Logikprogrammierung in Haskell
Backtracking-Monade und logische Variablen

20.01.2006

Verfasser: Tobias Harneit

Betreuer: Professor M. Hanus

Primärliteratur:

1 Ralf Hinze. Prolog's control constructs in a functional setting - Axioms and implementation. International Journal of Foundations of Computer Science, 12(2):125-170, 2001

2 Ralf Hinze. Deriving Backtracking Monad Transformers. In Phil Wadler, editor, Proceedings of the 2000 International Conference on Functional Programming, Montreal, Canada, September 18-20, 2000

3 Claessen, K. and Ljunglöf, P., Typed Logical Variables in Haskell. Proc. ACM SIGPLAN Haskell Workshop, 2000

Contents

1 Einleitung

Bei vielen Problemen bietet es sich an, deklarative Programmiersprachen anstelle von imperativen zu verwenden. Dazu würde man zum Beispiel das Damenproblem, das Rucksackproblem oder ähnliche Aufgaben zählen. Eine einfache Lösung ist sicherlich mit Haskell zu erreichen, noch besser und noch einfacher geht es aber mit der logischen Programmiersprache Prolog. Der Grund dafür ist einfach, dass es in Prolog Programmkonstrukte gibt, die es uns ermöglichen, das Problem sehr einfach darzulegen und umzusetzen.

Glücklicherweise ist es in Haskell gut möglich, andere Programmiersprachen einzubetten und so setzen wir uns in dieser Ausarbeitung das Ziel, die interessantesten Programmkonstrukte von Prolog, dabei vor allem Backtracking und logische Variablen, in Haskell zu implementieren, welches wir im Folgenden auch mit *Embedded Prolog* bezeichnen. Wir wollen versuchen, dass Vorteile, die Haskell gegenüber Prolog hat, nicht verlorengehen.

Die Ausarbeitung gliedert sich folgendermaßen: Der 2. Abschnitt soll vor allem als Einführung in die Funktionalität von Prolog genutzt werden können. Wir gehen zusätzlich darauf ein, was Monaden sind und stellen Monadentransformer vor. Im dritten Abschnitt geben wir naive Implementierungen der angesprochenen Prolog-Funktionen an. Im vierten Abschnitt stellen wir Monadentransformer für die Monaden aus Abschnitt 3 vor und zeigen, wie wir daraus ein Prolog-ähnliches Haskellsystem machen können. Danach widmen wir uns im fünften Abschnitt den logischen Variablen, geben dort aber nur eine kurze Einführung an. Der sechste und letzte Abschnitt fasst die Umsetzung zusammen.

2 Grundlagen

Das vorliegende Kapitel stellt zunächst die Programmiersprache Prolog vor. Wir erinnern außerdem noch einmal an die Implementierung von Monaden in Haskell und stellen danach das Konzept von Monadentransformern vor, die es einem erlauben, zu gegebenen Monaden gewisse Funktionen hinzuzufügen.

2.1 Prolog und Backtracking

Wir wollen versuchen, Funktionen, die in der logischen Programmiersprache Prolog bereits vorimplementiert sind, selber zu konstruieren. Zur Motivation wollen wir hier zunächst ein Beispiel zur Funktionalität des Prolog-Systems geben, dass all das bereits leistet, was wir noch erreichen möchten.

Um den Quelltext in diesem Abschnitt nachzuvollziehen, werden wir hier eine sehr kurze Prolog-Einführung geben, die nur sicherstellen soll, dass man den nachfolgenden Code versteht.

Ein Programm in Prolog besteht aus einer Menge von Prädikaten. Prädikate bestehen ihrerseits aus Fakten und Implikationen (Regeln). Sehr einfach zu verstehen ist das anhand des folgenden Beispiels:

Wir wollen einen Familienstammbaum andeuten und definieren uns dazu die beiden folgenden Regeln, die aber noch nicht in Prolog-Syntax geschrieben sind.

```
Vater      :  Vater(V,K) :<=> V ist Vater von K
Grossvater :  analog
```

Mit dieser Konvention können wir nun unser Prolog-Programm beginnen. Wir notieren die Fakten:

```
vater(fritz, thomas).
vater(thomas, maria).
vater(thomas, anna).
```

In Prolog beginnen Variablen mit einem Großbuchstaben und so fügen wir unserem Programm noch die folgende Regel hinzu:

```
grossvater(G,E) :- vater(G,V), vater(V,E)
```

Das `:-` ist dabei als Implikationspfeil (hier: $\Leftarrow$) und das Komma als logisches "Und" zu lesen. Die Codezeile besagt also einfach, dass `G` Großvater von `E` ist, falls `G` Vater von `V` und `V` Vater von `E` ist.

Bei der Anfrage `vater(fritz, thomas).` liefert uns das Prolog-System auch ein `yes`. Interessanter ist aber wohl, dass bei `vater(fritz, E).` zunächst die Systemantwort `E=maria?;`, nach einem Return dann `E=anna?;` und zuletzt `no.` ausgegeben wird. Die Variable `E` bezeichnet man auch als *logische Variable*. In Abschnitt 5 werden wir auf deren Implementierung in Haskell eingehen.

Wie geht nun das Prolog-System beim Versuch, die Aufgabe zu lösen, vor? Bei der Suche der Lösung wird einer von mehreren Lösungswegen ausgewählt und dieser wird über seine Entscheidungsknoten so lange verfolgt, bis die Lösung gefunden worden ist oder der Weg sich als definitiv falsch herausgestellt hat. Ist der Weg falsch, geht es bis zum letzten Entscheidungsknoten zurück und nimmt einen anderen Weg. Es werden auf diese Weise so viele Wege ausprobiert, bis einer von ihnen ans Ziel führt.

Diese Suchstrategie, die nach dem "trial and error"-Prinizip vorgeht, bezeichnen wir als *Backtracking* (engl. Rückverfolgung). In Prolog ist Backtracking bereits implementiert. Wir werden es in Abschnitt 4 schaffen, einen Monadentransformer zu implementieren, der es uns erlaubt, Backtracking in Haskell zu einer beliebigen Monade hinzuzufügen.

