4.7.2 Eingekapselte Suche

Wenn mittels logischer Programmiertechniken mehrere Werte oder Lösungen für ein Prädikat berechnet werden, ist es oft wünschenswert, diese Werte oder Lösungen zu vergleichen oder einer weiteren Verarbeitung (z.B. Ausgabe in einer Datei) zuzuführen. Hierzu müsste man die verschiedenen Werte in eine gemeinsame Datenstruktur (z.B. eine Liste) bringen.

Mit den bisherigen Sprachmitteln ist dies nicht möglich, da die verschiedenen Werte auf unterschiedlichen Lösungswegen berechnet werden, die zunächst einmal nichts miteinander zu tun haben. Die Zusammenfassung mehrerer nichtdeterministischer Berechnungen zu gemeinsamen Strukturen nennt man auch **eingekapselte Suche**: dies bezeichnet allgemein die Möglichkeit, nichtdeterministische Berechnungen, insbesondere deren Ergebnisse, in einer Datenstruktur zu verarbeiten.

Eine eingekapselte Suche hat verschiedene Vorteile:

- Man kann verschiedene Ergebnisse abspeichern oder in einer Benutzerschnittstelle präsentieren.
- Man kann verschiedene Ergebnisse vergleichen und z.B. das beste bezüglich einer Ordnung herausfiltern.
- Eingekapselte Suche ist auch notwendig bei Programmen, die Ein- und Ausgabe machen, da Ein/Ausgabe-Operationen (vgl. Kapitel 2.7) immer auf *einer* Welt arbeiten und eine nicht-deterministische Fortsetzung von Berechnungen auf unterschiedlichen Welten nicht möglich ist (denn wir können die Welt nicht einfach kopieren).

In diesem Kapitel wollen wir hierzu passende Techniken vorstellen und insbesondere diskutieren, welche Probleme hierbei entstehen können.

In der logischen Programmiersprache Prolog gibt es zur eingekapselten Suche das Prädikat `findall`. Ein Aufruf der Form

```
findall(X,Goal,Results)
```

liefert in der Liste `Results` alle Werte für `X`, die bei verschiedenen erfolgreichen Berechnungen für die Anfrage `Goal` ausgerechnet wurden. Bei der Verwendung von `findall` kommt die Variable `X` also in der Anfrage `Goal` vor.

In ähnlicher Weise kann man die eingekapselte Suche auch in einer logisch-funktionalen Sprache einführen. Hierzu stellt man eine Operation

```
findall :: (a → Bool) → [a]
```

zur Verfügung, die ein Prädikat (d.h. eine Abbildung von Werten in ein boolesches Ergebnis) als Argument hat und die Liste aller Werte liefert, für die dieses Prädikat zu `True`

auswertet.[8] Wenn wir z.B. die Vorfahrrelation `vorfahr` aus Kapitel 4.3 betrachten, dann würde durch den Aufruf

```
findall (\v → vorfahr v Peter)
```

die Liste aller Vorfahren von Peter (d.h. `[Monika,Johann,Christine,Anton]`) berechnet werden. Ebenso können wir dies auch zur Definition weiterer Operationen benutzen. Zum Beispiel könnten wir eine Operation, die die Liste aller Präfixe einer Liste berechnet, wie folgt definieren:

```
prefixesOf :: [a] → [[a]]
prefixesOf xs = findall (\p → p ++ _ =:= xs)
```

Damit würde der Ausdruck

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

zu `[[],[1],[1,2],[1,2,3]]` ausgewertet werden.

Obwohl es scheint, dass man mit `findall` eine einfache Möglichkeit zur eingekapselten Suche hat, ergeben sich bei einer genaueren Analyse einige mögliche Probleme:

1. Ein Prädikat könnte unendlich viele Lösungen haben, wie z.B.

   ```
   findall (\ (xs,ys) → xs ++ [42] =:= ys)
   ```

   (d.h. alle Listenpaare, wobei die zweite Liste im Vergleich zur ersten Liste das zusätzliche Element `42` am Ende hat).

