Seminar Funktionale Programmiertechniken
SS 2007

"Logikprogrammierung mit Haskell"

von Axel Stronzik
Betreuer: Michael Hanus

Inhaltsverzeichnis

Kapitel 1

Einleitung zum Thema

Um das Ziel der Einbettung logischer Lösungstrategien deutlicher zu erfassen, eine kurze Einführung in die logische Programmierung am Beispiel von Prolog. Programme in Prolog sind eine Menge von Fakten und Relationen. Im Gegensatz dazu sind Programme in Haskell eher als eine Menge von Funktionsdefinitionen anzusehen. Betrachtet wird die 'append' Relation, die bei einen Aufruf `appen(X,Y,Z)` prüft ob `Z` die Konkatenation von `X` und `y` ist. Die erste Programmregel besagt, dass man bei dem Fall `X == []` und `Y == Z` nichts mehr überprüfen muss. Die zweite Regel prüft, ob die jeweils ersten Elemente von `X` und `Z` gleich sind, und ruft sich rekursiv auf:

```
append([ ], Ys, Ys).
append([X|Xs], Ys, [X|Zs]) :- append(Xs, Ys, Zs).
```

Zu sehen sind einige deutliche Unterschiede zur funktionalen Programmierung in Haskell. Zum einen werden hier Listen mit `[X|Xs]` dargestellt, dies entspricht genau der Haskellsyntax `(x:xs)` . Zum anderen sieht man einen dritten Parameter, welcher Funktionswert enthält. Der Sinn wird klar, wenn man sich den typischen Aufruf in der Prologumgebung ansieht:

```
append([1,2], [3,4], X).
```

Das Prologsystem wird nun alle möglichen Lösungen für die Variable `X` ausgeben. In diesem Fall ist es: `X = [1,2,3,4]`.

Ein weiterer Unterschied zur funktionalen Programmierung ist die 'leere' erste Regel, die man in Prolog Fakt (oder auch Wissen) nennt. Da in Prolog eher Beweise geführt werden als berechnet wird, gelten diese leeren Regeln als 'wahr', müssen also nicht mehr bewiesen werden.

Sehr charakteristisch für logische Programmierung, und auch eine seiner großen Stärken, ist die Tatsache, das alles relational betrachtet wird. So kann man nicht nur den letzten Parameter der append Funktion als Variable übergeben um die Konkatenation zu berechnen, sondern an beliebiger Stelle. So kann

man z.B. die Anfrage `append(X, Y, [1,2,3,4])` an das Prologsystem stellen. Das System gibt uns dann alle Kombinationen von Lösungen von `X` und `Y` aus. Jede Kombination von `X` und `Y` wird konkateniert die Liste `[1,2,3,4]` ergeben.

Also sieht die Ausgabe so aus:

```
X = [1,2,3,4]
Y = []

X = [1,2,3]
Y = [4]

X = [1,2]
Y = [3,4]

X = [1]
Y = [2,3,4]

X = []
Y = [1,2,3,4]
```

Dieses relationale Berechnen funktioniert bei den meisten Prologprogrammen und kann so zu noch mehr Modularität und Wiederverwendung führen. Im Folgenden wird erläutert wie man ein Prologprogramm sinngemäß in ein Haskellprogramm überführen kann.

Kapitel 2

Semantisch korrekte Übersetzung von logischen Programmen in Haskell

Um ein reines Prolog Programm in die funktionale Sprache Haskell übersetzen zu können, braucht man zwei Datentypen: Terme (`Term`) und Prädikate (`Predicate`), sowie die folgenden vier Operatoren:

```
&, ||:        Predicate -> Predicate -> Predicate
≐:            Term -> Term -> Predicate
∃:            (Term -> Predicate) -> Predicate
```

