Motivation

Exzerpt von Aufgabenblatt 1:

> "Ein einfacher Editor kann in Haskell wie folgt realisiert werden:
>
> `type Editor = [Char]`
>
> Schreiben Sie eine Haskell-Rechenvorschrift `ersetze` mit der Signatur
>
> `ersetze :: Editor -> Int -> String -> String -> Editor`
>
> die angesetzt auf..."

Zwei naheliegende Fragen...

- Warum so wenige Klammern?
- Warum so viele Pfeile (`->`) und warum so wenige Kreuze ($\times$)? Warum nicht folgende Signaturzeile?

  "`ersetze :: (Editor x Int x String x String) -> Editor`"

  Beachte: Haskell-korrekt wäre (d.h. "," statt `x`)

  `ersetze :: (Editor,Int,String,String) -> Editor`

...die uns zum heutigen Thema führen

Und das wird sein: Mehr über Haskell, insbesondere über...

- Funktionen
  - ...und darüber wie man sie definieren/notieren kann
    - $\rightsquigarrow$ Notationelle Alternativen (siehe Vorlesungsteil 1)
    - $\rightsquigarrow$ **Funktionssignaturen, Funktionsausdrücke, Klammereinsparungsregeln**
    - $\rightsquigarrow$ *Abseitsregel und Layout-Konventionen*
  - *Klassifikation von Rekursionstypen*
  - *Anmerkungen zu Effektivität und Effizienz*
  - *Komplexitätsklassen*

Hinweis: Die kursiv hervorgehobenen Punkte beginnend mit "Abseitsregel und Layout-Konventionen" werden erst in der Vorlesung am 18.10.2007 besprochen.

Klammereinsparungsregeln in Funktionssignaturen

Konvention (von essentieller Bedeutung):

- Der *Typkonstruktor* $\rightarrow$ ist *rechtsassoziativ*!

Das bedeutet:

- Die Funktionssignatur

  ```
  f :: Int -> Float -> Int -> String -> Char
  ```

  steht abkürzend für

  ```
  f :: (Int -> (Float -> (Int -> (String -> Char))))
  ```

- Wann immer eine abweichende Klammerung intendiert ist, *muss* explizit geklammert werden!

  (*vgl. Klammereinsparungsregeln bei arithmetischen Ausdrücken*)

Funktionen und ihre Signaturen (1)

Zur Veranschaulichung, noch konkreter:

Die Signaturen der Funktionen `f`

```
f :: Int -> (Int -> Int)
```

(aufgrund der Klammereinsparungsregeln gleichbedeutend mit der ungeklammerten Kurzform `f :: Int -> Int -> Int`) und `g`

```
g :: (Int -> Int) -> Int
```

sind *grundsätzlich verschieden* und *unbedingt auseinanderzuhalten*!

Funktionen und ihre Signaturen (2)

Warum?

- `f` ist eine Funktion, die ganze Zahlen auf Abbildungen ganzer Zahlen in sich abbildet.

- `g` ist eine Funktion, die Abbildungen ganzer Zahlen in sich auf ganze Zahlen abbildet.

Zur Übung:

- Überlegen Sie sich, ob die Funktion $+$ (Addition auf ganzen Zahlen) dem Signaturschema von `f` oder dem von `g` folgt.

Funktionen und ihre Signaturen (3)

Ein weiteres Beispiel, noch konkreter und noch ein wenig komplexer...

Mit folgenden Deklarationen für `f` und `g`

```
f :: Int -> (Int -> Int -> Int)
f 1 = (+)
f 2 = (-)
f 3 = (*)
f _ = div

g :: (Int -> Int -> Int) -> Int
g h = h 6 3
```

...liefern die nachstehenden Aufrufe von `f` und `g` die angegebenen Resultate:

```
Main> f 1 2 3                              kurz fuer: (((f 1) 2) 3)
5

Main> f 3 2 3
6

Main> g (+)
9

Main> g (*)
18
```

Funktionen und ihre Signaturen (4)

Offenbar gilt: ...in `g` sind die Argumente 6 und 3 fest vorgegeben. Betrachte deshalb jetzt die folgende “Erweiterung” `k` von `g`, die das vermeidet:

```
k :: (Int -> Int -> Int) -> Int -> Int -> Int
k h x y = h x y
```

