**Skriptum zur Vorlesung:**

# Computational Logic und Wissensrepräsentation

# WS ’97/’98

**Univ.-Doz. Dr. Jürgen Dix**
TU Wien
Institut für Informationssysteme

**Oktober 1997**

# Inhaltsverzeichnis

# Abbildungsverzeichnis

# Tabellenverzeichnis

# Vorwort

Dieses Skriptum faßt den Stoff der im Wintersemester '97/'98 an der TU Wien gehalten Vorlesung *Künstliche Intelligenz und Wissensrepräsentation* zusammen. Die ersten 7 Kapitel beruhen auf jeweils im Wintersemester '95/'96 und '96/'97 von mir an der Universität Koblenz gehaltenen sechs-stündigen Vorlesung (davon 2 Stunden Übung) *Künstliche Intelligenz.*

Außer Kapitel 5 stammen alle anderen der ersten 7 Kapitel (mit einigen Modifikationen) aus dem Buch [RN95]. Die Autoren dieses Buches haben freundlicherweise die Abbildungen als postscript Dateien frei zugänglich gemacht.

Die Kapitel 8 bis 13 stammen aus [BD98c]. Sie wurden als Tutorial auf der *European Conference on Artificial Intelligence* (ECAI '96) sowie als einwöchige Vorlesung auf der *European Summer School on Logic, Language and Information* im Oktober 1997 in Aix-en-Provence von G. Brewka und J. Dix vorgetragen.

# 1 Einleitung

## 1.1 Was ist KI?

**1.** Kognitionswissenschaft,

**2.** Sokrates ist ein Mensch. Menschen sind sterblich. Also ist Sokrates sterblich. *Syllogismen* (Aristoteles).

*Informale Beschreibung* $\rightsquigarrow$ * *Formale Beschreibung*
*Formale Beschreibung* $\rightsquigarrow$ ** *Lösung des Problems*.

* ist problematisch, da oft nicht genügend spezifiziert,

** ist Deduktion (korrekte Inferenzen): nur aufzählbar, nicht rekursiv.

**3.** Turing Test.

- Üblicher Turing Test
- Totaler Turing Test

| | |
|---|---|
| "The exciting new effort to make computers think … *machines with minds*, in the full and literal sense" (Haugeland, 1985)<br>"[The automation of] activities that we associate with human thinking, activities such as decision-making, problem solving, learning …" (Bellman, 1978) | "The study of mental faculties through the use of computational models" (Charniak and McDermott, 1985)<br>"The study of the computations that make it possible to perceive, reason, and act" (Winston, 1992) |
| "The art of creating machines that perform functions that require intelligence when performed by people" (Kurzweil, 1990)<br>"The study of how to make computers do things at which, at the moment, people are better" (Rich and Knight, 1991) | "A field of study that seeks to explain and emulate intelligent behavior in terms of computational processes" (Schalkoff, 1990)<br>"The branch of computer science that is concerned with the automation of intelligent behavior" (Luger and Stubblefield, 1993) |

Tabelle 1.1: Einige Definitionen von künstlicher Intelligenz

Turing glaubte um 1950, daß im Jahr 2000 ein Computer mit $10^9$ Einheiten Speicher so programmiert werden kann, daß er sich 5 Minuten mit einem Menschen unterhalten kann und zu 30 % den Turing-Test besteht.

**4.** In Punkt **2.** wurde von *korrekten* Inferenzen gesprochen. Oft liegt nicht genügend Information vor um sinnvolle Handlungen (beweisbar korrekte) auszuführen ($\leadsto$ Kapitel 8).

Die Welt ist i.A. nicht genügend spezifiziert. Außerdem kann man rational agieren, ohne *korrekte* Inferenzen machen: *Reflexe*.

**Dieser Ansatz wird in der gesamten Vorlesung verfolgt.**

Beim *totalen Turing-Test* kommt noch eine physikalische Interaktion hinzu. Also ein Video-Signal. Man braucht daher noch *Computer-Vision* und *Robotics*.

Die Frage *Können Maschinen denken?* führt zu 2 KI-Thesen:

**Schwache KI-These:** Man kann Maschinen bauen, die sich so verhalten als ob sie intelligent sind.

**Starke KI-These:** Maschinen die sich intelligent verhalten besitzen kognitive Zustände, d.h. *Geist*.

Man beachte, daß aus der starken keineswegs die schwache KI-These folgt.

### Das chinesische Zimmer

Searle hat dies 1980 als Beispiel gebracht, daß ein System den Turing-Test bestehen kann, aber die starke KI-These widerlegt. In einem Zimmer befinden sich

**CPU:** ein Mensch, der nur Englisch nicht aber Chinesisch spricht,

**Programm:** ein Buch, das Regeln enthält die in Englisch formuliert sind und ihm sagen wie chinesische Texte übersetzt werden.

**Speicher:** genügend Papier und Schreibmaterial.

Es werden ihm chinesisch beschriebene Blätter gereicht, die er mit Hilfe des Buches übersetzt. *Kann man sagen, daß das Zimmer Chinesisch versteht?*

## 1.2 Grundlagen

450 v. Chr. : **Plato**, **Sokrates**, **Aristoteles**
Sokr.: *Was ist charakteristisch für das Gute?*
Aris.: *Welche Gesetze regeln den Verstand?*
800 n. Chr. : **Al Chwarizmi**, Arabien
1300 : **Raymundus Lullus**: *Ars Magna*

1596–1650 : **R. Descartes**:
*Verstand = physikalisches System*
Freier Wille, Dualismus
1623–1662 : **B. Pascal, W. Schickard**
Additions-Maschinen
1646–1716 : **G. W. Leibniz**
*Materialismus*, greift Idee der Ars Magna auf
um mechanisches Gerät zur Simulation des
menschlichen Verstandes zu bauen
1561–1626 : **F. Bacon**, *Empirismus*
1632–1704 : **J. Locke**, *Empirismus*
*"Nihil est in intellectu quod non antefuerat in sensu."*
1711–1776 : **D. Hume**, *Induktion*
1724–1804 : **I. Kant**,
*"Der Verstand schöpft seine Gesetze*
*nicht aus der Natur, sondern schreibt sie dieser vor."*
1805 : **Jacquard**-Webstuhl
1815–1864 : **G. Boole**
Formale Sprache,
Logik als **mathematische** Disziplin
1792–1871 : **Ch. Babbage**
*Difference Engine*: Ber. von Log.-Tafeln
*Analytical Engine*: besitzt schon (1) adr. Speicher,
(2) bed. Sprünge und (3) Programmspeicher
1848–1925 : **G. Frege**,
Begriffsschrift: 2-dim. Notation für PL1
1862–1943 : **D. Hilbert**,
Vortrag 1900 in Paris: 23 Probleme
23. Problem: *Entscheidungsproblem*
1872–1970 : **B. Russell**,
1910: Principia Mathematica
Logischer Positivismus, Wiener Kreis (1920–40)
1902–1983 : **A. Tarski**
Wahrheitsbegriff in den formalisierten Sprachen, 1936
1906–1978 : **K. Gödel**,
Vollst/Unvollst. Sätze, 1930/31
1912–1954 : **A. Turing**
Turing-Maschine 1936
Berechenbarkeit
1903–1995 : **A. Church**,
$\lambda$-Kalkül, Church'sche These

1940 : Erster Computer “Heath Robinson” zum
Dechiffrieren deutscher Codes (Turing),
1943 Collossus
1941 : Erster frei programmierbarer Computer:
Z 3 von **K. Zuse** ($\rightsquigarrow$ Deutsches Museum)
Gleitkomma-Arithmetik,
1945 Plankalkül: erste Progammier-Hochsprache
1940-1945 : **H. Aiken** entwickelt MARK I, II, III.
ENIAC: erster elektronischer Computer
1952 : IBM 701, erster Computer der Profit abwirft

## 1.3 Historisches zur KI

### 1943

McCulloch und W. Pitts bringen 3 Dinge miteinander in Verbindung

1. Physiologie und Funktion der Neuronen im Gehirn.
2. Aussagenlogik nach Russell/Whitehead.
3. Turing’s Theorie der Berechenbarkeit.

Sie konstruieren ein abstraktes Modell bestehend aus vernetzten Neuronen und beweisen

- *Berechenbare Funktionen können durch solche Modelle berechnet werden.*
- *Aussagenlogische Junktoren werden durch sehr einfache Netze implementiert.*

### 1951

Minsky und Edwards bauen den ersten Computer der auf Neuronalen Netzen basiert (Princeton).

### 1952

A. Samuel entwickelt Mühle-Programme die lernen ihre Spielstärke zu verbessern

### 1956

2 monatiger Workshop in Dartmouth (McCarthy, Minsky, Shannon, Rochester). Es werden die *Zusammenhänge von Automatentheorie, Neuronalen Netzen und Intelligenzforschung* diskutiert. (10 Teilnehmer)

Newell und Simon treten mit einem lauffähigen Programm auf: *LT, Logical Theorist*. LT kann viele Theoreme aus Kap. 2 der Principia Mathematica beweisen. Sogar eines mit einem kürzeren Beweis: das renommierte *Journal of Symbolic Logic* lehnt jedoch das eingereichte Papier der Autoren *Newell, Simon und LT* ab. Newell/Simon behaupten das *Leib-Seele-Problem* gelöst zu haben.

Der Begriff **Artificial Intelligence** wird als Name der neuen Disziplin vorgeschlagen und etabliert sich.

*GPS: General Problem Solver* von Newell/Simon ist die Weiterentwicklung von LT. Die Idee ist nun, menschliches Problem-Lösen zu imitieren.

## 1958

Geburtsjahr der KI. McCarthy geht ans MIT und entwickelt

1. **Lisp**,
2. **Time-Sharing**,
3. **Programs with Common-Sense**.
   *Advice-Taker*: hypothetisches Programm, das als erstes vollständiges KI-SYSTEM angesehen werden kann. Im Unterschied zu LT und GPS wird nicht nur spezifisches Domänen-Wissen benutzt, sondern auch allgemeines Wissen über die Welt.

## 1959

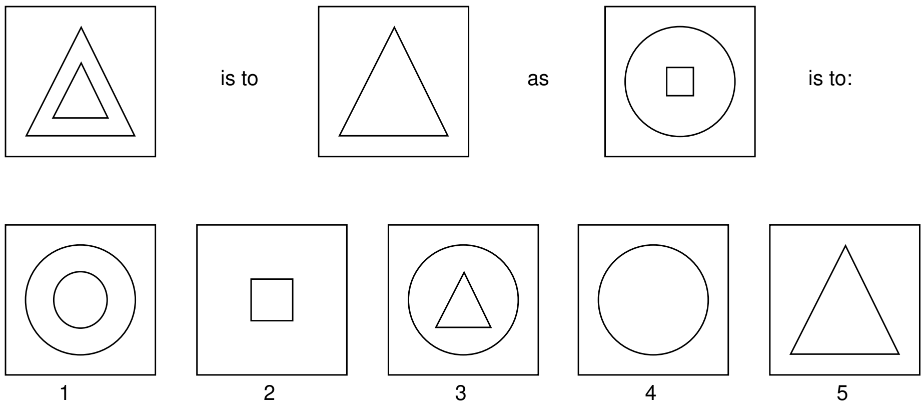
H. Gelernter: Geometrie Beweiser

## 1960-1966

McCarthy konzentriert sich auf *Wissensrepräsentation und Schließen in formaler Logik* ($\rightsquigarrow$ *Robinson's Resolution* (Kapitel 5),
$\rightsquigarrow$ *Green's Planner*, $\rightsquigarrow$ *Shakey* (Kapitel 6)).

Minsky konzentriert sich auf lauffähige Programme für spezielle Welten: *Mikrowelten*.

- SAINT, Integrations-Probleme des 1/2 Semesters
- ANALOGY, Einfache Analogien in IQ-Tests




Eine der wichtigsten Mikrowelten ist die *Blocksworld*. Weiterentwicklung der Ideen von McCulloch und Pitts. **Perceptron** von Rosenblatt und Konvergenz-Theorem für Perceptronen:

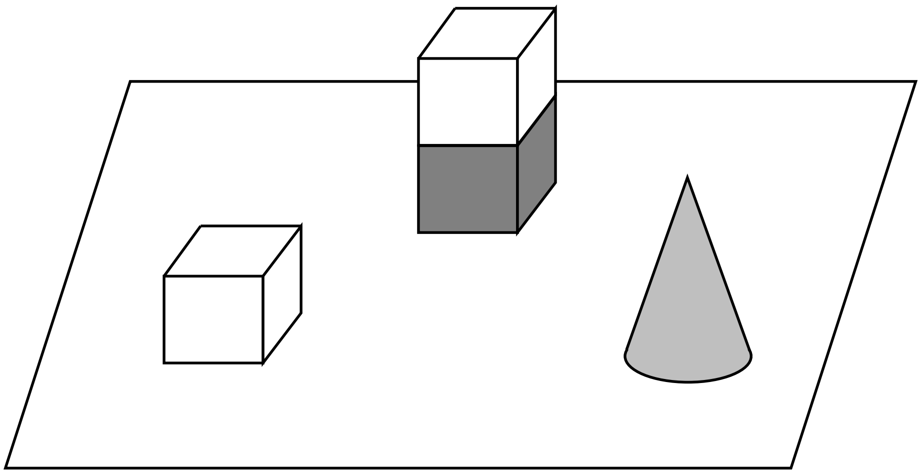


Abbildung 1.1: Die Block-Welt

- *Es gibt einen Lern-Algorithmus der ein Perceptron so aufbaut, daß es mit den Input-Daten übereinstimmt.*

Insgesamt gab es große Versprechungen für die Zukunft. Zunächst auch Anfangserfolge, aber nur miserable weiterführende Ergebnisse.

**1966:** Einstellen aller US-Funds für Übersetzungs-Syst. Angeblich wurde der Satz

*The spirit is willing but the flesh is weak.*

mit

*The vodka is good but the meat is rotten.*

übersetzt.

## 1966–1974

Simon 1958: *"In 10 Jahren wird der Schachweltmeister ein Computer sein."* Einfache Probleme deshalb lösbar, weil Suchraum klein. Ernsthafte Probleme bleiben unlösbar. *Hoffnung:* Schnellere Hardware, größerer Speicher werden alles lösen. $\leadsto$ NP-vollständige Probleme in Kapitel5, (S. Cook/R. Karp 1971/72), $P \neq NP$ ?

## 1973

Lighthill-Report führt zur Einstellung der Forschungsgelder für KI in Großbritannien. Erste Resultate über grundsätzliche Grenzen einiger Ansätze (Minsky's *"Perceptrons"*).

**1969–79** *Wissensbasierte Systeme*

*General Purpose*-Mechanismen benutzen keine oder nur sehr wenig Wissen über ein Anwendungsgebiet. Für komplexe Gebiete führt dies zur kombinatorischen Explosion. *Man benutze typisches Anwendungswissen und trenne dies von der deduktiven Komponente.*

*Beispiel:* 1969 DENDRAL. Dies führt zu Expertensystemen wie MYCIN (Diagnose von Blut-Infektionen).

**1973**

*PROLOG:* "Programmation en Logique"

DENDRAL bestimmt die Molekül-Struktur aus vorliegenden Massenspektren. Die naive Version funktioniert nicht (zähle alle Möglichkeiten auf und sondere jene aus, die den Messungen widersprechen) da es zuviele sind. Man betrachtet *special purpose rules* die, falls anwendbar schon eine gewisse Struktur vorgeben und damit die Möglichkeiten einschränken. Z.B. falls dort und dort ein Peak ist, dann enthält das Molekül eine **C=O** Bindung.

### Ein Schmankerl

1902 wurde ein deutsches Gedicht ins Japanische übersetzt. Die japanische Fassung wurde dann 1911 ins Französische übersetzt. Schließlich wurde diese Fassung (in der Annahme es handele sich um ein japanisches Gedicht) ins Deutsche übertragen.
Heraus kam schließlich folgendes:

> *Stille ist im Pavillon aus Jade,*
> *Krähen fliegen stumm*
> *Zu beschneiten Kirschbäumen im Mondlicht.*
> *Ich sitze*
> *und weine.*

Wie lautet das ursprüngliche Gedicht?

## 1.4 Aufgaben

**Aufgabe 1.1 (Was ist Intelligenz?)**
*Stimmen die folgenden Aussagen und Begründungen Ihrer Meinung nach?*

**Tiere** *können niemals Intelligenz besitzen; denn sie tun nur das, was ihre Gene ihnen vorgeben.*

**Computer** *können niemals Intelligenz besitzen, denn Sie tun nur das, was ein Programmierer in sie eingegeben hat.*

**Aufgabe 1.2 (Was ist KI?)**
*Wir wollen in der Übungsstunde eine Diskussion führen. Die folgenden Aufgaben sollen dazu als Grundlage dienen.*

1. *Was ist „Künstliche Intelligenz“ (abgekürzt KI, engl. „Artificial Intelligence“)? Überlegen Sie sich eine Definition! Konsultieren Sie Lexika, Lehrbücher, Zeitschriften etc.*

2. *Stellen Sie (wenige) Thesen auf, welches Ziel, welche möglichen Anwendungsgebiete und welchen Sinn die Forschung im Bereich „Künstliche Intelligenz“ hat bzw. haben soll – Ihrer Meinung nach!*

# 2 Intelligente Agenten

## 2.1 Definitionen von Agenten

**Definition 2.1.1 (Agent)**
*Ein Agent ist etwas, das seine Umgebung mittels Sensoren wahrnimmt und in dieser Umgebung mittels Werkzeugen agiert.*

**Definition 2.1.2 (Rationaler, allwissender Agent)**
*Rationale Agenten sind solche, die das* **Richtige** *tun. (Ein Maß für die Performanz wird benötigt.) Allwissende Agenten sind solche, die die Resultate ihrer Aktionen im voraus wissen.*

Rationale Agenten sind i.a. nicht allwissend! Aphorismus von Karl Kraus: *Im Zweifelsfall entscheide man sich für das richtige.*

Wie ist das **Richtige** definiert und wovon hängt es ab?

1. Performanz-Maß (möglichst objektiv),
2. Percept-Sequenz: was bisher wahrgenommen wurde,
3. Wissen des Agenten über die Umgebung,
4. wie der Agent agieren kann.

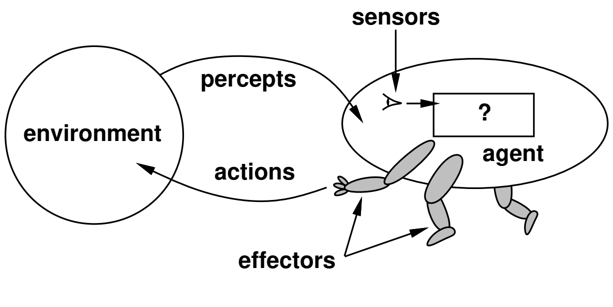



Abbildung 2.1: Ein Agent.

| Agent Type | Percepts | Actions | Goals | Environment |
|---|---|---|---|---|
| Medical diagnosis system | Symptoms, findings, patient's answers | Questions, tests, treatments | Healthy patient, minimize costs | Patient, hospital |
| Satellite image analysis system | Pixels of varying intensity, color | Print a categorization of scene | Correct categorization | Images from orbiting satellite |
| Part-picking robot | Pixels of varying intensity | Pick up parts and sort into bins | Place parts in correct bins | Conveyor belt with parts |
| Refinery controller | Temperature, pressure readings | Open, close valves; adjust temperature | Maximize purity, yield, safety | Refinery |
| Interactive English tutor | Typed words | Print exercises, suggestions, corrections | Maximize student's score on test | Set of students |

Tabelle 2.1: PAGE-Beschreibung

**Definition 2.1.3 (Idealer rationaler Agent)**
*Ein idealer rationaler Agent wählt für jede Percept-Sequenz genau die Aktion aus, die sein Performanz-Maß maximiert (bei gegebenem Wissen über die Umgebung).*

Agenten können durch Abbildungen

$$\text{Menge der Percept-Sequenzen} \mapsto \text{Menge der Aktionen}$$

mathematisch beschrieben werden. Die interne Struktur eines Agenten ist

**Agent = Architektur + Programm**

Die KI beschäftigt sich damit, Agenten-Programme zu entwerfen.

## Agenten und ihre PAGE-Beschreibung

*Einfachste Form von Agenten-Programmen:*

```
function SKELETON-AGENT( percept) returns action
   static: memory, the agent's memory of the world

   memory ← UPDATE-MEMORY(memory, percept)
   action ← CHOOSE-BEST-ACTION(memory)
   memory ← UPDATE-MEMORY(memory, action)
   return action
```

*Theoretisch ist alles trivial, da man ja in einer Tabelle nachschauen kann:*

```
function TABLE-DRIVEN-AGENT( percept) returns action
   static: percepts, a sequence, initially empty
           table, a table, indexed by percept sequences, initially fully specified

   append percept to the end of percepts
   action ← LOOKUP( percepts, table)
   return action
```

## 2.2 Beispiele von Agenten

**Beispiel 2.2.1 (Taxi-Fahrer)**

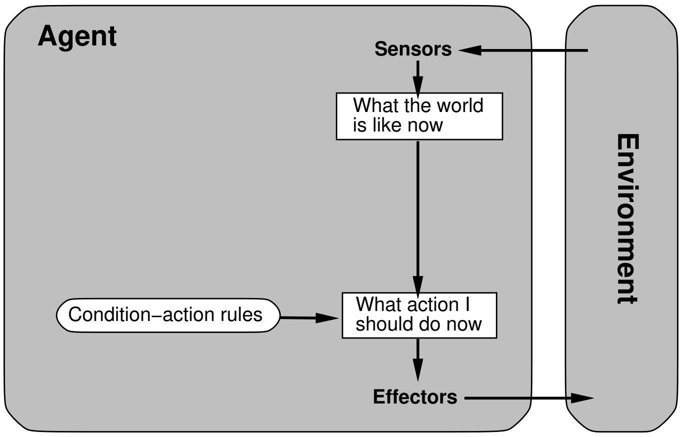
| Agent Type | Percepts | Actions | Goals | Environment |
|---|---|---|---|---|
| Taxi driver | Cameras, speedometer, GPS, sonar, microphone | Steer, accelerate, brake, talk to passenger | Safe, fast, legal, comfortable trip, maximize profits | Roads, other traffic, pedestrians, customers |

Wie können wir ein Programm entwerfen? Wir können das Problem unter verschiedenen Gesichtspunkten sehen:

1. Agenten, die auf einfache Reflexe reagieren,
2. Agenten, die berücksichtigen wie sich die Welt durch ihre Aktionen ändert,
3. Agenten, die zielgerichtet agieren,
4. Agenten, die so agieren, daß Zustände bevorzugt werden, die besonders nützlich sind (bezüglich eines vorgegebenen Performanz-Maßes).

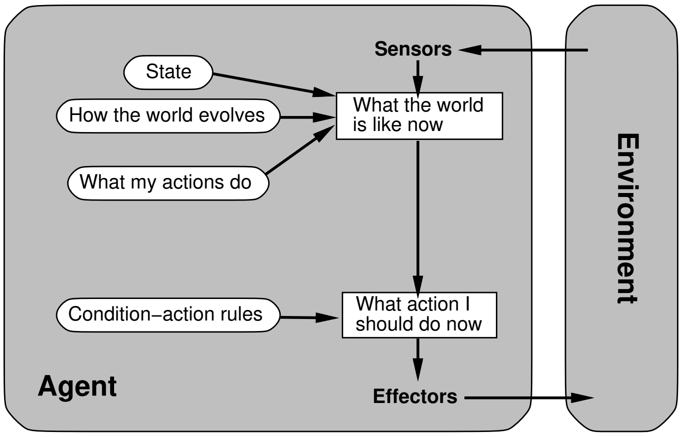
Hier einige Beispiele:

1. *Produktions-Regeln:* Falls der Vorgänger bremst, dann bremse auch.



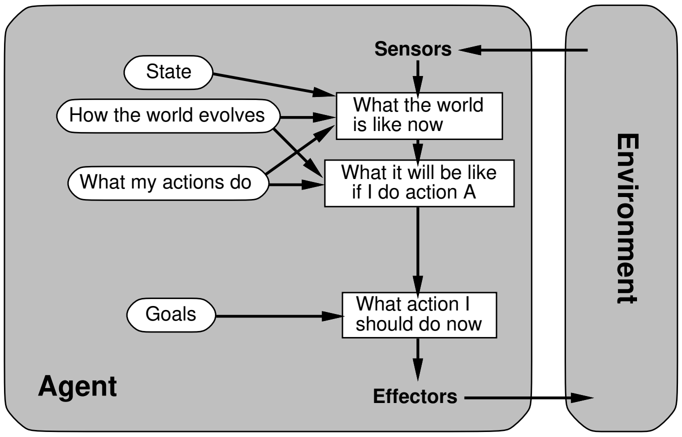

```
function SIMPLE-REFLEX-AGENT(percept) returns action
   static: rules, a set of condition-action rules

   state ← INTERPRET-INPUT(percept)
   rule ← RULE-MATCH(state, rules)
   action ← RULE-ACTION[rule]
   return action
```

2. *Zusätzlich interne Zustände:* was der Taxifahrer im Spiegel gesehen hat, sollte er sich merken.

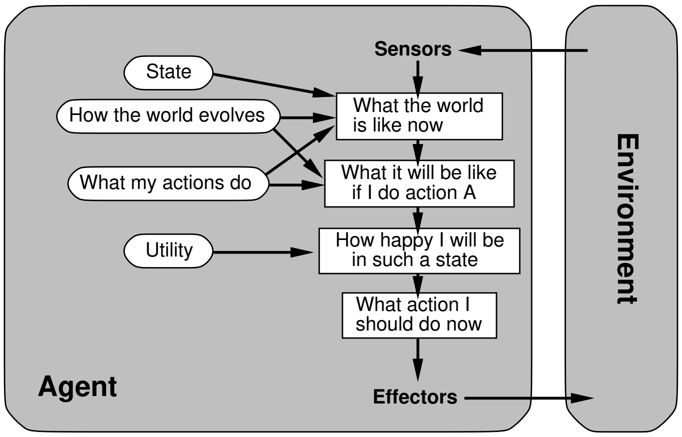


```
function REFLEX-AGENT-WITH-STATE( percept) returns action
   static: state, a description of the current world state
           rules, a set of condition-action rules

   state ← UPDATE-STATE(state, percept)
   rule ← RULE-MATCH(state, rules)
   action ← RULE-ACTION[rule]
   state ← UPDATE-STATE(state, action)
   return action
```

3. führt später zum *Planen*. Verfolgen von Zielen, Erkennen von Folgen.




4. z.B.: schnell, billig, sicher.




*Wie beeinflussen Eigenschaften der Umgebung das Design eines dazu passenden Agenten?* Hier einige grundlegende Eigenschaften:

**Zugänglich/Unzugänglich:** Falls nicht vollst. zugängl. braucht man interne Zustände.

**Determinist./Indeterm.:** Falls die Umgebung unzugänglich ist, kann sie indeterministisch scheinen, obwohl sie es nicht ist.

**Episodisch/Nichtepisodisch:** Percept-Aktion-Sequenzen sind voneinander unabhängig. Abgeschlossene Episoden.

**Statisch/Dynamisch:** Dynamisch heißt: während der Agent noch überlegt, ändert sich schon die Welt. Semi-dynamisch heißt: Zwar ändert sich nicht die Welt, aber das Performanz-Maß des Agenten.

**Diskret/Stetig:** bezieht sich auf die Menge der Wahrnehmungen und Aktionen.

| Environment | Accessible | Deterministic | Episodic | Static | Discrete |
|---|---|---|---|---|---|
| Chess with a clock | Yes | Yes | No | Semi | Yes |
| Chess without a clock | Yes | Yes | No | Yes | Yes |
| Poker | No | No | No | Yes | Yes |
| Backgammon | Yes | No | No | Yes | Yes |
| Taxi driving | No | No | No | No | No |
| Medical diagnosis system | No | No | No | No | No |
| Image-analysis system | Yes | Yes | Yes | Semi | No |
| Part-picking robot | No | No | Yes | No | No |
| Refinery controller | No | No | No | No | No |
| Interactive English tutor | No | No | No | No | Yes |

## 2.3 Aufgaben

**Aufgabe 2.1 (Rationale Agenten)**
*Betrachten Sie eine „Staubsauger-Welt“, d.h. einen Roboter, der eine Wohnung ganz sauber staubsaugt.*

1. *Schreiben Sie eine ausführliche PAGE-Beschreibung (percepts, actions, goals, environment) für einen intelligenten Agenten, d.h. den Roboter, in dieser Umgebung (environment).*

2. *Charakterisieren Sie die Umgebung hinsichtlich der folgenden Eigenschaften:*

   - *zugänglich / unzugänglich,*
   - *deterministisch / indeterministisch,*
   - *episodisch / nicht episodisch,*
   - *statisch / dynamisch,*
   - *diskret / stetig*

# 3 Problem-Lösen durch Suchen

**Gesucht:** *Agenten, die Probleme lösen.* Diese sind offensichtlich eine Teilklasse der Ziel-basierten Agenten.

**Struktur:** *Formuliere, Suche, Führe aus.*

**Formuliere:**

- **Ziel:** eine Menge von Zuständen,
- **Problem:** Zustände, Aktionen, die Zustande ineinander überführen, Transitionen

**Suche:** Welche Folge von Aktionen ist hilfreich?

**Führe aus:** Man wende die Lösungsfolge von Aktionen auf den Anfangszustand an.

## 3.1 Problem-Formulierung

Wir unterscheiden zwischen 4 verschiedenen Typen:

1. **1-Zustand-Probleme:** Aktionen sind vollständig beschrieben. Vollständige Information durch Sensoren zur eindeutigen Bestimmung des gegenwärtigen Zustands.

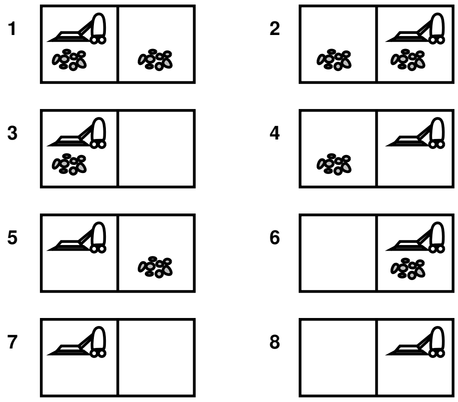
2. **Multiple-Zustand-Probleme:** Aktionen sind vollständig beschrieben, aber der Anfangszustand liegt nicht fest.

3. **Kontingenz-Probleme:** Manchmal gibt es keine fixe Sequenz als Lösung, sondern man muss den ganzen Baum betrachten.
($\rightsquigarrow$ Übung: Murphy's Gesetz, $\rightsquigarrow$ Kapitel 6)

4. **Explorations-Probleme:** Nicht einmal die Effekte der Aktionen sind bekannt. Man muss in der Welt suchen, nicht im abstrakten Modell.

```
function SIMPLE-PROBLEM-SOLVING-AGENT(p) returns an action
  inputs: p, a percept
  static: s, an action sequence, initially empty
          state, some description of the current world state
          g, a goal, initially null
          problem, a problem formulation

  state ← UPDATE-STATE(state, p)
  if s is empty then
     g ← FORMULATE-GOAL(state)
     problem ← FORMULATE-PROBLEM(state, g)
     s ← SEARCH(problem)
  action ← RECOMMENDATION(s, state)
  s ← REMAINDER(s, state)
  return action
```

Tabelle 3.1: Einfacher Problem-lösender Agent.




**Definition 3.1.1 (1-Zustand-Problem)**
*Ein 1-Zustand-Problem besteht aus*

- *einer Menge von Zuständen, darunter einer der als* Anfangszustand *ausgezeichnet ist,*
- *einer Menge von* $n$ Aktionen *(wir nennen sie auch* Operatoren*), die, in einem Zustand ausgeführt, zu einem anderen Zustand überleiten:*

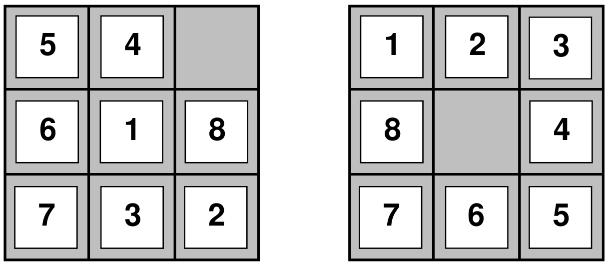
$$\textit{Operator}_i\textit{: Zustände} \mapsto \textit{Zustände}, \; i = 1, \ldots, n$$

- *einer Menge von* Ziel-Zuständen*, oder einem Ziel-Test, der, angewandt auf einen Zustand, bestimmt ob es sich um einen Ziel-Zustand handelt oder nicht,*
- *einer Kosten Funktion $g$, die jedem Pfad im Zustandsraum (Menge der erreichbaren Zustände) gewisse Kosten zuordnet. Üblicherweise ist $g$ additiv.*

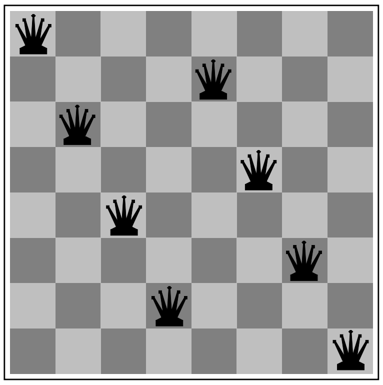
Wie geht das für Multiple Zustand-Probleme? Wie wählt man Aktionen und Zustände aus? Durch geeignete *Abstraktion*.

## 3.2 Einige Beispiele

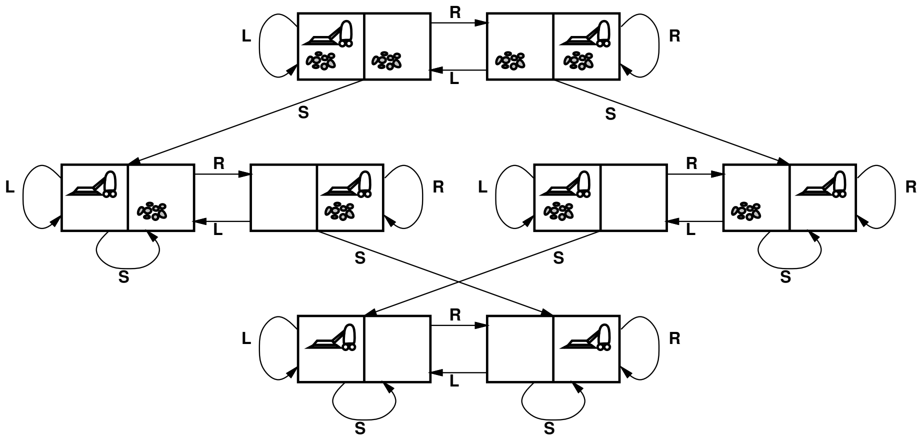
**Spiel-Beispiele:**




Start State Goal State



**Staubsauger Welt**



```
function GENERAL-SEARCH( problem, strategy) returns a solution, or failure
  initialize the search tree using the initial state of problem
  loop do
      if there are no candidates for expansion then return failure
      choose a leaf node for expansion according to strategy
      if the node contains a goal state then return the corresponding solution
      else expand the node and add the resulting nodes to the search tree
  end
```

Tabelle 3.2: Allgemeine Suche.

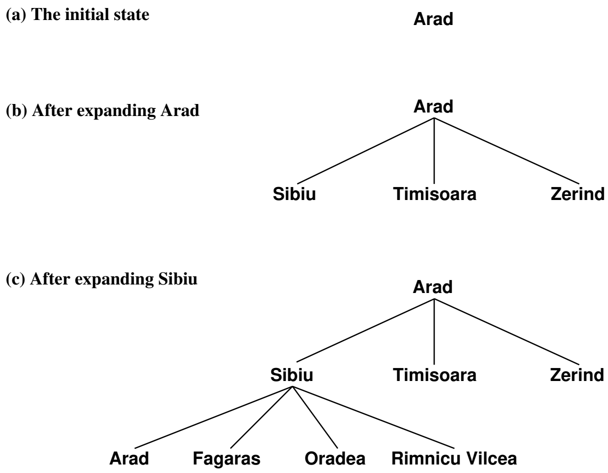
- Missionare vs. Kannibalen

**Real-World-Probleme:**

- Travelling-Salesman-Problem
- VLSI-Layout
- Landkarten beschriften
- Roboter-Navigation

## 3.3 Such-Strategien (allgemein)

*Prinzip: Auswählen, Testen, Expandieren.* ($\leadsto$ Such-Baum)




**Zustands-Raum versus Such-Baum:** Während der Zustandsraum üblicherweise endlich ist, ist der Such-Raum meist abzählbar unendlich.

**Definition 3.3.1 (Datentyp Knoten)**
*Der Datentyp* Knoten *ist durch die Komponenten* Zustand, Elter-Knoten, Operator, Tiefe *und* Pfad-Kosten *definiert.*

- Ein Knoten ist ein Buchhalter bezüglich einer Problem-Instanz und bezüglich eines bestimmten Algorithmus.
- Ein Zustand ist ein Schnappschuss der Welt.

Beachten Sie die *rekursive* Abhängigkeit (Knoten/Elter-Knoten). Falls wir die *Tiefe* weglassen, müssen wir einen ausgezeichneten Knoten $root$ einführen, die *Wurzel*. Dann ist die Tiefe *definierbar*. Umgekehrt ist $root$ durch die Tiefe *definierbar*: der Knoten, der sich selbst als Elter hat und die Tiefe 0.

Wir wollen versuchen, die Funktion GENERAL-SEARCH etwas konkreter zu machen und treffen folgende *Design-Entscheidung:* Die zu expandierenden Knoten stellen wir nicht als Menge sondern als **Queue** dar.

| | |
|---|---|
| **Make-Queue(Elements)** | **Remove-Front(Queue)** |
| **Empty?(Queue)** | **Queing-Fn(Queue,Elements)** |

```
function GENERAL-SEARCH(problem, QUEUING-FN) returns a solution, or failure

  nodes ← MAKE-QUEUE(MAKE-NODE(INITIAL-STATE[problem]))
  loop do
      if nodes is empty then return failure
      node ← REMOVE-FRONT(nodes)
      if GOAL-TEST[problem] applied to STATE(node) succeeds then return node
      nodes ← QUEUING-FN(nodes, EXPAND(node, OPERATORS[problem]))
  end
```

**Was sind interessante Anforderungen an Such-Strategien?**

- *Vollständigkeit*,
- *Zeit-Komplexität*,
- *Speicher-Komplexität*,
- *Optimalität (bez. Pfad-Kosten)*

Wir unterscheiden zwischen **Uninformierter** und **informierter Suche**.

## 3.4 Uninformierte Suche

**Breadth-First-Suche:** “Knoten der kleinsten Tiefe werden zuerst expandiert”,

Queing-Fn($Queue, Expand(\ldots)$) : neue Knoten ans Ende

**Vollständig:** Ja.
**Optimal:** Ja, falls alle Operatoren dieselben Kosten haben.

Für einen konstanten Verzweigungsfaktor $b$ gilt: In der Tiefe $d$ gibt es maximal

$$1 + b + b^2 + \ldots + b^{d-1} + b^d\text{-viele Knoten.}$$

| Depth | Nodes | Time | Memory |
|---|---|---|---|
| 0 | 1 | 1 millisecond | 100 bytes |
| 2 | 111 | .1 seconds | 11 kilobytes |
| 4 | 11,111 | 11 seconds | 1 megabyte |
| 6 | $10^6$ | 18 minutes | 111 megabytes |
| 8 | $10^8$ | 31 hours | 11 gigabytes |
| 10 | $10^{10}$ | 128 days | 1 terabyte |
| 12 | $10^{12}$ | 35 years | 111 terabytes |
| 14 | $10^{14}$ | 3500 years | 11,111 terabytes |

**Uniform-Cost-Suche:** "Knoten $n$ mit kleinsten Pfad-Kosten $g(n)$ werden zuerst expandiert",

Queing-Fn($Queue, Expand(\ldots)$) : neue Knoten werden gemäss ihrer Pfad-Kosten mit denen in der Queue verglichen und entsprechend eingefügt

**Vollständig:** Ja, falls jeder Operator die Pfadkosten um mindestens einen Betrag $\delta > 0$ anwachsen lässt (siehe unten).

Falls alle Operatoren die gleichen Kosten haben (insbesondere wenn $g(n) =$ Tiefe($n$)) gilt: Uniform-Cost=Breadth-First.

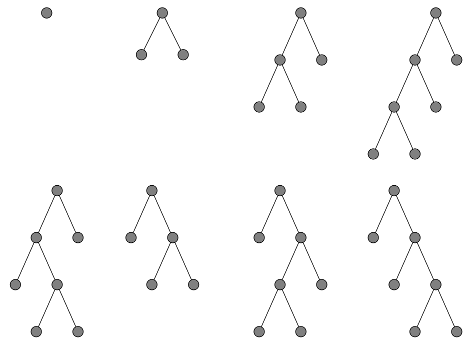
**Theorem 3.4.1 (Optimalität von Uniform-Cost)**
*Gilt* $\exists\delta > 0 : \; g(succ(n)) \geq g(n) + \delta$ *so ist Uniform-Cost optimal.*

**Depth-First-Suche:** "Knoten der grössten Tiefe werden zuerst expandiert".

Queing-Fn($Queue, Expand(\ldots)$) : neue Knoten werden am Anfang eingefügt

Falls Verzweigungsfaktor $b$ konstant und max. Tiefe $m$:

- **Speicherkomplexität:** $b \times m$,
- **Zeitkomplexität:** $b^m$.

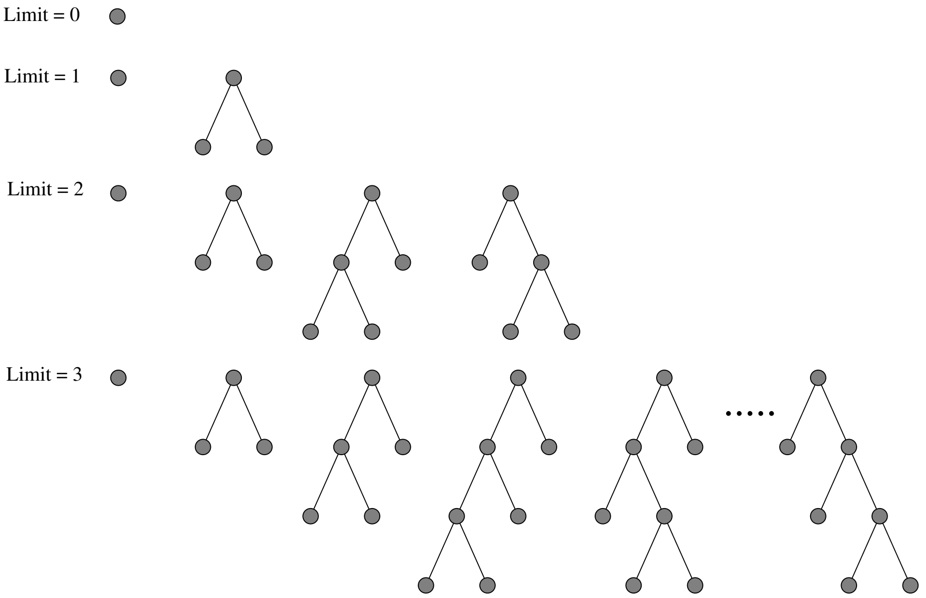


**Depth-First mit Tiefenbeschränkung $d$:** “gehe höchstens in Tiefe $d$”.

Insgesamt gibt es $1 + b + b^2 + \ldots + b^{d-1} + b^d$-viele Expansionsschritte.

**Iterative Deepening:** “Tiefe wird immer grösser”

```
function ITERATIVE-DEEPENING-SEARCH( problem) returns a solution sequence
   inputs: problem, a problem

   for depth ← 0 to ∞ do
      if DEPTH-LIMITED-SEARCH( problem, depth) succeeds then return its result
   end
   return failure
```




Wieviele Expansionsschritte gibt es insgesamt? Genau

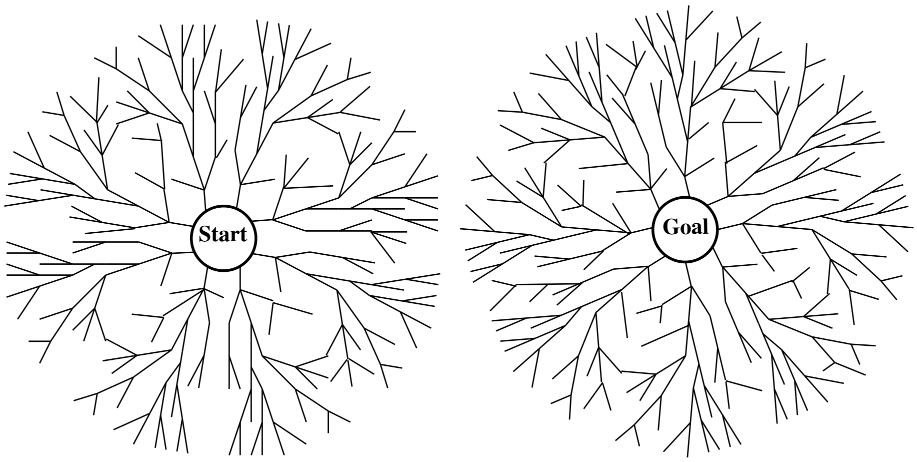
$$(d+1) \times 1 + d \times b + (d-1) \times b^2 + \ldots + 2 \times b^{d-1} + 1 \times b^d\text{-viele.}$$

| Criterion | Breadth-First | Uniform-Cost | Depth-First | Depth-Limited | Iterative Deepening | Bidirectional (if applicable) |
|---|---|---|---|---|---|---|
| Time | $b^d$ | $b^d$ | $b^m$ | $b^l$ | $b^d$ | $b^{d/2}$ |
| Space | $b^d$ | $b^d$ | $bm$ | $bl$ | $bd$ | $b^{d/2}$ |
| Optimal? | Yes | Yes | No | No | Yes | Yes |
| Complete? | Yes | Yes | No | Yes, if $l \geq d$ | Yes | Yes |

Tabelle 3.3: Eigenschaften von Such-Algorithmen

Man beachte, daß nur etwa doppelt soviele Schritte im Vergleich zu Depth-First mit Tiefenbeschränkung ausgeführt werden ($\rightsquigarrow$ geometrische Reihe). Der Aufwand für mehrfach berechnete Knoten hält sich also in Grenzen.

**Bidirektionale Suche:**




## 3.5 Aufgaben

**Aufgabe 3.1 (General Search)**
*Der General-Search-Algorithmus umfaßt drei Schritte: (i) Test auf Zielzustand, (ii) Generierungsphase und (iii) Anordnungsfunktion, in dieser Reihenfolge. Ein gewisser Nachteil dieser Formulierung besteht darin, daß nicht sofort erkannt wird, wenn generierte Knoten Zielzustände darstellen, weil die Anordnungsfunktion sie nicht an die erste Stelle plaziert. Schaffen Sie Abhilfe, indem sie den Algorithmus auf zwei Arten neu formulieren:*

1. *Jeder Knoten wird sofort nach seiner Generierung getestet.*
2. *Die Anordnungsfunktion wird entsprechend angepaßt.*

**Aufgabe 3.2 (Chain-Problem)**
*Das* Chain-Problem *besteht darin, Ketten verschiedener Länge umzubauen. Operatoren in diesem Zusammenhang sind* ÖFFNEN *und* SCHLIESSEN *eines Gliedes. In der Standard-Formulierung des Problems besteht der Anfangszustand aus vier Ketten mit jeweils drei Gliedern; der Zielzustand ist eine geschlossene Kette aus allen zwölf Gliedern. Definieren Sie das Problem formal und bestimmen Sie eine kürzeste Lösung!*

**Aufgabe 3.3 (Suche mit Prolog)**
*Gegeben sei der Graph $G(n)$ für $n \in \mathbb{N}$, bestehend aus einem Startknoten $start$ und den folgenden Knoten: $incest(i,j)$ für $i \in 0..1$ und $j \in 1..n$; $cycle(k)$ für $k \in 1..2$; $tree(l)$ für $l \in 1..2^n - 1$. Zwischen den Knoten bestehen folgende Verbindungen:*

$start \to incest(0,1)$, $start \to incest(1,1)$
$incest(i,j) \to incest(i',j+1)$ *für* $i, i' \in 0..1$ *und* $j < n$
$incest(1,n) \to cycle(1)$
$cycle(1) \to cycle(2)$, $cycle(2) \to cycle(1)$
$cycle(2) \to tree(1)$
$tree(l) \to tree(2*l)$, $tree(l) \to tree(2*l+1)$ *für* $l < 2^{n-1}$

*Dabei bedeutet die Schreibweise $u \to v$, daß es eine Verbindungskante von Knoten $u$ nach $v$ gibt (aber nicht notwendigerweise umgekehrt).*

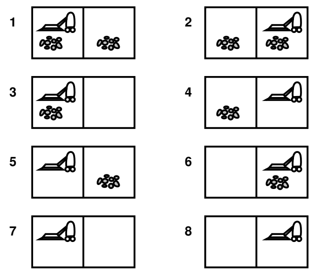
1. *Zeichnen Sie den Graphen $G(3)$! Beschreiben Sie, welche Probleme sich ergeben können bei der Suche eines Weges vom Startknoten aus! In welcher Reihenfolge werden bei Tiefen- und Breitensuche die Knoten im Graphen $G(3)$ besucht?*

2. *Programmieren Sie Tiefen- und Breitensuche in Graphen explizit in Prolog. Der zu untersuchende Graph sei dabei gegeben durch Fakten zum Prädikat $edge/2$. Der Startknoten $S$ sei durch das Faktum $init(S)$. ausgezeichnet.*

   *Definieren Sie ein Prädikat $search/2$, welches im zweiten Argument alle vom Startknoten aus erreichbaren Knoten genau einmal nacheinander durch Backtracking berechnet, und zwar in der durch die Suchstrategie festgelegten Reihenfolge. Jeder Knoten soll nur genau einmal berechnet werden.*

**Aufgabe 3.4 (Multiple-Zustand-Probleme)**
*In der Vorlesung haben Sie 1-Zustand-Probleme kennengelernt. Verallgemeinern Sie die Definition auf Multiple-Zustand-Probleme!*

**Aufgabe 3.5 (Murphy's Law)**
*Betrachten Sie die folgende sehr eingeschränkte und zugängliche Staubsauger-Welt mit acht Zuständen.*



*Zeigen Sie, daß zu jedem Anfangszustand eine Sequenz von Aktionen existiert, die einen Zielstand erreicht, d.h. beide Gebiete sind sauber, selbst wenn* Murphy's Law *gilt:*

> *Die Aktion* SAUGEN *verursacht manchmal Dreck, wenn vorher kein Schmutz dagewesen ist.*

*Zeichnen Sie auch ein Zustandübergangsdiagramm!*

# 4 Informierte Suche

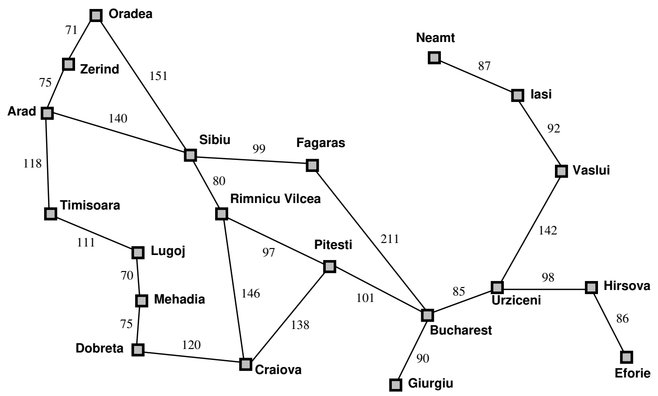
*Nutze Problem-spezifisches Wissen um die Suche intelligenter zu machen.* **General-Search** in Tabelle 3.2 ist schon recht weit instantiiert. Einzige Freiheit: Queing-Fn.