`&` und `||` bezeichnen die Konjuntion bzw. Disjunktion von Prädikaten, $\doteq$ ein Prädikat für die Äquivalenz, und später auch für die Unifikation von Termen, und $\exists$ den Existenzquantor. Im Sinne der Logikprogrammierung, nutzt man "`&`" um Literale aneinanderzufügen und "`||`" um Klauseln für ein Prädikat zu vereinen, $\doteq$ für die primitive Unifikation und $\exists$ zum Einführen neuer Variablen. Der Ausdruck $(\exists x_1.(\exists x_2.p(x_1 x_2)))$ wird im folgenden abgekürtzt durch $(\exists x_1, x_2.p(x_1 x_2))$ und im Haskellcode durch eine Funktion `exists :: (Term -> Predicate) -> Predicate` dargestellt. Der Parameter dieser Funktion ist ein λ-Ausdruck der die beiden Variablen erwartet und das Prädikat p mit diesen startet.
Störend bei der Einbettung sind hier das Patternmatching und das mehrfache Vorkommen einer Variable in einem Pattern. In Haskell möchte man das Programm in folgender Gestalt haben:

```
append(Ps, Qs, Rs) = (Ps ≐ nil & Qs ≐ Rs) ||
      (exists (\ X Xs Ys -> Ps ≐ cons(X,Xs) &
      Rs ≐ cons (X, Ys) &
      append(Xs, Qs, Ys))).
```

Die Konstruktoren `nil` und `cons` die typischen Listenkontrucktoren. Die zwei Regeln werden mit dem `||` Operator aneinander gehängt. Das Pattermatching wird explizit durch neue Variablen und das $\doteq$ ersetzt, Variablen quantifiziert und Funktionsanwendungen werden mit dem `&` verbunden.

Diese Transformation kann man auch systematisieren und insbesondere auch automatisieren. Diese Transformation nennt man die sogenannte Vervollständigung von Prädikaten. Die folgenden fünf Schritte, nach einer Idee von Clark, werden das Vorgehen verdeutlichen. Hier bezeichnen x_i neue Variablen, $\bar{x}$ und $\bar{y}$ Variablensequenzen, und $true, false, \vee, \wedge, \exists, \longleftrightarrow$ sind die typischen primitiven Operatoren der Prädikatenlogik erster Stufe und F steht für eine Regel erster Ordnung.

1. Entfernen von Pattern aus dem Regelkopf: ersetze jede Regel $r(t_1, \ldots, t_n) \leftarrow$ q durch $r(x_1, \ldots, x_n) \leftarrow x_1 = t_1 \wedge \ldots \wedge x_n = t_n \wedge q$

2. Lokale Variablen quantifizieren: ersetze jede Regel $r(\bar{x}) \leftarrow q$ die man aus 1. bekommt durch $r(\bar{x}) \leftarrow \exists \bar{y}.q$, wobei $\bar{y}$ alle Variablen enthält, die in q vorkommen, aber nicht in $\bar{x}$. Also die Liste aller ursprünglichen Variablen.

3. Einführung der Disjunktion: ersetze alle Regeln $r(\bar{x}) \leftarrow F_1, \ldots, r(\bar{x}) \leftarrow F_n$ die man aus 2. bekommt und zu dem Prädikat r gehören durch eine Regel $r(\bar{x}) \leftarrow F_1 \vee \ldots \vee F_n$. Falls $F_1 \vee \ldots \vee F_n$ leer ist, ersetze es durch *true*.

4. Einführung eines expliziten Fehlschlags: füge Regel $r(\bar{x}) \leftarrow$ *false* für jede Relation r die in keinem Kopf einer Regel des Programms vorkommt.

5. Ersetze Implikation durch Äquivalenz: ersetze jede Regel $r(\bar{x}) \leftarrow q$ durch $\forall.\bar{x}(r(\bar{x}) \longleftrightarrow F)$

Da es sich, nach Anwendung dieser Regeln, um Äquivalenzen handelt, ist es sogar mit dem funktionalen Gedanken kompatibel "Gleiches durch Gleiches zu ersetzen", und man kann es direkt in Haskellsyntax übersetzen.

Kapitel 3

Einbettung von Prolog in Haskell

Da man die oben genannten Vorteile von Prolog auch außerhalb von Prolog nutzen möchte, hat mich sich überlegt, ob man dieses relationale Verhalten auch in Haskell einbetten kann. Dieser Abschnitt befasst sich mit der Idee dieser Einbettung und ihrer konkreten Implementierung.