Beachte: ...aufgrund der Klammereinsparungsregeln gemäß der Rechtsassoziativität von $\rightarrow$ steht obige Deklaration von `k` abkürzend für:

```
k :: ((Int -> (Int -> Int)) -> (Int -> (Int -> Int)))
k h x y = h x y
```

Für `k` sind jetzt folgende Aufrufe mit variablen Argumenten möglich:

```
Main> k (*) 3 5
15

Main> k (+) 17 4
21

Main> k div 42 5
8
```

Funktionen und ihre Signaturen (5)

Zur Übung:

Vergleichen Sie die Deklaration der Funktion `f`

```
f :: Int -> (Int -> Int -> Int)
f 1 = (+)
f 2 = (-)
f 3 = (*)
f _ = div
```

...mit der Deklaration ihrer naheliegenden “Umkehrung” `g`:

```
g :: (Int -> Int -> Int) -> Int
g (+) = 1
g (-) = 2
g (*) = 3
g div = 42
g _   = 99
```

- Was beobachten Sie, wenn Sie die Funktionen `f` und `g` aufrufen?
- Haben Sie (schon) eine Erklärung dafür?

Funktionen und ihre Signaturen (6)

Bleiben Sie auch an folgender Frage dran...

- Warum möglicherweise sind die Klammereinsparungsregeln für $\rightarrow$

  ```
  f :: Int -> Int -> Int -> Int
  ```

  zugunsten der *Rechtsassoziativität* von $\rightarrow$

  ```
  f :: (Int -> (Int -> (Int -> Int)))
  ```

  und nicht der *Linksassoziativität* gefallen?

  ```
  f :: (((Int -> Int) -> Int) -> Int)
  ```

Funktionen und ihre Signaturen (7)

In jedem Falle gilt:

Die Einsicht in den Unterschied

- von

  ```
  f :: Int -> Int -> Int -> Int
  ```

  ...aufgrund der *Rechtsassoziativität* von $\rightarrow$ abkürzend und gleichbedeutend mit der vollständig, aber nicht überflüssig geklammerten Version

  ```
  f :: (Int -> (Int -> (Int -> Int)))
  ```

- und von

  ```
  f :: (((Int -> Int) -> Int) -> Int)
  ```

ist *essentiell* und von absolut *zentraler* Bedeutung!

Funktionen und ihre Signaturen (8)

Bewusst pointiert...

Ohne diese Einsicht ist erfolgreiche Programmierung (speziell) im funktionalen Paradigma

- nicht möglich
- oder allenfalls Zufall!

Bestandsaufnahme (1)

- Bis jetzt:
  ...Konzentration auf Funktionsdeklarationen und ihre *Signaturen* bzw. *Typen*

- Ab jetzt:
  ...Konzentration auf *Funktionsterme* und ihre *Signaturen* bzw. *Typen*

Bestandsaufnahme (2)

Tatsache:

Wir sind gewohnt, mit Ausdrücken der Art

```
add 2 3
```

umzugehen. (Auch wenn wir gewöhnlich `2+3` statt `add 2 3` schreiben.)

Frage:

- Warum könnte es sinnvoll sein, auch mit (*scheinbar unvollständigen*) Ausdrücken wie

  ```
  add 2
  ```

  umzugehen?

- Entscheidend für die Antwort: Können wir einem Ausdruck wie `add 2` sinnvoll eine Bedeutung geben und wenn ja, welche?

Funktionsterme und ihre Typen (1)

Betrachten wir die Funktion `add` zur Addition ganzer Zahlen noch einmal im Detail:

```
add :: Int -> Int -> Int
add m n = m+n
```

Dann sind die Ausdrücke `add`, `add 2` und `add 2 3` von den Typen:

```
add :: Int -> Int -> Int
add 2 :: Int -> Int
add 2 3 :: Int
```

Funktionsterme und ihre Typen (2)

Erinnerung:

```
add :: Int -> Int -> Int
```

entspricht wg. vereinbarter Rechtsassoziativität von $\rightarrow$

```
add :: Int -> (Int -> Int)
```

Somit *verbal* umschrieben:

- **`add :: Int -> Int -> Int`**
  ...bezeichnet eine Funktion, die ganze Zahlen auf Funktionen von ganzen Zahlen in ganze Zahlen abbildet (*Rechtsassoziativität von* $\rightarrow$!).

