

9.4 Funktoren

Da in **Ocaml** fast alles höherer Ordnung ist, wundert es nicht, dass es auch Strukturen höherer Ordnung gibt: die **Funktoren**.

- Ein Funktor bekommt als Parameter eine Folge von Strukturen;
- der Rumpf eines Funktors ist eine Struktur, in der die Argumente des Funktors verwendet werden können;
- das Ergebnis ist eine neue Struktur, die abhängig von den Parametern definiert ist.

Wir legen zunächst per Signatur die Eingabe und Ausgabe des Funktors fest:

```
module type Enum = sig
  type enum
  val null : enum
  val incr : enum -> enum
  val int_of_enum : enum -> int
end

module type Counter = sig
  type counter
  val incCounter : unit -> unit
  val getCounter : unit -> counter
  val int_of_counter : counter -> int
end

...
```

```

...
# module GenCounter (Enum : Enum) : Counter =
  struct
    open Enum
    type counter = enum
    let count = ref null
    let incCounter() = count := incr(!count)
    let getCounter() = !count
    let int_of_counter x = int_of_enum x
  end;;
module GenCounter : functor (Enum : Enum) -> Counter

```

Jetzt können wir den Funktor auf eine Struktur [anwenden](#) und erhalten eine neue Struktur ...

```

module PlusEnum = struct
  type enum = int
  let null = 0
  let incr x = x+1
  let int_of_enum x = x
end

# module PlusCounter = GenCounter (PlusEnum);;
module PlusCounter : Counter

# PlusCounter.incCounter ();
  PlusCounter.incCounter ();
  PlusCounter.int_of_counter (PlusCounter.getCounter ());;
- : int = 2

```

Eine erneute Anwendung des Funktors erzeugt eine **neue Struktur**:

```
module MalEnum = struct
  type enum = int
  let null = 1
  let incr x = 2*x
  let int_of_enum x = x
end

# module MalCounter = GenCounter (MalEnum);;
module MalCounter : Counter

# MalCounter.incCounter();
  MalCounter.incCounter();
  MalCounter.int_of_counter (MalCounter.getCounter());;
- : int = 4
```

Mit einem Funktor kann man sich auch **mehrere Instanzen** des gleichen Moduls erzeugen:

```
# module CountApples = GenCounter (PlusEnum);;
module CountApples : Counter
# module CountPears = GenCounter (PlusEnum);;
module CountPears : Counter
...

```

```
...
# CountApples.incCounter();;
- : unit = ()
# CountApples.incCounter();;
- : unit = ()
# CountPears.incCounter ();;
- : unit = ()
# CountApples.int_of_counter (CountApples.getCounter());;
- : int = 2
# CountPears.int_of_counter (CountPears.getCounter());;
- : int = 1
```

9.5 Constraints

Manchmal möchten wir zwei Typen, die in verschiedenen Strukturen definiert sind, [identifizieren](#):

Beispiel:

```
module type PrintCounter = sig
  type counter
  val incCounter : unit -> unit
  val getCounter : unit -> counter
  val int_of_counter : counter -> int
  val printCounter : unit -> unit
end
```

Nehmen wir an, wir hätten eine weitere Signatur:

```
module type Printable = sig
  type printable
  val print : printable -> unit
end
```

Wir wollen einen Funktor definieren, der eine Struktur produziert, die zählen und den Zählerstand ausdrucken kann:

```
module GenPrintCounter (E : Enum) (P : Printable)
  : PrintCounter = struct
  module Count = GenCounter (E)
  include Count
  let printCounter() = P.print (getCounter())
end
```

Leider bekommen wir den Fehler:

```
# module GenPrintCounter (E : Enum) (P : Printable)
                                : PrintCounter = struct
  module Count = GenCounter (E)
  include Count
  let printCounter() = P.print (getCounter())
end
```

this expression has type `counter = GenCounter(E).counter`
but is here used with type `P.printable`

Der Compiler kann nicht wissen, dass der Typ `Print.printable`
gleich `GenCounter(E).counter` sein soll :-)

... das muss man ihm durch **Constraints** mitteilen:

```
module GenCounter (E : Enum) : Counter
    with type counter = E.enum =
  struct open E
    type counter = enum
    let count = ref null
    let incCounter() = count := incr(!count)
    let getCounter() = !count
    let int_of_counter x = int_of_enum x
  end
  ...
```

Der Typ `counter` in der Rückgabe des Funktors `genCounter` soll gleich dem Typ `enum` des Arguments sein :-)

Außerdem soll der Typ `printable` des zweiten Arguments gleich dem Typ `enum` des ersten Arguments sein ...

```
module GenPrintCounter (E : Enum)
    (P : Printable
        with type printable = E.enum)
    : PrintCounter = struct

    module Count = GenCounter (E)
    type counter = P.printable
    include Count
    let printCounter() = P.print (getCounter())
end
```

Der Funktor darf nun allerdings nur noch auf solche Strukturen angewendet werden, bei denen die Constraints erfüllt sind :-)

9.6 Getrennte Übersetzung

- Eigentlich möchte man **Ocaml**-Programme nicht immer in der interaktiven Umgebung starten :-)
- Dazu gibt es u.a. den Compiler `ocamlc ...`

```
> ocamlc Test.ml
```

interpretiert den Inhalt der Datei `Test.ml` als Folge von Definitionen einer Struktur `Test`.

