4 Logisch-funktionale Programmierung

4.1 Motivation

Prinzipiell sind funktionale Sprachen berechnungsuniversell. Dies bedeutet, dass mit ihnen alles programmierbar ist, was man auch in imperativen Sprachen programmieren kann, und im Prinzip keine Erweiterungen notwendig sind.

Auf der anderen Seite ist jedoch der Programmierstil und die Lesbarkeit von Programmen sehr wichtig. Daher sind Erweiterungen wünschenswert, falls diese zu besser strukturierten oder lesbaren Programmen führen.

Aus diesem Grund betrachten wir noch einmal die wesentliche *Charakteristik funktionaler Sprachen*:

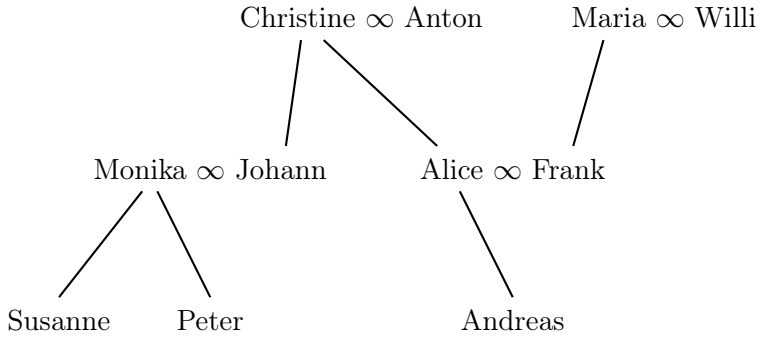
> Variablen stehen für unbekannte Werte, die in Regeln, nicht jedoch in zu berechnenden Ausdrücken vorkommen, d.h. die zu berechnenden Ausdrücke sind immer *Grundterme.*

Eine Lockerung dieser Einschränkung ist durchaus wünschenswert bei *Datenbankanwendungen*, wie das folgende Beispiel zeigt.

Beispiel: Betrachte eine Verwandtschaft, d.h. Personen und Beziehungen zwischen diesen. In der folgenden graphischen Darstellung einer Beispielverwandschaft drücken wir die folgenden Beziehungen aus:

∞ : verheiratet

/ : Mutter-Kind-Beziehung



Das Ziel ist es nun, aus diesen Basisbeziehungen abgeleitete Beziehungen, wie z.B. Tante oder Großvater, allgemein zu definieren.

Zunächst versuchen wir dies in Haskell zu modellieren. Zu diesem Zweck definieren wir zunächst den Datentyp von Personen (dies könnte auch String o.ä. sein):

```
data Person = Christine | Anton | Maria | Willi | Monika | Johann
            | Alice | Frank | Susanne | Peter | Andreas
  deriving (Eq,Show)
```

Die „verheiratet“-Beziehung modellieren wir als Funktion `ehemann`:

```
ehemann :: Person  → Person
ehemann   Christine = Anton
ehemann   Maria     = Willi
ehemann   Monika    = Johann
ehemann   Alice     = Frank
```

Die „Mutter-Kind“-Beziehung modellieren wir als Funktion `mutter`:

```
mutter :: Person  → Person
mutter   Johann  = Christine
mutter   Alice   = Christine
mutter   Frank   = Maria
mutter   Susanne = Monika
mutter   Peter   = Monika
mutter   Andreas = Alice
```

Aus diesen grundlegenden Beziehungen können wir weitere Beziehungen ableiten, wie z.B.:
Der Vater ist der Ehemann der Mutter (idealerweise!):

```
vater:: Person  → Person
vater kind = ehemann (mutter kind)
```

Allerdings ist die Großvater-Enkel-Beziehung im Allgemeinen eine *Relation*, die so modelliert werden könnte:

```
grossvater :: Person  → Person  → Bool
grossvater g e | g == vater (vater e)  = True
grossvater g e | g == vater (mutter e) = True
```

Hiermit sind z.B. folgende Berechnungen möglich:

- Wer ist der Vater von Peter?
  `vater Peter` ⇝ `Johann`
- Ist Anton Großvater von Andreas?
  `grossvater Anton Andreas` ⇝ `True`

Leider sind aber keine Berechnungen möglich, bei denen wir andere Werte wissen wollen,

wie z.B.:

1. Welche Kinder hat Johann?
2. Welche Großväter hat Andreas?
3. Welche Enkel hat Anton?

Im Prinzip wäre es möglich, diese Fragen auszudrücken, falls *Variablen in zu berechnenden Ausdrücken* zulässig wären:

1. `vater k == Johann` $\leadsto$ `k = Susanne` oder `k = Peter`
2. `grossvater g Andreas` $\leadsto$ `g = Anton` oder `g = Willi`
3. `grossvater Anton e` $\leadsto$ `e = Susanne` oder `e = Peter` oder `e = Andreas`

Die Idee ist also:

> Falls Variablen in Ausdrücken vorkommen, dann suche nach passenden Werten für diese Variablen, sodass die Berechnung des Ausdrucks möglich ist. Da solche Variablen initial keinen Wert oder Bindung haben, werden diese auch als **freie Variablen** bezeichnet.

Zu beachten ist, dass freie Variablen nicht in den zu berechnenden Ausdrücken funktionaler Sprachen erlaubt sind. In einer funktionalen Sprache müssten wir die Programme zur Erreichung der gleichen Funktionalität erweitern, in dem wir z.B. die Umkehrfunktion/-relation explizit programmieren (z.B. eine Funktion, die zu einem Vater die Menge seiner Kinder berechnet).

In logischen Programmiersprachen, wie z.B. Prolog, ist dagegen die Verwendung freier Variablen in Ausdrücken erlaubt (daher werden diese dort auch als „logische Variablen“ bezeichnet). Diese Sprachen basieren auf der Idee, dass *Lösungen* berechnet werden, d.h. Belegungen der Variablen, sodass der Ausdruck reduzierbar ist. Das Hauptproblem dabei ist natürlich, wie man Lösungen konstruktiv findet. Dies werden wir aber erst später erläutern.

Wenn man funktionale Sprachen so erweitert, dass diese auch Aspekte der logischen Programmiersprachen abdecken, dann spricht man auch von **logisch-funktionalen Programmiersprachen**. Als Beispiel für eine logisch-funktionale Sprachen betrachten wir hier die Sprache Curry[1], deren Syntax sehr ähnlich zu Haskell ist. Wir können also das obige Programm für unser Verwandtschaftsbeispiel nehmen und einfach als Curry-Programm interpretieren.[2] Dann können wir die obigen Ausdrücke mit Variablen wie folgt eingeben (man beachte, dass freie Variablen in initialen Ausdrücken explizit durch “`where...free`” deklariert werden müssen, um Tippfehler zu erkennen):

[1] `http://www.curry-lang.org`
[2] Dazu muss man nur die Dateiendung “`.hs`” gegen “`.curry`” austauschen.

```
Family> grossvater g Andreas  where g free
{g=Willi} True
{g=Anton} True
```

Falls freie Variablen in initialen Ausdrücken vorkommen, so sollen Werte für diese Variablen berechnet werden, mit denen dieser Ausdruck ausrechenbar ist. Diese Variablenwerte, auch **Bindungen** genannt, werden in geschweiften Klammern vor dem berechneten Wert ausgegeben. Da es im obigen Fall zwei Werte für die Variable `g` gibt, mit denen der Ausdruck zu `True` ausrechenbar ist, werden auch zwei Bindung/Wert-Paare als Ergebnis ausgegeben.

Natürlich kann es auch mehr als zwei Ergebnisse geben, wie die folgenden Beispiele zeigen:

```
Family> grossvater Anton e    where e free
{e=Susanne} True
{e=Peter} True
{e=Andreas} True

Family> vater k == Johann     where k free
{k=Johann} False
{k=Alice} False
{k=Frank} False
{k=Susanne} True
{k=Peter} True
{k=Andreas} False
```

Das letzte Beispiel zeigt, dass tatsächlich alle möglichen Lösungen, d.h. Variablenbindungen berechnet werden, wobei typischerweise die Lösungen mit dem Ergebnis `False` nicht interessieren. Wir können solche Ergebnisse vermeiden, indem wir diese mit der Operation `solve` „filtern“:

```
Family> solve $ vater k == Johann   where k free
{k=Susanne} True
{k=Peter} True
```

Die Operation `solve` ist wie folgt vordefiniert:

```
solve True = True
```

Somit liefert `solve` *kein* Ergebnis, falls das Argument nicht `True` ist, wodurch also die obigen `False`-Werte nicht ausgegeben werden.