3.1 Die Tiefensuche

Da Prolog zu einem Problem nicht nur eine Lösung liefert, sondern alle möglichen, nimmt man für die Neuimplementierung die naheliegenste Datenstruktur als Ergebnis, eine Liste. Daraus folgt auch, dass alle Prädikate die eine 'Antwort' nehmen, eine Liste von Antworten liefern müssen:

```
type Predicate = Answer → [Answer]
```

Da Antworten im Prinzip nur Substitutionen von Variablen sind, kann man den Typ von "Antwort" auch leicht festlegen, wobei man sich zusätzlich noch merken sollte, wie viele Variablen man schon betrachtet hat, um die Generierung neuer Variablen zu vereinfachen:

```
type Answer = (Subst, Int)

type Subst = [(Var, Term)]

data Term = Func Fname [Term] | Var Vname

type Fname = String
```

```
data Vname = Name String
```

Konstanten sind Funktionen mit Stelligkeit 0, also ohne Parameter. So kann man z.B. die Prologliste [1,2] darstellen durch `Func ''cons'' [ Func ''1'' [], Func ''cons'' [ ... ]]` (oder später direkt in Haskell: `cons a (cons b nil)`). Variablen haben immer folgende Gestalt: `Var (Name ''Variablenname'')`. Die Namen automatisch generierter Variablen werden aus einem kleinen `x` und einer Zahl größer null zusammengesetzt. Das Programm sorgt dafür das die generierten Variablen immer eindeutige Bezeichner haben
Primitive Prädikate wie `true` und `false` sind, angewendet auf eine Antwort, der sofortige Erfolg bzw. Fehlschlag:

```
true :: Predicate            false :: Predicate
true x = [x]                 false x = [ ]
```

Das Patternmatching wird durch den oben genannten $\doteq$ Operator realisiert. Dieser nutzt die `unify :: Subst -> (Term, Term) -> Maybe Subst` Funktion, welche den Standardunifikationsalgorithmus von J.A. Robinson implementiert. Sie liefert eine Substitution als Ergebnis, falls es eine gibt. Eine Substitution ist eine Liste von Variablen-Term-Paaren, in der jeder Variablen genau ein Term zugeordnet wird:

```
(≐) :: Term -> Term -> Predicate

(t≐u)(s,n)=[(s',n)| s' <- unify s (t,u)]
```

Eine Substitution $\sigma = \{x \backslash [1,2], y \backslash x\}$ hat folgende Gestalt:

```
sigma = [(Name ''x'', Func ''cons'' [Func ''1'' [],
                         Func ''cons'' [Func ''2'' [],
                         Func ''nil'' []]]),
        (Name ''y'',Var (Name ''x''))].
```

Wie bereits oben erwähnt hat man die $\exists$ Funktion um neue Variablen in unsere Berechnungen einfließen zu lassen. Man nutzt hier die Funktion `makevar`, die einen Term liefert, der die n-te generierte Variable repräsentiert:

```
exists :: (Term -> Predicate) -> Predicate

exists p (s,n) = p (makevar n) (s,n+1)
```

Der interessante Teil, der auch die eigentliche Suchstrategie festlegt, sind die Definitionen der Operatoren `&` und `||`. Sie agieren als Kombinatoren und berechnen die Antworten.
Der `||` Operator konkateniert einfach nur die zurückgelieferten Antwortlisten:

```
(||) :: Predicate -> Predicate

(p || q) x = p x ++ q x
```

Hier sieht man auch in welcher Reihenfolge die Prädikate angewendet werden, und man erkennt leicht den Tiefensuchlauf wie er auch in Prolog genutzt wird. Liefert p eine leere Liste als Antwort, entspricht dies einem fehlgeschlagenen Zweig des Suchbaumes (vgl. Definition von `false`).
Die Konjunktion durch den `&` Operator realisiert man, indem zuerst das linke Argument angewendet wird. Auf jedes Element der Ergebnisliste wird nun das rechte Argument angewendet. Um die richtigen Typen zu bewahren, wird die Funktion 'concat' genutzt, um aus der Liste von Listen wieder eine einzele Liste zu machen:

```
(&) :: Predicate -> Predicate -> Predicate

p & q = concat . map q . p
```

