Funktionales Programmieren

Teil 5

Carl Philipp Reh

Universität Siegen

24. November 2023

Auswertung

Wir betrachten folgende Beispielfunktion

```
square :: Int -> Int
square x = x * x
```

In welcher Reihenfolge wird folgender Ausdruck ausgewertet?

```
square (2 + 3)
```

Option 1 (Call by Value): Werte *zuerst* den Parameter `2 + 3` aus und ersetze dann die Definition von `square`:

```
square (2 + 3) = square 5 = 5 * 5 = 25
```

Option 2 (Call by Name): Ersetze *zuerst* die Definition von `square` und werte den Parameter aus, wenn nötig:

```
square (2 + 3) = (2 + 3) * (2 + 3)
               = 5 * (2 + 3) = 5 * 5 = 25
```

Auswertung

Nachteil bei Call by Name: Wir werten `2 + 3` zweimal aus.
Vorteil: Man kann sich möglicherweise die Auswertung des Parameters ganz sparen.

Haskell verwendet eine Variation von Call by Name, die Call by Need oder Lazy Evaluation genannt wird. Bei dieser werden alle Teilausdrücke, die aus demselben Ausdruck „entstanden“ sind, nur einmal ausgewertet:

```
square (2 + 3) = (2 + 3) * (2 + 3)
               = 5 * 5 = 25
```

Hier werden beide Teilausdrücke `2 + 3` „auf einmal“ ausgewertet.

Eine formale Beschreibung dieses „schrittweisen Auswertens“ nennt man auch *operationelle Semantik*. Diese ist vor allem dafür nützlich, einen Compiler zu implementieren, aber weniger nützlich, um über das „Verhalten“ eines Programms zu argumentieren.

Denotationelle Semantik

Bei denotationeller Semantik geht es darum, Ausdrücken mathematische Funktionen zuzuordnen. Beispielsweise würden wir gerne der Funktion `square` die mathematische Funktion $[\![\text{square}]\!] : \mathbb{Z} \to \mathbb{Z}$ mit $[\![\text{square}]\!](x) = x^2$ zuordnen.
Dass `Int` einen endlichen Wertebereich hat, ignorieren wir an der Stelle. Ein anderes Problem ist aber, dass es auch divergierende Ausdrücke in Haskell gibt. Zum Beispiel liefert folgende Funktion nie einen Wert:

```
undefined :: a -> a
undefined x = undefined x
```

Also liefert auch `square (undefined 2)` nie einen Wert.

Undefinierte Argumente

Betrachten wir folgende Definition von **and**:

```
and :: Bool -> Bool -> Bool
and x y = case x of
            False -> False
            True -> y
```

Was passiert bei folgendem Programm?

```
and (undefined True) True
```

Da **case** auf **undefined True** angewandt wird, terminiert das Programm nicht. Im Gegensatz dazu gilt

```
and False (undefined True) = False
```

weil **and** sofort **False** liefert, wenn das erste Argument **False** ist, ohne das zweite Argument auszuwerten.

Semantik von square

Wir benötigen für die Semantik von Funktionen also sowohl für die Parameter, als auch für die Ergebnisse noch den speziellen Wert „undefiniert“, den wir mit $\perp$ (Bottom) bezeichnen. Sei $\mathbb{Z}_\perp = \mathbb{Z} \cup \{\perp\}$. Für die Semantik von `square` können wir also $[\![\mathrm{square}]\!] : \mathbb{Z}_\perp \to \mathbb{Z}_\perp$ nehmen mit

$$[\![\mathrm{square}]\!](x) = \begin{cases} \perp & \text{falls } x = \perp, \\ x^2 & \text{falls } x \neq \perp. \end{cases}$$

Was allerdings denotationelle Semantik aufwändig macht, ist die Tatsache, dass wir Funktionen rekursiv definieren können und dass Haskell Funktionen höherer Ordnung hat.
Wir werden stetige Funktionen benötigen, um rekursiv definierten Funktionen eine Semantik zuordnen zu können, was etwas Vorbereitung braucht.