Schließlich können wir auch mehrere freie Variablen benutzen, um die gesamte Großvater/Enkel-Relation zu berechnen:

```
Family> grossvater g e  where g,e free
{g=Anton, e=Susanne} True
```

```
{g=Anton, e=Peter} True
{g=Willi, e=Andreas} True
{g=Anton, e=Andreas} True
```

An diesem Beispiel haben wir schon einige wesentliche Aspekte logisch-funktionaler Sprachen kennengelernt:

Variablen in initialen Ausdrücken: Diese werden mit "`free`" deklariert und sind existenzquantifiziert, d.h. der initiale Ausdruck

```
grossvater Anton e  where e free
```

bedeutet:

```
∃e: grossvater Anton e
```

Dagegen sind Variablen in Regeln allquantifiziert, d.h.

```
vater k = ehemann (mutter k)
```

bedeutet

```
∀k: vater k = ehemann (mutter k)
```

denn hier steht `k` für *jeden* beliebigen Wert (des korrekten Typs).

Variablen in Regeln: Regeln in Curry können wie in Haskell auch Bedingungen enthalten, sodass sie die folgende Form haben:

```
l | c = r
```

In funktionalen Sprachen wird üblicherweise gefordert, dass alle Variablen, die in c oder r vorkommen, auch in l vorkommen müssen. Dagegen ist es in logisch-funktionalen Sprachen erlaubt, dass, ähnlich wie in initialen Ausdrücken, hier auch **Extravariablen** vorkommen dürfen, d.h. Variablen in c oder r, die nicht in l vorkommen. Wir zeigen hierzu einige Beispiele, wann diese sinnvoll eingesetzt werden können.[3]

- Das letzte Element einer Liste kann mit folgender Funktion berechnet werden:

```
last xs | ys ++ [e] == xs = e
  where ys,e free
```

Hierbei sind `ys` und `e` Extravariablen, die explizit deklariert werden müssen. Falls eine Extravariable nur einmal vorkommt, wie `ys`, dann muss man sie

[3] Wir lassen in den folgenden Beispielen die Typangaben weg, da diese wegen der logischen Anteile, insbesondere der logischen Variablen besondere Typkontexte enthalten, die wir erst später erläutern.

eigentlich nicht explizit benennen. In diesem Fall können wir statt `ys` auch eine anonyme Variable durch einen Unterstrich und ohne Deklaration einführen:

```
last xs | _ ++ [e] == xs = e where e free
```

Man beachte, dass durch Wiederverwendung der Listenkonkatenation eine einfache Definition der Operation `last` ermöglicht wird. In Kapitel 4.7.3 werden wir eine noch kompaktere Definition dafür sehen.

- Wir können mit folgender Operation prüfen, ob ein Element in einer Liste enthalten ist:

```
member e xs | _ ++ (e : _) == xs  = True
```

- Ist eine Liste Teil einer anderen Liste?

```
sublist s xs | _ ++ s ++ _ == xs  = True
```

Intuitiv kann die Bedeutung von Extravariablen in Regeln wie folgt erklärt werden:

- Extravariablen sind existenzquantifiziert.
- Finde eine Belegung für diese, sodass die Bedingung beweisbar ist.

Fehlschlagende Berechnungen: Man beachte, dass `member` und `sublist` ähnlich wie in der Logikprogrammierung im positiven Fall nur den Wert `True` zurückliefern, aber im negativen Fall kein Ergebnis liefern, sondern einfach nur fehlschlagen. Ebenso liefert "`solve False`" kein Ergebnis. In rein funktionalen Sprachen würde dies als Fehler ausgegeben, aber in logischen Sprachen ist ein Fehlschlag kein Problem, weil es häufig Berechnungsalternativen gibt. Dies liegt daran, dass logische Berechnungen typischerweise nichtdeterministisch sind. Da man manchmal auch explizit ausdrücken möchte, dass ein Ausdruck keinen Wert hat, gibt es die vordefinierte Operation

```
failed  :: a
```

deren Berechnung kein Ergebnis liefert, sondern fehlschlägt.