2.2 Monaden

Wenn man in funktionalen Sprachen die Berechnungen in Programmen sequentiell anordnen möchte, so trifft man häufig auf ein Problem, welches wir anhand eines kleinen Beispiels verdeutlichen möchten.

Unser Ziel ist es, zunächst den Buchstaben `a` und dann den Buchstaben `b` auszugeben. Naiv könnten wir so an die Sache herangehen:

```
main ::  World -> World
main world = let _ = print "a" world
                 in print "b" world
```

Was liefert nun die `main`-Funktion? `a`?, `b`? oder gar `ab`? Tatsächlich wird eine Kopie der Welt angelegt, wir haben also eigentlich zwei statt einem Benutzer. Auch wenn das philosophisch vielleicht interessant sein mag, macht das in einer Programmiersprache wenig Sinn.

Während die Programmiersprache Clean das Problem umgeht, indem dessen Typsystem erlaubt, Werte anzugeben, von denen keine Kopie gemacht werden darf, wird bei Haskell ein anderer Ansatz zur Sequentialisierung von Programmen angewendet: *Monaden*.

Nach Definition ist eine Monade ein Typkonstruktor `m` mit den Operationen `return` und `(>>=)`, genannt `bind`. Dabei ist `m a` der Typ einer Berechnung, die ein `a` liefert. Man verwendet `return` für die triviale Berechung und `bind` für eine Sequenz von Berechnungen. In Haskell sieht das Ganze dann so aus:

```
class Monad m where
    return  :: a → m a
    (>>=)   :: m a → (a → m b) → m b
    (>>)    :: m a → m b → m b
    m >> n  =  m >>= const n
    fail    :: String → m a
    fail s  =  error s
```

Die Operation `(>>)` wird dabei so angewendet wie `bind`, jedoch mit dem Unterschied, dass bei `(>>)` das Vorergebnis der Berechnung keine Rolle spielt und nicht mit übergeben wird. Wie man unschwer erkennen kann, dient die Operation `fail` der Fehleranzeige.

Man kann zeigen, dass Monaden die folgenden, sogenannten *Monadengesetze* erfüllen:

```
return a >>= k      = k a
m >>= return        = m
(m >>= k1) >>= k2   = m >>= (λ a → k1 a >>= k2)
```

Wir werden uns im dritten Abschnitt dieser Ausarbeitung drei Monaden widmen, die insbesondere für unser eingebettetes Prolog relevant sind. Diesen Abschnitt schliessen wir ab, indem wir das obere Problem, mit Hilfe der in Haskell vordefinierten IO-Monade, lösen.

```
type IO a = World -> (a, World)
putChar ::  Char -> IO () -- also Char -> (World -> ((),World))

main ::  IO()
main = putChar 'a' >> putChar 'b'
```

2.3 Monadentransformer

Verschiedene Monaden unterscheiden sich durch die Berechnungsfunktion, die sie unterstützen. Diese sind zugänglich durch eine Anzahl von zusätzlichen Operationen, zum Beispiel unterstützt die Backtracking- Monade die Operationen `fail` für die Fehleranzeige und `(|)` für nichtdeterministische Auswahl.

Während es einfach ist, eine Monade zu konstruieren, die nur ein Berechnugsmerkmal unterstützt, gibt es keinen allgemeinen Weg, zwei Monaden zu kombinieren, die dann die verschiedenen Berechnungsmerkmale beider Ausgangsmonaden unterstützt, da diese sich untereinander beeinflussen könnten.

Allerdings gibt es eine allgemeine Methode, eine gegebebene Monade um eine bestimmte Berechnungsfunktion anzureichern. Dies geschieht durch sogenannte *Monadentransformer*, die in der folgenden Klasse implementiert werden können.

```
class MonadT τ where
  up   ::  (Monad m) ⇒ m a → τ m a
  down ::  (Monad m) ⇒ τ m a → m a
```

Ein Monadentransformer ist also zunächst ein Typkonstruktor τ, der eine Monade `m` zu einer Monade τ `m` überführt. Er muss zusätzlich die beiden folgenden Operationen bereitstellen:

- Eine Operation, um die Berechnung von der zugrundeliegenden Monade `m` zu der transformierten Monade τ `m` einzubetten (hier: `up`) und
- eine inverse Operation, welche uns erlaubt, die angereicherten Berechnungen in der zugrundeliegenden Monade zu observieren (hier: `down`).

Die Funktion `up` bettet also eine Berechnung in `m` in eine Monade τ `m` ein. `down` geht genau den umgekehrten Weg, vergißt daher aber die zusätzlich übernommene Struktur, die durch τ `m` bereitgestellt worden ist.

Wir werden im vierten Abschnitt noch einige Monadentransformer kennenlernen, so dass wir hier auf ein Beispiel zunächst verzichten wollen.

3 "Prolog"-Funktionen mit Haskell-Monaden

In diesem Abschnitt gehen wir auf drei Features ein, die in Prolog bereits fest eingebaut sind und realisieren diese Features in Haskell durch Monaden. Abschließend zeigen wir ein Anwendungsbeispiel, in dem die Funktionalitäten der Backtracking- und der Cut-Monade verdeutlicht werden.

3.1 Ausnahme-Monade

Zunächst möchten wir uns der Monade für Ausnahmebehandlung zuwenden. Wir benötigen eine Operationen, die uns signalisiert, dass ein Fehler aufgetreten ist, und eine Operation, die die Fehler auffängt. Um das zu realisieren, bietet sich der vordefinierte Typ `Either` an. Dabei führt `try m` zunächst `m` aus. Wenn `m`

bei der Ausführung einen Fehler signalisiert, sagen wir `raise e`, dann liefert `try m` die Fehlermeldung `Left e`. Ansonsten wird `Right a`, wobei `a` das Ergebnis von `m` ist, ausgegeben.