## 4.1 Best-First-Suche

Angenommen wir haben eine Evaluierungsfunktion $f$ die jedem Knoten $n$ einen Wert $f(n)$ zuordnet. Wir ändern nun die Queing-Fn so ab, daß die Knoten mit kleinstem $f$-Wert ganz vorne in der Queue liegen: die Queue wird also gemäss $f$ geordnet.

```
function BEST-FIRST-SEARCH(problem, EVAL-FN) returns a solution sequence
  inputs: problem, a problem
          Eval-Fn, an evaluation function

  Queueing-Fn ← a function that orders nodes by EVAL-FN
  return GENERAL-SEARCH(problem, Queueing-Fn)
```

**Beispiel 4.1.1 (Weg-Suche)**



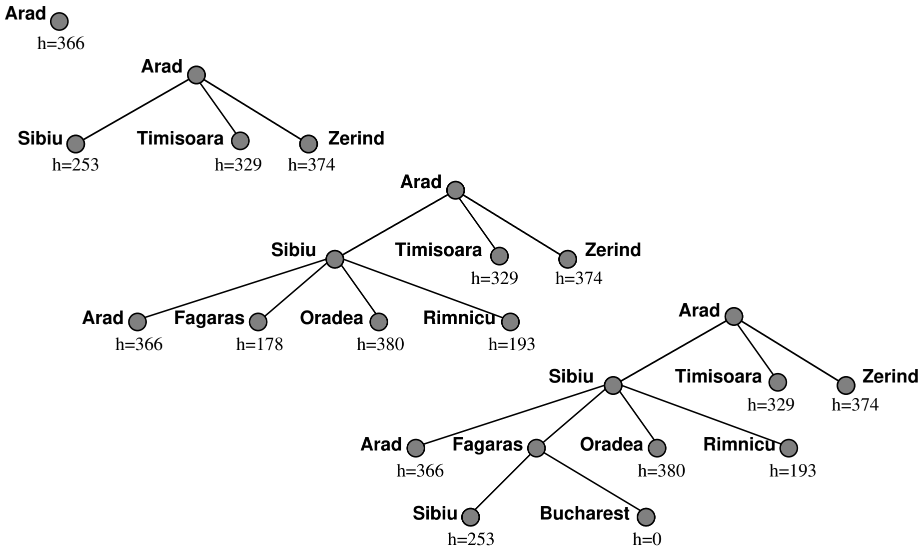

Straight−line distance to Bucharest

| | |
|---|---|
| **Arad** | 366 |
| **Bucharest** | 0 |
| **Craiova** | 160 |
| **Dobreta** | 242 |
| **Eforie** | 161 |
| **Fagaras** | 178 |
| **Giurgiu** | 77 |
| **Hirsova** | 151 |
| **Iasi** | 226 |
| **Lugoj** | 244 |
| **Mehadia** | 241 |
| **Neamt** | 234 |
| **Oradea** | 380 |
| **Pitesti** | 98 |
| **Rimnicu Vilcea** | 193 |
| **Sibiu** | 253 |
| **Timisoara** | 329 |
| **Urziceni** | 80 |
| **Vaslui** | 199 |
| **Zerind** | 374 |

*$h(n)$ kann als Luftlinien-Entfernung von Budapest bis zu der zu $n$ gehörenden Stadt definiert werden.*

**Greedy-Suche:** Hier benutzt man als Evaluierungsfunktion Eval_Fn($n$):= die erwarteten Kosten des *optimalen* Pfades vom Zustand in $n$ zum Ziel-Zustand. Optimal bedeutet dabei bezüglich der gegebenen Kostenfunktion $g$.

Diese Evaluierungsfunktion bezeichnet man auch als Heuristikfunktion uns schreibt sie als $h(n)$.




Ist die Greedy[1]-Suche optimal? Ist sie vollständig?

Greedy ist nicht optimal (in Beispiel wird ein nicht optimaler Weg gefunden). Greedy ist auch nicht vollständig, denn es ist Oszillation möglich: betrachte die Aufgabe einen Weg von Iasi nach Fagaras zu finden.

**A$^*$-Suche:** Hier benutzt man als Evaluierungsfunktion Eval_Fn($n$) die Summe aus einer Heuristikfunktion $h(n)$ und den tatsächlichen Pfadkosten $g(n)$ :

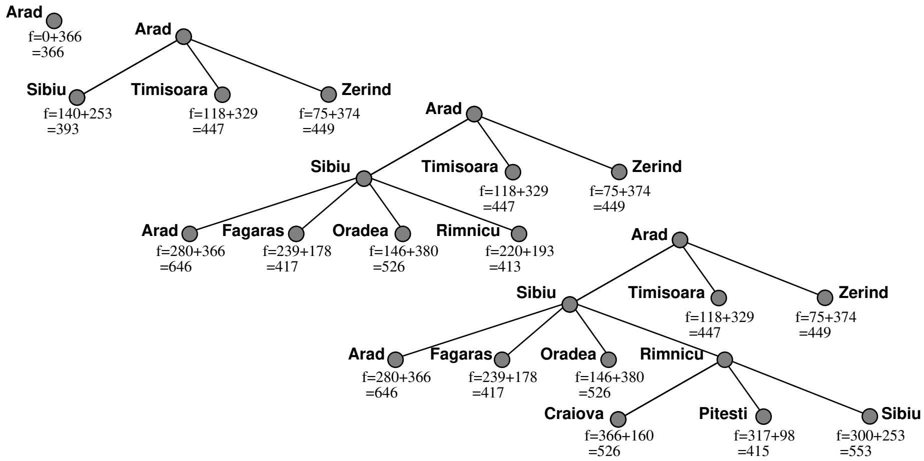
$$\text{Eval\_Fn}(n := h(n) + g(n)$$

Diese Evaluierungsfunktion schreiben wir auch einfach als $f(n)$.

A$^*$-Suche ist also "Greedy + Uniform-Cost", denn für Zielzustände $n_z$ gilt ja $h(n_z) = 0$, also Eval_Fn$(n_z) = g(n_z)$.

Wir stellen allerdings eine Bedingung an die Heuristikfunktion $h$: $h(n)$ soll immer kleiner sein als die optimalen Kosten von $n$ zu einem Ziel-Knoten ($h(n) < g(n_z) - g(n)$)

[1] "Greedy" heisst gierig: die Gier ist bekanntlich eine der sieben Todsünden.




In unserem Beispiel ist $f$ := Eval_Fn monoton. Im allgemeinen gilt das nicht. Allerdings können wir $f$ leicht monoton wachsend machen:

**Definition 4.1.2 ($f \leadsto$ monotones$f$)**
*Wir definieren eine modifizierte Funktion $f^{mon}$ rekursiv nach der Knotentiefe*

$$f^{mon}(n) := max\{f^{mon}(n'), f(n)\}$$

*wobei $n'$ der Elter von $n$ ist.*

**Theorem 4.1.3 (Optimalität von A$^*$)**
*A$^*$ ist optimal.*

Hier ein Beispiel für nichtmonotones $f$. Sei $n'$ der Elter von $n$. Sei $g(n') = 3$ und $g(n) = 4$, und $h(n') = 4$ und $h(n) = 2$. Damit ist $f(n') = 7$ und $f(n) = 6$.

**Beweis:** Sei G ein optimaler Zielzustand, und $G'$ ein suboptimaler, d.h. $g(G') > g(G)$. Angenommen, A$^*$ hat $G'$ aus der Queue ausgewählt. Dann hat A$^*$ irgendwann einmal einen Knoten $n_0$ der zum optimalen Pfad nach $G$ führt "verschmäht" (also nicht expandiert), d.h. $n_0$ kam in die Queue. Nun gilt aber

$$g(G) \geq f(n_0)$$

wegen der Bedingung an $h$, und es gilt auch

$$f(n_0) \geq f(G')$$

da ja $G'$ und nicht $n_0$ (der noch in der Queue sein muss) ausgewählt wurde.

Nimmt man beide Gleichungen zusammen erhält man

$$g(G) \geq f(G') = g(G')$$

was ein Widerspruch zu unserer Annahme ist. ∎

Für die Vollständigkeit müssen wir 2 Fälle ausschliessen:

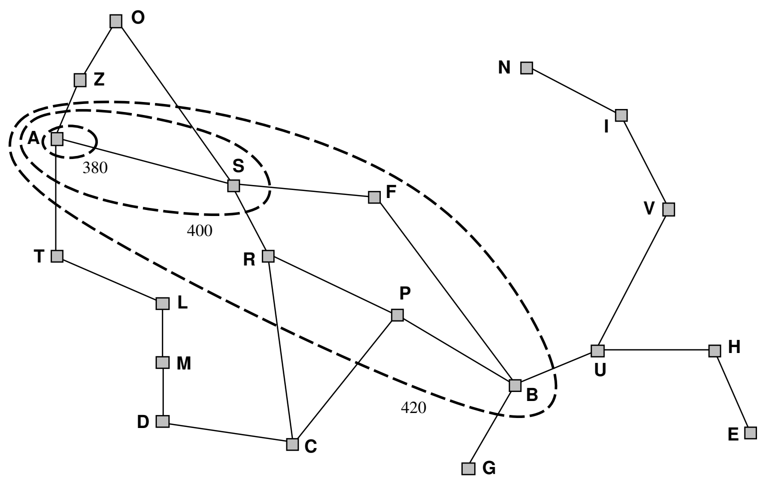



Abbildung 4.1: Der A$^*$-Algorithmus

1. Es werden auf einen Schlag unendlich viele Knoten erzeugt.
2. Es werden Knoten erzeugt deren Evaluierungsfunktion zwar wächst, aber unter einem bestimmten Wert ($g(Ziel)$) bleibt.

**Theorem 4.1.4 (Vollständigkeit von A$^*$)**
*A$^*$ ist vollständig für lokal endliche Graphen mit: $\exists \delta > 0$ : jeder Operator erhöht $g(n)$ um mindestens $\delta$.*

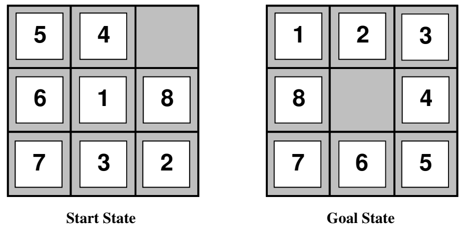
A$^*$ zieht Konturen die zielgerichtet sind. Was macht Uniform-Cost? Uniform-Cost zieht konzentrische Kreise!

Die Anzahl der Knoten im Speicher ist exponentiell in der Länge der Lösung. Auch bei der A$^*$-Suche ist der Speicher das Problem, nicht die Zeit.

Dies führt zu

- **IDA$^*$:** A$^*$ + Iterative Deepening,
- **SMA$^*$:** Simplified Memory Bounded A$^*$ ($\rightsquigarrow$ Abschnitt 4.3)

## 4.2 Beispiele für Heuristiken



Was ist eigentlich der Verzweigungsfaktor in diesem Problem? Ungefähr 3. Wieviele Zustände müssen schlimmstenfalls betrachtet werden? Ungefähr

$$3^g \approx 10^{10},$$

wobei $g$ die Anzahl der Züge ist, um zu einer Lösung zu gelangen. $g$ ist ungefähr 20. Aber es gibt ja nur $9! \approx 10^5$ Zustände! Mit anderen Worten: Testen auf Schleifen hilft sehr viel.

Welche Heuristikfunktionen bieten sich an?

**Hamming-Distanz:** $h_1$ ist Anzahl der Zahlen in falscher Position: $h_1(Anfang) = 8$.

**Manhattan-Distanz:** Wir rechnen für jedes Teil die Entfernung zur richtigen Position aus und summieren diese:

$$h_2 := \sum_{i=1}^{8} \text{Entfernung von } i \text{ zur richtigen Position}$$

Z.B. $h_2(Anfang) = 2 + 3 + 2 + 1 + 2 + 2 + 1 + 2 = 15$.

Strenggenommen gibt es keinen einheitlichen Verzweigungsfaktor. 1 Zustand hat VF 4 (das Kästchen in der Mitte), die 4 Eckzustände haben VF 2, und die 4 übrigen Zustände haben 3.

Wie kann man die Qualität einer Heuristik messen? Möglichst in einem einzigen Wert!

**Definition 4.2.1 (Effektiver Verzweigungsfaktor $b^*$)**
*Angenommen, $A^*$ findet die optimale Lösung für eine Problem Instanz in der Tiefe $d$ und musste $N$ Knoten expandieren. Dann heisst $b^*$ definiert durch*

$$N = 1 + b^* + (b^*)^2) + \ldots + (b^*)^d)$$

*der effektive Verzweigungsfaktor von $A^*$.*

Man beachte, daß $b^*$ nicht nur von $h$ sondern auch von der speziellen Problem-Instanz abhängt. Allerdings stellt man fest, daß $b^*$ ziemlich konstant für grosse Klassen von Problem-Instanzen ist.

| | Search Cost | | | Effective Branching Factor | | |
|---|---|---|---|---|---|---|
| $d$ | IDS | A*($h_1$) | A*($h_2$) | IDS | A*($h_1$) | A*($h_2$) |
| 2 | 10 | 6 | 6 | 2.45 | 1.79 | 1.79 |
| 4 | 112 | 13 | 12 | 2.87 | 1.48 | 1.45 |
| 6 | 680 | 20 | 18 | 2.73 | 1.34 | 1.30 |
| 8 | 6384 | 39 | 25 | 2.80 | 1.33 | 1.24 |
| 10 | 47127 | 93 | 39 | 2.79 | 1.38 | 1.22 |
| 12 | 364404 | 227 | 73 | 2.78 | 1.42 | 1.24 |
| 14 | 3473941 | 539 | 113 | 2.83 | 1.44 | 1.23 |
| 16 | – | 1301 | 211 | – | 1.45 | 1.25 |
| 18 | – | 3056 | 363 | – | 1.46 | 1.26 |
| 20 | – | 7276 | 676 | – | 1.47 | 1.27 |
| 22 | – | 18094 | 1219 | – | 1.48 | 1.28 |
| 24 | – | 39135 | 1641 | – | 1.48 | 1.26 |

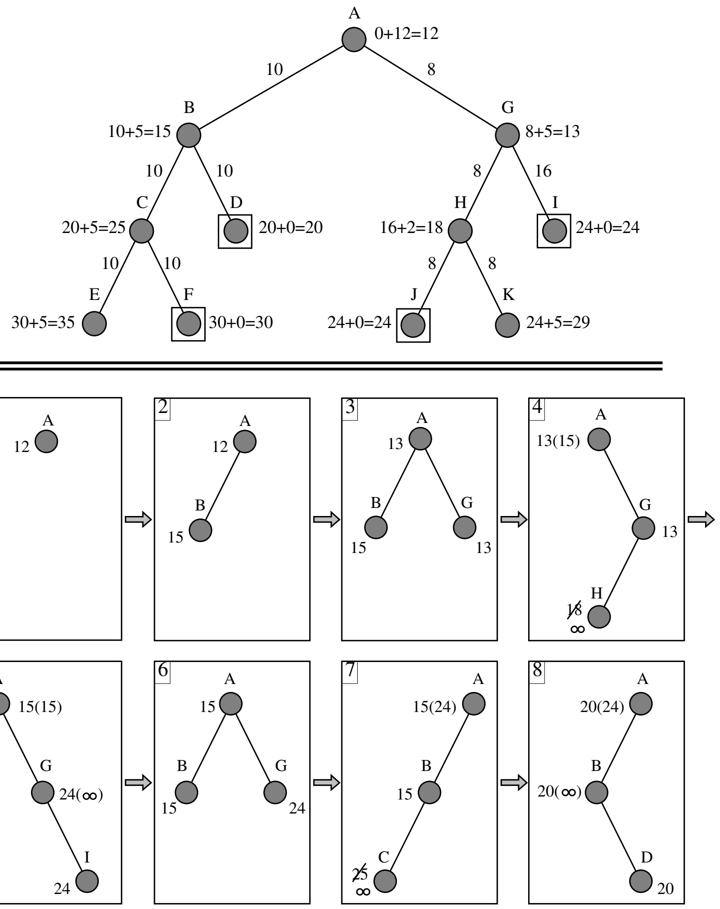
## 4.3 SMA$^*$-Suche

A$^*$ braucht viel Speicher. Was, wenn nur beschränkter Speicher vorhanden ist? Wenn also die Queue nur eine maximale Größe haben kann?

SMA$^*$ ist eine Erweiterung von A$^*$ die mit beschränktem Speicher auskommt. Falls kein Platz mehr da ist, aber Knoten expandiert werden müssen, werden Knoten aus der Queue gestrichen:

- und zwar solche mit möglichst großen $f$-Kosten (*forgotten nodes*). Deren $f$-Kosten merkt man sich aber. Später werden diese Knoten nur dann wieder betrachtet, wenn alle anderen Pfade zu höheren Kosten führen.

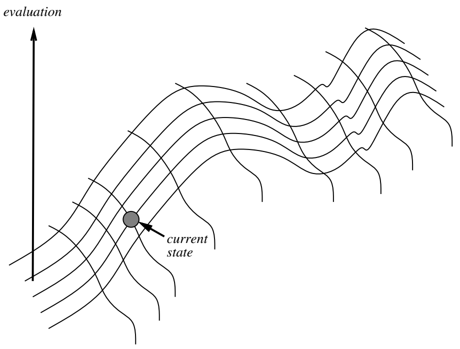
**Theorem 4.3.1 (Eigenschaften von SMA$^*$)**
*SMA$^*$ ist vollständig, falls genügend Speicher da ist um den* **kürzesten** *Lösungspfad zu speichern. SMA$^*$ ist optimal, falls genügend Speicher da ist um den* **optimalen** *Lösungspfad zu speichern.*



```
function SMA*(problem) returns a solution sequence
  inputs: problem, a problem
  static: Queue, a queue of nodes ordered by f-cost

  Queue ← MAKE-QUEUE({MAKE-NODE(INITIAL-STATE[problem])})
  loop do
      if Queue is empty then return failure
      n ← deepest least-f-cost node in Queue
      if GOAL-TEST(n) then return success
      s ← NEXT-SUCCESSOR(n)
      if s is not a goal and is at maximum depth then
          f(s) ← ∞
      else
          f(s) ← MAX(f(n), g(s)+h(s))
      if all of n's successors have been generated then
          update n's f-cost and those of its ancestors if necessary
      if SUCCESSORS(n) all in memory then remove n from Queue
      if memory is full then
          delete shallowest, highest-f-cost node in Queue
          remove it from its parent's successor list
          insert its parent on Queue if necessary
      insert s on Queue
  end
```

## 4.4 Iterative Verbesserungen

*Bei manchen Problemen ist nur der jeweilige Zustand wichtig, auf den Pfad der dorthin führt kommt es nicht an.*




Das Problem ist natürlich beliebig schwierig!

**Hill-Climbing:** Gradienten-Abstieg (bzw. Aufstieg). Gehe zufällig in irgendeine Richtung. Vergleiche die neue Evaluierung mit der alten. Gehe vom neuen Punkt aus, falls die Evaluierung besser ist.

**Probleme:**

- Lokale Maxima (verlaufen),
- Plateaux (umherirren),
- Grat (Umwege).

```
function HILL-CLIMBING(problem) returns a solution state
  inputs: problem, a problem
  static: current, a node
          next, a node

  current ← MAKE-NODE(INITIAL-STATE[problem])
  loop do
     next ← a highest-valued successor of current
     if VALUE[next] < VALUE[current] then return current
     current ← next
  end
```

**Simulated Annealing:** "Abkühlung", modifiziertes Hill-Climbing: auch schlechte Züge (Eval. ist klein) werden mit einer gewissen Wahrscheinlichkeit zugelassen, und zwar mit $e^{-\frac{|\Delta Eval|}{T}}$. Diese hängt ab von der "Temperatur" $T$, die von $\infty$ nach $0$ geht.

```
function SIMULATED-ANNEALING(problem, schedule) returns a solution state
  inputs: problem, a problem
          schedule, a mapping from time to "temperature"
  static: current, a node
          next, a node
          T, a "temperature" controlling the probability of downward steps

  current ← MAKE-NODE(INITIAL-STATE[problem])
  for t ← 1 to ∞ do
     T ← schedule[t]
     if T=0 then return current
     next ← a randomly selected successor of current
     ΔE ← VALUE[next] – VALUE[current]
     if ΔE > 0 then current ← next
     else current ← next only with probability e^(ΔE/T)
```

*Man finde den Gipfel des Mount Everest, und zwar bei dichtem Nebel und Gedächtnisverlust!*

## 4.5 Aufgaben

### Aufgabe 4.1 (Heuristische Suche)

*Beim 8-Schiebepuzzle haben Sie zwei Heuristiken kennengelernt: (1) Hamming-Distanz, d.h. Anzahl der falschen Positionen; (2) Manhattan-Distanz.*

1. *Welche Heuristik ist besser? Oder sind sie gar nicht miteinander vergleichbar?*
2. *Überlegen Sie sich ein formales Kriterium zum Vergleich der Güte von Heuristiken!*

**Aufgabe 4.2 (Rundreisen)**
*Ein Geschäftsmann will eine möglichst kurze Rundreise durch die vier Orte A, B, C und D machen. Beginn und Ende der Reise ist der Ort A. Die Entfernungen seien durch die folgende Matrix gegeben. Der letzte Eintrag in der ersten Zeile besagt z.B., daß die Entfernung von A nach D (in dieser Richtung) 7 Einheiten beträgt.*

| | A | B | C | D |
|---|---|---|---|---|
| A | - | 4 | 5 | 7 |
| B | 3 | - | 4 | 9 |
| C | 5 | 4 | - | 6 |
| D | 6 | 8 | 7 | - |

*Bestimmen Sie die kürzeste Rundreise unter Anwendung des A*-Algorithmus' und zeichnen Sie den Suchbaum! Als heuristische Funktion $h$ verwenden die direkte Entfernung vom erreichten Ort zurück nach A als Abschätzung für den restlichen Weg.*

**Aufgabe 4.3 (Greedy-Algorithmus)**

1. *Wie verhält sich der Greedy-Algorithmus, falls als heuristische Funktion $h(n) = -g(n)$ angesetzt wird? Zur Erinnerung: $g(n)$ sind die Kosten des Pfades bis zum Knoten $n$.*

2. *Beim Beispiel aus der Vorlesung (Reise durch Rumänien) läuft der Greedy-Algorithmus mit Luftlinien-Heuristk in die Irre beim Suchen eines Weges von Iasi nach Fagaras. In Gegenrichtung passiert das jedoch nicht. Gibt es Fälle, wo dies in beide Richtungen geschehen kann?*

**Aufgabe 4.4 (SMA*)**

1. *SMA* sieht von Pfaden ab, die allein den ganzen verfügbaren Speicher für sich beanspruchen, aber keinen Zielzustand erreichen. Zeigen Sie, daß das Verfahren ohne diesen Test in eine Endlosschleife gerät, falls nicht genügend Speicher da ist, um den kürzesten Lösungspfad abzulegen.*

2. *Zeigen Sie, daß kein Baum zweimal generiert wird im Verlaufe der Suche. Tip: Der Beweis kann per Induktion über die Baumtiefe geführt werden, beginnend mit der maximalen Tiefe.*

**Aufgabe 4.5 (Tiefensuche versus Iterative-Deepening)**
*Betrachten Sie exponentielle Suchbäume mit einem Verzweigungsfaktor $b > 1$ und einer Tiefe $d \in \mathbb{N}$.*

1. *Berechnen Sie, wie viele Knoten (i) bei Tiefensuche und (ii) bei Anwendung von Iterative-Deepening expandiert werden!*

2. *Zeigen Sie, daß für $d \to \infty$ der Quotient von (ii) und (i) gegen $b/(b-1)$ geht!*

# 5 Wissensrepräsentation I

## 5.1 Aussagenlogik (AL)

**Definition 5.1.1 (Aussagenlogik $\mathcal{L}_{AL}$, Sprache $\mathcal{L} \subseteq \mathcal{L}_{AL}$)**
*Die Sprache $\mathcal{L}_{AL}$ der Aussagenlogik besteht aus*

- $\Box$ *und* $\top$*: dem falsum und dem verum,*
- $p, q, r, x_1, x_2, \ldots x_n, \ldots$*: einer abzählbaren Menge* $\mathcal{AT}$ *von AL- Konstanten,*
- $\neg$, $\wedge$, $\vee$, $\rightarrow$*: den Junktoren (*$\neg$ *ist einstellig, alle anderen zweistellig),*
- $($, $)$*: den Klammern, die der eindeutigen Lesbarkeit dienen.*

*Üblicherweise betrachten wir nur eine endliche Menge von aussagenlogischen Konstanten. Allgemein legt eine Teilmenge von* $\mathcal{AT}$ *eine Sprache* $\mathcal{L} \subseteq \mathcal{L}_{AL}$ *fest. Daraus ergibt sich induktiv in natürlicher Weise die Menge aller* $\mathcal{L}$*-Formeln* $Fml_{\mathcal{L}}$*.*

**Definition 5.1.2 (Semantik, Bewertung, Modell)**
*Eine Bewertung* $v$ *für eine Sprache* $\mathcal{L} \subseteq \mathcal{L}_{AL}$ *ist eine Abbildung auf der Menge der durch* $\mathcal{L}$ *festgelegten aussagenlogischen Konstanten und* $\{\top, \Box\}$ *in die Menge* $\{\mathbf{t}, \mathbf{f}\}$ *mit* $v(\Box) = \mathbf{f}$, $v(\top) = \mathbf{t}$.

*Jede Bewertung* $v$ *kann zu einer eindeutigen Funktion* $\bar{v} : Fml_{\mathcal{L}} \rightarrow \{\mathbf{t}, \mathbf{f}\}$ *so fortgesetzt werden, daß gilt:*

- $\bar{v}(\neg p) = \begin{cases} \mathbf{t}, & \textit{falls } \bar{v}(p) = \mathbf{f}, \\ \mathbf{f}, & \textit{falls } \bar{v}(p) = \mathbf{t}. \end{cases}$
- $\bar{v}(\varphi \wedge \gamma) = \begin{cases} \mathbf{t}, & \textit{falls } \bar{v}(\varphi) = \mathbf{t} \textit{ und } \bar{v}(\gamma) = \mathbf{t}, \\ \mathbf{f}, & \textit{sonst.} \end{cases}$
- $\bar{v}(\varphi \vee \gamma) = \begin{cases} \mathbf{t}, & \textit{falls } \bar{v}(\varphi) = \mathbf{t} \textit{ oder } \bar{v}(\gamma) = \mathbf{t}, \\ \mathbf{f}, & \textit{sonst.} \end{cases}$
- $\bar{v}(\varphi \rightarrow \gamma) = \begin{cases} \mathbf{t}, & \textit{falls } \bar{v}(\varphi) = \mathbf{f} \textit{ oder } (\bar{v}(\varphi) = \mathbf{t} \textit{ und } \bar{v}(\gamma) = \mathbf{t}), \\ \mathbf{f}, & \textit{sonst.} \end{cases}$

*Eine Bewertung* $v$ *legt daher in eindeutiger Weise ein* $\bar{v}$ *fest. Wir nennen* $\bar{v}$ *auch* $\mathcal{L}$*-Struktur oder Modell* $\mathcal{A}_v$*. Ein Modell legt also für jede Formel fest, ob sie wahr ist oder falsch. Im folgenden werden wir sowohl von Modellen als auch von Bewertungen sprechen.*

**Definition 5.1.3 (Gültigkeit einer Formel, Tautologie)**

1. *Eine Formel $\varphi \in Fml_{\mathcal{L}}$ gilt unter der Belegung $v$, falls $\bar{v}(\varphi) = \mathbf{t}$. Wir schreiben auch $\bar{v} \models \varphi$ oder einfach $v \models \varphi$.*
2. *Eine Theorie ist eine Menge von Formeln : $T \subseteq Fml_{\mathcal{L}}$. $v$ erfüllt $T$ falls $\bar{v}(\varphi) = \mathbf{t}$ für alle $\varphi \subseteq T$. Wir schreiben $v \models T$.*
3. *Eine $\mathcal{L}$-Formel $\varphi$ heißt $\mathcal{L}$-Tautologie, falls für alle möglichen Bewertungen $v$ auf $\mathcal{L}$ gilt: $v \models \varphi$.*

Im folgenden lassen wir die zugrundeliegende Sprache $\mathcal{L}$ meistens weg, da sie aus dem Zusammenhang klar ersichtlich ist. Trotzdem muß sie natürlich immer festgelegt sein.

**Definition 5.1.4 (Folgerungsmenge $Cn(T)$)**
*Eine Formel $\varphi$ folgt aus $T$, falls für alle Bewertungen $v$ mit $v \models T$ auch $v \models \varphi$ gilt. Dafür schreiben wir: $T \models \varphi$.*

*Wir bezeichnen mit $Cn_{\mathcal{L}}(T) :=_{def} \{\varphi \in Fml_{\mathcal{L}} : T \models \varphi\}$, oder einfach $Cn(T)$, den semantischen Konsequenzoperator.*

**Lemma 5.1.5 (Eigenschaften von $Cn(T)$)**
*Der semantische Konsequenzoperator hat folgende Eigenschaften:*

1. $T$-Erweiterung: $T \subseteq Cn(T)$,
2. Monotonie: $T \subseteq T' \Rightarrow Cn(T) \subseteq Cn(T')$,
3. Abgeschlossenheit: $Cn(Cn(T)) = Cn(T)$.

**Lemma 5.1.6 ($\varphi \notin$ Cn(T))**
$\varphi \notin Cn(T)$ $\underline{\text{gdw}}$ *es gibt eine Bewertung $v$ mit $v \models T$ und $\bar{v}(\varphi) = \mathbf{f}$.*

**Definition 5.1.7 (MOD(T), $Cn(\mathcal{U})$)**
*Ist $T \subseteq Fml_{\mathcal{L}}$, so bezeichnen wir mit MOD$(T)$ die Menge aller $\mathcal{L}$-Strukturen $\mathcal{A}$ die Modelle von $T$ sind:*

$$\mathit{MOD}(T) :=_{def} \{\mathcal{A} : \mathcal{A} \models T\}.$$

*Ist $\mathcal{U}$ eine Menge von Modellen (also Bewertungen), so können wir alle jene Sätze betrachten, die in allen Modellen gelten. Diese Menge bezeichnen wir ebenfalls mit $Cn(\mathcal{U})$:*

$$Cn(\mathcal{U}) :=_{def} \{\varphi \in Fml_{\mathcal{L}} : \forall v \in \mathcal{U} : \bar{v}(\varphi) = \mathbf{t}\}.$$

Offenbar ist MOD der zu $Cn$ duale Begriff:

$$Cn(\mathrm{MOD}(T)) = Cn(T), \;\; \mathrm{MOD}(Cn(T)) = \mathrm{MOD}(T).$$

**Definition 5.1.8 (Vollständigkeit einer Theorie $T$)**
*$T$ heißt vollständig, falls für jede Formel $\varphi \in Fml$: $T \models \varphi$ oder $T \models \neg\varphi$.*

Diese letzte Bedingung ist nicht zu verwechseln mit der entsprechenden Eigenschaft einer Bewertung $v$ bzw. in einem Modell: Jede Bewertung hat natürlich diese Eigenschaft.

**Definition 5.1.9 (Konsistenz einer Theorie)**
*$T$ heißt konsistent, falls es eine Bewertung $v$ mit $\bar{v}(\varphi) = \mathbf{t}$ für alle $\varphi \in T$ gibt.*

**Lemma 5.1.10 (Ex Falso Quodlibet)**
*$T$ ist konsistent* $\underline{\text{gdw}}$ *$Cn(T) \neq Fml_{\mathcal{L}}$.*

## 5.2 Kalküle für die AL

**Definition 5.2.1 (Axiom, Inferenzregel)**
*Axiome der Aussagenlogik sind die folgenden Formeln:*

- $p \to \top$, $\Box \to p$, $\neg\top \to \Box$, $\Box \to \neg\top$,
- $(p \wedge q) \to p$, $(p \wedge q) \to q$,
- $p \to (p \vee q)$, $q \to (p \vee q)$,
- $\neg\neg p \to p$, $(p \to q) \to ((p \to \neg q) \to \neg p)$,
- $p \to (q \to p)$, $p \to (q \to (p \wedge q))$.

*Einzige Inferenzregel der Aussagenlogik ist die folgende Abbildung Modus Ponens*

$$MP : Fml \times Fml \to Fml : (\varphi, \varphi \to \psi) \mapsto \psi.$$

*Wir schreiben dafür auch kurz*

$$\textbf{(MP)}\ \frac{\varphi,\ \varphi \to \psi}{\psi}.$$

*Die Inferenzregel ist so zu verstehen, daß $\varphi, \psi$ beliebige zusammengesetzte Formeln sein können.*

**Definition 5.2.2 (Beweis)**
*Ein Beweis einer Formel $\varphi$ aus einer Theorie $T \subseteq Fml_{\mathcal{L}}$ ist eine Folge $\varphi_1, \ldots, \varphi_n$ von Formeln so, daß $\varphi_n = \varphi$ und für alle $i$ mit $1 \leq i \leq n$ gilt eine der folgenden Bedingungen:*

- *$\varphi_i$ ist Substitutions-Instanz eines Axioms,*
- *$\varphi_i \in T$,*
- *Es gibt $\varphi_l, \varphi_k = (\varphi_l \to \varphi_i)$ mit $l, k < i$. $\varphi_i$ ist also das Resultat der Anwendung des Modus Ponens auf Vorgänger-Formeln von $\varphi_i$.*

*Wir schreiben: $T \vdash \varphi$.*

**Theorem 5.2.3 (Vollständigkeit)**
*Eine Formel folgt semantisch aus einer Theorie $T$ genau dann, wenn sie abgeleitet werden kann:*

$$T \vdash \varphi \ \underline{\text{gdw}} \ T \models \varphi$$

**Theorem 5.2.4 (Kompaktheit)**
*Eine Formel folgt aus einer Theorie $T$ genau dann, wenn sie schon aus einer endlichen Teilmenge von $T$ folgt:*

$$Cn(T) = \bigcup\{Cn(T') : T' \subseteq T, T' \textit{ endlich}\}.$$

Obwohl wir in der Aussagenlogik mit den obengenannten Axiomen und dem Modus Ponens auskommen, ist es sinnvoll, allgemeinere Systeme zu betrachten. Dazu führen wir den folgenden Begriff ein:

**Definition 5.2.5 (Regelsystem MP + D)**
*Sei D eine Menge von allgemeinen Inferenzregeln, d.h. Abbildungen, die einer endlichen Menge von Formeln $\varphi_1, \varphi_2, \ldots, \varphi_n$ eine Formel $\psi$ zuordnen. Wir schreiben dies als*

$$\frac{\varphi_1, \varphi_2, \ldots, \varphi_n}{\psi}.$$

*Unter MP + D verstehen wir das Regelsystem, das entsteht, wenn die Regeln aus D zur Regel Modus Ponens hinzugenommen werden.*

*Für $W \subseteq Fml$ sei im folgenden*

$$Cn^D(W)$$

*die Menge aller Formeln $\varphi$, die man aus $W$ und den Inferenzregeln aus MP + D (im Sinne der Definition 5.12, nun dürfen aber zusätzlich die Regeln aus D angewandt werden) ableiten kann.*

Man beachte, daß $Cn(W)$ semantisch, $Cn^D(W)$ aber syntaktisch (über den Begriff des Beweises) definiert sind. Im Spezialfall $D = \emptyset$ stimmen beide Mengen nach dem Vollständigkeitssatz überein.

**Lemma 5.2.6 (Eigenschaften von $Cn^D$)**
*Sei D eine Menge von allgemeinen Inferenzregeln und $W \subseteq Fml$. Dann gilt:*

1. $Cn(W) \subseteq Cn^D(W)$.
2. $Cn^D(Cn^D(W)) = Cn^D(W)$.
3. *$Cn^D(W)$ ist die kleinste unter D abgeschlossene Menge die W enthält.*

Was ist der Unterschied zwischen einer Inferenzregel $\frac{\varphi}{\psi}$ und der Implikation $\varphi \to \psi$? Angenommen, wir haben eine Formelmenge $T$ gegeben und wir wählen zwei Konstanten $p, q \in \mathcal{L}$. Man könnte doch einerseits

**(1)** $T$ zusammen mit MP und $\{\frac{p}{q}\}$

und andererseits

**(2)** $T \cup \{p \rightarrow q\}$ zusammen mit MP

betrachten. Im ersten Fall bedeutet dies

$$Cn^{\{\frac{p}{q}\}}(T)$$

und im zweiten Fall

$$Cn(T \cup \{p \rightarrow q\}).$$

## Ein Resolutionskalkül für AL

Gegeben sei eine Menge $M$ von Klauseln der Form

$$A \vee \neg B \vee C \vee \ldots \vee \neg E$$

Solch eine Klausel schreiben wir auch als Menge

$$\{A, \neg B, C, \ldots, \neg E\}.$$

Wir definieren die folgende Inferenzregel:

**Definition 5.2.7 (AL-Resolution)**
*Leite aus $C_1 \cup \{A\}$ und $C_2 \cup \{\neg A\}$ die Klausel $C_1 \cup C_2$ ab.*

Ist dieser Kalkül korrekt und vollständig?
Jedes Problem der Form "$T \vdash \phi''$ ist äquivalent zu

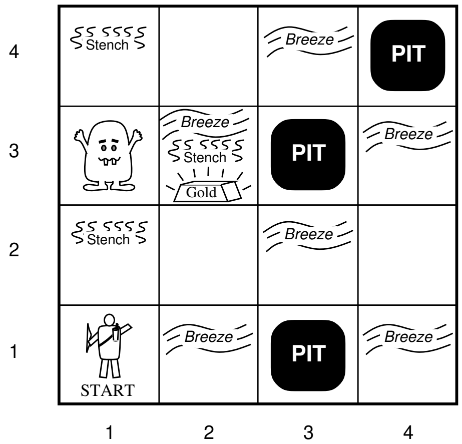
$$T \cup \{\neg\phi\} \text{ ist unerfüllbar"}$$

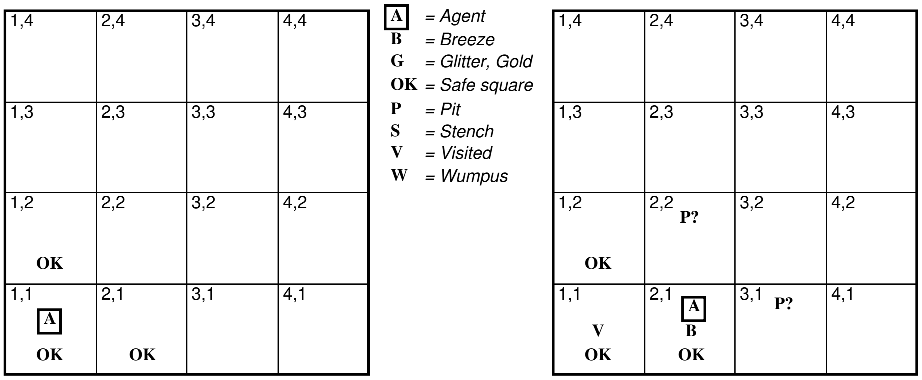
bzw. zu

$$T \cup \{\neg\phi\} \vdash \Box.$$

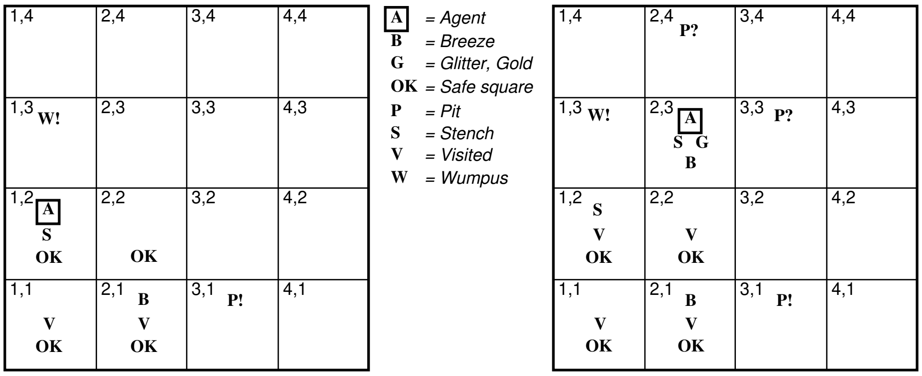
**Theorem 5.2.8 (Widerlegungsvollständigkeit)**
*Ist $M$ eine unerfüllbare Menge von Klauseln, so kann aus $M$ allein mit der Resolutionsregel die leere Klausel $\Box$ abgeleitet werden.*

## 5.3 Die Wumpus-Welt in AL




(a) (b)




(a) (b)

*Festlegung der Sprache:*

| | |
|---|---|
| $S_{i,j}$ | stinkt |
| $B_{i,j}$ | ist kalt |
| $Grube_{i,j}$ | ist eine Grube |
| $Gl_{i,j}$ | glitzert |
| $W_{i,j}$ | enthält Wumpus |

*Allgemeines Wissen:*

$$\begin{array}{lcl} \neg S_{1,1} & \longrightarrow & (\neg W_{1,1} \wedge \neg W_{1,2} \wedge \neg W_{2,1}) \\ \neg S_{2,1} & \longrightarrow & (\neg W_{1,1} \wedge \neg W_{2,1} \wedge \neg W_{2,2} \wedge \neg W_{3,1}) \\ \neg S_{1,2} & \longrightarrow & (\neg W_{1,1} \wedge \neg W_{1,2} \wedge \neg W_{2,2} \wedge \neg W_{1,3}) \\ & & \\ S_{1,2} & \longrightarrow & (W_{1,3} \wedge W_{1,2} \wedge W_{2,2} \wedge W_{1,1}) \end{array}$$

*Wissen nach dem 3. Zug:*

$$\neg S_{1,1} \wedge \neg S_{2,1} \wedge S_{1,2} \wedge \neg B_{1,1} \wedge \neg B_{2,1} \wedge \neg B_{1,2}$$

*Können wir herleiten, dass der Wumpus in (1,3) sitzt?* Ja, etwa mit Resolution oder unserem Hilbert-Kalkül.

Eigentlich wollen wir mehr: zu gegebener Situation möchten wir gerne die *beste Aktion* auswählen. Das aber geht in AL nicht. Jedoch können wir für jede Aktion prüfen, ob sie ausgeführt werden sollte oder nicht. Dazu benötigen wir zusätzliche Axiome:

*Zusätzliche Axiome:*

$$\begin{array}{lcl} A_{1,1} \wedge East \wedge W_{2,1} & \longrightarrow & \neg Forward \\ A_{1,1} \wedge East \wedge Grube_{2,1} & \longrightarrow & \neg Forward \\ A_{i,j} \wedge Gl_{i,j} & \longrightarrow & Take_{Gold} \end{array}$$

**Nachteile:**

- Aktionen können nur geraten werden,
- Datenbasis muss fortlaufend geändert werden,
- Regelmenge werden sehr gross, da keine Variablen vorhanden.

Bei geeigneter Formalisierung (zusätzliche Axiome) kann man jedenfalls prüfen ob

$$\text{KB} \vdash \neg \text{ aktion bzw. KB} \vdash \text{aktion}$$

Es kann auch sein, dass weder das Eine noch das Andere ableitbar ist.

## 5.4 Das P=NP Problem

*Beobachtung: Polynomiale Algorithmen sind schnell, exponentielle dagegen langsam.*

Um eine AL-Formel mit der Wahrheitstafel-Methode als unerfüllbar nachzuweisen, braucht man im schlimmsten Fall $2^n$ Belegungen (bei $n$-Konstanten). *Kann man dieses Problem vielleicht mit anderen Methoden in polynomialer Zeit lösen?*

Wir betrachten Ja-Nein Probleme. Für jede Instanz $\mathcal{I}$ eines Problems können wir fragen ob $\mathcal{I}$ die entsprechende Eigenschaft hat oder nicht: $\mathcal{I}$ ist Ja oder Nein-Instanz.

**Definition 5.4.1 (Klasse P)**
*Ein Problem liegt in der Klasse* **P** *der in polynomialer deterministischer Zeit erkennbaren Probleme, falls es einen Algorithmus* Alg. *und ein Polynom $p(n)$ gibt, so dass für jede Instanz $\mathcal{I}$ gilt:*

| | | |
|---|---|---|
| *$\mathcal{I}$ ist Ja-Instanz* | *gdw* | Alg. *antwortet "Ja" auf Eingabe von*<br>*$\mathcal{I}$ nach höchstens $p(\text{länge}(\mathcal{I}))$-Schritten.* |

**Definition 5.4.2 (Klasse NP)**
*Ein Problem liegt in der Klasse* **NP** *der in polynomialer nicht-deterministischer Zeit erkennbaren Probleme, falls es einen Algorithmus* Check *und ein Polynom $p(n)$ gibt so dass für jede Instanz $\mathcal{I}$ gilt:*

| | | |
|---|---|---|
| *$\mathcal{I}$ ist Ja-Instanz* | *gdw* | *es gibt eine Instanz $\mathcal{I}'$ eines anderen Problems mit*<br>*(1) $\text{länge}(\mathcal{I}') < p(\text{länge}(\mathcal{I}))$,*<br>*(2)* Check *antwortet "Ja" auf Eingabe von $\mathcal{I}'$*<br>*nach höchstens $p(\text{länge}(\mathcal{I}))$-Schritten.* |

Man sieht leicht, dass die Erfüllbarkeit einer AL-Formel in **NP** liegt, und dass **NP** jedenfalls **P** umfasst. Was sind die schwierigsten Probleme in **NP**?

Offenbar sind Probleme die man polynomial aufeinander reduzieren kann gleichwertig. Ein Problem $\text{Prob}_1$ ist polynomial transformierbar zu einem Problem $\text{Prob}_2$, falls man für jede Instanz $\mathcal{I}_1$ von $\text{Prob}_1$ in polynomialer Zeit eine Instanz $\mathcal{I}_2$ von $\text{Prob}_2$ berechnen können, so dass gilt:

$\mathcal{I}_1$ ist Ja-Instanz von $\text{Prob}_1$ **genau dann wenn** $\mathcal{I}_2$ ist Ja-Instanz von $\text{Prob}_2$

**Definition 5.4.3 (NP-Vollständigkeit)**
*Ein Problem aus* **NP** *heisst NP-vollständig, wenn sich alle anderen Probleme aus* **NP** *in polynomialer Weise auf es reduzieren lassen.*

Folgende Probleme sind NP-vollständig:

- Gegeben ein endlicher Graph und $k \in \mathbb{N}$. Kann der Graph mit $k$-Farben gefärbt werden?
- Gegeben ein endlicher Graph. Gibt es einen Hamilton-Kreis?
- Gegeben eine AL-Formel $\phi$. Ist $\phi$ erfüllbar?

- Gegeben eine Menge von Klauseln in den Konstanten $c_1, \ldots, c_n$. Ist die Menge erfüllbar?
- Gegeben eine Menge von Klauseln in den Konstanten $c_1, \ldots, c_n$. Jede Klausel enthält höchstens 3 Literale. Ist die Menge erfüllbar?

## 5.5 Prädikatenlogik erster Stufe (PL1)

**Definition 5.5.1 (Prädikatenlogik 1. Stufe $\mathcal{L}_{PL1}$, Sprache $\mathcal{L} \subseteq \mathcal{L}_{PL1}$)**
*Die Sprache $\mathcal{L}_{PL1}$ der Prädikatenlogik erster Stufe besteht aus*

- $x, y, z, x_1, x_2, \ldots, x_n, \ldots$*: einer abzählbaren Menge* $\mathcal{VAR}$ *von Variablen,*
- *für jedes* $k \in \mathbb{N}$*:* $P_1^k, P_2^k, \ldots, P_n^k, \ldots$ *einer abzählbaren Menge* $\mathcal{PRED}^k$ *von k-stelligen Prädikatssymbolen*[1]*,*
- *für jedes* $k \in \mathbb{N}$*:* $f_1^k, f_2^k, \ldots, f_n^k, \ldots$ *einer abzählbaren Menge* $\mathcal{FUNCT}^k$ *von k-stelligen Funktionssymbolen,*
- $\neg$, $\wedge$, $\vee$*: den Junktoren (*$\neg$ *ist einstellig, alle anderen zweistellig),*
- (, )*: den Klammern,*
- $\forall$, $\exists$*: den Quantoren.*

*0-stellige Funktionssymbole heißen Individuen-Konstanten, wobei wir die Klammern weglassen. Im allgemeinen werden wir, wie in der Aussagenlogik auch, nur eine gewisse Teilmenge der Prädikats- bzw. Funktionssymbole benötigen. Diese legen eine Sprache* $\mathcal{L} \subseteq \mathcal{L}_{PL1}$ *fest (analog zu Definition 5.1.1). Die zugrundeliegende Menge der Funktions- und Prädikatssymbole bezeichnet man auch als Signatur* $\Sigma$*. Daraus ergeben sich induktiv der Begriff des* $\mathcal{L}$*-Termes* $t$ *und der* $\mathcal{L}$*-Formel* $\varphi$*:*

**Term:** $\mathcal{L}$*-Terme* $t$ *sind induktiv wie folgt definiert:*

1. *Jede Variable ist ein* $\mathcal{L}$*-Term.*
2. *Falls* $f^k$ *ein Funktionssymbol mit Stelligkeit* $k$ *aus* $\mathcal{L}$ *ist und* $t_1, \ldots, t_k$ $\mathcal{L}$*-Terme, so ist auch* $f^k(t_1, \ldots, t_k)$ *ein* $\mathcal{L}$*-Term.*

*Die Menge aller* $\mathcal{L}$*-Terme die man mit einer Menge* $X \subseteq \mathcal{VAR}$ *bilden kann bezeichnen wir mit* $Term_{\mathcal{L}}(X)$ *oder mit* $Term_{\Sigma}(X)$*. Für* $X = \emptyset$ *erhalten wir die Menge der Grundterme* $Term_{\mathcal{L}}(\emptyset)$ *oder kurz* $Term_{\mathcal{L}}$*.*

**Formel:** $\mathcal{L}$*-Formeln* $\varphi$ *sind ebenfalls induktiv definiert:*

1. *Falls* $P^k$ *ein Prädikatssymbol mit Stelligkeit* $k$ *aus* $\mathcal{L}$ *ist und* $t_1, \ldots, t_k$ $\mathcal{L}$*-Terme, dann ist* $P^k(t_1, \ldots, t_k)$ *eine* $\mathcal{L}$*-Formel.*
2. *Für jede* $\mathcal{L}$*-Formel* $\varphi$ *ist auch* $(\neg\varphi)$ *eine* $\mathcal{L}$*-Formel.*

[1]Die 0-stelligen Prädikatssymbole sind gerade die aussagenlogischen Konstanten aus $\mathcal{AT}$ von $\mathcal{L}_{AL}$. Wir setzen voraus, daß $\Box$ und $\top$ vorhanden sind.