Bekannte Gesetze für `map` und `concat` sind:

$$map\ (f\ .\ g) = (map\ f)\ .\ (map\ g) \tag{3.1}$$
$$map\ f\ .\ concat = concat\ .\ map\ (map\ f) \tag{3.2}$$
$$concat\ .\ map\ concat = concat\ .\ concat \tag{3.3}$$

Zudem kann man noch einige algebraische Gesetzmäßigkeiten festhalten. Darunter die Assoziativität des `&` Operators mit `true` als Einselement. Das Prädikat `false` ist eine linke Null für `&`. Die Assoziativität des `||` Operators und `false` als die dazugehörige Eins. zudem die Rechstdistributivität von `&` zu `||`:

$$(p\ \&\ q)\ \&\ r = p\ \&\ (\ q\ \&\ r) \tag{3.4}$$
$$p\ \&\ true = true\ \&\ p = p \tag{3.5}$$
$$false\ \&\ p = false \tag{3.6}$$
$$(p\ ||\ q)\ ||\ r = p\ ||\ (q\ ||\ r) \tag{3.7}$$
$$p\ ||\ false = false\ ||\ p = p \tag{3.8}$$
$$(p\ ||\ q)\ \&\ r = (p\ \&\ r)\ ||\ (q\ \&\ r) \tag{3.9}$$

Zuletzt braucht man noch eine Funktion, welche die Hauptanfrage startet. Diese `solve` Funktion nimmt das Prädikat als Parameter und sorgt dafür, dass es auf dir initiale Antwort angewendet wird. Zur schöneren Ausgabe hat man noch die `print` Funktion, die nur alle Werte der Variablen der initialen Anfrage ausgibt und die dynamisch generierten Variablen auslässt.

```
solve :: Predicate -> [String]
```

```
solve p = map print (p([],0))
```

Der Aufruf `append x y [1,2]` liefert als Ergebnis folgende Antwort:

```
y = Func ''cons'' [Func ''1'' [],Func ''cons''
                    [Func ''2'' [],Func ''nil'' []]]
x = Func ''nil'' []

y = Func ''cons'' [Func ''2'' [],Func ''nil'' []]
x = Func ''cons'' [Func ''1'' [],Func ''nil'' []]

y = Func ''nil'' []
x = Func ''cons'' [Func ''1'' [],Func ''cons''
                    [Func ''2'' [],Func ''nil'' []]]
```

3.2 Breitensuche

Wenn man genau die eben vorgestellten Operatoren genauer betrachtet, fällt auf, das nur die Operatoren `||` und `&` für die Festlegung der Suchstrategien verantwortlich sind. Ändert man also diese beiden Operatoren, verändert man auch gleich die Strategie. Dieses Kapitel befasst sich mit der Breitensuche als alternative Suchstrategie. Die Idee hierbei ist es nicht ein Prädikat solange zu verfolgen bis die Berechnung ein Ende nimmt, sondern nach jedem Schritt der Berechnung auch einen Schritt aller Alternativen zu berechnen. Da man einer Antwort alleine nicht ansehen kann, aus wie vielen Berechnungsschritten enstanden ist, muss man sich die sogenannten "Kosten" der bisherigen Berechnung merken, also die Anzahl der Berechnungsschritte. Zum Speichern dieser zusätzlichen Informationen reicht eine Liste als Datenstruktur nicht aus. Daher muss auf man eine Matrix (oder auch eine Liste von Listen) umsteigen:

```
type Predicate = Answer -> [[Answer]]
```

Alle Antworten in derselben Liste haben dieselben Kosten, und je weiter hinten eine Liste von Antworten ist, desto höher sind sie. Intuitiv braucht man also eine Funktion, welche die Kosten nach einem Berechnungsschritt erhöht:

```
step :: Predicate -> Predicate

step p x = []:(p x)
```

Diese Funktion muss man nun in unserem übersetzten Programm auf den Rumpf der Funktion applizieren. (Bei der Tiefensuche ist `step` die Identität):

```
append(Ps, Qs, Rs) = step ((Ps ≐ nil & Qs ≐ Rs) ||
      (exists (\ X Xs Ys -> Ps ≐ cons(X,Xs) &
      Rs ≐ cons (X, Ys) &
      append(Xs, Qs, Ys)))(.
```