Eine Erweiterung der Klasse `Monad` um diese beiden Operationen liefert uns dann die folgende Klassendefinition:

```
type Err = String

class (Monad m) ⇒ Exc m where
  raise ::  Err → m a
  try   ::  m a → m (Either Err a)
```

3.2 Backtracking-Monade

In diesem Abschnitt möchten wir die Backtracking-Monade vorstellen. Per Definition ist eine Backtrackingmonade eine Monade mit zwei zusätzlichen Operationen: Der Konstanten `fail`, welche Fehler kennzeichnet und der binären Operation `(|)`, die für nichtdeterministische Auswahl steht.

Die Klassendefinition beinhaltet zwei weitere Operation, nämlich `once` und `sols`, die für die Ausgabe der Ergebnisse benötigt werden: Wenn wir `once m` haben, bekommen wir `Nothing` zurück, falls `m` fehlschlägt und andernfalls `Just a`, wobei `a` die erste Lösung unserer Suche ist. `sols m` liefert uns eine Liste mit allen Ergebnissen von `m`.

```
class (Monad m) ⇒ BACKTR m where
    fail ::  m a
    (|)  ::  m a -> m a -> m a
    once ::  m a -> m (Maybe a)
    sols ::  m a -> m [a]
```

Wir können die Operationen nun so interpretieren: `m | n` bedeutet, dass wir zuerst den Weg `m` nehmen und falls dieser fehlschlägt, nehmen wir `n`. Das `fail` bedeutet einfach, dass der gerade beschrittene Weg falsch ist, und wir einen anderen nehmen müssen.

3.3 Cut-Monade

Prolog liefert dem Anwender ein Prädikat `Cut`, bezeichnet mit `(!)`, mit dem man Laufzeit und Platzverbrauch von Programmen verbessern kann. Bei uns führen wir `Cut` als Unterklasse von `Backtr m` ein.

```
class (Backtr m) ⇒ Cut m where
    !    ::  m ()
    call ::  m a -> m a
```

`Cut` ist genau einmal erfolgreich und liefert `()`. Als Nebeneffekt werden alle vorigen Alternativen und Auswahlwege weggeworfen. Die Operation `call` haben wir, um den Effekt von `Cut` zu begrenzen. Dabei führt `call m` zunächst `m` aus und falls dann `Cut` in `m` aufgerufen wird, werden nur die Auswahlmöglichkeiten weggeworfen, die nach dem Aufruf von `m` getroffen wurden.

3.4 Anwendungsbeispiel

Eine interessante Anwendung für Backtracking und Cut ist ein einfacher Parser, den wir für die folgende Grammatik in EBNF-Notation entwickeln möchten:

```
program  ⟶ decl*
decl     ⟶ var var* = exp ;
exp      ⟶ var aexp* | aexp
aexp     ⟶ int | (exp)
```

Beginnen wir mit der Entwicklung eines (allgemeinen) Parsers. Ein Parser ist eine Funktion, der eine Liste von Tokens nimmt und ein Paar, bestehend aus einem abstrakten Syntaxbaum und den unverbrauchten Suffixen der Tokenliste, ausgibt.

```
type Token          = String
newtype Parser m a  = [Token] → m (a , [Token])
```

Wenn `m` nun eine Backtracking-Monade ist, können wir `Parser m` zu einer Instanz von `Monad` und `Backtr` machen.

```
instance (Monad m) ⇒ Monad (Parser m) where
   return a    = λinp → return (a, inp)
   m >>= k     = λinp → m inp >>= λ(a, inp') → k a inp'

instance (Backtr m) ⇒ Backtr (Parser m) where
   fail        = λinp → fail
   m | n       = λinp → m inp | n inp
```

Man verwendet zum Parsen gerne Monaden, da wir die Grammatiken in EBNF einfach zu einem rekursiv absteigenden Parser konvertieren können. Der Grund hierfür ist, dass wir für jedes Grammatikkonstrukt einen korrespondierenden Parserkombinator haben.

In unserer Grammatik benötigen wir Sequentialisierung, Alternation und Wiederholung. Es ist klar, dass `bind` Sequentialisierung und `(|)` Alternation repräsentieren. Die Wiederholung könnten wir so umsetzen:

```
many         ::  (Backtr m) ⇒ m a → m [a]
many m       = ms
    where ms = do a ← m
                  as ← ms
                  return (a :  as)
              | return[ ]
```

Terminalsymbole werden durch `literal` erkannt. Der Parser `satisfy p` entscheided, ob der nächste eingegebene Token zu einer Klasse von Symbolen gehört, die durch das Prädikat `p` spezifiziert werden. Zuletzt geben wir noch die Funkton `apply` an, mit der ein Parser zu einer gegebenen Liste mit Tokens gestartet wird. Die drei Funktionen können wir folgendermaßen umsetzen:

```
literal   ::  (Backtr m) ⇒ Token → Parser m Token
literal a = satisfy (a ==)

satisfy   ::  (Backtr m) ⇒ (Token → Bool) → Parser m Token
satisfy p = λinp → case inp of  [ ] → fail
                                a :  as | p a → return (a, as)
                                        | otherwise → fail

apply ::  (Backtr m) ⇒ Parser m a → [Token] → Parser m a
apply p inp = λinp 0 → do (a, rest) p inp
                          guard (null rest)
                          return (a, inp')
```

Der Guard in der zweiten Zeile von `apply` prüft dabei einfach nur, ob die Eingabe komplett von dem Parser verbraucht wurde. Wir haben nun alles, um die Beispielgrammatik umsetzen zu können. Hier ist der Parser für für die komplette Beispielgrammatik:

```
data Program = Program [Decl]
data Decl    = Decl Var [ Var ] Exp
data Exp     = Int Int
             | App Var [Exp]
type Var     = String

program      ::  (Backtr m) ⇒ Parser m Program
program      = map Program (many decl)

decl         ::  (Backtr m) ⇒ Parser m Decl
decl         = do f ← var
                  as ← many var
                  literal "="
                  e ← exp
                  literal ";"
                  return (Decl f as e)

exp          ::  (Backtr m) ⇒ Parser m Exp
exp          = do f ← var
                  as ← many aexp
                  return (App f as)
             | aexp
```

```
aexp          ::  (Backtr m) ⇒ Parser m Exp
aexp          = map Int Int
              | do literal "("
                   e exp
                   literal ")"
                   return e

var           ::  (Backtr m) ⇒ Parser m String
var           = satisfy (all isAlpha)

int           ::  (Backtr m) ⇒ Parser m Int
int           = map read (satisfy (all isDigit))
```