3. *Für alle $\mathcal{L}$-Formeln $\varphi$ und $\psi$ sind auch $(\varphi \wedge \psi)$ und $(\varphi \vee \psi)$ $\mathcal{L}$-Formeln.*
4. *Falls $x$ eine Variable und $\varphi$ eine $\mathcal{L}$-Formel, so sind auch $(\exists\, x\, \varphi)$ und $(\forall\, x\, \varphi)$ $\mathcal{L}$-Formeln.*

*Atomare $\mathcal{L}$-Formeln sind die, die gemäß 1. aufgebaut sind, wir bezeichnen sie mit $At_{\mathcal{L}}(X)$ ($X \subseteq \mathcal{VAR}$). Die Menge aller $\mathcal{L}$-Formeln bezüglich $X$ bezeichnen wir mit $Fml_{\mathcal{L}}(X)$.*

*Positive Formeln, $Fml^{+}_{\mathcal{L}}(X)$ sind solche, die nur mittels* 1.*,* 3. *und* 4. *gebildet werden.*

*Falls $\varphi$ eine $\mathcal{L}$-Formel ist, die als Teil einer $\mathcal{L}$-Formel $\psi$ auftritt, so heißt $\varphi$ Teilformel von $\psi$.*

## Ein kleines Beispiel zur Illustration

**Beispiel 5.5.2 (Von Halbgruppen zu Ringen)**
*Wir betrachten $\mathcal{L} = \{0, 1, +, \cdot, \leq, =\}$, wobei $0, 1$ Konstanten, $+, \cdot$ zweistellige Funktionszeichen und $\leq, =$ zweistellige Relationszeichen sind. Was kann man in dieser Sprache aufschreiben?*

| | | |
|---|---|---|
| *Ax 1:* | $\forall x \forall y \forall z$ | $x + (y + z) = (x + y) + z$ |
| *Ax 2:* | $\forall x$ | $x + 0 = 0 + x$ |
| *Ax 3:* | $\forall x \forall y$ | $x + y = 0 \wedge y + x = 0$ |
| *Ax 4:* | $\forall x \forall y$ | $x + y = y + x$ |
| *Ax 5:* | $\forall x \forall y \forall z$ | $x \cdot (y \cdot z) = (x \cdot y) \cdot z$ |
| *Ax 6:* | $\forall x \forall y \forall z$ | $x \cdot (y + z) = x \cdot y + x \cdot z$ |
| *Ax 7:* | $\forall x \forall y \forall z$ | $x \cdot (y + z) \cdot x = y \cdot x + z \cdot x$ |

*Ax 1 beschreibt eine* Halbgruppe*, Ax 1-2 ein* Monoid*, Ax 1-3 eine* Gruppe*, Ax 1-4 eine* abelsche Gruppe*, Ax 1-7 einen* Ring.

**Definition 5.5.3 ($\mathcal{L}$-Struktur $\mathcal{A} = (U_{\mathcal{A}}, I_{\mathcal{A}})$)**
*Eine $\mathcal{L}$-Struktur oder $\mathcal{L}$-Interpretation ist ein Paar $\mathcal{A} :=_{def} (U_{\mathcal{A}}, I_{\mathcal{A}})$, wobei $U_{\mathcal{A}}$ eine beliebige nicht leere Menge ist, die die Grundmenge (oder das Universum bzw. der Individuenbereich) von $\mathcal{A}$ genannt wird. Ferner ist $I_{\mathcal{A}}$ eine Abbildung, die:*

- *jedem $k$-stelligen Prädikatssymbol $P^k$ aus $\mathcal{L}$ ein $k$-stelliges Prädikat über $U_{\mathcal{A}}$ zuordnet,*
- *jedem $k$-stelligen Funktionssymbol $f^k$ aus $\mathcal{L}$ eine $k$-stellige Funktion auf $U_{\mathcal{A}}$ zuordnet.*

*Mit anderen Worten: Der Definitionsbereich von $I_{\mathcal{A}}$ ist gerade die Menge der Prädikats- und Funktionssymbole von $\mathcal{L}$. Der Wertebereich von $I_{\mathcal{A}}$ besteht aus Prädikaten und Funktionen auf $U_{\mathcal{A}}$. Wir schreiben:*

$$I_{\mathcal{A}}(P) = P^{\mathcal{A}},\ I_{\mathcal{A}}(f) = f^{\mathcal{A}}.$$

*Sei $\varphi$ eine $\mathcal{L}_1$-Formel und $\mathcal{A} :=_{def} (U_\mathcal{A}, I_\mathcal{A})$ eine $\mathcal{L}$-Struktur. $\mathcal{A}$ heißt zu $\varphi$ passend, falls $I_A$ für alle in $\varphi$ vorkommenden Prädikatssymbole und Funktionssymbole definiert ist, falls also $\mathcal{L}_1 \subseteq \mathcal{L}$.*

**Definition 5.5.4 (Variablenbelegung $\varrho$)**
*Eine Variablenbelegung $\varrho$ über einer $\mathcal{L}$-Struktur $\mathcal{A} = (U_\mathcal{A}, I_\mathcal{A})$ ist eine Funktion*

$$\varrho : \; \mathcal{VAR} \; \rightarrow \; U_\mathcal{A}; \; x \mapsto \varrho(x).$$

**Definition 5.5.5 (Semantik der Prädikatenlogik)**
*Sei $\varphi$ eine Formel, $\mathcal{A}$ eine zu $\varphi$ passende Struktur und $\varrho$ eine Variablenbelegung über $\mathcal{A}$. Für jeden Term $t$, den man aus den Bestandteilen von $\varphi$ bilden kann, definieren wir nun induktiv den* Wert von $t$ in der Struktur $\mathcal{A}$, *den wir mit $\mathcal{A}(t)$ bezeichnen.*

1. *Für eine Variable $x$ ist $\mathcal{A}(x) :=_{def} \varrho(x)$.*
2. *Falls $t$ die Form hat $t = f^k(t_1, \ldots, t_k)$, wobei $t_1, \ldots, t_k$ Terme und $f^k$ ein $k$-stelliges Funktionssymbol ist, so ist $\mathcal{A}(t) :=_{def} f^\mathcal{A}(\mathcal{A}(t_1), \ldots, \mathcal{A}(t_k))$.*

*Nun definieren wir induktiv den Wahrheitswert einer Formel $\varphi$ in $\mathcal{A}$:*

1. *Falls $\varphi :=_{def} P^k(t_1, \ldots, t_k)$ mit den Termen $t_1, \ldots, t_k$ und $k$-stelligem Prädikatssymbol $P^k$, so ist*
$$\mathcal{A}(\varphi) :=_{def} \begin{cases} \mathbf{t}, & falls\,(\mathcal{A}(t_1), \ldots, \mathcal{A}(t_k)) \in P^\mathcal{A}, \\ \mathbf{f}, & sonst. \end{cases}$$
2. *Falls $\varphi :=_{def} \neg\psi$, so ist*
$$\mathcal{A}(\varphi) :=_{def} \begin{cases} \mathbf{t}, & falls\, \mathcal{A}(\psi) = \mathbf{f}, \\ \mathbf{f}, & sonst. \end{cases}$$
3. *Falls $\varphi :=_{def} (\psi \wedge \eta)$, so ist*
$$\mathcal{A}(\varphi) :=_{def} \begin{cases} \mathbf{t}, & falls\, \mathcal{A}(\psi) = \mathbf{t}\, und\, \mathcal{A}(\eta) = \mathbf{t}, \\ \mathbf{f}, & sonst. \end{cases}$$
4. *Falls $\varphi :=_{def} (\psi \vee \eta)$, so ist*
$$\mathcal{A}(\varphi) :=_{def} \begin{cases} \mathbf{t}, & falls\, \mathcal{A}(\psi) = \mathbf{t}\, oder\, \mathcal{A}(\eta) = \mathbf{t}, \\ \mathbf{f}, & sonst. \end{cases}$$
5. *Falls $\varphi :=_{def} \forall\, x\, \psi$, so ist*
$$\mathcal{A}(\varphi) = \; :=_{def} \begin{cases} \mathbf{t}, & falls\, für\, alle\; d \in U_\mathcal{A} : \mathcal{A}_{[x/d]}(\psi) = \mathbf{t}, \\ \mathbf{f}, & sonst. \end{cases}$$
6. *Falls $\varphi :=_{def} \exists\, x\, \psi$, so ist*
$$\mathcal{A}(\varphi) :=_{def} \begin{cases} \mathbf{t}, & falls\; es\; gibt\; ein\; d \in U_\mathcal{A} : \mathcal{A}_{[x/d]}(\psi) = \mathbf{t}, \\ \mathbf{f}, & sonst. \end{cases}$$

*In den Fällen 5. und 6. wurde die Notation $[x/d]$ benutzt. Sie ist wie folgt definiert: Für $d \in U_{\mathcal{A}}$ sei $\mathcal{A}_{[x/d]}$ die Struktur $\mathcal{A}'$, die überall mit $\mathcal{A}$ identisch ist, bis auf die Definition von $x^{\mathcal{A}'}$: $x^{\mathcal{A}'} :=_{def} d$ (unabhängig davon, ob $I_{\mathcal{A}}$ für $x$ definiert ist oder nicht). Wir schreiben:*

- *$\mathcal{A} \models \varphi[\varrho]$ für $\mathcal{A}(\varphi) = \mathbf{t}$: $\mathcal{A}$ ist Modell für $\varphi$ bezüglich $\varrho$.*
- *Enthält $\varphi$ keine freien Variablen, so ist $\mathcal{A} \models \varphi[\varrho]$ unabhängig von $\varrho$. Wir lassen dann $\varrho$ einfach weg.*
- *Falls es mindestens ein Modell für $\varphi$ gibt, so heißt $\varphi$ erfüllbar oder konsistent.*

**Definition 5.5.6 (Tautologie)**

1. *Eine Theorie ist eine Menge von Formeln ohne freie Variablen: $T \subseteq Fml_{\mathcal{L}}$. $\mathcal{A}$ erfüllt $T$ falls $\mathcal{A} \models \varphi$ für alle $\varphi \in T$. Wir schreiben $\mathcal{A} \models T$.*
2. *Eine $\mathcal{L}$-Formel $\varphi$ heißt $\mathcal{L}$-Tautologie, falls für alle möglichen $\mathcal{L}$-Strukturen $\mathcal{A}$ gilt: $\mathcal{A} \models \varphi$.*

Im folgenden lassen wir die zugrundeliegende Sprache $\mathcal{L}$ meistens weg, da sie aus dem Zusammenhang klar ersichtlich ist. Trotzdem muß sie natürlich immer festgelegt sein.

**Definition 5.5.7 (Folgerungsmenge $Cn(T)$)**
*Eine Formel $\varphi$ folgt aus $T$, falls für alle Strukturen $\mathcal{A}$ mit $\mathcal{A} \models T$ auch $\mathcal{A} \models \varphi$ gilt. Dafür schreiben wir: $T \models \varphi$.*

*Wir bezeichnen mit $Cn_{\mathcal{L}}(T) :=_{def} \{\varphi \in Fml_{\mathcal{L}} : T \models \varphi\}$, oder einfach $Cn(T)$, den semantischen Konsequenzoperator.*

**Lemma 5.5.8 (Eigenschaften von $Cn(T)$)**
*Der semantische Konsequenzoperator hat folgende Eigenschaften:*

1. $T$-Erweiterung: $T \subseteq Cn(T)$,
2. Monotonie: $T \subseteq T' \Rightarrow Cn(T) \subseteq Cn(T')$,
3. Abgeschlossenheit: $Cn(Cn(T)) = Cn(T)$.

**Lemma 5.5.9 ($\varphi \notin$ Cn(T))**
$\varphi \notin Cn(T)$ gdw *es gibt eine Struktur $\mathcal{A}$ mit $\mathcal{A} \models T$ und $\mathcal{A} \models \neg\varphi$.*

**Definition 5.5.10 (MOD(T), $Cn(\mathcal{U})$)**
*Ist $T \subseteq Fml_{\mathcal{L}}$, so bezeichnen wir mit MOD$(T)$ die Menge aller $\mathcal{L}$-Strukturen $\mathcal{A}$ die Modelle von $T$ sind:*

$$\textit{MOD}(T) :=_{def} \{\mathcal{A} : \mathcal{A} \models T\}.$$

*Ist $\mathcal{U}$ eine Menge von Strukturen, so können wir alle Sätze betrachten, die in allen Strukturen gelten. Diese Menge bezeichnen wir ebenfalls mit $Cn(\mathcal{U})$:*

$$Cn(\mathcal{U}) :=_{def} \{\varphi \in Fml_{\mathcal{L}} : \forall \mathcal{A} \in \mathcal{U} : \mathcal{A} \models \varphi\}.$$

Offenbar ist MOD der zu $Cn$ duale Begriff:

$$Cn(\text{MOD}(T)) = Cn(T), \ \ \text{MOD}(Cn(T)) = \text{MOD}(T).$$

**Definition 5.5.11 (Vollständigkeit einer Theorie $T$)**
*$T$ heißt vollständig, falls für jede Formel $\varphi \in Fml_{\mathcal{L}}$: $T \models \varphi$ oder $T \models \neg\varphi$.*

Diese letzte Bedingung ist nicht zu verwechseln mit der entsprechenden Eigenschaft in einer Struktur: Jede Struktur hat natürlich diese Eigenschaft.

**Lemma 5.5.12 (Ex Falso Quodlibet)**
*$T$ ist konsistent* $\underline{\text{gdw}}$ $Cn(T) \neq Fml_{\mathcal{L}}$.

Welche Axiome erfüllen die folgenden Strukturen:

**Beispiel 5.5.13 (Die natürlichen Zahlen in versch. Sprachen)**

- $\mathcal{N}_{Pr} = (\mathbb{N}, 0^{\mathcal{N}}, +^{\mathcal{N}}, =^{\mathcal{N}})$ *(Presburger Arithmetik),*
- $\mathcal{N}_{PA} = (\mathbb{N}, 0^{\mathcal{N}}, +^{\mathcal{N}}, \cdot^{\mathcal{N}}, =^{\mathcal{N}})$ *(Peano Arithmetik),*
- $\mathcal{N}_{PA'} = (\mathbb{N}, 0^{\mathcal{N}}, 1^{\mathcal{N}}, +^{\mathcal{N}}, \cdot^{\mathcal{N}}, =^{\mathcal{N}})$ *(Variante der Peano Arithmetik).*

*Es geht also jedes Mal um die natürlichen Zahlen, aber jeweils in verschiedenen Sprachen $\mathcal{L}_{Pr}$, $\mathcal{L}_{PA}$, $\mathcal{L}_{PA'}$.*

Ist die Sprache grösser, kann man natürlich auch mehr ausdrücken. Ist aber $\mathcal{L}_{PA'}$ echt ausdrucksstärker als $\mathcal{L}_{PA}$?

Nein, denn man kann die $1^{\mathcal{N}}$ durch eine $\mathcal{L}_{PA}$-Formel definieren: es gibt eine $\mathcal{L}_{PA}$-Formel $\phi(x)$ sodass für jede Variablenbelegung $\rho$ gilt

$$\mathcal{N}_{PA'} \models_{\rho} \phi(x) \ \ \text{gdw} \ \ \rho(x) = 1^{\mathbb{N}}$$

Man wähle etwa für $\phi(x)$ die Formel $\forall z(z \cdot x = z)$.

Ist vielleicht $\mathcal{L}_{PA}$ ausdrucksstärker als $\mathcal{L}_{Pr}$, oder kann man die Multiplikation vielleicht auch irgendwie definieren? Später werden wir sehen (in Kapitel 6) dass $\mathcal{L}_{PA}$ tatsächlich echt ausdrucksstärker ist: denn die Menge der in $\mathcal{N}_{Pr}$ gültigen Sätze ist entscheidbar, während die Menge der in $\mathcal{N}_{PA}$ gültigen Sätze noch nicht einmal rekursiv aufzählbar ist.

Wir haben schon einen zur Aussagenlogik $\mathcal{L}_{PL\,1}$ korrekten und vollständigen Kalkül angegeben. Solche Kalküle gibt es auch für die gerade definierte Prädikatenlogik $\mathcal{L}_{PL\,1}$.

**Theorem 5.5.14 (Vollständigkeit der Prädikatenlogik)**
*Eine Formel folgt semantisch aus einer Theorie $T$ genau dann, wenn sie abgeleitet werden kann:*

$$T \vdash \varphi \ \ \underline{\text{gdw}} \ \ T \models \varphi$$

**Theorem 5.5.15 (Kompaktheit der Prädikatenlogik)**
*Eine Formel folgt aus einer Theorie $T$ genau dann, wenn sie schon aus einer endlichen Teilmenge von $T$ folgt:*

$$Cn(T) = \bigcup\{Cn(T') : T' \subseteq T, T' \textit{ endlich}\}.$$

## 5.6 Herbrand's Theorem

Die bisher eingeführte Relation $T \models \phi$ sagt aus, daß jedes Modell von $T$ auch eines von $\phi$ ist. Da es aber sehr viele Modelle mit sehr grossen Universen gibt, stellt sich die Frage, ob man das nicht einschränken kann.

**Theorem 5.6.1 (Löwenheim-Skolem)**
*$T \models \phi$ gilt genau dann, wenn $\phi$ in allen abzählbaren Modellen von $T$ gilt.*

Oftmals bestehen die Universen der Modelle die uns eigentlich interessieren genau aus den Grundtermen $Term_{\mathcal{L}}(\emptyset)$. Dies führt uns zu dem folgenden Begriff:

**Definition 5.6.2 (Herbrand-Modell)**
*Ein Modell $\mathcal{A}$ heißt Herbrand-Modell bezüglich einer Sprache $\mathcal{L}$, falls das Universum von $\mathcal{A}$ genau aus $Term_{\mathcal{L}}(\emptyset)$ besteht und die Funktionssymbole $f_i^k$ wie folgt interpretiert werden:*

$$f_i^{k\,\mathcal{A}} : Term_{\mathcal{L}}(\emptyset) \times \ldots \times Term_{\mathcal{L}}(\emptyset) \; \rightarrow Term_{\mathcal{L}}(\emptyset);\; (t_1, \ldots, t_k) \mapsto f_i^k(t_1, \ldots, t_k)$$

*Wir schreiben $T \models_{Herb} \phi$ falls jedes Herbrand Modell von $T$ auch Modell von $\phi$ ist.*

**Theorem 5.6.3**
*Ist $T$ universell und $\phi$ existentiell, so gilt*

$$T \models_{Herb} \quad \textit{genau dann wenn} \;\; T \models_{Herb} \phi$$

Ist $T \models_{Herb} \phi$ nicht viel einfacher, da man nur Herbrand-Modelle beachten muß? Ist es vielleicht sogar entscheidbar? Mehr darüber in Kapitel 6.

Der folgende Satz von Herbrand ist die Grundlage der Resolution. Er führt, sozusagen, PL1 auf AL zurück.

**Theorem 5.6.4 (Herbrand)**
*Sei $T$ universell und $\phi$ enthalte keine Quantoren. Dann gilt:*

$$\begin{array}{lll} T \models \exists\phi & \textit{gdw} & \textit{es gibt } t_1, \ldots, t_n \in Term_{\mathcal{L}}(\emptyset) \\ & & \textit{mit: } T \models \phi(t_1) \vee \ldots \vee \phi(t_n) \end{array}$$

Oder: Sei $M$ eine Menge von Klauseln der PL1 (also Formeln der Form $P_1(t_1) \vee \neg P_2(t_2) \vee \ldots \vee P_n(t_n)$ wobei $t_i \in Term_{\mathcal{L}}(X)$). Dann ist $M$ unerfüllbar genau dann, wenn es eine endliche unerfüllbare Menge $M_{inst}$ von Grundinstanzen von $M$ gibt.

Im automatischen Beweisen interessiert uns immer die Frage

$$M \models \exists x_1, \ldots x_n \; \bigwedge_i \phi_i$$

Dann ist $M \cup \{\neg\exists x_1, \ldots x_n \; \bigwedge_i \phi_i\}$ eine Menge von Klauseln, die genau dann unerfüllbar ist, wenn die obige Relation gilt.

## 5.7 Robinson's Resolution

**Theorem 5.7.1 (Unifikation)**
*Gegeben sei eine endliche Menge von Gleichungen zwischen Termen oder Gleichungen zwischen Literalen. Dann gibt es einen Algorithmus der entweder eine allgemeinste Lösungssubstitution berechnet (d.h. eine Substitution für die beteiligten Variablen derart, daß linke und rechte Seiten aller Gleichungen syntaktisch identisch werden) oder* fail *zurückgibt. Im letzten Fall gibt es gar keine Lösungssubstitution. Im ersten Fall ist die allgemeinste Lösungssubstitution bis auf Variablenumbenennungen eindeutig bestimmt: sie heißt*

**mgU**, allgemeinster Unifikator

- $p(x, a) = q(y, b)$,
- $p(g(a), f(x)) = p(g(y), z)$.
  Grundsubstitutionen sind: $[y/a, x/a, z/f(a)]$, $[y/a, x/f(a), z/f(f(a))], \ldots$
  Der mgU lautet: $[y/a, z/f(x)]$ .

Wir wollen den auf der vorigen Folie erwähnten Algorithmus nun anhand eines Beispiels kurz skizzieren.

Gegeben: $f(x, g(h(y), y)) = f(x, g(z, a))$

Der Algorithmus berechnet nacheinander folgende Mengen von Gleichungen:

$$\begin{array}{l} \{\ x = x,\ g(h(y), y) = g(z, a)\ \} \\ \{\ g(h(y), y) = g(z, a)\ \} \\ \{\ h(y) = z,\ y = a\ \} \\ \{\ z = h(y),\ y = a\ \} \\ \{\ z = h(a),\ y = a\ \} \end{array}$$

Damit ergibt sich der mgU als: $[x/x, y/a, z/h(a)]$ .

**Definition 5.7.2 (Robinson's Resolution für PL1)**
*Die Resolution besteht aus zwei Regeln*

$$\textit{Resolution: } \frac{C_1 \cup \{A_1\} \quad C_2 \cup \{\neg A_2\}}{(C_1 \cup C_2) mgU(A_1, A_2)}$$

*wobei* $C_1 \cup \{A_1\}$ *und* $C_2 \cup \{A_2\}$ *als variablendisjunkt vorausgesetzt werden.*

$$\textit{Faktorisierung: } \frac{C_1 \cup \{L_1, L_2\}}{(C_1 \cup \{L_1\}) mgU(L_1, L_2)}$$

Als Beispiel betrachte man $M = \{r(x) \vee \neg p(x), p(a), s(a)\}$ und die Frage $M \models \exists x(s(x) \wedge r(x))$?

Warum braucht man Faktorisierung? Man betrachte

$$M = \{s(x_1) \vee s(x_2), \neg s(y_1) \vee \neg s(y_2)\}$$

Resolviert man beide Klauseln miteinander, erhält man

$$\{s(x_1)\} \cup \{\neg s(y_1)\}$$

oder Varianten davon. Resolviert man diese neue Klausel mit eine der beiden in $M$, erhält man nur Varianten der entsprechenden Klausel in $M$. *Also ist $\Box$ niemals ableitbar (mit Resolution allein).* Faktorisierung löst das Problem sofort, denn man kann sowohl $s(x)$ als auch $\neg s(y)$ ableiten, und daraus die leere Klausel.

## 5.8 Der Situationskalkül

Wie axiomatisieren wir die Wumpus-Welt in PL1?

```
function KB-AGENT(percept) returns an action
   static: KB, a knowledge base
           t, a counter, initially 0, indicating time

   TELL(KB, MAKE-PERCEPT-SENTENCE(percept, t))
   action ← ASK(KB, MAKE-ACTION-QUERY(t))
   TELL(KB, MAKE-ACTION-SENTENCE(action, t))
   t ← t + 1
   return action
```

*Um Aktionen, bzw. deren Effekte konsistent zu beschreiben, stellen wir die Welt als Folge von Situationen (Schnappschüsse der Welt) dar. Dazu müssen wir jedes Prädikat um ein zusätzliches Argument erweitern.*

Wir benutzen ein Funktionssymbol

$$result(aktion, situation)$$

als Term für die Situation die ensteht, wenn in $situation$ die Aktion $aktion$ ausgeführt wird. Abbildung 5.1 veranschaulicht die Bedeutung der Terme $result(\dots)$.

**Aktionen:** $Turn\_right$, $Turn\_left$, $Foreward$, $Shoot$, $Grab$, $Release$, $Climb$. Wir brauchen auch ein Gedächtnis, in Form eines 3-stelligen Prädikates

$$At(person, location, situation)$$

wobei $person$ entweder $Wumpus$ oder der $Agent$ ist und $location$ für den momentanen Ort, codiert als Paar [i,j] steht.

Wichtige Axiome sind die sogenannten "Successor-state Axioms", die beschreiben, welche Auswirkungen Aktionen auf Situationen haben. Die allgemeine Form dieser

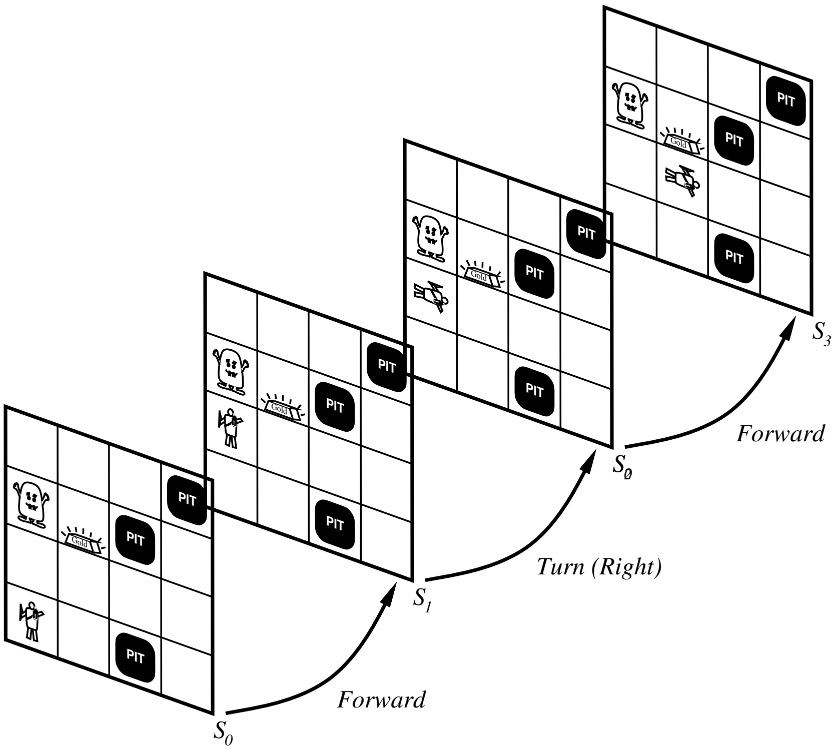



Abbildung 5.1: Der Situationskalkül.

Axiome ist

| | | |
|---|---|---|
| true afterwards | $\Longleftrightarrow$ | an action made it true<br>or it is already true and<br>no action made it false |

**Axiome über** $At(p, l, s)$**:**

$$At(p, l, result(a, s)) \leftrightarrow ((l = location\_ahead(p, s) \wedge \neg Wall(l) \wedge a = Forward) \\ \vee (At(p, l, s) \wedge \neg a = Forward))$$

$$At(p, l, s) \rightarrow Location\_ahead(p, s) = Location\_toward(l, Orient.(p, s))$$
$$Wall([x, y]) \leftrightarrow (x = 0 \vee x = 5 \vee y = 0 \vee y = 5)$$

$$Location\_toward([x, y], 0) = [x + 1, y]$$
$$Location\_toward([x, y], 90) = [x, y + 1]$$
$$Location\_toward([x, y], 180) = [x - 1, y]$$
$$Location\_toward([x, y], 270) = [x, y - 1]$$

$$
\begin{aligned}
&Orient.(Agent, s_0) = 90\\
&Orient.(p, result(a, s)) = d \leftrightarrow ((a = turn\_right \wedge d = mod(Orient.(p, s) - 90, 360))\\
&\qquad \vee (a = turn\_left \wedge d = mod(Orient.(p, s) + 90, 360))\\
&\qquad \vee (Orient.(p, s) = d \wedge \neg(a = Turn\_right \vee a = Turn\_left))
\end{aligned}
$$

Dabei ist $mod(x, y)$ die eingebaute “modulo”-Funktion, die jedes $x$ auf einen eindeutigen Wert zwischen $0$ und $y$ setzt.

**Axiome über Wahrnehmungen, sinnvolle neue Begriffe durch Definitionen:**

$$
\begin{array}{lcl}
Percept([Stench, b, g, u, c], s) & \rightarrow & Stench(s)\\
Percept([a, Breeze, g, u, c], s) & \rightarrow & Breeze(s)\\
Percept([a, b, Glitter, u, c], s) & \rightarrow & At\_Gold(s)\\
Percept([a, b, g, Bump, c], s) & \rightarrow & At\_Wall(s)\\
Percept([a, b, g, u, Scream], s) & \rightarrow & Wumpus\_dead(s)\\
\\
At(Agent, l, s) \wedge Breeze(s) & \rightarrow & Breezy(l)\\
At(Agent, l, s) \wedge Stench(s) & \rightarrow & Smelly(l)
\end{array}
$$

$$
\begin{array}{lcl}
Adjacent(l_1, l_2) & \leftrightarrow & \exists\, d\; l_1 = Location\_toward(l_2, d)\\
Smelly(l_1) & \rightarrow & \exists l_2\; At(Wumpus, l_2, s) \wedge (l_2 = l_1 \vee Adjacent(l_1, l_2))
\end{array}
$$

$$
\begin{array}{lcl}
Percept([none, none, g, u, c], s) \wedge At(Agent, x, s) \wedge Adjacent(x, y) & \rightarrow & OK(y)\\
(\neg At(Wumpus, x, t) \wedge \neg Pit(x)) & \rightarrow & OK(y)\\
\\
At(Wumpus, l_1, s) \wedge Adjacent(l_1, l_2) & \rightarrow & Smelly(l_2)\\
At(Pit, l_1, s) \wedge Adjacent(l_1, l_2) & \rightarrow & Breezy(l_2)
\end{array}
$$

**Axiome zur Beschreibung von Aktionen:**

$$
\begin{array}{lcl}
Holding(Gold, result(Grab, s)) & \leftrightarrow & (At\_Gold(s) \vee Holding(Gold, s))\\
Holding(Gold, result(Release, s)) & \leftrightarrow & \square\\
Holding(Gold, result(Turn\_right, s)) & \leftrightarrow & Holding(Gold, s)\\
Holding(Gold, result(Turn\_left, s)) & \leftrightarrow & Holding(Gold, s)\\
Holding(Gold, result(Forward, s)) & \leftrightarrow & Holding(Gold, s)\\
Holding(Gold, result(Climb, s)) & \leftrightarrow & Holding(Gold, s)
\end{array}
$$

Alle Effekte müssen sorgsam beschrieben werden.

**Axiome die Präferenzen zwischen Aktionen beschreiben:**

$$\begin{array}{ll}
Great(a,s) & \rightarrow Action(a,s)\\
(Good(a,s) \land \neg\exists b\, Great(b,s)) & \rightarrow Action(a,s)\\
(Medium(a,s) \land \neg\exists b\, (Great(b,s) \lor Good(b,s))) & \rightarrow Action(a,s)\\
(Risky(a,s) \land \neg\exists b\, (Great(b,s) \lor Good(b,s) \lor Medium(a,s))) & \rightarrow Action(a,s)
\end{array}$$

$$\begin{array}{ll}
At(Agent,[1,1],s) \land Holding(Gold,s) & \rightarrow Great(Climb,s)\\
At_Gold(s) \land \neg Holding(Gold,s) & \rightarrow Great(Grab,s)\\
At(Agent,l,s) \land \neg Visited(Location_ahead(Agent,s)) \land & \\
\land OK(Location_ahead(Agent,s)) & \rightarrow Good(Forward,s)
\end{array}$$

$$\begin{array}{ll}
Visited(l) & \leftrightarrow \exists s\, At(Agent,l,s)
\end{array}$$

Eigentlich geht es nicht nur darum das Gold zu finden, sondern auch wieder lebend zurückzukehren. Man bsaucht also noch Axiome wie $Holding(Gold,s) \rightarrow Go\_back(s)$ usw.

## 5.9 Wichtige Probleme der Wissensrepräsentation

Es haben sich 3 sehr wichtige Repräsentationsprobleme bei der Axiomatisierung einer sich ändernden Welt herausgestellt:

**Frame-Problem:** *Aktionen ändern meist nur sehr wenig. Man braucht sehr viele Aktionen um invariante Eigenschaften zu beschreiben.*

Ideal wäre es, nur was sich **nicht ändert** zu axiomatisieren und dann eine Aussage der Form “Sonst ändert sich nichts” hinzuzunehmen.

**Qualification-Problem:** *Das Aufzählen aller Bedingungen, unter denen eine Aktion Erfolg hat, ist nötig.* Z.B.

$$\begin{array}{lll}
\forall x & (Vogel(x) & \land\neg Pinguin(x) \land \neg Tot(x) \land\\
 & & \land\neg Strauss(x) \land \neg Flügel_gebr.(x) \land\\
 & & \land \ldots\\
 & \longrightarrow Fliegt(x) &
\end{array}$$

Ideal wäre es, nur “Normalerweise fliegen Vögel” aufzuschreiben.

**Ramification-Problem:** *Wie sollen implizite Konsequenzen von Aktionen gehandhabt werden?* Z.B $Grab(Gold)$ $Gold$ kann radioaktiv verseucht sein. Die Aktion $Grab(Gold)$ ist dann jedenfalls nicht optimal.

### Programmieren versus Knowledge Engineering

| **Programmieren** | **Knowledge Engineering** |
|---|---|
| Wähle Prog.-sprache | Wähle Logik |
| Schreibe Programm | Definiere KB |
| Schreibe Compiler | Implementiere Kalkül |
| Lass Programm laufen | Leite neue Fakten ab |

## 5.10 Aufgaben

**Aufgabe 5.1 (Aussagenlogik)**
*Welche der folgenden aussagenlogischen Formeln $F$ sind gültig, erfüllbar bzw. unerfüllbar? Nennen Sie alle Modelle zu $F$! Bestimmen Sie $Cn(\{F\})$!*
*(a)* $\phi \wedge \psi$ *(b)* $\phi \vee \psi$ *(c)* $(\phi \wedge (\phi \vee \psi)) \to \phi$ *(d)* $(\phi \to \psi) \to (\neg\phi \to \neg\psi)$

**Aufgabe 5.2 (Konnektive)**
*Welche logischen Konnektive außer $\wedge, \vee, \neg, \to$ halten Sie für angebracht? Wie viele binäre Konnektive kann es überhaupt geben? Warum sind nicht alle sinnvoll?*

**Aufgabe 5.3 (n-SAT)**
*Das $n$-SAT-Problem besteht darin herauszufinden, ob eine gegebene endliche Menge von Klauseln mit höchstens jeweils $n$ Literalen erfüllbar ist. 3-SAT ist bekanntlich NP-vollständig. Was können Sie über die Komplexität von (a) 4– bzw. (b) 2-SAT sagen?*

**Aufgabe 5.4 (GSAT)**
*Betrachten Sie die folgende Prozedur GSAT – ein Greedy-Algorithmus zum Lösen des aussagenlogischen Erfüllbarkeitsproblems mit sogenanntem Random-Walk:*

```
PROCEDURE GSAT
INPUT: S : Klauselmenge; m,n : Integers; p : Wahrscheinlichkeit
OUTPUT: T : Modell von S
FOR i := 1 TO m
  T := "Zufallsbelegung der Variablen in S";
  FOR j := 1 TO n
    IF "T erfüllt S" THEN RETURN(T);
    IF "Zufallszahl aus [0,1[" < p
      THEN x := "Variable aus unerfüllter Klausel in T"
      ELSE x := "Variable aus T, deren Belegungsänderung die Anzahl erfüllter Klauseln maxi-
    miert"
    T[x] := 1-T[x]    – "Wahrheitswert in T ändern"
```

1. *Ist der Algorithmus korrekt und vollständig, d.h. ist jede ausgegebene Belegung ein Modell bzw. wird in jedem Fall ein Modell gefunden, falls existent?*

2. *Implementieren Sie den Algorithmus in einer Programmiersprache Ihrer Wahl. Geben Sie neben dem Code bitte auch einige Testläufe mit ab!*

**Aufgabe 5.5 (Prädikatenlogik)**
*Was ist falsch an folgenden Argumentationen?*

1. *Falls eine Formel $F$ nicht gültig ist, läßt sich folgendermaßen ein Beispiel zu $\neg F$ finden: Zähle solange alle möglichen Herbrand-Strukturen $\mathcal{A}$ passend zu $F$ auf, bis ein Modell von $\neg F$ gefunden wird. Also ist das Erfüllbarkeitsproblem semi-entscheidbar.*

2. *Zur Herleitung der leeren Klausel wird Faktorisierung nicht benötigt, z.B. für die Menge $M = \{K_1, K_2\}$, wobei $K_1 = s(x_1) \vee s(x_2)$ und $K_2 = \neg s(y_1) \vee \neg s(y_2)$. Bilde dazu zunächst die Resolvente $R = s(x_2) \vee \neg s(y_2)$ aus $K_1$ und $K_2$. Resolviere dann $R$ und $K_1$ zu $R' = s(x_2)$. Zweimalige Resolution von $R'$ mit $K_2$ ergibt schließlich die leere Klausel.*

**Aufgabe 5.6 (Resolution)**
*Widerlegen Sie* $\forall x((p(x) \vee p(f(x))) \wedge (\neg p(x) \vee \neg p(f(f(x)))))$ *mittels prädikatenlogischer Resolution. Geben Sie jeweils eine Herleitung an, die (i) sowohl linear als auch eingabebeschränkt (d.h. Input-Ableitung), (ii) linear, aber nicht eingabebeschränkt, (iii) nicht linear, aber eingabebeschränkt bzw. (iv) weder linear noch eingabebeschränkt ist, oder überlegen Sie, warum dies nicht möglich ist!*

**Aufgabe 5.7 (Situationskalkül)**
*Formalisieren Sie die Block-Welt im Situationskalkül. Objekte in dieser Domäne seien Blöcke, Tische und Situationen. Als Prädikate seien*

| | |
|---|---|
| on(x,y,s) | *"x ist auf y in Situation s",* |
| clear(x,s) | *"Block x ist frei in Situation s, d.h. liegt zuoberst" und* |
| block(x) *bzw.* table(x) | *"x ist ein Block bzw. Tisch".* |

*gegeben. Die einzig mögliche Aktion sei* puton(x,y)*, die als Vorbedingung* clear(x) *und* clear(y) *erfordert. – Im folgenden seien* a*,* b *und* c *Blöcke und* t *ein Tisch. Die Schreibweise* xyzt *bezeichne den Sachverhalt, daß* x *auf* y *auf* z *auf* t *liegt.*

1. *Geben Sie Axiome zur Beschreibung von* puton(x,y) *an! – Beschreiben Sie den Anfangszustand* $s_0$=abct*. – Welche Anfrage ist an einen Theorembeweiser zu stellen, damit dieser einen Plan findet, um* cbat *zu erzeugen?*

2. *Versuchen Sie, das Ganze in Prolog zu realisieren!*

# 6 Planen

## 6.1 Planen vs. Problem-Lösen

**Problem-lösender Agent:** Die Effekte einer fixen Sequenz von Aktionen werden bestimmt.

**Wissensbasierter Agent:** Aktionen können ausgewählt werden.

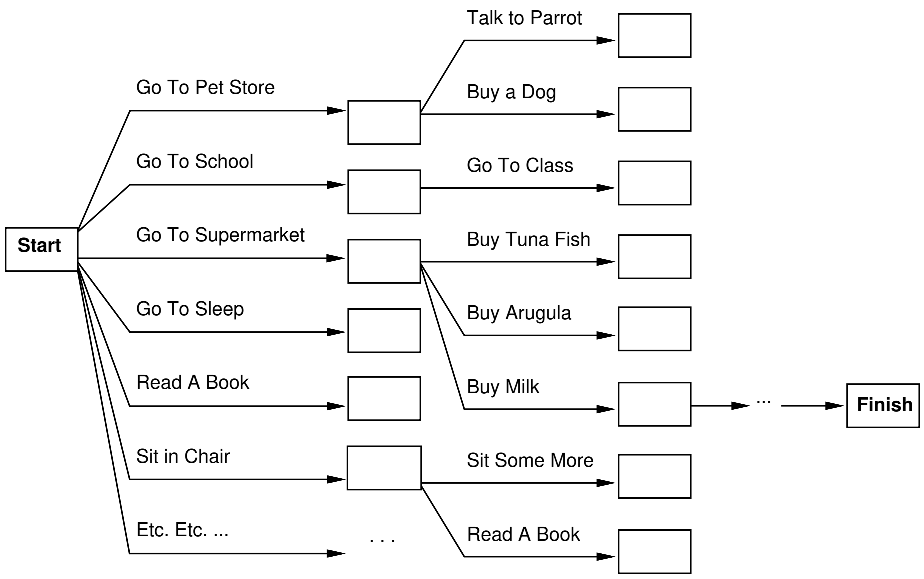
Beide zusammen versuchen wir zu einem **Planungsagenten** zu verschmelzen.

**Beispiel 6.1.1**
*Wir wollen eine frische Bananenmilch trinken und dann ein paar Löcher zuhause in die Wand bohren. Dazu brauchen wir Bananen, Milch und eine Bohrmaschine.*

Wie geht ein Problem-lösender Agent damit um?

```
function SIMPLE-PLANNING-AGENT(percept) returns an action
   static: KB, a knowledge base (includes action descriptions)
           p, a plan, initially NoPlan
           t, a counter, initially 0, indicating time
   local variables: G, a goal
                    current, a current state description

   TELL(KB, MAKE-PERCEPT-SENTENCE(percept, t))
   current ← STATE-DESCRIPTION(KB, t)
   if p = NoPlan then
      G ← ASK(KB, MAKE-GOAL-QUERY(t))
      p ← IDEAL-PLANNER(current, G, KB)
   if p = NoPlan or p is empty then action ← NoOp
   else
      action ← FIRST(p)
      p ← REST(p)
   TELL(KB, MAKE-ACTION-SENTENCE(action, t))
   t ← t + 1
   return action
```

Tabelle 6.1: Einfacher Planungs-Agent




Planen unterscheidet sich vom Problem-Lösen in folgenden Punkten:

1. Benutze PL1 um Verbindungen zwischen Zuständen und Aktionen herzustellen. Wenn $have(Milk)$ das Ziel ist und $Buy(x) \rightarrow Have(x)$, so ist es sinnvoll, $Buy(Milk)$ zu betrachten.

   *Dies vermeidet irrelevante Aktionen.*

2. Beim Zusammenstellen eines Planes dürfen Aktionen beliebig addiert werden: es muß nicht in sturer Reihenfolge gehen. Achtung: Reihenfolge des Plans hat zunächst nichts mit der Reihenfolge der Ausführung zu tun.

   *Dies setzt den Verzweigungsfaktor herab.*

   Zustände sind Mengen von Sätzen, daher nicht notwendig vollständig.

3. Die meisten Teile der Welt sind unabhängig voneinander: $Have(Milk) \wedge Have(Bananas) \wedge Have(Drill)$.

   *Dies ermöglicht Divide-et-Impera Strategien.*

## Planen im Situationskalkül

**Anfangszustand:** $At(Home, S_0)$, $\neg Have(Milk, S_0)$, $\neg Have(Bananas, S_0)$, $\neg Have(Drill, S_0)$.

**Ziel:** $\exists s\ (At(Home, s) \wedge Have(Milk, s) \wedge Have(Bananas, s) \wedge Have(Drill, s)$.

**Axiome:** z.B. über "Milch kaufen":

$$\forall a, s\ Have(Milk, result(a, s)) \leftrightarrow$$
$$(a = Buy(Milk) \wedge At(Supermarket, s)) \vee$$
$$(Have(Milk, s) \wedge a \neq Drop(Milk))$$

Wir brauchen nun noch einen Term $Result'(l, s)$: die Situation die entsteht, wenn in $s$ die Sequenz $l$ von Aktionen ausgeführt wird.

$$\begin{array}{lll} \forall s & Result'([], s) & := \quad s \\ \forall a, p, s & Result'([a|p], s) & := \quad Result'(p, result(a, s)) \end{array}$$

Die Aufgabe ist es nun, ein $p$ zu finden, mit $At(Home, Result'(p, S_0)) \wedge Have(Milk, Result'(p, S_0)) \wedge Have(Bananas, Result'(p, S_0)) \wedge Have(Drill, Result'(p, S_0))$.
Es erbenen sich folgende Probleme:

1. $T \vdash \phi$ ist nur semi-entscheidbar.
2. Ist $p$ ein Plan, so auch $[Leere\_Aktion|p]$ und $[A, A^{-1}|p]$.

Also sollte man möglichst keinen allgemeinen Beweiser, sondern einen auf den speziellen Fall zugeschnittenen benutzen. Außerdem sollte man die Sprache einschränken.

Man mache sich den Unterschied klar zwischen

- Goal an Planner und
- Query an Theorembeweiser.

Im ersten Fall sucht man einen Plan derart, daß *nach Ausführung* des Planes das Goal gilt. Im zweiten Fall fragt man ob die Query bezüglich der KB wahr gemacht werden kann: $KB \models \exists x \phi(x)$. Das dynamische steckt in den Termen. Die Logik selbst ist statisch.

## 6.2 STRIPS

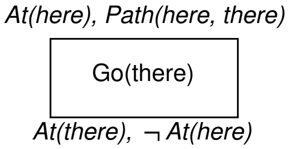
STRIPS steht für **ST**anford **R**esearch **I**nstitute **P**roblem **S**olver.

**Zustände:** Konjunktion von Funktions-freien Grund-Literalen

**Ziel:** Konjunktion von Funktions-freien Literalen

**Aktionen:** STRIPS-Operatoren bestehen aus 3 Komponenten

1. *Description*, Name der Aktion,
2. *Precondition*, Konjunktion von Atomen,
3. *Postcondition*, Konjunktion von Literalen.




**Definition 6.2.1 (Anwendbarer Operator)**
*Ein Operator $Op$ ist anwendbar in einem Zustand $s$, falls seine Variablen so instantiiert werden können, daß alle Preconditions in $s$ wahr sind: $Precond(s) \subseteq s$.*

*Im resultierenden Zustand gelten alle positiven Literale aus $Effect(Op)$ und alle Literale die in $s$ gelten, bis auf die negierten Literale aus $Effect(Op)$.*

Bisher kann man alles auch als Problem-Lösen sehen. Wir benutzen STRIPS als Repräsentationsformalismus und suchen nach einem Lösungspfad:

- Knoten im Suchbaum $\approx$ Situationen,
- Lösungspfade $\approx$ Pläne.

*Suche im Raum aller Pläne:* Fange mit partiellem Plan an und erweitere sukzessive. Dazu brauchen wir Operatoren die auf Plänen operieren. Wit unterscheiden zwischen 2 Arten:

**Refinement-Op:** Einem Plan werden Constraints angehängt. Ein Plan steht dann für die Menge aller vollständigen Pläne (analog steht $Cn(T)$ für $MOD(T)$).

**Modification-Op:** Alle anderen.

Man mache sich eindringlich den Unterschied zum Problem-Lösen klar. *Suche im Raum aller Pläne*, also eine Abstraktionsebene höher.

Wie repräsentiert man Pläne? Es gilt zwei wichtige Punkte zu beachten:

- *Variablen-Instantiierungen:* Instantiiere nur, falls notwendig. Nimm also stets den mgU.
- *Partielle Ordnung:* Sieh' zunächst von der genauen Reihenfolge ab (dadurch wird der Suchraum kleiner).

**Definition 6.2.2 (Plan)**
*Ein Plan ist ein Quadrupel bestehend aus*

1. *Einer Multi-Menge von* Schritten. *Ein Schritt ist ein Operator des Problems.*
2. *Einer Menge von* Ordnungs-Bedingungen *(auch* Constraints *genannt) an die Schritte. Diese haben die Form $S_i \prec S_j$: $S_i$ muß vor $S_i$ ausgeführt werden.*
3. *Einer Menge von* Instantiierungs-Bedingungen. *Diese haben die Form $v = x$ wobei $v$ eine Variable in einem Schritt und $x$ eine Konstante oder Variable ist.*

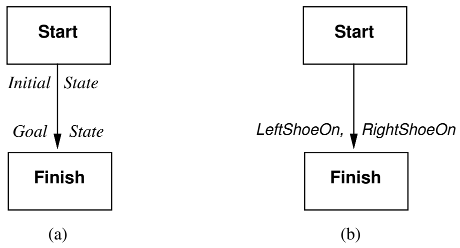


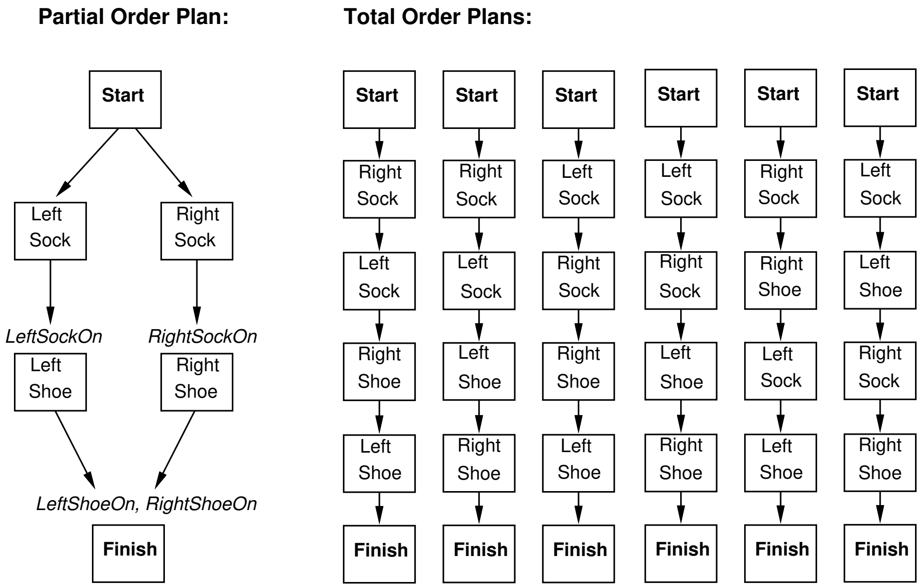

Abbildung 6.1: Der Anfangsplan

4. *Einer Menge von* kausalen Links. *Diese haben die Form* $S_i \xrightarrow{c} S_j$*:* $S_i$ *stellt* $c$ *sicher für* $S_j$.

Um den Anfangsplan eines Problems darzustellen definieren wir 2 Operatoren START und FINISH wie folgt
Als Anfangsplan eines Problems ergibt sich damit:

$$\textbf{Plan(}\quad \begin{array}{ll} Steps: & \{S_1 : START, S_2 : FINISH\} \\ Orderings: & \{S_1 \prec S_2\} \\ Bindings: & \emptyset \\ Links: & \emptyset \end{array}$$
$$\textbf{)}$$

Wir wollen nun das Problem Schuhe und Socken anzuziehen betrachten.




*Was ist eine Lösung?* Das ist einfach, falls wir nur voll instantiierte, linear geordnete Pläne betrachten: dann können wir es nämlich einfach überprüfen. In unserem Fall ist es aber komplizierter:

**Definition 6.2.3 (Lösung eines Planes)**
*Eine Lösung ist ein* vollständiger *und* konsistenter *Plan. Dabei bedeutet vollständig, daß jede Precondition eines Schrittes von einem anderen Schritt sichergestellt wird und konsistent, daß es keine Widersprüche in den Instantiierungs oder Ordnungs-Bedingungen gibt.*

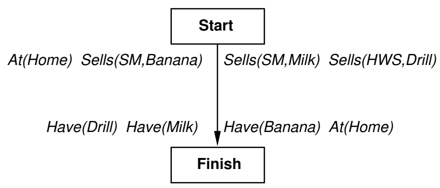
Noch präziser:

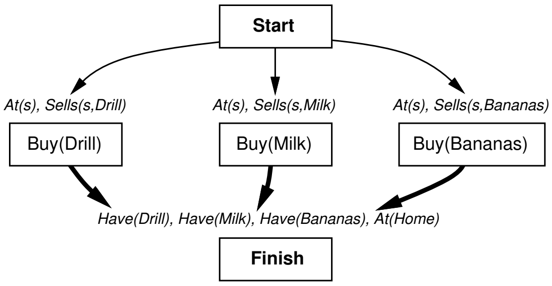
- "$S_i$ stellt $c$ sicher für $S_j$" bedeutet
  - $c \in Precondition(S_j)$,
  - $S_i \prec S_j$,
  - $\nexists S_k : \neg c \in Effect(S_k)$ mit $S_i \prec S_k \prec S_j$ in irgendeiner Linearisierung des Plans.
- "Keine Widersprüche" bedeutet
  - Weder ($S_i \prec S_j$ und $S_j \prec S_i$) noch ($v = A$ und $v = B$ für verschiedene Konstanten $A, B$).

