Funktionale Programmierung

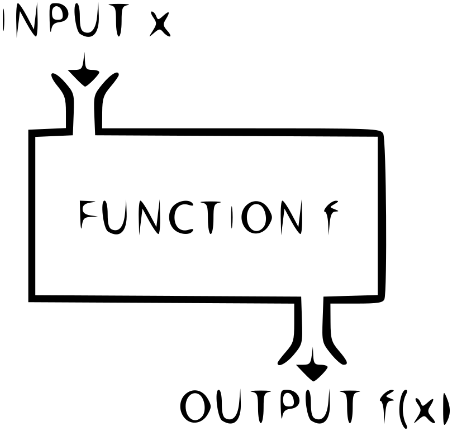
Funktionen und Typen II

Woche	Thema	Praktika
8	Organisatorisches, historisches, Begriff der funktionalen Programmierung, Einführung F#	1
9	Funktionen und Typen I: Werte, Unions (Listen), Produkte	2
10	Funktionen und Typen II: Options, Funktionstyp, "partial application", Currying	2,3
11	Funktionen und Typen III: Kombinatoren und höhere Funktionen	3
12	Rekursion I: Formen der Rekursion	4
13	Rekursion II: Fixpunkte	4,5
14	Rekursion III: Rekursion und Compiler Optimierungen	5

Programm

Detail

- Funktionen
  - Der Funktionstyp
  - Funktionssignaturen als "Dokumentation"
  - Currying und partielle Anwendung
- Partielle Funktionen und optionale Werte
  - Partielle Funktionen
  - Options


INPUT x
FUNCTION f
OUTPUT f(x)

Eine Funktion stellt jeder Eingabe x genau eine Ausgabe $f(x)$ gegenüber.
Eine reine Funktion hat keinerlei Nebeneffekte.

Eine reine Funktion:

```
let f x = 3 * x
```

Keine reine Funktion:

```
let f x =
    if x = 0 then failwith "Bla"
    else "OK"
```

Funktionstypen werden mit dem Typkonstruktor $\rightarrow$ dargestellt. In der Mengenanalogie ergibt sich folgende Entsprechung:

Ein Ausdruck von der Form

$$Typ_1 \rightarrow Typ_2$$

entspricht der Menge

$$\{f \mid f : Typ_1 \rightarrow Typ_2\}.$$

Die Meldung "`f:a -> b`" ist somit als "`f` ist eine Funktion von `a` nach `b`" zu lesen.

Der Pfeil $\rightarrow$ wird bei Rückgaben der FSI rechtsassoziativ gelesen.

Die Rückmeldung

```
f: int -> int -> int
```

ist beispielsweise als

```
f: int -> (int -> int)
```

zu verstehen.

Aufgabe

Bestimmen Sie (ohne FSI) den Typ der Funktion `x`

```
let x y z = y (z y)
```

Aufgabe

Bestimmen Sie (ohne FSI) den Typ der Funktion `x`

```
let x y z = y (z y)
```

```
y : a -> b
z : (a -> b) -> c // c muss = a weil y : a -> ..
x : (a -> b) -> ((a -> b) -> a) -> b
```

Definition

Funktionen deren Eingabe aus mehreren Argumenten besteht, nennt man mehrstellige Funktionen.

Beispiele mehrstelliger Funktion:

$$add(x, y) = x + y$$

$$max(x, y, z) = \begin{cases} x & \text{falls } x \geq y, z \\ y & \text{falls } y \geq x, z \\ z & \text{sonst} \end{cases}$$

$$avg(x_1, \ldots, x_n) = \frac{1}{n} \sum_{i=1}^{n} x_i$$

Zwei Sichtweisen auf mehrstellige Funktionen:

[1] Wegen der Rechtsassoziativität werden die Klammern typischerweise nicht geschrieben.

Zwei Sichtweisen auf mehrstellige Funktionen:

- In der ersten Variante versteht man eine *n*-stellige Funktion als Funktion, die *n*-Tupel als Eingabewerte akzeptiert. Eine mehrstellige Funktion nach dieser Auffassung hat dann einen Typ von der Form:

```
(a1 * a2 * ..* an) -> b
```

[1] Wegen der Rechtsassoziativität werden die Klammern typischerweise nicht geschrieben.