Wir haben nun also einen Parser für die oben angegebene Grammatik. Allerdings ist dieser Backtracking Parser von sich aus sehr verschwenderisch mit dem Speicherplatzverbrauch. Betrachten wir `m | n`. Die Eingabe kann nicht aus dem Speicher entfernt werden, solange `n` nicht aufgerufen wurde. Wenn wir beide Verzweigungen nehmen müssen und der Zweig von `m` sehr groß ist, wird ein großer Bedarf an Speicher benötigt, um auch bereits besuchte Abschnitte zu behalten, auch wenn für den Fall, dass `m` letztendlich fehlschlägt und dann `n` betrachtet werden muss.

An dieser Stelle kommt `Cut` ins Spiel. Bevor wir dessen Nutzen demonstrieren, müssen wir zu dem Bereich der Parser befördern:

```
instance (Cut m) ⇒ Cut (Parser m) where
!          = λinp → !  >>= λa → return (a, inp)
call m     = λinp → call (m inp)
```

Damit können wir die Definition von `decl` anpassen. Diese Produktion ist die einzige, die ein Literal "`=`" enthält. Sobald das Zeicken "`=`" auftritt, gibt es keinen Grund mehr, beim Backtracking zu diesem Punkt zurückzukehren.

```
decl   ::  (Cut m) ⇒ Parser m Decl
decl   = do f ← var
            as ← many var
            literal "="
            !
            e ← exp
            literal ";"
            return (Decl f as e)
```

Cut beseitigt nun effizient alle vorigen Auswahlpunkte und gibt so Speicher frei.

4 Monadentransformer für Prolog-Funktionen

Wir haben im vorigen Abschnitt einige Monaden kennengelernt, die ähnliche Funktionen liefern, wie das Prolog-System. Wenn wir nun ein eingebettetes Prolog implementieren möchten, dann könnten wir uns eine Funktion oder eine Monade definieren, die all diese Funktionen zusammenfasst. Man kann sich denken, dass das eine sehr schwierige Aufgabe werden kann und dass die Monade sehr fehleranfällig sein könnte.

Aus diesem Grund gehen wir ein wenig anders vor. Für jede Operation, die uns interessiert, geben wir einen Monadentransformer an. Dadurch können wir zu bestimmten Monaden entsprechende Features hinzufügen. Wir werden feststellen, dass zum Beispiel Backtracking zu jeder Monade hinzugefügt werden kann.

4.1 Ausnahme-Monadentransformer

Unser erster Monadentransformer, den wir angeben möchten, ist der Ausnahme-Monadentransformer. Wie bereits in Abschnitt 3.1, beschreiben wir das Vorkommen einer Ausnahme mit `Left e` und repräsentieren das richtige Ergebnis `a` durch `Right a`.

Zur Implementierung kombinieren wir einfach die Basismonade mit `Either Err`. Das liefert uns:

```
newtype ExcT m a   = m (Either Err a)

instance (Monad m) ⇒ Monad (ExcT m) where
     m >>= k       = m >>= either (return ∘ Left) k
     return        = return ∘ Right

instance (Monad m) ⇒ Exc (ExcT m) where
     raise         = return ∘ Left
     try m         = map Right m
```

Für den Monadentransformer gilt nun, dass eine Berechnung in `m` immer erfolgreich ist. Daher wird sie mit `Right` versehen, wenn sie mittels `up` zur neuen Monade geleitet werden soll. Runterlaufende Ausnahmen liefern Laufzeitfehler unter der Benutzung der vordefinierten Funktion `error`.

```
instance MonadT ExcT where
     up m          = map Right m
     down m        = m >>= either error return
```

Man kann beweisen, dass, falls `m` eine Monade ist, auch `ExcT m` eine Monade ist, die einigen signifikanten Bedingungen für Fehler genügt. Vergleiche hierzu [1, Kapitel 6.3].

4.2 Backtracking-Monadentransformer

Wir haben bereits angedeutet, dass wir es schaffen können, Backtracking zu einer beliebigen Monade hinzuzufügen. Um das zu gewährleisten, müssen die Operationen für Backtracking unabhängig von den Basisfunktionen der ursprünglichen Monade operieren. Aber fangen wir einfach mal an:

Unsere Hauptidee ist zunächst, eine Berechnung `m` in eine Funktion `m'` zu überführen, so dass `m' f = m | f` ist. Eine effiziente Variante erhalten wir dann, indem wir in `m' fail` weglassen, also dass `m = m' fail` gilt. Dabei ist der Typ von `m'` dann `m a -> m a`. Die natürliche Frage ist nun, ob es möglich ist, `m a -> m a` zu einer Monade zu machen. Wenn das der Fall ist, dann müssen die Mappings

```
up m     = λf → m | f
down m   = m fail
```

zu Monadenmorphismen werden. Weiterhin können wir uns auf die isomorphe Kopie von `m` einschränken, da wir nicht beabsichtigen, weitere Struktur zu `m` hinzuzufügen. Dabei ist klar, dass `up` und `down` gegenseitig invers sind, wenn `m f = m fail | f` gilt. Wenn wir das voraussetzen, können wir schon mal die Monadenoperationen von `m a → m a` herleiten.

```
fail     = up fail
         = λf → fail | f
         = λf → f

m | n    = up (down (m | n))
         = up (down m | down n)
         = λf → (m fail | n fail ) | f
         = λf → m fail | (n fail | f )
         = λf → m (n f)
```

Also ist `fail` dann identisch zur identischen Funktion. Interessanterweise berufen sich weder `fail` noch `(|)` auf die Primitiven von `m`.

