Prof. Martin Hofmann, PhD
Dr. Steffen Jost

Ludwig-Maximilians-Universität München
Institut für Informatik

14. Juli 2015

12. Musterlösung zur Vorlesung

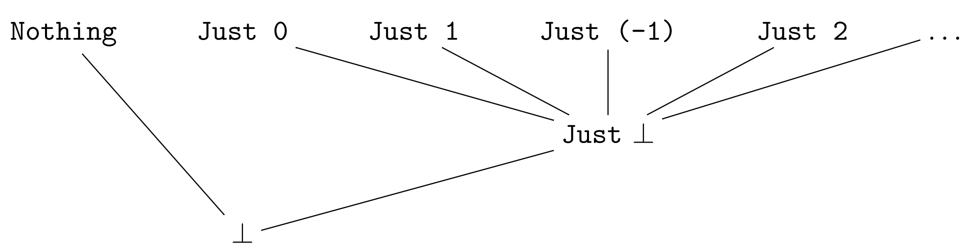
Programmierung und Modellierung

A12-1 ***Hasse Diagramme II*** Zeichen Sie jeweils ein Hasse Diagramm für die vollständige Halbordnung (engl. directed-complete partial order, kurz: **dcpo**), welche die nachfolgenden Datentypen im Sinne der denotationellen Semantik modelliert.

Dazu müssen Sie sich überlegen: Welche Werte enthalten diese Datentypen? Welche teilweise-definierten Ergebnisse können auftreten? Orientieren Sie sich an den Hasse Diagrammen für `Maybe Bool` und `[()]` im Wikibuch-Kapitel Haskell/Denotational_semantics.

a) `Maybe Int`

Beispiel: Für den Datentyp `Maybe Int` können wir in Haskell folgende Werte definieren: `undefined`, `Nothing`, `Just undefined`, `Just 1`, `Just 2`, `Just -1`, ... Wir können leicht Programme angeben, welche sich für je zwei dieser Werte unterschiedlich verhalten. Zum Beispiel gilt `isJust (Just undefined) == True`, dagegen aber `isJust undefined == undefined`, d.h. die Funktion `isJust` aus Modul `Data.Maybe` kann diese Werte unterscheiden, weshalb unsere denotationelle Semantik diese ebenfalls unterscheiden muss. Natürlich könnte man auch `error "e"` hinzunehmen, aber da wir kein Programm angeben können, welches zwischen `undefined`, `error "e"` oder nicht-termination unterscheidet,[1] interpretieren wir all diese Ausdrücke mit $\perp$. Wir zeichnen entsprechend:

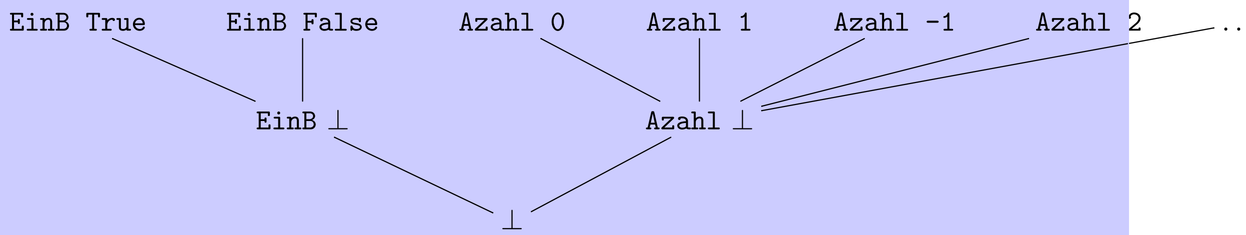


Hinweis: Für das Diagramm ist es unerheblich, in welcher Zeile wir `Nothing` eintragen; wichtig ist nur, dass es oberhalb von $\perp$ liegt.

