Korrektheit und Vollständigkeit

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

- Korrektheit eines Beweissystems

$$\Phi \vdash \phi \quad \text{impliziert} \quad \Phi \models \phi$$

- Vollständigkeit eines Beweissystems

$$\Phi \models \phi \quad \text{impliziert} \quad \Phi \vdash \phi$$

Gödelscher Vollständigkeitssatz

Theorem (Gödelscher Vollständigkeitssatz)

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

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

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

Endlichkeits-, Kompaktheitssatz

Wir haben gezeigt, dass aus dem Gödelschen Vollständigkeitssatz und dem Endlichkeitssatz unmittelbar folgt:

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.

Ähnlich zeigt man die folgende Aussage

Satz (Kompaktheitssatz)

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

Ausdrucksmächtigkeit der Prädikatenlogik

Viele Sachverhalte können in der Prädikatenlogik ausgedrückt werden:

- Mathematische Theorien, z.B. Gruppentheorie, Mengenlehre, ...
- Chemische Reaktionen
- Abläufe (durch Formalisierung von Situationen, Fluenten, etc.)

Aber: Manche Sachverhalte lassen sich nicht beschreiben, z.B. die transitive Hülle: Wir haben das Prädikat E für Eltern definiert. Das Prädikat V soll die Vorfahren repräsentieren, $V(x, y)$ genau dann, wenn y ein Vorfahre von x ist.

- $P(x, y) \Rightarrow V(x, y)$
- $(\exists z.P(x, z) \wedge V(z, y)) \Rightarrow V(x, y)$

Ist V eine korrekte Formalisierung des Begriffs „Vorfahre“?

Transitiver Abschluss

Frage: Ist V eine korrekte Formalisierung des Begriffs „Vorfahre“?

Antwort: Nein. V erzwingt zwar die Transitivität, stellt aber nicht sicher, dass das Modell von V minimal ist. Z.B. kann man V in einer Struktur $\mathcal{A}$ durch A^2 interpretieren. Was wir suchen ist die minimale Relation, für die die geforderte Abschlusseigenschaft gilt; *nur* die Elemente, die durch die Axiome gefordert sind sollen in V enthalten sein.

Frage: Wie können wir die Minimalität von V in der Prädikatenlogik ausdrücken?

Antwort: Gar nicht. Die Minimalität von V ist eine Eigenschaft, die über alle möglichen Prädikate redet („V impliziert *jedes andere Prädikat*, das . . . “). Das ist in der Prädikatenlogik nicht ausdrückbar.

Prädikatenlogik zweiter Stufe

Erweitert man die Prädikatenlogik um Variablen für Funktionen und Prädikate und erlaubt Quantifikation über diese Variablen, so erhält man die *Prädikatenlogik zweiter Stufe*. Damit kann man z.B. schreiben

$$\forall P.\forall x.P(x) = P(x)$$

Mit dieser Erweiterung kann man die Transitive Hülle V von P folgendermaßen charakterisieren:

$$\forall Q.\Big[P(x,y) \Rightarrow Q(x,y) \wedge \big((\exists z.P(x,z) \wedge Q(z,y)) \Rightarrow Q(x,y)\big)\Big] \Rightarrow V(x,y) \Rightarrow Q(x,y)$$

Prädikatenlogik zweiter Stufe

Als Argumente von Funktionen und Prädikaten sind in der Prädikatenlogik zweiter Stufe nur Individuen zulässig, keine Funktions- oder Prädikatenvariablen oder -konstanten. Z.B. ist $P(f)$ kein zulässiger Term, wenn f eine Funktionsvariable ist.

Modelle der Prädikatenlogik zweiter Stufe

Definition

Ein *Modell* $\mathcal{M}$ einer prädikatenlogischen Sprache zweiter Stufe $\mathcal{L}$ ist ein Paar $(\mathcal{A}, w)$, bestehend aus einer $\mathcal{L}$-Struktur $\mathcal{A}$ (mit Träger A) und *Belegungen* $w_I : Var_I \to A$, $w_F : Var_F \to \mathcal{F}(A)$, $w_P : Var_P \to \mathcal{R}(A)$. 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.

