Zusammenfassung Notizen zu Deskriptive Programmierung Prof. Dr. R. Manthey, Dr. A. Behrend

SS 2008, Autorin: Diana v. Gallera

V 1.0β ohne Gewährleistung. Anmerkungen bitte an: vongalle@cs.uni-bonn.de

Kap 1:

Funktionen gehören zu Termen, Relationen zu Formeln
Terme dürfen in Formeln vorkommen, umgekehrt aber nicht
funktionale Programmierung: keine Relationen, keine Quantoren, d.h. keine Formeln sondern nur Terme und funktionale Logik
logische Programmierung: keine Funktionen, keine Quantoren, keine Terme sondern nur Formeln und relationale Logik

Kap 2.1:

Haskell: *Fallunterscheidung und Reduktion*

Aufbau einer Funktionsdefinition: Name, Parameter und auszuwertender Ausdruck(ggf. Fallunterscheidung)

Syntax Haskell:

Funktionsabstraktion: auf der linken Seite einer definierenden Gleichung dürfen keine auswertbaren Ausdrücke (z.B. 2+3) stehen, nur Variablen und Konstanten, jede Variable max. 1 mal (formale Parameter)
auf der rechten Seite dürfen ausschließlich auf der linken Seite eingeführte Variablen vorkommen (es sei denn, man verwendet lokale Variablen mittels let .. in .. oder where .. , sonst keine Einschränkungen

Funktionsapplikation: bei der Funktionsapplikation dürfen keine Variablen verwendet werden (anders in Prolog), aber weitere Funktionsapplikationen. Zusammengesetzte Ausdrücke müssen allerdings geklammert werden (z.B. f (2+3) (4+5) bei einer Funktion mit zwei Parametern).
Wächter werden immer von oben nach unten durchlaufen bis eine Bedingung erfüllt ist, alles was darunter folgt, wird ignoriert (deswegen otherwise immer als letzten Wächter verwenden)

Auswahlkriterien (i.d.Reihenfolge):
pattern matching (Suche nach passender Funktionsdefinition von oben nach unten) und danach Auswertung der Wächterbedingung

pattern matching - Regeln:
Konstanten matchen sich selbst und jede Variable
komplexe Ausdrücke matchen jede Variable und diejenige Konstante, die ihren Funktionswert bezeichnet (z.B. fib(3) <-> 3)

Variablen beginnen in Haskell mit Kleinbuchstaben
Operatoren:

Infix-Operatoren müssen mit einfachen Anführungszeichen ´ umschlossen sein, sonst kann Haskell sie nicht von Parametern unterscheiden
Wächter sind boolesche Ausdrücke mit Wahrheitswerten True und False (groß geschrieben)
boolesche Operatoren: && , || , not
Vergleichsoperatoren: <=, >=, <, >, ==, /= (auch auf anderen Typen als Zahlen definiert)

Prolog: *Logische Programmierung*
Ein Programm ist eine *Relationsdefinition* und wird mittels *Resolution* abgearbeitet, als Input wird eine Anfrage als Ausdruck erwartet und als Output die Antwort, ob sie wahr gemacht werden konnte und eine passende *Variablensubstitution* (allgemeinster Unifikator) zurückgegeben, sofern die Anfrage wahr gemacht werden konnte und sich Variablen in ihr befanden.
neue Unteranfragen erfolgen von oben nach unten und von links nach rechts
Alternativen werden mittels *Backtracking*/Reevaluierung gesucht und dem Operator ; angestoßen
-> in Haskell können Relationen mittels boolescher Funktionen dargestellt werden

Bezeichnungen:
prädikat(konstante1,konstante2). % Fakt
prädikat(Var1,Var2) : - prädikat2(Var1,Var3) , prädikat2(Var2,Var3). % Regel , Implikation :-
?- prädikat(Var1, konstante2). % Anfrage
Und alles insgesamt sind Klauseln, sowie jedes einzelne Prädikat ein Literal.

