Einführung in die Informatik 2

Prof. Dr. Seidl, J. Kranz, N. Hartmann, J. Brunner
WS 2016/17

Übungsblatt 11
Abgabefrist: 23.1.2017

Aufgabe 11.1 (P) Diskussion über Module und Funktoren

Diskutieren Sie Funktoren in der Gruppe. Besprechen Sie dabei insbesondere folgende Themen:

- Wozu dienen Funktoren? Besprechen Sie insbesondere abstrakt, wieso uns Funktoren in der letzten Hausaufgabe 10.4 fehlten.
- In welcher Relation stehen Funktor, Modul und Signatur?
- Welche Parallelen zwischen Funktoren und bekannten Konzepten, insbesondere in Java, gibt es?
- Worin besteht der Unterschied zwischen `open` und `include`?

Lösungsvorschlag 11.1

- In Aufgabe 10.4 wurde eine Typ `key_info` benötigt, der Funktionen enthielt, um Schlüssel zu vergleichen bzw. einen Hashwert zu berechnen. Negativ fiel dabei insbesondere auf, dass in `key_info` Funktionen zum Hashing und Vergleichen von Schlüsseln vereint wurden, obwohl sie jeweils nur in einer der beiden Implementierungen benötigt wurden. Dies lässt sich mit Funktoren sehr gut lösen, wie man auf diesem Blatt sehen kann.
- Signaturen beschreiben das Interface von sowohl den Erzeugnissen eines Funktors, als auch seinem Parameter. Ein Funktor enthält ein Modul als Parameter und erzeugt daraus ein neues Modul.
- Funktoren erinnern an generische Klassen in Java. Auch hier wird ein neuer Typ aus einem Parameter-Typen gebaut, der bestimmte Eigenschaften erfüllen muss. Ein Beispiel:

```
1 class Map<T extends Comparable<T>> {
2   // ...
3 }
```

- `open` öffnet ein Modul, sodass die Funktionen darin anschließend genutzt werden können; `include` dagegen bindet die Funktionen des Moduls direkt im aktuellen Kontext ein, sodass sie auch wieder exportiert werden für Nutzer des einbindenden Moduls.

Aufgabe 11.2 (P) Foonktor für Ausdrücke

In dieser Aufgabe soll ein einfacher Funktor geschrieben werden, der Module zur Auswertung von Ausdrücken generiert. Dazu sei zunächst folgender Datentyp für Ausdrücke gegeben:

```
type 'a expr =
    Const of 'a
  | Addition of 'a expr * 'a expr
  | Subtraction of 'a expr * 'a expr
  | Multiplication of 'a expr * 'a expr
```

Ein `ExprEvaluator` wertet nun Ausdrücke über einem bestimmten Basistypen `elem` aus:

```
module type ExprEvaluator = sig
  type elem
  type t = elem expr

  val evaluate : t -> elem
end
```

Zur Auswertung von Ausdrücken müssen die Basisoperationen für den jeweiligen Typen bekannt sein. Ein passendes Modul habe folgende Signatur:

```
module type Ops = sig
  type elem

  val add : elem -> elem -> elem
  val sub : elem -> elem -> elem
  val mul : elem -> elem -> elem
end
```

Bearbeiten Sie nun nachstehende Teilaufgaben.

1. Implementieren Sie den Funktor `MakeExprEvaluator`, der, gegeben ein Modul der Signatur `Ops`, ein Modul des Typen `ExprEvaluator` erzeugt. Achten Sie darauf, dass in der Signatur des erzeugten Moduls der Typ `elem` demjenigen Typen entspricht, auf dem das übergebene Modul `Ops` arbeitet.
2. Nutzen Sie Ihren Funktor, um die Module `IntExprEvaluator`, `FloatExprEvaluator` und `VectorExprEvaluator` zu erzeugen. Vektoren sollen hier durch Listen von Integer-Zahlen repräsentiert werden, wobei die Multiplikation (unüblich, vereinfachend) punktweise definiert ist.
3. Testen Sie Ihre Implementierung! Werten Sie dazu einige Ausdrücke mithilfe der erzeugten Module aus.

Hinweis: Nutzen Sie einen sog. *Sharing Constraint*[1], um den inneren Typen des erzeugten Moduls von außen sichtbar zu machen.

Lösungsvorschlag 11.2

Die Lösung befindet sich in der Datei `ocaml/p11_sol.ml`.

[1] `https://realworldocaml.org/v1/en/html/functors.html`

Aufgabe 11.3 (P) Fun with Funktoren

In dieser Aufgabe soll ein Funktor für Matrizen implementiert werden.

1. Es sei folgende Signatur gegeben:

```
module type Number = sig
  type z
  val zero : z
  val ( +. ) : z -> z -> z
  val ( *. ) : z -> z -> z
  val string_of_number : z -> string
end
```

   (a) Definieren Sie ein Modul `Boolean`, das von dieser Signatur ist. Der $+$.-Operator bzw. der $*$.-Operator dieses Moduls soll dem logischen Oder- bzw. dem logischen Und-Operator entsprechen. Das Element `zero` soll das neutrale Element der Addition sein, d.h. in diesem Fall `false`.