Dabei steht $\mathcal{F}(A)$ für die Menge aller Funktionen über A und $\mathcal{R}(A)$ für die Menge aller Relationen über A; die Belegungen w_F und w_P müssen die Stelligkeiten der Funktions- bzw. Prädikatenvariablen berücksichtigen.

Substitution in Modellen wird sinngemäß wie bei der Prädikatenlogik erster Stufe definiert.

Semantik von Termen

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

$$
\begin{aligned}
[\![x]\!]_{\mathcal{M}} &= w_I(x) \\
[\![c]\!]_{\mathcal{M}} &= c^{\mathcal{A}} \\
[\![f(t_1, \ldots, t_n)]\!]_{\mathcal{M}} &= w_F(f)([\![t_1]\!]_{\mathcal{M}}, \ldots, [\![t_n]\!]_{\mathcal{M}}) && f \text{ Funktionsvariable} \\
[\![f(t_1, \ldots, t_n)]\!]_{\mathcal{M}} &= f^{\mathcal{A}}([\![t_1]\!]_{\mathcal{M}}, \ldots, [\![t_n]\!]_{\mathcal{M}}) && f \text{ Funktionssymbol}
\end{aligned}
$$

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:

$$\mathcal{M} \models P(t_1, \ldots, t_n) \iff w_P(P)([\![t_1]\!]_{\mathcal{M}}, \ldots, [\![t_n]\!]_{\mathcal{M}}) \quad P \text{ Prädikatenvariable}$$

$$\mathcal{M} \models P(t_1, \ldots, t_n) \iff P^{\mathcal{A}}([\![t_1]\!]_{\mathcal{M}}, \ldots, [\![t_n]\!]_{\mathcal{M}}) \quad P \text{ Prädikatensymbol}$$

$$\mathcal{M} \models t_1 = t_2 \iff [\![t_1]\!]_{\mathcal{M}} = [\![t_2]\!]_{\mathcal{M}}$$

$$\mathcal{M} \models \neg\phi \iff \mathcal{M} \models \phi \text{ ist falsch } (\mathcal{M} \not\models \phi)$$

$$\mathcal{M} \models \phi \wedge \psi \iff \mathcal{M} \models \phi \text{ und } \mathcal{M} \models \psi$$

$$\mathcal{M} \models \phi \vee \psi \iff \mathcal{M} \models \phi \text{ oder } \mathcal{M} \models \psi$$

$$\mathcal{M} \models \phi \Rightarrow \psi \iff \mathcal{M} \not\models \phi \text{ oder } \mathcal{M} \models \psi$$

$$\mathcal{M} \models \phi \Leftrightarrow \psi \iff (\mathcal{M} \models \phi \text{ und } \mathcal{M} \models \psi) \text{ oder } (\mathcal{M} \not\models \phi \text{ und } \mathcal{M} \not\models \psi)$$

Erfüllungsrelation

Die Semantik von quantifizierten Formeln wird folgendermaßen definiert:

$$\mathcal{M} \models \forall x.\phi \iff \mathcal{M}[x \mapsto a] \models \phi \text{ für alle Elemente } a \in A$$
$$\mathcal{M} \models \exists x.\phi \iff \text{es gibt ein Element } a \in A \text{ mit } \mathcal{M}[x \mapsto a] \models \phi$$
$$\mathcal{M} \models \forall f.\phi \iff \mathcal{M}[f \mapsto g] \models \phi \text{ für alle Funktionen } g \in \mathcal{F}(A)$$
$$\mathcal{M} \models \exists f.\phi \iff \text{es gibt eine Funktion } g \in \mathcal{F}(A) \text{ mit } \mathcal{M}[f \mapsto g] \models \phi$$
$$\mathcal{M} \models \forall P.\phi \iff \mathcal{M}[P \mapsto R] \models \phi \text{ für alle Relationen } R \in \mathcal{R}(A)$$
$$\mathcal{M} \models \exists x.\phi \iff \text{es gibt eine Relation } R \in \mathcal{R}(A) \text{ mit } \mathcal{M}[P \mapsto R] \models \phi$$

Auch hier müssen die Funktionen und Relationen die Stelligkeit beachten.