- **`add 2 :: Int -> Int`**
  ...bezeichnet eine Funktion, die ganze Zahlen auf ganze Zahlen abbildet.

- **`add 2 3 :: Int`**
  ...bezeichnet eine ganze Zahl (nämlich 5).

Funktionsterme und ihre Typen (3)

Damit haben wir eine Antwort auf unsere Ausgangsfrage...

- Warum könnte es sinnvoll sein, auch mit (*scheinbar unvollständigen*) Ausdrücken wie

  ```
  add 2
  ```

  umzugehen?

- Entscheidend für die Antwort: Können wir einem Ausdruck wie `add 2` sinnvoll eine Bedeutung geben und wenn ja, welche?

Nämlich:

Es ist sinnvoll, mit Ausdrücken der Art `add 2` umzugehen, weil

- wir ihnen sinnvoll eine Bedeutung zuordnen können!

- im Falle von `add 2`:
  ...`add 2` bezeichnet eine Funktion auf ganzen Zahlen, die angewendet auf ein Argument dieses Argument um 2 erhöht als Resultat liefert.

Funktionsterme und ihre Typen (4)

Betrachte auch folgendes Beispiel von vorhin unter dem neuen Blickwinkel auf Funktionsterme und ihre Typen:

```
k :: (Int -> Int -> Int) -> Int -> Int -> Int
k h x y = h x y
```

Dann gilt:

```
k :: (Int -> Int -> Int) -> Int -> Int -> Int
k add :: Int -> Int -> Int
k add 2 :: Int -> Int
k add 2 3 :: Int
```

Zur Übung:

- Ausprobieren! In Hugs lässt sich mittels des Kommandos `:t <Ausdruck>` der Typ eines Ausdrucks bestimmen!

  Bsp.: `:t k add 2` liefert `k add 2 :: Int -> Int`

Funktionsterme und ihre Typen (5)

Beachte:

Der Ausdruck (Funktionsterm)

```
k add 2 3
```

steht kurz für

```
(((k add) 2) 3)
```

Analog stehen die Ausdrücke (Funktionsterme)

```
k add
k add 2
```

kurz für

```
(k add)
((k add) 2)
```

Funktionsterme und ihre Typen (6)

Beobachtung (anhand des vorigen Beispiels):

- Funktionen in Haskell sind grundsätzlich *einstellig*!
- Wie die Funktion `k` zeigt, kann dieses Argument komplex sein, bei `k` z.B. eine Funktion, die ganze Zahlen auf Funktionen ganzer Zahlen in sich abbildet.

Beachte:

Die Sprechweise, Argument der Funktion `k` sei eine zweistellige Funktion auf ganzen Zahlen, ist *lax* und *unpräzise*, gleichwohl (aus Gründen der Einfachheit und Bequemlichkeit) üblich.

Funktionsterme und ihre Typen (7)

Konsequenz aus voriger Beobachtung:

- Wann immer man nicht durch Klammerung etwas anderes erzwingt, ist (aufgrund der vereinbarten Rechtsassoziativität des Typoperators $\rightarrow$) das "eine" Argument der in Haskell grundsätzlich einstelligen Funktionen von demjenigen Typ, der links vor dem ersten Vorkommen des Typoperators $\rightarrow$ in der Funktionssignatur steht.

- Wann immer dies nicht erwünscht ist, muss dies durch explizite Klammerung in der Funktionssignatur ausgedrückt werden.

Funktionsterme und ihre Typen (8)

Beispiele:

- *Keine Klammerung* ($\leadsto$ Konvention greift!)

  ```
  f :: Int -> Tree -> Graph -> ...
  ```

  f ist einstellige Funktion auf ganzen Zahlen, nämlich `Int`, die diese abbildet auf...

- *Explizite Klammerung* ($\leadsto$ Konvention aufgehoben, wo gewünscht!)

  ```
  f :: (Int -> Tree) -> Graph -> ...
  ```

  f ist einstellige Funktion auf Abbildungen von ganzen Zahlen auf Bäume, nämlich `Int -> Tree`, die diese abbildet auf...

Hinweis: Wie wir Bäume und Graphen in Haskell definieren können, lernen wir bald.