Nun muss man noch die Kombinatoroperatoren `||` und `&` an die neue Datenstruktur anpassen, sodass sie die Kosten bewahren. Die Disjunktion hängt wieder nur die Antwortenlisten aneinander, allerdings nur die mit gleichen Kosten. Dazu nutzen wir Funktion `zipWith :: (a -> b -> c) -> [a] -> [b] -> [c]` aus der Haskell-Prelude. Die Konjunktion muss allerdings die Kosten ihrer Parameter addieren. Man berechnet hier zuerst alle Antworten von p, und wendet q auf jede Antwort an. Daraus resultiert eine Matrix von Matrizen. Um Typsicherheit zu garantieren, muss diese Ergebnismatrix zu einer einfachen Matrix abgeflacht werden:

```
(p || q) x = zipwith (++) (p x) (q x)

p & q = shuffle . mmap q . p
```

Die `mmap` Funktion die Komposition von `map` mit sich selbst ist. Die Funktion `shuffle` ist zum Abflachen der Matrix da, jedoch technisch zu aufwendig, weshalb hier auf die konkrete Implementierung verzichtet wird. Eine Antwort bei dieser Strategie kann wie folgt aussehen:

```
sigma = [[(Name ''xs'', Func ''cons'' [Func ''1'' [], Func ''nil'' []])]
      [(Name ''x'', Func ''2'' []), (Name ''y'', Var (Name ''z''))]
      [(Name ''z'', Func ''nil'' [])]]
```

Auch hier kann man einige algebraische Eigenschaften festhalten:

$$mmap\ (f\ .\ g) = (mmap\ f)\ .\ (mmap\ g) \tag{3.10}$$
$$mmap\ f\ .\ shuffle = shuffle\ .\ mmap\ (mmap\ f) \tag{3.11}$$
$$shuffle\ .\ mmap\ shuffle = shuffle\ .\ shuffle \tag{3.12}$$
$$zipwith\ f\ (zipwith\ f\ l_1\ l_2)\ l_3 = zipwith\ f\ l_1\ (zipwith\ f\ l_2\ l_3) \tag{3.13}$$
$$mmap\ f\ .\ zipwith\ g\ l_1\ l_2 = zipwith\ g\ (mmap\ f\ l_1)\ (mmap\ f\ l_2) \tag{3.14}$$
$$shuffle\ .\ zipwith\ (++)\ l_1\ l_2 = zipwith\ (++)\ (shuffle\ l_1)\ (shuffle\ l_2) \tag{3.15}$$

Das Prädikat `false` in diesem Modell ist dasselbe wie im vorigen, ein Fehlschlag gibt einfach eine leere Liste zurück. Jedoch muss man `true` auf die Datenstruktur der Matrix anheben, also: `true x = [[x]]` .

Die Gesetze (3.4-3.9) gelten auch für dieses Modell. Die Beweise belaufen sich auf einfache Anwendungen der oben festgehaltenen Regeln. So kann man zum Beweis der Assoziativität von `||` die Assoziativität von `zipwith` (3.13) ausnutzen. Als konkretes Beispiel folgt hier der Beweis der Assoziativität von &:

(p & q) & r

$= shuffle\ .\ mmap\ r\ .\ shuffle\ .\ mmap\ q\ .\ p$	*Def. von &*
$= shuffle\ .\ shuffle\ .\ (mmap\ mmap\ r)\ .\ mmap\ q\ .\ p$	*nach (3.11)*
$= shuffle\ .\ mmap\ shuffle\ .\ (mmap\ mmap\ r)\ .\ mmap\ q\ .\ p$	*nach (3.12)*
$= shuffle\ .\ mmap\ (shuffle\ .\ mmap\ r\ .\ q)\ .\ p$	*nach (3.10)*
$= p\ \&\ (q\ \&\ r)$	*Def. von &*

Nimmt man bei dieser Strategie denselben Aufruf (`append x y [1,2]`) wie bei der Tiefensuche, werden auch dieselben Antworten (evtl. in anderer Reihenfolge) als Ergebnis berechnet. Diese Strategie ist berechnungsstärker als die Tiefensuche, da hier eine Antwort gefunden wird, wenn es eine gibt. Die ist bei der Tiefensuche im Allgemeinen nicht der Fall, wenn der erste Zweig des Suchbaumes ein unendlicher Zweig ist.