Ausdrucksmächtigkeit der Prädikatenlogik zweiter Stufe

- Endlichkeit des Modells:

$$\forall f.(f(x) = f(y) \Rightarrow x = y) \Rightarrow \forall y.\exists x.f(x) = y$$

  (Jede injektive Funktion ist surjektiv.)
- Unendlichkeit des Modells:

$$\exists R.(\forall x.\forall y.\forall z.R(x,y) \wedge R(y,y) \Rightarrow R(x,z)) \wedge \forall x.\neg R(x,x) \wedge \exists y.R(x,y)$$

  (Es gibt eine vollständige, transitive, irreflexive Relation.)
- Wohlordnung (wenn eine Prädikat erfüllbar ist, so gibt es ein kleinstes Element, welches das Prädikat erfüllt):

$$\forall P.(\exists x.P(x)) \Rightarrow \exists x.P(x) \wedge \forall y.P(y) \Rightarrow x \leq y$$

- Transitive Hülle (s.o.)
- „There are some critics who admire only each other“

Eigenschaften der Prädikatenlogik zweiter Stufe

Sei $\phi_{\geq n}$ die Formel

$$\exists x_1 \ldots \exists x_n . x_1 \neq x_2 \wedge x_1 \neq x_3 \wedge \cdots \wedge x_{n-1} \neq x_n$$

die besagt, dass es mindestens n verschiedene Individuen gibt, sei

$$\phi_{endl} = \forall f . (f(x) = f(y) \Rightarrow x = y) \Rightarrow \forall y . \exists x . f(x) = y$$

die Formel, die besagt, dass alle Modelle endlich sein müssen, und sei

$$\Phi = \{\phi_{endl}\} \cup \{\phi_{\geq n} \mid n \in \mathbb{N}\}$$

Dann ist jede endliche Teilmenge von Φ erfüllbar aber Φ nicht (denn kein endliches Modell kann $\phi_{\geq n}$ für alle n erfüllen, kein unendliches Modell ϕ_{endl}.) Damit ist klar, dass der Kompaktheitssatz nicht gelten kann.

Eigenschaften der Prädikatenlogik zweiter Stufe

- Der Kompaktheitssatz gilt nicht
- Da der Kompaktheitssatz unmittelbar aus dem Vollständigkeitssatz und den Eigenschaften von Herleitungen folgt, kann der Vollständigkeitssatz auch nicht gelten: Kein korrekter Beweiskalkül für die Prädikatenlogik zweiter Stufe ist vollständig
- Die Menge der allgemeingültigen Formeln zweiter Stufe ist nicht rekursiv aufzählbar
- Der Satz von Löwenheim-Skolem gilt nicht für die Prädikatenlogik zweiter Stufe

Prädikatenlogiken höherer Stufe

Frage: Was passiert, wenn man Funktionen und Prädikate als Argumente von Funktionen und Prädikaten zulässt? Bekommt man immer ausdrucksstärkere Logiken?

Antwort: Nein. Man kann zeigen, dass Logiken höherer Stufe keine weitere Steigerung der Ausdrucksmächtigkeit bringen. Sie lassen sich alle auf die Prädikatenlogik zweiter Stufe zurückführen.

Für die praktische Anwendung ist es aber vorteilhaft Terme und Formeln höherer Ordnung zuzulassen und zu systematisieren. Damit kommt man zur sog. (klassischen) *Typtheorie*.