Zwei Sichtweisen auf mehrstellige Funktionen:

- In der ersten Variante versteht man eine n-stellige Funktion als Funktion, die n-Tupel als Eingabewerte akzeptiert. Eine mehrstellige Funktion nach dieser Auffassung hat dann einen Typ von der Form:

```
(a1 * a2 * ..* an) -> b
```

- In der zweiten Variante versteht man eine n-Stellige Funktion als Funktion, die $n-1$-stellige Funktionen zurückgibt. Eine mehrstellige Funktion nach dieser Auffassung hat dann einen Typ von der Form[1]:

```
a1 -> (a2 ->( ... -> (an -> b)..)
```

[1] Wegen der Rechtsassoziativität werden die Klammern typischerweise nicht geschrieben.

Beide Sichtweisen auf mehrstellige Funktionen werden in F# direkt unterstützt:

```
let add (x,y) = x + y
let max (x,y,z) = max x (max y z)
let avg (x,y,z,u) = (x + y + z + u) / 4
```

```
let add x y = x + y
let max x y z = max x (max y z)
let avg x y z u = (x + y + z + u) / 4
```

Unter Currying und Uncurrying versteht man das Übersetzen zwischen den genannten Ansätzen.

Informell lassen sich diese Übersetzungen wie folgt als Funktionen beschreiben:

```
curry f = fun x1 -> (fun x2 -> (..(f (x1,..,xn)))
```

```
uncurry f = fun (x1,..,xn) -> f x1 x2 .. xn
```

Aufgabe

Implementieren Sie Currying

```
curry: ('a * 'b -> 'c) -> 'a -> 'b -> 'c
```

und Uncurrying

```
uncurry: ('a -> 'b -> 'c) -> 'a * 'b -> 'c
```

für zweistellige Funktionen.

Aufgabe

Implementieren Sie Currying

```
curry: ('a * 'b -> 'c) -> 'a -> 'b -> 'c
```

und Uncurrying

```
uncurry: ('a -> 'b -> 'c) -> 'a * 'b -> 'c
```

für zweistellige Funktionen.

```
let rec curry f x y = f (x,y)
let uncurry f (x,y) = f x y
```

Partielle Anwendung bedeutet eine “curried” Funktion nicht “erschöpfend” mit Argumenten zu versehen:

```
let plus4 = (+) 4
```

Partielle Anwendung bedeutet eine “curried” Funktion nicht “erschöpfend” mit Argumenten zu versehen:

```
let plus4 = (+) 4
```

Bemerkungen zur partiellen Anwendung

- Partielle Anwendung (partial application) ist die ganz normale Anwendung (in der zweiten Sichtweise) von mehrstellige Funktionen.

Partielle Anwendung bedeutet eine “curried” Funktion nicht “erschöpfend” mit Argumenten zu versehen:

```
let plus4 = (+) 4
```

Bemerkungen zur partiellen Anwendung

- Partielle Anwendung (partial application) ist die ganz normale Anwendung (in der zweiten Sichtweise) von mehrstellige Funktionen.
- Partielle Anwendung hat nichts mit partiellen Funktionen zu tun.

Partielle Anwendung bedeutet eine “curried” Funktion nicht “erschöpfend” mit Argumenten zu versehen:

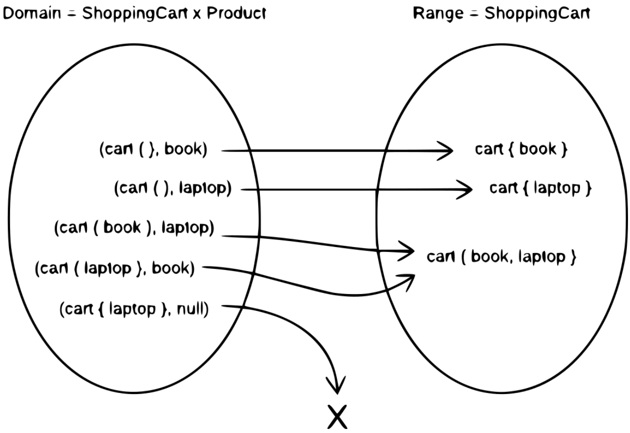
```
let plus4 = (+) 4
```

Bemerkungen zur partiellen Anwendung

- Partielle Anwendung (partial application) ist die ganz normale Anwendung (in der zweiten Sichtweise) von mehrstellige Funktionen.
- Partielle Anwendung hat nichts mit partiellen Funktionen zu tun.
- Partielle Anwendung ist ein Mittel der funktionalen Programmierung um “generischen” Code zu erzeugen.