Leider klappt das nicht so ohne Weiteres bei `return` und `bind`. Um das dennoch zu erreichen, müssen wir eine weitere Transformation durchführen, die `(>==)` abstrahiert. Jede Operation `n` wird zu `n'`, wobei `n' k = n >>= k` gilt. Also wenden wir `n'` auf `return` an, d.h. `n = n' return`. Das gibt uns zwei Morphismen:

```
up n     = λκ → n >>= κ
down n   = n return
```

und eine Isomorphiebedingung:

```
n κ      = n return >>= κ
```

Nun können wir das wiederum benutzen, um die Monadenoperationen herzuleiten. Für `return` und `bind` erhalten wir:

```
return a = up (return a)
         = λκ → return a >>= κ
         = λκ → κ a

m >>= k  = up (down (m>>= k))
         = up (down m >>= down ∘ k)
         = λκ → (m return >>= κ a → k a return)>>= κ
         = λκ → m return >>= κ a → k a return >>= κ
         = λκ → m (κ a → k a κ)
```

Wir wollen nun den Typ der Berechnungen bestimmen, die wir durch die beiden Abstraktionsschritte erhalten haben. Beachte, dass `(>>=)` folgenden Typ hat:

$\forall$`a: ` $\forall$`b: n a → (a → n b) → n b` ,
$\Leftrightarrow$
$\forall$`a: n a →` $\forall$`b: (a → n b) → n b` .

Wir wissen, dass wir `up n` mit `(>>=) n` vergleichen können, also hat das konsequenterweise den Typ:
$\forall$`b: (a → n b) → n b`.

Wenn wir nun noch den ersten Abstraktionsschritt in Betracht ziehen, der uns `n b = m b → m b` liefert, gelangen wir zu der folgenden Definition:

```
type CPS a ans         = (a → ans) → ans
newtype BacktrT m a    = ∀ans.CPS a (m ans → m ans)

instance Monad (BacktrT m) where
          return a     = λκ → κ a
          m >>= k      = λκ → m (λa → k a κ)
```

Wir haben nun den Vorteil, dass `BacktrT m` eine Monade ist, und das unabhängig von `m`. Die Abhängigkeiten von der Basismonade sind vollständig beseitigt. Wir wollen sehen, ob das auch für `fail` und `(|)` gilt. Es ist:

```
fail     = up fail
         = λκ → fail >>= κ
         = λκ → fail
         = λκ → id

m | n    = up (down (m | n))
         = up (down m | down n)
         = λκ → (down m | down n) >>= κ
         = λκ → down m >>= κ | down n >>= κ
         = λκ → up (down m) κ | up (down n) κ
         = λκ → m κ | n κ
         = λκ → m κ ∘ n κ.
```

Weder `fail` noch `(|)` benötigen die primitiven Operationen aus der Basismonade `m`. Mit anderen Worten haben wir unser Ziel erreicht. Wir können jeder beliebigen Monaden Backtracking hinzufügen. Hier ist die entsprechende Instanz für `MonadT`:

```
instance MonadT BacktrT where
         up m         = λκ f → m >>= λa → κ a f
         down m       = m (λa → return a) err
         err          = error "no solution"
```

Die Definition für `up` erhalten wir aus den beiden oben angegebenen `up`s. Das erhaltene Ergebnis muss zusätzlich vereinfacht werden, indem `m` immer deterministisch ist. Ähnliches gilt für `down`.

```
instance (Monad m) ⇒ Backtr (BacktrT m) where
         fail         = λκ → id
         l | r        = λκ → l κ ∘ r κ
         once r       = up (r first nothing)*
         sols r       = up (r cons nil)

first ::  (Monad m) ⇒ a → m (Maybe a) → m (Maybe a)
first a _ = return (Just a)
```

Die Definitionen für `once` und `sols` lassen sich folgendermaßen erklären: Eine Berechnung in `BacktrT m`, sagen wir `n` nimmt zwei Argumente: eine Erfolgs- und eine Fehler-Weiterführung. Wenn `n` erfolgreich ist, dann ruft es die Erfolgs-Weiterführung auf und nimmt dabei den berechneten Wert und die Fehler-Weiterführung (von `return`) mit. Wenn `n` fehlschlägt, dann ruft es einfach die Fehler-Weiterführung auf. Sowohl `once` als auch `sols` sind so implementiert, dass sie spezielle Erfolgs- und Fehler-Weiterführungen ihrer Argumente unterstützen. Zum Beispiel vernachlässigt die Erfolgs-Weiterführung `first` alle Fehler-Weiterführungen und gibt nur die erste Lösung aus. Hingegen führt `cons` die Fehler-Weiterführung aus und konstruiert die gewünschte Liste der Lösungen.

Zuletzt müssen wir noch angeben, wie die Hilfsfunktion `nothing` in * definiert ist, die wir auch im Folgenden noch benutzen werden.

```
nothing   ::  (Monad m) ⇒ m (Maybe a)
nothing   = return Nothing
```

Wir sind jetzt im Prinzip mit dem Backtracking-Monadentransformer fertig. Wir werden diese Implementierung im nächsten Abschnitt noch ein wenig modifizieren, weil wir dort `Cut` hinzufügen möchten. Nichtsdestotrotz wollen wir unser Erreichtes kurz bilanzieren: Wir haben nun die Möglichkeit, die Ausnahme-Monade mit Backtracking anzureichern.