   (b) Definieren Sie ein Modul `MinPlusNat`, das von dieser Signatur ist. Die Elemente sind alle natürlichen Zahlen inklusive der 0 erweitert um ∞. Der $+$.-Operator dieses Moduls soll das Minimum zweier Zahlen berechnen. Der $*$.-Operator soll der gewöhnlichen Addition entsprechen. Das Element `zero` soll das neutrale Element der Addition sein, d.h. in diesem Fall ∞.

2. Matrizen sind durch folgende Signatur definiert:

```
module type Matrix = sig
  type e
  type m
  val matrix_zero : m
  val ( **. ) : m -> m -> m
  val set_entry : m -> int * int -> e -> m
  val string_of_matrix : m -> string
end
```

   Dabei ist `e` der Typ für die Einträge der Matrix und `m` der Typ für die Matrix selbst. Der Wert `matrix_zero` ist eine Matrix, die nur aus Null-Einträgen besteht. Der Operator `**.` ist die Matrix-Multiplikation. Der Aufruf `set_entry m (i, j) e` liefert eine Matrix zurück, die der Matrix `m` entspricht, bis darauf, dass der Eintrag in der `i`-ten Zeile und der `j`-ten Spalte `e` ist.

   (a) Definieren Sie einen Funktor `MakeMatrix`, der eine Struktur der Signatur `Number` als Argument erhält und eine Struktur der Signatur `Matrix` zurückliefert. Die Matrix-Multiplikation `**.` soll die Multiplikation `*.` und die Addition `+.` der übergebenen `Number`-Struktur verwenden.

   (b) Testen Sie Ihre Implementierung anhand folgender Zeilen:

```
module BooleanMatrix = MakeMatrix(Boolean)
open BooleanMatrix
let a = set_entry matrix_zero (1,1) true
let a = set_entry a    (2,2) true
```

```
5  let a = set_entry a    (3,3) true
6  let a = set_entry a    (1,2) true
7  let a = set_entry a    (2,3) true
8  let a = set_entry a    (3,1) true
9
10 module MinPlusNatMatrix = MakeMatrix(MinPlusNat)
11 open MinPlusNat
12 open MinPlusNatMatrix
13 let b = set_entry matrix_zero (1,1) (Value 0)
14 let b = set_entry b    (2,2) (Value 0)
15 let b = set_entry b    (3,3) (Value 0)
16 let b = set_entry b    (1,2) (Value 1)
17 let b = set_entry b    (2,3) (Value 1)
18 let b = set_entry b    (3,1) (Value 1)
19 let b = set_entry b    (1,3) (Value 5)
```

Lösungsvorschlag 11.3

Die Lösung befindet sich in der Datei `ocaml/p11_sol.ml`.

Allgemeine Hinweise zur Hausaufgabenabgabe

Die Hausaufgabenabgabe bezüglich dieses Blattes erfolgt über das Ocaml-Abgabesystem. Sie erreichen es unter `https://vmnipkow3.in.tum.de`. Sie können hier Ihre Abgaben einstellen und erhalten Feedback, welche Tests zu welchen Aufgaben erfolgreich durchlaufen. Sie können Ihre Abgabe beliebig oft testen lassen. Bitte beachten Sie, dass Sie Ihre Abgabe stets als **eine einzige UTF8-kodierte Textdatei mit dem Namen *ha11.ml* hochladen müssen**. Die hochgeladene Datei muss ferner den Signaturen in *ha11.mli* entsprechen.

Aufgabe 11.4 (H) Modulare Wörterbücher mit Funktoren [10 Punkte]

In dieser Aufgabe wird die Hausaufgabe des letzten Blattes mithilfe von Funktoren weiterentwickelt. Dazu wird die Signatur für Wörterbücher (*Maps*) wie folgt geändert:

```
module type Map = sig
  type key
  type 'v t

  val create : unit -> 'v t
  val size : 'v t -> int
  val insert : 'v t -> key -> 'v -> 'v t
  val remove : 'v t -> key -> 'v t
  val lookup : 'v t -> key -> 'v option
end
```

Die Funktionen haben dabei jeweils folgende Bedeutung:

- `create`: Erzeugt ein neues Wörterbuch
- `size`: Gibt die Anzahl der Elemente im Wörterbuch zurück
- `insert`: Fügt ein neues Mapping in das Wörterbuch ein; sollte zum übergebenen Schlüssel bereits ein Eintrag vorhanden sein, so soll das jeweilige Datum ersetzt werden.
- `remove`: Entfernt ein Element aus dem Wörterbuch; ist kein passendes Element vorhanden, soll nichts passieren (insbesondere soll kein Fehler erzeugt werden).
- `lookup`: Sucht ein Element im Wörterbuch; wird kein Element gefunden, so gibt die Funktion `None` zurück.