Partielle Ordnungen

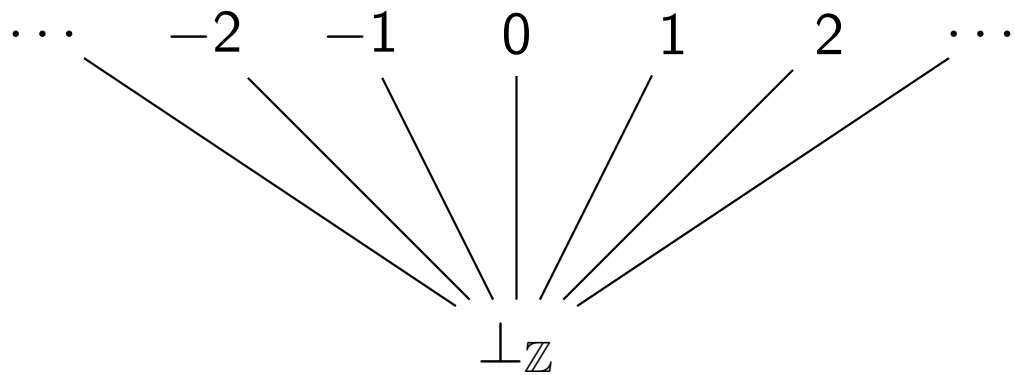
Ein Paar $(D, \sqsubseteq_D)$, wobei D eine Menge ist und $\sqsubseteq_D \subseteq D \times D$, heißt *partielle Ordnung*, wenn $\sqsubseteq_D$ Folgendes erfüllt:

- Reflexivität: Für alle $d \in D$ gilt $d \sqsubseteq_D d$.
- Antisymmetrie: Für alle $d, d' \in D$ gilt: Wenn $d \sqsubseteq_D d'$ und $d' \sqsubseteq_D d$, dann gilt $d = d'$. Alternativ: Es gibt keine $d \neq d' \in D$ mit $d \sqsubseteq_D d'$ und $d' \sqsubseteq_D d$.
- Transitivität: Für alle $d, d', d'' \in D$ gilt: Wenn $d \sqsubseteq_D d'$ und $d' \sqsubseteq_D d''$, dann gilt $d \sqsubseteq_D d''$.

Wir werden oft einfach nur D statt $(D, \sqsubseteq_D)$ und $\sqsubseteq$ statt $\sqsubseteq_D$ schreiben, wenn $\sqsubseteq_D$ aus dem Kontext bekannt ist.
Alle Ordnungen, die wir betrachten, haben die Intuition, dass $d \sqsubseteq_D d'$ gilt, wenn d „weniger definiert“ ist als d'.

Flache Ordnungen

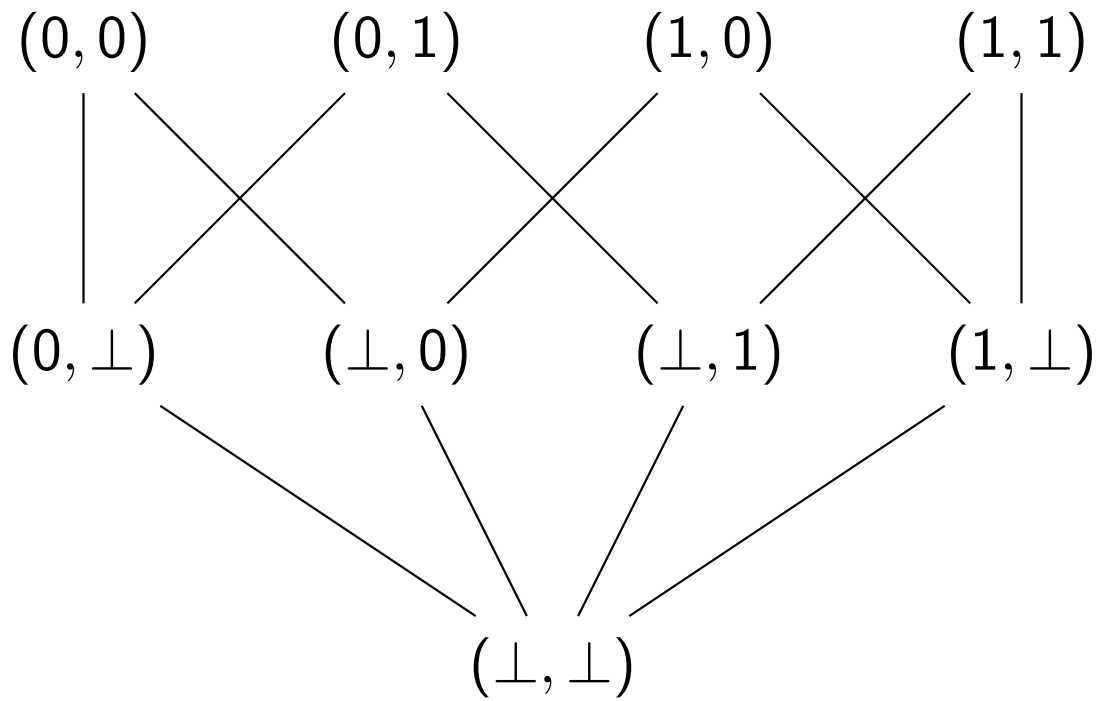
Für eine Menge D ist die *flache Ordnung* $(D_\perp, \sqsubseteq_{D_\perp})$ definiert als $D_\perp = D \uplus \{\perp_D\}$ und $d \sqsubseteq_{D_\perp} d'$ genau dann, wenn $d = \perp_D$ oder $d = d'$. Dies definiert eine partielle Ordnung.
Beispiel: $(\mathbb{Z}_\perp, \sqsubseteq_{\mathbb{Z}_\perp})$ ist die Ordnung, bei der gilt, dass $\perp_\mathbb{Z} \sqsubseteq_{\mathbb{Z}_\perp} d$ für alle $d \in \mathbb{Z}_\perp$. In diesem Sinne ist also $\perp_\mathbb{Z}$ „undefinierter“ als alle ganzen Zahlen, aber keine ganze Zahl ist „undefinierter“ als eine andere. Grafisch:



Wir schreiben oft einfach nur $\perp$ statt $\perp_D$, wenn D aus dem Kontext bekannt ist.

Produkt-Ordnung

Zu $n \geq 0$ partiellen Ordnungen $(D_1, \sqsubseteq_{D_1}), \ldots, (D_n, \sqsubseteq_{D_n})$ definieren wir die *Produkt-Ordnung* $(D_1 \times \cdots \times D_n, \sqsubseteq_{D_1 \times \cdots \times D_n})$ wie folgt: Es gilt $(d_1, \ldots, d_n) \sqsubseteq_{D_1 \times \cdots \times D_n} (d'_1, \ldots, d'_n)$ genau dann, wenn $d_i \sqsubseteq_{D_i} d'_i$ für alle $1 \leq i \leq n$.
Als Beispiel sei $\mathbb{Z}_2 = \{0, 1\}$ und wir betrachten die partielle Ordnung $((\mathbb{Z}_2)_\perp \times (\mathbb{Z}_2)_\perp, \sqsubseteq_{(\mathbb{Z}_2)_\perp \times (\mathbb{Z}_2)_\perp})$. Grafisch:



Die Produkt-Ordnung ist eine partielle Ordnung

Lemma 1

Die Produkt-Ordnung ist eine partielle Ordnung.

Beweis.

Reflexivität: Sei $(d_1, \ldots, d_n) \in D_1 \times \cdots \times D_n$. Es gilt $d_i \sqsubseteq_{D_i} d_i$ für alle $1 \leq i \leq n$, weil $\sqsubseteq_{D_i}$ reflexiv ist. Also folgt $(d_1, \ldots, d_n) \sqsubseteq (d_1, \ldots, d_n)$.
Antisymmetrie: Wenn $(d_1, \ldots, d_n) \sqsubseteq (d'_1, \ldots, d'_n)$ und $(d'_1, \ldots, d'_n) \sqsubseteq (d_1, \ldots, d_n)$, dann gilt $d_i \sqsubseteq_{D_i} d'_i$ und $d'_i \sqsubseteq_{D_i} d_i$ für alle $1 \leq i \leq n$. Weil $\sqsubseteq_{D_i}$ antisymmetrisch ist, gilt $d_i = d'_i$ für alle $1 \leq i \leq n$. Also folgt $(d_1, \ldots, d_n) = (d'_1, \ldots, d'_n)$.
Transitivität: Wenn $(d_1, \ldots, d_n) \sqsubseteq (d'_1, \ldots, d'_n)$ und $(d'_1, \ldots, d'_n) \sqsubseteq (d''_1, \ldots, d''_n)$, dann gilt $d_i \sqsubseteq_{D_i} d'_i$ und $d'_i \sqsubseteq_{D_i} d''_i$ für alle $1 \leq i \leq n$. Wegen Transitivität von $\sqsubseteq_{D_i}$ gilt auch $d_i \sqsubseteq_{D_i} d''_i$ für alle $1 \leq i \leq n$. Also folgt $(d_1, \ldots, d_n) \sqsubseteq (d''_1, \ldots, d''_n)$. □

