Prof. Martin Hofmann, PhD
Dr. Steffen Jost

Ludwig-Maximilians-Universität München
Institut für Informatik

14. Juli 2015

12. Übung zur Vorlesung

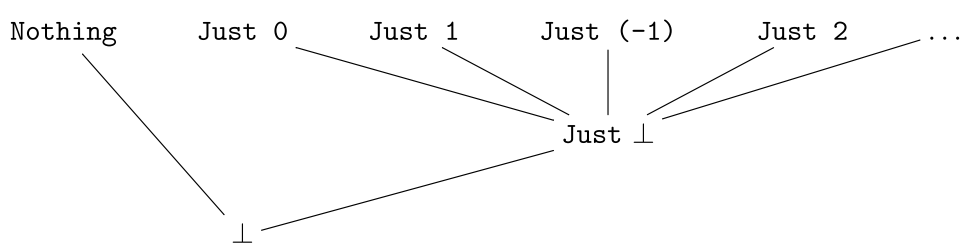
Programmierung und Modellierung

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.

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

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

[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.

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)`

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

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

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.

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.

Keine Abgabe von Hausübungen