Nichtdeterministische Berechnungen: Auch bei konfluenten Programmen sind Berechnungen eventuell nichtdeterministisch, da im Allgemeinen mehr als eine Lösung existiert. Z.B. liefert der Ausdruck

```
member x [1,2,3] where x free
```

die Lösungen

```
{x=1} True
{x=2} True
{x=3} True
```

Falls Extravariablen in Regeln vorkommen, können auch Ausdrücke ohne freie Variablen mehrere Werte liefern. Wenn wir z.B. die Operation

```
someElem xs | member x xs = x  where x free
```

definieren, dann erhalten wir z.B. folgende Ergebnisse:

```
> someElem [1,2,3]
1
2
3
```

Nichtdeterministische Operationen: Die Operation `someElem` zeigt, dass logisch-funktionale Sprachen die Definition **nichtdeterministischer Operationen** erlauben, d.h. Operationen, die zu mehr als einem Wert auswerten (auch wenn diese auf einem festen Argument angewendet werden). Dies kann in vielen Anwendungen sinnvoll sein, wie wir noch sehen werden. Der Prototyp einer nichtdeterministischen Operation ist der Infixoperator "`?`", der eines seiner Argumente zurück gibt. Dieser ist in Curry durch die Regeln

```
(?) :: a → a → a
x ? y = x
x ? y = y
```

vordefiniert. Falls wir also eine Operation `coin` durch

```
coin :: Int
coin = 0 ? 1
```

definieren, wird der Ausdruck `coin` zu 0 oder 1 ausgewertet.

Flexiblere Programmierung: Die Umkehrfunktion/-relation zu einer definierten Funktion steht durch die logischen Anteile automatisch zur Verfügung. Wenn wir z.B. eine Funktion

```
f x = ...
```

definieren, dann können wir diese z.B. in einer Bedingung

```
... | f y == e = ...
```

benutzen und erhalten dadurch einen Wert v für die Variable `y`, sodass der Ausdruck `f` v zu (einem Wert von) e auswertet.

Automatische Lösungssuche: Die Suche nach Lösungen muss nicht explizit programmiert werden, sondern das System sucht automatisch nach passenden Lösungen.

Wie diese Lösungen gefunden werden, muss nicht unbedingt festgelegt werden. So können zur Suche Backtracking-Strategien verwendet werden, aber Breitensuche und parallele Suchstrategien sind auch anwendbar.

In logischen Programmiersprachen, wie z.B. Prolog, sind diese Programmiertechniken ebenfalls möglich. Allerdings ist in reinen Logiksprachen die Syntax eingeschränkter, denn dort ist nur die Definition positiver Relationen erlaubt, d.h. die rechte Seite ist immer `True`, sodass jede Gleichung die Form

l `|` c_1 `&&` ... `&&` c_n `=` `True`

hat, wobei $c_1, \ldots, c_n$ Literale sind (Anwendung eines Prädikats auf Argumente).

Da nur solche Definitionen erlaubt sind, gibt es hierfür in Prolog eine spezielle Schreibweise:

l `:-` c_1`,...,`c_n`.`

Somit können Logiksprachen als Spezialfall logisch-funktionaler Sprachen angesehen werden.

Bevor wir uns genauer mit den Auswertungsstrategien logisch-funktionaler Sprachen beschäftigen, zeigen wir noch einige Beispiele zur Programmierung.

Beispiel: Spiel 24

Logisch-funktionale Sprachen sind gut geeignet zum Lösen von Suchproblemen, falls man also keinen direkten Algorithmus hat, um die Lösung eines Problems zu berechnen. Dies können wir gut an der Lösung folgender Aufgabe sehen:

> Bilde arithmetische Ausdrücke, die genau die Zahlen $2, 3, 6, 8$ enthalten, sodass das Ergebnis der Wert 24 ist (wobei eine Division nur zulässig ist, wenn das Ergebnis ganzzahlig ist).

Beispiele für solche Ausdrücke sind: $(3 * 6 - 2) + 8$, $3 * 6 + 8 - 2$,...