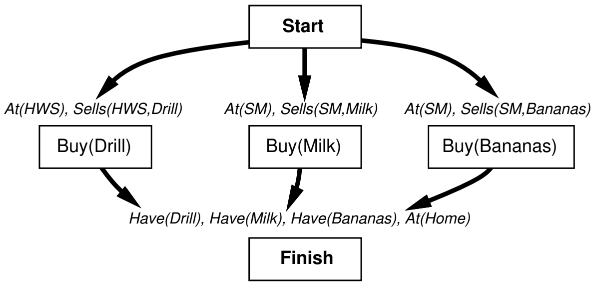
  Man beachte, daß dies nicht nur explizit gefordert wird. Die entsprechenden Aussagen können auch folgen, da $\prec$ und $=$ etwa transitiv sind.

## 6.3 Partial-Order Planning

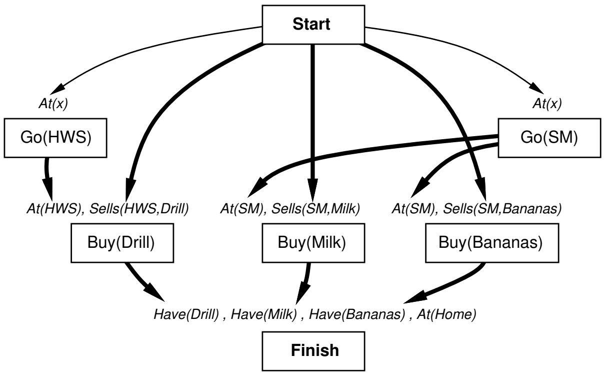
Wir betrachten unser Bananen-Milch-Beispiel. Als Operatoren verwenden wir $Buy$ und $Go$.

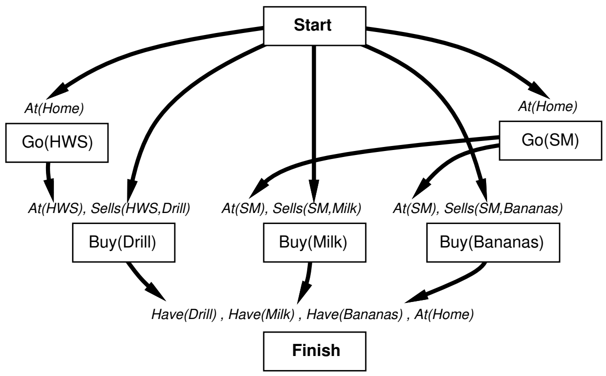




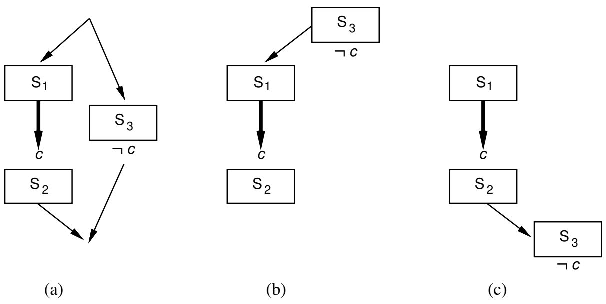

Kausale Links werden geschützt! Kausale Links sind fett eingezeichnet. Sie schließen den entsprechenden Ordnungs-Constraint ein!





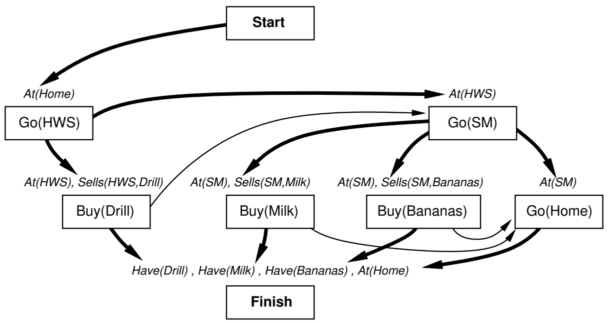

Der letzte partielle Plan kann nicht mehr erweitert werden. Wie stellt man das fest? Indem man feststellt, daß ein kausaler Link bedroht wird und die Drohung nicht aufgehoben werden kann!

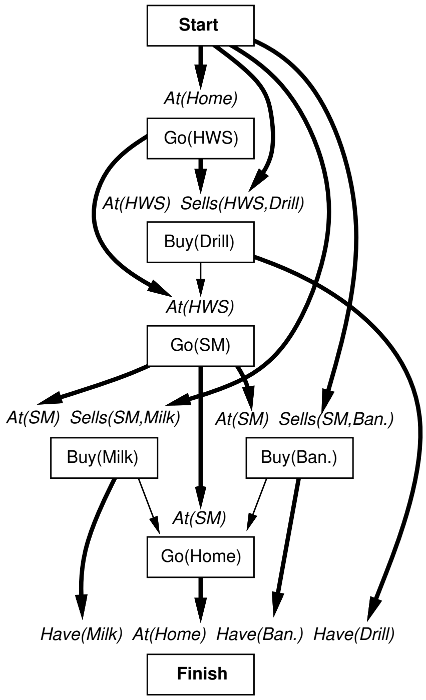
**Definition 6.3.1 (Drohung)**




*Eine Drohung kann aufgehoben werden, indem man die Drohung vor- oder nachschaltet (das muss nicht immer gehen!).*

Wir müssen einen Go-Schritt einführen, um die letzte Vorbedingung zu sichern. Aber wie können wir die Vorbedingung des Go-Schrittes sichern? Es gibt jetzt jede Menge Bedrohungen, von denen sich die meisten nicht auflösen lassen. Schließlich kommen wir zu






Das führt also zu folgendem Algorithmus:

- In jeder Runde wird der Plan erweitert um die Vorbedingung eines Schrittes zu sichern. Dies geschieht durch Auswahl eines geeigneten Operators.
- Der entsprechende kausale Link wird eingeführt. Drohungen werden durch Ordnungs-Constraints abgewehrt (es gibt 2 Fälle: der neue Schritt bedroht bestehende oder bestehende bedrohen den neuen).
- Falls es keinen Operator gibt oder die Drohungen nicht abgewehrt werden können — backtracken.

**Theorem 6.3.2 (POP)**
*POP ist vollständig und korrekt.*

Ist unser Beispiel eigentlich fair? Wir hatten doch nur sehr wenige Operatoren (ganze 2) und brauchen uns daher nicht zu wundern, daß der Verzweigungsfaktor so klein ist!

Dieser Einwand ist nicht gerechtfertigt, da auch wenn wir noch 1000 andere irrelevante Operatoren zulassen, diese nicht betrachtet werden, da wir sie nicht benutzen können, um unsere Vorbedingungen zu sichern. POP ist eben sehr ziel-gerichtet und vermeidet absolut irrelevante Aktionen.

Bisher haben wir uns überhaupt nicht um Variablen-Substitutionen gekümmert. Angenommen, $S_1$ sichert die $At(home)$ Vorbedingung eines anderen Schrittes $S_2$ und

**function** POP(*initial, goal, operators*) **returns** *plan*

  *plan* ← MAKE-MINIMAL-PLAN(*initial, goal*)
  **loop do**
    **if** SOLUTION?(*plan*) **then return** *plan*
    $S_{need}$, $c$ ← SELECT-SUBGOAL(*plan*)
    CHOOSE-OPERATOR(*plan, operators*, $S_{need}$, $c$)
    RESOLVE-THREATS(*plan*)
  **end**

---

**function** SELECT-SUBGOAL(*plan*) **returns** $S_{need}$, $c$

  pick a plan step $S_{need}$ from STEPS(*plan*)
    with a precondition $c$ that has not been achieved
  **return** $S_{need}$, $c$

---

**procedure** CHOOSE-OPERATOR(*plan, operators*, $S_{need}$, $c$)

  **choose** a step $S_{add}$ from *operators* or STEPS(*plan*) that has $c$ as an effect
  **if** there is no such step **then fail**
  add the causal link $S_{add} \xrightarrow{c} S_{need}$ to LINKS(*plan*)
  add the ordering constraint $S_{add} \prec S_{need}$ to ORDERINGS(*plan*)
  **if** $S_{add}$ is a newly added step from *operators* **then**
    add $S_{add}$ to STEPS(*plan*)
    add $Start \prec S_{add} \prec Finish$ to ORDERINGS(*plan*)

---

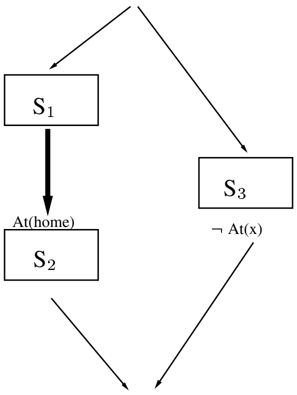
**procedure** RESOLVE-THREATS(*plan*)

  **for each** $S_{threat}$ that threatens a link $S_i \xrightarrow{c} S_j$ in LINKS(*plan*) **do**
    **choose** either
      *Promotion:* Add $S_{threat} \prec S_i$ to ORDERINGS(*plan*)
      *Demotion:* Add $S_j \prec S_{threat}$ to ORDERINGS(*plan*)
    **if not** CONSISTENT(*plan*) **then fail**
  **end**

Tabelle 6.2: POP-Algorithmus.

es gibt einen paralellen Schritt $S_3$ mit der Nachbedingung $\neg At(x)$. Ist dies eine Bedrohung?

Wir nennen solch eine Drohung *möglich* und ignorieren sie zunächst, behalten sie aber im Hinterkopf. Falls aber irgendwann einmal $x$ mit $home$ instantiiert wird, dann entsteht eine echte Drohung die aufgelöst werden muss.




**procedure** CHOOSE-OPERATOR(*plan, operators,* $S_{need}$, *c*)

**choose** a step $S_{add}$ from *operators* or STEPS(*plan*) that has $c_{add}$ as an effect
such that *u* = UNIFY(c, $c_{add}$, BINDINGS(*plan*))
**if** there is no such step
**then fail**
add *u* to BINDINGS(*plan*)
add $S_{add} \xrightarrow{c} S_{need}$ to LINKS(*plan*)
add $S_{add} \prec S_{need}$ to ORDERINGS(*plan*)
**if** $S_{add}$ is a newly added step from *operators* **then**
add $S_{add}$ to STEPS(*plan*)
add $Start \prec S_{add} \prec Finish$ to ORDERINGS(*plan*)

---

**procedure** RESOLVE-THREATS(*plan*)

**for each** $S_i \xrightarrow{c} S_j$ **in** LINKS(*plan*) **do**
**for each** $S_{threat}$ **in** STEPS(*plan*) **do**
**for each** $c'$ **in** EFFECT($S_{threat}$) **do**
**if** SUBST(BINDINGS(*plan*), *c*) = SUBST(BINDINGS(*plan*), ¬ $c'$) **then**
**choose** either
*Promotion:* Add $S_{threat} \prec S_i$ to ORDERINGS(*plan*)
*Demotion:* Add $S_j \prec S_{threat}$ to ORDERINGS(*plan*)
**if not** CONSISTENT(*plan*)
**then fail**
**end**
**end**
**end**

## SHAKEY

STRIPS wurde ursprünglich für SHAKEY entworfen, einen kleinen mobilen Roboter. SHAKEY wurde duch 6 Operatoren beschrieben: $Go(x)$, $Push(b,x,y)$, $Climb(b)$, $Down(b)$, $Turn\_On(ls)$, $Turn\_Off(ls)$.

Damit SHAKEY den Lichtschalter an oder ausstellen kann, muss er auf einer Box stehen. Damit SHAKEY bei einer Go-Aktion nicht von einer Box fällt, ist $On(Shakey, floor)$ eine Vorbedingung der Go-Aktion.

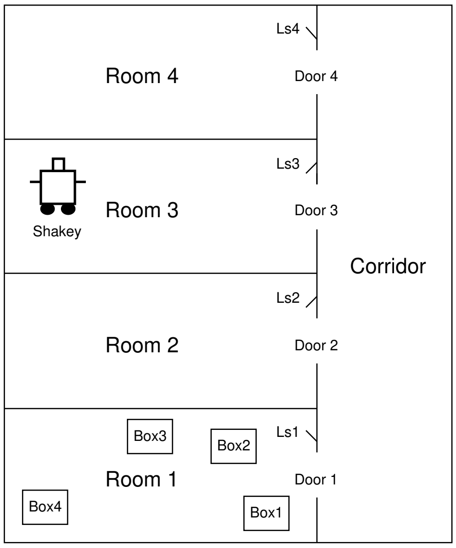



Abbildung 6.2: Shakey

## 6.4 Conditional Planning

Was tun, wenn die Welt nicht vollständig zugänglich ist? (Wo bekommt man Milch? Milch-Preis hat sich verdoppelt, man hat nicht genug Geld dabei!)

*Führe neue Aktionen ein um gewisse Gegebenheiten abzufragen und entsprechend zu reagieren.* (Wieviel kostet die Milch?)

### Ein platter Reifen

$$
\begin{array}{lll}
\textbf{Op(} & Action: & Remove(x), \\
 & Prec: & On(x), \\
 & Effect: & Off(x) \wedge Clear\_Hub(x) \wedge \neg On(x) \\
\textbf{)} & & \\
\textbf{Op(} & Action: & Put\_on(x), \\
 & Prec: & Off(x) \wedge Clear\_Hub(x), \\
 & Effect: & On(x) \wedge \neg Clear\_Hub(x) \wedge \neg Off(x) \\
\textbf{)} & & \\
\textbf{Op(} & Action: & Inflate(x), \\
 & Prec: & Intact(x) \wedge Flat(x), \\
 & Effect: & Inflated(x) \wedge \neg Flat(x) \\
\textbf{)} & &
\end{array}
$$

**Ziel:** $On(x) \wedge Inflated(x)$,

```
function CONDITIONAL-PLANNING-AGENT(percept) returns an action
   static: KB, a knowledge base (includes action descriptions)
           p, a plan, initially NoPlan
           t, a counter, initially 0, indicating time
           G, a goal

   TELL(KB, MAKE-PERCEPT-SENTENCE(percept, t))
   current ← STATE-DESCRIPTION(KB, t)
   if p = NoPlan then p ← CPOP(current, G, KB)
   if p = NoPlan or p is empty then action ← NoOp
   else
       action ← FIRST(p)
       while CONDITIONAL?(action) do
           if ASK(KB, CONDITION-PART[action]) then p ← APPEND(THEN-PART[action], REST(p))
           else p ← APPEND(ELSE-PART[action], REST(p))
           action ← FIRST(p)
       end
       p ← REST(p)
   TELL(KB, MAKE-ACTION-SENTENCE(action, t))
   t ← t + 1
   return action
```

Tabelle 6.3: Ein bedingter Planungs-Agent.

**Anfangszustand:** $Inflated(Spare) \wedge Intact(Spare) \wedge Off(Spare) \wedge On(Tire_1) \wedge Flat(Tire_1)$.

Was liefert unser POP? Da $Intact(Tire_1)$ nicht vorhanden, liefert POP als Plan

$$[Remove(Tire_1), Put\_on(Spare)].$$

Würden Sie das auch so machen? Besser wäre es doch, zunächst einmal nachzusehen, ob der Reifen wirklich kaputt ist, und nur in diesem Fall den ersatzreifen zu nehmen. Wir brauchen also einen *konditionalen Plan*:

$$\text{Falls } Intact(Tire_1) \text{ dann: } Inflate(Tire_1).$$

Dazu müssen wir konditionale Schritte schon bei der Planerstellung erlauben:

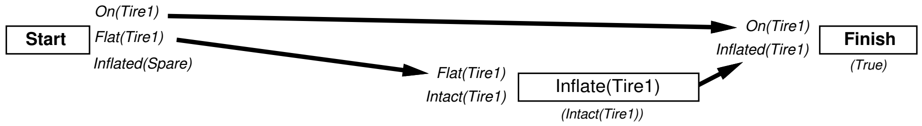
**Definition 6.4.1 (Konditionale Schritte)**
*Ein konditionaler Schritt in einem Plan hat die Form*

$$\mathit{If}(\{Condition\}, \{Then\_Part\}, \{Else\_Part\})$$

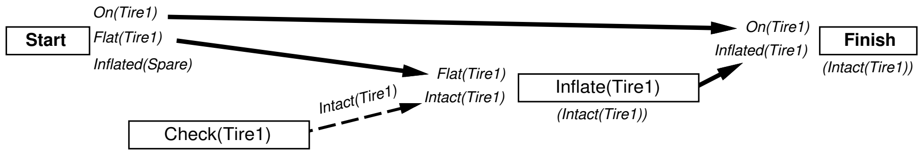
Der entsprechende Planungsagent sieht dann so aus: Der Agent muß beim Ausführen des Planes wissen (also erst zur Laufzeit) ob die entsprechende If-Bedingung gilt oder nicht. Daher führt man neue Aktionen zur Überprüfung ein:

$$\mathbf{Op(}\quad \begin{array}{ll} Action: & Check\_Tire(x), \\ Prec: & Tire(x), \\ Effect: & \text{“Man weiss, ob } Intact(Tire) \text{ oder nicht.”} \end{array}$$
$$\mathbf{)}$$

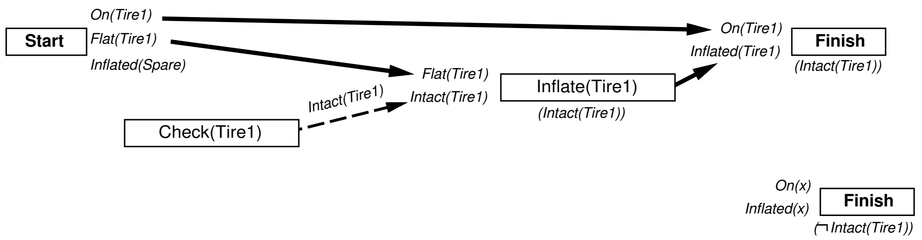
**Definition 6.4.2 (Kontext)**
*Wir assoziieren mit jedem Schritt einen* Kontext*: das ist die Menge der Bedingungen (Conditions), die gelten müssen, damit der Schritt ausgeführt werden kann.*

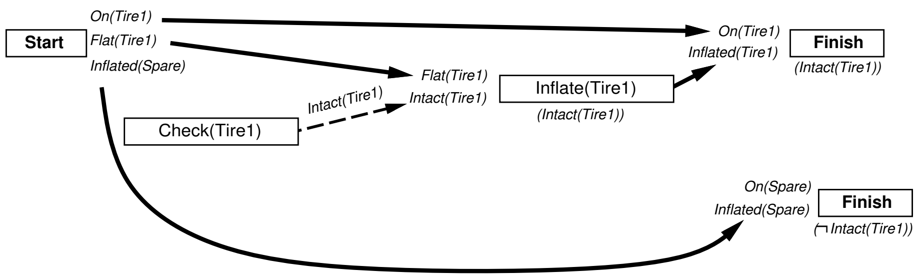

Hier würde POP schon backtracken, da $Intact(Tire_1)$ nicht gezeigt werden kann. Daher führt man eine neue Sorte von Links ein: *conditional links*.



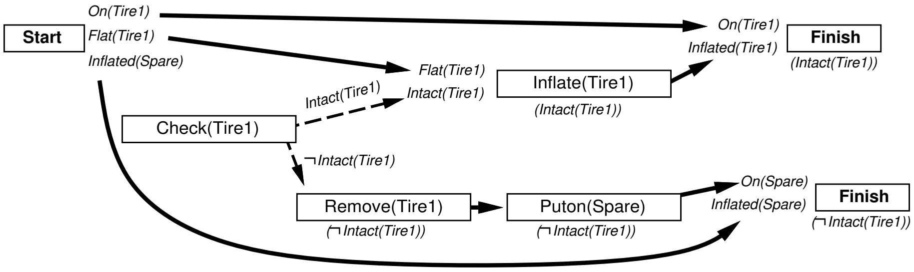

Man beachte, daß alle Fälle abgedeckt werden müssen. Also:




Wie können wir nun $Inflated(x)$ im FINISH-Schritt wahr machen? Das Hinzunehmen des Schrittes $Inflate(Tire_1)$ macht keinen Sinn, da ja die Vorbedingungen + Kontext inkonsistent miteinander sind.




Anstelle der letzten Modifikation könnte man auch einen $Inflate(Spare)$ Schritt einführen, aber das führt dann zu nichts. Es fehlt jetzt nur noch $On(Spare)$.




Achtung: zunächst steht "True" unter $Remove(Tire_1), Put\_on(Spare)$. Dann gibt es aber eine Drohung mit dem oberen Link. *Diese kann aufgelöst werden, indem die Kontexte inkompatibel gemacht werden. Die entsprechenden Kontexte vererben sich.* Genauer: Suche einen konditionalen Schritt, dessen Nachbedingung die Kontexte inkompatibel macht und damit Drohungen auflöst.

## 6.5 Erweiterungen

**1.** Ein Plan kann auch aus folgenden Gründen schiefgehen

- Aktionen können unerwartete Effekte haben, aber diese können aufgezählt werden (als Disjunktion).
- Die unerwarteten Effekte sind unbekannt. Dann muss man neu planen: *Replanning*.

```
function REPLANNING-AGENT(percept) returns an action
  static: KB, a knowledge base (includes action descriptions)
          p, an annotated plan, initially NoPlan
          q, an annotated plan, initially NoPlan
          G, a goal

  TELL(KB, MAKE-PERCEPT-SENTENCE(percept, t))
  current ← STATE-DESCRIPTION(KB, t)
  if p = NoPlan then
     p ← PLANNER(current, G, KB)
     q ← p
     if p = NoPlan or p is empty then return NoOp
  if PRECONDITIONS(p) not currently true in KB then
     p' ← CHOOSE-BEST-CONTINUATION(current, q)
     p ← APPEND(PLANNER(current, PRECONDITIONS(p'), KB), p')
     q ← p
  action ← FIRST(p)
  p ← REST(p)
  return action
```

D.h. man nimmt wahr und plant nur, wenn sich etwas verändert hat.

**2.** Man verbinde *Replanning* und *Conditional Planning*. Planen und Ausführung werden miteinander integriert.

**function** CPOP(*initial, goals, operators*) **returns** *plan*

*plan* ← MAKE-PLAN(*initial, goals*)
**loop do**

*Termination*:
**if** there are no unsatisfied preconditions
and the contexts of the finish steps are exhaustive
**then return** *plan*

*Alternative context generation*:
**if** the plans for existing finish steps are complete and have contexts $C_1 \ldots C_n$ **then**
add a new finish step with a context $\neg (C_1 \vee \ldots \vee C_n)$
this becomes the *current context*

*Subgoal selection and addition*:
find a plan step $S_{need}$ with an open precondition $c$

*Action selection*:
**choose** a step $S_{add}$ from *operators* or STEPS(*plan*) that adds $c$ or
knowledge of $c$ and has a context compatible with the current context
**if** there is no such step
**then fail**
add $S_{add} \xrightarrow{c} S_{need}$ to LINKS(*plan*)
add $S_{add} < S_{need}$ to ORDERINGS(*plan*)
**if** $S_{add}$ is a newly added step **then**
add $S_{add}$ to STEPS(*plan*)
add $Start < S_{add} < Finish$ to ORDERINGS(*plan*)

*Threat resolution*:
**for each** step $S_{threat}$ that potentially threatens any causal link $S_i \xrightarrow{c} S_j$
with a compatible context **do**
**choose** one of
*Promotion:* Add $S_{threat} < S_i$ to ORDERINGS(*plan*)
*Demotion:* Add $S_j < S_{threat}$ to ORDERINGS(*plan*)
*Conditioning*:
find a conditional step $S_{cond}$ possibly before both $S_{threat}$ and $S_j$, where
1. the context of $S_{cond}$ is compatible with the contexts of $S_{threat}$ and $S_j$;
2. the step has outcomes consistent with $S_{threat}$ and $S_j$, respectively
add conditioning links for the outcomes from $S_{cond}$ to $S_{threat}$ and $S_j$
augment and propagate the contexts of $S_{threat}$ and $S_j$

**if** no choice is consistent
**then fail**
**end**
**end**

Tabelle 6.4: Der Conditional-Planning-Algorithmus

```
function SITUATED-PLANNING-AGENT(percept) returns an action
  static: KB, a knowledge base (includes action descriptions)
          p, a plan, initially NoPlan
          t, a counter, initially 0, indicating time
          G, a goal

  TELL(KB, MAKE-PERCEPT-SENTENCE(percept, t))
  current ← STATE-DESCRIPTION(KB, t)
  EFFECTS(START(p)) ← current
  if p = NoPlan then
     G ← ASK(KB, MAKE-GOAL-QUERY(t))
     p ← MAKE-PLAN(current, G, KB)
  action ← NoOp (the default)

   Termination:
     if there are no open preconditions and p has no steps other than START and FINISH then
        p ← NoPlan and skip remaining steps

   Resolving standard flaws:
     resolve any open condition by adding a causal link from any existing
        possibly prior step or a new step
     resolve potential threats by promotion or demotion

   Remove unsupported causal links:
     if there is a causal link START --c--> S protecting a proposition c
       that no longer holds in START then
        remove the link and any associated bindings

   Extend causal links back to earliest possible step:
     if there is a causal link S_j --c--> S_k such that
       another step S_i exists with S_i < S_j and the link S_i --c--> S_k is safe then
        replace S_j --c--> S_k with S_i --c--> S_k

   Remove redundant actions:
     remove any step S that supplies no causal links

   Execute actions when ready for execution:
     if a step S in the plan other than FINISH satisfies the following:
        (a) all preconditions satisfied by START;
        (b) no other steps necessarily between START and S; and
        (c) S does not threaten any causal link in p then
            add ordering constraints to force all other steps after S
            remove S from p, and all causal links to and from S
            action ← the action in S

  TELL(KB, MAKE-ACTION-SENTENCE(action, t))
  t ← t + 1
  return action
```

## 6.6 Aufgaben

**Aufgabe 6.1 (Partiell geordnete Pläne)**
*Definieren Sie ein Operatoren-Schema für das Problem, Schuhe, Socken, Hut und Mantel anzuziehen. Es sei angenommen, daß Hut und Mantel vorbedingungslos angezogen werden können. Geben Sie einen partiell geordneten Plan zur Lösung des Problems*

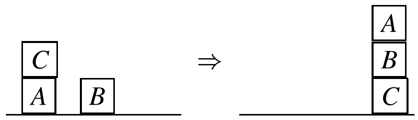
*an! Wie viele Linearisierungen gibt es?*

**Aufgabe 6.2 (Makro-Operationen)**

1. *Die STRIPS-Operatoren* pickup(x) *und* stack(x,y) *in einer Block-Welt kann man zu einem Makro-Operator* puton(x,y) *zusammenfassen. Wie sehen die drei Komponenten (Aktionsbeschreibung, Vorbedingung, Effekt) des Makro-Operators aus?*

2. *Versuchen Sie Bedingungen anzugeben, wie ganz allgemein Makro-Operatoren aus zwei gegebenen STRIPS-Operatoren automatisch abgeleitet werden können.*

**Aufgabe 6.3 (Sussman-Anomalie)**
*Betrachten Sie das untenstehende Problem aus der Block-Welt. Formulieren Sie es mittels STRIPS-Operatoren und benutzen Sie die Funktion POP (siehe unten), um es zu lösen!*




**Aufgabe 6.4 (Algorithmus POP)**
*Eine wichtige Verfeinerung des POP-Algorithmus ist, daß der ausgewählte Schritt und die Vorbedingung in der Funktion* SELECT-SUBGOAL nicht *Backtracking unterworfen werden. Wie wirkt sich das auf (a) die Korrektheit, (b) die Vollständigkeit und (c) die Komplexität des Verfahrens aus?*

**Aufgabe 6.5 (Shakey's World I)**
*Beschreiben Sie die 6 Aktionen von* Shakey *(a) im Situationskalkül und (b) als STRIPS-Operatoren. Betrachten Sie dazu auch Abbildung 6.3.*

**Aufgabe 6.6 (Shakey's World II)**

1. *Konstruieren Sie einen partiell geordneten Plan, um* Box2 *in Raum* Room2 *zu befördern, ausgehend von der Startsituation in Abbildung 6.3.*

2. *Angenommen,* Shakey *muß $n$ Boxen von einem Raum in einen anderen befördern. Wie groß ist die Komplexität des Planungsprozesses in Abhängigkeit von $n$?*

**Aufgabe 6.7 (Conditional Planning)**
*Diskutieren Sie die Anwendung von* Conditional Planning *auf (a) die Staubsauger-Welt und (b) die Wumpus-Welt.*

# 7 Lernen aus Beobachtungen

**Bisher:** Die Intelligenz eines Agenten steckt in seinem Programm, sie ist fest verdrahtet.

**Jetzt:** Der Agent soll autonomer **werden**, durch eigene **Beobachtungen** (Erfahrungen) soll er seine Umgebung verstehen lernen.

**Wichtig:** Falls der Bereich in dem er arbeitet nur unvollständig beschrieben werden kann.

## 7.1 Grundlagen

**Performance-Element:** Beobachtet und wählt Aktionen aus. War bisher der gesamte Agent.

**Kritik:** Beobachtet das Resultat der Aktionen und bewertet sie. Bewertet anhand eines Standards **von aussen**!! (Warum? sonst wählt er seinen eigenen Standard so niedrig, dass er ihn immer erfüllt!)

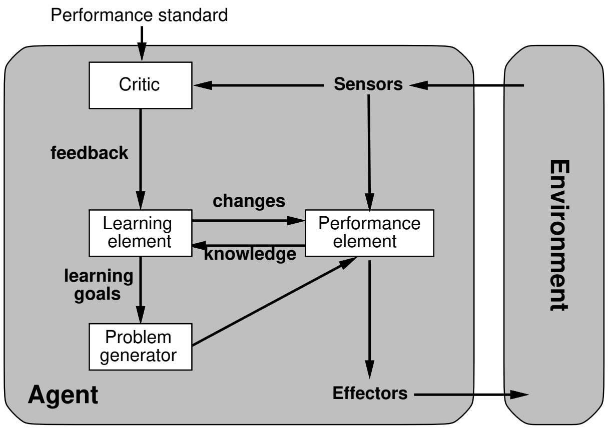



Abbildung 7.1: Ein Lern-Agent

**Lern-Element:** Modifiziert das Performance-Element indem es die Kritik berücksichtigt (und den inneren Aufbau des Perf.-Elem.). Außerdem stellt es neue Ziele auf (um die Effekte von Aktionen besser zu verstehen).

**Problem Generator:** Er schlägt vor Aktionen auszuführen, um die Goals des Learning-Elementes zu befriedigen. Dies müssen nicht immer die "besten" Aktionen sein (i. S. d. Perf.-Elem.): aber sie sollen **informativ** sein und neues Wissen über die Welt liefern.

Wovon hängt Lernen (also der Entwurf des Lern-Elementes) eigentlich ab?

1. Welche Komponenten des Performance-Elementes sollen verbessert werden?
2. Wie sind diese Komponenten repräsentiert?
   *($\leadsto$ Kapitel 7, Abschnitte 7.2 und 7.3*
3. Welches feedback gibt es?
   - **Überwachtes L.:** Das Ausführen einer inkorrekten Aktion erzeugt als feedback die "richtige" Lösung (z.B. wie stark aufs Bremspedal treten). *Fahrlehrer*.
   - **Reinforcement L.:** Nur das Resultat wird wahrgenommen. Kritik sagt, ob es gut oder schlecht, aber nicht was richtig gewesen wäre.
   - **Nicht-Überwachtes L.:** Gar keine Hinweise über korrekte Aktionen.
4. Welche a-priori-Informationen gibt es?
   *(Oft gibt es nützliches Hintergrundwissen.)*

**Komponenten des Performance Elementes**

1. Abbildung von Bedingungen des aktuellen Zustandes in die Menge der Aktionen.
2. Ableiten relevanter Eigenschaften der Welt aus den Wahrnehmungen.
3. Informationen darüber, wie sich die Welt entwickelt.
4. Informationen über die Folgen von Aktionen.
5. Bewertungsfunktion über die Zustände der Welt.
6. Bewertungsfunktion über die Güte einzelner Aktionen in einem Zustand.
7. Beschreibung von Zustandsklassen, die die Nutzenfunktion des Agenten maximieren.

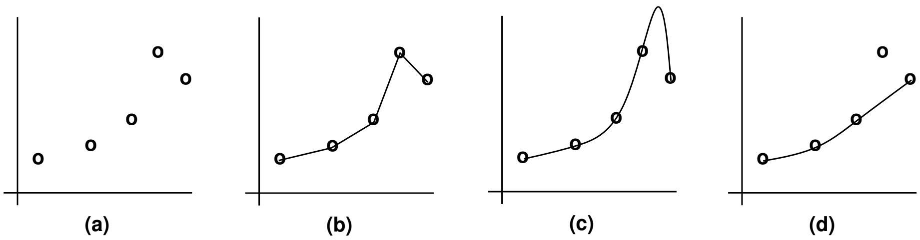
Alle diese Komponenten sind, mathematisch gesehen, *Funktionen*.

**Lernen heißt diese Funktionen zu repräsentieren.**

Beim Induktiven Lernen sind Paare $(x, y)$ gegeben und die Aufgabe besteht darin, die Funftion $f$ mit $f(x) = y$ zu bestimmen.

**Beispiel 7.1.1 (Zahlenfolgen fortsetzen)**
*Wie lautet die nächste Zahl:*

- *3, 5, 7, ?*
- *3, 5, 17, 257, ?*




Einfacher Reflex-Lern Agent:

```
global examples ← {}
---------------------------------------------------------------
function REFLEX-PERFORMANCE-ELEMENT(percept) returns an action

  if (percept, a) in examples then return a
  else
     h ← INDUCE(examples)
     return h(percept)
---------------------------------------------------------------
procedure REFLEX-LEARNING-ELEMENT(percept, action)
  inputs: percept, feedback percept
          action, feedback action

  examples ← examples ∪ {(percept,action)}
```

**Beispiel 7.1.2 (Verifizieren und falsifizieren)**
*Man betrachte eine Menge von Karten. Jede Karte enthält auf einer Seite einen Buchstaben und auf der anderen Seite eine Zahl. Nachdem man sich einige Karten angeschaut hat kommt man zu der Hypothese*

Ist auf einer Seite einer Karte ein Vokal, so ist auf der anderen Seite eine gerade Zahl.

*Es liegen nun folgende Karten auf dem Tisch:*

**A T 4 7**.

*Sie dürfen höchstens 2 Karten herumdrehen und sollen die Hypothese prüfen. Welche Karte drehen Sie herum?*

Es müssen die **A** und die **7** herumgedreht werden. Die **A** weil damit die Hypothese direkt überprüft werden kann, die **7** weil das die einzige Möglichkeit ist, die Hypothese auch zu falsifizieren. Dies wird oft nicht gemacht: man sucht nur nach Bestätigung, aber man versucht nicht die Hypothese zu widerlegen.

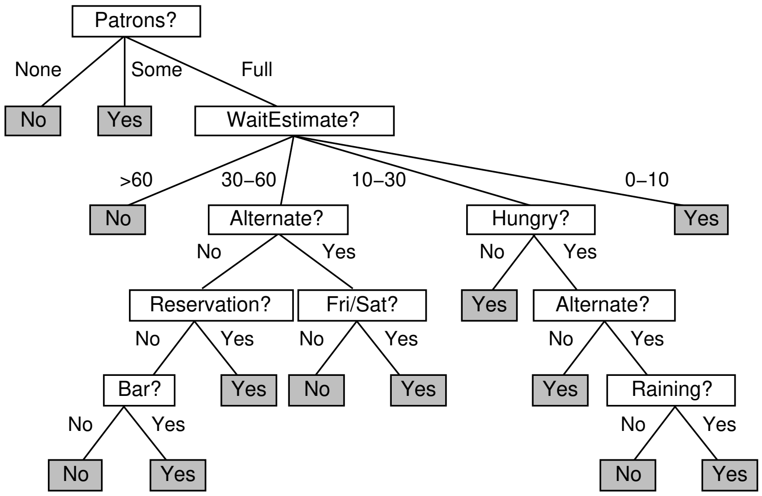



Abbildung 7.2: Ein Entscheidungsbaum

## 7.2 Entscheidungsbäume

Entscheidungsbäume sind Repräsentanten von booleschen Funktionen. Ein Beispiel: Man geht essen und erreicht ein Restaurant. Soll man auf einen freien Tisch warten oder weitergehen?

Entscheidungsbaum = Konjunktion von Implikationen[1]

$$\forall r \ (Patrons(r, Full) \wedge Wait\_estimate(r, 10-30) \wedge \neg Hungry(r)) \longrightarrow Will\_Wait(r)$$

Dies ist zwar in PL1 geschrieben, aber Entscheidungsbäume betreffen immer nur einzelne Objekte (das $r$ oben), dessen Eigenschaften abgefragt werden. Also geht es um Aussagenlogik.

*Kann jede boolesche Funktion als Entscheidungsbaum geschrieben werden?* Ja: Zeile der Wahrheitstabelle entspricht Pfad im Baum.

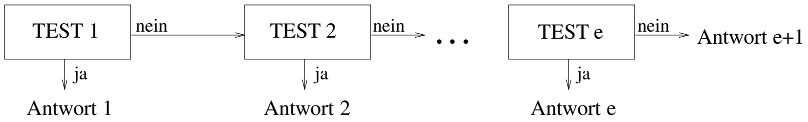
*E-Bäume können sehr viel kleiner sein!* Aber es gibt auch bool. Funktionen, die nur durch exponentiell große Bäume dargestellt werden können:

$$\text{Paritätsfunktion: } par(x_1, \dots, x_n) := \begin{cases} 1, & \text{falls } \sum_{i=1}^n x_i \text{ ist gerade} \\ 0, & \text{sonst} \end{cases}$$

Eine Variante der E-Bäume sind

[1] Implikation = Jeder Pfad zu einem Blatt.

**Beispiel 7.2.1 (Entscheidungslisten)**
*Alle Attribute seien booleschwertig. Eine Entscheidungsliste hat die Form*




*wobei $Antwort_i \in \{Ja, NEIN\}$ und $Test_i$ eine Konjunktion von (eventuell negierten) Attributen ist. (⇝ Aufgaben 7.1 und 7.2).*

*Sei $k - DL(n)$ die Menge der booleschen Funktionen in $n$-Attributen, die durch E-Listen mit höchstens $k$ Abfragen in jedem Test dargestellt werden können.*

Offensichtlich gilt:

- $n - DL(n)$ = Menge aller booleschen Funktionen,
- $card(n - DL(n)) = 2^{2^n}$.

Daß gilt $card(n - DL(n)) = 2^{2^n}$, sieht man anhand der Wahrheitstafel: für eine spezielle Funktion muss man $2^n$ Zeilen festlegen um sie zu definieren. In jeder Zeile gibt es 2 Möglichkeiten (0 oder 1).

## Wie lernt man denn nun E-Bäume?

| Example | Attributes | | | | | | | | | | Goal |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | *Alt* | *Bar* | *Fri* | *Hun* | *Pat* | *Price* | *Rain* | *Res* | *Type* | *Est* | *WillWait* |
| $X_1$ | *Yes* | *No* | *No* | *Yes* | *Some* | *$$$* | *No* | *Yes* | *French* | *0–10* | *Yes* |
| $X_2$ | *Yes* | *No* | *No* | *Yes* | *Full* | *$* | *No* | *No* | *Thai* | *30–60* | *No* |
| $X_3$ | *No* | *Yes* | *No* | *No* | *Some* | *$* | *No* | *No* | *Burger* | *0–10* | *Yes* |
| $X_4$ | *Yes* | *No* | *Yes* | *Yes* | *Full* | *$* | *No* | *No* | *Thai* | *10–30* | *Yes* |
| $X_5$ | *Yes* | *No* | *Yes* | *No* | *Full* | *$$$* | *No* | *Yes* | *French* | *>60* | *No* |
| $X_6$ | *No* | *Yes* | *No* | *Yes* | *Some* | *$$* | *Yes* | *Yes* | *Italian* | *0–10* | *Yes* |
| $X_7$ | *No* | *Yes* | *No* | *No* | *None* | *$* | *Yes* | *No* | *Burger* | *0–10* | *No* |
| $X_8$ | *No* | *No* | *No* | *Yes* | *Some* | *$$* | *Yes* | *Yes* | *Thai* | *0–10* | *Yes* |
| $X_9$ | *No* | *Yes* | *Yes* | *No* | *Full* | *$* | *Yes* | *No* | *Burger* | *>60* | *No* |
| $X_{10}$ | *Yes* | *Yes* | *Yes* | *Yes* | *Full* | *$$$* | *No* | *Yes* | *Italian* | *10–30* | *No* |
| $X_{11}$ | *No* | *No* | *No* | *No* | *None* | *$* | *No* | *No* | *Thai* | *0–10* | *No* |
| $X_{12}$ | *Yes* | *Yes* | *Yes* | *Yes* | *Full* | *$* | *No* | *No* | *Burger* | *30–60* | *Yes* |

Die Menge der Beispiele aus denen gelernt werden soll heißt *Trainingsmenge*. Beispiele können positiv (Attribut trifft zu) oder negativ (trifft nicht zu) sein.

Triviale Lösung des Lernens: Die Pfade im Baum sind genau die Beispiele. *Nachteil:* Neue Fälle können nicht behandelt werden.

*Nimm den einfachsten Baum (bzw. den allgemeinsten) der mit den Beispielen verträglich ist.*

**Ockham's Messer:** *Entia non sunt multiplicanda praeter necessitatem.*

**Beispiel 7.2.2 (Raten Sie!)**
*Ein Computerprogramm kodiert Zahlentripel gemäß einer bestimmten Vorschrift. Sie sollen diese Vorschrift herausfinden.*

*Sie geben dem Programm Tripel* $(x_1, x_2, x_3)$ *Ihrer Wahl vor (*$X_i \in \mathbb{N}$*) und bekommen als Antwort "Ja" oder "Nein".*

*Zur Vereinfachung gibt Ihnen das Programm schon zu Anfang bekannt, daß folgende Tripel in der Menge enthalten sind:*

$$(4, 6, 8),\ (6, 8, 12),\ (20, 22, 40)$$

*Stellen Sie weitere Anfragen (etwa 10) und versuchen Sie die Eigenschaft herauszufinden.*

Das Problem ist, daß man nur Tripel der Form

$$(x, x + 2, 2x)$$

ausprobiert, also nur positive Beispiele versucht. Man ist zu speziell und versucht nicht, die einmal gefasste Hypothese kritisch zu hinterfragen, also sie zu falsifizieren. Man fragt nach einer speziellen Vorschrift, nicht nach der allgemeinsten, wie z.B. $\{(x, y, z) : x < y < z\}$.

## Idee des Lern-Algorithmus

*Ziel:* Möglichst kleiner Baum. Teste zunächst die *wichtigsten* Attribute (um eine möglichst schnelle Klassifikation zu bekommen). *Dies formalisieren wir später, unter Rückgriff auf etwas Informationstheorie..* Fahre dann rekursiv so fort: d.h. es gibt immer weniger Beispiele und Attribute. Wir mache folgende Fallunterscheidungen:

**1.** *Es gibt positive und negative Beispiele.*
Wähle das beste Attribut.

**2.** *Es gibt nur positive oder nur negative Beispiele.*
Fertig: Eindeutige Antwort ist möglich.

**3.** *Es gibt keine Beispiele mehr.*
Dann muss man einen Default-Wert nehmen, etwa die Mehrheit der Beispiele des Elter-Knotens.

**4.** *Es gibt sowohl positive als auch negative Beispiele, aber es gibt keine Attribute mehr.*
Dann reichen die zugrundeliegenden Attribute eben nicht aus, um zu einer Entscheidung zu kommen. Es ist einfach nicht genügend Information gegeben.

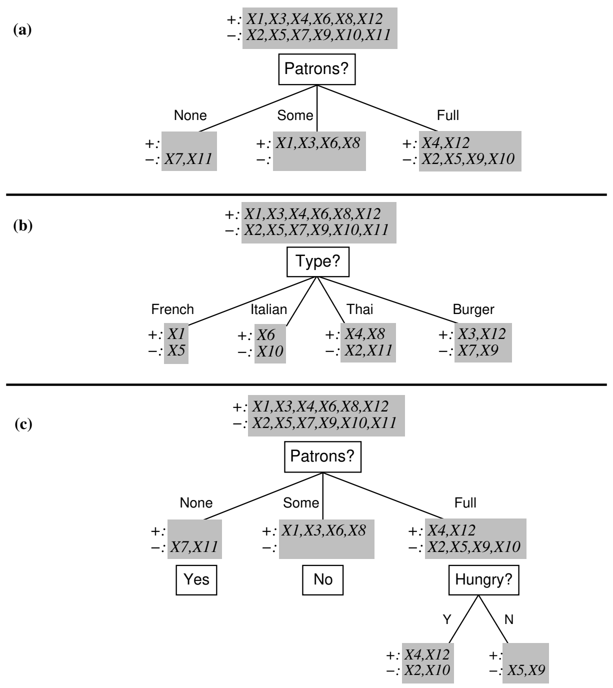
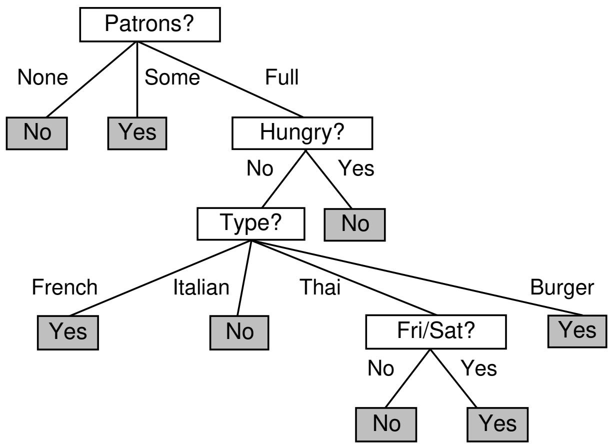



Abbildung 7.3: Auswahl der Attribute

```
function DECISION-TREE-LEARNING(examples, attributes, default) returns a decision tree
  inputs: examples, set of examples
          attributes, set of attributes
          default, default value for the goal predicate

  if examples is empty then return default
  else if all examples have the same classification then return the classification
  else if attributes is empty then return MAJORITY-VALUE(examples)
  else
     best ← CHOOSE-ATTRIBUTE(attributes, examples)
     tree ← a new decision tree with root test best
     for each value v_i of best do
        examples_i ← {elements of examples with best = v_i}
        subtree ← DECISION-TREE-LEARNING(examples_i, attributes − best,
                                          MAJORITY-VALUE(examples))
        add a branch to tree with label v_i and subtree subtree
     end
     return tree
```

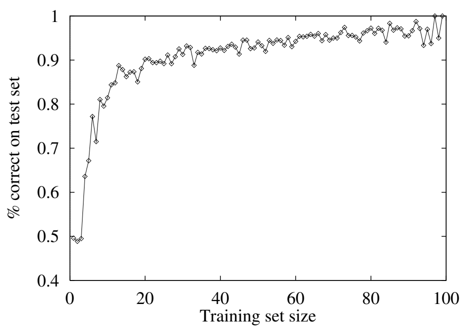
Tabelle 7.1: Lernen von Entscheidungsbäumen.




Der Algorithmus entwirft einen möglichst einfachen Baum, der mit den gegebenen Beispielen konsistent ist. *Wie gut ist der generierte Baum?* Wie weit ist er von dem "tatsächlichen" Baum entfernt? Kann man a-priori Abschätzungen geben? ($\leadsto$ Abschnitt 7.4).

## Empirisches Vorgehen

1. Wähle eine Menge $M_{Bsp}$ von Beispielen aus.
2. Teile diese in 2 Mengen auf: $M_{Bsp} = M_{Trai} \cup M_{Test}$.
3. Wende Lernalgorithmus auf $M_{Trai}$ an und erhalte eine Hypothese H.
4. Bestimme den Anteil der Elemente aus $M_{Test}$ die korrekt klassifiziert werden.
5. Wiederhole 1.-4. für viele $M_{Trai} \cup M_{Test}$ mit zufällig erzeugten $M_{Trai}$.



## Informationstheorie

Wie wählt man das "beste" Attribut aus? Das beste Attribut ist dasjenige, das die größtmögliche Information liefert. Als Beispiel betrachten wir den fairen Münzwurf.

**Definition 7.2.3 (1 Bit, Information)**
*1 Bit ist die Information die man braucht, um bei einem fairen Münzwurf das richtige Ergebnis anzusagen.*

*Allgemeiner: Angenommen, ein Experiment kann $n$-Ausgänge $v_1, \ldots, v_n$ haben. Jeder Ausgang $v_i$ ergibt sich mit einer Wahrscheinlichkeit $P(v_i)$. Die Information die im Ergebnis (dem ausführen des Experimentes) steckt, ist definiert als*

$$\mathbf{I}(P(v_1), \ldots, P(v_n)) := \sum_{i=1}^{n} -P(v_i) log_2 P(v_i)$$

Angenommen eine Münze ist präpariert. Zu 90 % kommt Kopf. Dann ist

$$\mathbf{I}(0.1, 0.9) = \ldots \approx 0.47$$

*Für jedes Attribut $A$ ist die folgende Frage wichtig:* Wenn man diese Attribut auf der momentanen Trainigsmenge auswertet, wieviel Information gewinnt man dadurch? Das "beste" Attribut ist das mit der größtmöglichen Information!

Angenommen $A$ hat $\nu$ verschiedene Ausgänge (Patron: 3, Hungry: 2, Type: 4) und teilt $M_{Trai}$ in $E_1, \ldots, E_\nu$ auf. Jedes $E_i$ enthalte $p_i$ positive und $n_i$ negative Beispiele. In jedem Zweig brauchen wir also noch

$$\mathbf{I}(\frac{p_i}{p_i + n_i}, \frac{n_i}{p_i + n_i})$$

Information um jedes Beispiel eindeutig zu klassifizieren.

Ein Beispiel ist in $E_i$ mit der Wahrscheinlichkeit $\frac{p_i+n_i}{p+n}$ (wir nehmen an, daß $M_{Trai}$ genau $p$ positive und $n$ negative Beispiele enthält).

Die Information um jedes Beispiel aus $M_{Trai} = \bigcup_i^\nu E_i$ zu klassifizieren ist

$$\text{Fehlende\_Inf}(A) = \sum_{i=1}^{\nu} \frac{p_i + n_i}{p + n} \mathbf{I}(\frac{p_i}{p_i + n_i}, \frac{n_i}{p_i + n_i})$$

Also gewinnen wir durch den Test des Attributes $A$ folgende Information

$$\text{Gewinn}(A) = \mathbf{I}(\frac{p}{p + n}, \frac{n}{p + n}) - \text{Fehlende\_Inf}(A)$$

**Choose_Attribute wählt das A aus mit maximalem Gewinn(A).**
Aus Abb. 18.6 ergibt sich Gewinn(Patron)$\approx 0.54$. Berechnen Sie Gewinn(Type), Gewinn(Hungry) (Hungry als erstes Attribut), Gewinn(Hungry) (wenn vorher Patron ausgewählt wurde).

