Institut für Programmstrukturen und Datenorganisation
Prof. Dr.-Ing. Gregor Snelting — gregor.snelting@kit.edu
Prof. Dr. Ralf Reussner — reussner@kit.edu

Programmierparadigmen – WS 2021/22

https://pp.ipd.kit.edu/lehre/WS202122/paradigmen/uebung

Blatt 8: Cuts

Abgabe: 17.12.2021, 14:00
Besprechung: 20.12. – 21.12.2021

Reichen Sie Ihre Abgabe bis zum 17.12.2021 um 14:00 in unserer Praktomat-Instanz unter https://praktomat.cs.kit.edu/pp_2021_WS ein.

1 Tester, Generatoren, Ausführungsbäume

In der Vorlesung wurden Prädikate `odd(X)` und `even(X)` vorgestellt. Diese *testen* ob `X` gerade ist, bzw. ob `X` ungerade ist. So wird die Anfrage `?even(4)` erfüllt, `?odd(4)` hingegen nicht.

```
even(0).
even(X) :- X>0, X1 is X-1, odd(X1).

odd(1).
odd(X)  :- X>1, X1 is X-1, even(X1).
```

Listing 1: Tester

Die folgende Variante ist geeignet zum *Generieren* aller geraden bzw. aller ungeraden Zahlen. Beispielsweise gibt die Anfrage `?even(X)` bei wiederholter Neuerfüllung alle geraden Zahlen aus.

```
even(0).
even(X) :-  odd(Y), X is Y+1, X>0.

odd(1).
odd(X)  :- even(Y), X is Y+1, X>1.
```

Listing 2: Generatoren

1. Warum ist keine der beiden Varianten sowohl als Tester als auch als Generator anwendbar?

   Beispiellösung: Wir betrachten beispielhaft den Tester `even`. Bei Anfrage `?even(X)` generiert dieser zunächst zwar (mit Regel $\texttt{even}_1$) die Instanziierung `X=0`, bei Reerfüllung aber werden die Teilziele `X>0, X1 is X-1, odd(X1)` für *uninstanziiertes* `X` generiert. Arithmetischer Vergleich mit > funktioniert jedoch nur, wenn alle Variablen in den zu vergleichenden Termen bereits zu einer Zahl instanziiert sind! Würde `X>0` weggelassen, würde man an **`is`** scheitern.

   Für die Generatoren (beispielsweise: `odd`) gilt: Erfüllbare Anfragen funktionieren zunächst. Beispielsweise wird die Anfrage `?odd(7)` mit **`true`** bestätigt. Dazu generiert Prolog so lange alle ungeraden Zahlen `X`, bis `X=7` gilt, und hat damit einen Beweis gefunden, dass 7 ungerade ist. Allerdings kann Prolog nicht feststellen, dass es anhand der Generator-Regeln keinen weiteren

Beweis dafür gibt, dass 7 ungerade ist. Reerfüllung von `?odd(7)` führt daher zu einer Endlosschleife: es werden immer weitere ungerade Zahlen `X` > 7 generiert.
Genauso kann Prolog mit den Generator-Regeln nicht feststellen, dass z.B. 8 nicht ungerade ist: Anfrage `?odd(8).` terminiert nicht.

Beim Tester hätte Prolog hier feststellen können, dass mit den Tester-Regeln kein Beweis für `odd(8)` möglich ist: die Anfrage reduziert schließlich zum (einzigen) Teilziel `odd(0)`. Regel $\texttt{odd}_1$ ist nicht anwendbar: `0` unifiziert nicht mit `1`. Regel $\texttt{odd}_2$ wird zunächst angewandt, aber der Test `0>1` schlägt sofort fehl.