3.3 Korrektheit und Vollständigkeit

Zur Korrektheit und Vollständigkeit sollte man hier nur erwähnen, das man sie beweisen kann. Die Beweise halten sich sehr nahe an denen von Clark für die Korrektheit und Vollständigkeit (für positive Literale) bei der Berechnung von vollständigen logischen Programmen. Allerdings muss man bei den Beweisen, für diese Implementierung, auf Einschränkungen achten die bei logischen Prädikaten erster Stufe nicht vorhanden sind. Die Operatoren die in Haskell implementiert wurden sind spezieller, so kann man nicht davon ausgehen das der & Operator immer kommutativ ist.
Die Beweisführung ist meist duch Induktion über die Terme durchführbar, jedoch sprengt das benötigte Hintergrundwissen den Rahmen dieser Ausarbeitung. Bei Interesse ist dies in [1] nachzulesen.

Kapitel 4

Verallgemeinerung der Suchstrategien

4.1 Die generalisierte Suchstrategie

Dieses Kapitel wird auf eine generalisierte Idee der Suchstrategien eingegangen. Später wird auch gezeigt das die Tiefen- und Breitensuche Spezialfälle dieser allgemeinen Suchstrategie sind. Die Funktionen werden so ummodelliert, dass sie mit Wälder arbeiten. Hierbei soll jeder innere Knoten beliebig viele Söhne haben und die Antworten aller Prädikate sind in den Blättern zu finden. Dazu muss man auch den Typ von `Predicate` leicht abändern:

```
type Predicate = Answer -> Forest Answer

type Forest a = [Tree a]

data Tree a = Leaf a | Fork (Forest a)
```

Die Kosten einer Antwort korrespondieren zu der Tiefe des Knotens. Folglich muss die `step` Funktion, die die Kosten um eins erhöht, alle bisherigen Antworten eine Ebene herunterschieben und eine neue Wurzel hinzufügen:

```
step p x = [Fork (p x)]
```

Die Operatoren sind ähnlich denen der Tiefensuche, der `||` Operator bleibt sogar gleich. Der `&` Operator berechnet auch hier zuerst alle Antworten des linken Arguments und wendet das rechte Argument mit einer Funktion `fmap :: b -> (b -> a -> b -> b) -> Tree a -> b` auf den Ergebniswald an. Daraus resultiert ein Wald von Antworten an jedem Blatt, diese werden dann von der `fgraft` Funktion in den 'alten' Wald wieder eingearbeitet:

```
(p || q) x = p x ++ q x

p & q = fgraft . fmap q . p
```

Der Grund warum das Modell des Waldes dem des einfachen Baumes als Antworten vorzuziehen ist, liegt daran, dass man keine kostenbewahrenden Implementierungen von `||` und `&` hat. Bei einfachen Bäumen würde `p|| q` seine Bäume kombinieren müssen, indem man sie unter einer neuen Wurzel zu einem neuen Baum zusammenfasst. Dadurch würden allerdings die Kosten für alle Antworten von `p|| q` um 1 steigen. Zum Beispiel würden dann die Antworten von `p|| false` mehr Kosten als die Antworten von `p`, was falsch wäre, da die Anzahl der Resolutionsschritte gleich ist.

4.2 Beziehungen zwischen den Suchstrategien

In der eben vorgestellten allgemeinen Suchstrategie erhält man aus einer Anfrage an das logische Programm einen ganzen Wald von Antworten. Da die anderen Suchstrategien dieselben Antworten liefern, nur in eine andere Datenstruktur verpackt, liegt die Idee nahe eine Konvertierungsfunktion zu schreiben, die den Wald auf dieselbe Weise durchläuft, wie die Tiefensuche bzw. Breitensuche und die dementsprechende Datenstruktur mit Antworten liefert. Die Funktionen `dfs` und `bfs` sollen dies bewerkstelligen.