Objekte in Prolog: Konstanten, Strukturen, Operatoren, Variablen, Terme
Konstanten: Zahlen, Atome (keine Zahlen, Konstanten und Zeichenfolgen in ` ', Folgen nur aus Sonderzeichen)

In Prolog gibt es keine Typisierung

Strukturen sind Objekte, die aus mehreren anderen Objekten zusammengesetzt sind
Variablen: mit Großbuchstaben beginnend oder _ (anonyme Variable)
Gültigkeitsbereich einer Variablen ist lokal für die Klausel
Terme: Konstanten, Variablen und Strukturen, Grundterme enthalten keine Variablen

In Prolog sind zwei Terme genau dann gleich, wenn sie strukturell gleich sind, d.h. es wird für den Vergleich nichts ausgewertet. Somit ist die Antwort auf die Anfrage ?- 2+3 = 5 No. Dennoch kann eine Auswertung mittels dem is Operator erzwungen werden, wobei der auszuwertende Ausdruck unbedingt rechts davon stehen muss.
Unifizierbarkeit = Gleichheit
Sind T1 und T2 Terme, dann ist T1 = T2 (= (T1,T2)) konstruktiv nachweisbar, wenn T1 und T2 unifizierbar sind, d.h. wenn die Terme durch Ersetzung der Variablen ausgeglichen werden können: Substitution.

Ein *Unifikator* ist eine Substitution, die zwei Terme gleichmacht
allgemeinster Unifikator: möglichst viele Variablen sollen unverändert/ungebunden bleiben, das was

zuerst kommt wird immer ersetzt durch das, was zuletzt kommt: T1/T2
Korrektheit: kann eine Aussage in mehreren Schritten unter Verwendung des Resolutionsprinzips auf die leere Aussage zurückgeführt werden, so ist sie beweisbar.
Unentscheidbarkeit: Prolog antwortet mit yes , mit no oder terminiert nicht.
Listen: Listen dürfen in Prolog auch inhomogen sein, d.h. verschiedene Typen beinhalten
Listen können mit dem Konstruktor . als Binärbäume dargestellt werden:
z.B. [a,b,1,2] = .(a,.(b,.(1,.(2,[])))) dabei ist . die jeweilige Wurzel und der nächste Eintrag der rechte Sohn

Haskell: *Listen*
Listenkonstruktor: **:**
Konkatenationsoperator: ++
Zugriff auf einzelne Elemente (infix): **!!** z.B. ([1,2,3] !! 2) holt das dritte Element
Haskell erlaubt nur homogene Listen, d.h. alle Elemente müssen den selben Typ besitzen

Es ist nur eine Funktionsdefinition pro Name erlaubt, das gilt auch bei verschiedener Stelligkeit

List comprehension: eine list-comprehension besteht aus:
einem Muster, einem obligatorischen Generator je Variable und einem optionalen Filter :
[muster | generatoren, filter] z.B. [x | x <- [1,2 ..] , x < 50] = [1,2 .. 49]
Muster: Ausdruck mit min. einer Variablen, deren Werte durch den Generator erzeugt werden
Generator: Ausdruck der Form <Variable> <- <Liste> wobei <- für element steht
Filter: boolescher Ausdruck, der die Liste auf die Bindungen der Variable beschränkt, bei denen er den Wert wahr hat

arithmetische Sequenzen:
[von .. bis] : erzeugt eine Liste mit allen Elementen die zwischen von und bis liegen
[von ..] : erzeugt eine unendliche Liste die mit von beginnt
[von, nächstes .. bis] : erzeugt eine Liste der Elemente zwischen von und bis mit der Schrittweite nächstes - von
alternative Definition von n! ohne Rekursion: fac n = product [1 .. n]
parameterlose Funktionen sind als Namengebung für Listen praktisch, daraus können beispielsweise Folgen erzeugt werden (z.B. naturals = [1 ..]) , allerdings ist Vorsicht geboten: nicht terminierende Ausgabe, besser arbeiten mit endl. Teillisten (z.B. durch anwenden von !! oder first' auf unendliche Listen, wegen Haskells lazy evaluation Vorgehen ist das möglich)

In Haskell gibt es keinen Datentyp für Mengen, stattdessen für Listen (nicht duplikatfrei)