## 7.3 PL1 formalisierte Bereiche

**Ziel:** Allgemeinerer Rahmen.

**Idee:** Lernen ist Suchen im Hypothesen-Raum. ($\rightsquigarrow$ Kapitel 6)

**Zielprädikat:** $Q(x)$, einstellig (bisher: $Will\_Wait$)

Wir suchen eine Definition von $Q(x)$, d.h. eine Formel $C(x)$ mit

$$\forall x\, Q(x) \leftrightarrow C(x)$$

Jedes Beispiel $X_i$ stellt eine Menge von Bedingungen dar, unter denen $Q(X_i)$ gilt oder nicht. Wir suchen eine Erklärung dafür: eine Formel $C(x)$, die die in den Beispielen auftretenden Prädikate benutzt. Z.B.:

$$\begin{array}{lllll} \forall r & Will\_Wait(r) & \leftrightarrow & & Patrons(r, Some) \\ & & & \lor & (Patrons(r, Full) \land \neg Hungry(r) \land Type(r, French) \\ & & & \lor & (Patrons(r, Full) \land \neg Hungry(r) \land Type(r, Thai) \land Fri\_Sat(r)) \\ & & & \lor & (Patrons(r, Full) \land \neg Hungry(r) \land Type(r, Burger) \end{array}$$

**Definition 7.3.1 (Hypothese, Kandidatenfunktion)**
*Eine Formel $C_i(x)$ mit $\forall x C(x) \leftrightarrow C_i(x)$ heißt Kandidatenfunktion. Die gesamte Formel heißt Hypothese:*

$$H_i: \ \forall x C(x) \leftrightarrow C_i(x)$$

**Definition 7.3.2 (Hypothesenraum $\mathcal{H}$)**
*Der Raum aller Hypothesen $\mathcal{H}$ eines Lern-Alg. ist die Menge all jener Hypothesen, die der Algorithmus bilden kann.*

*Unter der Ausprägung einer Hypothese $H$ bezüglich des Zielprädikates $Q$ verstehen wir die Menge der Beispiele, Die $H$ erfüllen (sie bestätigen $H$).*

Hypothesen mit unterschiedlicher Ausprägung sind zusammen inkonsistent. Hypothesen mit gleicher Ausprägung sind logisch äquivalent.

### Allgemeine Situation

- Wir haben eine Menge von Beispielen $\{X_1, \ldots, X_n\}$. Jedes Beispiel $X$ beschreiben wir durch einen Satz und die Angabe $Q(x)$ (trifft zu) oder $\neg Q(x)$ (trifft nicht zu).

| Example | Attributes | | | | | | | | | | Goal |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | *Alt* | *Bar* | *Fri* | *Hun* | *Pat* | *Price* | *Rain* | *Res* | *Type* | *Est* | *WillWait* |
| $X_1$ | *Yes* | *No* | *No* | *Yes* | *Some* | *$$$* | *No* | *Yes* | *French* | *0–10* | *Yes* |
| $X_2$ | *Yes* | *No* | *No* | *Yes* | *Full* | *$* | *No* | *No* | *Thai* | *30–60* | *No* |
| $X_3$ | *No* | *Yes* | *No* | *No* | *Some* | *$* | *No* | *No* | *Burger* | *0–10* | *Yes* |
| $X_4$ | *Yes* | *No* | *Yes* | *Yes* | *Full* | *$* | *No* | *No* | *Thai* | *10–30* | *Yes* |
| $X_5$ | *Yes* | *No* | *Yes* | *No* | *Full* | *$$$* | *No* | *Yes* | *French* | *>60* | *No* |
| $X_6$ | *No* | *Yes* | *No* | *Yes* | *Some* | *$$* | *Yes* | *Yes* | *Italian* | *0–10* | *Yes* |
| $X_7$ | *No* | *Yes* | *No* | *No* | *None* | *$* | *Yes* | *No* | *Burger* | *0–10* | *No* |
| $X_8$ | *No* | *No* | *No* | *Yes* | *Some* | *$$* | *Yes* | *Yes* | *Thai* | *0–10* | *Yes* |
| $X_9$ | *No* | *Yes* | *Yes* | *No* | *Full* | *$* | *Yes* | *No* | *Burger* | *>60* | *No* |
| $X_{10}$ | *Yes* | *Yes* | *Yes* | *Yes* | *Full* | *$$$* | *No* | *Yes* | *Italian* | *10–30* | *No* |
| $X_{11}$ | *No* | *No* | *No* | *No* | *None* | *$* | *No* | *No* | *Thai* | *0–10* | *No* |
| $X_{12}$ | *Yes* | *Yes* | *Yes* | *Yes* | *Full* | *$* | *No* | *No* | *Burger* | *30–60* | *Yes* |

$X_1$ ist beschrieben durch

$$Alt(X_1) \wedge \neg Bar(X_1) \wedge \neg Fri\_Sat(X_1) \wedge Hungry(X_1) \wedge \ldots$$

und $Will\_Wait(X_1)$. $\mathcal{H}$ ist die Menge der Entscheidungsbäume.

- Die Trainingsmenge ist die Menge all dieser Sätze.
- *Wir suchen eine Hypothese, die konsistent mit der Trainingsmenge ist.*

Unter welchen Bedingungen ist eine Hypothese inkonsistent mit einem Beispiel $X$?

**Falsch negativ:** Hypothese sagt nein ($\neg Q(X)$) aber in Wirklichkeit gilt '$Q(X)$.

**Falsch positiv:** Hypothese sagt ja ($(X)$) aber in Wirklichkeit gilt '$\neg Q(X)$.

Induktives Lernen in logik-basierten Bereichen heißt mit jedem Beispiel die Menge der möglichen Hypothesen weiter einschränken.

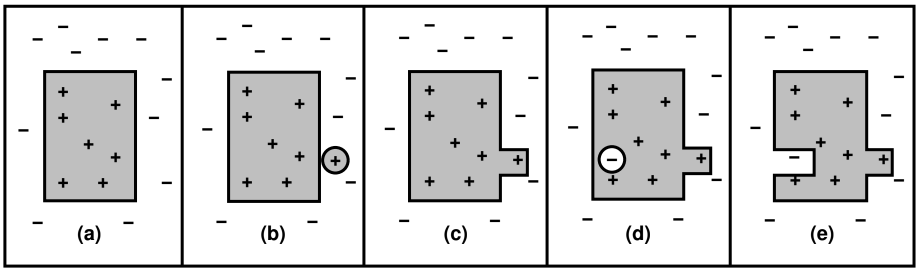
Für PL1 ist $\mathcal{H}$ im allgemeinen unendlich, daher ist automatisches Beweisen viel zu allgemein.

**1. Ansatz:** Wir behalten eine einzige Hypothese und modifizieren sie, falls die Beispiele mit ihr inkonsistent sind.

**2. Ansatz:** Wir behalten den ganzen Teilraum der noch mit den Beispielen konsistent ist (*Version-Space*. Dieser wird durch 2 Mengen effizient repräsentiert (so wie man mit [1,2] das entsprechende Intervall aller reellen Zahlen repräsentiert).

## 1. Ansatz: Current-Best-Hypothesis-Suche

Starte mit einfacher Hypothese $H$. Falls ein neues Beispiel damit konsistent: ok. Falls es "falsch negativ": Verallgemeinere $H$. Falls es "falsch positiv": Spezialisiere $H$.




Das ergibt einen Algorithmus:

```
function CURRENT-BEST-LEARNING(examples) returns a hypothesis

  H ← any hypothesis consistent with the first example in examples
  for each remaining example in examples do
     if e is false positive for H then
        H ← choose a specialization of H consistent with examples
     else if e is false negative for H then
        H ← choose a generalization of H consistent with examples
     if no consistent specialization/generalization can be found then fail
  end
  return H
```

Wie verallgemeinert, bzw. spezialisiert man?

$$H_1 : \quad \forall x \; (Q(x) \leftrightarrow C_1(x))$$
$$H_2 : \quad \forall x \; (Q(x) \leftrightarrow C_2(x))$$

$H_1$ verallgemeinert $H_2$, falls $C_2(x) \rightarrow C_1(x)$, $H_1$ spezialisiert $H_2$, falls $C_1(x) \rightarrow C_2(x)$. Verallgemeinern bedeutet: in einer Konjunktion lässt man $\wedge$-Glieder weg. Spezialisieren bedeutet: in einer Disjunktion nimmt man $\vee$-Glieder hinzu.

## 2. Ansatz: Version-Space

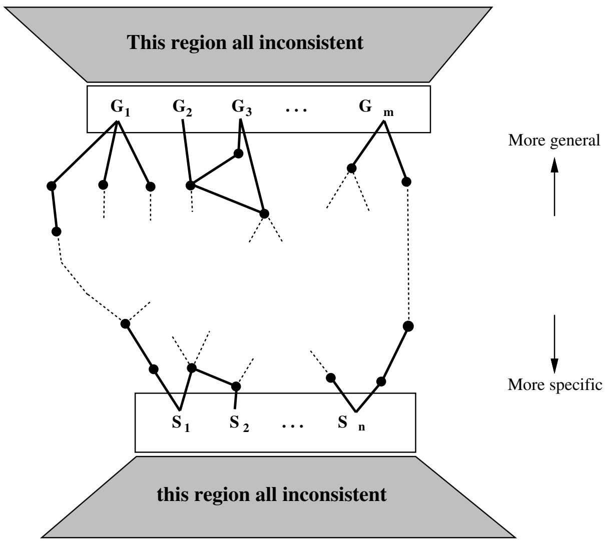
```
function VERSION-SPACE-LEARNING(examples) returns a version space
  local variables: V, the version space: the set of all hypotheses

  V ← the set of all hypotheses
  for each example e in examples do
     if V is not empty then V ← VERSION-SPACE-UPDATE(V, e)
  end
  return V
------------------------------------------------------------------
function VERSION-SPACE-UPDATE(V, e) returns an updated version space
  V ← {h ∈ V : h is consistent with e}
```

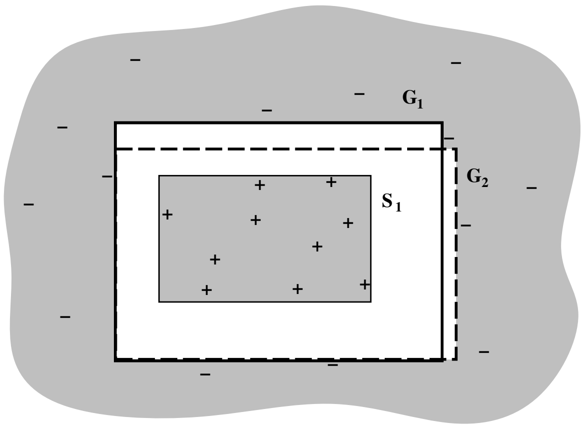
*$\mathcal{H}$ ist eine große Disjunktion $H_1 \vee \ldots \vee H_n$. Wie soll man das repräsentieren?*

Wie repräsentiert man die Menge aller reellen Zahlen zwischen 0 und 1? Als Intervall [0,1]. In unserem Fall gibt es eine partielle Ordnung auf $\mathcal{H}$ (Verallg./Spezial.). Die Grenzen sind gegeben durch

- *G-Menge*: ist konsistent mit allen bisherigen Beispielen und es gibt keine konsistente Hypothese die allgemeiner ist,
- *S-Menge*: ist konsistent mit allen bisherigen Beispielen und es gibt keine konsistente Hypothese die spezifischer ist.

Werden neue Beispiele betrachtet, müssen die G- und S-Mengen in VERSION-SPACE-UPDATE geeignet modifiziert werden.




## Was kann passieren?

1. Es bleibt nur eine Hypothese übrig: Hurra!

2. Der Raum kollabiert: $G = \emptyset$ oder $S = \emptyset$.

3. Keine Beispiele mehr, aber mehrere Hypothesen. D.h. wir enden mit einer grossen Disjunktion.

## 7.4 Warum funktioniert das alles?

*Wie nah ist eine aus dem Lern-Algorithmus berechnete Hypothese $h$ von der richtigen Funktion $f$ entfernt?* Das führt zur *Computational Learning Theory*: **PAC**-Lernen – **P**robably **A**pproximately **C**orrect. Die Grundidee dabei ist: *Ist eine Hypothese konsistent mit einer großen Trainingsmenge, so kann sie nicht völlig falsch sein.*

Wie verhalten sich Trainings und Test-Menge zueinander? Wir machen folgende Annahme:

*Trainings- und Testmenge sind gemäß derselben Wahrscheinlichkeitsverteilung aus der Menge $X$ aller Beispiele entnommen.*

**Definition 7.4.1 (error(h), $\epsilon$-approximativ korrekt)**
*Sei $h \in \mathcal{H}$ eine Hypothese und $f$ die richtige (zu lernende) Funktion. Interessant ist die Menge*

$$Diff(f, h) := \{x : h(x) \neq f(x)\}.$$

*Wir bezeichnen mit $error(h)$ die Wahrscheinlichkeit, daß ein beliebiges Beispiel aus $X$ in der Menge $Diff(f, h)$ liegt.*

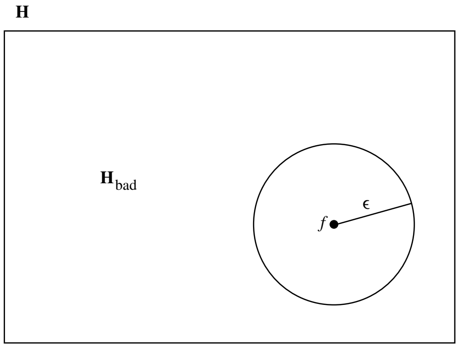
*Für $\epsilon > 0$ heißt die Hypothese $h$ $\epsilon$-approximativ korrekt, falls $error(h) \leq \epsilon$.*

*Sei $\epsilon > 0$ gegeben. Wieviele Beispiele muß die Trainingsmenge enthalten, damit die durch einen Lern.-Alg. erzeugte Hypothese $\epsilon$-approximativ korrekt ist?*

**Definition 7.4.2 (Beispielkomplexität)**
*Gegeben $\delta > 0$ und $\epsilon > 0$. Unter der Beispielkomplexität verstehen wir die Zahl $m$ von Beispielen die ein beliebiger Lern-Algorithmus braucht, damit die von ihm erzeugte Hypothese $h$ mit mindestens Wahrscheinlichkeit $1-\delta$ jedenfalls $\epsilon$-approximativ korrekt ist.*

**Theorem 7.4.3**
*Die Beispielkomplexität $m$ hängt in folgender Weise von $\epsilon, \delta$ und dem Hypothesenraum $\mathcal{H}$ ab:*

$$m \geq \frac{1}{\epsilon}(ln\frac{1}{\delta} + ln|\mathcal{H}|)$$




Was bedeutet das letzte Ergebnis? Komplexität hängt von $ln(|\mathcal{H}|)$ ab. Für boolesche Funktionen in $n$-Attributen ist

$$ln(|\mathcal{H}|) = 2^n$$

also braucht man exponentiell viele Beispiele selbst um nur mit einer Wahrscheinlichkeit $\epsilon$-approximativ korrekt zu sein! Also:

- Verschiedene Mengen von Beispielen ergeben verschiedene Aussagen und damit Werte von $\epsilon$. Es kommt also nicht nur darauf an, *wieviele* sondern auch *welche* Beispiele man auswählt.

  Wir relativieren darum unsere Frage und wollen nur noch fordern, daß die Hypothese mit einer Wahrscheinlichkeit von $1 - \delta$, wobei $\delta > 0$,$\epsilon$-approximativ korrekt ist.

- Wir möchten auch von speziellen Lern-Algorithmen abstrahieren und eine Aussage über alle möglichen Lern-Algorithmen machen. Wir gehen daher davon aus, daß jeder Lern-Alg. jedenfalls eine Hypothese berechnet, die mit den bisherigen Beispielen konsistent ist.

**Beweis:** Sei $h_b \in \mathcal{H}$ mit $error(h_b) > \epsilon$.

- Wie groß ist die Wahrscheinlichkeit $P_b$, daß $h_b$ konsistent mit $m$ gewählten Beispielen ist? Für ein Beispiel ist es gerade $\leq (1 - \epsilon)$ wegen der Definition von $error$. Also:

$$P_b \leq (1 - \epsilon)^m$$

- Wie groß ist die Wahrscheinlichkeit $P'$, daß es überhaupt eine Hypothese $h_b$ mit $error(h_b) > \epsilon$ gibt, die mit $m$ Beispielen konsistent ist?

$$\begin{array}{rcl} P' & \leq & \sum_{Hypothesen}(1-\epsilon)^m \\ & \leq & |\{h : error(h) > \epsilon\}|(1-\epsilon)^m \\ & \leq & |\mathcal{H}|(1-\epsilon)^m \end{array}$$

- Wir wollen, daß $P' \leq \delta$:

  $$|\mathcal{H}|(1-\epsilon)^m \leq \delta$$

  Einges umformen ergibt:

  $$\begin{array}{lcl} (1-\epsilon)^m & \leq & \frac{\delta}{|\mathcal{H}|} \\ mln(1-\epsilon) & \leq & ln(\delta) - ln(|\mathcal{H}|) \\ m & \geq & -\frac{1}{ln(1-\epsilon)}(ln(\frac{1}{\delta}) + ln(|\mathcal{H}|)) \\ m & \geq & frac1\epsilon(ln(\frac{1}{\delta}) + ln(|\mathcal{H}|)) \end{array}$$

  Die letzte Zeile gilt wegen $ln(1-\epsilon) < -\epsilon$.

■

das Theorem kann man auch so ausdrücken:

*Lernen ist nie besser, als in einer Tabelle nachzuschlagen!*

**1. Ausweg:** Wir fordern, daß nicht eine beliebige konsistente Hypothese generiert wird, sondern möglichst spezielle. (Das ist aber dann von der Komplexität noch schlechter.)

**2. Ausweg:** Wir geben es auf, beliebige boolesche Funktionen zu lernen und beschränken uns auf geeignete Teilklassen.

So können wir etwa **Entscheidungslisten** betrachten.

**Theorem 7.4.4 (E-Listen werden in polynomialer Zeit gelernt)**
*Jede Funktion aus $k-DL(n)$ (E-Listen mit höchstens k-Tests) haben eine PAC-Komplexität von*

$$m = \frac{1}{\epsilon}(ln(\frac{1}{\delta}) + O(n^k log_2(n^k)))$$

Mit anderen Worten: Jeder Algorithmus der für eine Menge von Beispielen eine konsistente Entscheidungsliste zurückgibt, kann in einen PAC-Lern-Algorithmus verwandelt werden, der eine $k-DL(n)$ Funktion nach höchstens $m$-Beispielen lernt.

Folgende Abschätzungen sind wichtig:

- $|Conj(n,k)| = \sum_{i=0}^{k} \binom{2n}{i} = \mathrm{O}(n^k)$,
- $|k-DL(n)| \leq 3^{|Conj(n,k)|}|Conj(n,k)\mathbf{!}$,
- $|k-DL(n)| \leq 2^{\mathrm{O}(n^k log_2(n^k))}$.

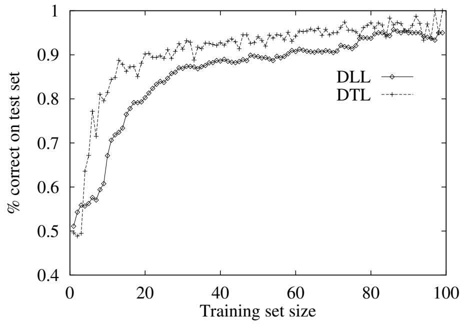



Abbildung 7.4: Vergleich E-Listen und E-Bäume

**function** DECISION-LIST-LEARNING(*examples*) **returns** a decision list, *No* or failure

**if** *examples* is empty **then return** the value *No*
t ← a test that matches a nonempty subset $examples_t$ of *examples*
such that the members of $examples_t$ are all positive or all negative
**if** there is no such *t* **then return** failure
**if** the examples in $examples_t$ are positive **then** $o \leftarrow Yes$
**else** $o \leftarrow No$
**return** a decision list with initial test *t* and outcome *o*
and remaining elements given by DECISION-LIST-LEARNING($examples - examples_t$)

## 7.5 Rauschen und Overfitting

**Rauschen:**

- Beispiele sind inkonsistent (einmal $Q(x)$, dann $\neg Q(x)$).
- Keine Attribute mehr um Beispiele zu klassifizieren.
- Macht Sinn, wenn die Umgebung nichtdeterministisch

**Overfitting:** Dual zum Rauschen: überbestimmt.

- Verbleibende Beispiele können mittels Attributen klassifiziert werden, die eine *Ordnung* ergeben, welche nicht vorhanden ist (irrelevante Attribute). *Beispiel:* Mehrere farbige Würfel werden geworfen. Jeder Wurf wird durch (Tag, Monat, Uhrzeit, Farbe) beschrieben. Solnge es keine Inkonsistenz gibt, ist jeder Wurf einzig und durch eine völlig überbestimmte Hypothese beschrieben.

Andere Beispiele:

- Pyramiden,
- Astrologie,
- “Mein magisches Fahrrad”.

## 7.6 Aufgaben

**Aufgabe 7.1 (Entscheidungsbäume)**
*Wir haben es zugelassen, daß die Attribute in einem Entscheidungsbaum auch mehr als zwei Werte annehmen können.*

1. *Zeigen Sie, daß man jeden Entscheidungsbaum in einen binären Entscheidungsbaum mit gleicher Funktionalität transformieren kann, der nur noch Boole'sche Attribute enthält.*
2. *Wie verändert sich die Anzahl der Attribute und die Tiefe des Baumes?*
3. *Gibt es Unterschiede zwischen binären Entscheidungsbäumen und Entscheidungslisten?*

**Aufgabe 7.2 (Entscheidungslisten)**
*Zeigen Sie, daß Entscheidungslisten*

1. *jede beliebige Boole'sche Funktion darstellen können, falls die Tests keiner Größenbeschränkung unterliegen, bzw.*
2. *Entscheidungsbäume der Tiefe $k$ simulieren können, falls alle Tests höchstens $k$ Literale enthalten.*

**Aufgabe 7.3 (Fehlerabschätzungen)**
*Bei der rekursiven Konstruktion eines Entscheidungsbaumes kann es passieren, daß an einem Blatt eine gemischte Menge von Beispielen übrigbleibt, d.h. $p > 0$ positive und $n > 0$ negative Beispiele. Trotzdem muß der Algorithmus eine Entscheidung treffen, obwohl kein Attribut mehr zur Verfügung steht.*

1. *Zeigen Sie, daß der* DECISION-TREE-LEARNING-*Algorithmus, der sich in diesem Fall nach der Mehrheit der Beispiele entscheidet, den absoluten Fehler minimiert.*
2. *In der Vorlesung wurde ein modifizierter Algorithmus besprochen, der nur eine Wahrscheinlichkeit zurückgibt, nämlich $p/(p+n)$. Zeigen Sie, daß diese Vorgehensweise die Summe der Fehler-Quadrate minimiert.*

**Aufgabe 7.4 (Zahlen lernen)**
*Zu lernen ist die Eigenschaft natürlicher Zahlen, daß sie durch 6 teilbar sind. Als Lernbeispiele sind die natürlichen Zahlen von 1 bis 60 gegeben. Die Reste bei der Division durch 2, 3, 4 bzw. 5 stehen als Attribute zur Verfügung.*

1. *Berechnen Sie den Informationsgewinn der vier Attribute!*

2. *Wie sieht ein Entscheidungsbaum mit möglichst wenig Knoten aus?*

**Aufgabe 7.5 (Wahrscheinlichkeiten)**
*Ein Lern-Algorithmus suche eine mit den gegebenen Daten konsistente Hypothese. Die Zuordnung der Beispiele zu den Klassen „positiv" und „negativ" erfolge jedoch vollkommen zufällig. Die Beispiele haben $n$ binäre Merkmale.*

*Sei nun eine Menge von $x$ Beispielen gleichwahrscheinlich zufällig aus den $2^n$ möglichen ausgewählt. Schätzen Sie die Anzahl von Beispielen $x$ ab, die genommen werden muß, so daß die Wahrscheinlichkeit, daß eine Inkonsistenz auftritt, gerade $0.5$ ist.*

**Aufgabe 7.6 (Informationsgewinn)**
*Ein Attribut $A$ unterteile eine Menge von Beispielen $E$ in Teilmengen $E_i$, wobei jede Teilmenge $p_i$ positive und $n_i$ negative Beispiele enthalte. Zeigen Sie, daß $\text{GEWINN}(A) > 0$ ist, außer wenn $p_i/(p_i + n_i)$ für alle $i$ gleich ist.*

# 8 Knowledge Representation with Non-classical Logic

One of the major reasons for the success story (if one is really willing to call it a success story) of human beings on this planet is our ability to invent tools that help us improve our — otherwise often quite limited — capabilities. The invention of machines that are able to do interesting things, like transporting people from one place to the other (even through the air), sending moving pictures and sounds around the globe, bringing our email to the right person, and the like, is one of the cornerstones of our culture and determines to a great degree our everyday life.

Among the most challenging tools one can think of are machines that are able to handle knowledge adequately. Wouldn't it be great if, instead of the stupid device which brings coffee from the kitchen to your office every day at 9.00, and which needs complete reengineering whenever your coffee preferences change, you could (for the same price, admitted) get a smart robot whom you can simply tell that you want your coffee black this morning, and that you need an extra Aspirin since it was your colleague's birthday yesterday? To react in the right way to your needs such a robot would have to know a lot, for instance that Aspirin should come with a glass of water, or that people in certain situations need their coffee extra strong.

Building smart machines of this kind is at the heart of Artificial Intelligence (AI). Since such machines will need tremendous amounts of knowledge to work properly, even in very limited environments, the investigation of techniques for representing knowledge and reasoning is highly important.

In the early days of AI it was still believed that modeling general purpose problem solving capabilites, as in Newell and Simon's famous GPS (General Problem Solver) program, would be sufficient to generate intelligent behaviour. This hypothesis, however, turned out to be overly optimistic. At the end of the sixties people realized that an approach using available knowledge about narrow domains was much more fruitful. This led to the expert systems boom which produced many useful application systems, expert system building tools, and expert system companies. Many of the systems are still in use and save companies millions of dollars per year[1].

Nevertheless, the simple knowledge representation and reasoning methods underlying the early expert systems soon turned out to be insufficient. Most of the systems

[1] We refer the interested reader to the recent book [RN95] which gives a very detailed and nice exposition of what has been done in AI since its very beginning until today.

were built based on simple rule languages, often enhanced with ad hoc approaches to model uncertainty. It became apparent that more advanced methods to handle incompleteness, defeasible reasoning, uncertainty, causality and the like were needed.

This insight led to a tremendous increase of research on the foundations of knowledge representation and reasoning. Theoretical research in this area has blossomed in recent years. Many advances have been made and important results were obtained. The technical quality of this work is often impressive.

On the other hand, most of these advanced techniques have had surprisingly little influence on practical applications so far. To a certain degree this is understandable since theoretical foundations had to be laid first and pioneering work was needed. However, if we do not want research in knowledge representation to remain a theoreticians' game more emphasis on computability and applicability seems to be needed. We strongly believe that the kind of research presented in this tutorial, that is research aiming at interesting combinations of ideas from logic programming and nonmonotonic reasoning, provides an important step into this direction.

## 8.1 Some History

Historically, logic programs have been considered in the logic programming community for more than 20 years. It began with [CKPR73, Kow74, vEK76] and led to the definition and implementation of *PROLOG*, a by now theoretically well-understood programming language (at least the declarative part consisting of Horn-clauses: *pure PROLOG*). Extensions of PROLOG allowing negative literals have been also considered in this area: they rely on the idea of *negation-as-finite-failure*, we call them *Logic-Programming-semantics* (or shortly *LP-semantics*).

In parallel, starting at about 1980, *Nonmonotonic Reasoning* entered into computer science and began to constitute a new field of active research. It was originally initiated because *Knowledge Representation* and *Common-Sense Reasoning* using classical logic came to its limits. Formalisms like classical logic are inherently monotonic and they seem to be too weak and therefore inadequate for such reasoning problems.

In recent years, independently of the research in logic programming, people interested in knowledge representation and nonmonotonic reasoning also tried to define declarative semantics for programs containing *default* or *explicit* negation and even *disjunctions*. They defined various semantics by appealing to (different) intuitions they had about programs.

This second line of research started in 1986 with the *Workshop on the Foundations of Deductive Databases and logic programming* organized by Jack Minker: the revised papers of the proceedings were published in [Min88]. The *stratified* (or the similar *perfect*) semantics presented there can be seen as a splitting-point: it is still of interest for the logic programming community (see [CL89]) but its underlying intuitions were inspired by nonmonotonic reasoning and therefore much more suitable for knowledge representation tasks. Semantics of this kind leave the philosophy underlying classical logic programming in that their primary aim is not to model *negation-as-finite-failure*, but to construct new, more powerful semantics suitable for applications in knowledge representation. Let us call such semantics *NMR-semantics*.

Nowadays, due to the work of Apt, Blair and Walker, Fitting, Lifschitz, Przymusinski and others, very close relationships between these two independent research lines became evident. Methods from logic programming, e.g. least fixpoints of certain operators, can be used successfully to define NMR-semantics.

The NMR-semantics also shed new light on the understanding of the classical nonmonotonic logics such as *Default Logic*, *Autoepistemic Logic* and the various versions of *Circumscription*. In addition, the investigation of possible semantics for logic programs seems to be useful because

1. parts of nonmonotonic systems (which are usually defined for full predicate logic, or even contain additional (modal)-operators) may be "implemented" with the help of such programs,

2. nonmonotonicity in these logics may be described with an appropriate treatment of negation in logic programs.

## 8.2 Non-Monotonic Formalisms in KR

As already mentioned above, research in nonmonotonic reasoning has begun at the end of the seventies. One of the major motivations came from reasoning about actions and events. John McCarthy and Patrick Hayes had proposed their situation calculus as a means of representing changing environments in logic. The basic idea is to use an extra situation argument for each fact which describes the situation in which the fact holds. Situations, basically, are the results of performing sequences of actions. It soon turned out that the problem was not so much to represent what changes but *to represent what does not change* when an event occurs. This is the so-called *frame problem*. The idea was to handle the frame problem by using a default rule of the form

> *If a property $P$ holds in situation $S$ then $P$ typically also holds in the situation obtained by performing action $A$ in $S$.*

Given such a rule it is only necessary to explicitly describe the changes induced by a particular action. All non-changes, for instance that the *real colour* of the kitchen wall does not change when the light is turned on, are handled implicitly. Although it turned out that a straightforward formulation of this rule in some of the most popular nonmonotonic formalisms may lead to unintended results the frame problem was certainly the challenge motivating many people to join the field.

In the meantime a large number of different nonmonotonic logics have been proposed. We can distinguish four major types of such logics:

1. Logics using nonstandard inference rules with an additional consistency check to represent default rules. Reiter's default logic (see Appendix A.3) and its variants are of this type.

2. Nonmonotonic modal logics using a modal operator to represent consistency or (dis-) belief. These logics are nonmonotonic since conclusions may depend on disbelief. The most prominent example is Moore's autoepistemic logic.

3. Circumscription (see Appendix A.4) and its variants. These approaches are based on a preference relation on models. A formula is a consequence iff it is true in all most preferred models of the premises. Syntactically, a second order formula is used to eliminate all non-preferred models.

4. Conditional approaches which use a non truth-functional connective $\mid\!\sim$ to represent defaults. A particularly interesting way of using such conditionals was proposed by Kraus, Lehmann and Magidor. They consider $p$ as a default consequence of $q$ iff the conditional $q \mid\!\sim p$ is in the closure of a given conditional knowledge base under a collection of rules. Each of the rules directly corresponds to a desirable property of a nonmonotonic inference relation.

The various logics are intended to handle different intuitions about nonmonotonic reasoning in a most general way. On the other hand, the generality leads to problems, at least from the point of view of implementations and applications. In the first order case the approaches are not even semi-decidable since an implicit consistency check is needed. In the propositional case we still have tremendous complexity problems. For instance, the complexity of determining whether a formula is contained in all extensions of a propositional default theory is on the second level of the polynomial hierarchy. As mentioned earlier we believe that logic programming techniques can help to overcome these difficulties.

Originally, nonmonotonic reasoning was intended to provide us with a *fast* but *unsound* approximation of classical reasoning in the presence of incomplete knowledge. Therefore one might ask whether the higher complexity of NMR-formalisms (compared to classical logic) is not a real drawback of this aim? The answer is that NMR-systems allow us to formulate a problem in a very *compact* way as a theory $T$. It turns out that any equivalent formulation in classical logic (if possible at all) as a theory $T'$ is much larger: the size of $T'$ is exponential in the size of $T$! We refer to [GPSK95] and [CDS96, CDSS97, CDLS95] where such problems are investigated.

# 9 KR with Definite Logic Programs

In this section we consider the most restricted class of programs: *definite* logic programs, programs without any negation at all. All the extensions of this basic class we will introduce later contain at least some kind of negation (and perhaps additional features). But here we also allow the ocurrence of free variables as well as function symbols.

In Section 9.1 we introduce as a representative for the *Top-Down* approach the SLD-Resolution. Section 9.2 presents the main competing approach of SLD: *Bottom-Up Evaluation*. This approach is used in the Database community and it is efficient when additional assumptions are made (*finiteness-assumptions*, *no function symbols*). In Section 9.3 we consider the influence and appropriateness of Herbrand models and their underlying intuition. Finally in Section 9.4 we present and discuss two important examples in KR: *Reasoning in Inheritance Hierarchies* and *Reasoning about Actions*. Both examples clearly motivate the need of extending definite programs by a kind of *default-negation* "*not*".

First some notation used throughout this paper. A language $\mathcal{L}$ consists of a set of relation symbols and a set of function symbols (each symbol has an associated arity). Nullary functions are called constants. Terms and atoms are built from $\mathcal{L}$ in the usual way starting with variables, applying function symbols and relation-symbols.

Instead of considering arbitrary $\mathcal{L}$-formulae, our main object of interest is a program:

**Definition 9.0.1 (Definite Logic Program)**
*A* definite *logic program consists of a finite number of* rules *of the form*

$$A \leftarrow B_1, \ldots, B_m,$$

*where* $A, B_1, \ldots, B_m$ *are positive atoms (containing possibly free variables). We call* $A$ *the* head *of the rule and* $B_1, \ldots, B_m$ *its* body. *The comma represents conjunction* $\wedge$.

We can think of a program as formalizing our knowledge about the world and how the world behaves. Of course, we also want to derive new information, i.e. we want to ask queries:

**Definition 9.0.2 (Query)**
*Given a definite program we usually have a definite query in mind that we want to be solved. A definite query $Q$ is a conjunction of positive atoms $C_1 \wedge \ldots \wedge C_l$ which we denote by*

$$?\text{-}\ C_1, \ldots, C_l.$$

*These $C_i$ may also contain variables. Asking a query $Q$ to a program $P$ means asking for all possible substitutions $\Theta$ of the variables in $Q$ such that $Q\Theta$ follows from $P$. Often, $\Theta$ is also called an answer to $Q$. Note that $Q\Theta$ may still contain free variables.*

Note that if a program $P$ is given, we usually assume that it also determines the underlying language $\mathcal{L}$, denoted by $\mathcal{L}_P$, which is generated by exactly the symbols ocurring in $P$. The set of all these atoms is called the *Herbrand base* and denoted by $B_{\mathcal{L}_P}$ or simply $B_P$. The corresponding set of all ground terms is the *Herbrand universe*. Another important notion that we are not explaining in detail here is that of *Unification*. Given two atoms $A$ and $B$ with free variables we can ask if we can compute two substitutions $\Theta_1, \Theta_2$ for the variables such that

$$A\Theta_1 \ \text{ is identical to } \ B\Theta_2,$$

or if we can decide that this is not possible at all. In fact, if the two atoms are *unifiable* we can indeed compute a *most general unifier*, called mgU (see [Llo87]). The mgU $\Theta$ is a substitution defined on the set of variables occurring in both $A$ and $B$ such that $A\Theta$ is identical to $B\Theta$.

This will be important in our framework because if an atom appears as a subgoal in a query, we may want to determine if there are rules in the program whose heads unify with this atom.

How are our programs related to classical predicate logic? Of course, we can map a program-rule into classical logic by interpreting "$\leftarrow$" as material implication "$\supset$" and universally quantifying. This means we view such a rule as the following universally quantified formula

$$B_1 \wedge \ldots \wedge B_m \ \supset \ A.$$

However, as we will see later, there is a great difference: a logic program-rule takes some orientation with it. This makes it possible to formulate the following principle as an underlying intuition of all semantics of logic programs:

**Principle 9.0.3 (Orientation)**
*If a ground atom $A$ does not unify with some head of a program rule of $P$, then this atom is considered to be false. In this case we say that "* not *$A$" is derivable from $P$ to distinguish it from classical $\neg A$.*

The orientation principle is nothing but a weak form of *negation-by-failure*. Given an intermediate goal *not* $A$, we first try to prove $A$. But if $A$ does not unify with any head, $A$ fails and this is the reason to derive *not* $A$.

## 9.1 Top-Down

SLD-Resolution[1] is a special form of Robinson's general Resolution rule. While Robinson's rule is complete for full first order logic, SLD is complete for definite logic programs (see Theorem 9.1.1). We do not give a complete definition of SLD-Resolution (see [Llo87]) but rather prefer to illustrate its behaviour on the following example.

**Example 9.1.1 (SLD-Resolution)**
*Let the program $P_{SLD}$ consist of the following three clauses*

$$\begin{array}{lll} (1) & p(x,z) & \leftarrow \quad q(x,y), p(y,z) \\ (2) & p(x,x) & \\ (3) & q(a,b) & \end{array}$$

*The query $Q$ we are interested in is given by $p(x,b)$. I.e. we are looking for all substitutions $\Theta$ for $x$ such that $p(x,b)\Theta$ follows from $P$.*

Figure 9.1 illustrates the behaviour of SLD-resolution. We start with our query in the form $\leftarrow Q$. Sometimes the notation $\Box \leftarrow Q$ is also used, where $\Box$ denotes the falsum. In any round the selected atom is underlined: numbers 1, 2 or 3 indicate the number of the clause which the selected atom is resolved against. Obviously, there are three different sorts of branches, namely

1. *infinite* branches,
2. branches that *end up with the empty clause*, and
3. branches that *end in a deadlock ("Failure")*: no applicable rule is left.

In this example we always resolve with the *last* atom in the goal under consideration. If we choose always the *first* atom in the goal, we will obtain, at least in this example, a *finite* tree.

Definite programs have the nice feature that the intersection of all Herbrand-models exists and is again a Herbrand model of $P$. It is denoted by $M_P$ and called the least Herbrand-model of $P$. Note that our original aim was to find substitutions $\Theta$ such that $Q\Theta$ is derivable from the program $P$. This task as well as $M_P$ is closely related to SLD:

**Theorem 9.1.1 (Soundness and Completeness of SLD)**
*The following properties are equivalent:*

- $P \models \forall\, Q\Theta$, *i.e.* $\forall\, Q\Theta$ *is true in all models of* $P$,
- $M_P \models \forall\, Q\Theta$,
- *SLD computes an answer $\tau$ that subsumes*[2] $\Theta$ *wrt* $Q$.

[1] **SL**-resolution for **D**efinite clauses. **SL**-resolution stands for **L**inear resolution with **S**election function.
[2] i.e. $\exists \sigma : Q\tau\sigma = Q\Theta$.

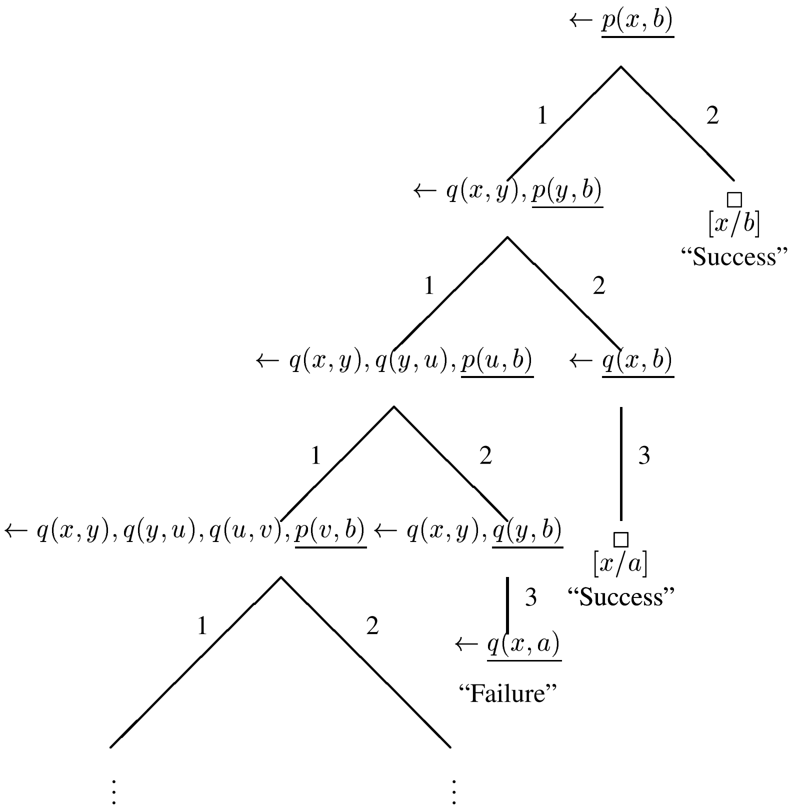



Abbildung 9.1: An Infinite SLD-Tree

Note that not any correct answer is computed, only the most general one is (which of course subsumes all the correct ones).

The main feature of SLD-Resolution is its *Goal-Orientedness*. SLD automatically ensures (because it starts with the Query) that we consider only those rules that are relevant for the query to be answered. Rules that are not at all related are simply not considered in the course of the proof.

## 9.2 Bottom-Up

We mentioned in the last section the least Herbrand model $M_P$. The bottom-up approach can be described as computing this least Herbrand model from below. We start first with rules with empty bodies (in our example these are all instantiations of rules (2) and (3)). We get as facts all atoms that are in the heads of rules with empty bodies (namely $p(a,a), p(b,b), q(a,b)$ in Example 9.1.1). In the next round we use the facts that we computed before and try to let the rules “fire”, i.e. when their bodies are true, we add their heads to the atoms we already have (this gives us $p(a,b)$).

To be more precise we introduce the immediate consequence operator $T_P$ which associates to any Herbrand model another Herbrand model.

**Example 9.2.1 ($T_P$)**
*Given a definite program $P$ let $T_P : 2^{B_P} \longmapsto 2^{B_P}; \mathcal{I} \longmapsto T_P(\mathcal{I})$*

$$T_P(\mathcal{I}) := \{A \in B_P : \text{ there is an instantiation of a rule in } P \text{ s.t. } A \text{ is the head of this rule and all body-atoms are contained in } \mathcal{I}\ \}$$

It turns out that $T_P$ is monotone and continuous so that (by a general theorem of Knaster-Tarski) the least fixpoint is obtained after $\omega$ steps. Moreover we have

**Theorem 9.2.1 ($T_P$ and $M_P$)**
$M_P = T_P{\uparrow}^{\omega} = lfp(T_P)$.

This approach is especially important in Database applications, where the underlying language does not contain function symbols (DATALOG) — this ensures the Herbrand universe to be finite. Under this condition the iteration stops after finitely many steps. In addition, rules of the form

$$p \leftarrow p$$

do not make any problems. They simply can not be applied or do not produce anything new. Note that in the Top-Down approach, such rules give rise to infinite branches! Later, elimination of such rules will turn out to be an interesting property. We therefore formulate it as a principle:

**Principle 9.2.2 (Elimination of Tautologies)**
*Suppose a program $P$ has a rule which contains the same atom in its body as well as in its head (i.e. the head consists of exactly this atom). Then we can eliminate this rule without changing the semantics.*

Unfortunately, such a bottom-up approach has two serious shortcomings. First, the goal-orientedness from SLD-resolution is lost: we are always computing the whole $M_P$, even those facts that have nothing to do with the query. The reason is that in computing $T_P$ we do not take into account the query we are really interested in. Second, in any step facts that are already computed before are recomputed again. It would be more efficient if only new facts were computed. Both problems can be (partially) solved by appropriate refinements of the naive approach:

- *Semi-naive* bottom-up evaluation ([Bry90, Ull89b]),
- *Magic Sets* techniques ([BR91, Ull89a]).

## 9.3 Herbrand-Models and the underlying language

Usually when we represent some knowledge in first order logic or even in logic programs, it is understood that the underlying language is given exactly by the symbols that occur in the formal theory. Suppose we have represented some knowledge about the world as a theory $T$ in a language $\mathcal{L}$. Classical predicate logic formalizes the notion of a formula $\phi$ *derivable* from the theory $T$. This means that $\phi$ is true in all models of $T$ (we denote this set by MOD($T$)). Why are we considering all models? Doesn't it make sense to look only at Herbrand models, i.e. to models generated by the underlying language? After all we are not interested in models that contain elements which are not representable as terms in our language. These requirements are usually called *Unique Names Assumption* and *Domain Closure Assumption*:

**Definition 9.3.1 (UNA and DCA)**
*Let a language $\mathcal{L}$ be given. We understand by the* Unique Names Assumption *the restriction to those models $\mathcal{I}$, where syntactically different ground $\mathcal{L}$-terms $t_1, t_2$ are interpreted as nonidentical elements: $t_1^{\mathcal{I}}$ is not identical to $t_2^{\mathcal{I}}$.*

*By the* Domain Closure Assumption *we mean the restriction to those models $\mathcal{I}$ where for any element $a$ in $\mathcal{I}$ there is a $\mathcal{L}$-term $t$ that represents this element: $a = t^{\mathcal{I}}$.*

As an example, in Theorem 9.1.1 of Section 9.1 we referred to $M_P$, the least Herbrand model of $P$. The reason that the first equivalence in this theorem holds is given by the fact that for universal theories $T$ and existential formulae $\phi$ the following holds

$$\text{MOD}(T) \models \phi \quad \text{iff} \quad \text{Herb}_{\mathcal{L}}\text{-MOD}(T) \models \phi.$$

In our particular case, where $T$ is a definite program $P$, we can even replace Herb$_{\mathcal{L}}$-MOD($T$) in the above equation by the single model $M_P$.

This last result does not hold in general. But what happens if we nevertheless are interested in only the Herbrand-models of a theory $T$ (and therefore automatically[3] assume UNA and DCA)? At first sight one can argue that such an approach is much simpler: in contrast to *all* models we only need to take care about the very specific Herbrand models. But it turns out that determining the truth of a formula in all Herbrand models is a much more complex task (namely $\Pi_1^1$-complete) than to determine if it is true in all models. This latter task is also undecidable in general, but it is recursively enumerable, i.e. $\Pi_1^0$-complete. The fact that this task is recursively enumerable is the content of the famous completeness theorem of Gödel, where "truth of a formula in all models" is shown to be equivalent to deriving this formula in a particular axiomatization of the predicate calculus of first order. We refer to the appendix (Section A.1 and Section A.2) where the necessary notions are introduced.

But we have still a problem with Theorem 9.1.1 in our restricted setting:

**Example 9.3.1 (Universal Query Problem)**
*Consider the program $P := p(a)$, the query $Q := p(x)$ and the empty substitution $\Theta := \epsilon$. We have*

- $M_P \models \forall x p(x)$,
- *but SLD only computes the answer $x/a$.*

[3]The only difference between Herbrand models and models satisfying UNA and DCA is that the interpretation of terms is uniquely determined in Herbrand models. It is required that a term "$f(t_1, \ldots, t_n)$" is interpreted in a Herbrand model $\mathcal{I}$ as "$f(t_1^{\mathcal{I}}, \ldots, t_n^{\mathcal{I}})$".

*Przymusinski called this* the universal query problem.

There are essentially two solutions to avoid this behaviour: to use a language which is *rich enough* (i.e. contains sufficiently many terms, not only those ocurring in the program $P$ itself) or to consider arbitrary models, not only Herbrand models. Both approaches have been followed in the literature but they are beyond the scope of this paper.

## 9.4 Why going beyond Definite Programs?

So far we have a nice query-answering procedure, SLD-Resolution, which is goal-oriented as well as sound and complete with respect to general derivability. But note that up to now we are not able to derive any *negative* information. Not even our queries allow this. From a very pragmatic viewpoint, we can consider "*not* $A$" to be derivable if $A$ is not. Of course, this is not sound with respect to classical logic but it is with respect to $M_P$.

In KR we do not only want to formulate negative queries, we also want to express *default-statements* of the form

*Normally, unless something abnormal holds,* then $\psi$ implies $\phi$.

Such statements were the main motivation for nonmonotonic logics, like Default Logic or Circumscription (see Section A.3 and Section A.4 of the appendix). How can we formulate such a statement as a logic program? The most natural way is to use negation "*not* "

$$\phi \leftarrow \psi, \; not\ ab$$

where $ab$ stands for *abnormality*. Obviously, this forces us to extend definite programs by negative atoms.

A typical example for such statements occurs in Inheritance Reasoning. We take the following example from [BG94]:

**Example 9.4.1 (Inheritance Hierachies)**
*Suppose we know that birds typically fly and penguins are non-flying birds. We also know that Tweety is a bird. Now an agent is hired to build a cage for Tweety. Should the agent put a roof on the cage? After all it could be still the case that Tweety is a penguin and therefore can not fly, in which case we would not like to pay for the unneccessary roof. But under normal conditions, it should be obvious that one should conclude that Tweety is flying.*

*A natural axiomatization is given as follows:*

$$\begin{array}{lllll} P_{Inheritance}: & flies(x) & \leftarrow & bird(x), & \text{not } ab(r_1,x) \\ & bird(x) & \leftarrow & penguin(x) & \\ & ab(r_1,x) & \leftarrow & penguin(x) & \\ & make\_top(x) & \leftarrow & flies(x) & \end{array}$$

*together with some particular facts, like e.g.* $bird(Tweety)$ *and* $penguin(Sam)$. *The first rule formalizes our default-knowledge, while the third formalizes that the default-rule should not be applied in abnormal or exceptional cases. In our example, it expresses the famous* Specificity-Principle *which says that more specific knowledge should override more general one ([Tou86, THT88, HTT90]).*

*For the query "*$make\_top(Tweety)$*" we expect the answer "yes" while for the query "*$make\_top(Sam)$*" we expect the answer "no".*

Another important KR task is to formalize knowledge for reasoning about action. We again consider a particular important instance of such a task, namely *temporal projection*. The overall framework consists in describing the initial state of the world as well as the effects of all actions that can be performed. What we want to derive is how the world looks like after a sequence of actions has been performed.

**Example 9.4.2 (Temporal Projection: Yale-Shooting Problem)**
*We distinguish between three sorts*[4] *of variables:*

- *situation variables:* $S, S', \ldots,$
- *fluent variables:* $F, F', \ldots,$
- *action variables:* $A, A', \ldots.$

*The initial situation is denoted by the constant* $s_0$, *and the two-ary function symbol* $res(A, S)$ *denotes the situation that is reached when in situation* $S$ *the action* $A$ *has been performed. The relation symbol* $holds(F, S)$ *formalizes that the fluent* $F$ *is true in situation* $S$.

*For the YSP there are three actions (wait,* $load$ *and shoot) and two fluents (*$alive$ *and* $loaded$*). Initially a turkey called* $Fred$ *is alive. We then load a gun, wait and shoot. The effect should be that Fred is dead after this sequence of actions. The common-sense argument from which this should follow is the*

**Law of Inertia:** *Things normally tend to stay the same.*

*Using our intuition from the last example, a natural formalization is given as follows:*

$$\begin{array}{lll} P_{YSP}: & holds(F, res(A, S)) & \leftarrow & holds(F, S),\ \text{not}\ ab(r_1, A, F, S) \\ & holds(loaded, res(load, S)) & \leftarrow & \\ & ab(r_1, shoot, alive, S) & \leftarrow & holds(loaded, S) \\ & holds(alive, s_0) & \leftarrow & \end{array}$$

*Such a straightforward formalization leads in most versions of classical nonmonotonic logic to the unexpected result, that Fred is not neccesarily dead. But obviously we expect to derive* $holds(alive, res(load, s_0))$ *and*

$$\text{not}\ holds(alive\ ,\ res(shoot, res(wait, res(load, s_0))))$$

[4]To be formally correct we have to use many-sorted logic. But since all this could also be coded in predicate logic by using additional relation symbols, we do not emphasize this fact. We also understand that instantiations are done in such a way that the sorts are respected.