Lösungsidee:

1. Bilde Pemutationen der Liste `[2,3,6,8]`
2. Erzeuge arithmetische Ausdrücke über diesen Permutationen
3. Prüfe, ob der Wert 24 ist

Die Berechnung von Permutationen einer Zahlenliste kann wie folgt realisiert werden:

```
permute :: [Int]  → [Int]
permute []     = []
permute (x:xs) | us++vs == permute xs
               = us++[x]++vs
  where us, vs free
```

Hierbei sollte beachtet werden, dass `permute` *eine* beliebige Permutation liefert, es aber nicht festgelegt ist, welche diese ist. Z.B. wertet `permute [1,2,3]` zu den Werten

```
[1,2,3]
[1,3,2]
[2,1,3]
[2,3,1]
[3,1,2]
[3,2,1]
```

aus. `permute` ist somit eine nichtdeterministische Operation.

Zur Repräsentation von Ausdrücken definieren wir einen Datentyp:

```
data Exp = Num Int      -- Zahl
         | Add Exp Exp
         | Mul Exp Exp
         | Sub Exp Exp
         | Div Exp Exp
```

Nun definieren wir eine Operation, die für einen Ausdruck testet, ob dieser nur Ziffern aus einer gegebenen Zahlenliste enthält, und dabei den Wert des Ausdrucks berechnet:

```
test :: Exp  →  [Int]  →  Int
test (Num y)   [z] | y == z       = y
test (Add x y) zs  | split u v zs = test x u + test y v   where u, v free
test (Sub x y) zs  | split u v zs = test x u - test y v   where u, v free
test (Mul x y) zs  | split u v zs = test x u * test y v   where u, v free
test (Div x y) zs  | split u v zs = opdiv (test x u) (test y v)
  where
    u, v free
    opdiv a b = if b == 0 || a `mod` b /= 0 then failed else a `div` b
```

Hierbei ist `split` ein Prädikat, das erfüllt ist, wenn die ersten beiden Zahlenlisten nichtleer sind und zusammen konkateniert gleich der dritten Liste sind:

```
split :: [Int]  →  [Int]  →  [Int]  →  Bool
split (u:us) (v:vs) xs | (u:us)++(v:vs) == xs = True
```

Nun können wir sehr einfach durch positive Ergebnisse einer Gleichheitsbedingung Lösungen für unser Problem berechnen:

```
solve $ test e (permute [2,3,6,8]) == 24 where e free
```

Damit erhalten wir folgende Lösungen:

```
{e = Add (Sub (Mul (Num 3) (Num 6)) (Num 2)) (Num 8)} True
{e = Add (Mul (Num 3) (Num 6)) (Sub (Num 8) (Num 2))} True
{e = Add (Sub (Mul (Num 6) (Num 3)) (Num 2)) (Num 8)} True
{e = Add (Num 6) (Mul (Num 3) (Sub (Num 8) (Num 2)))} True
{e = Add (Mul (Num 6) (Num 3)) (Sub (Num 8) (Num 2))} True
{e = Add (Num 6) (Mul (Sub (Num 8) (Num 2)) (Num 3))} True
{e = Add (Mul (Num 3) (Sub (Num 8) (Num 2))) (Num 6)} True
{e = Add (Sub (Num 8) (Num 2)) (Mul (Num 3) (Num 6))} True
{e = Add (Mul (Sub (Num 8) (Num 2)) (Num 3)) (Num 6)} True
{e = Add (Num 8) (Sub (Mul (Num 3) (Num 6)) (Num 2))} True
{e = Add (Sub (Num 8) (Num 2)) (Mul (Num 6) (Num 3))} True
{e = Add (Num 8) (Sub (Mul (Num 6) (Num 3)) (Num 2))} True
{e = Sub (Mul (Num 3) (Num 6)) (Sub (Num 2) (Num 8))} True
{e = Sub (Add (Mul (Num 3) (Num 6)) (Num 8)) (Num 2)} True
{e = Sub (Num 6) (Mul (Num 3) (Sub (Num 2) (Num 8)))} True
{e = Sub (Mul (Num 6) (Num 3)) (Sub (Num 2) (Num 8))} True
{e = Sub (Add (Mul (Num 6) (Num 3)) (Num 8)) (Num 2)} True
{e = Sub (Num 6) (Mul (Sub (Num 2) (Num 8)) (Num 3))} True
{e = Sub (Mul (Num 3) (Add (Num 2) (Num 8))) (Num 6)} True
{e = Sub (Mul (Num 3) (Add (Num 8) (Num 2))) (Num 6)} True
{e = Sub (Mul (Add (Num 2) (Num 8)) (Num 3)) (Num 6)} True
{e = Sub (Num 8) (Sub (Num 2) (Mul (Num 3) (Num 6)))} True
```

```
{e = Sub (Mul (Add (Num 8) (Num 2)) (Num 3)) (Num 6)} True
{e = Sub (Add (Num 8) (Mul (Num 3) (Num 6))) (Num 2)} True
{e = Sub (Num 8) (Sub (Num 2) (Mul (Num 6) (Num 3)))} True
{e = Sub (Add (Num 8) (Mul (Num 6) (Num 3))) (Num 2)} True
```