Funktionsterme und ihre Typen (9)

Auch noch zu...

- ...
- Wann immer dies nicht erwünscht ist, muss dies durch explizite Klammerung in der Funktionssignatur erzwungen werden.

Beispiele:

- *Keine Klammerung*

```
f :: Int -> Tree -> Graph -> ...
```

  f ist einstellige Funktion auf ganzen Zahlen, nämlich `Int`, die diese abbildet auf...

- *Explizite Klammerung*

```
f :: (Int,Tree) -> Graph -> ...
```

  f ist einstellige Funktion auf Paaren aus ganzen Zahlen und Bäumen, nämlich `(Int,Tree)`, die diese abbildet auf...

Funktionsterme und ihre Typen (10)

Noch einmal zurück zum Beispiel der Funktion `k`:

```
k :: (Int -> Int -> Int) -> Int -> Int -> Int
```

...`k` ist eine einstellige Funktion, die eine zweistellige Funktion auf ganzen Zahlen als Argument erwartet (*lax!*) und auf eine Funktion abbildet, die ganze Zahlen auf Funktionen ganzer Zahlen in sich abbildet.

Zur Deutlichkeit die Signatur von `k` auch noch einmal vollständig, aber nicht überflüssig geklammert:

```
k :: ((Int -> (Int -> Int)) -> (Int -> (Int -> Int)))
```

Funktionsterme und ihre Typen (11)

Das Beispiel von `k` fortgesetzt:

```
k add :: Int -> Int -> Int
```

...`k add` ist eine einstellige Funktion, die ganze Zahlen auf Funktionen ganzer Zahlen in sich abbildet.

Zur Deutlichkeit auch hier noch einmal vollständig, aber nicht überflüssig geklammert:

```
(k add) :: (Int -> (Int -> Int))
```

Funktionsterme und ihre Typen (12)

Das Beispiel von `k` weiter fortgesetzt:

```
k add 2 :: Int -> Int
```

...`k add 2` ist eine einstellige Funktion, die ganze Zahlen in sich abbildet.

Zur Deutlichkeit auch hier wieder vollständig, aber nicht überflüssig geklammert:

```
((k add) 2) :: (Int -> Int)
```

Funktionsterme und ihre Typen (13)

Das Beispiel von `k` abschließend fortgesetzt:

```
k add 2 3 :: Int
```

`k add 2 3` bezeichnet ganze Zahl; in diesem Falle `5`.

Zur Deutlichkeit auch dieser Funktionsterm vollständig, aber nicht überflüssig geklammert:

```
(((k add) 2) 3) :: Int
```

Wichtige Vereinbarungen in Haskell

Wenn in Haskell durch Klammerung nichts anderes ausgedrückt wird, gilt für

- Funktionssignaturen *Rechtsassoziativität*, d.h.

  ```
  k :: (Int -> Int -> Int) -> Int -> Int -> Int
  ```

  steht für

  ```
  k :: ((Int -> (Int -> Int)) -> (Int -> (Int -> Int)))
  ```

- Funktionsterme *Linksassoziativität*, d.h.

  ```
  k add 2 3 :: Int
  ```

  steht für

  ```
  (((k add) 2) 3) :: Int
  ```

als vereinbart!

Zum Abschluss des Signaturthemas (1)

Frage:

- Warum mag uns ein Ausdruck wie

  ```
  add 2
  ```

  "unvollständig" erscheinen?

Zum Abschluss des Signaturthemas (2)

...weil wir im Zusammenhang mit der Addition tatsächlich weniger an Ausdrücke der Form

```
add 2 3
```

als vielmehr an Ausdrücke der Form

```
add' (2,3)
```

gewohnt sind!

Erinnern Sie sich?

$$+ : \mathbb{Z} \times \mathbb{Z} \rightarrow \mathbb{Z}$$

Zum Abschluss des Signaturthemas (3)

Der Unterschied liegt in den Signaturen der Funktionen `add` und `add'`:

```
add  :: Int -> (Int -> Int)

add' :: (Int,Int) -> Int
```

Mit diesen Signaturen von `add` und `add'` sind einige Beispiele...