Prolog: *Listen*
intern sind Listen als Binärbäume implementiert, deren Blätter die Elemente der Liste sind. Der Abschluss erfolgt mit der leeren Liste [].
Konstruktor : . z.B. .(4,[1,2,3])
Zugriff auf Listenelemente erfolgt mittels pattern matching und Zerlegung

vergleichsweise Anwendung von list-comprehension in Prolog erfolgt mittels dem Prädikat
findall(Template,Goal,List) wobei Template einerseits die Variablen enthält, deren Inhalte man aus Goal filtern möchte, andererseits eine Formatvorgabe für die Zielliste List bildet. Goal ist ein Prädikat und/oder eine Bedingung (sozusagen Generator und Filter) und List beschreibt die Ausgabeliste. Bsp: findall(Y, (member(X,[1,2,3,4]),0 is X mod 2, Y is X*X),L)

die Beweisstrategie und Verwendung nicht-logischer Aspekte (Bsp. durch is -Operator) können eine relationale Definition verhindern -> deswegen explizite Angabe von Aufrufbeschränkungen

generate & test:
raten einer möglichen Lösung und testen auf Korrektheit:
loesung(X) :- moeglicheLoesung(X),korrekteLoesung(X).
allerdings nur geeignet bei kleinem, endlichem Suchraum oder wenn man sonst keine Lösung weiß, z.B. für Sudoku.
absolut ineffizient bei geringfügig größerem Suchraum, nicht geeignet für Sortierungen!

Haskell: *Typen*
Strings: .... steht auch für : ['.' , '.' , '.' , ...] (also eine Liste von Buchstaben: Stringmanipulation mittels gewöhnlicher Listenmanipulation möglich)
Typzuordnung geschieht mit dem Operator: **::**
Typbezeichner beginnen generell mit Großbuchstaben
Int: -2 ^31 .. 0 .. (2^31) -1 beschränkt, weswegen Überlauf möglich!
Integer: ... -1 0 1 ... quasi unendlich
strukturierte Typen:
Tupeltypen: (1,'d',True) ::(Integer, Char, Bool)
Listentypen: [1,2,3] :: [Integer], [['a','b','c'],['d'],['e','f']] :: [[Char]] (Liste von Listen von Char)
Schachtelung von Tupel- und Listentypen ist erlaubt
Datentypdeklaration:
data Typkonstruktor = Datenkons1 | Datenkons2 | Datenkons3 ..
z.B. data Gender = Male | Female

Datenkonstruktoren mit Parametern: data Point = Pt(Integer,Integer) <- Parametertypen
Dabei ist die Verwendung des gleichen Bezeichners für Typ- und Datenkonstruktoren zulässig!
Funktionstypen: **<Parametertypen> -> <Resultattyp>**
mehrstellige Funktionen -> Umwandlung in mehrstufige Funktionen mittels *Curryfizierung*, da es eigentlich nur einstellige Funktionen in Haskell gibt
Vorteil Curryfizierung: so können Funktionen überladen werden, mehrere Stelligkeiten sind möglich: partielle Funktionale
Polymorphie: Verwendung von Typvariablen: parametrisierte Typen von Funktionen
z.B. length :: [a] -> Integer nimmt Listen beliebigen Typs an, oder last :: [a] -> a
In Haskell hat jeder Ausdruck genau einen Datentyp, die Sprache ist streng typisiert.
Funktionen dürfen ebenfalls Funktionen als Parameter nehmen und Funktionen als Ausgabewerte haben: *Funktionen höherer Ordnung*.

