TECHNISCHE UNIVERSITÄT MÜNCHEN
FAKULTÄT FÜR INFORMATIK

Lehrstuhl für Sprachen und Beschreibungsstrukturen
Einführung in die Informatik 2
Prof. Dr. Helmut Seidl, Ralf Vogler, Stefan Schulze Frielinghaus

WS 2015/16
Übungsblatt 13

Aufgabe 13.1 [6 Punkte] **OCaml: Nebenläufigkeit: Futures**

In Ahnlehnung and das Future-Modul aus der Tutoraufgabe soll nun ein Modul mit der Signatur

```
type 'a t
val create : ('a -> 'b) -> 'a -> float -> 'b t
val get : 'a t -> 'a option
```

implementiert werden, welches Timeouts unterstützt. Ist die Berechnung `f a` nicht innerhalb von `t` Sekunden abgeschlossen, soll der Thread `create f a t` beendet werden und `get` entsprechend `None` liefern. Bis zum Erreichen des Timeouts soll der Aufruf, wie zuvor auch, blockieren.

Da `Thread.kill` in aktuellen OCaml-Versionen nicht implementiert ist, wird diese Aufgabe manuell bewertet. Stellen Sie sicher, dass Ihre Implementierung mit dem obigen Interface kompiliert (siehe Lösung zur Tutoraufgabe) und laden Sie sie als `timedFuture.ml` auf Moodle hoch. Nicht kompilierende Abgaben müssen nicht bewertet werden.

Aufgabe 13.2 [4 Punkte] **MiniOCaml-Verifikation**

Gegeben sei folgende MiniOCaml-Funktion:

```
let rec f = fun a ->
              match a with
                (z,[])    -> []
              | (z,x::xs) -> (z+x)::(f (z,xs))
```

Zeigen Sie mit Hilfe der BigStep operationellen Semantik, dass der Aufruf `f (7,l)` für jeden Listenwert `l` terminiert.

Aufgabe 13.3 Tutoraufgabe: OCaml: Nebenläufigkeit: Futures

Implementieren Sie ein Modul Future mit der Signatur

```
type 'a t
val create : ('a -> 'b) -> 'a -> 'b t
val get : 'a t -> 'a
```

wobei `'a t` die asynchrone Berechnung eines Werts vom Typ `'a` darstellt. `create f a` bekommt die Funktion `f` mit dem Argument `a` und gibt eine Future zurück, die `f a` asynchron berechnet. `get f` blockiert bis das Ergebnis der Future `f` berechnet wurde und gibt dieses dann zurück. Achten Sie darauf, dass folgende Aufrufe weiterhin das Ergebnis liefern und nicht blockieren!

Aufgabe 13.4 Tutoraufgabe: OCaml

a) Definieren Sie eine OCaml-Funktion

```
jedes_n_te : int -> 'a list -> 'a list
```

Ein Aufruf `jedes_n_te n l` soll jedes `n`-te Element aus der Liste `l` nehmen und aus diesen Elementen eine Liste konstruieren und zurückliefern.

Beispiel: `jedes_n_te 3 [1;2;3;4;5;6;7]` liefert als Ergebnis `[3;6]`

b) Definieren Sie eine OCaml-Funktion

```
m : ('a -> 'b -> 'c) list -> 'a list -> 'b list -> 'c list
```

Für

```
fs = [f_1; ...; f_n],   xs = [x_1; ...; x_n],   ys = [y_1; ...; y_n]
```

soll der Aufruf

```
m fs xs ys
```

das gleiche Ergebnis liefern wie die Auswertung des Ausdrucks

```
[f_1 x_1 y_1; ...; f_n x_n y_n].
```

Beispiel: `m [(fun x y -> x+y); (fun x y -> x-y)] [2;2] [2;2]` liefert als Ergebnis `[4;0]`