- *korrekter* Aufrufe:

```
add 2 3              => 5 :: Int
add' (2,3)           => 5 :: Int
add 2                     :: Int -> Int
```

- *inkorrekter* Aufrufe:

```
add (2,3)
add' 2 3     -- beachte: add' 2 3 steht kurz fuer (add' 2) 3
add' 2
```

Zum Abschluss des Signaturthemas (4)

Mithin...

- ...die Funktionen `+` und `add'` sind echte *zweistellige* Funktionen

wohingegen...

- ...die Funktion `add` einstellig ist und nur aufgrund der Klammereinsparungsregeln scheinbar ebenfalls “zweistellige” Aufrufe zulässt:

  ```
  add 17 4
  ```

  Aber: `add 17 4` steht kurz für `(add 17) 4`. Die geklammerte Variante macht deutlich: Ein Argument nach dem anderen und nur eines zur Zeit...

Fazit zum Signaturthema (1)

Wir müssen nicht nur sorgfältig

- zwischen

  ```
  f :: Int -> Int -> Int
  ```

  ...aufgrund der *Rechtsassoziativität* von $\rightarrow$ abkürzend und gleichbedeutend ist mit

  ```
  f :: Int -> (Int -> Int)
  ```

- und

  ```
  f :: (Int -> Int) -> Int
  ```

unterscheiden, sondern ebenso sorgfältig auch

- zwischen

  ```
  f :: (Int,Int) -> Int
  ```

- und

  ```
  f :: Int -> (Int,Int)
  ```

und nicht zuletzt zwischen allen vier Varianten insgesamt!

Fazit zum Signaturthema (2)

Mithin, schreiben Sie

```
f :: Int -> Int -> Int
```

nur, wenn Sie auch wirklich

```
f :: Int -> (Int -> Int)
```

meinen und nicht etwa

```
f :: (Int -> Int) -> Int
```

oder

```
f :: (Int,Int) -> Int
```

oder

```
f :: Int -> (Int,Int)
```

Es macht einen Unterschied!

Und deshalb die Bitte:

- Gehen Sie die vorausgegangenen Beispiele noch einmal Punkt für Punkt durch und vergewissern Sie sich, dass Sie sie im Detail verstanden haben.

Das ist wichtig, weil...

- dieses Verständnis und der aus diesem Verständnis heraus mögliche kompetente und selbstverständliche Umgang mit komplexen Funktionssignaturen und Funktionstermen essentiell für alles weitere ist!

Ein kurzer Ausblick

Wir werden auf die Unterschiede und die Vor- und Nachteile von Deklarationen in der Art von

```
add :: Int -> (Int -> Int)
```

und

```
add' :: (Int,Int) -> Int
```

im Verlauf der Vorlesung unter den Schlagwörtern *Funktionen höherer Ordnung*, *Currifizierung*, *Funktionen als "first class citizens*" wieder zurückkommen.

Behalten Sie die Begriffe im Hinterkopf und blättern Sie zu gegebener Zeit in Ihren Unterlagen wieder hierher zurück.

Zum ersten Aufgabenblatt...

- ...erhältlich seit 09.10.2007 im Web unter folgender URL
  `http://www.complang.tuwien.ac.at/knoop/fp185161_ws0708.html`

- Abgabe: Di, den 16.10.2007, 15:00 Uhr

- Zweitabgabe: Di, den 23.10.2007, 15:00 Uhr

Vorschau:
Ausgabe des...

- zweiten Aufgabenblatts: Di, den 16.10.2007
  ...Abgabetermine: Di, 23.10.2007, und Di, 30.10.2007

- dritten Aufgabenblatts: Di, den 23.10.2007
  ...Abgabetermine: Di, 30.10.2007, und Di, 06.11.2007

Vorschau auf die nächsten Vorlesungstermine...

- Do, 18.10.2007, Vorlesung von 16:30 Uhr bis 18:00 Uhr im Radinger-Hörsaal
- *Do, 25.10.2007: Keine Vorlesung*
- Di, 30.10.2007: Vorlesung von 13:00 Uhr s.t. bis 14:00 Uhr im Informatik-Hörsaal
- *Do, 01.11.2006: Keine Vorlesung! (Allerheiligen)*
- Do, 08.11.2007, Vorlesung von 16:30 Uhr bis 18:00 Uhr im Radinger-Hörsaal
- Do, 15.11.2007, Vorlesung von 16:30 Uhr bis 18:00 Uhr im Radinger-Hörsaal