Operator zum komponieren von Funktionen: . Bsp.: (f . g) (x) = (f(g(x)))

Prolog:

besonders gut für musterorientierte Manipulation von Strukturen geeignet (wg. Pattern-Matching)
z.B. syntaktisches Differnzieren/unbestimmtes Integrieren

Fakten haben gegenüber Listen den Vorteil, dass sie während der Ausführung des Programms geändert werden können: assert(X) fügt das Prädikat X der Datenbank hinzu, retract(X) entfernt es wieder

Akkumulatoren:
sind Variablen, die das bisherige Zwischenergebnis in einem Suchlauf speichert und an die nachfolgende Rekursionsstufe übergeben wird. Mit ihnen wird Endrekursion möglich, da danach keine Nachbearbeitung mehr notwenig ist. Wenn die Liste abgearbeitet wurde, muss das Zwischenergebnis in das Endergebnis abgelegt werden, weswegen
prädikat([],Zwischen,Zwischen). nicht fehlen darf.

Negation: \+
Variablen werden durch Negation nicht gebunden.

Cut: **!**
Mit ! kann hoher Verwaltungsaufwand unterbunden werden, durch Abschneiden bestimmter SLD-Teilbäume (dadurch Verhindern von Alternativen und Backtracking).
Dadurch ergibt sich eine kürzere Laufzeit und weniger Speicherbedarf.
Vorsicht: korrekte Antworten können evtl. nicht mehr gefunden werden, bzw.
weitere Lösungspfade können ausgeschlossen werden, wenn bisherige Lösung die einzig richtige/notwenige ist (roter Cut).
Mittels dem Cutoperator ist die Definition einer totalen Relation mit catch-all Regel möglich.
Auch kann man ihn als bedingte Anweisung lesen:
p :- q , !, s.
p: - r . ist äquivalent zu if q then s else r , alternative Notation: p:- q ->s ; r.

Durch Negation kann Cut vermieden werden und umgekehrt.
Negation durch Cut:
not(X):- call(X),!,fail.
not(X).
Unterscheidung zweier verschiedener Cuts:
grüner Cut: schneidet Misserfolge und unendliche Pfade ab.
roter Cut: schneidet logisch korrekte Antworten ab (kann aber erwünscht sein).

Haskell:

generate & test ist auch hier relativ ineffizient. Wenn schon am Anfang der Abarbeitung klar ist, dass die Lösung nicht richtig sein kann, kann schon direkt nach der Feststellung abgebrochen werden: Implementierung durch Verzahnung: generate 1. Schritt test generate 2. Schritt test ..

Noch besser ist forward checking : im Vorhinein gucken, welche nächsten Schritte überhaupt noch erlaubt sind (dazu z.B. Verwendung der vordef. Funktion filter Bedingung [Liste] = [Ergebnisliste])

Prolog: *Generierung aller Lösungen*
Beispielfunktionen:
findall(?Template,+Goal,?List).
bagof(?Template,+Goal,?List). <- wie findall, allerdings werden die nicht in Template vorkommenden freien Variablen nacheinander gebunden und die dazu gehörigen Teillisten ausgegeben, es sei denn, diese Variablen sind mit Var^Goal markiert
setof(?Template,+Goal,?List). <- wie bagof, allerdings ohne Duplikate

weitere Operatoren
X =:= Y True, wenn X und Y den gleichen Wert haben
X = \ = Y True, wenn X und Y nicht den gleichen Wert haben
is : nur partiell, da nicht alle Variablen ungebunden sein dürfen, Variablen auf der rechten Seite müssen an einen Grundterm gebunden sein

Haskell:
let .. in .. (implizit) erlaubt die Verwendung lokaler Variablen innerhalb des Funktionsrumpfes, alternative Notationsweise: let { .. ; ..} in ... (explizit)
bedingte Ausdrücke: f x = if cond x then g x else h x (f, g und h sind Funktionen)

Kap 3: Theorie
Eine *Signatur* gibt Belegung/Stelligkeit/Zugehörigkeit von Relations- und Funktionssymbolen an.

formale Definition (Signatur) : Eine Signatur ist ein Paar (F,R) (Tripel (F,R,K)) mit
F ist endliche Menge von Funktionssymbolen fi mit Stelligkeit mi (0 <= i <= m)
R ist endliche Menge von Relationssymbolen rj mit Stelligkeit nj (0 <= j <= n).
(K ist endliche Menge von Konstantensymbolen)

atomare Terme: Variablen und Konstanten
komplexe Terme: Anwendung von Funktionen und Termschachtelungen

Noch einmal: Relationen (Formeln, boolesch) können **nicht** in Funktionen (Termen) vorkommen, umgekehrt ab schon.
komplexe Formeln: enthalten andere Formeln

Formeln und Terme sind reine Syntax.