Dazu implementieren wir eine Variante von `try`. Die Idee ist dabei, Erfolg und Mißerfolg durch Maybe und normale bzw. abnormale Termination durch Either auszudrücken:

```
instance (Exc m) → Exc (BacktrT m) where
      raise   = up ∘ raise
      try m   = λκ f → let  δ (Left e) = κ (Left e) f
                            δ (Right Nothing) = f
                            δ (Right (Just a)) = κ (Right a) f
                       in try (m first nothing) >>= δ
```

Der Aufruf `try (m first nothing)` führt die Dekodierung durch. Die Funktion δ dekodiert die Ergebnisse und ruft die zugehörigen Berechnungen auf. Um `try` wieder erfüllbar zu machen, müssen wir die Fehler-Weiterführung an δ weitergeben, welche `first` einfach weglässt. Ein geeigneter Datentyp für dieses Ziel wäre:

```
data Answer m a          = No
                         | Yes a (m (Answer m a))

answer ::(Monad m)⇒ m b → (a → m b → m b) → Answer m a → m b
answer no yes No         = no
answer no yes (Yes a m)  = yes a (m >>= answer no yes)
```

Der Konstruktor `no` korrespondiert zu `Nothing`. `Yes` erweitert `Just` um ein zusätzliches Feld für die Fehler-Weiterleitung. Beachte, dass die Fehler-Weiterleitung ein Element `Answer m a` ausgibt, d.h. `Answer` ist rekursiv definiert.

```
instance (Exc m) → Exc (BacktrT m) where
      raise   = up ∘ raise
      try m   = λκ f → let  δ (Left e) = κ (Left e) f
                            δ (Right No) = f
                            δ (Right (Yes a f'))
                              = κ (Right a) (try f' >>= δ)
                       in try (m yes no) >>= δ

no ::  (Monad m) ⇒ m (Answer m a)
no             = return No

yes ::  (Monad m) ⇒ a → m (Answer m a) → m (Answer m a)
yes a m        = return (Yes a m)
```

Der Code zeigt, dass die Dekodier-Funktion δ rekursiv durch die Fehler-Continuation `f'` angewendet wird. Das ist nötig, da `f'` möglicherweise Exceptions auswirft, die eingeschlossen werden müssen.

In [1, Theorem 3] wird gezeigt, dass, falls `m` eine Monade ist, auch `BacktrT m` eine Monade ist, die signifikante Backtracking Eigenschaften unterstützt. Wir wollen hier auf den Beweis verzichten.

4.3 Cut-Monadentransformer

Als letzten Monadentransformer möchten wir auf den Monadentransformer für `Cut` eingehen. Momentan werden die Alternativen einer Berechnung durch die

Fehler-Weiterführung repräsentiert und die Idee am `Cut` ist nun einfach die, dass wir einfach die aktuelle Fehler-Weiterführung durch eine initiale Fehler-Weiterführung ersetzen, die sozusagen einen globalen Fehler repräsentiert. Die Realisierung dieser Idee wird dadurch behindert, dass es keine einfache initiale Weiterführung gibt. Die verschiedenen Konstrukte `down`, `sols`, `once` und `try` haben alle verschiedene Initialisierungen. Aus diesem Grund müssen wir zunächst die initiale Fehler-Weiterführung standarisieren.

```
newtype CutT m a =
  ∀ ans.CPS a (m (Answer m ans) → (Answer m ans))
```

Die Funktion `no` arbeitet als Standard Fehlerweiterführung. Die `Cut`-Operationen können nun wie folgt implementiert werden:

```
!  = λκ f → κ () no
```

`Cut` ruft einfach die Erfolgs-Weiterleitung auf und ersetzt die Fehlerweiterführung durch `no`. Folgt `Cut` direkt nach `fail`, so wird `no` aufgerufen und die laufende Berechnung terminiert. Die `cut-fail` Kombination reduziert sich zu `λκ f → () no`.

Die Instanzen `Monad` und `Exc` für `CutT` sind identisch zu denen von `BacktrT`, da `CutT` eine Typspezialisierung von `BacktrT` ist. Die übrigen Operationen können wir so implementieren:

```
instance MonadT CutT where
      up m     = λκ f → m >>= λa !  κ a f
      down m   = m yes no >>= answer err (λa → return a)

instance (Monad m) → Backtr (CutT m) where
      fail     = λκ → id
      m | n    = λκ → m κ∘ n κ
      once m   = up (m yes no >>= answer nothing first)
      sols m   = up (m yes no >>= answer nil cons)
```

Die Ausdrücke der Form `m κ f`, die in der BacktrT-Instanz auftreten, wurden systematisch durch `m yes no >>= answer f κ` ersetzt. Das nutzen wir auch bei der Implementierung von call:

```
instance (Monad m) ⇒ Cut (CutT m) where
!  = λκ f → κ () no
call m = λκ f → m yes no >>= answer f κ
```

Wir können zeigen, dass `CutT` sinnvollen Bedingungen an den Prolog-`Cut` genügt [1, Theorem 4]. Damit haben wir unser Ziel erreicht. Wir setzen

```
type Prolog = CutT (ExcT IO))
```

und können dies als Standardmodel für unser eingebettetes Prolog sehen. Die Monadentransformer `BacktrT` und `CutT` sind indes unabhängig vom Aufruf der Evaluation und effizient. In [1, Abschnitt 6.6] werden Benchmarkergebnisse von einem typischen Problem mit Backtracking aufgelistet, aus denen hervorgeht, dass eine effiziente Haskell-Implementierung von Prolog, zumindest für das angegebene Beispiel, sogar schneller arbeitet als Prolog selber. Außerdem erlaubt `CutT` eine einfache Implementierung von `cut` und beide Monadentransformer können Backtracking zu einer beliebigen Monade hinzufügen.

5 Logische Variablen

In diesem Abschnitt möchten wir, wie bereits am Anfang dieser Seminarausarbeitung angedeutet, kurz auf die Implementierung von logischen Variablen eingehen. Wir beschränken uns auf drei Möglichkeiten, einfache logische Sprachen in Haskell zu implementieren.