Die `dfs` Funktion wendet die Hilfsfunktion `dfs2` auf jeden Baum der Liste an, und konkateniert die Ergebnisse. Wobei die Funkton `dfs2` wie bei einer Tiefensuche die Blätter aus jedem Baum zurückliefert, und rekursiv `dfs` aufruft:

```
dfs :: Forest Answer -> [Answer]
dfs = concat . map dfs2

dfs2 (Leaf x) = [x]
dfs2 (Fork xf) = dfs xf
```

Die `bfs` Funktion muss die Kosten in Betracht ziehen. Sie macht dies indem sie die Bäume des eingegebenen Waldes, mit der Hilfsfunktion `bfs2`, in Matrizen (Listen von Listen) umwandelt. Sie holt sich dafür aus allen Bäumen die Antworten mit gleichen Kosten und fügt sie in eine Liste ein:

```
bfs :: Answer -> [[Answer]]
bfs xf = foldr (zipwith (++)) [ ] (map bfs2 xf)

bfs2 (Leaf x) = [x]:repeat [ ]
bfs2 (Fork xf) = [ ]:bfs xf
```

Kapitel 5

Mathematischer Hintergrund: Monade

Bei der im vorigen Kapitel angebenen Funktion `dfs` kann man auf Grund der einfachen Implementierung schon erkennen, dass hier die gleichen Ergebnisse geliefert werden, egal ob man es direkt mit der Tiefensuchstrategie berechnet, oder mit der allgemeinen. Bei der `bfs` Funktion wird das schon schwieriger. Um zu beweisen das es sich um dieselben Antworten handelt, wird in diesem Kapitel die Monade als mathematisches Gerüst vorgestellt. Man wird dann sehen, das jede Suchstrategie eine Monade darstellt und die Funktionen `dfs` und `bfs` Morphismen zwischen diesen Monaden sind.

Eine Monade ist ein Typkonstruktor T mit einem Tripel $(map_T,\ unit_T,\ join_T)$, wobei $(map_T, unit_T$ und $join_T)$ polymorphe Funktionen mit folgender Typsignatur sind:

$$map_T :: (a \to b) \to T\ a \to T\ b$$

$$unit_T :: a \to T\ a$$

$$join_T :: T\ (T\ a) \to T\ a$$

Außerdem müssen für diese Funktionen folgende Gesetzmäßigkeiten gelten:

$$map_T\ id = id \tag{5.1}$$
$$map_T\ (f\ .\ g) = map_T\ f\ .\ map_T\ g \tag{5.2}$$
$$map_T\ f\ .\ unit_T = unit_T\ .\ f \tag{5.3}$$
$$map_T\ f\ .\ join_T = join_T\ .\ map_T\ (map_T\ f) \tag{5.4}$$
$$join_T\ .\ unit_T = id \tag{5.5}$$
$$join_T\ .\ map_T\ unit_T = id \tag{5.6}$$
$$join_T\ .\ map_T\ join_T = join_T\ .\ join_T \tag{5.7}$$

Betrachtet man nun das Modell der Tiefensuche, ist der Typkonstruktor T die Liste mit der gearbeitet wird. Nimmt man dann die normale `map` Funktion als map_T, den Listenkonstruktor `[_]` für $unit_T$ und `concat` als $join_T$, kann man leicht zeigen, dass `(map, [_], concat)` eine Monade ist. Die Gleichungen (5.2), (5.4) und (5.7) stimmten mit den Standardgesetzen für Listen (3.1-3.3) überein. Die restlichen Gleichungen folgen direkt aus den Definitionen für `map` und `concat`.
Ähnliches gilt für die Breitensuche. Hier ist T gleich der Matrix und man betrachet das Tripel `(mmap, [[_]], shuffle)` . Auch hier kann man diese sieben Gesetze direkt aus den Definitionen der Funktionen ableiten. Dasselbe gilt für das generalisierte Suchverfahren, für das `(fmap, [Leaf _], fgraft)` die Monade bildet.