Diese könnte man dann noch natürlich noch in einem schöneren Format ausdrucken.

Es bleibt noch anzumerken, dass in Curry ähnlich wie in Prolog (und im Gegensatz zu Haskell) in Mustern Variablen auch mehrfach vorkommen dürfen, die dann als Gleichheitsbedingung zwischen den verschiedenen Vorkommen interpretiert werden. Somit können wir die erste Regel von `test`

```
test (Num y) [z] | y == z = y
```

auch kompakter in der folgenden Form aufschreiben:

```
test (Num x) [x] = x
```

Reguläre Ausdrücke

In diesem Beispiel wollen wir reguläre Ausdrücke zum Pattern Matching verwenden. Hierzu definieren wir zunächst einen Datentyp zur Darstellung regulärer Ausdrücke über einem Alphabet `a`:

```
data RE a = Lit a
          | Alt  (RE a) (RE a)
          | Conc (RE a) (RE a)
          | Star (RE a)
```

Als Beispiel können wir z.B. den Ausdruck $(a|b|c)$ definieren:

```
abc = Alt (Alt (Lit 'a') (Lit 'b')) (Lit 'c')
```

Ein weiteres Beispiel ist der Ausdruck (ab^*):

```
abstar = Conc (Lit 'a') (Star (Lit 'b'))
```

Wir können aber auch einfach die Sprache der regulären Ausdrücke um weitere Konstrukte erweitern, wie z.B. einen `plus`-Operator, der eine mindestens einmalige Wiederholung bezeichnet:

```
plus :: RE a  → RE a
plus re = Conc re (Star re)
```

Die Semantik regulärer Ausdrücke kann direkt durch eine Semantik-Funktion definiert werden, wie man dies auch in der Theorie der regulären Ausdrücke macht. Somit wird durch die Semantik jedem regulären Ausdruck ein dadurch beschriebenes Wort zugeordnet. Da es mehrere mögliche Worte geben kann, beschreiben wir dies durch eine nichtdeterministische Operation.

```
sem :: RE a  → [a]
sem (Lit c)    = [c]
sem (Alt  a b) = sem a ? sem b
sem (Conc a b) = sem a ++ sem b
sem (Star a)   = [] ? sem (Conc a (Star a))
```

Beispielauswertungen:

`sem abc` $\leadsto$ `a` oder `b` oder `c`

Wenn wir einen String gegen einen regulären Ausdruck matchen wollen, können wir dies einfach durch folgendes Prädikat beschreiben:

```
match :: RE Char  → String  → Bool
match r s | sem r == s = True
```

Ein Prädikat, das ähnlich wie das Kommando `grep` von Unix feststellt, ob ein regulärer Ausdruck irgendwo in einem String enthalten ist, können wir wie folgt definieren:

```
grep :: RE Char  → String  → Bool
grep r s | _ ++ sem r ++ _ == s = True
```

Beispielauswertungen:

```
grep abstar "dabe"  ⇝ True
```

Beachte, dass der letzte Ausdruck eine endliche Auswertung hat, obwohl es unendlich viele Wörter gibt, zu denen `abstar` auswerten kann.

Wir können durch eine einfache Modifikation auch die Position ausgeben, bei der der enthaltene Ausdruck beginnt:

```
grepPos :: RE Char  → String  → Int
grepPos r s | xs ++ sem r ++ _ == s
            = length xs
 where xs free
```