Terme/Formeln

Die Typtheorie hat die folgenden Symbole:

- Für jeden Typ α gibt es eine abzählbare Menge von Variablen x_α
- Die logischen Konstanten $\neg_{o\to o}$, $\vee_{o\to o\to o}$, $\Pi_{(\alpha\to o)\to o}$ und $\iota_{(\alpha\to o)\to\alpha}$, für jeden Typ α
- Nichtlogische (domänenspezifische) Konstanten

Terme/Formeln sind folgendermaßen definiert:

- Jede Variable und Konstante vom Typ α ist ein Term vom Typ α
- Sind $A_{\alpha\to\beta}$ und B_α Terme, so ist AB ein Term vom Typ β
- Ist x_α eine Variable und A_β ein Term so ist $\lambda x.A$ ein Term vom Typ $\alpha \to \beta$

PVS

Die Logik von PVS basiert auf der gerade beschriebenen Typtheorie, beinhaltet aber einige Erweiterungen, die die Mächtigkeit der Sprache wesentlich erhöhen:

- Mehrere Grunddatentypen
- Tupel, Strukturen (Records)
- Prädikaten-Subtypen
- Strukturelle Subtypen
- Abhängige Typen (dependent Types)
- Syntaktische Erweiterungen für algebraische Datentypen

Andere Features erlauben die einfachere Strukturierung von Spezifikationen, z.B. hierarchische, parametrisierbare Theorien.

Typen in PVS

- Basistypen: `number`, `nat`, `boolean`, ...
- Aufzählungstypen: `{red, green, blue}`
- Funktionstypen: `[number -> number]`
- Record-Typen: `[# flag: boolean, value: number #]`
- Tupel-Typen: `[boolean, number]`
- Cotupel-Typen (disjunkte Summen): `[boolean + number]`

Typen in PVS

- Algebraische Datentypen (und Codatentypen)

```
list[T: TYPE]: DATATYPE BEGIN
      null: null?
      cons(car: T, cdr: list): cons?
END DATATYPE
```

- Prädikaten-Subtypen:
  - `{x: real | x /= 0}`
  - `{f: [real -> real] | injective?(f)}`
  - `{x: T | P(x)}` kann als `(P)` geschrieben werden
- Strukturelle Subtypen:
  - `[# x,y: real, c: color #]` ist Subtyp von `[# x,y: real #]`
  - Updates respektieren Subtypen
- Abhängige Typen (dependent Types) für Funktionen, Tupel, Records und algebraische Datentypen:

```
[n: nat -> {m: nat | m <= n}]
```

PVS Ausdrücke

- Logik: `TRUE`, `FALSE`, `AND`, `OR`, `XOR`, `NOT`, `IMPLIES`, `FORALL`, `EXISTS`, `=`, ...
- Arithmetik: `+`, `-`, `*`, `/`, `<`, `<=`, `>`, `>=`, `0`, `1`, `2`, ...
- Funktionen:
  - Applikation (`f(x)`)
  - Abstraktion (`LAMBDA (x): A = ...`)
  - Update (`f WITH [(X) := 1]`)
- Hilbert-Operator: `epsilon`
- Typumwandlungen

PVS Ausdrücke

- Records: Konstruktion (`(# size := 0 #)`), Selektion (`size(r)`), Update (`r WITH [size := 1]`)
- Tupel: Konstruktion (`(0, 1)`), Selektion (`proj_1(t)` oder `t`1`), Update (`t WITH [1 := 1]`)
- Konditionale: `IF-THEN-ELSE`, `COND`
- Destrukturierung von Records und Tupeln: (`LET ... = ... IN ...`)
- Pattern-Matching von (Co-)Datentypen `CASES`
- Tabellen

Beispiel: Listen ganzer Zahlen

```
intlist: DATATYPE
 BEGIN
  null: null?
  cons(car: int, cdr: intlist): cons?
 END intlist
```

Temporallogik

Eine Temporallogik ist eine Logik, in der man Operatoren hat, die derartige „zeitlichen“ Zusammenhänge beschreiben. Dabei unterscheidet man Logiken, mit

- linearer Zeit: diese Temporallogiken beschränken sich auf Fragen, die einen zukünftigen Ablauf, bzw. alle in einem Knoten beginnenden Abläufe betreffen, und solche mit
- verzweigter Zeit: in diesen Logiken kann man über zukünftige Abläufe existenziell und universell quantifizieren.

Wir werden im Folgenden die Logik CTL* (Computation Tree Logic*) betrachten, der ein verzweigtes Modell der Zeit zugrundeliegt, die aber sowohl Aussagen über lineare Zeit als auch verzweigte Zeit machen kann.