Eine alternative Definition für diese Monaden kann man mit Hilfe eines neuen Operator erreichen, die Kleisli-Komposition:

```
(*) :: (a -> T b) -> (b -> T c) -> (a -> T c)
p * q = join . map q . p
```

Der Kleisli Operator ersetzt nun das $join_T$, so dass man das Tripel $(map_T,\ unit_T,\ *_T)$ hat. Dieses Tripel ist eine Monade wenn folgende Eigenschaften gelten:

$$(unit_T\ a) * k = k\ a \tag{5.8}$$
$$m * unit_T = m \tag{5.9}$$
$$m * (p * q)) = (m * p) * q \tag{5.10}$$

Die Gleichungen (5.8 - 5.10) werden von den Gleichungen (5.1 - 5.7) impliziert. Umgekehrt kann man aber auch die Funktionen map_T und $join_T$ mit Hilfe von $*_T$ definieren, dann implizieren die neuen Gleichungen unsere ursprünglichen. Damit sind die beiden Definitionen äquivalent.

Bei der zweiten Definition fällt auf, dass der Operator $*_T$ sich genauso verhält wie der `&` Operator in jedem unsere Modelle, und $unit_T$ entspricht der Funktion `true` jeden Modells. Es wurde schon gezeigt das `&` assoziativ (3.4), mit `true` als Einselement (3.5), ist. Damit genügt es schon den Gleichungen (5.8 - 5.10). Wenn man nun zu der Monade die anderen strukturellen Teile der drei Modelle hinzufügt, kann man eine erweiterte Monade T^+ als 5-Tupel definieren:

$$T^+ = (map_T,\ true_T,\ false_T,\ ||_T,\ \&_T)$$

Ein Morphismus zwischen zwei erweiterten Monaden ist eine Funktion, welche die Struktur der Monade erhält. Sie muss also `true` der einen Monade auf `true` der anderen Monade abbilden, genauso wie mit `false`, $||_T$, und $\&_T$. Bei genauer Betrachtung der oben schon eingeführten Funktionen `dfs` und `bfs` fällt auf, dass diese genau die Anforderungen erfüllen. Als Beispiel hier die zu zeigenden Gleichungen für die `dfs` Funktion, W steht für das Waldmodell, also

die generalisierte Suchstrategie und L ist die Tiefensuche, mit der Liste als Datenstruktur:

$$dfs\ .\ true_W = true_L \tag{5.11}$$
$$dfs\ .\ false_W = false_L \tag{5.12}$$
$$dfs\ .\ fmap\ f = map\ f\ .\ dfs \tag{5.13}$$
$$dfs\ .\ (p\ \&_W\ q) = (dfs\ .\ p)\ \&_L\ (dfs\ .\ q) \tag{5.14}$$
$$dfs\ .\ (p\ ||_W\ q) = (dfs\ .\ p)\ ||_L\ (dfs\ .\ q) \tag{5.15}$$

Die Gleichungen (5.11) und (5.12) werden direkt aus den Definitionen von `dfs, true, false` bewiesen. Die restlichen über Induktionen über `dfs` und einfachen gesetzen über die `map` und `concat` Funktionen.

Kapitel 6

Zusammenfassung

Die Einbindung logischer Auswertung in Haskell ist nicht sehr aufwendig. Mit einfachen Datenstrukturen kann man nicht nur das konkrete Prolog simulieren, sondern mehr Variationen in der Auswertungsstrategie einführen. Dabei kommen uns die Vorteile von Haskell zugute. Zum einen die eingebaute faule Auswertung ("lazy evaluation"), die bei der Implementierung der Funktionen unendliche Listen zuläßt. Zum anderen kann man aus mathematischer Sicht bisher nur in funktionaler Programmierung leicht über die Äquivalenz des 'klaren' Programms zum korrespondierenden effizienten Programm mathematisch argumentieren. Beweise über Haskell Programme nutzen im Allgemeinen eine begrentzte Folge von algeraischen Gesetzen für induktive Argumentation über Funktionen und Listen.

Literatur

1. S. Seres: The algebra of logic programming. Ph.d. thesis, Oxford University, 2001.
   `http://users.comlab.ox.ac.uk/silvija.seres/Papers/seres_thesis.pdf`

2. S. Seres, M. Spivey, T. Hoare: Algebra of Logic Programming. Proc. ICLP'99, MIT Press, 1999
   `http://users.comlab.ox.ac.uk/silvija.seres/Papers/seres_iclp99.pdf`