2. Was passiert, wenn bei den Generatoren die Teilziele `X>0` und `X>1` weggelassen werden?

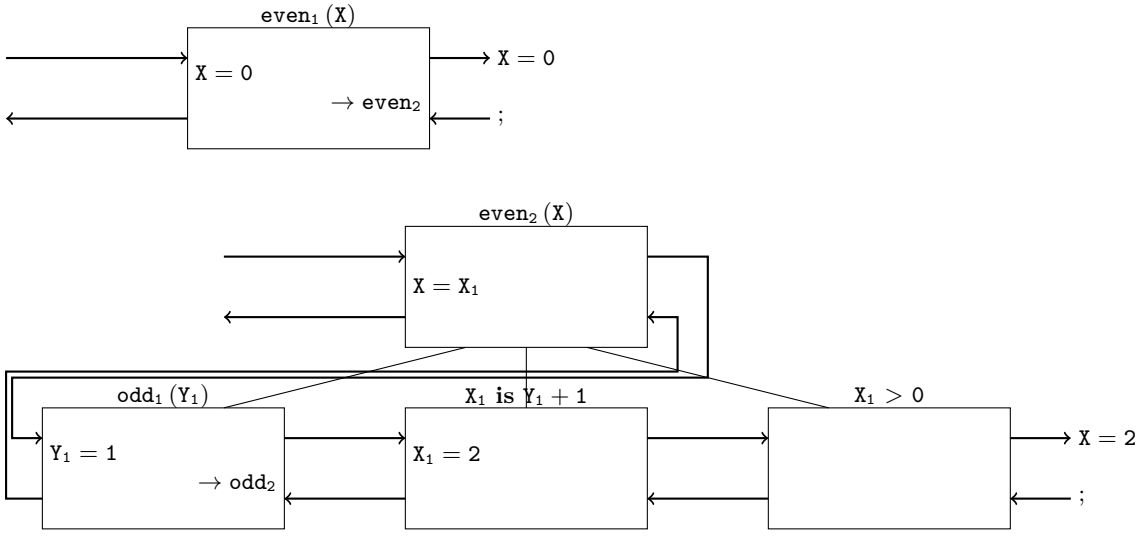
   Beispiellösung: Lässt man den Test `X>1` weg, so bemerken wir zunächst, dass die Aussage `odd(1)` dann auf zwei Wegen beweisbar ist: einmal direkt mit dem Fakt $\texttt{odd}_1\texttt{(1)}$, aber auch auch mit dem Fakt $\texttt{even}_1\texttt{(0)}$ und einer Anwendung von Regel $\texttt{odd}_2\texttt{(X)}$.

   Bei einer Anfrage `even(X)` basiert nun jede Generierung einer geraden Zahl (0 ausgenommen) auf der Generierung der ungeraden Zahl 1 – da dies auf zwei Wegen möglich ist, wird in dieser Anfrage also jede gerade Zahl doppelt generiert. Genauso werden alle ungeraden Zahlen bei Anfrage `?odd(X)` doppelt generiert.

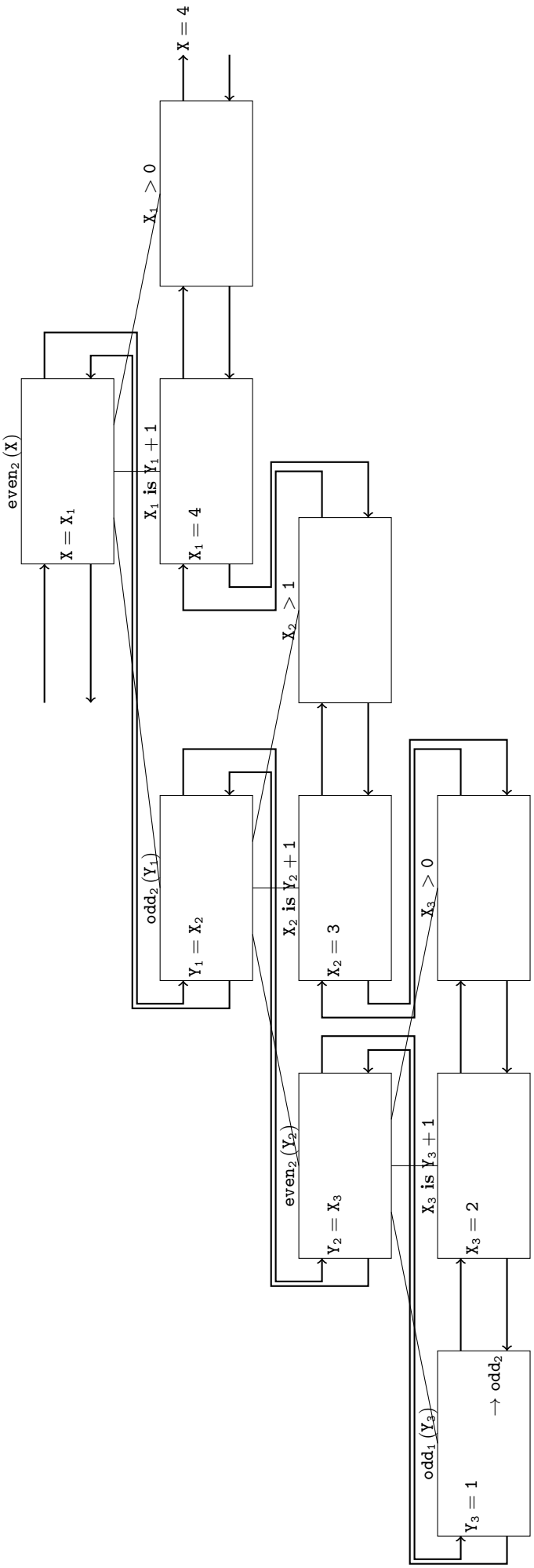
   Das Teilziel `X>0` wegzulassen hingegen macht keinen Unterschied. Dieses Teilziel ist ohnehin überflüssig: immer wenn es aufgerufen wird, ist X sowieso mit einer Zahl > 0 instanziiert.