Syntax von CTL*

In CTL* werden die Aussagenlogischen Formeln erweitert um

- Die Pfadquantoren **A** (für alle Pfade) und **E** (es gibt einen Pfad)
- Die Temporaloperatoren
  - $\mathbf{X}\,\phi$ (neXt time): die Eigenschaft ϕ gilt im nächsten Zustand des Pfades
  - $\mathbf{F}\,\phi$ (Future): die Eigenschaft ϕ wird auf einem Zustand des Pfades gelten
  - $\mathbf{G}\,\phi$ (Globally): die Eigenschaft ϕ gilt für jeden Zustand des Pfades
  - $\phi\,\mathbf{U}\,\psi$ (Until): gilt, wenn es einen Zustand auf dem Pfad gibt, für den ψ gilt, und wenn in allen vorhergehenden Zuständen ϕ gilt
  - $\phi\,\mathbf{R}\,\psi$ (Release): gilt, wenn ψ bis zum ersten Zustand gilt, in dem ϕ gilt (einschließlich dieses Zustandes). ϕ muss aber nicht gelten.

Syntax von CTL*

In CTL* gibt es *Zustandsformeln (state formulas)* und *Pfadformeln (path formulas)*. Sei AP die Menge der atomaren Aussagen.

- *Zustandsformeln* sind folgendermaßen rekursiv definiert:
  - Ist $A \in AP$, dann ist A eine Zustandsformel
  - Sind ϕ und ψ Zustandsformeln, so sind auch $\neg\phi$, $\phi \wedge \psi$ und $\phi \vee \psi$ Zustandsformeln
  - Ist ϕ eine Pfadformel, dann sind $\mathbf{A}\,\phi$ und $\mathbf{E}\,\phi$ Zustandsformeln
- *Pfadformeln* sind folgendermaßen rekursiv definiert:
  - Ist ϕ eine Zustandsformel, dann ist ϕ auch eine Pfadformel
  - Sind ϕ und ψ Pfadformeln, dann sind auch $\neg\phi$, $\phi \wedge \psi$, $\phi \vee \psi$, $\mathbf{X}\,\phi$, $\mathbf{F}\,\phi$, $\mathbf{G}\,\phi$, $\phi\,\mathbf{U}\,\psi$ und $\phi\,\mathbf{R}\,\psi$ Pfadformeln

  CTL* ist die Menge der durch diese Regeln generierten Zustandsformeln.

Transitionssysteme und Kripke-Strukturen

Sei $\mathcal{L}$ eine logische Sprache mit Atomen AP.

Definition

Ein Transitionssystem $\mathcal{M} = (S, \to, L)$ besteht aus einer Menge von Zuständen S, einer binären Relation $\to$ auf S, so dass es für jedes $s \in S$ ein Element $s' \in S$ gibt, für das $s \to s'$ gilt, und einer Funktion $L : S \to \mathfrak{P}(A)$, die jedem Zustand $s \in S$ die Menge der Atome zuordnet, die in s gelten.

Ein Transitionssystem $\mathcal{M}$ zusammen mit einer Menge $S_0 \subseteq S$ nennt man eine *Kripke-Struktur*.

Ein *Pfad* in $\mathcal{M}$ ist eine unendliche Folge von Zuständen, $\pi = s_0, s_1, \ldots$, so dass für alle i gilt $s_i \to s_{i+1}$. (Ein Pfad ist somit ein unendlicher Ast in dem vom Transitionssystem erzeugten Baum.)