5.1 Prolog-Einbettung mit Prädikaten

Wir gehen zunächst auf die vielleicht intuitivste Einbettung ein. Sie beinhaltet einen Datentyp `Term` für Terme, `Pred` für Prädikate, ein Unifikationsprädikat `(≐)`, sowie `(∨)`, `(∧)` und `(∃)`:

```
(≐) ::  Term → Term → Pred
(∧),(∨)::  Pred → Pred → Pred
(∃) ::  (Term → Pred) → Pred
```

An Term müssen gewisse Voraussetzungen gestellt werden, die uns hier nicht weiter interessieren sollen, da wir uns im Folgenden auf den Typ für Binärbäume beschränken möchten. Dabei ist `Var i` eine Referenz zu einer Variablen mit dem eindeutigen Identifikator `i`:

```
data Term = Var Id | Atom String | Nil | Term :::  Term
```

Im Gegensatz zu unserer Implementierung könnte man Backtracking auch einfach als Listen implementieren, was aber im Gegensatz zu unserer Implementierung in Abschnitt 3 einige Nachteile hat. Dennoch soll uns das im Moment nicht weiter stören. Weiterhin ist anzumerken, dass wir uns nicht auf die Standard-Monaden stützen, sondern Monaden mit einer Null (`mzero`) und einem Plus (`+++`). Wir definieren uns die folgenden Typen

```
type Backtr a = [a]
type Pred = State → Backtr State
type State = (Subst, [Term])
type Subst = [(Id, Term)]
```

Der Typ `Pred` für Prädikate ist eine Funktion, die einen Berechnungszustand nimmt und uns neue Zustände liefert. Ein Berechnungszustand State beinhaltet die aktuellen Werte der logischen Variablen (genannt Substitutionen) und eine Kette von uninstanziierten Variablen.

Das Prädikat für die Unifikation (`≐`) nutzt die (Standard-) Unifikationsfunktion `unify :: Term → Term → Subst → Backtr Subst`, welche zwei Terme und eine Substitution nimmt und entweder Fehler oder eine neue Substitution liefert. Wir implementieren dann folgendes:

```
(≐) ::  Term → Term → Pred
a ≐ b = λ(sub,vs) → do  sub' <- unify a b sub
                        return (sub',vs)
```

(`∧`) mappt das zweite Prädikat auf das Ergebnis des ersten Prädikats und konkateniert dabei die resultierende Liste von Listen. Hingegen konkateniert (`∨`) lediglich die Ergebnisse.

```
(∧),(∨) ::  Pred → Pred → Pred
p ∧ q = λst → p st >>= q
p ∨ q = λst → p st +++ q st
```

Zuletzt geben wir noch den Quantifizierer (∃) an, der einer gegebenen Funktion eine unverbrauchte Variable aus dem Zustand liefert. Die Definition der Erfolgs- und Mißerfolgsprädikate ist dann einfach.

```
(∃) ::  (Term → Pred) → Pred
∃p = λ(sub,v:vs) → p v (sub,vs)

true, false ::  Pred
true = λst → return st
false = λst → mzero
```

Auf diese Art und Weise haben wir das Verhalten von der Standardtiefensuche implementiert. Es ist auch möglich, andere Suchstrategien umzusetzen. Dazu müssen dann Pred, ∧ und ∨ entsprechend angepasst werden.

Zur Veranschaulichung möchten wir nun ein Beispiel angeben. Wir wollen einen Pfad ausgeben, der einen Graphen durchläuft, ohne Knoten doppelt zu besuchen. In Prolog würde man so vorgehen: Für die (gerichteten) Kanten von einem Knoten zum Nächsten definieren wir das Prädikat `edge(X,Y)`. Nun können wir unseren Pfad und ein paar Knoten definieren:

```
path(X,X,[X]).
path(X,Z,X:Nodes) :- edge(X,Y), path(Y,Z,Nodes).

edge(a,b).  edge(a,d).  edge(b,c).  edge(b,d).  edge(c,d).
edge(c,e).  edge(d,e).
```

Das Gleiche können wir in Haskell erledigen, was allerdings noch nicht ganz so elegant aussieht. Dennoch folgt analog:

```
path x z nodes =
          x ≐ z ∧ nodes ≐ x :::  Nil
          ∨ ∃ (λ nodes' → nodes ≐ x :::  nodes'
          ∧ ∃ (λy → edge x y ∧ path y z nodes' ) )

edge ::  Term → Term → Pred
edge x y =  (x ≐ Atom "a" ∧ y ≐ Atom "b")
          ∨ (x ≐ Atom "a" ∧ y ≐ Atom "d")
          ∨ (x ≐ Atom "b" ∧ y ≐ Atom "c")
          ∨ (x ≐ Atom "b" ∧ y ≐ Atom "d")
          ∨ (x ≐ Atom "c" ∧ y ≐ Atom "d")
          ∨ (x ≐ Atom "c" ∧ y ≐ Atom "e")
          ∨ (x ≐ Atom "d" ∧ y ≐ Atom "e")
```

5.2 Prolog-Einbettung mit Monaden

Alles, was wir soeben implementiert haben, können wir auch monadisch implementieren. Genauer gesagt ist der Typ von `Pred` isomorph zu einer Instanz der Backtracking Status Monade BS. Wir redefinieren unsere Funktionen folgendermaßen:

```
newtype BS a = BS (State → Backtr (State,a))
type Pred    = BS ()
```

Die Monade hat die folgenden Definitionen für `bind`, `return`, `zero` und `+++`, die alle auf die Backtracking-Monade und nicht auf die BS-Monade appliziert werden.

```
instance Monad BS where
   return a      = BS (λst → return (st,a))
   BS f >>= k    = BS (λst → do (st',a) f st
                                let BS g = k a in g st')

instance MonadPlus BS where
   mzero         = BS (λst → mzero)
   BS f +++ BS g = BS (λst → f st +++ g st)
```

Nach einigen Überlegungen können wir sehen, dass `(∧)`, `(∨)`, `true` und `false` zu `(>>)`, `(+++)`, `return()` und `mzero` korrespondieren. Die Unifikation `(≐)` bleibt wie zuvor und der Existenzquantor kann einfach definiert werden, indem die Hilfsfunktion `free` verwendet wird, die eine neue, ungebundene Variable berechnet.

```
(≐) ::  Term → Term → Pred
a ≐ b   = λ(sub,vs) → do sub unify a b sub
                         return ((sub,vs),())

free ::  BS Term
free    = λ(sub,v:vs) → return ((sub,vs),v)

(∃) ::  (Term → BS a) → BS a
∃p      = do a free ; p a
```