Ein Universum ist ein Individuenbereich, über den Aussagen gemacht werden sollen. Dabei bezeichnen Konstanten diese Individuen.
Funktionssymbole leisten dabei eine Abbildung zwischen den verschiedenen Individuen, sind auch als Tupel darstellbar. Sie sind partiell, wenn sie nicht für alle Tupel definiert sind, sonst total.

Relationssymbole sind eine Menge von Tupeln, die Beziehungen zwischen Tupeln darstellen.
Sie sind immer partiell.
Interpretationen legen die Bedeutung fest, Signaturen die Syntax.

Interpretationen bestehen aus: (rot weil in der Übung als falsch deklariert)
einer nichtleeren Menge von Objekten, dem Univerum U
einer Konstanteninterpretation I[K], die jeder Konstanten c ∈ K ein Objekt I[K](c) ∈ U zuordnet
einer Funktionsinterpretation I[F], die jedem n-stelligen Funktionssymbol f ∈ F eine n-äre Abbildung I[F](f) ∈ U zuordnet
einer Relationsinterpretation I[R], die jedem n-stelligen Relationssymbol r ∈ R eine n-äre Relation I[R](r) ∈ U zuordnet

Modell: Ein Modell ist eine Interpretation, in der eine bestimmte, geschlossene Formel/-menge erfüllt ist (d.h. formula = wahr), das gilt relativ zur vorgegebenen Formel (für manche Formeln kann eine Interpretation ein Modell sein, für andere nicht). Eine Interpretation ist nur dann ein Modell für eine Formelmenge, wenn sie alle der in der Menge vorkommenden Formeln wahr macht.

alternative Bezeichnungen:
erfüllbar = konsistent, widerspruchsfrei ; unerfüllbar = inkonsistent, widersprüchlich

Eine Formel F ist *falsifizierbar*, wenn es mindestens eine Interpretation gibt, die kein Modell von F ist.
Eine Formel F ist *allgemeingültig* (Tautologie), wenn jede Interpretation ein Modell von F ist.
F ist allgemeingültig <=> nicht(F) ist unerfüllbar ; F ist falsifizierbar <=> nicht(F) ist erfüllbar

Inferenzregeln (rein syntaktische Folgerungsverfahren):
Korrektheit: Jede Inferenzregel muss zu Schlussfolgerungen kommen, die gemäß dem semantischen Folgerungsbegriff akzeptiert werden.
Nachweis Inkorrektheit kann leicht mit einem Gegenbeweis erfolgen.
Nachweis Korrektheit kann nur auf Meta-Ebene erfolgen.

syntaktische Folgerungsbeziehung:
F |- G g.d.w. es eine Herleitung F = F1 = F2 = ... = Fn = G mit den Regeln des Kalküls gibt.

Ein Kalkül ist korrekt, wenn F |- G(Syntax) nur dann gilt, wenn auch F |= G (Semantik) gilt.
Ein Kalkül ist vollständig, wenn immer, wenn F |=G gilt, auch F |- G herleitbar ist.
-> allerdings muss diese Herleitung nicht immer terminieren!

Transformationsprozess beliebiger Formel in Prolog Formel:
beliebige Formel -> Eliminierung mehrfach vorkommender Bindungen der gleichen Variablen -> Pränex-Normalform -> konjunktive Normalform -> Skolem-Normalform -> Klausel-Normalform
Dabei ist zu beachten, dass es sich dabei um starke Äquivalenzumformungen handelt, bis auf die zwei Umformungen KNF -> SNF und SNF-> Kl-NF, die nur noch schwach äquivalent sind.

