Parcial 17 Winter 2015/2016, Fragen

Einführung in die Informatik 2 (IN0003) (Technische Universität München)

Einführung in die Informatik 2

Prof. Dr. Seidl, Ralf Vogler, Stefan Schulze Frielinghaus 04.04.2016

Wiederholungsklausur

Vorname	Nachname
Matrikelnummer	Unterschrift

- Füllen Sie die oben angegebenen Felder aus.
- Schreiben Sie nur mit einem dokumentenechten Stift in schwarzer oder blauer Farbe.
- Verwenden Sie kein "Tipp-Ex" oder ähnliches.
- Die Arbeitszeit beträgt **90** Minuten.
- Prüfen Sie, ob Sie **18** Seiten erhalten haben.
- Sie können maximal **65** Punkte erreichen. Zum Bestehen benötigen Sie **26** Punkte.
- Als Hilfsmittel ist nur ein beidseitig handbeschriebenes A4-Blatt zugelassen.

1	2	3	4	5	6	Σ

Aufgabe 1. Multiple-Choice [7 Punkte]

Kreuzen Sie zutreffende Antworten an bzw. geben Sie die richtige Antwort. Punkte werden nach folgendem Schema vergeben:

- Richtige Antwort: $1/2$ Punkt
- Falsche Antwort: $-1/2$ Punkt
- Keine Antwort: 0 Punkte

Eine negative Gesamtpunktzahl wird zu 0 aufgerundet.

Verifikation Wahr Falsch

Wenn in einem lokal konsistent annotierten Kontrollfluß-Diagramm der Start-Knoten mit `true` und jeder Stop-Knoten mit `false` annotiert ist, dann terminiert das Programm nie. ☐ ☐

$\mathsf{WP}[\![s]\!](A) \implies A$ gilt für alle Anweisungen s. ☐ ☐

$(A \vee B \implies C) \implies (A \implies C) \wedge (B \implies C)$ ☐ ☐

$y \leq x$ ist eine Schleifen-Invariante für ☐ ☐

```
x = 5; y = 0; while y <= x do y++; end
```

OCaml

Die Typen `a -> b -> c` und `a -> (b -> c)` sind äquivalent. ☐ ☐

Die folgende Funktion ist tail-rekursiv: ☐ ☐

```
let rec sum' s k =
  match s with
  | [] -> k 0
  | x::xs -> sum' xs (fun a -> k (x + a))
```

Die Funktion `let rec fix f x = f (fix f) x` hat den Typ `('a -> 'b) -> 'a -> 'b`. ☐ ☐

Der folgende OCaml-Code ist fehlerfrei: ☐ ☐

```
let rec f = function
  | [] -> None
  | [x::_] -> Some x
  | _::x -> f x
```

Der Ausdruck `let x,y = 1,2 in (fun x -> x+y) y` ist korrekt und liefert 4. ☐ ☐

	Wahr	Falsch
Der Ausdruck `(fun x -> x % x) ((+) 1) 1` ist korrekt und liefert 3. Der Typ von `%` befindet sich im Anhang.	☐	☐
Der Ausdruck `List.fold_right (-) [1;2;3] 0` ist korrekt und liefert -6.	☐	☐

Verifikation funktionaler Programme

Jeder Beweis einer Aussage der Form $\texttt{e} \Rightarrow \texttt{v}$ zeigt insbesondere, dass die Auswertung des Ausdrucks `e` terminiert.	☐	☐
`(fun x -> x 1) x` ist ein Wert.	☐	☐
Die folgende abgeleitete Regel ist gültig:	☐	☐

