TECHNISCHE UNIVERSITÄT MÜNCHEN
FAKULTÄT FÜR INFORMATIK

Lehrstuhl für Sprachen und Beschreibungsstrukturen WS 2015/16
Einführung in die Informatik 2 **Übungsblatt 13**
Prof. Dr. Helmut Seidl, Ralf Vogler, Stefan Schulze Frielinghaus

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.

Lösungsvorschlag 13.1

```
type 'a t = 'a option Event.channel

let create f a t =
  let c = Event.new_channel () in
  let rec answer x = Event.(sync (send c x)); answer x in
  let task () =
    let b = f a in
    answer (Some b)
  in
  let w = Thread.create task () in
  let time () =
    Thread.delay t;
    try Thread.kill w with _ -> (); (* currently not implemented *)
    answer None
  in
  let _ = Thread.create time () in
  c

let get c = Event.(sync (receive c))

let test =
  let f x = Thread.delay 1.0; x+1 in
```

```
let g x = Thread.delay 5.0; x+1 in
let ff = create f 1 3.0 in
let fg = create g 1 3.0 in
let p n f =
  print_endline ("Evaluating "^n);
  let r = match get f with
    | Some x -> string_of_int x
    | None -> "timeout"
  in print_endline ("Result: "^r)
in
p "f" ff;
p "g" fg
```

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.

Lösungsvorschlag 13.2

Setze π gleich mit GD $\dfrac{\texttt{f} = \texttt{(fun a -> match a with (z,[]) -> [] | (z,x::xs) -> (z+x)::(f (z,xs)))}}{\texttt{f} \Rightarrow \texttt{(fun a -> match a with (z,[]) -> [] | (z,x::xs) -> (z+x)::(f (z,xs)))}}$

Induktionsanfang $n = 0$:

$$\textsc{App}\ \dfrac{\pi \qquad \textsc{PM}\ \dfrac{\texttt{[]} \Rightarrow \texttt{[]}}{\texttt{match (7,[]) with (z,[]) -> [] | (z,x::xs) -> (z+x)::(f (z,xs))} \Rightarrow \texttt{[]}}}{\texttt{f (7,[])} \Rightarrow \texttt{[]}}$$

Induktionsschritt $n > 0$:

$$\textsc{App}\ \dfrac{\pi \qquad \textsc{PM}\ \dfrac{\textsc{List}\ \dfrac{\texttt{7+v1} \Rightarrow \texttt{u1} \qquad \texttt{f (7,[v2;...;vn])} \Rightarrow \texttt{[u2;...;un]}}{\texttt{(7+v1)::(f (7,[v2;...;vn]))} \Rightarrow \texttt{[u1;...;un]}}}{\texttt{match (7,[v1;...;vn]) with (z,[]) -> [] | (z,x::xs) -> (z+x)::(f (z,xs))} \Rightarrow \texttt{[u1;...;un]}}}{\texttt{f (7,[v1;...;vn])} \Rightarrow \texttt{[u1;...;un]}}$$

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!

Lösungsvorschlag 13.3

```
(*
Compile & run:
  ocamlbuild -lib unix -tag thread future.byte && ./future.byte
Alternatively if you don't have ocamlbuild:
  ocamlc -thread unix.cma threads.cma future.ml && ./a.out
REPL with support for threads:
  utop -I +threads
Alternatively if you don't have utop:
  ocamlmktop -thread unix.cma threads.cma -o threaded_top
  ./threaded_top -I +threads
*)

module Future : sig
  type 'a t
  val create : ('a -> 'b) -> 'a -> 'b t
  val get : 'a t -> 'a
end = struct
  type 'a t = 'a Event.channel

  let create f a =
    let c = Event.new_channel () in
    let rec loop f = f (); loop f in
    let task () =
      let b = f a in
      loop (fun () -> Event.(sync (send c b)))
    in
    let _ = Thread.create task () in
    c

  let get c = Event.(sync (receive c))
end

let test =
  let f x = x+1 in
  let ff = Future.create f 1 in
  print_int (Future.get ff)
```

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]`