Pränex-Normalform:
verschiedene Quantoren sind nicht vertauschbar!
Ziel: Formel in Pränex-Normalform, wobei alle Quantoren vor der eigentlichen Formel stehen und die Formel an sich quantorenfrei ist
Dazu muss beachtet werden, dass auch sämtliche Negationen hinter die Quantoren gehören und sich dadurch Quantoren vertauschen (All wird zu Existenz und umgekehrt).

konjunktive Normalform (KNF):
Eine KNF ist eine Matrix, welche eine Konjunktion von Disjunktionen ist.
Dazu müssen alle Pfeiloperatoren (=>,<=>) entsprechend aufgelöst werden und die Formel bis auf oberste Ebene entschachtelt werden. Das bedeutet, dass Negationen so weit wie möglich nach innen gezogen werden müssen, also nur noch für ein Literal gelten
(Bsp.: not(a or b) -> not(a) and not(b) (deMorgan, wer's vergaß)).

Skolem-Normalform (SNF): Eliminierung der Existenzquantoren
für jeden Existenzquantor wird ein neues Funktionssymbol(*Skolemfunktion*) eingeführt, wobei die Stelligkeit dieser Funktion der Anzahl der vor diesem Existenzquantor vorkommenden Allquantoren entspricht und die Parameter dieser Funktion sind die Variablen, die durch die Allquantoren gebunden wurden. Dabei handelt es sich bei einer 0-stelligen Funktion (Existenzquantor ohne Allquantor davor) um eine Konstante(*Skolemkonstant*e). Danach werden alle Existenzquantoren entfernt.
Nach einem Satz muss es diese Skolemfunktionen geben, solange die Formel erfüllbar ist.
Die Skolem-Normalform ist zu der vorhergehenden Formel nur noch schwach äquivalent!

Nun werden auch alle Allquantoren entfernt da jede Variable als implizit allquantifiziert angenommen wird und Konjunktionen durch Kommas ersetzt, da die in Prolog leichter zu notieren sind.

Eine *Klausel* ist eine Disjunktion von Literalen.
Eine *Hornklausel* ist eine Klausel mit max. einem positivem Literal (ideal für Prolog, denn das kann man durch Äquivalenzumformung auch schreiben als positiv <= negativ1 und negativ2 und . . bzw. positiv :- negativ1, negativ2, ... , denn wir wissen ja, dass not A1 or not A2 or B äquivalent ist zu A1 and A2 => B) .

Verschiedene Arten von Hornklauseln:
genau ein positives Literal, min. ein negatives Literal:
L1 and L2 and ... and Ln => K (Regeln)
genau ein positives Literal und kein negatives Literal:
true => K (Fakten)
kein positives Literal, min. ein negatives Literal:
L1 and L2 and ... and Ln => false (Anfrage)
kein positives und kein negatives Literal:

true => false (Widerspruchsbeweis)

Seien F,G geschlossene Formeln. G folgt aus F (F |= G) g.d.w. gilt: alle Modelle von F sind auch Modelle von G.

F |= G g.d.w. (F => G) allgemeingültig.

Der Beweis, dass G nicht aus F folgt, geschieht am Besten per Gegenbeispiel durch Angabe eines Modells, in dem das nicht gilt.

F und G heißen stark äquivalent (F |=| G) g.d.w. F und G unter jeder Interpretation denselben Wahrheitswert besitzen.

Wenn G aus F konstruierbar ist, dann ist G eine Folgerung von F (d.h. es reicht aus, syntaktisch eine Konstruktion von G aus F zu basteln (z.B. per Inferenzregeln) um Folgerung zu beweisen, anstatt unendlich viele Modelle überprüfen zu müssen).

Herbrand-Universum:

Unter Umständen ist die Verwendung einer geeigneten Referenzinterpretation ausreichend, um Erfüllbarkeit nachzuweisen. Dabei werden Terme als Objekte aufgefasst und interpretieren sich selbst. Dann befinden sie sich in einem speziellen Universum, nämlich dem Herbrand-Universum.

Wenn Universum und Terme fixiert sind, befindet man sich in einer Herbrand-Interpretation.