Funktionen

Partielle Funktionen und optionale Werte



Definition

Eine partielle Funktion $f : X \nrightarrow Y$ ist eine Funktion $f : X' \to Y$, wobei $X' \subseteq X$.

Definition

Eine partielle Funktion $f : X \nrightarrow Y$ ist eine Funktion $f : X' \to Y$, wobei $X' \subseteq X$.

- Eine partielle Funktion gibt (eventuell) für gewisse Eingaben keinen Funktionswert zurück.

Definition

Eine partielle Funktion $f : X \nrightarrow Y$ ist eine Funktion $f : X' \to Y$, wobei $X' \subseteq X$.

- Eine partielle Funktion gibt (eventuell) für gewisse Eingaben keinen Funktionswert zurück.
- Partielle Funktionen sind in der Mathematik und insbesondere in der Informatik häufig. Beispiel $f(x, y) = \frac{x}{y}$.

Oft stellen funktionale Programmiersprachen einen expliziten Typ zur Modellierung von optionalen/partiellen Rückgabewerten bereit[2]. Der Option-Typ in F# kann durch einen einfachen Summentyp realisiert werden:

```
type 'a Option =
    | Some of 'a
    | None
```

[2] F#,Scala: Option, Haskell: Maybe, ..

Der Option-Type wird in F# z.B. wie folgt eingesetzt um partielle Funktionen zu modellieren:

```
let parse (x: string) =
    try Some <| int x
    with _ -> None
```

Der Option-Type im Vergleich zu `Null`-Referenzen:

Der Option-Type im Vergleich zu `Null`-Referenzen:

Explizität:
Durch Verwendung eines Option-Types wird explizit gemacht, dass eine gegebene Funktion partiell ist (Types as Documentation).

Der Option-Type im Vergleich zu `Null`-Referenzen:

Type-Safety:

- Unterschied zwischen einer `Null`-Referenz und einem validen Objekt sind für das Typensystem "unsichtbar".
- `Null`-Referenzen machen im Kontext von funktionalen Sprachen wenig Sinn, Werte sind unveränderlich und Namen stehen nicht für Speicherbereiche.
- `Null`-Referenzen "zerstören" referenzielle Transparanz, weil eine `NullReferenceException` einen Seiteneffekt darstellt.

Der Option-Type im Vergleich zu `Null`-Referenzen:

Komponierbarkeit:

```
let f x =
  x |> String.map (function '1' -> '2' | y -> y)
    |> String.filter (fun x -> x<>'a')
    |> parse
    |> Option.map (fun x -> x * x)
    |> Option.map (printfn "%A")
```

Aufgabe

Implementieren Sie die Funktion `div: int -> int Option`, entsprechend der Definition

$$div(x, y) = \begin{cases} \frac{x}{y} & \text{falls } y \neq 0 \\ \text{undefiniert} & \text{sonst.} \end{cases}$$

"Komponieren" Sie die Funktion `div 5` mit der Funktion `sqr: int -> int` um die Funktion $p(x) = (\frac{5}{x})^2$ zu deklarieren.

Aufgabe

Implementieren Sie die Funktion `div: int -> int Option`, entsprechend der Definition

$$div(x, y) = \begin{cases} \frac{x}{y} & \text{falls } y \neq 0 \\ \text{undefiniert} & \text{sonst.} \end{cases}$$

"Komponieren" Sie die Funktion `div 5` mit der Funktion `sqr: int -> int` um die Funktion $p(x) = (\frac{5}{x})^2$ zu deklarieren.

```
let div x = function
    | 0 -> None
    | y -> Some (x/y)

let p = div 5 >> Option.map (fun x -> x*x)
```