Wir können nun unser Beispiel anpassen und erhalten so eine Form, die schon eher monadisch ist:

```
path ::  Term → Term → Term → Pred
path x z nodes = do x ≐ z
                    nodes ≐ x :::  Nil
                ∨ do nodes' ← free
                    nodes ≐ x :::  nodes'
                    y ← free
                    edge x y
                    path y z nodes'
```

Mit einen kleinen Interpreter `solve` wird nun das Ergebnis ähnlich wie in Prolog ausgegeben.

```
Main> solve (path (Atom "a") (Atom "e") (Var "Xs"))
      yes, Xs=a :::  b :::  c :::  d :::  e :::  Nil
      yes, Xs=a :::  b :::  c :::  e :::  Nil
      yes, Xs=a :::  b :::  d :::  e :::  Nil
      yes, Xs=a :::  d :::  e :::  Nil
      no (more) solutions
```

5.3 Funktional logische Prolog-Einbettung

Da Monaden ohnehin immer ein Ergebnis ausgeben, können wir das einfach nutzen, um unsere Implementierung noch ein wenig zu vereinfachen.

```
path ::  Term → Term → BS Term
path x z nodes = do x ≐ z
                    return (x :::  Nil)
                ∨ do y ← neighbour x
                    nodes ← path y z
                    return (x::nodes)
```

Diese Definition benutzt das Prädikat `neighbor`, welches die Nachbarn des Arguments ausgibt:

```
neighbour ::  Term → BS Term
neighbour x = do y ← free ; edge x y ; return y

Main> solve (path (Atom "a") (Atom "e"))
      a :::  b :::  c :::  d :::  e :::  Nil
      a :::  b :::  c :::  e :::  Nil
      a :::  b :::  d :::  e :::  Nil
      a :::  d :::  e :::  Nil
      no (more) solutions
```

5.4 Ausblick

Wir können selbstverständlich auch den Prädikaten-, funktional logischen und monadischen Stil untereinander mischen. Allerdings haben wir immer noch nicht die volle Funkitonalität der logischen Programmierung erreicht. Zum Beispiel benötigen wir Unifikation, was wir nur in sehr abgespeckter Version erreichen könnten. Dennoch möchten wir diesen Ausflug zu den logischen Variablen hiermit abschließen.

In [3, Abschnitt 4] wird die soeben angefangene Implementierung noch weiter ausgeführt. Dort wird erreicht, dass man zu der Implementierung auch Haskells starke Typisierung zu der Einbettung hinzufügen kann, anstatt weiterhin auf den Typ `Term` zurückgreifen zu müssen. Dazu wird eine neue Monade LP (für logische Programmierung) eingeführt, die auf die State-Monade aufbaut, die in Erweiterungen von Haskell und GHC vorimplementiert ist.

```
type LP s = BacktrT (ST s)
```

Hier wird dann der Typ `Maybe` verwendet, um zu veranschaulichen, dass eine logische Variable entweder uninstanziiert ist, oder den Wert eines bestimmten Typs hat.

```
type Var s a = LPRef s (Maybe a)
```

Nun könnten beliebige Typen definieret werden, die dann anstelle von Term genutzt werden können.

Allerdings ist dabei zu beachten, dass verschiedene Konstruktoren für die verschiedenen Typen der Variablen benötigt werden. Aus diesem Grund müsste `free` abgeändert werden. Wie das gemacht wird, ist dort umfangreich erklärt und ebenso, wie man `unify` anpassen muss, damit wir dort keine Probleme bekommen. Mit ein paar Anpassungen kann man dann erreichen, dass Haskells starke Typisierung auch in der Einbettung von Prolog noch brauchbar ist.

6 Zusammenfassung

Unser Ziel war es, zwei interessante Aspekte von Prolog auch in Haskell einsetzen zu können. Zunächst haben wir uns für die Erweiterung von Monaden mit Prolog-Funktionen interessiert.

Für die Implementierung von Backtracking haben wir es geschafft, durch durchgängige Anwendung von Monaden und Monadentransformer, eine sinnvoll strukturierte Version zu kreieren, die wir eingebettetes Prolog genannt haben. Unser Vorgehen war dabei so, dass wir von einer einfachen Monade ausgegangen sind (`Exc`) und zu dieser zunächst Backtracking mittels des Backtracking-Monadentransformers `BacktrT` hinzugefügt haben. Abschließend haben wir diese neue Monade noch mit Cut angereichert.

Dieses Thema wurde in dieser Ausarbeitung nicht vollständig abgearbeitet. So wurde zum Beispiel nicht gezeigt, dass die Implementierung der Monadentransformer auch wirklich korrekte Lösungen bietet. Der interessierte Leser kann dies aber in [1] in aller Ausführlichkeit nachlesen. Weiterhin sind dort weitere Beispiele zu finden, die die Anwendung von den Prolog-Funktionen veranschaulichen.

Was hier außerdem nicht angesprochen wurde, ist die tatsächliche Implementierung der in Abschnitt 4 vorgestellten Monadentransformer. Weiteres Material dazu liefert die nicht ganz einfache Arbeit[2], in dem systematisch die Monadentransformer hergeleitet werden.

Das zweite Thema dieser Ausarbeitung widmete sich den logischen Variablen. In diesem Bereich haben wir nur eine kurze Einleitung gegeben und drei mögliche Stilrichtungen für die Implementierung von logischen Variablen gegeben. In dem Ausblick haben wir angedeutet, wie wir auch eine Implementierung einer logischen Sprache in Haskell erreichen können, ohne auf die starke Typisierung von Haskell verzichten zu müssen.

Wer sehen möchte, wie das gemacht wird, kann sich den Ausführungen in [3] zuwenden. Allerdings ist darauf hinzuweisen, dass es relativ aufwendig ist, kompliziertere Datentypen zu der dort angegebenen Einbettung hinzuzufügen.

Insgesamt muss man vielleicht noch darauf hinweisen, dass wir jetzt mit unseren paar Funktionen nicht komplett auf Prolog verzichten können. Es gibt einige sinnvolle Erweiterungen, auf die wir überhaupt nicht eingegangen sind.