Semantik von CTL*

$$
\begin{aligned}
\mathcal{M}, s \models A &\iff p \in L(s) \\
\mathcal{M}, s \models \neg\phi &\iff \mathcal{M}, s \not\models \phi \\
\mathcal{M}, s \models \phi \vee \psi &\iff \mathcal{M}, s \models \phi \text{ oder } \mathcal{M}, s \models \psi \\
\mathcal{M}, s \models \phi \wedge \psi &\iff \mathcal{M}, s \models \phi \text{ und } \mathcal{M}, s \models \psi \\
\mathcal{M}, s \models \mathbf{E}\,\phi &\iff \text{Es gibt einen Pfad } \pi^s \text{ mit } \mathcal{M}, \pi \models \phi \\
\mathcal{M}, s \models \mathbf{A}\,\phi &\iff \text{Für alle Pfade } \pi^s \text{ gilt } \mathcal{M}, \pi \models \phi
\end{aligned}
$$

Semantik von CTL*

$$\mathcal{M}, \pi \models \phi \iff ZF(\phi),\ \pi \text{ beginnt mit } s \text{ und } \mathcal{M}, s \models \phi$$

$$\mathcal{M}, \pi \models \neg\phi \iff \mathcal{M}, \pi \not\models \phi$$

$$\mathcal{M}, \pi \models \phi \vee \psi \iff \mathcal{M}, \pi \models \phi \text{ oder } \mathcal{M}, \pi \models \psi$$

$$\mathcal{M}, \pi \models \phi \wedge \psi \iff \mathcal{M}, \pi \models \phi \text{ und } \mathcal{M}, \pi \models \psi$$

$$\mathcal{M}, \pi \models \mathbf{X}\,\phi \iff \mathcal{M}, \pi^1 \models \phi$$

$$\mathcal{M}, \pi \models \mathbf{F}\,\phi \iff \text{es gibt } k \text{ mit } \mathcal{M}, \pi^k \models \phi$$

$$\mathcal{M}, \pi \models \mathbf{G}\,\phi \iff \text{für alle } i \geq 0 \text{ gilt } \mathcal{M}, \pi^i \models \phi$$

$$\mathcal{M}, \pi \models \phi\,\mathbf{U}\,\psi \iff \exists k \geq 0. \mathcal{M}\pi^k \models \psi \wedge \forall 0 \leq i < k. \mathcal{M}, \pi^i \models \phi$$

$$\mathcal{M}, \pi \models \phi\,\mathbf{R}\,\psi \iff \forall j \geq 0.\ \text{If } \forall i < j. \mathcal{M}, \pi^i \not\models \phi \text{ then } \mathcal{M}, \pi^j \models \psi$$

Einschränkungen: LTL und CTL

- CTL (Computation-Tree Logic): Jeder der Operatoren **X**, **F**, **G**, **U** und **R** muss unmittelbar nach einem Pfadquantor stehen. CTL ist eine Sprache, die nur verzweigte Zeit beinhaltet
- LTL (Linear-Time Logic): Nur Formeln der Form $\mathbf{A}\,\phi$ mit einer Pfadformel ϕ, in der alle Zustands-Teilformeln atomar sind. (Typischerweise wird der Allquantor weggelassen.) LTL ist eine Sprache, die nur lineare Zeit beinhaltet

Model Checking

Sei $\mathcal{M}$ eine Kripke-Struktur, die ein (nebenläufiges System) mit endlich vielen Zuständen beschreibt, und sei ϕ eine temporallogische Formel, die eine gewünschte Eigenschaft des Systems beschreibt.

- Finde alle Zustände für die gilt $\mathcal{M}, s \models \phi$, also $MS = \{s \in S \mid M, s \models \phi\}$
- Damit die Eigenschaft für das System erfüllt ist, muss gelten $S_0 \subseteq MS$
- Wenn die Eigenschaft nicht erfüllt ist kann meist ein Pfad angegeben werden, der die Eigenschaft nicht erfüllt.

Vorteile/Nachteile des Model Checkings

- Model-Checking ist weitgehend automatisch
- Model-Checking kann zur Verifikation einzelner Eigenschaften hergenommen werden; ein vollständiges Systemmodell ist nicht notwendig
- Model-Checking eignet sich für nebenläufige und reaktive Systeme
- Gilt eine Eigenschaft nicht, so können Model-Checker typischerweise Fehler-Traces ausgeben
- Problem: Explosion des Zustandsraums *(State Explosion)*
- Model-Checking funktioniert nicht für datenabhängige Systeme

Ausblick

. . .

Vielen Dank. . .

für Ihre Aufmerksamkeit