   Dies ist eigentlich kein Problem, da in nicht-strikten Sprachen Listen auch unendlich sein können, d.h. in diesem Fall könnte `findall` die unendliche Liste aller Lösungen liefern, die dann bedarfsgesteuert berechnet wird.[9]

2. Da im Prinzip nicht festgelegt ist, in welcher Reihenfolge nichtdeterministische Berechnungen abgearbeitet werden, kann `findall` eine beliebige Reihenfolge wählen. Beispielsweise könnte der obige Ausdruck

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

   sowohl zu

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

   aus auch zu

[8] Eine Implementierung von `findall` steht z.B. in der Bibliothek `Control.Findall` des Pakets `searchtree` zur Verfügung.

[9] Dies ist allerdings nicht bei allen Curry-Implementierungen der Fall. Z.B. übersetzt PAKCS Curry-Programme in Prolog-Programme, sodass `findall` wie in Prolog strikt ausgewertet wird.

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

ausgewertet werden (oder auch zu jeder anderen Permutation dieser Liste). Tatsächlich könnte es bei einer parallelen oder nebenläufigen Auswertung aller nichtdeterministischen Verzweigungen bei wiederholten Auswertungen des gleichen Ausdrucks zu unterschiedlichen Ergebnissen kommen, was natürlich bei einer deklarativen Sprache nicht erwünscht ist.

Dieses Problem könnte man dadurch umgehen, indem man keine Liste, sondern eine Menge oder Multimenge von Ergebnissen zurückliefert.

3. Ein weiteres Problem ist die Frage, was eigentlich eingekapselt werden soll. Betrachten wir hierzu folgendes Beispiel, bei dem alle Präfixe einer nichtdeterministischen Liste berechnet werden sollen:

```
oneTwoPrefixes = prefixesOf oneOrTwo
  where oneOrTwo = [1] ? [2]
```

Wenn `findall` alle nichtdeterministischen Berechnungen einkapselt, sollte es alle Präfixe der Listen `[1]` und `[2]` zurückliefern, d.h. die Liste

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

Wenn allerdings die Liste *vor* dem Aufruf von `prefixesOf` ausgewertet wird, dann würde hierdurch eine nichtdeterministische Berechnung mit zwei Alternativen entstehen, sodass dann `prefixesOf` jeweils für jede Alternative aufgerufen wird.

Betrachten wir z.B. die folgende Variante der letzten Definition:

```
oneTwoNEPrefixes | length oneOrTwo > 0 = prefixesOf oneOrTwo
  where oneOrTwo = [1] ? [2]
```

Da beide Alternativen von `oneOrTwo` nicht-leere Listen sind, sollte diese Definition gleichbedeutend mit der vorherigen sein. Durch die Auswertung von `length` entstehen allerdings zwei Berechnungszweige:

```
oneTwoNEPrefixes
→* (length oneOrTwo > 0) &> (prefixesOf oneOrTwo)
  →* (length ([1] ? [2]) > 0) &> (prefixesOf ([1] ? [2]))
    →* (length [1] > 0) &> (prefixesOf [1])
      →* prefixesOf [1] →* [[],[1]]
    →* (length [2] > 0) &> (prefixesOf [2])
      →* prefixesOf [2] →* [[],[2]]
```

Insgesamt erhalten wir hier die zwei Werte `[[],[1]]` und `[[],[2]]`, im Gegensatz zur vorherigen Definition.

Das Besondere an diesem Beispiel ist die Tatsache, dass `oneOrTwo` *außerhalb* des Aufrufes von `prefixesOf` definiert bzw. eingeführt wurde. Daher könnte man auch argumentieren, dass `findall` nur den Nichtdeterminismus einkapseln soll, der im Argument von `findall` auftaucht. Da sowohl bei `oneTwoPrefixes` als auch bei `oneTwoNEPrefixes` der Nichtdeterminismus von `oneOrTwo` außerhalb des Aufrufs eingeführt wurde, sollte dieser dann auch nicht eingekapselt werden, sodass bei beiden Varianten die zwei Ergebnisse `[[],[1]]` und `[[],[2]]` berechnet werden.