3. Im Folgenden sind für die Anfrage `?even(X)` unter Verwendung der Generator-Regeln von oben die Ausführungsbäume zum Zeitpunkt der Ausgabe von `X=0` bzw. `X=2` dargestellt.

   Zeichnen Sie den Ausführungsbaum zum Zeitpunkt der Ausgabe von `X=4`. Welches Teilziel schlägt dabei einmal fehl?



Beispiellösung: Zwischen dem Zeitpunkt der Ausgabe `X=2` und dem der Ausgabe `X=4` (abgebildet auf nächster Seite) schlägt einmal das Teilziel $\texttt{X}_2 > 1$ fehl. Zu diesem Zeitpunkt wurde noch versucht, das Teilziel $\texttt{even}\,(\texttt{Y}_2)$ mit Regel $\texttt{even}_1$ zu lösen - es galt also $\texttt{X}_2 = 1$


X = 4
X1 > 0
even2 (X)
X = X1
X1 is Y1 + 1
X1 = 4
X2 > 1
odd2 (Y1)
Y1 = X2
X2 is Y2 + 1
X2 = 3
X3 > 0
even2 (Y2)
Y2 = X3
X3 is Y3 + 1
X3 = 2
odd1 (Y3)
Y3 = 1
→ odd2

2 Prolog, freie Variablen [basierend auf Klausuraufgabe vom WS 2014/2015]

Gegeben seien folgende Varianten eines Prädikats `del(L1,X,L2)` zum Löschen von `X` in Liste `L1`.

```
del1([],_,[]).
del1([X|T1],X,L2)      :- !, del1(T1,X,L2).
del1([Y|T1],X,[Y|T2]) :- del1(T1,X,T2).

del2([],_,[]).
del2([X|T1],X,L2)      :- del2(T1,X,L2).
del2([Y|T1],X,[Y|T2]) :- del2(T1,X,T2), not(X=Y).

del3([X|L],X,L).
del3([Y|T1],X,[Y|T2]) :- del3(T1,X,T2).
```

1. Geben Sie für jede Variante an, was diese unter der gegebenen Anfrage ausgibt. [6 Punkte]

   Anfrage: `del`$_i$`([1,2,1],X,L).`

   Mögliche Ausgaben:

   a) `X=1,L=[2]` und
   `X=2,L=[1,1]`

   b) `X=1,L=[2]`

   c) `X=1,L=[2,1]` und
   `X=2,L=[1,1]` und
   `X=1,L=[1,2]`

   d) nichts (Endlosschleife, Stacküberlauf o. Ä.)

   Beispiellösung:

.. gibt aus ..	a)	b)	c)	d)
`del`$_1$	☐	☒	☐	☐
`del`$_2$	☒	☐	☐	☐
`del`$_3$	☐	☐	☒	☐

2. Folgende Anfragen versuchen, die Rückwärtsausführung des Prädikats `del` ausnutzen. Welche Anfragen sind mit welcher Variante von `del`$_i$ erfüllbar? Warum ist das so?

   a) `del`$_i$`(L, 2, [1, 3]).`

   b) `del`$_i$`([1, 2, 3], X, [1, 3]).`

   c) `del`$_i$`([1, 2, 3, 2], X, [1, 3]).`

   d) `del`$_i$`([1, 2, 3, 2], X, [1, 2, 3]).`

   e) `del`$_i$`([1|L], 1, X).`

   Beispiellösung:

erfüllt	a)	b)	c)	d)	e
`del`$_1$	☐	☐	☐	☐	☒
`del`$_2$	☐	☒	☒	☐	☒
`del`$_3$	☒	☒	☐	☒	☒

In Prolog können λ-Ausdrücke als Terme dargestellt werden. Wir betrachten nur Terme der Form `42` (Integer-Konstante), `x` (Variable), `abs(x,T)` (λ-Abstraktion) und `app(T,U)` (Applikation).