Ordnung auf Funktionen

Seien D eine Menge und $(E, \sqsubseteq_E)$ eine partielle Ordnung. Wir definieren die partielle Ordnung $(D \to E, \sqsubseteq_{D \to E})$ wie folgt: Für $f, f' \colon D \to E$ gilt $f \sqsubseteq_{D \to E} f'$ genau dann, wenn für alle $d \in D$ gilt, dass $f(d) \sqsubseteq_E f'(d)$.
In gewissem Sinne ist dies analog zu Tupeln, wenn man sich eine Funktion $D \to E$ als ein $|D|$-stelliges Tupel mit Komponenten aus E vorstellt. $|D|$ kann allerdings unendlich sein.

Lemma 2

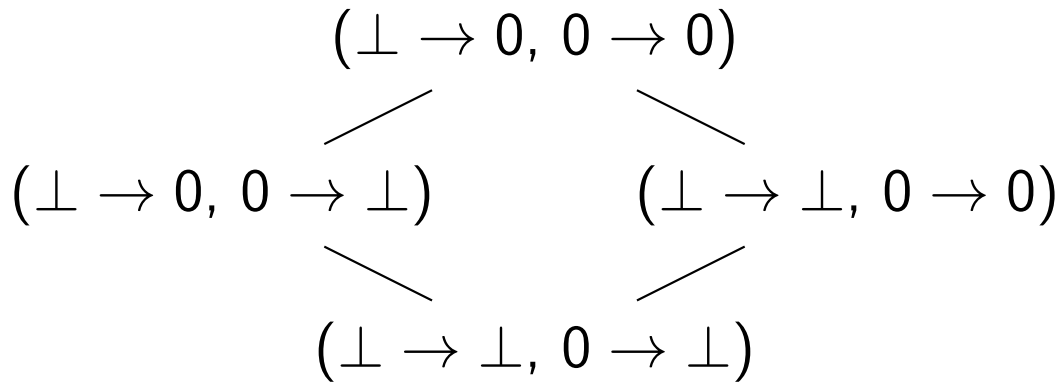
$(D \to E, \sqsubseteq_{D \to E})$ *ist eine partielle Ordnung.*

Beweis.

Übung. □

Ordnung auf Funktionen

Sei $\mathbb{Z}_1 = \{0\}$. Wir betrachten die Funktionen $(\mathbb{Z}_1)_\perp \to (\mathbb{Z}_1)_\perp$, für die wir folgende Ordnung erhalten:



Die Schreibweise $(\perp \to x,\ 0 \to y)$ bedeutet, dass $\perp$ auf x und 0 auf y abgebildet werden.
Die Funktion $(\perp \to \perp,\ 0 \to \perp)$ ist also „am wenigsten definiert“ und die Funktion $(\perp \to 0,\ 0 \to 0)$ ist „am meisten definiert“.

Monotone Funktionen

Seien $(D, \sqsubseteq_D)$ und $(E, \sqsubseteq_E)$ partielle Ordnungen. Eine Funktion $f : D \to E$ heißt *monoton*, wenn für alle $d, d' \in D$ mit $d \sqsubseteq_D d'$ gilt, dass $f(d) \sqsubseteq_E f(d')$.
Die Intuition ist, dass ein Argument, das „mehr definiert“ ist, nicht zu einem Ergebnis führen kann, das „weniger“ definiert ist.
In unserem vorherigen Beispiel ist die Funktion f mit $f(\bot) = 0$ und $f(0) = \bot$ also nicht monoton, weil $\bot \sqsubseteq 0$ gilt, aber $f(\bot) = 0 \not\sqsubseteq \bot = f(0)$. Diese Funktion lässt sich auch nicht in Haskell implementieren, weil die Implementierung testen müsste, ob ihr Argument divergiert.
Die anderen drei Funktionen sind monoton, da

- Identitäten $f(x) = x$ monoton sind, denn es gilt für alle $d \sqsubseteq d'$, dass $f(d) = d \sqsubseteq d' = f(d')$, und
- konstante Funktionen $f(x) = c$ monoton sind, denn es gilt für alle $d \sqsubseteq d'$, dass $f(d) = c \sqsubseteq c = f(d')$.