Up to now we only have stated some very "natural" axiomatizations of given knowledge. We have motivated that something like default-negation "*not*" should be added to definite programs in order to do so and we have explicitly stated the answers to particular queries. What is still missing are solutions to the following very important problems

- *How should an appropriate query answering mechanism handling default-negation "* not *" look like?*
- *What is the formal semantics that such a procedural mechanism should be checked against?*

Such a semantics is certainly not classical predicate logic because of the default character of "*not*" — *not* is not classical $\neg$. Both problems will be considered in detail in Section 10.

## 9.5 What is a Semantics?

In the last sections we have introduced two principles (*Orientation* and *Elimination of Tautologies*) and used the term *semantics of a program* in a loose, imprecise way. We end this section with a precise notion of what we understand by a semantics.

As a first attempt, we can view a semantics as a mapping that associates to any program a set of positive atoms and a set of default atoms. In the case of SLD-Resolution the positive atoms are the ground instances of all derivable atoms. But sometimes we also want to derive negative atoms (like in our two examples above). Our *Orientation*-Principle formalizes a minimal requirement for deriving such default-atoms.

Of course, we also want that a semantics SEM should *respect* the rules of $P$, i.e. whenever SEM makes the body of a rule true, then SEM should also make the head of the rule true. But it can (and will) happen that a semantics SEM does not always decide *all* atoms. Some atoms $A$ are not derivable nor are their default-counterparts *not* $A$. This means that a semantics SEM can view the body of a rule as being *undefined*.

This already happens in classical logic. Take the theory

$$T := \{(A \wedge B) \supset C,\ \neg A \supset B\}.$$

What are the atoms and negated atoms derivable from $T$, i.e. true in all models of $T$? No positive atom nor any negated atom is derivable! The classical semantics therefore makes the truthvalue of $A \wedge B$ undefined in a sense.

Suppose a semantics SEM treats the body of a program rule as undefined. What should we conclude about the head of this rule? We will only require that this head is not treated as false by SEM — it could be true or undefined as well. This means that we require a semantics to be compatible with the program *viewed as a 3-valued theory* — the three values being "true", "false" and "undefined". For the understanding it is not neccessary to go deeper into 3-valued logic. We simply note that we interpret "$\leftarrow$" as the Kleene-connective which is true for "$undefined \leftarrow undefined$" and false for "$false \leftarrow undefined$".

Our discussion shows that we can view a semantics SEM as a 3-valued model of a program. In classical logic, there is a different viewpoint. For a given theory $T$ we consider there the set of all classical models MOD($T$) as the semantics. The intersection of all these models is of course a 3-valued model of $T$, but MOD($T$) contains more information. In order to formalize the notion of semantics as general as possible we define

**Definition 9.5.1 (SEM)**
*A semantics SEM is a mapping from the class of all programs into the powerset of the set of all 3-valued structures. SEM assigns to every program $P$ a set of 3-valued models of $P$:*

$$\mathit{SEM}(P) \subseteq \mathit{MOD}^{\mathcal{L}_P}_{3-val}(P).$$

This definition covers both the classical viewpoint (classical models are 2-valued and therefore special 3-valued models) as well as our first attempt in the beginning of this section. Later on, in most cases we will be really interested only in Herbrand models.

Formally, we can associate to any semantics SEM in the sense of Definition 9.5.1 two entailment relations

**sceptical:** SEM$^{scept}(P)$ is the set of all atoms or default atoms that are true in *all models* of SEM$(P)$.

**credulous:** SEM$^{cred}(P)$ is the set of all atoms or default atoms that are true in *at least one model* of SEM$(P)$.

In this tutorial we only consider the sceptical viewpoint. Also, to facilitate notation, we will not formally distinguish between SEM and SEM$^{scept}$. In cases where by definition SEM can only contain a single model (like in the case of well-founded semantics) we will omit the outer brackets and write

$$\text{SEM}(P) = M$$

instead of SEM$(P) = \{M\}$. We will also slightly abuse notation and write $l \in$ SEM$(P)$ as an abbreviation for $l \in M$ for all $M \in$ SEM$(P)$.

# 10 KR with Normal Programs

In the last section we have illustrated that logic programs with negation are very suitable for KR — they allow a natural and straightforward formalization of default-statements. The problem still remained to define an appropriate semantics for this class and, if possible, to find efficient query-answering methods. Both points are adressed in this section.

We can distinguish between two quite different approaches:

**LP-Approach:** This is the approach taken mainly in the Logic Programming community. There one tried to stick as close as possible to SLD-Resolution and treat negation as "Finite-Failure". This resulted in an extension of SLD, called SLDNF-Resolution, a procedural mechanism for query answering. For a nice overview, we refer to [AB94].

**NML-Approach:** This is the approach suggested by non-monotonic reasoning people. Here the main question is *"What is the right semantics?"* I.e. we are looking first for a semantics that correctly fits to our intuitions and treats the various KR-Tasks in the right (or appropriate) way. It should allow us to jump to conclusions even when only little information is available. Here it is of secondary interest how such a semantics can be implemented with a procedural calculus. Interesting overviews are [Min93] and [Dix95d].

The LP-Approach is dealt with in Section 10.1. It is still very near to classical predicate logic — default negation is interpreted as *Finite-Failure*. To get a stronger semantics, we interpret "*not*" as *Failure* in Section 10.2. The main difference is that the principle *Elimination of Tautologies* holds. We then introduce a principle GPPE which is related to partial evaluation. In KR one can see this principle as allowing for definitional extensions — names or abbreviations can be introduced without changing the semantics.

All these principles do not yet determine a unique semantics — there is still room for different semantics and a lot of them have been defined in the last years. We do not want to present the whole zoo of semantics nor to discuss their merits or shortcomings. We refer the reader to the overview articles [AB94] and [Dix95d] and the references given therein. We focus on the two main competing approaches that still have survived. These are the Wellfounded semantics WFS (Section 10.3) and the Stable semantics STABLE (Section 10.4). Finally, in Chapter 12 we discuss complexity and expressibility results for the semantics presented so far.

## 10.1 Negation-as-Finite-Failure

The idea of negation treated as *finite-failure* can be best illustrated by still considering definite programs, but queries containing default-atoms. How should we handle such default-atoms by modifying our SLD-resolution? Let us try this:

- If we reach a default-atom "*not* $A$" as a subgoal of our original query, we keep the current SLD-tree in mind and start a new SLD-tree by trying to solve "$A$".
- If this succeeds, then we falsified "*not* $A$", the current branch is failing and we have to backtrack and consider a different subquery.
- But it can also happen that the SLD-tree for "$A$" is *finite with only failing branches*. Then we say that $A$ *finitely fails*, we turn back to our original SLD-tree, consider the subgoal "*not* $A$" as successfully solved and go on with the next subgoal in the current list.

It is important to note that an SLD-tree for a positive atom can fail without *being finite*. The SLD-tree for the program consisting of the single rule $p \leftarrow p$ with respect to the query $p$ is infinite but failing (it consists of one single infinite branch). In Figure 9.1 the leftmost branch is also failing but infinite.

Although this idea of *Finite-Failure* is very procedural in nature, there is a nice modeltheoretical counterpart — Clark's completion $comp(P)$ ([Cla78]). The idea of Clark was that a program $P$ consists not only of the implications, but also of the information that *these are the only ones.* Roughly speaking, he argues that one should interpret the "$\leftarrow$"-arrows in rules as equivalences "$\equiv$" in classical logic. We do not give the exact definitions here, as they are very complex; in the non-propositional case, a symbol for equality, together with axioms describing it[1], has to be introduced. However, for the propositional case, $comp(P)$ is obtained from $P$ by just

1. collecting all given clauses with the same head into one new "clause" with this respective head and a disjunctive body (containing all bodies of the old clauses), and
2. replacing the implication-symbols "$\leftarrow$" by "$\equiv$".

**Definition 10.1.1 (Clark's Completion $comp(P)$)**
*Clark's semantics for a program P is given by the set of all classical models of the theory $comp(P)$.*

We can now see the classical theory $comp(P)$ as the information contained in the program $P$. $comp(P)$ is like a sort of closed world assumption applied to $P$. We are now able to derive negative information from $P$ by deriving it from $comp(P)$. In fact, the following soundness and completeness result for definite programs $P$ and definite queries $Q = \bigwedge_i A_i$ (consisting of only *positive* atoms) holds:

[1]CET: Clark's Equational Theory. CET($\mathcal{L}_P$) axiomatizes the equality theory of all Herbrand($\mathcal{L}_P$)-models. See [MMP88, She92] for the problem of equality and the underlying language.

**Theorem 10.1.2 (COMP and Fair FF-Trees)**
*The following conditions are equivalent:*

- $comp(P) \models \forall\neg Q$
- *Every fair SLD-tree for $P$ with respect to $Q$ is finitely failed.*

Note that in the last theorem we did not use default negation but classical negation $\neg$ because we just mapped all formulae into classical logic. We need the fairness assumption to ensure that the selection of atoms is reasonably well-behaving: we want that every atom or default-atom occurring in the list of preliminary goals will eventually be selected.

But even this result is still very weak — after all we want to handle not only negative queries but programs containing default-atoms. From now on we consider programs with default-atoms in the body. We usually denote them by

$$A \leftarrow \mathcal{B}^+ \wedge \mathit{not}\,\mathcal{B}^-,$$

where $\mathcal{B}^+$ contains all the positive body atoms and $\mathit{not}\,\mathcal{B}^-$ all default atoms "*not C*".

Our two motivating examples in Section 9.4 contain such default atoms. This gives rise to an extension of SLD, called SLDNF, which treats negation as *Finite-Failure*

SLDNF = SLD + *not* $L$ succeeds, if $L$ finitely fails.

The precise definitions of SLDNF-*resolution*, *tree*, etc. are very complex: we refer to [Llo87, Apt90]. Recently, Apt and Bol gave interesting improved versions of these notions: see [AB94, Section 3.2]. In order to get an intuitive idea, it is sufficient to describe the following underlying principle:

**Principle 10.1.3 (A "Naive" SLDNF-Resolution)**
*If in the construction of an SLDNF-tree a default-atom* not $L_{ij}$ *is selected in the list* $\mathcal{L}_i = \{L_{i1}, L_{i2}, \dots\}$*, then we try to prove* $L_{ij}$.
*If this* fails finitely *(it fails because the generated subtree is finite and failing), then we take* not $L_{ij}$ *as proved and we go on to prove* $L_{i(j+1)}$.
*If* $L_{ij}$ *succeeds, then* not $L_{ij}$ *fails and we have to backtrack to the list* $\mathcal{L}_{i-1}$ *of preliminary subgoals (the next rule is applied: "backtracking").*

Does SLDNF-Resolution properly handle Examples 9.4.1 and 9.4.2? It does indeed:

**Inheritance:** The query $make\_top(Tweety)$ generates an SLD-tree with one main branch, the nodes of which are:

$$\begin{array}{l} flies(Tweety), \\ bird(Tweety),\ \mathit{not}\ ab(r_1, Tweety), \\ \mathit{not}\ ab(r_1, Tweety), \\ Success. \end{array}$$

The third node has a sibling-node $penguin(Tweety),\ \mathit{not}\ ab(r_1, Tweety)$ which immediately fails because $Tweety$ does not unify with $Sam$. The $Success$-node is obtained from $\mathit{not}\ ab(r_1, Tweety)$ because the corresponding SLD-tree for $ab(r_1, Tweety)$ fails finitely (this tree consists only of $ab(r_1, Tweety)$ and $penguin(Tweety)$).

**YSP:** The crucial query is

$$?\text{- } \textit{not } holds(alive\,,\,res(shoot, res(wait, res(load, s_0))))$$

So we consider *?-* $holds(alive\,,\,res(shoot, res(wait, res(load, s_0))))$ Again the SLD-tree for this query consists mainly of one branch: the nodes are obtained from the query by applying successively the first program rule (*law of inertia*). By evaluation of the holds-predicate, we eventually arrive at the fact $holds(alive, s_0)$ and the "*not* $ab$" predicates remain to be solved. For any of these predicates we again have to consider separate SLD-trees. But for the predicate $ab(r_1, shoot, alive, res(wait, res(load, s_0)))$ it is easy to see that the associated tree already finitely fails (because it generates the subgoal "*not* $ab(r_1, wait, loaded, res(load, s_0))$" the corresponding SLD-tree of which immediately finitely fails) and therefore, since no backtracking is possible, the tree for

$$?\text{- } holds(alive\,,\,res(shoot, res(wait, res(load, s_0))))$$

finitely fails and our original query succeeds: *Fred is dead.*

Up to now it seems that SLDNF-resolution solves all our problems. It handles our examples correctly, and is defined by a procedural calculus strongly related to SLD. There are two main problems with SLDNF:

- SLDNF can not handle free variables in negative subgoals,
- SLDNF is still too weak for Knowledge Representation.

The latter problem is the most important one. By looking at a particular example, we will motivate in Section 10.2 the need for a stronger semantics. This will lead us in the remaining sections to the wellfounded and the stable semantics.

For the rest of this section we consider the first problem, known as the *Floundering Problem*. This problem will also occur later in implementations of the wellfounded or the stable semantics. We consider the program $P_{flounder}$ consisting of the three facts

$$p(c, c),\ q(b),\ r(f(c)).$$

Our query is *?-* $p(x, c)$, *not* $q(x), r(f(x))$ that is, we are interested in instantiations of $x$ such that the query follows from the program. The situation is illustrated in Figure 10.1. Let us suppose that we always select the first atom or default-atom: it is underlined in the sequel. The SLDNF-tree of this trivial example is linear and has three nodes: the first node is the query itself

$$?\text{- } \underline{p(x, c)},\ \textit{not } q(x), r(f(x))$$

the second node is *?-* $\underline{\textit{not } q(c)}, r(f(c))$ Now, we enter the negation-as-failure mode and ask *?-* $q(c)$ This query immediately fails (the generated tree exists, is finite and

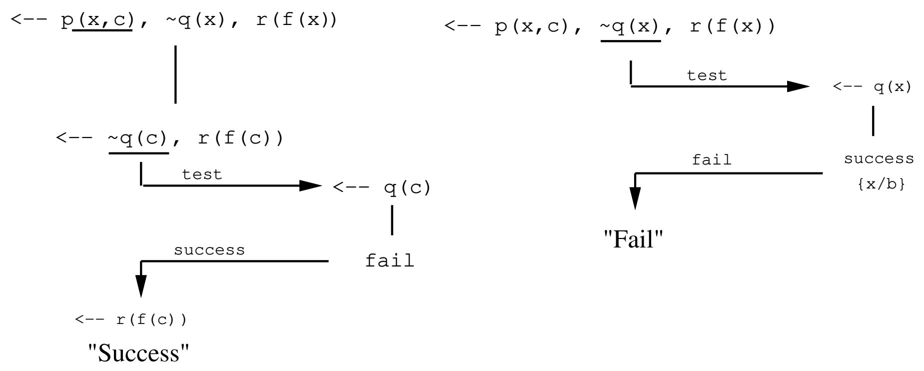



Abbildung 10.1: The Floundering-Problem

fails) so that we give back the answer "yes, the default atom $\overline{\textit{not } q(c)}$ succeeds and can be skipped from the list". The last node is *?-* $r(f(c))$ which immediately succeeds.

Note that in the last step, the test for *?-*$q(c)$ has to be finished before the tree can be extended. If we get no answer, the SLDNF-tree simply does not exist: this can not happen with SLD-trees.

So far everything was fine. But what happens if we select the second atom in the first step

$$?\text{-}p(x,c), \underline{\textit{not } q(x)}, r(f(x))$$

**Example 10.1.1 (Floundering)**
*We again consider the program $P_{flounder}$ consisting of the three facts*

$$p(c,c),\ q(b),\ r(f(c)).$$

*Our query is* ?-$p(x,c)$, not $q(x), r(f(x))$ *and in the first step we will select the second default-atom, i.e. one with a free variable. Thus we enter the negation-as-failure mode with the query* ?- not $q(x)$ *In this case,* $x$ *may be instantiated to* $b$ *so that we have to give back the answer "no, the default-atom* not $q(x)$ *fails" and the whole query will fail. This is because SLDNF treats the subgoal as "*$\forall x$ not $q(x)$*" instead of "*$\exists x$ not $q(x)$*" which is intended. There exist approaches to overcome this shortcoming by treating negation as* constructive negation*: see [Cha88, CW89, Dra94].*

In the classical SLDNF-resolution *negation-as-finite-failure is only a test, no bindings are produced*. On the one hand this may be considered a shortcoming, on the other hand, it makes the SLDNF procedure more tractable. Note that the problem to decide if a given program flounders is undecidable [Bör87]). See also [She91] for more unsolvable problems related to SLDNF.

SLDNF is a procedural mechanism. It would be nice to have a modeltheoretical counterpart. In Theorem 10.1.2 we already related a restricted form of finite failure to Clark's completion. We will see later that $comp(P)$ is inconsistent even in cases where we would not expect it. Therefore Fitting [Fit85] introduced a three-valued formulation $comp_3(P)$ of the original completion. Kunen ([Kun87]) then proved in the propositional case *SLDNF is sound and complete with respect to* $comp_3(P)$.

In the predicate logic case, SLDNF is not complete but it is always correct [She88, Theorem 39]) with respect to $comp_3(P)$: given a query $Q$,

- if SLDNF succeeds with answer $\Theta$, then $comp_3(P) \models_3 \forall Q\Theta$, and
- if SLDNF fails, then $comp_3(P) \models_3 \neg\exists Q$.

This correctness result is also the reason for the incompleteness of SLDNF with respect to two-valued $comp(P)$. It states that any formula derivable by SLDNF is a three-valued consequence of $comp_3(P)$. But, since there are two-valued consequences of a theory that are not three-valued ones (three-valued logic is weaker than two-valued logic), SLDNF can not be complete. Extensions of the above completeness result to certain subclasses of predicate logic programs require severe restrictions on the syntactic form of $P$. To define these syntactic restrictions, we need the notion of the dependency-graph:

**Definition 10.1.4 (Dependency-Graph $\mathcal{G}_P$)**
*For a logic program $P$ with negation, the* dependency graph $\mathcal{G}_P$ *is a finite directed graph whose vertices are the predicate symbols from P. There is a* positive *(respectively* negative*) edge from $R$ to $R'$ if and only if there is a clause in P with $R$ in its head and $R'$ occurring positively (respectively negative) in its body.*

*We also say*

- $R$ depends on $R'$ *if there is a path in $\mathcal{G}_P$ from $R$ to $R'$ (by definition, $R$ depends on itself),*
- $R$ depends positively *(resp.* negatively*) on $R'$ if there is a path in $\mathcal{G}_P$ from $R$ to $R'$ containing only* positive *edges (resp.* at least one negative *edge). (by definition $R$ depends positively on itself),*
- $R$ depends evenly *(resp.* oddly*) on $R'$ if there is a path in $\mathcal{G}_P$ from $R$ to $R'$ containing an* even *(resp.* odd*) number of negative edges (by definition $R$ depends evenly on itself).*

The following properties of a program $P$ turn out to be very important:

*stratified:* no predicate depends negatively on itself[2],
*strict:* there are no dependencies that are both even and odd,
*call-consistent:* no predicate depends oddly on itself[3],
*allowedness:* every variable occurring in a clause must occur in at least one positive atom of the body of that clause.

Strictness and allowedness turn out to be the most important restrictions that imply completeness results for SLDNF:

[2]or: there are no cycles containing at least one *negative* edge.
[3]or: there are no odd cycles.

| Prog. P | Semantics | Completeness |
|---|---|---|
| allowed + hierarchical<br>allowed + stratified<br>allowed | $comp(P)$<br>$comp(P)$<br>$comp_3(P)$ | yes, no recursion at all<br>yes, if $P \cup \{\leftarrow A\}$ strict<br>yes, w.r.t. $\vdash_3$ |
| allowed + call-consistent | $comp(P)$ | yes, if $P \cup \{\leftarrow A\}$ strict:<br>$comp(P) \vdash \forall A$ if and only if<br>$comp_3(P) \vdash_3 \forall A$ |

Tabelle 10.1: Completeness for SLDNF

While *strictness* excludes situations of the form $p(x) \leftarrow q(x)$, $p(c) \leftarrow \neg q(f(c'))$, *allowedness* excludes constructs of the form $equal(x, x) \leftarrow$ and also solves the floundering-problem.

Strictness implies that $comp_3(P)$ and $comp(P)$ are equivalent [Kun89])

$$comp_3(P) \models_3 \ \forall Q\Theta \quad \text{if and only if} \ \ comp(P) \models \ \forall Q\Theta.$$

Table 10.1 gives an overview of the different completeness results. Note that the query $A$ is always considered to be allowed.

Much work was done in LP (see [DC90, BM86, Stä94]) to find other syntactically characterizable classes, for which SLDNF is also *complete*.

## 10.2 Negation-as-Failure

Let us first illustrate that SLDNF answers quite easily our requirements of a semantics SEM (stated explicitly in Definition 9.5.1). We can formulate these requirements as two program-transformations (they will be used later for computing a semantics). We call them *Reductions* for obvious reasons.

**Principle 10.2.1 (Reduction)**
*Suppose we are given a program $P$ with possibly default-atoms in its body. If a ground atom $A$ does not unify with any head of the rules of $P$, then we can delete in every rule any occurrence of "* not $A$*" without changing the semantics.*

*Dually, if there is an instance of a rule of the form "$B \leftarrow$ " then we can delete all rules that contain "* not $B$*" in their bodies.*

It is obvious that SLDNF "implements" these two reductions automatically. The weakness of SLDNF for Knowledge Representation is in a sense inherited from SLD. When we consider rules of the form "$p \leftarrow p$", then SLD resolution gets into an infinite loop and no answer to the query *?-p* can be obtained. This has often the effect that when we enter into negation-as-failure mode, the SLD-tree to be constructed is not finite, although it is not successful and therefore should be considered as failed.

Let us discuss this point with a more serious example.

**Example 10.2.1 (The Transitive Closure)**
*Assume we are given a graph consisting of nodes and edges between some of them. We want to know which nodes are reachable from a given one. A natural formalization of the property "reachable" would be*

$$reachable(x) \leftarrow edge(x, y), reachable(y).$$

*What happens if we are given the following facts*

$$edge(a, b),\ edge(b, a),\ edge(c, d)$$

*and $reachable(c)$? Of course, we expect that neither $a$ nor $b$ are reachable because there is no path from $c$ to either $a$ or $b$.*

*But SLDNF-Resolution does not derive "* not $reachable(a)$*"!*

How does this result relate to Theorem 10.1.2? Note that our query has exactly the form as required there. Clark's completion of our program rule is

$$reachable(x)\ \equiv\ (x \doteq c\ \vee\ \exists y\,(reachable(y)\ \wedge\ edge(y, x)))$$

from which, together with our facts about the edge-relation, $\neg reachable(a)$ is indeed not derivable. This is due to the wellknown fact that transitive closure is not expressible in first order predicate logic.

Note also that our Principle 9.2.2 does not help, because it simply does not apply. It turns out that we can augment our two principles by a third one, that constitutes together with them a very nice calculus handling the above example in the right way. This principle is related to *Partial Evaluation*, hence its name GPPE[4]. Let us motivate this principle with the last example. The query "*not* $reachable(a)$" leads to "$reachable(a)\ \leftarrow\ edge(a, b), reachable(b)$" and "$reachable(b)$" leads to a similar query, namely "$reachable(b) \leftarrow edge(b, a), reachable(a)$". Both rules can be seen as definitions for $reachable(a)$ and $reachable(b)$ respectively. So it should be possible to replace in these rules the body atoms of $reachable$ by their definitions. Thus we obtain the two rules

$$\begin{array}{l} reachable(a) \leftarrow edge(a, b), edge(b, a), reachable(a) \\ reachable(b) \leftarrow edge(b, a), edge(a, b), reachable(b) \end{array}$$

that can both be eliminated by applying Principle 9.2.2. So we end up with a program that does neither contain $reachable(a)$ nor $reachable(b)$ in one of the heads. Therefore, according to Principle 9.0.3 both atoms should be considered false. The precise formulation of this principle is as follows:

[4]Generalized Principle of Partial Evaluation

**Principle 10.2.2 (GPPE,[BD94, SS94])**
*We say that a semantics SEM satisfies GPPE, if the following transformation does not change the semantics.* Replace a rule $A \leftarrow \mathcal{B}^+ \wedge \textit{not}\ \mathcal{B}^-$ where $\mathcal{B}^+$ contains a distinguished atom $B$ by the rules

$$A \cup \left(\mathcal{A}_i \setminus \{B\}\right) \ \leftarrow\ \left(\mathcal{B}^+ \setminus \{B\}\right) \cup \mathcal{B}_i^+ \ \wedge\ \textit{not}\ \left(\mathcal{B}^- \cup \mathcal{B}_i^-\right)\ (i = 1, \ldots, n)$$

*where* $B \leftarrow \mathcal{B}_i^+ \wedge \text{not}\ \mathcal{B}_i^-$ *(*$i = 1, \ldots, n$*) are all rules with head* $B$*.*

Note that any semantics SEM satsfying GPPE and Elimination of Tautologies can be seen as extending SLD by doing some *Loop-checking*. We will call such semantics *NMR-semantics* in order to distinguish them from the classical *LP-semantics* which are based on SLDNF or variants of Clark's completion $comp(P)$:

- *NMR-Semantics = SLDNF + Loop-check.*

The following, somewhat artificial example illustrates this point.

**Example 10.2.2 (COMP vs. NMR)**

$$\begin{array}{rlcl}
P_{NMR}: & p & \leftarrow & p \\
& q & \leftarrow & \text{not}\ p
\end{array}
\qquad\qquad
\begin{array}{rlcl}
P'_{NMR}: & p & \leftarrow & p \\
& q & \leftarrow & \text{not}\ p \\
& r & \leftarrow & \text{not}\ r
\end{array}$$

$$\begin{array}{rlcl}
comp(P_{NMR}): & p & \equiv & p \\
& q & \equiv & \neg p
\end{array}
\qquad\qquad
\begin{array}{rlcl}
comp(P'_{NMR}): & p & \equiv & p \\
& q & \equiv & \neg p \\
& r & \equiv & \neg r
\end{array}$$

| | | | |
|---|---|---|---|
| *?-q*: | *No (COMP).* | *?-p*: | *Yes (COMP).* |
| | *Yes (NMR).* | | *No (NMR).* |

*For both programs, the answers of the completion-semantics do not match our NMR-intuition! In the case of* $P_{NMR}$ *we expect q to be derivable, since we expect* not $p$ *to be derivable: the only possibility to derive p is the rule* $p \leftarrow p$ *which, obviously, will never succeed. But* $q \notin Th(\{q \equiv \neg p\}) = comp(P_{NMR})$*! In the case of* $P'_{NMR}$ *we expect p not to be derivable, for the same reason: the only possibility to derive p is the rule* $p \leftarrow p$*. But* $p \in Fml = Th(\{r \equiv \neg r\}) = comp(P'_{NMR})$*!*

*Note that the answers of the completion-semantics agree with the mechanism of SLDNF:* $p \leftarrow p$ *represents a loop. The completion of* $P'$ *is inconsistent: this led Fitting to consider the three-valued version of* $comp(P)$ *mentioned at the end of Section 10.1. This approach avoids the inconsistency (the query* ?-$p$ *is not answered "yes") but it still does not answer "no" as we would like to have.*

The last principle in this section is related to *Subsumption*: we can get rid of non-minimal rules by simply deleting them.

**Principle 10.2.3 (Subsumption)**
*In a program* $P$ *we can delete a rule* $A \leftarrow \mathcal{B}^+ \wedge \text{not}\ \mathcal{B}^-$ *whenever there is another rule* $A \leftarrow \mathcal{B}'^+ \wedge \text{not}\ \mathcal{B}'^-$ *with*

$$\mathcal{B}'^+ \subseteq \mathcal{B}^+ \text{ and } \mathcal{B}'^- \subseteq \mathcal{B}^-.$$

As a simple example, the rule $A \leftarrow B, C, \textit{not } D, \textit{not } E$ is subsumed by the 3 rules $A \leftarrow C, \textit{not } D, \textit{not } E$ or $A \leftarrow B, C, \textit{not } E$ and by $A \leftarrow C, \textit{not } E$.

## 10.3 The Wellfounded Semantics: WFS

The wellfounded semantics, originally introduced in [VGRS88], is the *weakest* semantics satisfying our 4 principles (see [BD98b, BD98a, Dix95b]). We call a semantics

$$\text{SEM}_1 \text{ weaker than } \text{SEM}_2, \text{ written } \text{SEM}_1 \leq_k \text{SEM}_2,$$

if for all programs $P$ and all atoms or default-atoms $l$ the following holds: $\text{SEM}_1(P) \models l$ implies $\text{SEM}_2(P) \models l$. I.e. all atoms derivable from $\text{SEM}_1$ with respect to $P$ are also derivable from $\text{SEM}_2$. The notion $\leq_k$ refers to the knowledge ordering in three-valued logic. This is a nice theorem and gives rise to the following definition:

**Theorem 10.3.1 (WFS, [BD98b])**
*There exists the weakest semantics satisfying our four principles Elimination of Tautologies, Reduction, Subsumption and GPPE. This semantics is called wellfounded semantics WFS.*

It can also be shown, that for propositional programs, our transformations can be applied to compute this semantics.

**Theorem 10.3.2 (Confluent Calculus for WFS,[BD98a])**
*The calculus consisting of these four transformations is confluent, i.e. whenever we arrive at an irreducible program, it is uniquely determined. The order of the transformations does not matter.*

*For finite propositional programs, it is also terminating: any program $P$ is therefore associated a unique normalform $res(P)$. The wellfounded semantics of $P$ can be read off from $res(P)$ as follows*

$$WFS(P) = \{A : A \leftarrow \ \in res(P)\} \cup \{\text{ not } A : A \textit{ is in no head of } res(P)\}$$

We note that the size of the residual program is in general *exponential* in the size of the original program. Recently it was shown in [BZF97] how a small modification of the residual program, which still satisfies the nice characterization of computing WFS as given in Theorem 10.3.2, results in a *polynomial* computation.

Therefore the wellfounded semantics associates to every program $P$ with negation a set consisting of atoms and default-atoms. This set is a 3-valued model of $P$. It can happen, of course, that this set is empty. But it is always consistent, i.e. it does not contain an atom $A$ and its negation $\textit{not } A$. Moreover, it extends SLDNF: whenever SLDNF derives an atom or default-atom and does not flounder, then WFS derives it as well. Therefore the two examples of Section 9.4 are handled in the right way. But also for Example 10.2.1 we get the desired answers.

Let us discuss whether *every sequence of program transformations terminates*, i.e. if our calculus is *strongly terminating*. Already the simple program consisting of just the loop "$A \leftarrow A$" shows a problem. Applying GPPE leads to the same program, so GPPE can be applied infinitely often without leading to the residual program. Of course, in this particular case an application of the *Elimination of Tautologies* immediately leads to the residual program. But still another problem can occur[5]:

$$\begin{array}{llcl} P_{loop}: & A & \leftarrow & B \\ & B & \leftarrow & A \\ & C & \leftarrow & A, \neg C \end{array}$$

If we apply GPPE$_A$ to the third clause, this clause is replaced by $C \leftarrow B, \neg C$. We can now apply GPPE$_B$ again to this clause and get the original program. So we have an oscillation

$$GPPE_B \circ GPPE_A(P_{loop}) = P_{loop}$$

To summarize, not every sequence of transformations terminates. But a simple additional property will ensure this.

**Definition 10.3.3 (Fair Sequences)**
*We call a sequence of program transformations* fair*, if in the corresponding sequence of programs*

1. *every positive body-atom is eventually removed (either by removing the whole clause using a suitable transformation or by an application of GPPE), and*
2. *every tautology clause is eventually removed (either by applying* Elimination of Tautologies *or another suitable transformation).*

**Theorem 10.3.4 (Strong Termination for Fair Sequences)**
*Our calculus of transformations is strongly terminating for fair sequences of transformations. Such fair sequences therefore always lead to the residual program.*

As we said above, loop-checking is in general undecidable. Therefore WFS is in the most general case where variables and function-symbols are allowed, undecidable. Only for finite propositional programs it is decidable. In fact, it is of quadratic complexity (seeChapter 12).

Let us end this section with another example, which contains negation.

**Example 10.3.1 (Van Gelder's Example)**
*Assume we are describing a two-players game like checkers. The two players alternately move a stone on a board. The moving player wins when his opponent has no more move to make. We can formalize that by*

- *wins(x) ← move_from_to(x,y),* not *wins(y)*

*meaning that*

- *the situation x is won (for the moving player A), if he can lead over*[6] *to a situation y that can never be won for B.*

[5]brought to our attention by Frieder Stolzenburg
[6]With the help of a regular move, given by the relation $move\_from\_to/2$.

*Assume we also have* $move\_from\_to(a,b)$, $move\_from\_to(b,a)$ *and* $move\_from\_to(b,c)$. *Our query to this program* $P_{game}$ *is* ?-$wins(b)$ *Here we have no problems with floundering, but using SLDNF we get an infinite sequence of oscillating SLD-trees (none of which finitely fails).*

WFS, however, derives the right results

$$WFS(P_{game}) = \{\textit{ not } wins(c), wins(b), \textit{ not } wins(a)\}$$

which matches completely with our intuitions.

## 10.4 The Stable Semantics: STABLE

We defined WFS as the weakest semantics satisfying our four principles. This already indicates that there are even stronger semantics. One of the main competing approaches is the stable semantics STABLE. The stable semantics associates to any program $P$ a set of 2-valued models, like classical predicate logic. STABLE satisfies the following property, in addition to those that have been already introduced:

**Principle 10.4.1 (Elimination of Contradictions)**
*Suppose a program* $P$ *has a rule which contains the same atom* $A$ *and* not $A$ *in its body. Then we can eliminate this rule without changing the semantics.*

This principle can be used, in conjunction with the others to define the stable semantics

**Theorem 10.4.2 (STABLE,[BD97])**
*There exists the weakest semantics satisfying our five principles Elimination of Tautologies, Reduction, Subsumption, GPPE and Elimination of Contradictions.*

If a semantics SEM satisfies *Elimination of Contradictions* it is based on 2-valued models ([BD97]). The underlying idea of STABLE is that any atom in an intended model should have a definite reason to be true or false. This idea was made explicit in [BF91a, BF91b] and, independently, in [GL88]. We use the latter terminology and introduce the Gelfond-Lifschitz transformation: for a program $P$ and a model $N \subseteq B_P$ we define

$$P^N := \{rule^N : rule \in P\}$$

where $rule := A \leftarrow B_1, \ldots, B_n, \textit{not } C_1, \ldots, \textit{not } C_m$ is transformed as follows

$$(rule)^N := \begin{cases} A \leftarrow B_1, \ldots, B_n, & \text{if } \forall j : C_j \notin N, \\ \mathbf{t}, & \text{otherwise.} \end{cases}$$

Note that $P^N$ is always a *definite* program. We can therefore compute its least Herbrand model $M_{P^N}$ and check whether it coincides with the model $N$ with which we started:

**Definition 10.4.3 (STABLE)**
$N$ *is called a* stable *model*[7] *of* $P$ *if and only if* $M_{P^N} = N$.

[7]Note that we only consider Herbrand models.

What is the relationship between STABLE and WFS? We have seen that they are based on rather identical principles.

- Stable models $N$ extend WFS: $l \in$ WFS$(P)$ implies $N \models l$.
- If WFS$(P)$ is two-valued, then WFS$(P)$ is the unique stable model.

But there are also differences. We refer to Example 10.3.1 and consider the program $P$ consisting of the clause

$$wins(x) \leftarrow move\_from\_to(x, y),\ \textit{not}\ wins(y)$$

together with the following facts: $move\_from\_to(a, b)$, $move\_from\_to(b, a)$, as well as $move\_from\_to(b, c)$, and $move\_from\_to(c, d)$. In this particular case we have two stable models: $\{wins(a),\ wins(c)\}$ and $\{wins(b),\ wins(c)\}$ and therefore

$$\text{WFS}(P) = \{wins(c),\ \textit{not}\ wins(d)\} = \bigcap_{\mathcal{N} \text{ a stable model of } P} \mathcal{N}.$$

This means that the 3-valued wellfounded model is exactly the set of all atoms or default-atoms true in all stable models. But this is not always the case, as the program of $P_{splitting}$ shows:

**Example 10.4.1 (Reasoning by cases)**

$$\begin{array}{llll} P_{splitting}: & a & \leftarrow & \text{not } b \\ & b & \leftarrow & \text{not } a \\ & p & \leftarrow & a \\ & p & \leftarrow & b \end{array}$$

*Although neither $a$, nor $b$ can be derived in any semantics based on two-valued models (as STABLE for example), the disjunction $a \vee b$, thus also $p$, is true. In this way the example is handled by the completion semantics, too. WFS($P$), however, is empty; if the WFS cannot decide between $a$ or* not $a$*, then $a$ is undefined.*

The main differences between STABLE and WFS are

- STABLE is not always consistent,
- STABLE does not allow for a goal-oriented implementation.

The inconsistency comes from odd, negative cycles

$$STABLE(p \leftarrow \textit{not}\ p) = \emptyset.$$

The idea to consider 2-valued models for a semantics neccessarily implies its inconsistency ([BD95b]). Note that $WFS(p \leftarrow \textit{not}\ p) = \{\emptyset\}$ which is quite different! Sufficient criteria for the existence of stable models are contained in [Dun92, Fag93].

That STABLE does not allow for a Top-Down evaluation is a more serious drawback and has nothing to do with inconsistency. This behaviour led Dix to define the notion of *Relevance* and *Modularity* (see Section 13.1 and [Dix92a, Dix92b, Dix95b]. Recently, Bry reinvented *Modularity* (he termed it *compositionality*) and argued that a semantics should satisfy it.

**Example 10.4.2 (STABLE is not Goal-Oriented)**

$$\begin{array}{llll@{\qquad}llll} P_{rel(a)}: & a & \leftarrow & \text{not } b & P: & a & \leftarrow & \text{not } b \\ & b & \leftarrow & \text{not } a & & b & \leftarrow & \text{not } a \\ & & & & & p & \leftarrow & \text{not } p \\ & & & & & p & \leftarrow & a \end{array}$$

*$P_{rel(a)}$ is the subprogram of $P$ that consists of all rules that are relevant to answer the query ?-a It has two stable models $\{a\}$ and $\{b\}$ — $a$ is not true in all of them. But the program $P$ has the unique stable model $\{p, a\}$, so $a$ is true in all stable models of $P$.*

The last example shows that the truthvalue of an atom $a$ also depends on atoms that are totally unrelated with $a$! This is considered a drawback of STABLE by many people. Note that a straightforward modification of STABLE is not possible ([DM94b, DM94c]).

We end this section with another description of WFS and STABLE that will be useful in later sections. It was introduced in [BS91, BS92]:

**Definition 10.4.4 (Antimonotone Operator $\gamma_P$)**
*For a program $P$ and a set $N \subset B_P$ we define an operator $\gamma_P$ mapping Herbrand-structures to Herbrand structures:*

$$\gamma_P(N) := M_{P^N}.$$

*It is easy to see that $\gamma_P$ is antimonotone. Therefore its twofold application $\gamma^2$ is monotone ([Tar55]).*

Obviously, the stable models of a program $P$ are exactly the fixpoints of $\gamma_P$. This is just a reformulation of Definition 10.4.3. WFS is related to $\gamma$ as follows

**Theorem 10.4.5 (WFS and $\gamma^2$)**
*A positive atom $A$ is in WFS(P) if and only if $A \in lfp(\gamma_P^2)$. A default-atom* not $A$ *is in WFS(P) if and only if $A \notin gfp(\gamma_P^2)$:*

$$WFS(P) \ = \ lfp(\gamma_P^2) \cup \{ \text{not } A : \ A \notin gfp(\gamma_P^2)\}.$$

*Atom or default-atoms that do occur in neither of the two sets are undefined.*

# 11 KR with Disjunctive Programs

In this section we will extend our programs to *disjunctive* statements. In Knowledge Representation it often occurs that we know $A \vee B \vee C$ without being sure which of these propositions hold. In fact, such a disjunction leaves it open: there might be states in the world where $A$ holds or $B$ or $C$ or any combination thereof. Nevertheless, we can have information that $A$ implies $D$ and $B$ implies $D$ and $C$ implies $D$ from which we would like to derive that $D$ holds for sure. We will see in Chapter 12 that even with disjunctive programs without negation we can already express relations which belong to the second level of the polynomial hierarchy.

Concerning the right semantics for such programs, we are in the same situation as in Section 10 — for positive programs there is general agreement while for disjunctive programs with default-negation there exist several competing approaches.

We present in Section 11.1 the generalized closed world assumption introduced by Minker. In Section 11.2 we show that our definition of WFS from Section 10.3 immediately carries over to the disjunctive case. The original definition of STABLE (Definition 10.4.3) also carries over — we present it in Section 11.3. We mention some other attempts to define disjunctive semantics in Section 11.4. Finally we discuss complexity and expressibility in Chapter 12.

## 11.1 GCWA

GCWA was defined by Minker ([Min82]) and can bee seen as a refined version of the CWA introduced by Reiter ([Rei78]):

**Definition 11.1.1 (CWA)**

$$\mathrm{CWA}(\mathrm{DB}) = \mathrm{DB} \cup \{\neg P(t) : \ \mathrm{DB} \not\models P(t)\} ,$$

*where $P(t)$ is a ground predicate instance.*

That is, if a ground term cannot be inferred from the database, its negation is added to the closure. A weakness of CWA is that already for very simple theories, like $A \vee B$ it is inconsistent. Since neither $\neg A$ nor $\neg B$ is derivable, we have to add them both which makes the whole set inconsistent.

GCWA is defined for positive disjunctive programs consisting of rules of the form

$$A_1 \vee \ldots \vee A_n \leftarrow B_1, \ldots, B_m$$

by declaring all the minimal models to be the intended ones:

**Definition 11.1.2 (GCWA)**
*The generalized closed world assumption GCWA of $P$ is the semantics given by the set of all minimal Herbrand models of $P$:*

$$\textit{GCWA}(P) := \textit{Min-MOD}(P)$$

Originally, Minker denoted by GCWA(P) a set of negative atoms with the property that $P \cup GCWA(P) \models$ *not* $A$ if and only if MinMOD(P)$\models$ *not* $A$ but we prefer here to denote by GCWA a semantics in the sense of Definition 9.5.1.

GCWA is very important because it plays the same role for positive disjunctive programs as the least Herbrand model $M_P$ does for definite programs. In addition it turns out that some semantics SEM defined for arbitrary disjunctive programs (i. e. with default-negation) can be characterized, sometimes even implemented, by reducing them to positive programs and then applying recursively GCWA. Thus an appropriate procedure iterating GCWA can "implement" such semantics SEM.

Note also that as far as we consider deriving *positive* disjunctions, we stay entirely within classical logic — a positive disjunction is true in GCWA if and only if it follows from the program considered as a classical theory. Therefore this task can be accomplished be methods and techniques developed in theorem proving in the last 30 years. In fact this was one of the main starting points of the DisLoP-project in Koblenz (see Section 13.2). As an example, in [BFS95] the authors show how to *compute* definite answers from a positive disjunctive program using an underlying theorem prover. While indefinite answers are easy to obtain (because most of them are trivial) definite ones are much harder to obtain.

Of course, GCWA is nothing else than Circumscription (see Section A.4) for a special class of theories. Methods developed for CIRC can be used to compute GCWA. For recent approaches that work in *polynomial space* see [Nie96a, Nie96b].

In Sections 9 and 10 we have introduced the general notion of a semantics and various principles. Do they carry over to the disjunctive case? Fortunately, the answer is yes. In addition, GCWA not only satisfies all these properties, it is also uniquely characterized by them as the next theorem shows (we will introduce these properties in the next section).

**Theorem 11.1.3 (Characterization of GCWA, [BD97])**
*Let SEM be a semantics satisfying* GPPE *and* Elimination of Tautologies.

**a)** *Then: SEM$(P) \subseteq$ Min-MOD$_{2-val}(P)$ for positive disj. programs $P$.*

*I.e. any such semantics is already based on 2-valued minimal models. In particular, GCWA is the weakest semantics with these properties.*

**b)** *If SEM is non-trivial and satisfies in addition*[1] Isomorphy *and* Relevance*, then it coincides with GCWA on positive disjunctive programs.*

We end this section with the discussion of a well-known example that can not be handled adequately by Circumscription:

**Example 11.1.1 (Poole's Broken Arm)**
*Usually, a person's left arm is useable. But if the left arm is broken, it is an exception. The same statement holds for the right arm. Suppose that we saw Fred yesterday with a broken arm but we do not remember if it was the left or the right one. We also know that Fred can make out a cheque if he has at least one useable arm (he is ambidextrous) but that he is completely disabled if both arms are broken. Here is the natural formalization:*

$$\begin{array}{lll}
left_use(x) & \leftarrow & \text{not } ab(left, x) \\
ab(left, x) & \leftarrow & left_brok(x) \\
right_use(x) & \leftarrow & \text{not } ab(right, x) \\
ab(right, x) & \leftarrow & right_brok(x) \\
left_brok(Fred) \vee right_brok(Fred) & \leftarrow & \\
make_cheque(x) & \leftarrow & left_use(x) \\
make_cheque(x) & \leftarrow & right_use(x) \\
disabled(x) & \leftarrow & left_brok(x), right_brok(x)
\end{array}$$

*Of course, we expect that Fred is able to make out a cheque even without knowing which arm he is actually using. Also we derive that he is not (completely) disabled.*

For general Circumscription, the problem is to rule out the unintended model where both arms are broken and Fred is disabled. As we will see later, both D-WFS and DSTABLE derive that Fred is not disabled but only DSTABLE is strong enough to also conclude that Fred can make out a cheque.

## 11.2 D-WFS

Before we can state the definition of D-WFS we have to extend our principles to disjunctive programs with default-negation. We abbreviate general rules

$$A_1 \vee \ldots \vee A_k \leftarrow B_1, \ldots, B_m, \textit{not } C_1, \ldots, \textit{not } C_n,$$

by

$$\mathcal{A} \leftarrow \mathcal{B}^+, \textit{not } \mathcal{B}^-$$

where $\mathcal{A} := \{A_1, \ldots, A_k\}$, $\mathcal{B}^+ := \{B_1, \ldots, B_m\}$, $\mathcal{B}^- := \{C_1, \ldots, C_n\}$. We also generalize our notion of a semantics slightly:

**Definition 11.2.1 (Operator $\mid\!\sim$, Semantics $\mathcal{S}_{\mid\sim}$)**
*By a semantic operator $\mid\!\sim$ we mean a binary relation between logic programs and pure disjunctions which satisfies the following three arguably obvious conditions:*

*1.* Right Weakening: *If $P \mid\!\sim \psi$ and $\psi \subseteq \psi'$*[2]*, then $P \mid\!\sim \psi'$.*

[1] See Section 13.1 for the precise definitions of Relevance and Isomorphy.

[2] I. e. $\psi$ is a subdisjunction of $\psi'$.

2. Necessarily True: *If* $\mathcal{A} \leftarrow true \in P$ *for a disjunction* $\mathcal{A}$*, then* $P \mathrel{|\!\sim} \mathcal{A}$.
3. Necessarily False: *If* $A \notin Head\_atoms(P)^3$ *for some* $\mathcal{L}$*-ground atom* $A$*, then* $P \mathrel{|\!\sim}$ not $A$.

*Given such an operator* $\mathrel{|\!\sim}$ *and a logic program* $P$*, by the semantics* $\mathcal{S}_{|\!\sim}(P)$ *of* $P$ *determined by* $\mathrel{|\!\sim}$ *we mean the set of all pure disjunctions derivable by* $\mathrel{|\!\sim}$ *from* $P$*, i.e.,* $\mathcal{S}_{|\!\sim}(P) := \{\psi \mid P \mathrel{|\!\sim} \psi\}$.

In order to give a unified treatment in the sequel, we introduce the following notion:

**Definition 11.2.2 (Invariance of $\mathrel{|\!\sim}$ under a Transformation)**
*Suppose that a program transformation* Trans : $P \mapsto$ Trans$(P)$ *mapping logic programs into logic programs is given. We say that the operator* $\mathrel{|\!\sim}$ *is invariant under* Trans *(or that* Trans *is a* $\mathrel{|\!\sim}$*-equivalence transformation) iff*

$$P \mathrel{|\!\sim} \psi \iff \text{Trans}(P) \mathrel{|\!\sim} \psi$$

*for any pure disjunction* $\psi$ *and any program* $P$.

All our principles introduced below can now be naturally extended.

**Definition 11.2.3 (Elimination of Tautologies, Non-Minimal Rules)**
*Semantics* $\mathcal{S}_{|\!\sim}$ *satisfies* **a)** *the* Elimination of Tautologies*, resp.* **b)** *the* Elimination of Non-Minimal Rules *iff* $\mathrel{|\!\sim}$ *is invariant under the following transformations:*

**a)** *Delete a rule* $\mathcal{A} \leftarrow \mathcal{B}^+ \wedge$ not $\mathcal{B}^-$ *with* $\mathcal{A} \cap \mathcal{B}^+ \neq \emptyset$.

**b)** *Delete a rule* $\mathcal{A} \leftarrow \mathcal{B}^+ \wedge$ not $\mathcal{B}^-$ *if there is another rule* $\mathcal{A}' \leftarrow \mathcal{B}^{+\prime} \wedge$ not $\mathcal{B}^{-\prime}$ *with* $\mathcal{A}' \subseteq \mathcal{A}$*,* $\mathcal{B}^{+\prime} \subseteq \mathcal{B}^+$*, and* $\mathcal{B}^{-\prime} \subseteq \mathcal{B}^-$.

Our partial evaluation principle has now to take into account disjunctive heads. The following definition was introduced independently by Sakama/Seki and Brass/Dix ([BD94, BD97, SS94]):

**Definition 11.2.4 (GPPE)**
*Semantics* $\mathcal{S}_{|\!\sim}$ *satisfies GPPE iff it is invariant under the following transformation:* Replace a rule $\mathcal{A} \leftarrow \mathcal{B}^+ \wedge$ *not* $\mathcal{B}^-$ where $\mathcal{B}^+$ contains a distinguished atom $B$ by the rules

$$\mathcal{A} \cup \big(\mathcal{A}_i \setminus \{B\}\big) \;\leftarrow\; \big(\mathcal{B}^+ \setminus \{B\}\big) \cup \mathcal{B}_i^+ \;\wedge\; \textit{not}\ \big(\mathcal{B}^- \cup \mathcal{B}_i^-\big)\ (i = 1, \ldots, n)$$

*where* $\mathcal{A}_i \leftarrow \mathcal{B}_i^+ \wedge$ not $\mathcal{B}_i^-$ *(*$i = 1, \ldots, n$*) are all the rules with* $B \in \mathcal{A}_i$.

Note that we are free to select a specific positive occurrence of an atom $B$ and then perform the transformation. The new rules are obtained by replacing $B$ by the bodies of all rules $r$ with head literal $B$ and adding the remaining head atoms of $r$ to the head of the new rule.

Here is the analogue of Principle 10.2.1:

[3] We denote by $Head\_atoms(P)$ the set of all (instantiations of) atoms ocurring in some rule-head of $P$.

**Definition 11.2.5 (Positive and Negative Reduction)**
*Semantics $\mathcal{S}_{\mid\sim}$ satisfies* **a)** Positive*, resp.* **b)** Negative *Reduction iff $\mid\sim$ is invariant under the following transformations:*

**a)** *Replace* $\mathcal{A} \leftarrow \mathcal{B}^+ \wedge \text{not}\ \mathcal{B}^-$ *by* $\mathcal{A} \leftarrow \mathcal{B}^+ \wedge \text{not}\ \big(\mathcal{B}^- \cap Head\_atoms(P)\big)$.