Das letzte Problem hängt mit der Kombination von eingekapselter Suche und lazy Auswertung zusammen (und taucht daher z.B. bei Prolog nicht auf). Tatsächlich wurden diese Probleme in früheren Arbeiten zur eingekapselten Suche [Hanus/Steiner 98] übersehen, sodass verschiedene Curry-Implementierungen dies unterschiedlich implementieren. Zum Beispiel liefert PAKCS bei `oneTwoPrefixes` einen Wert, wogegen MCC hier zwei alternative Werte liefert. Dieses Problem wurde erst vor einigen Jahren erkannt und wird seitdem diskutiert. Im Folgenden stellen wir hierzu eine Lösung vor.

Zunächst wollen wir allerdings noch einmal die verschiedenen Ansätze zur eingekapselten Suche erläutern. Zur Vereinfachung nehmen wir an, dass eine Operation `allValues` existiert, die die Liste aller Werte ihres Arguments liefert (wir ignorieren hier also zunächst das Problem 2 der Reihenfolge der Ergebnisse):

```
allValues :: a → [a]
```

Wir betrachten die schon früher eingeführte nichtdeterministische Operation `coin`:

```
coin = 0
coin = 1
```

Somit sollte also `allValues coin` zu der Liste `[0,1]` ausgewertet werden. In ähnlicher Weise wird der Ausdruck

```
allValues coin ++ allValues coin
```

zu der Liste `[0,1,0,1]` ausgewertet. Wenn wir allerdings den Ausdruck

```
let x = coin in allValues x ++ allValues x
```

betrachten, ist das Ergebnis nicht mehr so klar (vgl. obiges Problem 3). Hier kann man zwei Sichtweisen unterscheiden:

- Bei der **starken Einkapselung** (*strong encapsulation*) wird jeder auftretende Nichtdeterminismus eingekapselt, unabhängig davon, wo dieser herkommt. Hier wäre also das Ergebnis `[0,1,0,1]`.

  Bei der starken Einkapselung spielt also die syntaktische Form des Ausdrucks im Programm eine geringere Rolle, wichtig ist nur die Form, wenn die eingekapselte Suche gestartet wird. Allerdings kann dann, wie wir oben gesehen haben, die Auswertungsreihenfolge einen Einfluss auf das Ergebnis haben.

- Bei der **schwachen Einkapselung** (*weak encapsulation*) wird nur der Nichtdeterminismus eingekapselt, der im Argument von `allValues` auftritt, d.h. nichtdeterministische Operationen, die außerhalb definiert sind und als Parameter hineingereicht werden, werden nicht gekapselt. Hier gäbe es also die Ergebnisse `[0,0]` und `[1,1]`.

  Bei der schwachen Einkapselung ist also die syntaktische Form des Aufrufs von `allValues` relevant, allerdings hat die Auswertungsreihenfolge keinen Einfluss auf das Ergebnis. Letzteres ist bei deklarativen Sprachen wünschenswert.

Um noch einmal die Abhängigkeit der starken Einkapselung von der Auswertungsstrategie zu verdeutlichen, betrachten wir den folgenden Ausdruck:

```
let x = coin in allValues x ++ [x] ++ allValues x
```

Da der mittlere Ausdruck `[x]` zu den Werten `0` oder `1` ausgerechnet wird, könnte man mit der starken Einkapselung die beiden Gesamtergebnisse `[0,1,0,0,1]` und `[0,1,1,0,1]` erwarten. Tatsächlich wird durch die Links-Rechts-Auswertung der Konkatenationen etwas anderes ausgewertet. Nachdem das erste `allValues` zu `[0,1]` ausgewertet ist, wird nun `[x]` ausgewertet. Dadurch werden zwei alternative Berechnungen angestoßen:

- In einer Berechnung wird `x` an `0` gebunden. Da aber `x` auch im letzten Aufruf von `allValues` vorkommt, wird da also `allValues 0` ausgerechnet, was zum Ergebnis `[0]` führt. Insgesamt erhalten wir also `[0,1]++[0]++[0]`, was zu `[0,1,0,0]` ausgewertet wird.

- In einer weiteren Berechnung wird `x` an `1` gebunden, was analog zu dem Gesamtergebnis `[0,1,1,1]` führt.

Somit erhalten wir also die zwei Werte `[0,1,0,0]` und `[0,1,1,1]` an Stelle der Werte `[0,1,0,0,1]` und `[0,1,1,0,1]`. Man beachte, dass, falls der Teilausdruck `[x]` vorne oder hinten steht, wiederum andere Werte errechnet werden (welche?).