Eine geschlossene Formel in Skolem-Normalform ist genau dann erfüllbar, wenn sie ein Herbrand-Modell besitzt. Damit verzichtet man auf auswertbare Funktionen. Und zu guter letzt: alle Terme in logischen Programmen sind Herbrand-Terme! Listen in der logischen Programmierung sind Herbrand-Terme, mittels . gebildet .

Inferenzregeln ohne Vorraussetzungen heißen *Axiome*.

verschiedene Inferenzregeln: Modus ponens, Spezialisierungsregel (Wenn für alle X die Formel F gilt, dann gilt F auch für das X, das durch die Konstante a ersetzt wurde), Transitivitätsregel (Wenn aus H F folgt und aus F G, dann folgt auch aus H G), verallgemeinerte Transitivitätsregel (für mehrstellige Implikationen), Schnittregel(s.u.), Resolutionsregel(s.u.).

Schnittregel:

(ggf. vorher noch durch Spezialisierungsregel gleich machen.)

1. Finde identische Literale auf komplementären Seiten von implikativen Klauseln
2. Streiche diese Literale aus den beiden Klauseln
3. verschmelze Klauselreste zu einer Klausel (was links von der einen Implikation stand wird mit dem linken Teil der anderen Implikation ver-und-et , was rechts der einen Implikation stand wird wiederum mit dem rechten Teil der anderen Implikation ver-oder-t).

Resolutionsregel:

wie Schnittregel, aber die herauszuschneidenden Literale müssen nicht mehr völlig identisch sein, sondern nur noch unifizierbar.

Unifikation:

Zwei Literale L1 und L2 heißen unifizierbar, g.d.w. es ein Literal gibt L3 gibt, das sowohl eine Instanz von L1 sowie auch von L2 ist. Dazu ist ein Unifikator nötig, am besten der allgemeinste

Unifikator, um allgemeinste gemeinsame Instanz zu bilden.
Definition (allgemeinste gemeinsame Instanz): L3 ist allgemeinste gemeinsame Instanz von L1 und L2, g.d.w. jede andere gemeinsame Instanz L3' ihrerseits Instanz von L3 ist.
Definition (allgemeinster Unifikator): U ist allgemeinster Unifikator von L1 und L2, g.d.w. U die Literale L1 und L2 in eine allgemeinste gemeinsame Instanz überführt. Abgesehen von individuellen Variablensymbolen sind allg. Instanz und Unifikator sogar eindeutig.

Faktorisierung: Unifizierbare Literale auf der selben Seite einer Klausel dürfen auch gegeneinander resolviert werden. Damit werden zusammen mit Resolution alle Klauseln widerlegungsvollständig.
Nur mit Resolution alleine sind nur Hornklauseln widerlegungsvollständig. (Russelsches Paradoxon)

Widerspruchsbeweis: Wenn (F und nicht(G)) unerfüllbar, dann muss F |= G gelten.
Ein Kalkül mit Widerspruchsbeweis kann sogar vollständig sein (nicht nur widerlegungsvollständig).
Anfragen werden intern in Prolog als negative Einheitsklauseln dargestellt:
not(anfrage) bzw. anfrage => false .
Wegen der Notwendigkeit der Hornklauseln in der logischen Programmierung lassen sich negative Informationen eigentlich nicht innerhalb eines logischen Programmes darstellen.

Ordnung der Operationalisierung in logischen Programmen:
auf der linken Seite stehen die Resolventen, auf der rechten nur Klauseln aus dem Programm

SLD-Resolution: Einschränkungen der Resolutionsmöglichkeiten (effizienter!):

- Regeln dürfen nicht mit anderen Regeln resolviert werden
- Resolvente aus Schritt i muss sofort für Input von Schritt i+1 verwendet werden
- eine Resolvente darf nur einmal Input sein, Regeln aber beliebig oft
- Startklausel ist immer die negative Klausel = Anfrage

Die SLD-Resolution ist widerlegungsvollständig für Hornklauseln, für alle anderen nicht.

Die Effizienz eines Prologprogramms ist aber letztendlich vom Programmierer abhängig. Dazu gilt nach wie vor zu beachten, dass die Selektion innerhalb Prolog von oben nach unten (Regelköpfe) und von links nach rechts (Literale) erfolgt.