**b)** *Delete* $\mathcal{A} \leftarrow \mathcal{B}^+ \wedge \text{not}\ \mathcal{B}^-$ *if there is a rule* $\mathcal{A}' \leftarrow true$ *with* $\mathcal{A}' \subseteq \mathcal{B}^-$.

Now the definition of a disjunctive counterpart of WFS is straightforward:

**Definition 11.2.6 (D-WFS)**
*There exists the weakest semantics satisfying positive and negative Reduction, GPPE, Elimination of Tautologies and non-minimal Rules. We call this semantics D-WFS.*

As it was the case for WFS, our calculus of transformations is also confluent ([BD98a, BD96a]).

**Theorem 11.2.7 (Confluent Calculus for D-WFS)**
*The calculus consisting of our four transformations is confluent and terminating for propositional programs. I.e. we always arrive at an irreducible program, which is uniquely determined. The order of the transformations does not matter.*

*Therefore any program $P$ is associated a unique normalform $res(P)$. The disjunctive wellfounded semantics of $P$ can be read off from $res(P)$ as follows*

$$\psi \in \textit{D-WFS}(P) \iff \begin{array}{l} \textit{there is } \mathcal{A} \subseteq \psi \textit{ with } \mathcal{A} \leftarrow true \in res(P) \textit{ or} \\ \textit{there is } \text{not}\ A \in \psi \textit{ and } A \notin Head\_atoms(res(P)). \end{array}$$

Note that the original definition of WFS, or any of its equivalent characterizations, does not carry over to disjunctive programs in a natural way.

Let us see how Example 11.1.1 is handled by D-WFS. Applying GPPE and Reduction gives us the following residual program (we consider just the $Fred$-instantiations):

$$\begin{array}{lcl}
left_use(F) & \leftarrow & \textit{not } ab(left, F) \\
ab(left, F) \vee right_brok(F) & \leftarrow & \\
right_use(F) & \leftarrow & \textit{not } ab(right, F) \\
ab(right, F) \vee left_brok(F) & \leftarrow & \\
left_brok(F) \vee right_brok(F) & \leftarrow & \\
make_cheque(F) & \leftarrow & \textit{not } ab(left, F) \\
make_cheque(F) & \leftarrow & \textit{not } ab(right, F)
\end{array}$$

Therefore we derive *not* $disabled(F)$, because it does not appear in any head of the residual program. All the remaining atoms are undefined.

Two properties of D-WFS are worth noticing

- For positive disjunctive programs, D-WFS coincides with GCWA.
- For non-disjunctive programs with negation, D-WFS coincides with WFS.

## 11.3 DSTABLE

Unlike the wellfounded semantics, the original definition of stable models carries over to disjunctive programs quite easily:

**Definition 11.3.1 (DSTABLE)**
*$N$ is called a* stable *model*[4] *of $P$ if and only if $N \in$ Min-Mod$(P^N)$.*

In the last definition $P^N$ is the positive disjunctive program obtained from $P$ by applying the Gelfond/Lifschitz transformation (as introduced before Definition 10.4.3 — its generalization to disjunctive programs is obvious).

Analogously to D-WFS the following two properties of DSTABLE hold:

- For positive disjunctive programs, DSTABLE coincides with GCWA.
- For non-disjunctive programs with negation, DSTABLE coincides with STABLE.

What about our transformations introduced to define D-WFS? Do they hold for DSTABLE? Yes, they are indeed true. The most difficult proof is the one for GPPE. It was proved in [BD95c, SS94] independently that stable models are preserved under GPPE. Moreover, Brass/Dix proved in [BD97] that STABLE can be almost uniquely determined by GPPE:

**Theorem 11.3.2 (Characterization of DSTABLE)**
*Let SEM be a semantics satisfying* GPPE*,* Elimination of Tautologies*, and* Elimination of Contradictions*. Then: SEM$(P) \subseteq$ STABLE$(P)$.*

*Moreover, DSTABLE is the weakest semantics satisfying these properties.*

DSTABLE is stronger than D-WFS as can be seen from Example 11.1.1. There we have exactly two stable models

1. $left\_use(F)$, *not* $ab(left, F)$, $ab(right, F)$, *not* $right\_use(F)$, $right\_brok(F)$, *not* $left\_brok(F)$, $make\_cheque(F)$, *not* $disabled(F)$,
2. $right\_use(F)$, *not* $ab(right, F)$, $ab(left, F)$, *not* $left\_use(F)$, $left\_brok(F)$, *not* $right\_brok(F)$, $make\_cheque(F)$, *not* $disabled(F)$.

In all of them, Fred is not disabled and can make out a cheque.

Of course, DSTABLE inherits the shortcomings of STABLE such as inconsistency and no goal-orientedness.

## 11.4 Other Semantics

In this section we just want to mention some other disjunctive semantics proposed in the last years. First, there are semantics differing from GCWA in that they interpret "$\vee$" inclusively, rather than exclusively (like GCWA does).

[4]Note that we only consider Herbrand models.

The corresponding semantics is called WGCWA (see [RLM89]) and is equivalent to the disjunctive database rule DDR considered in [RT88]. WGCWA has been considered as a more tractable (weaker) variant of GCWA (from the procedural point of view developed in [RLM89]).

**Example 11.4.1 (Inclusive versus Exclusive)**

$$\begin{array}{llll} P_{incl/excl}: & a \vee b & & \\ & c & \leftarrow & a, b \end{array}$$

*Under an* exclusive *interpretation,* not $c$ *should be derivable. Indeed, this is the case:* *GCWA*$(P_{incl/excl}) = \{\text{ not } c\}$.
*Under an* inclusive *interpretation however,* not $c$ *should not be derivable. This is the case for WGCWA:* $M_{P^*_{incl/excl}} = \{a, b, c\}$. *The set of* positive *derivable literals is in both cases the same! If we replace the first clause with* $a$ *or* $b$*, then* not $c$ *is derivable.*

There are extensions of WGCWA to disjunctive programs with negation: [SI93, Ros89, Dix92b, DM94a].

There is also the book [LMR92] — the first in-depth-study of disjunctive semantics with negation. However, we feel that these semantics have a drawback in that they are based on rather technical, complicated and not-easy-to-understand fixpoint definitions. These definitions leave a lot of room for modifications. But small modifications usually have a tremendous impact on the outcoming semantics. In addition these semantics do not allow for a proper treatment of definitional extensions (see Example 13.1.1).

Let us discuss one more time the *inclusive/exclusive* meaning of $\vee$. Chiaki Sakama noted that inclusive vs. exclusive is not always fully determined by the underlying semantics. For example in the program $p \vee q$, $p \leftarrow q$, $q \leftarrow p$, GCWA derives both $p$ and $q$ so it is better to say that GCWA *tends to interpret* $\vee$ *exclusively unless specified otherwise*. But weaker semantics cannot specify exclusive $\vee$. Therefore Sakama proposed in [Sak89] to consider programs with two different kinds of semantics: one for inclusive and one for exclusive behaviour. He defined a corresponding semantics, called *possible model semantics* PMS (see also [SI93, SI94]). Chan introduced this same idea under the name possible world semantics in [Cha93]. PMS has the nice feature that it lies on the first level of the polynomial hierarchy (see [**?**]).

Other approaches are due to Przymusinski: *stationary*-semantics $\mathcal{STN}$ defined in [Prz91a] and *static*-semantics STATIC defined in [Prz95, BDP96a]. STATIC is an improvement of his former stationary semantics that is very close to D-WFS: in fact it coincides with D-WFS if it is restricted to a common sublanguage ([BDNP96, BDNP98]). This approach also allows us to consider a larger class of programs, namely those that contain *not* $(A_1 \wedge \ldots \wedge A_n)$ in their bodies. Such programs are more expressible and therefore turn out to be even better suited for representation tasks.

Another approach differing from GCWA and WGCWA is considered in [DGM94, DGM96, Bon93].

# 12 Complexity and Expressibility

In this section we collect some complexity and expressibility results for the semantics considered so far. Complexity is about the question of how expensive decision procedures for certain questions about semantics (e.g. derivability of literals) are. The expressive power, on the other hand, is concerned with what relations can be expressed (i.e. simulated) given certain classes of programs.

For the exact terminology, definitions and results presented in this section we refer the reader to the following interesting overviews [Sch90, Sch92, CS93]. Further results are contained in [EGM93, Sac93, CS90, EG93].

## 12.1 Complexity

The reason why NMR-semantics are, in the general case (free variables and function symbols), undecidable is strongly related to loop-checking. Let us consider the program

$$P(x) \leftarrow P(f(x))$$

or, equivalently, the infinite propositional program

$$p_0 \leftarrow p_1,\ p_1 \leftarrow p_2,\ \ldots\ , p_i \leftarrow p_{i+1}, \ldots$$

Any NMR-semantics should derive "*not* $P(t)$" (resp. "*not* $p_i$") for all terms $t$, but a procedure to detect such infinite loops is impossible in general. Our principles *GPPE* and *Elimination of Tautologies* can detect *finite* loops.

From a model theoretic point of view it is easy to *define* a semantics that derives "*not* $P(t)$": we could just take all minimal Herbrand models as the intended semantics. Of course, this does not change the general undecidability.

We consider mainly the complexity of deciding if a *given ground atom or default-atom is contained in the respective semantics* (i.e. if it is true in all *intended* models).

Our complexity tables consist of two columns. For the 1. column, we consider arbitrary first-order programs with function symbols. We therefore get undecidability results of varying strength. Since we restrict to Herbrand models, we can assume (by standard recursive encoding techniques, like *Gödel-numberings*) that all models have universes which are subsets of the natural numbers $\mathbb{N}$. The completeness results mean that for every set of the respective complexity class there is a program that defines this set under the respective sceptical semantics. Unless indicated otherwise, there is no difference between deciding ground *atoms* or ground *negated atoms*.

For the 2. column, we consider *propositional* programs. Hence we get decidable problems of various degrees. We denote by $|P|$ the total length of the program and by $\#At$ the number of distinct proposition letters in $P$. See also [BED92, Sch92, Imi91, MRT92, Wit91b, JdL92] for more results on the complexity of propositional programs.

| | **Complexity** | |
|---|---|---|
| | *1. ord. prog. (with functions)* | *prop. prog. (no variables)* |
| $\mathbf{M_P}$ *($P$ is Horn)* | $A$: $\mathbf{\Sigma^0_1}$-compl.<br>*not* $A$: $\mathbf{\Pi^0_1}$-compl. | **linear** in $\|P\|$ |
| $\mathbf{M_P^{supp}}$ *($P$ is stratified)* | **arithm.**-compl. ($M_P^{supp}$ is $\mathbf{\Sigma^0_n}$) | **linear** in $\|P\|$ |
| **COMP** | $\mathbf{\Pi^1_1}$-compl. over $\mathbb{N}$ | **co-NP**-compl. |
| $\mathbf{COMP_3}$ | $\mathbf{\Pi^1_1}$-compl. over $\mathbb{N}$ | **linear** in $\|P\|$ |
| **STABLE** | $\mathbf{\Pi^1_1}$-compl. over $\mathbb{N}$ | **co-NP**-compl. |
| **REG-SEM** | $\mathbf{\Pi^1_1}$-compl. over $\mathbb{N}$ | **co-NP**-compl. |
| **WFS** | $\mathbf{\Pi^1_1}$-compl. over $\mathbb{N}$ | **linear** in $\#At \times \|P\|$ |
| **WFS′** | $\mathbf{\Pi^1_1}$-compl. over $\mathbb{N}$ | **co-NP**-compl. |
| $\mathbf{WFS^+}$ | $\mathbf{\Pi^1_1}$-compl. over $\mathbb{N}$ | **co-NP**-compl. |

Tabelle 12.1: Complexity of Non-Disjunctive Semantics

It is worth noting that in the general predicate logic case, all semantics are highly undecidable. The entries for comp and comp$_3$ are to be understood as restricted to Herbrand models.

In the propositional case, WFS is of quadratic complexity (a folklore result — for a proof see [Wit91a]), while STABLE is **co-NP**-complete. The low complexity of WFS can be traced back to Dowling and Gallier's result whereby satisfiability of Horn clauses can be tested in linear time([DG84]). In Dowling and Gallier's approach it is actually a minimal model of a Horn theory that is computed in linear time. Since minimal models of Horn theories are equivalent to closures of rules without negation the result is directly applicable to well-founded semantics for logic programs with default-atoms.

From the complexity point of view GCWA lies between CWA (which is $\Pi^0_1$-complete, see [AB90] and general Circumscription ($\Sigma^1_1$-complete, see [CEG92]): GCWA is $\Pi^0_2$-complete. For propositional programs we have to distinguish between deriving an *atom* or a *literal*. The first problem is **co-NP**-complete while the second is even $\Pi^P_2$-complete (see [Imi91]).

| | **Complexity** | |
|---|---|---|
| | *1. ord. prog. (with functions)* | *prop. prog. (no variables)* |
| **GCWA** ($P$ *is positive*) | $A$: $\mathbf{\Sigma_1^0}$-compl.<br>*not* $A$: $\mathbf{\Pi_2^0}$-compl. | $A$: **co-NP**-compl.<br>*not* $A$: $\mathbf{\Pi_2^P}$-compl. |
| **WGCWA** ($P$ *is positive*) | $A$: $\mathbf{\Sigma_1^0}$-compl.<br>*not* $A$: $\mathbf{\Pi_1^0}$-compl. | $A$: **co-NP**-compl.<br>*not* $A$: **linear** in $\|P\|$ |
| **PERFECT** ($P$ *is stratified*) | **arithm.**-compl. | $\mathbf{\Pi_2^P}$-compl. |
| **PMS** | not yet studied | **co-NP**-compl |
| **WPERFECT** ($P$ *is stratified*) | **arithm.**-compl. | $\mathbf{\Pi_2^P}$-compl. |
| **D-WFS** | $\mathbf{\Pi_1^1}$-compl. over $\mathbb{N}$ | $\mathbf{\Pi_2^P}$-compl. |
| **DSTABLE** | $\mathbf{\Pi_1^1}$-compl. over $\mathbb{N}$ | $\mathbf{\Pi_2^P}$-compl. |

Tabelle 12.2: Complexity of Disjunctive Semantics

For deriving negated literals *not* $A$, WGCWA is $\Pi_1^0$-complete (like CWA) and therefore "better" than GCWA ($\Pi_2^0$-complete). In the propositional case, WGCWA is polynomial while GCWA is $\Pi_2^p$-complete (both for the derivation of literals of the form *not* $A$).

## 12.2 Expressibility

To define the expressive power of classes of programs under certain semantics, one usually adopts the database viewpoint and distinguishes between EDB-relations (relations that do not appear in the head of a program) and IDB-relations (which are contained in some heads).

More generally, abstracting from programs and semantics, let us consider arbitrary relational systems. We fix a countable domain $DOM$ and consider a relational schema $\mathbf{R}$, which stands for finitely many relation symbols $R_1, \ldots, R_n$. An instance $\mathcal{D}$ of $\mathbf{R}$ is then a finite relational structure

$$\mathcal{D} := \langle U^{\mathcal{D}}, R_1^{\mathcal{D}}, \ldots R_n^{\mathcal{D}} \rangle$$

where $U^{\mathcal{D}} \subseteq DOM$ is finite, called the *universe*. We denote by $\mathbf{Inst}(\mathbf{R})$ the set of all instances over $\mathbf{R}$.

Here is the general notion of a *query-mapping*, which has to be distinguished from the notion of a query introduced in Definition 9.0.2:

**Definition 12.2.1 (Query Mapping)**
*Let two disjoint relational schemas* $\mathbf{R}, \mathbf{S}$ *be given. A query* $q$ *is a partial recursive mapping*

$$q : \mathbf{Inst}(\mathbf{R}) \to \mathbf{Inst}(\mathbf{S}),$$

*i.e.* $q$ *maps instances of* $\mathbf{R}$ *to instances of* $\mathbf{S}$. *We assume* $U^{\mathcal{D}} = U^{q(\mathcal{D})}$ *for all* $\mathcal{D} \in \mathbf{Inst}(\mathbf{R})$.

Here are some simple examples of queries:

**(union)** $q_{union}$**:** which maps two relations $R_1, R_2$ to their union $R_1 \cup R_2$,

**(negation)** $q_{neg}$ **:** which maps a relation $R$ to its complement $\overline{R}$,

**(trans. closure)** $q_{tc}$**:** which maps a relation $R$ to its transitive closure $R_{tc}$,

**(even)** $q_{even}$**:** which maps an instance $\mathcal{D}$ of $\mathbf{R}$ to “true” iff the universe $U^{\mathcal{D}}$ has an even number of elements.

Every query $q$ has associated a *recognition problem*: given an instance $\mathcal{D}_{\mathbf{R}}$ of $\mathbf{R}$ and a tuple $\underline{t}$ (matching the format of a relation $S$ of $q(\mathbf{R})$, i.e. the arities correspond and $\underline{t}$ is a sequence of elements of the universe of $U^{\mathcal{D}}$), decide whether

$$\underline{t} \in S^{q(\mathcal{D})}.$$

In all our previous examples, the output schema $\mathbf{S}$ consists just of one relation: $\mathbf{S} = \{S\}$ (single relation output). Note that the recognition problems of all these queries can be decided in log-SPACE, and therefore also in PTIME.

### 12.2.1 Queries in Classical Logic

In this subsection we will discuss how certain subclasses of formulae of first order logic give rise to formulate queries. The larger the allowed class, the more queries can be expressed. It is a well-known result that even the full class of first-order logic can not express the transitive closure query $q_{tc}$.

Let $\mathbf{R}$ be a schema and let a finite set of constants (elements from the underlying domain) be given. $\mathbf{R}$ together with these constants determine the underlying language $\mathcal{L}$. We define the following classes of formulae:

$C_1$: formulae built up from “$\exists, \wedge, =$” in the language $\mathcal{L}$.

$C_2$: formulae built up from “$\exists, \wedge, \vee, =$” in the language $\mathcal{L}$.

$C_3$: formulae built up from “$\exists, \wedge, \neg, =$” in the language $\mathcal{L}$. Obviously, $\forall$ and $\vee$ can be built from the other constructs as usual.

Every class $C_i$ of formulae determines in a natural way a class of queries as follows. For a fixed formula $\varphi$ with free variables $x_1, \ldots, x_n$ and every instance $\mathcal{D}$ of $\mathbf{R}$, the set

$$\{(t_1, \ldots, t_n) \mid \mathcal{D} \models \varphi[t_1, \ldots, t_n]\},$$

constitutes an instance of a schema $\mathbf{S} = \{S\}$, namely the interpretation of $S$. We define the universe of the instance to be the same as the universe of the original instance (the range of the terms $t_i$). Note that once the instance $\mathcal{D}$ of $\mathbf{R}$ is fixed, we can evaluate the formulae $\varphi$ in this instance and determine for which elements it is true. These elements then define the output relation $S$.

The class of queries corresponding to $C_i$ is then given by letting $\varphi$ range over all formulae in $C_i$. More specifically, the class corresponding to $C_1$ is called *conjunctive relational algebra*. The class corresponding to $C_2$ is called *positive relational algebra*. The class corresponding to $C_3$ is called *relational algebra*.

The following facts are easily shown:

1. The corresponding recognition problems of the classes $C_i$ are all in log-SPACE.
2. The query $q_{union}$ is not expressible in $C_1$.
3. The query $q_{negation}$ is not expressible in $C_2$.
4. The query $q_{tc}$ is not expressible in $C_3$.

Let us end this subsection with two additional remarks.

**Domain independence:** An important point in the general theory of expressibility over databases is that evaluating a query in an instance should be independent of the underlying domain (if it contains at least the underlying constants). This can be obtained by imposing additional syntactic restrictions on the class of formulae. The resulting formulae are called *domain independent*.

**Equality:** Our classes $C_i$ introduced above contain the equality predicate. One might argue that equality can be replaced by using multiple occurrences of the same variable or constant. This is only half of the truth. For example, $C_1$ formulae without equality are always satisfiable, while those with equality are not ($x = a \wedge x = b$ is not satisfiable in the Herbrand domain). The same holds for $C_2$ formulae. But the following turns out to hold:

- Satisfiable $C_1$ queries with equality are equivalent to $C_1$ queries without equality.
- satisfiable domain independent $C_2$ queries with equality are equivalent to domain independent $C_2$ queries without equality.

In contrast to the last two results, $C_3$ queries with equality are strictly more expressive than $C_3$ queries without equality (even if restricted to satisfiable domain independent queries).

### 12.2.2 Queries in Logic programs

How can we now define queries with programs under certain semantics? For a given program $P$ we can view any instance $\mathcal{D}$ of the (finite) EDB-relations as an *input* argument and then compute the (finite) IDB-relations (the *output*) under the respective sceptical semantics.

Thus the EDB-relations correspond to the $\mathbf{R}$ schema while the IDB-relations correspond to the **S** schema in Definition 12.2.1, the output of the query. Any such program $P$ determines a query under a semantics SEM as follows: an instance $\mathcal{D}$ over $\mathbf{R}$ determines the semantics $SEM(P + \mathcal{D})$, i.e. all IDB relations are determined and thus constitute an instance over the schema consisting of all IDB relations. One of them has to be distinguished as the answer (or output) relation. As our semantics are in general 3-valued, we map both the false as well as the undefined atoms to false in order to obtain a 2-valued relational instance.

Note that a query defined by a program determines implicitly the underlying language, simply by the constants occurring in $P$. In fact, one often restricts to *allowed* (also called *range-restricted*) programs (see Definition 10.1.4), because for such programs the underlying domain of a query does matter: evaluating formulae is independent of it.

If we compare the query classes $C_i$ introduced in the previous section, it is easy to see that

1. $C_1$ corresponds to definite programs consisting of exactly one rule,
2. $C_2$ corresponds to non-recursive definite programs,
3. $C_3$ corresponds to non-recursive normal programs,

all under the natural semantics. It can be easily shown that equality can be defined in these programs so that we can assume w.l.o.g. that equality is not contained.

We are in particular interested to express all relations of some complexity class (note that the complexity is always with respect to the finite relational database as input, the program is fixed). It is well-known that the relations *inductively definable* over $\mathcal{D}$, we denote them by **IND**($\mathcal{D}$) (or simply **IND** to avoid the explicit occurrence of the EDB), is a strict subclass of the relations that are *polynomial* over $\mathcal{D}$ (see [Bar75, Mos74, Gur88]). This class of queries is also called *fixpoint queries* and corresponds to $C_3$ enlarged with a least fixpoint operator $\mu$.

In contrast to classical logic, already the definite programs under the $M_P$ semantics capture the transitive closure query $q_{tc}$. But not even normal programs under WFS nor positive disjunctive programs under GCWA capture the even query $q_{even}$, and therefore not all polynomial time queries can be expressed. On the other hand, some $\mathbf{\Pi_2^P}$-hard queries can be expressed using positive disjunctive logic programs. Stable semantics for normal programs capture **co-NP**, while STABLE for disjunctive programs can express all of $\mathbf{\Pi_2^P}$. As an example, STABLE can express the satisfiability problem. WFS is not able to do this (unless the polynomial hierachy collapses).

| | **Expressibility** *1. ord. prog.* *(no functions)* |
|---|---|
| $\mathbf{M_P}$ *(P is Horn)* | $\subset$ **IND** (thus $\subset$ **P**) |
| **COMP** | $=$ **co-NP** |
| $\mathbf{COMP_3}$ | $=$ **IND** (thus $\subset$ **P**) |
| **STABLE** | $=$ **co-NP** |
| **REG-SEM** | $= \mathbf{\Pi_2^P}$ |
| **WFS** | $=$ **IND** (thus $\subset$ **P**) |
| **WFS′** | $=$ **IND** (thus $\subset$ **P**) |
| $\mathbf{WFS^+}$ | $=$ **IND** (thus $\subset$ **P**) |

Tabelle 12.3: Expressibility of Non-Disjunctive Semantics

| | **Expressibility** *1. ord. prog.* *(no functions)* |
|---|---|
| **GCWA** *(P is positive)* | $\subset \mathbf{\Pi_2^P}$ |
| **WGCWA** *(P is positive)* | $\subset \mathbf{\Pi_2^P}$ |
| **PERFECT** *(P is stratified)* | $= \mathbf{\Pi_2^P}$ |
| **WPERFECT** *(P is stratified)* | $= \mathbf{\Pi_2^P}$ |
| **D-WFS** | $= \mathbf{\Pi_2^P}$ |
| **DSTABLE** | $= \mathbf{\Pi_2^P}$ |

Tabelle 12.4: Expressibility of Disjunctive Semantics

# 13 What Do We Want and What Is Implemented?

In this part we first consider the question *Is there an optimal semantics?* (Section 13.1) and give in Section 13.2 an overview of all the existing implementations we are aware of. We also describe theoretical approaches that have not yet been implemented.

## 13.1 What is the Best Semantics?

Most probably there is no definite answer to the question in the title. Different knowledge representation tasks may ask for different semantics. Some might be better suited in special domains than others. What are reasonable properties that semantics should be checked against?

While many people defined in the last years new semantics by considering only few examples and appealing to their own personal intuitions they had about how these few examples should be handled, Dix tried to adjust and investigate abstract properties known in general nonmonotonic reasoning to semantics of logic programs ([Dix91, Dix92b, Dix95a, Dix95b]). He showed for example that WFS is cumulative and rational and that a semantics defined independently by Schlipf and Dix is the weakest extension of WFS satisfying *Cut* and *Supraclassicality*. Figure 13.1 illustrates the properties and the relationship between many semantics (note that $\leq_k$ refers to the knowledge ordering discussed in the beginning of Section 10.3: a semantics is *strictly weaker*, i.e. $\leq_k$, than another semantics if it derives strictly less literals). The distinction between *inclusive* and *exclusive* semantics is perhaps a bit misleading. We refer to the discussion in Section 11.4.

In Figure 13.2 normal programs are considered.

Besides such properties (which he calls *strong*) he defined also *weak* properties — these are conditions that any reasonable semantics should satisfy ([Dix92a, Dix95b]). The principles we have introduced in Sections 9, 10 belong to this sort. Let us take a closer look into some weak properties already mentioned (but not yet defined). We start with a property that is satisfied for any semantics we know:

**Definition 13.1.1 (Isomorphy)**
*A semantics SEM satisfies Isomorphy, if and only if*

$$SEM(\mathcal{I}(P)) = \mathcal{I}(SEM(P))$$

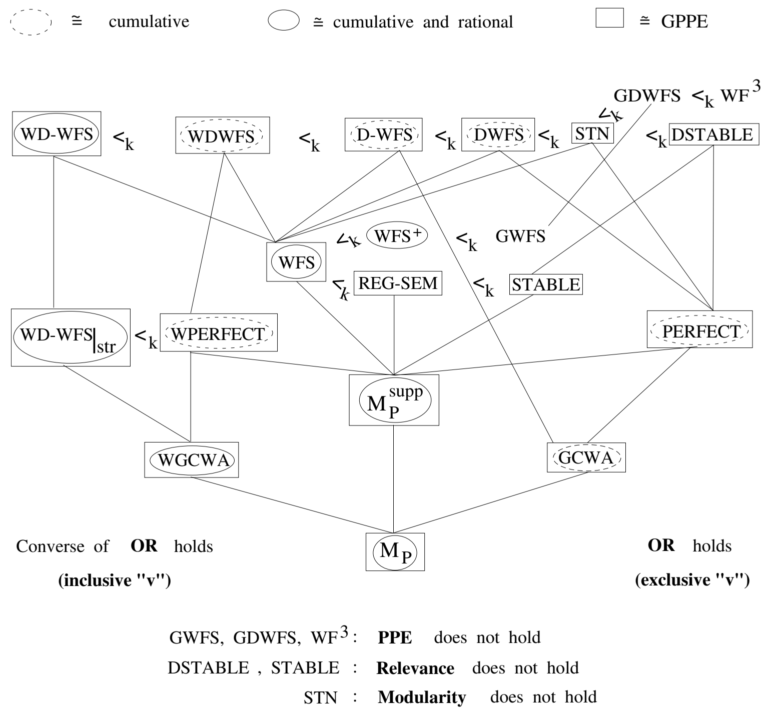



Abbildung 13.1: Semantics for Disjunctive Programs

*for all programs $P$ and isomorphisms $\mathcal{I}$ on the Herbrand base $B_P$.*

*Isomorphy* formalizes the intuition that a renaming of the program should have no influence on the semantics, as long as we also apply this same renaming to the semantics.

The next property gives a formal definition of the notion *Goal-Orientedness*. To state this conditions, we need the notion of the Dependency-Graph (Definition 10.1.4) and the two definitions

- $dependencies\_of(X) := \{A : X$ depends on $A\}$, and
- $rel\_rul(P, X)$ is the set of *relevant rules* of $P$ with respect to $X$, i.e. the set of rules that contain an $A \in dependencies\_of(X)$ in their head.

Given any semantics SEM and a program $P$, it is perfectly reasonable that the truth-value of a literal $L$, with respect to SEM(P), only depends on the subprogram formed from the *relevant rules* of $P$ with respect to $L$.[1] This idea is formalized by:

**Definition 13.1.2 (Relevance)**
*The principle of Relevance states:* $L \in SEM(P)$ *iff* $L \in SEM(rel\_rul(P, L))$.

[1] Let $dependencies\_of(not\ X) := dependencies\_of(X)$, and $rel\_rul(P, not\ X) := rel\_rul(P, X)$.

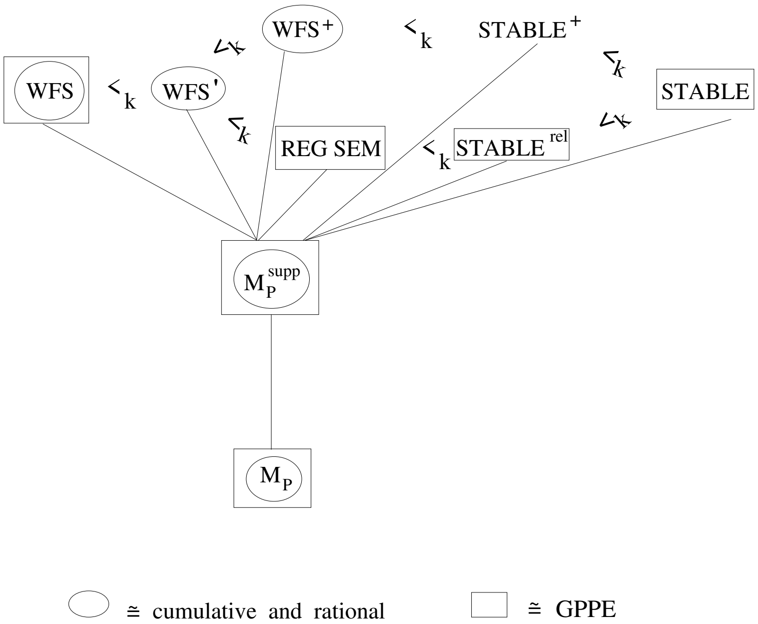



Abbildung 13.2: Semantics for Normal Programs

Note that the set of relevant rules of a program $P$ with respect to a literal $L$ contains all rules, that could ever contribute to $L$'s derivation (or to its nonderivability). In general, $L$ depends on a large set of atoms: $dependencies\_of(L) := \{A : L \text{ depends on } A\}$. But rules that do not contain these atoms in their heads, will never contribute to their derivation or non-derivation. Therefore, these rules should not affect the meaning of $L$ in $P$. STABLE does not satisfy this principle. This is due to the nonexistence of stable models by adding a clause "$c \leftarrow$ *not* $c$" to a program.

We have already introduced GPPE above. It is an extension of the following property for non-disjunctive programs:

**Definition 13.1.3 (PPE)**
*Let $P$ be an instantiated program and let the atom $c$ occur positively in $P$. Let $c \leftarrow rhs_1, \ldots, c \leftarrow rhs_n$ be all the rules of $P$ with $c$ in their heads.*

*Any program clause of the form "$head \leftarrow c, body$" can be replaced by the rules*

$$\begin{array}{rcl} head & \leftarrow & rhs_1, body \\ & \vdots & \\ head & \leftarrow & rhs_n, body \end{array}$$

*Note that the rules $c \leftarrow rhs_1, \ldots c \leftarrow rhs_n$ are not removed (in contrast to the weak version of PPE). We call the program obtained in this way $P'$.*

*The principle of partial evaluation is:* $SEM(P') = SEM(P)$.

GPPE is obtained from PPE by weakening the assumption that *c only occurs positively*. We note that most semantics defined by Minker and his group do not satisfy this condition:

**Example 13.1.1 (Extension-by-Definition, [Dix91])**
*We consider the following two programs:*

$$
\begin{array}{llll}
P_{GWFS}: & p & \leftarrow & \text{not } b \\
& a & \leftarrow & \text{not } b \\
& b & \leftarrow & c \\
& c & \leftarrow & p, \text{not } a
\end{array}
\qquad
\begin{array}{llll}
P_{GWFS_c}: & p & \leftarrow & \text{not } b \\
& a & \leftarrow & \text{not } b \\
& b & \leftarrow & p, \text{not } a \\
& c & \leftarrow & p, \text{not } a
\end{array}
$$

*GWFS($P_{GWFS}$) entails* not *c, because Min-MOD*$(P_{GWFS}) = \{\ \{p,a\},\ \{b\}\ \}$ *and thus also (by simple negation-as-failure reasoning)* not *b, p and a. Also Min-MOD(*$P_{GWFS_c}$*)=*$\{\ \{p,a\},\ \{b\}\ \}$ *but negation-as-failure can not be applied like before. Therefore GWFS(*$P_{GWFS_c}$*) does not entail* not *b, nor p nor a.*

$P_{GWFS\,c}$ *partial evaluates* $P_{GWFS}$: the last but one clause was transformed into another one by expanding the definition of $c$. Obviously, a semantics should assign the same meaning to these programs: unfortunately GWFS does not!

The next principle, *Modularity*, has some similarities with PPE. It enables us to compute a semantics by modularizing it into certain "subprograms" (formed of the relevant rules). The semantics of these modules can be computed first and the semantics of the whole program can be determined by reducing this program with literals that were already determined.

**Definition 13.1.4 (Modularity)**
*Let* $P = P_1 \cup P_2$ *and for every* $A \in B_{P_2}$: $rel\_rul(P, A) \subseteq P_2$.
*The principle of Modularity is:* $SEM(P) = SEM({P_1}^{SEM(P_2)} \cup P_2)$.

To illustrate this property, we compare the program

$$
\begin{array}{llll}
P_1: & b & \leftarrow & y \\
& y \vee x & & \\
& z \vee y & & \\
& m & \leftarrow & x, z, b \\
& y & \leftarrow & \textit{not } m
\end{array}
$$

with the union of the following two programs

$$
\begin{array}{llll}
P_1': & b & \leftarrow & y \\
& y \vee x & & a \\
& z \vee y & & a \\
& m & \leftarrow & x, z, b \\
& y & \leftarrow & \textit{not } m
\end{array}
\qquad
\begin{array}{llll}
P_2: & a & \leftarrow & e, \textit{not } g \\
& a & \leftarrow & f, \textit{not } g \\
& a & \leftarrow & f, \textit{not } e \\
& a & \leftarrow & g, \textit{not } e \\
& e \vee f \vee g & &
\end{array}
$$

| Properties of Logic-Programming Semantics | | | | | | | |
|---|---|---|---|---|---|---|---|
| Semantics | Reference | Domain | Taut. | GPPE | Red. | NMin. | Rel. |
| comp | Cla78 | Nondis. | — | • | • | • | — |
| GCWA | Min82 | Pos. | • | • | • | • | • |
| WGCWA | RosTop88 | Pos. | — | • | • | — | • |
| DSTABLE | GelLif91 | Dis. | • | • | • | • | — |
| WFS | vGeld.etal88 | Nondis. | • | • | • | • | • |
| $\mathcal{STN}$ | Prz91 | Dis. | • | • | • | • | • |
| STATIC | Prz95 | Dis. | • | • | • | • | • |
| D-WFS | BraDix95 | Dis. | • | • | • | • | • |
| DWFS | Dix92 | Dis. | • | • | • | • | • |
| Str. WFS | Ros92 | Dis. | — | — | • | — | • |
| WD-WFS | BraDix95 | Dis. | — | • | • | — | • |
| WDWFS | Dix92 | Dis. | — | • | • | — | • |
| PMS | SakIno94 | Dis. | — | — | • | — | — |

Tabelle 13.1: Semantics and Their Equivalence-Transformations

$P_2$ is a stratified program and $\mathcal{STN}$ derives $a$. Concerning $P_1$, different intuitions seem possible. One can argue, that *not* $m$ should be derivable, since the only way to derive $m$ is by using the fourth clause, which means deriving $b$, which means deriving $y$ which excludes deriving $x$ or $z$. This is the way, $P_1$ is handled by the first version of $\mathcal{STN}$. The second (final) version $\mathcal{STN}$ does not derive *not* $m$. But if we apply $\mathcal{STN}$ to $P_1' \cup P_2$, then *not* $m$ is derivable. This shows that *weak Modularity* is not satisfied: we consider this to be a serious shortcoming.

Typical results of Dix are

- WFS is the weakest semantics satisfying some of these weak properties,
- WFS can be uniquely characterized if some strong properties are added.

We conclude with Table 13.1: an overview of the properties of some semantics mentioned above.

The bad properties of the PMS (failure of Relevance) stem from the fact that it was originally based on stable models. But the underlying idea of PMS is to transform disjunctive programs into non-disjunctive ones and then applying a semantics for non-disjunctive programs. By choosing semantics different from STABLE, PMS inherits other properties (see [SI94]).

## 13.2 Query-Answering Systems and Implementations

In this section we give a rough overview of what semantics have been implemented so far and where they are available. As already explained in Chapter 12 our NMR-semantics are undecidable in general. Nevertheless we think it is very important to have running systems that

1. can handle programs with free variables, and
2. are Goal-Oriented.

To ensure *completeness* (or *termination*) we need then additional requirements like *allowedness* (to prevent floundering, see Section 10.1) and no function symbols.

Although these restrictions ensure the Herbrand-universe to be finite (and thus we are really considering a propositional theory) we think that such a system has great advantages over a system that can just handle ground programs. For a language $\mathcal{L}$, the fully instantiated program can be quite large and difficult to handle effectively.

The goal-orientedness (or *Relevance* as introduced in Section 13.1) is also important — after all this was one reason of the success of SLD-Resolution. As noted above, such a goal-oriented approach is not possible for STABLE.

## LP-Semantics

Various commercial PROLOG-systems perform variants of SLDNF-Resolution. Chan's constructive negation has also been implemented as part of the master-theses [Lud91, Vor91].

Currently, a library of implemented logic programming systems and interesting test-cases for such systems is collected as a project of the artificial intelligence group at Koblenz. We refer to `http://www.uni-koblenz.de/ag-ki/LP/>`.

## Non-Disjunctive NMR-Semantics

There are many theoretical papers that deal with the problem of implementation ([BD93, KSS91, DN95, FLMS93]) but only few running systems. The problem of handling and representing ground programs given a non-ground one has also been adressed [KNS94, KNS95, ELS97].

In [BNNS93, BNNS94] the authors showed how the problem of computing stable models can be transformed to an Integer-Linear Programming Problem. This has been extended in [DM93] to disjunctive programs.

Inoue et. al. show in [IKH92] how to compute stable models by transforming programs into propositional theories and then using a model-generation theorem prover.

In Berne, Switzerland, a group around G. Jäger is building a non-monotonic reasoning system which incorporates various monotonic and non-monotonic logics. We refer to `http://lwbwww.unibe.ch:8080/LWBinfo.html`.

Extended logic programs under the well-founded semantics are considered by Pereira and his colleagues: [PAA93, ADP95, AP96]. The REVISE system, which deals with contradiction removal pro paraconsistent programs in this semantics, can be found in `<http://www.uni-koblenz.de/ag-ki/LP/>` too.

In [NS96], an implementation of WFS and STABLE with a special eye on complexity is described.

The most advanced system has been implemented by David Warren and his group in Stony Brook based on OLDT-algorithm of [TS86]. They first developed a meta-interpreter (SLG, see [CW96]) in PROLOG and then directly modified the WAM for a direct implementation of WFS (XSB). They use tabling-methods and a mixture of Top-Down and bottom-up evaluation to detect loops. Their system is complete and terminating for non-floundering DATALOG. It also works for general programs but termination is not guaranteed. This system is described in [CW93, CSW95, CW95], and is available by anonymous ftp from `ftp.cs.sunysb.edu/pub/XSB/`.

## Disjunctive NMR-Semantics

There are theoretical descriptions of implementations that have not yet been implemented: [FM95, MR95, CL95]. Also Sakama and Seki describe an approach for first-order disjunctive programs ([SS97]).

Here are some implemented systems. Inoue et. al. show in [IKH92] how to compute stable models for extended disjunctive programs in a bottom-up-fashion using a theorem prover.

The approach of Bell et. al. ([NNS91]) was used by Dix/Müller to implement versions of the stationary semantics of Przymusinski ([Prz91b]): [MD93, DM92, Mül92].

Brass/Dix have implemented both D-WFS and DSTABLE for allowed DATALOG programs ([BD95a][2]). An implementation of static semantics is described in [BDP96b][3].

Seipel has implemented in his DisLog-system various (modified versions of) semantics of Minker and his group. His system is publicly available at the URL `http://sunwww.informatik.uni-tuebingen.de:8080/dislog/dislog.tar.Z`. However we again point to the very irregular behaviour of these semantics illustrated by Example 13.1.1.

Finally, there is the DisLoP project undertaken by the Artificial Intelligence Research Group at the University of Koblenz and headed by J. Dix and U. Furbach ([DF96a, ADN97a, ADN97b]). This project aims at extending certain theorem proving concepts, such as restart model elimination [BF94] and hyper tableaux [BFN96] calculi, for disjunctive logic programming. The hyper tableaux calculus can handle positive queries with respect to positive disjunctive logic programs and seems to facilitate minimal model generation. Restart model elimination calculus does not use any contrapositives of the given clauses and thus allows for their procedural reading. Moreover, it is answer complete for positive queries [BFS95]. Thus, they are suitable for implementing an interpreter for positive progams and the DisLoP system extends this further for non-monotonic negations too.

Currently, DisLoP system can perform minimal model reasoning based on GCWA, WGCWA. Minimal model reasoning is an important problem to tackle, since any well-known semantics for negation is a conservative extension of that. DisLoP can perform minimal model reasoning in both top-down and bottom-up manners. The bottom-up approach employs the hyper tableaux calculus to generate potential minimal models and then uses a novel technique to check the minimality of the generated model without any reference to other models. This approach is described in [Nie96a, Nie96b]. The top-down approach is based on an abductive framework studied in [Ara96]. This introduces an inference rule, negation as failure to explain, which allows us to assume the negation

[2] `ftp://ftp.informatik.uni-hannover.de/software/index.html`

[3] `ftp://ftp.informatik.uni-hannover.de/software/static/static.html`

of a sentence if there are no abductive explanations for that. The DisLoP system uses a modified restart model elimination calculus to generate abductive explanations of the given sentence and employs *negation-as-failure-to-explain* inference rule for minimal model reasoning.

This system can be extended to handle non-monotonic semantics such as D-WFS, STATIC etc. In particular, an implementation of D-WFS for general disjunctive programs which works *in polynomial space* is available ([BDNP98]). Currently, an extension to first-order programs is on its way ([DS97, DS98]). Information on the DisLoP project and related publications can be obtained from the WWW page `<http://www.uni-koblenz.de/ag-ki/DLP/>`.

An important outcome of the Dagstuhl Seminar 9627 ([DLMW96]) was to construct a web page to collect and disseminate information on various logic programming systems that concentrate on non-monotonic aspects (different kinds of negation, disjunction, abduction etc.). This web page is actively maintained at the URL `<http://www.uni-koblenz.de/ag-ki/LP/>`. In addition the *Logic Programming and Nonmonotonic Reasoning*-conference 1997 ([DFN97]) contains a special track on implementations and working systems.

# A Appendix

## A.1 Predicate Logic

We assume the reader is familiar with the basic notions of predicate logic such as *models*, *formulae*, *satisfiability* $\models$ and *derivability* $\vdash$. There exist several calculi for first-order predicate logic like Hibert-style, Resolution-style, Gentzen-style or natural deduction-style calculi. One of the main theorems states the completeness of such calculi with respect to the semantics given by models:

**Theorem A.1.1 (Completeness)**
*A formula $\varphi$ follows semantically from a theory $T$ (is true in all models of $T$) if and only if $\varphi$ is derivable from $T$ by means of a particular calculus.*

$$T \vdash \varphi \quad \textit{if and only if} \quad T \models \varphi$$

This theorem tells us that we can enumerate all the theorems of a theory, but it does not provide us with a decision-method to do so. In fact, as we will explain now, such a method does not exist.

Before turning to *undecidability*, let us emphasize that in the whole paper we are dealing with predicate logic *without equality* "$\doteq$". But we can try to simulate "$\doteq$" as follows. We introduce a binary relation-symbol $eq$ and require that it satisfies the following axioms with respect to an underlying language $\mathcal{L}$:

$\forall x \; eq(x, x)$

for all function-symbols $f$ of suitable arity:
$\forall x_1 \ldots x_n, y_1 \ldots y_n$
$(eq(x_1, y_1) \ldots eq(x_n, y_n)) \rightarrow eq(f(x_1, \ldots, x_n), f(y_1, \ldots, y_n))$

for all predicate-symbols $P$ of suitable arity:
$\forall x_1 \ldots x_n y_1 \ldots y_n$
$(eq(x_1, y_1) \ldots eq(x_n, y_n)) \rightarrow (P(x_1, \ldots, x_n) \rightarrow P(y_1, \ldots, y_n))$.

This set, is denoted by $\mathrm{EQ}_{\mathcal{L}}$. It can be shown that transitivity and symmetry of $eq$ follow from these axioms. Let us consider the language of *Arithmetic* $\mathcal{L}_{Ar}$ which contains: 0 (a constant), $s$ (a unary function-symbol), $eq$ (a two-ary relation-symbol) and $\oplus$, $\otimes$ (ternary relation-symbols).

We have in mind to axiomatize the theory of natural numbers. Before we do so we introduce the following abbreviation. The formula $\exists! z\, \phi(z)$ stands for

$$\exists z \,(\, \phi(z) \,\wedge\, \forall y\, \phi(y) \rightarrow eq(y,z)).$$

**Definition A.1.2 (Arithmetic Ar$_{fin}$)**
*$Ar_{fin}$ is the finite set consisting of $EQ_{\mathcal{L}_{Ar}}$ and the following axioms:*

$$\begin{array}{ll} \forall x \forall y \exists! z & \oplus(x,y,z) \\ \forall x & \oplus(x,0,x) \\ \forall x \forall y \forall z & \oplus(x,y,z) \rightarrow \oplus(x,s(y),s(z)) \end{array}$$

$$\begin{array}{ll} \forall x \forall y \exists! z & \otimes(x,y,z) \\ \forall x & \otimes(x,0,0) \\ \forall x \forall y \forall z \forall z' & \otimes(x,y,z) \rightarrow (\otimes(x,s(y),z') \wedge \oplus(z,x,z')) \end{array}$$

The set of natural numbers $\mathcal{N} := (\mathbb{N}, 0^{\mathcal{N}}, s^{\mathcal{N}}, \oplus^{\mathcal{N}}, \otimes^{\mathcal{N}}, eq^{\mathcal{N}})$ is a model of Ar$_{fin}$. Here $0^{\mathcal{N}}$ is the "true" 0, $s^{\mathcal{N}}$ is the successor-function, $\oplus^{\mathcal{N}}$ is addition and $\otimes^{\mathcal{N}}$ is multiplication (viewed as relations), $eq^{\mathcal{N}}$ is identity. We note the following facts:

1. The set $\{\phi :\ Ar_{fin} \models \phi\}$ is recursively enumerable but not recursive.
2. The set $\{\phi :\ \mathcal{N} \models \phi\}$ is not even recursively enumerable.

We even have

**Theorem A.1.3 (Gödel)**
*No set of formulae containing $Ar_{fin}$ and having $\mathcal{N}$ as model, is recursive.*

*Every recursively enumerable set of formulae $\Phi$ that contains $Ar_{fin}$ and has $\mathcal{N}$ as a model, is incomplete, i.e. there is $\psi$ with: $\mathcal{N} \models \psi$ but $\Phi \not\models \psi$. Therefore no complete axiomatization of $\mathcal{N}$ is possible.*

Note that, although $\mathcal{N}$ formally is not a Herbrand model, it is isomorphic to such a model. In fact, the axioms immediately imply that there is, up to isomorphy, only one single Herbrand-model of Ar$_{fin}$ with respect to $\mathcal{L}_{Ar}$. Therefore to determine if a formula is true in all Herbrand-models of Ar$_{fin}$ is just as complicated as the theory of $\mathcal{N}$ itself. $\mathcal{N}$ contains, for example, famous statements (or their negation) from number theory like Goldbach-conjecture or Fermat's last theorem.

## A.2 Complexity Theory

We assume some familiarity with the classes P (problems solvable in deterministic polynomial time) and NP (problems solvable in nondeterministic polynomial time). The class co-NP is the complement of NP, i.e. a problem is in co-NP if its complement is in NP. From these sets we can build larger classes by considering problems solvable in deterministic (resp. nondeterministic) time where we allow to ask queries to an NP-oracle: i.e. whenever we come up with a subproblem that lies in NP, we just ask an oracle which immediately gives us the answer (we count this as just one step). This gives rise to the polynomial hierarchy:

**Definition A.2.1 (Polynomial Hierarchy)**
*For a complexity class $C$ we denote by* $\mathrm{P}^C$ *(resp.* $\mathrm{NP}^C$*) the class of problems solvable in deterministic polynomial (resp. nondeterministic polynomial) time using $C$-oracles. Let* $\Sigma_0 := \Pi_0 := \mathrm{P}$ *and*

$$\begin{aligned}\Sigma_{k+1} &:= \mathrm{NP}^{\Sigma_k}\\ \Pi_{k+1} &:= \text{co-NP}^{\Sigma_k}\\ \Delta_{k+1} &:= \mathrm{P}^{\Sigma_k}\end{aligned}$$

*Thus $\Sigma_1$ is NP with queries to a P-oracle, i.e.* $\Sigma_1 = \mathrm{NP}$. *Similarly we have* $\Pi_1 =$ co-NP *and* $\Delta_1 = \mathrm{P}$. *A problem is in* $\Delta_2 = \mathrm{P}^{\mathrm{NP}}$ *if it can be solved in deterministic polynomial time with subcalls to an NP-oracle. Although the index is 2, $\Delta_2$ is considered to belong to the first level of the polynomial hierarchy.*

*The second level of this hierarchy consists of $\Sigma_2$, $\Pi_2$ and $\Delta_3$. Here* $\Sigma_2 := \mathrm{NP}^{\mathrm{NP}}$*: nondeterministic polynomial time with queries to an NP-oracle.* $\Pi_2 :=$ co-$\mathrm{NP}^{\mathrm{NP}}$ *and* $\Delta_3 := \mathrm{P}^{[\mathrm{NP}^{\mathrm{NP}}]}$.

It is immediate that

$$\Sigma_k \cup \Pi_k \subseteq \Delta_{k+1} \subseteq \Sigma_{k+1} \cap \Pi_{k+1}$$

but it has not yet been proved that the inclusions are proper. That is, it is not known if the hierarchy collapses at some point or not.