[1] Da ghc eingebaute Mechanismen zu Fehlerbehandlung bietet, könnte man in der Praxis schon zwischen verschiedenen Fehlern unterscheiden. Eine Fehlerbehandlung ohne eine explizite Fehlermonade (wie anhand von `Maybe` oder `Either` gezeigt) passt eigentlich nicht in die rein funktionalen Welt hinein, weshalb wir in unserer denotationellen Semantik auch darauf verzichten.

b) `data Entweder = EinB Bool | Azahl Int`

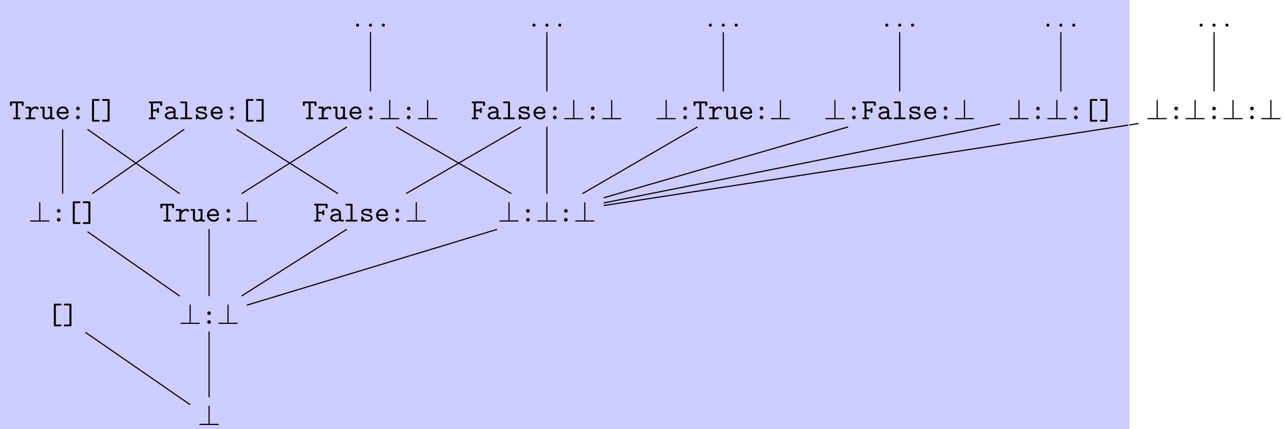
LÖSUNGSVORSCHLAG:



c) `[Bool]`
Hinweis: Beschränken Sie Ihr Hasse-Diagramm auf Werte mit bis zu 3 Konstruktoren.

LÖSUNGSVORSCHLAG:

In jedem Schritt nach "oben" ersetzen wir ein ⊥ durch einen weiter definierten Wert. Steht dabei ⊥ vor einem `:`, dann müssen wir einmal `True` und einmal `False` einsetzen; steht es am Ende der Liste, dann müssen wir einmal `[]` und einmal ⊥`:`⊥ einsetzen:



In der untersten Zeile haben wir 0 Konstruktoren, darüber 1, dann 2 und schließlich 3. In der nächsten Zeile hätten wir dann alle Werte mit 4 Konstruktoren:

True:True:⊥	False:True:⊥	⊥:True:True	⊥:False:True	⊥:⊥:⊥:⊥:⊥
True:False:⊥	False:False:⊥	⊥:True:False	⊥:False:False	
True:⊥:[]	False:⊥:[]	⊥:True:[]	⊥:False:[]	
True:⊥:⊥:⊥	False:⊥:⊥:⊥	⊥:True:⊥:⊥	⊥:False:⊥:⊥	

A12-2 ***Approximation Rekursiver Funktionen*** Berechnen Sie f_0, f_1, f_2, f_3 und g analog zu dem Beispiel aus Abschnitt "Recursive Definitions as Fixed Point Iterations" des Wikibuch-Kapitels Haskell/Denotational_semantics. Implementieren Sie g anschliessend in Haskell!

a) `f(x) = if x==0 then 0 else x + f(x-1)`

LÖSUNGSVORSCHLAG:

Wie in der Vorlesung geben wir die Funktionen durch eine Wertetabelle an:

	0	1	2	3	4	5	6	7	...
f_0	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	
f_1	0	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	
f_2	0	1	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	
f_3	0	1	3	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	
f_4	0	1	3	6	$\perp$	$\perp$	$\perp$	$\perp$	
f_5	0	1	3	6	10	$\perp$	$\perp$	$\perp$	
f_6	0	1	3	6	10	15	$\perp$	$\perp$	
f_7	0	1	3	6	10	15	21	$\perp$	

Alternativ hier als Haskell Code:

```haskell
f n = if n==0 then 0 else n + f(n-1)

f0   n = undefined
f1   n = if n==0 then 0 else undefined
f2   n = if n==0 then 0 else n + (if (n-1)==0 then 0 else undefined)
f2'  n = if n==0 then 0 else if n==1 then 1 else undefined
f2'' 0 =  0 -- if-then-else wird schnell unübersichtlich,
f2'' 1 =  1 -- pattern-matching ist hier vermutlich einfacher
f2'' _ = undefined
f3   0 =  0
f3   1 =  1
f3   2 =  3
f3   _ = undefined
f4   0 =  0
f4   1 =  1
f4   2 =  3
f4   3 =  6
f4   _ = undefined

g :: (Integer -> Integer) -> Integer -> Integer
g x = \n -> if n==0 then 0 else n + x (n-1)

fn = iterate g f0
```

b) McCarthy 91-Funktion (siehe auch 03-18):

```
mc91 n | n > 100   = n - 10
       | otherwise = mc91(mc91(n+11))
```

LÖSUNGSVORSCHLAG:

Zur Verdeutlichung der Vorgehensweise schreiben wir hier erst noch mal die einzelnen Definitionen auf, welche Einführen, um die Rekursion zu eliminieren:

```
mcf0 n = undefined
mcf1 n | n > 100   = n - 10
       | otherwise = mcf0(mcf0(n+11))
mcf2 n | n > 100   = n - 10
       | otherwise = mcf1(mcf1(n+11))
mcf3 n | n > 100   = n - 10
       | otherwise = mcf2(mcf2(n+11))
mcf4 n | n > 100   = n - 10
       | otherwise = mcf3(mcf3(n+11))
 ...
```

Jetzt falten wir diese aus, um die einzelnen partiellen Funktionen direkt zu beschreiben.

```
mcf0  n = undefined
mcf1  n | n >  100  = n - 10
        | otherwise = undefined
mcf2  n | n >  100  = n - 10
        | n >= 100  = 91
        | otherwise = undefined
mcf3  n | n > 100   = n - 10
        | n >= 99   = 91
        | otherwise = undefined
mcf4  n | n > 100   = n - 10
        | n >= 98   = 91
        | otherwise = undefined
mcf5  n | n > 100   = n - 10
        | n >= 97   = 91
        | otherwise = undefined
 ...
```

Aufgrund des doppelten Aufrufs müssen wir jedoch Sprünge beachten:

```
 ...
mcf11 n | n > 100   = n - 10
        | n >= 91   = 91
        | otherwise = undefined
mcf12 n | n > 100   = n - 10
        | n >= 80   = 91
        | otherwise = undefined
mcf13 n | n > 100   = n - 10
        | n >= 69   = 91
        | otherwise = undefined
mcf14 n | n > 100   = n - 10
        | n >= 58   = 91
        | otherwise = undefined
...
mcf19 n | n > 100   = n - 10
        | n >=  3   = 91
        | otherwise = undefined
mcf20 n | n > 100   = n - 10
        | n >= -8   = 91
        | otherwise = undefined
mcf21 n | n > 100   = n - 10
        | n >=-19   = 91
        | otherwise = undefined
```

Damit können wir die McCarthy 91-Funktion auch als Fixpunkt folgender Iteration beschreiben:

```
mcg x = \n -> if n > 100 then n - 10 else x(x(n+11))
```

Die Funktion `mcf16` erhalten wir dann direkt durch `head $ drop 16 $ iterate mcg undefined`

A12-3 ***Ein Supremum*** Betrachten Sie die rekursive Definition `altl = True:False:altl`. Programmieren Sie eine Haskell Funktion `g`, so dass `(iterate g undefined) :: [[Bool]]` eine Liste ergibt, deren Elemente eine Kette von partiell definierten Werten bilden, deren Supremum `altl` ist, also `[True,False,True,False,..]`. *Hinweis:* Überlegen Sie sich zu Beginn wie die Kette aussieht, d.h. gegeben Sie die Anfangsglieder der Kette an.