3. Geben Sie ein Prolog-Prädikat `fv(T,F)` an, das zu einem Lambda-Ausdruck `T` die Liste `F` der in `T` frei vorkommenden Variablen berechnet. [6 Punkte]

   Beispiel: für den Lambda-Ausdruck (λx. λy. x z 17) u

   ```
   ?fv(app(abs(x, abs(y, app(app(x, z), 17))), u), F).
   F = [z,u];
   no.
   ```

   Hinweis: Sie können der Einfachheit halber annehmen, dass λ-gebundene Variablen nicht anderswo im Term frei verwendet werden.
   Verwenden Sie Listenprädikate wie `append`, $\texttt{del}_i$, ... sowie Testprädikate **`integer`**`(X)` und **`atom`**`(X)`.

   Beispiellösung:

   ```
   fv(T, []) :- integer(T).
   fv(T, [T])  :- atom(T).
   fv(app(T,U), F) :- fv(T, F1), fv(U, F2), append(F1, F2, F).
   fv(abs(X,T), F) :- fv(T, F1), del2(F1, X, F).
   ```

   Bemerkung: $\texttt{del}_3$ ist genau das `delete` aus den Vorlesungsfolien.

   In `fv` kann statt $\texttt{del}_2$ auch $\texttt{del}_1$ verwendet werden: beide Varianten „löschen“ bei instanziierten `L1` und `X` *alle* Vorkommen von `X`:

   ```
   ? del2([1,2,1],1,L2).          ? del1([1,2,1],1,L2).
     L2 = [2] ;                     L2 = [2] ;
   ```

   Anders als $\texttt{del}_1$ und $\texttt{del}_2$ ist $\texttt{del}_3$ ist bei solchen Anfragen reerfüllbar: bei jeder Erfüllung wird *ein* Vorkommen von `X` „gelöscht“. Würde man $\texttt{del}_3$ in `fv` verwenden, gäbe es z.B. Probleme bei λx. x x:

   ```
   ? del3([1,2,1],1,L2).          ? fv(abs(x,app(x,x)),F).
     L2 = [2,1] ;                   F = [x] ;
     L2 = [1,2] ;                   F = [x] ;
   ```

3 Haskell, Prolog, Listenverarbeitung [alte Klausuraufgabe, 15 Punkte]

1. Implementieren Sie eine Haskell-Funktion [8 Punkte]

   ```
   splits :: [t] -> [([t], [t])]
   ```

   die alle möglichen Zerlegungen der übergebenen Liste in einen Anfangsteil und einen Endteil berechnet, so dass Anfangsteil und Endteil aneinandergehängt wieder die ursprüngliche Liste ergeben. Anfangs- und Endteil sollen jeweils in einem Tupel gespeichert und alle möglichen Tupel als Liste zurückgeliefert werden.
   Hinweis: Sie können List Comprehensions verwenden.
   Beispiel:

   ```
   > splits [1,2,3]
   [ ([],[1,2,3]), ([1],[2,3]), ([1,2],[3]), ([1,2,3],[]) ]
   ```

Beispiellösung:

```
splits :: [t] -> [([t], [t])]
splits l  = [(take n l, drop n l) | n <- [0..length l]]

-- Alternativ
splits' :: [t] -> [([t], [t])]
splits' []    = [([], [])]
splits' (x:l) = ([], x:l) : [(x:s, e) | (s, e) <- splits' l]
```

2. Implementieren Sie ein Prolog-Prädikat [7 Punkte]

```
splits(L, Res)
```

das **bei Reerfüllung** alle möglichen Zerlegungen der übergebenen Liste `L` in einen Anfangsteil und einen Endteil generiert, so dass Anfangsteil und Endteil aneinandergehängt wieder die ursprüngliche Liste ergeben. Anfangs- und Endteil sollen jeweils als Tupel in `Res` zurückgeliefert werden.
Beispiel:

```
? splits([1,2,3], Res).
Res = ([], [1,2,3]) ;
Res = ([1], [2,3]) ;
Res = ([1,2], [3]) ;
Res = ([1,2,3], []) ;
No
```

Beispiellösung:

```
splits(L, ([], L)).
splits([X|L], ([X|S], E)) :- splits(L, (S, E)).

% Alternativ (nutzt Rueckwaertsausfuehrung von append)
splits(L, (Xs, Ys)) :- append(Xs, Ys, L).
```