Programmierparadigmen

Sommerersemester 2017

Martin Wittiger, Felix Krause, Timm Felden

6. Übung

Abgabe bis 12. Juli um 4:44

Beachten Sie die Abgabehinweise auf der Vorlesungswebseite!

Aufgabe 6.1 ..... 6 Punkte

Dateiname: `plant.hs` ★★☆

Gegeben seien folgende Haskell-Datentyp-Definitionen:

```
data Farbe = Rot | Rosa | Weis | Blau | Lila | Grun | Gelb
   deriving (Show, Eq)

data Pflanze =
      Blatt
   |  Blute Farbe
   |  Stiel Pflanze Pflanze
   deriving (Show, Eq)
```

Mit diesen Typen lassen sich Pflanzen konstruieren. Pflanzen haben eine – im Informatik-Sinne – baumförmige Struktur. Blüten haben eine Farbe. Es gibt auch keine »leere« Pflanze. Eine Pflanze besteht zumindest aus einem Blatt oder einer Blüte.

1. Geben Sie einen Beispiel-Ausdruck an, der eine Pflanze mit zwei Blättern und einer roten Blüte erzeugt.
2. Implementieren Sie eine Haskell-Funktion `fold_pflanze`, die den folgenden Typ hat:

   ```
   (a -> a -> a) -> (Farbe -> a) -> a -> Pflanze -> a
   ```

   Die Funktion hat die für Folds übliche Semantik. Das bedeutet insbesondere, dass `fold_pflanze Stiel Blute Blatt` die Identität auf Pflanzen ist. Auch muss sich die Funktion eignen, die folgenden Aufgaben zu lösen.
3. Implementieren Sie eine Haskell-Funktion `blattanzahl`. Die Funktion hat den Typ `Pflanze -> Integer`. Wenn ihr eine Pflanze übergeben wird, wird die Zahl der in der Pflanze enthaltenen Blätter zurückgegeben. Verwenden Sie zur Implementierung die Funktion `fold_pflanze`.
4. Implementieren Sie eine Haskell-Funktion `blutenfarben`. Die Funktion hat den Typ `Pflanze -> [Farbe]`. Wenn ihr eine Pflanze übergeben wird, wird eine Liste aus allen in der Pflanze vorkommenden Blütenfarben zurückgegeben. Die Liste enthällt jede Farbe so oft, wie es Blüten dieser Farbe in der Pflanze gibt. Die Reihenfolge ist nicht definiert. Verwenden Sie zur Implementierung die Funktion `fold_pflanze`.

5. Die Schönheit einer Pflanze ist die Summe der Schönheit ihrer Bestandteile. Die Bestandteile der Pflanze haben folgende Schönheiten:
   - Ein Blatt hat die Schönheit 1.
   - Ein Stiel hat die Schönheit -2.
   - Eine grüne Blüte hat die Schönheit -1
   - Rote, rosa und weiße Blüten haben jeweils die Schönheit 2, 3 und 5.
   - Eine lila Blüte hat die Schönheit 10.
   - Eine blaue Blüte hat die Schönheit 15.
   - Eine gelbe Blüte hat die Schönheit 0.

   Implementieren Sie eine Haskell-Funktion `schonheit`. Die Funktion hat den Typ `Pflanze -> Integer`. Wenn ihr eine Pflanze übergeben wird, wird die Schönheit der Pflanze zurückgegeben.

6. Implementieren Sie eine Haskell-Funktion `rosabluhend`. Die Funktion hat den Typ `Pflanze -> Bool`. Wenn ihr eine Pflanze übergeben wird, wird zurückgegeben, ob alle Blüten der Pflanze rosa sind. Verwenden Sie zur Implementierung die Funktion `fold_pflanze`.

Wir werden diese Aufgabe zum Teil manuell bewerten. Achten Sie ganz besonders darauf, dass Ihre Lösung übersichtlich dargestellt ist und keine unnötigen Teile enthält. Gute Lösungen sind kurz und bündig. Kommentare sind dennoch gerne gesehen.

Keine Zeile ist länger als 80 Zeichen. Ihre Datei enthält maximal 60 Zeilen. Ihre Datei ist so vollständig, dass sie sich mit ghci laden lässt. Kopieren Sie insbesondere die Datentypdefinitionen an den Beginn der Datei oder verwenden Sie das Grundgerüst, das auf der Vorlesungseite steht.