Ferner seien die Signaturen `Hashable` und `Comparable` gegeben:

```
module type Hashable = sig
  type key

  val hash : key -> int
end

module type Comparable = sig
```

```
8   type key
9
10  val compare : key -> key -> int
11 end
```

Bearbeiten Sie die folgenden Teilaufgaben.

1. Implementieren Sie den Funktor `MakeHashMap`, der ein Modul der Signatur `Map` erzeugt. Nutzen Sie für Ihre Implementierung *Hashing with Chaining*. Verwenden Sie als Repräsentation der Hashtabelle ein Array[2] der Größe 37. Ihre Datenstruktur soll *veränderbar* sein (das Einfügen und Löschen von Elementen erzeugt keine neue Hashtabelle).
2. Implementieren Sie den Funktor `MakeTreeMap`, der ein Modul der Signatur `Map` erzeugt. Die Daten sollen hier in einem (unbalancierten) binären Buchbaum gespeichert werden. Nutzen Sie den in `ha10_angabe.ml` gegebenen Typen `bin_tree` sowie die ebenfalls gegebenen Funktionen `insert` und `remove` für Ihr Modul.
3. Nutzen Sie die Funktoren, um die Module `IntHashMap` und `IntTreeMap` zu generieren.

Hinweis: Nutzen Sie einen *Sharing Constraint*[3], um den inneren Typen der erzeugten Module von außen sichtbar zu machen. Testen Sie Ihre Implementierung, indem Sie in die jeweiligen Wörterbücher einige Daten einfügen, löschen und nachschlagen.

Lösungsvorschlag 11.4

Die Lösung befindet sich in der Datei `ocaml/ha11_sol.ml`.

Korrekturhinweise:

- Auf diesem Blatt ging es zentral darum, das Konzept der Funktoren zu verstehen. Es stellte sich heraus, dass einige Studenten größere Probleme hatten, überhaupt kompilierfähige Abgaben herzustellen. Wir bitte daher darum, auf diesem Blatt ausnahmsweise auch nicht kompilierende Abgaben anzuschauen und zu bewerten, ob bei diesen evtl. nur eine Kleinigkeit fehlt.

Aufgabe 11.5 (H) Klausuraufgabe zu Funktoren [10 Punkte]

Wir wollen im Folgenden einen Funktor `Lift` definieren, dessen Anwendung ein Modul mit der Signatur `Base` um nützliche Funktionen erweitert. Der Funktor soll auf beliebigen, einfach polymorphen, zyklenfreien Datenstrukturen arbeiten können. Dabei ist `'a t` der Typ, `empty` liefert eine leere Datenstruktur, `insert` fügt Daten in diese ein (Duplikate bleiben erhalten), und `fold` faltet eine Funktion über die Daten.

Achten Sie darauf, dass `fold insert empty x = x` gilt!

1. Implementieren Sie den Funktor `Lift`, der ein Modul des Typs `Base` erwartet und ein Modul der Signatur `Extended` erzeugt, wobei die Funktionen die vom `List`-Modul bekannte Semantik haben sollen. Wandeln Sie die Datenstruktur nicht erst in eine Liste um, sondern nutzen Sie direkt die Funktion `fold`.

[2] `https://realworldocaml.org/v1/en/html/imperative-programming-1.html`
[3] `https://realworldocaml.org/v1/en/html/functors.html`

2. Implementieren Sie das Basismodul `List` und nutzen Sie den Funktor `Lift`, um ein Modul `ExtendedList` zu erzeugen. Das Modul `List` nutzt eine Liste als inneren Typ. Die `fold` faltet dabei von rechts, `insert` fügt vorne in die List ein.

3. Implementieren Sie das Basismodul `SeachTree` und nutzen Sie den Funktor `Lift`, um ein Modul `ExtendedSearchTree` zu erzeugen. Das Modul `SearchTree` nutzt einen unbalancierten binären Suchbaum als inneren Typ. Die Funktion `fold` soll dabei ein *pre-order traversal* durchführen.

Hinweis: Nutzen Sie einen *Sharing Constraint*, um den inneren Typen der erzeugten Module von außen sichtbar zu machen.

Lösungsvorschlag 11.5

Die Lösung befindet sich in der Datei `ocaml/ha11_sol.ml`.

Korrekturhinweise:

- Auf diesem Blatt ging es zentral darum, das Konzept der Funktoren zu verstehen. Es stellte sich heraus, dass einige Studenten größere Probleme hatten, überhaupt kompilierfähige Abgaben herzustellen. Wir bitte daher darum, auf diesem Blatt ausnahmsweise auch nicht kompilierende Abgaben anzuschauen und zu bewerten, ob bei diesen evtl. nur eine Kleinigkeit fehlt.
- Bitte Teilpunkte nicht vergessen (wenn einzelne Funktionen des Tests fehlschlagen).