The polynomial hierarchy classifies a subclass of all decidable problems, namely those that are NP-hard. A problem is called NP-hard if any other problem in NP can be polynomially reduced to it. Of particular interest are those problems in a class $\Pi_k$ or $\Sigma_k$ that are *the hardest ones*: they are called *complete*. This means that all problems in the respective class can be polynomially reduced to such a complete problem and the problem itself belongs to this class. As an example, to determine if a formula is valid is co-NP-complete. Thus, satisfiability of a propositional formula is NP-complete.

An analogue hierarchy exists (in fact it was the prototype of the polynomial hierarchy) for undecidable problems. The notation is analog to the one just introduced. Therefore one often adds a superscript $P$ to the $\Pi_k$ and $\Sigma_k$ which stands for *polynomial* (but not for an oracle) to denote the polynomial hierarchy.

To introduce the arithmetical hierarchy we consider the model $\mathcal{N}$ of the natural numbers and $\mathcal{L}_{Ar}$-formulae. We call such formulae for short *arithmetical*. We classify arithmetical formulae according to their quantifier-alternations:

**Definition A.2.2 (Arithmetical Hierarchy)**
*We call an arithmetical formula* $\Sigma^0_k$ *(resp.* $\Pi^0_k$*) if it is of the form* $\exists\forall\ldots\phi$ *(resp.* $\forall\exists\ldots\phi$*) where $\phi$ is quantifier-free and there are at most $k-1$ alternations of quantifier-blocks.*

*We call a set $M$ of natural numbers $\Sigma^0_k$-definable, if $M$ is definable by a $\Sigma^0_k$-formula. This means that there is a $\Sigma^0_k$-formula $\phi(x)$ with one free variable $x$ such that*

$$\mathcal{N} \models \phi(i) \text{ if and only if } i \in M.$$

Note that the $\Sigma_0^0$-definable sets coincide with the $\Pi_0^0$-definable ones: they are exactly the recursive sets. The recursive enumerable sets are the $\Sigma_1^0$-definable ones, the $\Pi_1^0$-definable sets are their complements. The set corresponding to the famous *Halting Problem*, i.e. the set of all Gödel numbers of those Turing-machines that stop on their own Gödel number, is $\Sigma_1^0$, so this problem is located very low in the hierarchy.

The higher a problem lies in the hierarchy, the more undecidable it is. For example a problem located at the second level, say $\Sigma_2^0$, can be thought of as *being recursively enumerable using an oracle which solves $\Sigma_1^0$-problems* (like the halting problem).

Analogously to the polynomial hierarchy we have the notions of $\Sigma_k^0$-complete and $\Pi_k^0$-complete. As an example, the halting problem is $\Sigma_1^0$-complete.

In contrast to the polynomial hierarchy, the arithmetical hierarchy is strict. We denote by $\Delta_k^0$ the intersection of $\Sigma_{k+1}^0$ and $\Pi_{k+1}^0$. We have

$$\Sigma_k^0 \cup \Pi_k^0 \subset \Delta_k^0 = \Sigma_{k+1}^0 \cap \Pi_{k+1}^0 .$$

Are there more undecidable problems, not yet captured by our hierarchy? Yes, take for example the theory of $\mathcal{N}$ considered in Section A.1. Obviously, the general problem to determine if an arbitrary formula is true or not in $\mathcal{N}$ can not be captured at a certain level, because the class of formulae in question can have unlimited alternations of quantifiers. The careful reader may have asked himself what the superscript $0$ means in $\Sigma_k^0$? It just means that we consider just first-order formulae and we do not allow our arithmetical formulae to contain second-order quantifiers.

This remark gives rise to the analytical hierarchy, denoted by $\Sigma_k^1$, $\Pi_k^1$, where we consider *second-order* arithmetical formulae. We only count the alternations of the quantifiers over sets. So any $\Sigma_k^0$-formula is in $\Sigma_0^1$.

Note that for the arithmetical hierarchy the identity $\Sigma_0^0 = \Sigma_1^0 \cap \Pi_1^0$ holds. The analogue for the analytical hierarchy does not hold. A counterexample is given by the theory of the natural numbers $\mathcal{N}$: the set of true sentences in arithmetic is in $\Sigma_1^1 \cap \Pi_1^1$ but not in $\Sigma_0^1$. This set is also called *hyperarithmetical* for obvious reasons.

For a more detailed treatment of the topics in this section we refer the reader to the standard literature: [BDG88, GJ79, Joh90] and [Pap94, Odi89] for undecidability.

## A.3 Default Logic

Reiter's default logic [Rei80] is one of the most prominent nonmonotonic logics. Default logic assumes knowledge to be represented in terms of a default theory. A default theory is a pair $(D, W)$. $W$ is a set of first order formulas representing the facts which are known to be true with certainty. $D$ is a set of defaults the form

$$\frac{A : B_1, ..., B_n}{C}$$

where $A, B_i$ and $C$ are classical formulas. We will also frequently use the alternative, less space consuming notation $A{:}B_1, \ldots, B_n/C$ for this default. The default has the intuitive reading: if $A$ is provable and, for all $i$ $(1 \leq i \leq n)$, $\neg B_i$ is not provable, then derive $C$. $A$ is called the *prerequisite*, $B_i$ a *consistency condition* or *justification*, and $C$ the *consequent* of the default. For a default $d$ we use $pre(d)$, $just(d)$, and $cons(d)$ to denote the prerequisite, the set of justifications, and the consequent of $d$, respectively. Open defaults, i.e., defaults with free variables, are usually interpreted as schemata representing all of their closed instances.[1]

Default theories induce so-called extensions which represent acceptable belief sets a reasoner may adopt based on the available information. A formula $p$ is called a skeptical consequence of $(D, W)$ iff $p$ is contained in all extensions of $(D, W)$. $p$ is called a credulous consequence of $(D, W)$ iff $p$ is contained in at least one extension of $(D, W)$.

We will first present a definition of extensions which is slightly different from (but equivalent to) Reiter's original definition. We have found that this definition is somewhat easier to digest. The original definition will be presented later.

Intuitively, $E$ is an extension of $(D, W)$ iff $E$ is a deductively closed (in the sense of classical logic) superset of $W$ satisfying the following two properties

1. all defaults that are "applicable" with respect to $E$ have been applied,
2. every formula in $E$ has a "derivation" from $W$ and applicable defaults.

To make the two requirements more precise we introduce the following notion:

**Definition A.3.1 (Default Proof)**
*Let $(D, W)$ be a default theory, $S$ a set of formulas, and $p$ a formula. A $(D, W)$-default proof for $p$ is a finite sequence $P = (d_1, \ldots, d_n)$ of defaults in $D$ such that:*

1. $W \cup \{cons(d_1), \ldots, cons(d_{i-1})\} \vdash pre(d_i)$, *for* $i \in \{1, \ldots, n\}$,
2. $W \cup \{cons(d_1), \ldots, cons(d_n)\} \vdash p$.

*$P$ is valid in $S$ iff $S$ does not contain the negation of a justification of a default in $P$.*

As usual $\vdash$ denotes classical provability. We now can state the definition of extensions formally:

**Definition A.3.2 (Extension 1)**
*Let $(D, W)$ be a default theory. $E$ is an extension of $(D, W)$ iff $E$ is a deductively closed superset of $W$ satisfying the conditions*

1. *if $A{:}B_1, \ldots, B_n/C \in D$, $A \in E$ and for all $i$ $(1 \leq i \leq n)$ $\neg B_i \notin E$, then $C$ in $E$, and*
2. *$p \in E$ implies there is a $(D, W)$-default proof for $p$ valid in $E$.*

Reiter's equivalent original definition is more compact. It defines extensions as fixed points of a certain operator.

**Definition A.3.3 (Extension 2)**
*Let $(D, W)$ be a default theory, $S$ a set of formulas. Let $\Gamma(S)$ be the smallest set such that:*

[1] Reiter treats open defaults somewhat differently and uses a more complicated method to define extensions for them.

1. *$W \subseteq \Gamma(S)$,*
2. *$Th(\Gamma(S)) = \Gamma(S)$,*
3. *if $A{:}B_1, \ldots, B_n/C \in D$, $A \in \Gamma(S)$, $\neg B_i \notin S$ $(1 \leq i \leq n)$, then $C \in \Gamma(S)$.*

*$E$ is an extension of $(D, W)$ iff $E = \Gamma(E)$, that is, if $E$ is a fixed point of $\Gamma$.*

We finally give a third, quasi-inductive characterization of extensions, also due to Reiter. This version is often used in proofs about default logic and makes the way in which formulas have to be grounded in the premises more explicit. Let $E$ be a set of formulas and define, for a given default theory $(D, W)$, a sequence of sets of formulas as follows:

$E_0 = W$, and for $i \geq 0$
$E_{i+1} = Th(E_i) \cup \{C \mid A{:}B_1, \ldots, B_n/C \in D, A \in E_i, \neg B_i \notin E\}$.

It can be shown that $E$ is an extension of $(D, W)$ iff $E = \bigcup_{i=0}^{\infty} E_i$. The appearance of $E$ in the definition of $E_{i+1}$ is what renders this alternative definition of extensions non-constructive.

Default theories may have an arbitrary number of extensions (including zero). Extensions are always consistent if $W$ is and if there are no degenerate defaults without consistency conditions. If $W$ is inconsistent then the single extension of $(D, W)$ is the set of all formulas. Extensions are maximal in the following sense: if $E$ is an extension then there is no extension $E'$ such that $E' \subset E$.

## A.4 Circumscription

Circumscription is a method of computing the closure of a theory by restricting its models to those that have minimal extensions of some of the predicates and functions. Since its first formulation by McCarthy [McC80], it has taken on several different forms, including domain circumscription [McC79] (minimizing the elements in the universe of models), and the most popular and useful version, parallel predicate circumscription [McC80, McC86, Lif85] which we present here.

Although circumscription was originally presented as a schema for adding more formulas to a theory (just as Clark's completion does), here we describe it in terms of restricting the models of the theory. This view leads to the generalization of circumscription by model preference theories, and is more useful analytically in relating circumscription to other nonmonotonic formalisms. More detailed references to circumscription can be found in Lifschitz' excellent survey article [Lif94].

Choose a language $\mathcal{L}$, and let $P$ be the set of predicate symbols that we are interested in minimizing, and $Z$ another set of predicate symbols whose interpretation we allow to vary across compared models. For example, if we wish to minimize the number of cannibals, we would let $P = \{C\}$, and $Z$ be all other predicate symbols (the importance of $Z$ will be indicated later). Suppose $A$ is a theory containing the statements $C(p_1)$, $C(p_2)$, and $C(p_3)$, but no other assertions using $C$. Then every model of $A$ will have at least the individuals referred to by $p_1$, $p_2$, and $p_3$ with property $C$. Now

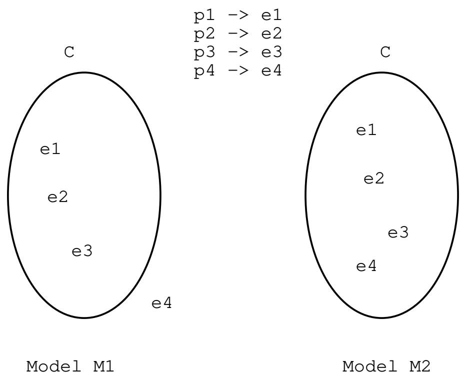



Abbildung A.1: Two Models of a Theory with the Same Valuation.

consider two models with the same valuation function from terms to individuals, as in Figure A.1. In model $M_1$, the extension of the predicate $C$ includes just the three individuals $e_1$, $e_2$, and $e_3$. In model $M_2$ there is a fourth individual, $e_4$, who is a cannibal. Circumscription would prefer $M_1$ to $M_2$, since the extension of $C$ in $M_1$ is a proper subset of its extension in $M_2$. Under appropriate assumptions (that these terms refer to different individuals), circumscription would yield the result $\neg C(p_4)$, which is not present in the original theory.

Let $A(P, Z)$ be a first-order sentence containing the symbols $P$ and $Z$. Circumscription prefers models of $A(P, Z)$ that are minimal in the predicates $P$, assuming that these models have the same interpretation for all symbols not in $P$ or $Z$. $A$ may contain predicates other than $P$ and $Z$; these are called the *fixed symbols*.

To state this more formally, let $M_1$ and $M_2$ be two models of $A(P, Z)$. $|M|$ is the universe of model $M$, and $M[\![K]\!]$ is the interpretation of the symbol $K$ in $M$. Then

**Definition A.4.1 (Minimal Models)**

$$M_1 \leq^{P;Z} M_2 \quad \textit{if and only if} \quad \begin{cases} 1. & |M_1| = |M_2|. \\ 2. & M_1[\![K]\!] = M_2[\![K]\!] \textit{ for all } K \textit{ not in } P, Z. \\ 3. & M_1[\![P_i]\!] \subseteq M_2[\![P_i]\!] \textit{ for all } P_i \in P. \end{cases}$$

$\leq^{P;Z}$ is a preorder relation (reflexive and transitive) on models, but not necessarily a partial order, since it is not antireflexive. We define the strict order $M_1 <^{(P;Z)} M_2$ as $M_1 \leq^{P;Z} M_2$ and not $M_2 \leq^{P;Z} M_1$. The preferred models of $A(P, Z)$ are those that are minimal according to the strict ordering.

## Zeitplan der Blockvorlesung

| **Abschnitte** | **Dauer** | **Gesamt** |
|---|---|---|
| Kapitel 1, 2, 3 | 1 Doppel-Std. | |
| Kapitel 4 | 1 Doppel-Std. | |
| Kapitel 5 | 1 Doppel-Std. | |
| Kapitel 6 | 2 Doppel-Std. | |
| Kapitel 7 | 2 Doppel-Std. | |
| Kapitel 8, 9 | 1 Doppel-Std. | |
| Kapitel 10 | 1 Doppel-Std. | |
| Kapitel 11, 12 | 1 Doppel-Std. | |
| Kapitel 13 | 1 Doppel-Std. | |

# Literaturverzeichnis


[AB90] Krzysztof R. Apt and Howard A. Blair. Arithmetic Classification of perfect Models of stratified Programs. *Fundamenta Informaticae*, XIII:1–17, 1990. Addendum in Vol. XIV, pages 339-344, 1991.

[AB94] Krzysztof R. Apt and Roland N. Bol. Logic Programming and Negation: A Survey. *Journal of Logic Programming*, 19-20:9–71, 1994.

[ADN97a] Chandrabose Aravindan, Jürgen Dix, and Ilkka Niemelä. Dislop: A research project on disjunctive logic programming. *AI Communications*, 10:151–165, 1997.

[ADN97b] Chandrabose Aravindan, Jürgen Dix, and Ilkka Niemelä. DisLoP: Towards a Disjunctive Logic Programming System. In J. Dix, U. Furbach, and A. Nerode, editors, *Logic Programming and Non-Monotonic Reasoning, Proceedings of the Fourth International Conference*, LNAI 1265, pages 342–353, Berlin, June 1997. Springer.

[ADP95] J. J. Alferes, Carlos Viegas Damasio, and L. M. Pereira. A logic programming system for non-monotonic reasoning. *Journal of Automated Reasoning*, 14(1):93–147, 1995.

[AP96] Jose Julio Alferes and Luiz Moniz Pereira, editors. *Reasoning with Logic Programming*, LNAI 1111, Berlin, 1996. Springer.

[Apt90] Krzysztof R. Apt. Logic programming. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science, Vol. B*, chapter 10, pages 493–574. Elsevier Science Publishers, 1990.

[Ara96] Chandrabose Aravindan. An abductive framework for negation in disjunctive logic programming. In J. J. Alferes, L. M. Pereira, and E. Orlowska, editors, *Proceedings of Joint European workshop on Logics in AI*, number 1126 in Lecture Notes in Artificial Intelligence, pages 252–267. Springer-Verlag, 1996. A related report is available on the web from <http://www.uni-koblenz.de/∼arvind/papers/>.

[Bar75] Jon Barwise. *Admissible Sets and Structures*. Springer, 1975.

[BD93] Roland N. Bol and L. Degerstedt. Tabulated resolution for well–founded semantics. In *Proc. Int. Logic Programming Symposium'93*, Cambridge, Mass., 1993. MIT Press.

[BD94] Stefan Brass and Jürgen Dix. A disjunctive semantics based on unfolding and bottom-up evaluation. In Bernd Wolfinger, editor, *Innovationen bei Rechen- und Kommunikationssystemen,* (IFIP '94-Congress, Workshop FG2: Disjunctive Logic Programming and Disjunctive Databases), pages 83–91, Berlin, 1994. Springer.

[BD95a] Stefan Brass and Jürgen Dix. A General Approach to Bottom-Up Computation of Disjunctive Semantics. In J. Dix, L. Pereira, and T. Przymusinski, editors, *Nonmonotonic Extensions of Logic Programming*, LNAI 927, pages 127–155. Springer, Berlin, 1995.

[BD95b] Stefan Brass and Jürgen Dix. Characterizations of the Stable Semantics by Partial Evaluation. In A. Nerode, W. Marek, and M. Truszczyński, editors, *Logic Programming and Non-Monotonic Reasoning, Proceedings of the Third International Conference*, LNCS 928, pages 85–98, Berlin, June 1995. Springer.

[BD95c] Stefan Brass and Jürgen Dix. Disjunctive Semantics based upon Partial and Bottom-Up Evaluation. In Leon Sterling, editor, *Proceedings of the 12th Int. Conf. on Logic Programming, Tokyo*, pages 199–213. MIT Press, June 1995.

[BD96a] Stefan Brass and Jürgen Dix. Characterizing D-WFS: Confluence and Iterated GCWA. In L.M. Pereira J.J. Alferes and E. Orlowska, editors, *Logics in Artificial Intelligence (JELIA '96)*, LNCS 1126, pages 268–283. Springer, 1996. (Extended version will appear in the *Journal of Automated Reasoning* in 1998.).

[BD96b] Gerhard Brewka and Jürgen Dix. Knowledge representation with logic programs. Technical report, Tutorial Notes of the 12th European Conference on Artificial Intelligence (ECAI '96), 1996. Also appeared as Technical Report 15/96, Dept. of CS of the University of Koblenz-Landau. Will appear as Chapter 6 in *Handbook of Philosophical Logic*, 2nd edition (1998), Volume 6, Methodologies.

[BD97] Stefan Brass and Jürgen Dix. Characterizations of the Disjunctive Stable Semantics by Partial Evaluation. *Journal of Logic Programming*, 32(3):207–228, 1997. (Extended abstract appeared in: Characterizations of the Stable Semantics by Partial Evaluation *LPNMR, Proceedings of the Third International Conference, Kentucky*, pages 85–98, 1995. LNCS 928, Springer.).

[BD98a] Stefan Brass and Jürgen Dix. Characterizations of the Disjunctive Well-founded Semantics: Confluent Calculi and Iterated GCWA. *Journal of*

*Automated Reasoning*, 20(1):143–165, 1998. (Extended abstract appeared in: Characterizing D-WFS: Confluence and Iterated GCWA. *Logics in Artificial Intelligence, JELIA '96*, pages 268–283, 1996. Springer, LNCS 1126.).

[BD98b] Stefan Brass and Jürgen Dix. Semantics of Disjunctive Logic Programs Based on Partial Evaluation. *Journal of Logic Programming*, accepted for publication, 1998. (Extended abstract appeared in: Disjunctive Semantics Based upon Partial and Bottom-Up Evaluation, *Proceedings of the 12-th International Logic Programming Conference, Tokyo*, pages 199–213, 1995. MIT Press.).

[BD98c] Gerhard Brewka and Jürgen Dix. Knowledge representation with extended logic programs. In D. Gabbay and F. Guenthner, editors, *Handbook of Philosophical Logic, 2nd Edition, Volume 6, Methodologies*, chapter 6. Reidel Publ., 1998.

[BDG88] J.L. Balcázar, I. Díaz, and J. Gabarró. *Structural Complexity I.* Springer-Verlag, Berlin, 1988.

[BDK97] Gerd Brewka, Jürgen Dix, and Kurt Konolige. *Nonmonotonic Reasoning: An Overview*. CSLI Lecture Notes 73. CSLI Publications, Stanford, CA, 1997.

[BDNP96] Stefan Brass, Jürgen Dix, Ilkka Niemelä, and Teodor. C. Przymusinski. A Comparison of Static Semantics and D-WFS. Technical Report TR 2/96, University of Koblenz, Department of Computer Science, Rheinau 1, February 1996.

[BDNP98] Stefan Brass, Jürgen Dix, Ilkka Niemelä, and Teodor. C. Przymusinski. A Comparison of the Static and the Disjunctive Well-founded Semantics and its Implementation. In A. G. Cohn, L. K. Schubert, and S. C. Shapiro, editors, *Principles of Knowledge Representation and Reasoning: Proceedings of the Sixth International Conference (KR '98)*. San Francisco, CA, Morgan Kaufmann, May 1998. appeared also as TR 17/97, University of Koblenz.

[BDP96a] Stefan Brass, Jürgen Dix, and Teodor. C. Przymusinski. Characterizations and Implementation of Static Semantics of Disjunctive Programs. Technical Report TR 4/96, University of Koblenz, Department of Computer Science, Rheinau 1, February 1996.

[BDP96b] Stefan Brass, Jürgen Dix, and Teodor. C. Przymusinski. Super Logic Programs. In L. C. Aiello, J. Doyle, and S. C. Shapiro, editors, *Principles of Knowledge Representation and Reasoning: Proceedings of the Fifth International Conference (KR '96)*, pages 529–541. San Francisco, CA, Morgan Kaufmann, 1996.

[BED92] Rachel Ben-Eliyahu and Rina Dechter. Propositional Semantics for Disjunctive Logic Programs. In K. R. Apt, editor, *LOGIC PROGRAMMING: Proceedings of the 1992 Joint International Conference and Symposium*, pages 813–827, Cambridge, Mass., November 1992. MIT Press.

[BF91a] Nicole Bidoit and Christine Froidevaux. General logical Databases and Programs: Default Logic Semantics and Stratification. *Information and Computation*, 91:15–54, 1991.

[BF91b] Nicole Bidoit and Christine Froidevaux. Negation by Default and unstratifiable logic Programs. *Theoretical Computer Science*, 78:85–112, 1991.

[BF94] P. Baumgartner and U. Furbach. Model Elimination without Contrapositives and its Application to PTTP. *Journal of Automated Reasoning*, 13:339–359, 1994. Short version in: Proceedings of CADE-12, Springer LNAI 814, 1994, pp 87–101.

[BFN96] P. Baumgartner, U. Furbach, and I. Niemelä. Hyper Tableaux. In *Proc. JELIA 96*, number 1126 in LNAI. European Workshop on Logic in AI, Springer, 1996. (Long version in: *Fachberichte Informatik*, 8–96, Universität Koblenz-Landau).

[BFS95] P. Baumgartner, U. Furbach, and F. Stolzenburg. Model Elimination, Logic Programming and Computing Answers. In *Proceedings of IJCAI '95*, 1995. (to appear, Long version in: Research Report 1/95, University of Koblenz, Germany).

[BG94] Chitta Baral and Michael Gelfond. Logic Programming and Knowlege Representation. *Journal of Logic Programming*, 19-20, 1994.

[BM86] R. Barbuti and M. Martelli. Negation as Failure. Completeness of the Query Evaluation Process for Horn Clause Programs with Recursive Definition. *Journal of Automated Reasoning*, 2:155–170, 1986.

[BNNS93] Colin Bell, Anil Nerode, Raymond T. Ng, and V. S. Subrahmanian. Implementing Stable Semantics by Linear Programming. In Luis Moniz Pereira and Anil Nerode, editors, *Logic Programming and Non-Monotonic Reasoning, Proceedings of the Second International Workshop*, pages 23–42, Cambridge, Mass., July 1993. Lisbon, MIT Press.

[BNNS94] Colin Bell, Anil Nerode, Raymond T. Ng, and V. S. Subrahmanian. Mixed Integer Programming Methods for Computing Non-Monotonic Deductive Databases. *Journal of the ACM*, 41(6):1178–1215, November 1994.

[Bon93] Piero Bonatti. Shift-based semantics: general results and applications. Technical Report CD-TR-93-59, Technical University of Vienna, Inst. für Informationssysteme, 1993.

[Bör87] Egon Börger. Unsolvable Decision Problems For Prolog Programs. In Egon Börger, editor, *Computation Theory and Logic*, LNCS 270, pages 37–47, Berlin, 1987. Springer.

[BR91] Catril Beeri and Raghu Ramakrishnan. On the power of magic. *The Journal of Logic Programming*, 10:255–299, 1991.

[Bry90] François Bry. Query evaluation in recursive databases: bottom-up and top-down reconciled. *Data & Knowledge Engineering*, 5:289–312, 1990.

[BS91] Chitta Baral and V.S. Subrahmanian. Dualities between Alternative Semantics for Logic Programming and Non-monotonic Reasoning. In Anil Nerode, Wiktor Marek, and V. S. Subrahmanian, editors, *Logic Programming and Non-Monotonic Reasoning, Proceedings of the first International Workshop*, pages 69–86, Cambridge, Mass., July 1991. Washington D.C, MIT Press.

[BS92] Chitta Baral and V.S. Subrahmanian. Stable and Extension Class Theory for Logic Programs and Default Logics. *Journal of Automated Reasoning*, 8, No. 3:345–366, 1992.

[BZF97] Stefan Brass, Ulrich Zukowski, and Burkhardt Freitag. Transformation Based Bottom-Up Computation of the Well-Founded Model. In J. Dix, L. Pereira, and T. Przymusinski, editors, *Nonmonotonic Extensions of Logic Programming*, LNAI 1216, pages 171–201. Springer, Berlin, 1997.

[CDLS95] M. Cadoli, F. M. Donini, P. Liberatore, and M. Schaerf. The size of a revised knowledge base. In *PODS '95*, pages 151–162, 1995.

[CDS96] Marco Cadoli, Francesco M. Donini, and Marco Schaerf. Is intractability of non-monotonic reasoning a real drawback? *Artificial Intelligence Journal*, 88:215–251, 1996.

[CDSS97] Marco Cadoli, Francesco M. Donini, Marco Schaerf, and Riccardo Silvestri. On compact representations of propositional circumscription. *Theoretical Computer Science*, 182:183–202, 1997. (Extended abstract appeared in: On Compact Representations of Propositional Circumscription. *STACS '95*, pages 205–216, 1995.).

[CEG92] Marco Cadoli, Thomas Eiter, and Georg Gottlob. An efficient method for eliminating varying predicates from a circumscription. *Artificial Intelligence Journal*, 54:397–410, 1992.

[Cha88] David Chan. Constructive negation based on the completed database. In *Proc. 1988 Conf. and Symp. on Logic Programming*, pages 111–125, September 1988.

[Cha93] Edward P.F. Chan. A Possible World Semantics for Disjunctive Databases. *IEEE Trans. on Knowledge and Data Engineering*, 5(2):282–292, 1993.

[CKPR73] A. Colmerauer, H. Kanoui, R. Pasero, and P. Roussel. Un système de communication homme-machine en français. Technical report, Groupe de Intelligence Artificielle Universite de Aix-Marseille II, 1973.

[CL89] L. Cavedon and J.W. Lloyd. A Completeness Theorem for SLDNF-Resolution. *Journal of Logic Programming*, 7:177–191, 1989.

[CL95] Stefania Costantini and Gaetano A. Lanzarone. Static Semantics as Program Transformation and Well-founded Computation. In J. Dix, L. Pereira, and T. Przymusinski, editors, *Nonmonotonic Extensions of Logic Programming*, LNAI 927, pages 156–180. Springer, Berlin, 1995.

[Cla78] Keith L. Clark. Negation as Failure. In H. Gallaire and J. Minker, editors, *Logic and Data-Bases*, pages 293–322. Plenum, New York, 1978.

[CS90] Jan Chomicki and V.S. Subrahmanian. Generalized Closed World Assumption is $\Pi^0_2$-Complete. *Information Processing Letters*, 34:289–291, 1990.

[CS93] Marco Cadoli and Marco Schaerf. A Survey of Complexity Results for Non-Monotonic Logics. *Journal of Logic Programming*, 17:127–160, 1993.

[CSW95] Weidong Chen, Terrance Swift, and David S. Warren. Efficient Top-Down Computation of Queries under the Well-Founded Semantics. *Journal of Logic Programming*, 24(3):219–245, 1995.

[CW89] David Chan and Mark Wallace. An Experiment with programming using pure Negation. Technical Report TR, ECRC, July 1989.

[CW93] Weidong Chen and David S. Warren. A Goal Oriented Approach to Computing The Well-founded Semantics. *Journal of Logic Programming*, 17:279–300, 1993.

[CW95] Weidong Chen and David S. Warren. Computing of Stable Models and its Integration with Logical Query Processing. *IEEE Transactions on Knowledge and Data Engineering*, 17:279–300, 1995.

[CW96] Weidong Chen and David S. Warren. Tabled Evaluation with Delaying for General Logic Programs. *Journal of the ACM*, 43(1):20–74, January 1996.

[DC90] Hendrik Decker and Lawrence Cavedon. Generalizing syntactic properties which ensure that SLDNF-Resolution is complete and flounder-free. Technical report, ECRC Munich, January 1990.

[DF96a] J. Dix and U. Furbach. The DFG-Project DisLoP on Disjunctive Logic Programming. *Computational Logic*, 2(2):89–90, 1996.

[DF96b] J. Dix and U. Furbach. Logisches Programmieren mit Negation und Disjunktion. *KI*, 3:24–31, 1996. Special Issue on Logic Programming.

[DFN97] J. Dix, U. Furbach, and A. Nerode, editors. *Logic Programming and Nonmonotonic Reasoning*, LNAI 1265, Berlin, 1997. Springer.

[DG84] W.F. Dowling and J.H. Gallier. Linear Time Algorithms for Testing the Satisfiability of Propositional Horn Formulae. *Journal of Logic Programming*, 1:267–284, 1984.

[DGM94] Jürgen Dix, Georg Gottlob, and Viktor Marek. Causal Models for Disjunctive Logic Programs. In Pascal Van Hentenryck, editor, *Proceedings of the 11th Int. Conf. on Logic Programming, S. Margherita Ligure*, pages 290–302. MIT, June 1994.

[DGM96] Jürgen Dix, Georg Gottlob, and Viktor Marek. Reducing disjunctive to non-disjunctive semantics by shift-operations. *Fundamenta Informaticae*, XXVIII(1/2):87–100, 1996.

[Dix91] Jürgen Dix. Classifying Semantics of Logic Programs. In Anil Nerode, Wiktor Marek, and V. S. Subrahmanian, editors, *Logic Programming and Non-Monotonic Reasoning, Proceedings of the first International Workshop*, pages 166–180, Cambridge, Mass., July 1991. Washington D.C, MIT Press.

[Dix92a] Jürgen Dix. A Framework for Representing and Characterizing Semantics of Logic Programs. In B. Nebel, C. Rich, and W. Swartout, editors, *Principles of Knowledge Representation and Reasoning: Proceedings of the Third International Conference (KR '92)*, pages 591–602. San Mateo, CA, Morgan Kaufmann, 1992.

[Dix92b] Jürgen Dix. Classifying Semantics of Disjunctive Logic Programs. In K. R. Apt, editor, *LOGIC PROGRAMMING: Proceedings of the 1992 Joint International Conference and Symposium*, pages 798–812, Cambridge, Mass., November 1992. MIT Press.

[Dix95a] Jürgen Dix. A Classification-Theory of Semantics of Normal Logic Programs: I. Strong Properties. *Fundamenta Informaticae*, XXII(3):227–255, 1995.

[Dix95b] Jürgen Dix. A Classification-Theory of Semantics of Normal Logic Programs: II. Weak Properties. *Fundamenta Informaticae*, XXII(3):257–288, 1995.

[Dix95c] Jürgen Dix. *Nichtmonotones Schliessen*. Universität Koblenz-Landau, Lecture Notes, 1995. Vorlesungsskriptum.

[Dix95d] Jürgen Dix. Semantics of Logic Programs: Their Intuitions and Formal Properties. An Overview. In Andre Fuhrmann and Hans Rott, editors, *Logic, Action and Information – Essays on Logic in Philosophy and Artificial Intelligence*, pages 241–327. DeGruyter, 1995.

[DLMW96] Jürgen Dix, Donald Loveland, Jack Minker, and David. S. Warren. Disjunctive Logic Programming and databases: Nonmonotonic Aspects. Technical Report Dagstuhl Seminar Report 150, IBFI GmbH, Schloß Dagstuhl, 1996.

[DM92] Jürgen Dix and Martin Müller. Abstract Properties and Computational Complexity of Semantics for Disjunctive Logic Programs. In *Proc. of the Workshop W1, Structural Complexity and Recursion-theoretic Methods in Logic Programming, following the JICSLP '92*, pages 15–28. H. Blair and W. Marek and A. Nerode and J. Remmel, November 1992. also available as Technical Report 13/93, University of Koblenz, Department of Computer Science.

[DM93] Jürgen Dix and Martin Müller. Implementing Semantics for Disjunctive Logic Programs Using Fringes and Abstract Properties. In Luis Moniz Pereira and Anil Nerode, editors, *Logic Programming and Non-Monotonic Reasoning, Proceedings of the Second International Workshop*, pages 43–59, Cambridge, Mass., July 1993. Lisbon, MIT Press.

[DM94a] Jürgen Dix and Martin Müller. An Axiomatic Framework for Representing and Characterizing Semantics of Disjunctive Logic Programs. In Pascal Van Hentenryck, editor, *Proceedings of the 11th Int. Conf. on Logic Programming, S. Margherita Ligure*, pages 303–322. MIT, June 1994.

[DM94b] Jürgen Dix and Martin Müller. Partial Evaluation and Relevance for Approximations of the Stable Semantics. In Z.W. Ras and M. Zemankova, editors, *Proceedings of the 8th Int. Symp. on Methodologies for Intelligent Systems, Charlotte, NC, 1994*, LNAI 869, pages 511–520, Berlin, 1994. Springer.

[DM94c] Jürgen Dix and Martin Müller. The Stable Semantics and its Variants: A Comparison of Recent Approaches. In L. Dreschler-Fischer and B. Nebel, editors, *Proceedings of the 18th German Annual Conference on Artificial Intelligence (KI '94), Saarbrücken, Germany*, LNAI 861, pages 82–93, Berlin, 1994. Springer.

[DN95] Lars Degerstedt and Ulf Nilsson. Magic Computation of Well-founded Semantics. In J. Dix, L. Pereira, and T. Przymusinski, editors, *Nonmonotonic Extensions of Logic Programming*, LNAI 927, pages 181–204. Springer, Berlin, 1995.

[Dra94] Wlodzimierz Drabent. What is failure? A constructive approach to negation. *Acta Informatica*, 32(1):27–29, 1994.

[DS97] Jürgen Dix and Frieder Stolzenburg. Computation of Non-Ground Disjunctive Well-Founded Semantics with Constraint Logic Programming (preliminary report). In J. Dix, L. Pereira, and T. Przymusinski, editors, *Nonmonotonic Extensions of Logic Programming*, LNAI 1216, pages 202–226. Springer, Berlin, 1997.

[DS98] Jürgen Dix and Frieder Stolzenburg. A Framework to incorporate Nonmonotonic Reasoning into Constraint Logic Programming. *Journal of Logic Programming*, 1998. Special Issue on *Constraint Logic Programming*, Guest Editors: Kim Marriott and Peter Stuckey, to appear in 1998. Extended abstract appeared in Computation of Non-Ground Disjunctive Well-Founded Semantics with Constraint Logic Programming (Preliminary Report). *Nonmonotonic Extensions of Logic Programming*, pages 202–226, 1997. LNAI 1216, Springer.).

[Dun92] P. M. Dung. On the relations between stable and wellfounded semantics of logic programs. *Theoretical Computer Science*, 105:7–25, 1992.

[EG93a] Thomas Eiter and Georg Gottlob. Complexity aspects of various semantics for disjunctive databases. In *Proc. of the Twelfth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS'93)*, pages 158–167, 1993.

[EG93b] Thomas Eiter and Georg Gottlob. Propositional Circumscription and Extended Closed World Reasoning are $\Pi_2^P$-complete. *Theoretical Computer Science*, 144(2):231–245, Addendum: vol. 118, p. 315, 1993, 1993.

[EGM93] Thomas Eiter, Georg Gottlob, and Heikki Mannila. Expressive Power and Complexity of Disjunctive DATALOG. In *Proceedings of Workshop on Logic Programming with Incomplete Information, Vancouver Oct. 1993, following ILPS' 93*, pages 59–79, 1993.

[ELS97] T. Eiter, J. Lu, and V. S. Subrahmanian. Computing Non-Ground Representations of Stable Models. In J. Dix, U. Furbach, and A. Nerode, editors, *Logic Programming and Non-Monotonic Reasoning, Proceedings of the Fourth International Conference*, LNAI 1265, pages 198–217, Berlin, July 1997. Springer.

[Fag93] F. Fages. Consistency of Clark's completion and existence of stable models. *Methods of Logic in Computer Science*, 2, 1993.

[Fit85] Melvin C. Fitting. A Kripke-Kleene Semantics of logic Programs. *Journal of Logic Programming*, 4:295–312, 1985.

[FLMS93] J. A. Fernández, J. Lobo, J. Minker, and V.S. Subrahmanian. Disjunctive LP + Integrity Constraints = Stable Model Semantics. *Annals of Mathematics and Artificial Intelligence*, 8(3-4), 1993.

[FM95] J. A. Fernández and J. Minker. Bottom-Up Computation of Perfect Models for Disjunctive Theories. *Journal of Logic Programming*, 25(1):33–51, 1995.

[GJ79] M.R. Garey and D.S. Johnson. *Computers and Intractability*. W.H. Freeman and Company, San Francisco, 1979.

[GL88] Michael Gelfond and Vladimir Lifschitz. The Stable Model Semantics for Logic Programming. In R. Kowalski and K. Bowen, editors, *5th Conference on Logic Programming*, pages 1070–1080. MIT Press, 1988.

[GPSK95] Goran Gogic, Christos Papadimitriou, Bart Selman, and Henry Kautz. The Comparative Linguistics of Knowledge Representation. In *Proceedings of the 14th International Joint Conference on Artificial Intelligence*, pages 862–869, Montreal, Canada, August 1995. Morgan Kaufmann Publishers.

[Gur88] Y. Gurevich. Logic and the Challenge of Computer Science. In E. Börger, editor, *Trends in Theoretical Computer Science*, chapter 1. Computer Science Press, 1988.

[HTT90] Jeff Horty, Richmond Thomason, and D. S. Touretzky. A skeptical Theory of Inheritance in Nonmonotonic Semantic Networks. *Artificial Intelligence*, 42:311–348, 1990.

[IKH92] Katsumi Inoue, M. Koshimura, and R. Hasegawa. Embedding negation-as-failure into a model generation theorem prover. In Deepak Kapur, editor, *Automated Deduction — CADE-11*, number 607 in LNAI, Berlin, 1992. Springer.

[Imi91] T. Imielinski. Incomplete Deductive Databases. *Annals of Mathematics and Artificial Intelligence*, 3:259–294, 1991.

[JdL92] Catholijn Jonker and G.R.Renardel de Lavalette. A tractable algorithm for the wellfounded model. Technical Report Logic Group Preprint Series No. 746, Utrecht University, Dept. of CS, 1992.

[Joh90] D.S. Johnson. A catalog of complexity classes. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume A. Algorithms and Complexity, pages 67–161. Elsevier, 1990.

[KNS94] Vadim Kagan, Anil Nerode, and V. S. Subrahmanian. Computing Definite Logic Programs by Partial Instantiation. *Annals of Pure and Applied Logic*, 67:161–182, 1994.

[KNS95] Vadim Kagan, Anil Nerode, and V. S. Subrahmanian. Computing Minimal Models by Partial Instantiation. *Theoretical Computer Science*, 155:157–177, 1995.

[Kow74] R.A. Kowalski. Predicate logic as a programming language. In *Proceeedings IFIP' 74*, pages 569–574. North Holland Publishing Company, 1974.

[KSS91] David B. Kemp, Peter J. Stuckey, and Divesh Srivastava. Magic Sets and Bottom-Up Evaluation of Well-Founded Models. In Vijay Saraswat and Kazunori Ueda, editors, *Proceedings of the 1991 Int. Symposium on Logic Programming*, pages 337–351. MIT, June 1991.

[Kun87] Kenneth Kunen. Negation in Logic Programming. *Journal of Logic Programming*, 4:289–308, 1987.

[Kun89] Kenneth Kunen. Signed Data Dependencies. *Journal of Logic Programming*, 7:231–245, 1989.

[Lif85] Vladimir Lifschitz. Computing Circumscription. In *Proceedings of the International Joint Conference on Artificial Intelligence, Los Angeles, California*, pages 121–127, 1985.

[Lif94] V. Lifschitz. *Circumscription*, pages 297–353. Clarendon, Oxford, 1994.

[Llo87] John W. Lloyd. *Foundations of Logic Programming*. Springer, Berlin, 1987. 2nd edition.

[LMR92] Jorge Lobo, Jack Minker, and Arcot Rajasekar. *Foundations of Disjunctive Logic Programming*. MIT-Press, 1992.

[Lud91] Bertram Ludäscher. CNF-Prolog: A Meta-Interpreter for Chan's Constructive Negation, Implementation. Technical report, Master Thesis, Karlsruhe University (in german), 1991.

[McC79] John McCarthy. First order theories of individual concepts and propositions. In B. Meltzer and D. Michie, editors, *Machine Intelligence 9*, pages 120–147. Edinburgh University Press, Edinburgh, 1979.

[McC80] John McCarthy. Circumscription: A Form of Nonmonotonic Reasoning. *Artificial Intelligence*, 13:27–39, 1980.

[McC86] John McCarthy. Applications of Circumscription to formalizing Common-Sense Reasoning. *Artificial Intelligence*, 28:89–116, 1986.

[MD93] Martin Müller and Jürgen Dix. Implementing Semantics for Disjunctive Logic Programs Using Fringes and Abstract Properties. In Luis Moniz Pereira and Anil Nerode, editors, *Logic Programming and Non-Monotonic Reasoning, Proceedings of the Second International Workshop*, pages 43–59, Cambridge, Mass., July 1993. Lisbon, MIT Press.

[Min82] Jack Minker. On indefinite databases and the closed world assumption. In *Proceedings of the 6th Conference on Automated Deduction, New York*, pages 292–308, Berlin, 1982. Springer.

[Min88] Jack Minker. *Foundations of Deductive Databases*. Morgan Kaufmann, 95 First Street, Los Altos, CA 94022, 1st edition, 1988.

[Min93] Jack Minker. An Overview of Nonmonotonic Reasoning and Logic Programming. *Journal of Logic Programming, Special Issue*, 17(2/3/4):95–126, 1993.

[MMP88] Paolo Mancarella, Simone Martini, and Dino Pedreschi. Complete logic Programs with domain-closure Axiom. *Journal of Logic Programming*, 5:263–276, 1988.

[Mos74] Y. N. Moschovakis. *Elementary Induction on Abstract Structures*. North-Holland, 1974.

[MR95] Jack Minker and Carolina Ruiz. Computing stable and partial stable models of extended disjunctive logic programs. In J. Dix, L. Pereira, and T. Przymusinski, editors, *Nonmonotonic Extensions of Logic Programming*, LNAI 927, pages 205–229. Springer, Berlin, 1995.

[MRT92] Wiktor Marek, Arcot Rajasekar, and Mirek Truszczyński. Complexity of Computing with Extended Propositional Logic Programs. In *Proc. of the Workshop W1, Structural Complexity and Recursion-theoretic Methods in Logic Programming, following the IJCSLP '92*, pages 93–102. H. Blair and W. Marek and A. Nerode and J. Remmel, November 1992.

[Mül92] Martin Müller. Examples and Run-Time Data from KORF, 1992.

[Nie96a] Ilkka Niemelä. Implementing circumscription using a tableau method. In W. Wahlster, editor, *Proceedings of the European Conference on Artificial Intelligence*, pages 80–84, Budapest, Hungary, August 1996. John Wiley.

[Nie96b] Ilkka Niemelä. A tableau calculus for minimal model reasoning. In P. Miglioli, U. Moscato, D. Mundici, and M. Ornaghi, editors, *Proceedings of the Fifth Workshop on Theorem Proving with Analytic Tableaux and Related Methods*, pages 278–294, Terrasini, Italy, May 1996. LNAI 1071, Springer-Verlag.

[NNS91] Anil Nerode, Raymond T. Ng, and V.S. Subrahmanian. Computing Circumscriptive Deductive Databases. CS-TR 91-66, Computer Science Dept., Univ. Maryland, University of Maryland, College Park, Maryland, 20742, USA, December 1991.

[NS96] Ilkka Niemelä and Patrik Simons. Efficient Implementation of the Well-founded and Stable Model Semantics. In M. Maher, editor, *Proceedings of the Joint International Conference and Symposium on Logic Programming*, pages 289–303, Bonn, Germany, September 1996. The MIT Press.

[Odi89] P. Odifreddi. *Classical Recursion Theory*. North-Holland, 1989.

[PAA93] L. M. Pereira, J. N. Aparício, and J. J. Alferes. Non-Monotonic Reasoning with Logic Programming. *Journal of Logic Programming*, 17:227–264, 1993.

[Pap94] C.H. Papadimitriou. *Computational Complexity*. Addison-Wesley, 1994.

[Prz91a] Teodor Przymusinski. Semantics of disjunctive logic programs and deductive databases. In Claude Delobel, Michael Kifer, and Yoshifumi Masunaga, editors, *Deductive and Object-Oriented Databases, 2nd Int. Conf. (DOOD'91)*, number 566 in LNCS, pages 85–107, Berlin, 1991. Springer.

[Prz91b] Teodor Przymusinski. Stationary Semantics for Normal and Disjunctive Logic Programs. In C. Delobel, M. Kifer, and Y. Masunaga, editors, *DOOD '91, Proceedings of the 2nd International Conference*, Berlin, December 1991. Muenchen, Springer. LNCS 566.

[Prz95] Teodor Przymusinski. Static Semantics For Normal and Disjunctive Logic Programs. *Annals of Mathematics and Artificial Intelligence*, 14:323–357, 1995.

[Rei78] Raymond Reiter. On closed world data bases. In Hervé Gallaire and Jack Minker, editors, *Logic and Data Bases*, pages 55–76, New York, 1978. Plenum.

[Rei80] Raymond Reiter. A Logic for Default-Reasoning. *Artificial Intelligence*, 13:81–132, 1980.

[RLM89] Arcot Rajasekar, Jorge Lobo, and Jack Minker. Weak Generalized Closed World Assumption. *Journal of Automated Reasoning*, 5:293–307, 1989.

[RN95] Stuart Russel and Peter Norvig. *Artificial Intelligence — A Modern Approach*. Prentice Hall, New Jersey 07458, 1995.

[Ros89] Kenneth A. Ross. The well-founded semantics for disjunctive logic programs. In *Proceedings of the first International Conference on Deductive and Object Oriented Databases, Kyoto, Japan*, pages 1–22, 1989.

[RT88] Kenneth A. Ross and Rodney A. Topor. Inferring negative Information from disjunctive Databases. *Journal of Automated Reasoning*, 4:397–424, 1988.

[Sac93] Domenico Sacca. The Expressive Power of Stable Models For DATALOG Queries with Negation. In *Proceedings of Workshop on Logic Programming with Incomplete Information, Vancouver Oct. 1993, following ILPS' 93*, pages 150–162, 1993.

[Sak89] Chiaki Sakama. Possible Model Semantics for Disjunctive Databases. In Won Kim, Jean-Marie Nicolas, and Shojiro Nishio, editors, *Deductive and Object-Oriented Databases, Proceedings of the First International Conference (DOOD89)*, pages 369–383, Kyoto, Japan, 1989. North-Holland Publ.Co.

[Sch90] John S. Schlipf. The Expressive Powers of the Logic Programming Semantics. In *Proceedings of the Ninth ACM Symposium on Principles of Databases*, pages 196–204, 1990.

[Sch92] John S. Schlipf. A Survey of Complexity and Undecidability Results in Logic Programming. In H. Blair, W. Marek, A. Nerode, and J. Remmel, editors, *Proceedings of the Workshop on Complexity and Recursion-theoretic Methods in Logic Programming, following the JICSLP'92*. informal, 1992.

[She88] John C. Shepherdson. Negation in Logic Programming. In Jack Minker, editor, *Foundations of Deductive Databases*, chapter 1, pages 19–88. Morgan Kaufmann, 1988.

[She91] John C. Shepherdson. Unsolvable Problems for SLDNF-Resolution. *Journal of Logic Programming*, 10:19–22, 1991.

[She92] John C. Shepherdson. SLDNF-Resolution with Equality. *Journal of Automated Reasoning*, 8, No. 2:297–306, 1992.

[SI93] Chiaki Sakama and Katsumi Inoue. Negation in Disjunctive Logic Programs. In D. Warren and Peter Szeredi, editors, *Proceedings of the 10th Int. Conf. on Logic Programming, Budapest*, pages 703–719, Cambridge, Mass., July 1993. MIT Press.

[SI94] Ch. Sakama and K. Inoue. An Alternative Approach to the Semantics of Disjunctive Logic Programs and Deductive Databases. *Journal of Automated Reasoning*, 13:145–172, 1994.

[SS94] Chiaki Sakama and Hirohisa Seki. Partial Deduction of Disjunctive Logic Programs: A Declarative Approach. In *Logic Program Synthesis and Transformation – Meta Programming in Logic*, LNCS 883, pages 170–182, Berlin, 1994. Springer.

[SS97] Chiaki Sakama and Hirohisa Seki. Partial Deduction in Disjunctive Logic Programming. *Journal of Logic Programming*, 32(3):229–245, 1997.

[Stä94] Robert F. Stärk. Input/output dependencies of normal logic programs. *Journal of Logic and Computation*, 4(3):249–262, 1994.

[Tar55] A. Tarski. A lattice-theoretical fixpoint theorem and its applications. *Pacific Journal of Mathematics*, 5:285–309, 1955.

[THT88] David S. Touretzky, Jeff Horty, and Richmond Thomason. A Clash of Intuitions: The current State of Nonmonotonic Multiple IHS. In *Proceedings IJCAJ*, 1988.

[Tou86] D. S. Touretzky. *The Mathematics of Inheritance*. Research Notes in Artificial Intelligence. Pitman, London, 1986.

[TS86] H. Tamaki and T. Sato. OLD Resolution with Tabulation. In *Proceedings of the Third International Conference on Logic Programming, London*, LNAI, pages 84–98, Berlin, June 1986. Springer.

[Ull89a] Jeffrey D. Ullman. Bottom-up Beats Top-down for Datalog. In *Proc. of the Eight ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, Philadelphia, Pennsylvania*, pages 140–149. ACM Press, March 1989.

[Ull89b] Jeffrey D. Ullman. *Principles of Database and Knowledge-Base Systems, Vol. 2*. Computer Science Press, Rockville, 1989.

[vEK76] M.H. van Emden and R.A. Kowalski. The semantics of predicate logic as a programming language. *JACM*, 23:733–742, 1976.

[VGRS88] Allen Van Gelder, Kenneth A. Ross, and J. S. Schlipf. Unfounded Sets and well-founded Semantics for general logic Programs. In *Proceedings 7th Symposion on Principles of Database Systems*, pages 221–230, 1988.

[Vor91] Martin Vorbeck. CNF-Prolog: A Meta-Interpreter for Chan's Constructive Negation, Theory. Technical report, Master Thesis, Karlsruhe University (in german), 1991.

[Wit91a] Cees Witteveen. Partial Semantics for Truth Maintenance. In J. van Eijck, editor, *Logics in AI*, LNAI 478, Berlin, 1991. Springer.

[Wit91b] Cees Witteveen. Skeptical Reason Maintenance is Tractable. In J. Allen, R. Fikes, and B. Sandewall, editors, *Proceedings of the second Conference on Principles of Knowledge Representation and Reasoning, Cambridge, Massachusetts*, pages 570–581. Morgan Kaufmann, 1991.