Aufgabe 6.2 ..... 4 Punkte

Dateiname: `grow.hs` ★☆☆

In dieser Aufgabe geht es um dieselben Pflanzen wie in der vorherigen Aufgabe. Auch die Schönheit ist genauso definiert. Implementieren Sie die Haskell-Funktion `grow`. Diese gibt, wenn sie auf die Zahl n angewendet wird, eine Pflanze zurück, die die Schönheit n hat.

Formaler definiert muss gelten, dass `schonheit . grow` die Identität auf Integer ist. Es muss gelten, dass `(schonheit . grow) n == n` für jeden Integer-Wert `n` zu `True` reduziert wird. Ihre Pflanze muss also insbesondere endlich sein.

Sie dürfen Hilfsfunktionen definieren. Achten Sie darauf, den Namespace nicht unnötig zu füllen. Definieren Sie auf keinen Fall ein Symbol `main` oder ein Modul.

Aufgabe 6.3 ..... 4 Punkte

Dateiname: `tricky.pdf` ★★★

Es seien folgende Haskell-Definitionen gegeben:

```
lucky = all (/=13)

unlucky1 = any (==13)
unlucky2 = not . lucky

catenate as []     = as
catenate as (b:bs) = b : (catenate as bs)

test_luck1 as bs = lucky as && lucky bs
test_luck2 as bs = lucky (catenate as bs)
```

Die Funktionen `test_luck1` und `test_luck2` ähneln sich sehr, oder etwa nicht? Sie sind allerdings nicht gleich. Es gibt Ausdrücke, bei denen sich `test_luck1` und `test_luck2` unterschiedlich verhalten.

1. Geben Sie Haskell-Ausdrücke `a`, `b`, `c` und `r`, `s`, `t` an, für die gilt:
   a) `test_luck1 a b` wird zu `c` reduziert, aber `test_luck2 a b` kann nicht zu `c` reduziert werden.
   b) `test_luck2 r s` wird zu `t` reduziert, aber `test_luck1 r s` kann nicht zu `t` reduziert werden.

   Eine formal richtige Abgabe kann zum Beispiel so aussehen:

   ```
   a = [1,2,3]
   b = [13] ++ tail [13,14,15]
   c = False
   ```

   Inhaltlich ist diese Abgabe allerdings falsch. Es wird zwar wie gefordert `test_luck1 a b` zu `False` reduziert, allerdings ebenso `test_luck2 a b`.
2. Erklären Sie klar und deutlich, wie sich die beiden Funktionen unterscheiden. Charakterisieren Sie genau, bei welchen Eingaben Unterschiede auftreten.
3. Unterscheiden sich `unlucky1` und `unlucky2` in ähnlicher Weise? Falls ja, charakterisieren Sie genau, bei welchen Eingaben Unterschiede auftreten.

Aufgabe 6.4 ..... 3 Punkte

Dateiname: `simplify.hs`

In dieser Teilaufgabe geht es um die in der Datei `simplify.hs` definierten Funktionen `f`, `g` und `h`.

Diese sind auf ziemlich abstruse Weise definiert. Sie sollen die Definitionen vereinfachen, dabei aber die Semantik erhalten. Versuchen Sie bei der Definition auf Prelude-Funktionen zurück zu greifen, wo sich das anbietet.

Sie dürfen davon ausgehen, dass die Addition auf beliebigen Zahlen assoziativ und kommutativ ist.

Sie sollen jede der Funktionen in einer Zeile definieren, die maximal 20 Zeichen enthält. Ihre Abgabe soll sich ohne Fehlermeldungen in GHCi laden lassen.

Aufgabe 6.5 ..... 4 Punkte

Dateiname: `sbahn.pl`

★★★

Auf der Webseite der Veranstaltung finden Sie ein Prolog-Script (`sbahn.pl`). Dieses können Sie mit dem Online-Interpreter auf `http://swish.swi-prolog.org/` verwenden.

In der Datei sind einige Fakten definiert. Diese geben (ausschnittsweise) das S-Bahn-Netz rund um Stuttgart wieder. (Vergleich auch: `http://www.vvs.de/download/SBahn_Liniennetz.pdf`) `connection(xxx,yyy)` bedeutet, dass zwischen `xxx` und `yyy` (in beiden Richtungen) S-Bahnen verkehren.