Wie man sieht, ist also das Ergebnis der starken Einkapselung sehr stark von der Auswertungsreihenfolge abhängig, sodass textuell identische Aufrufe (wie z.B. `allValues x`) im Verlaufe der Berechnung zu unterschiedlichen Ergebnissen führen. Bei der schwachen Einkapselung ist dies nicht der Fall. Hier würde das `coin`, das ja außerhalb von jedem `allValues` definiert ist, nie innerhalb von `allValues` ausgewertet, sodass wir die Ergebnisse `[0,0,0]` und `[1,1,1]` erhalten.

Ähnliche Effekte können auch bei der Verwendung logischer Variablen auftreten, denn logische Variablen sind ja sehr ähnlich zu nichtdeterministischen Operationen. Tatsächlich ist es nicht sinnvoll, logische Variablen, die außerhalb definiert werden, innerhalb einer eingekapselten Suche zu binden, denn man könnte diese ja in verschiedenen Suchzweigen an verschiedene Werte binden, sodass es außerhalb unklar wäre, welche Bindungen man betrachten soll. Dies sollte durch folgendes Beispiel klar werden:

```
boolVal False = 0
boolVal True  = 1
```

```
main | allValues (boolVal x) /= []
     = x
 where x free
```

Aus diesem Grund werden logische Variablen in einer eingekapselten Suche überhaupt nicht gebunden: wenn man den Wert einer außerhalb eingeführten logischen Variablen innerhalb einer eingekapselten Suche benötigt, wird die Suche so lange suspendiert, bis der Wert außerhalb gebunden wurde. Dieses Prinzip erinnert an Residuation, weswegen es manchmal auch als **rigide eingekapselte Suche** bezeichnet wird.

Ein Nachteil der schwachen Einkapselung ist die Abhängigkeit von der syntaktischen Struktur des jeweiligen Aufrufs, sodass scheinbar ähnliche Aufrufe zu unterschiedlichen Ergebnissen führen. Z.B. liefern die Aufrufe

```
allValues coin
```

und

```
let x = coin in allValues x
```

unterschiedliche Ergebnisse bzgl. der schwachen Einkapselung (nicht jedoch bei der starken Einkapselung). Dies ist auch der Grund, warum man nicht einfach für `allValues` weitere Abstraktionen definieren kann. Zum Beispiel könnte man auf die Idee kommen, einen neuen Suchoperator zu definieren, der in den Ergebnissen identische Werte herausfiltert:

```
allUniqValues x = nub (allValues x)
 where
  nub []     = []
  nub (x:xs) = x : nub (filter (/=x) xs)
```

Dann liefern die Ausdrücke

```
allValues coin
```

und

```
allUniqValues coin
```

bzgl. der starken Einkapselung identische Werte, während bei der schwachen Einkapselung beim ersten Ausdruck der Wert `[0,1]` und beim zweiten Ausdruck die Werte `[0]` oder `[1]` berechnet werden (weil hier `coin` als Parameter zu `allValues` hineingereicht wird!).

Diese Abhängigkeit von der Aufrufstruktur ist zwar ein Nachteil der schwachen Einkapselung, allerdings wiegt der Vorteil der Unabhängigkeit von der Berechnungsreihenfolge doch sehr stark, sodass letztendlich die schwache Einkapselung für eine deklarative Spra-

che sinnvoll ist. Auf der anderen Seite ist es nützlich, wenn man klare syntaktische Konventionen hat, welche Teile bei der eingekapselten Suche wirklich eingekapselt werden und welche nicht. Macht man dies nicht deutlich, dann führt dies leicht zu Missverständnissen wie im Beispiel `allUniqValues`.