LÖSUNGSVORSCHLAG:
Die Kette könnte z.B. so aussehen: $\bot \sqsubseteq$ `True:False:`$\bot \sqsubseteq$ `True:False:True:False:`$\bot \sqsubseteq \cdots$
In Haskell Notation: `[undefined, True:False:undefined, True:False:True:False:undefined, ..]`
Die Funktion g bekommt als Argument ein Element dieser Liste und soll einfach nur das jeweils nächste Element der Liste berechnen, d.h. einmal `True` und `False` hinzufügen:

```
galt :: [Bool] -> [Bool]
galt x = True:False:x
```

A12-4 ***Erhalt von Suprema*** Es sei $(X, \sqsubseteq)$ ein dcpo und $(x_i)_i$ mit $x_i \in X$ sei eine Kette, also $x_0 \sqsubseteq x_1 \sqsubseteq x_2 \sqsubseteq \cdots$. Das Supremum einer aufsteigenden Kette bezeichnen wir mit $\sup_i x_i$. Beweisen Sie, dass für jede andere Kette $(y_i)_i$ im gleichen dcpo mit mit $x_i = y_{i+1}$ für alle $i \in \mathbb{N}$ auch schon $\sup_i x_i = \sup_j y_j$ gilt.

Vereinfacht in Worten ausgedrückt: Beweisen Sie, dass das Supremum einer Kette unverändert bleibt, wenn die Kette um ein kleineres Element nach unten erweitert wird.

LÖSUNGSVORSCHLAG:
Nach der Eigenschaft der Anti-Symmetrie von $\sqsubseteq$ genügt es, wenn wir $\sup x_i \sqsubseteq \sup y_j$ und $\sup y_j \sqsubseteq \sup x_j$ beweisen.

Fall $\sup_j y_j \sqsubseteq \sup_i x_i$**:** Es genügt zu zeigen, dass $\sup_i x_i$ *eine* obere Schrank der Kette $(y_j)_j$ ist, denn da $\sup_j y_j$ *die kleinste* obere Schranke der Kette $(y_j)_j$ ist, folgt dann automatisch $\sup_j y_j \sqsubseteq \sup_i x_i$.

Wir müssen also für alle $j \in \mathbb{N}$ zeigen, dass $y_j \sqsubseteq \sup_i x_i$ gilt. Für $j > 0$ haben wird $y_j = x_{j-1}$ und damit auch schon $y_j \sqsubseteq \sup_i x_i$, denn für alle $j \in \mathbb{N}$ gilt $x_j \sqsubseteq \sup_i x_i$.

Für $j = 0$ beobachten wir: Es gilt $y_0 \subseteq y_1$ und $y_1 = x_0$ und $x_0 \sqsubseteq \sup_i x_i$. Aus der Transitivität von $\sqsubseteq$ folgt damit auch schon $y_0 \subseteq \sup_i x_i$ wie benötigt, was den Beweis dieses Falles vervollständigt.

Fall $\sup_i x_i \sqsubseteq \sup_j y_j$**:** Wegen $x_i = y_{i+1}$ gilt für alle $i \in \mathbb{N}$ auch $x_i \sqsubseteq \sup y_j$. Damit ist $\sup_j y_j$ *eine* obere Schranke der Kette $(x_i)_i$. Nach Definition ist $\sup_i x_i$ *die kleinste* aller oberen Schranken, somit folgt $\sup_i x_i \sqsubseteq \sup_j y_j$ wie benötigt.

Keine Abgabe von Hausübungen