- Als Ergebnis der Übersetzung liefert `ocamlc` die Dateien:

<code>Test.cmo</code>	Bytecode für die Struktur
<code>Test.cmi</code>	Bytecode für das Interface
<code>a.out</code>	lauffähiges Programm

- Gibt es eine Datei `Test.mli` wird diese als Definition der Signatur für `Test` aufgefasst. Dann rufen wir auf:

```
> ocamlc Test.mli Test.ml
```

- Benutzt eine Struktur `A` eine Struktur `B`, dann sollte diese mit übersetzt werden:

```
> ocamlc B.mli B.ml A.mli A.ml
```

- Möchte man auf die Neuübersetzung von `B` verzichten, kann man `ocamlc` auch die vor-übersetzte Datei mitgeben:

```
> ocamlc B.cmo A.mli A.ml
```

- Zur praktischen Verwaltung von benötigten Neuübersetzungen nach Änderungen von Dateien bietet `Linux` das Kommando `make` an. Das Protokoll der auszuführenden Aktionen steht dann in einer Datei `Makefile` :-)

10 Formale Methoden für Ocaml

Frage:

Wie können wir uns versichern, dass ein Ocaml-Programm das macht, was es tun soll ???

Wir benötigen:

- eine formale Semantik;
- Techniken, um Aussagen über Programme zu beweisen ...

10.1 MiniOcaml

Um uns das Leben leicht zu machen, betrachten wir nur einen kleinen Ausschnitt aus Ocaml. Wir erlauben ...

- nur die Basistypen `int`, `bool` sowie Tupel und Listen;
- rekursive Funktionsdefinitionen nur auf dem `Top-Level` :-)

Wir verbieten ...

- veränderbare Datenstrukturen;
- Ein- und Ausgabe;
- lokale rekursive Funktionen :-)

Dieses Fragment von **Ocaml** nennen wir **MiniOcaml**.

Ausdrücke in **MiniOcaml** lassen sich durch die folgende Grammatik beschreiben:

$$\begin{aligned} E ::= & \text{const} \mid \text{name} \mid \text{op}_1 E \mid E_1 \text{op}_2 E_2 \mid \\ & (E_1, \dots, E_k) \mid \text{let name} = E_1 \text{ in } E_0 \mid \\ & \text{match } E \text{ with } P_1 \rightarrow E_1 \mid \dots \mid P_k \rightarrow E_k \mid \\ & \text{fun name} \rightarrow E \mid E_1 E_2 \end{aligned}$$
$$P ::= \text{const} \mid \text{name} \mid (P_1, \dots, P_k) \mid P_1 :: P_2$$

Achtung:

- Die Menge der **erlaubten** Ausdrücke muss weiter eingeschränkt werden auf diejenigen, die **typkorrekt** sind, d.h. für die der **Ocaml**-Compiler einen Typ herleiten kann ...
 - (1, [true; false]) **typkorrekt**
 - (1 [true; false]) nicht **typkorrekt**
 - ([1; true], false) nicht **typkorrekt**
- Wir verzichten auf `if ... then ... else ...`, da diese durch `match ... with true -> ... | false -> ...` simuliert werden können :-)
- Wir hätten auch auf `let ... in ...` verzichten können (wie?)

Ein **Programm** besteht dann aus einer Folge wechselseitig rekursiver globaler Definitionen von Variablen f_1, \dots, f_m :

```
let rec  $f_1 = E_1$   
      and  $f_2 = E_2$   
      ...  
      and  $f_m = E_m$ 
```

10.2 Eine Semantik für MiniOcaml

Frage:

Zu welchem Wert wertet sich ein Ausdruck E aus ??

Ein Wert ist ein Ausdruck, der nicht weiter ausgerechnet werden kann :-)

Die Menge der Werte lässt sich ebenfalls mit einer Grammatik beschreiben:

$$V ::= \text{const} \mid \text{fun name} \rightarrow E \mid (V_1, \dots, V_k) \mid [] \mid V_1 :: V_2$$