Ein Konzept, was diese Forderungen unterstützt, sind **Mengenfunktionen** oder **Set Functions** [Antoy/Hanus 09]. Nach unserer bisherigen Diskussion ist die Definition recht einfach:

Definition 4.13 (Set Function) *Für eine Funktion*

$$f \ :: \ \tau_1 \ \to \cdots \ \to \tau_n \ \to \tau$$

ist die zugehörige Mengenfunktion oder Set Function als

$$f_{\mathcal{S}} \ :: \ \tau_1 \ \to \cdots \ \to \tau_n \ \to 2^{\tau}$$

definiert, wobei 2^{τ} alle Mengen von Werten aus τ bezeichnet. Der Ausdruck $f_{\mathcal{S}}\ e_1 \ldots e_n$ liefert die Menge aller Werte, zu der der Ausdruck

$$\begin{array}{l} \texttt{let}\ x_1 = e_1 \\ \quad\ \vdots \\ \quad\ x_n = e_n \\ \texttt{in}\ f\ x_1 \ldots x_n \end{array}$$

ausgewertet wird, wobei der Nichtdeterminismus in den Argumenten $e_1, \ldots, e_n$ nicht eingekapselt wird.

Damit entspricht der Ausdruck $f_{\mathcal{S}}\ e_1 \ldots e_n$ also dem Aufruf

$$\begin{array}{l} \texttt{let}\ x_1 = e_1 \\ \quad\ \vdots \\ \quad\ x_n = e_n \\ \texttt{in allValues}\ (f\ x_1 \ldots x_n) \end{array}$$

bezüglich der schwachen Einkapselung, wobei das Ergebnis keine Liste, sondern eine Menge ist. Der Ausdruck $f_{\mathcal{S}}\ e_1 \ldots e_n$ liefert also Wertemengen, wobei nur der Nichtdeterminismus von f, nicht jedoch von den Argumenten eingekapselt wird. Damit wird durch jede definierte Funktion eine syntaktische „Einkapselungsgrenze“ bereitgestellt. Betrachten wir hierzu folgende Definitionen:

```
coin = 0 ? 1

bigCoin = 2 ? 4

f x = coin + x
```

Dann liefert der Ausdruck "`f`$_{\mathcal{S}}$ `bigCoin`" die beiden Mengen `{2,3}` und `{4,5}` als Ergebnis.

Die Wertemengen sind ein abstrakter Datentyp, auf dem bestimmte Operationen definiert sind, wobei man aber z.B. nicht auf das „erste" Element zugreifen kann. Die Curry-Implementierungen PAKCS und KiCS2 stellen Mengenfunktionen in Form der Bibliothek `Control.SetFunctions` zur Verfügung.[10] Wenn man diese Bibliothek importiert, erhält man die Mengenfunktion zu einer n-stelligen Funktion f durch den Ausdruck `set`n f. Die Wertemenge wird durch den polymorphen abstrakten Datentyp `Values` repräsentiert, auf dem im Modul `Control.SetFunctions` einige Operationen zum Leerheitstest, Elementtest und weiteren Verarbeitung definiert sind, wie z.B.

```
isEmpty     :: Values a → Bool                -- leere Wertemenge?
valueOf     :: Eq a => a → Values a → Bool    -- Element enthalten?
foldValues  :: (a → a → a) → a → Values a → a
mapValues   :: (a → b) → Values a → Values b
maxValue    :: Ord a => Values a → a
maxValueBy  :: (a → a → Ordering) → Values a → a
minValue    :: Ord a => Values a → a
minValueBy  :: (a → a → Ordering) → Values a → a
sortValues  :: Ord a => Values a → [a]
values2list :: Values a → IO [a]
printValues :: Show a => Values a → IO ()
```

Mit `sortValues` kann man eine Menge in eine sortierte Liste umwandeln. `values2list` und `printValues` wandelt die Menge ohne Sortierung um (was unproblematisch ist, weil dies in der I/O-Monade stattfindet).

Somit können wir das letzte Beispiel wie folgt programmieren:

```
coin = 0 ? 1

bigCoin = 2 ? 4

f x = coin + x

main1 = sortValues (set1 f bigCoin)

main2 = foldValues (+) 0 (set1 f bigCoin)
```

Hier wird `main1` zu den beiden Listen `[2,3]` und `[4,5]` ausgewertet, und `main2` wird zu den Werten `5` und `9` ausgewertet.

[10] Diese befindet sich im Paket `setfunctions`, d.h. vor der Benutzung muss man dieses Paket z.B. mit dem Kommando "`cypm add setfunctions`" installieren.