SLDNF ist eine Erweiterung von SLD für negative Klauseln.
Wird ein negatives Literal gefunden, wird eine Seitenauswertung gestartet, in der das eigentlich negative Literal als positiv zu beweisen versucht wird, danach wird das Gegenteil des Ergebnis zurückgegeben.

Kap 4.1:
In Haskell wird wiederholt nach Teilausdrücken gesucht (von oben nach unten), die auf die linke Seite einer Gleichung passen und bei Erfolg durch die rechte Seite der Gleichung ersetzt.
Die Schwierigkeit besteht darin, aus Problemlösungen Rechenaufgaben zu machen.

In Haskell sind Ergebnisse einer Rechnung immer eindeutig (Konfluenz; Church-Rosser-Eigenschaft).
Allgemein benötigt Haskells lazy evaluation höchstens soviele Schritte wie eine eager -Strategie, da jeder Teilausdruck höchstens einmal ausgewertet werden muss.

Rekursionen:
linear: iterative Rekursion, Endrekursion
nicht-linear: geschachtelte und ungeschachtelte Baumrekursion

linear bedeutet, dass max. ein rekursiver Aufruf pro Funktionsapplikation stattfindet (z.B. fac n)
Endrekursion: Das Ergebnis ist direkt das Resultat des Unteraufrufs, es müssen keine Nachbearbeitungen mehr stattfinden (keine Multiplikationen, Additionen etc.), Vorteil: der Speicherbedarf ist unabhängig von der Rekusionstiefe
iterative Rekursion: Endrekursiv und zusätzliche sind die Argumente Basiswerte

nicht-linear bedeutet, das mehr als ein rekursiver Aufruf pro Funktionsapplikation auftritt
ungeschachtelte Baumrekursion: hier sind die Argumente rekursionsfrei
(z.B. fib n = fib n-1 + fib n-2)
geschachtelte Baumrekursion: hier können sogar die Argumente Rekusionen enthalten
(z.B. Ackermann)

In Prolog gibt es nur die Endrekursion als Optimierung: jede Regel hat max. einen rekursiven Aufruf und dieser steht immer als letztes Literal.

Baumtraversierung: Überführung von Bäumen in Listen.
Tiefendurchläufe: folgen der rekursiven Definition von Bäumen
Breitendurchläufe: ebenenweise Aufzählung der Markierungen

inorder: linker Teilbaum, wurzel, rechter Teilbaum (Wurzeln wo sie hingehören)
preorder: wurzel, linker Teilbaum, rechter Teilbaum (Wurzeln zuerst)
postorder: linker Teilbaum, rechter Teilbaum, Wurzel (Wurzeln zuletzt)

In Haskell können Funktionen durch *lambda-Abstraktionen* dargestellt werden. Deren Applikationen werden durch Termersetzung formalisiert (z.B.: (\x y -> +(*(x,x),y)) 2 3). Damit sind anonyme Funktionen möglich.
Funktion foldr fn x [y1:ys] = fn(... ,fn(y3,fn(y2,fn(x,y1))) ..)

Kap 4.2:
Prolog: Programme sind Daten, gelegentlich auch umgekehrt.
vordefinierte Meta-Prädikate: var(X), nonvar(X), atom(A), integer(I), atomic(A), functor(Term,Funktor,Stelligkeit), arg(Position,Term, Ziel), name(Wort,Liste)

Univ-Operator: **=..**

T =.. L ist beweisbar, wenn T ein beliebiger Term ist und L eine Liste, deren erstes Element der Funktor von T und die restlichen Elemente die Argumente von T.

-> geeignet für die Konstruktion Relationen höherer Ordnung

Differenzlisten:

Motivation: schnell auf das Listenende zugreifen und zwar mittels einer Variablen, die das Listenende explizit referenziert (z.B.: ([1,2,3|Xs],Xs))

sie steigern genau dann die Effizienz, wenn man auf das rekursive Durchhangeln der Listen verzichten kann (z.B. wenn man nicht die ganze Liste bearbeiten will, sondern nur das letzte Element).