1. Definieren Sie ein Prolog-Prädikat `connected(A,B)`, das ermittelt, ob zwei Stationen direkt oder indirekt miteinander verbunden sind. Ihr Prädikat muss mit der Datenbasis wirklich funktionieren! `?-connected(renningen,vaihingen)` soll `true` ergeben und `?-connected(hbf,isartor)` hingegen `false`.

2. Definieren Sie ein Prolog-Prädikat `valid(PFAD)`, das ermittelt, ob die Liste `PFAD` ein valider Pfad im Netz ist. `?-valid([korntal,A,B,C,D,E,vaihingen])` soll als Antwort etwas wie `A = zuffenhausen, B = hbf, C = universitaet, D = vaihingen, E = boeblingen` ergeben.

   `?-valid([hbf,zuffenhausen,korntal])` ergibt `true`.

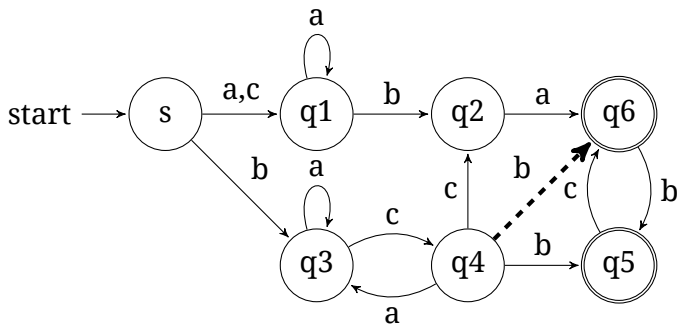
   `?-valid([hbf,cannstatt,tamm])` und `?-valid([hbf,odeonsplatz])` ergeben hingegen jeweils `false`.

Aufgabe 6.6 ..... 3 Punkte

Dateiname: `plstate.pdf`

★★☆

Gegeben sei folgender deterministische Automat (bitte ignorieren Sie zunächst den gestrichelten Übergang zwischen den Zuständen **4** und **6** und nehmen diesen **nicht** mit in Ihre Fakten auf):



1. Tragen Sie die gegebenen Fakten in eine Prolog-Wissenbasis ein. Hinterlegen Sie mit `final(...)` Fakten, die die Endzustände markieren. Zur Modellierung der Übergänge nutzen Sie `transition(x,m,y)`, wobei dies den Übergang von `x` zu `y` bei zu verarbeitender Eingabe `m` darstellen soll.

2. Schreiben Sie einen Simulator, der bei Angabe eines Startzustands und einer Liste von Eingabezeichen prüft, ob ein Endzustand erreicht wird, indem Sie ein Prädikat der Form `fsmsim(s, [...])` mit Prolog-Regeln spezifizieren.

3. Welche Ergebnisse liefert Ihr Automat jeweils für die folgenden Aufrufe?

```
fsmsim(s, []).
fsmsim(s, [b,a,a,c,a,a,a,c,b,c]).
fsmsim(S, []).
fsmsim(s, [b,c,b]).
fsmsim(s, [c,a,a,b,a,b,c]).
fsmsim(s, [a,b,a]).
```

4. Nehmen Sie nun den in Teilaufgabe 1 ausgelassenen (gestrichelten) Übergang in Ihre Wissensbasis mit auf. Bitte platzieren Sie diesen Übergang am Schluss Ihrer bisherigen Fakten-Liste. Führen Sie nun den Test `fsmsim(s, [b,c,b])` aus Teilaufgabe 3 erneut aus. In welchem Endzustand landet Ihr Automat?

5. Platzieren Sie nun den neu hinzugefügten Übergang am Beginn Ihrer Fakten-Liste und testen Sie erneut mit der Eingabe `fsmsim(s, [b,c,b])`. Ändert sich etwas am erreichten Endzustand, wenn ja, warum?

6. Zuletzt testen Sie mit der Eingabe `fsmsim(s, [b,c,b,c])`! Ändert sich etwas am erreichten Endzustand, wenn ja, warum? Wie „löst“ Prolog das Problem?

Hinweis: Für die Teilaufgaben 4 bis 6 sollten Sie die eingebauten Prädikate `trace` bzw. `notrace` nutzen, um so dem Interpreter genauer auf die Finger schauen zu können. Geben Sie für diese drei Teilaufgaben jeweils die erste gefundene Lösung an. Ihre Abgabe für diese Aufgabe ist eine PDF-Datei, die Ihre Ergebnisse kurz und übersichtlich darstellt.