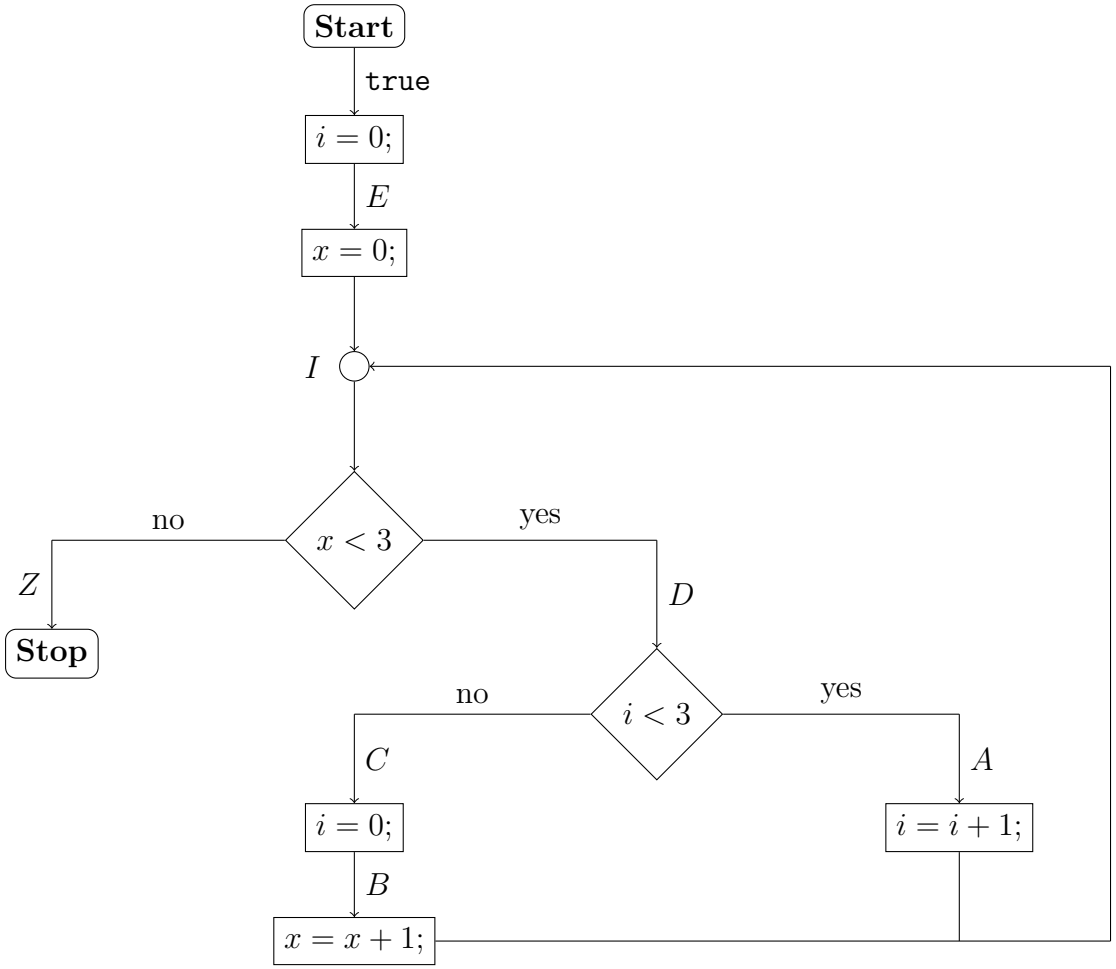
$$\frac{\texttt{e}_1\ \texttt{e}_2 \Rightarrow \texttt{fun}\ \texttt{x} \rightarrow \texttt{e}_0 \qquad \texttt{e}_3 \Rightarrow \texttt{v}_3 \qquad \texttt{e}_0[\texttt{v}_3/\texttt{x}] \Rightarrow \texttt{v}_0}{\texttt{e}_1\ \texttt{e}_2\ \texttt{e}_3 \Rightarrow \texttt{v}_0}$$

Aufgabe 2. Weakest Precondition [12 Punkte]

Gegeben sei das Programm

```
int i = 0;
int x = 0;
while (x < 3) {
  if (i < 3) {
    i = i+1;
  } else {
    i = 0;
    x = x+1;
  }
}
```

mit dem Kontrollflussgraphen



Geben Sie lokal konsistente Vorbedingungen für die Punkte A, B, C, D, E, I an für

$$Z := (x = 3 \wedge i \leq 3).$$

Aufgabe 3. OCaml: Komprimierte Binärbäume [10 Punkte]

Wir wollen Binärbäume vom Typ `'a t` in komprimierte Binärbäume vom Typ `'a c` umwandeln. Dabei steht der Konstruktor `Eq`/`Uneq` für einen Baum dessen linker und rechter Teilbaum jeweils gleich/ungleich ist.

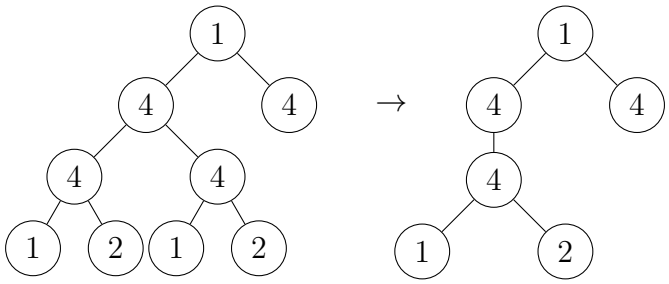
```
type 'a t = Leaf | Node of 'a * 'a t * 'a t
type 'a c = CLeaf | Uneq of 'a * 'a c * 'a c | Eq of 'a * 'a c
```

Implementieren Sie:

```
val compress : 'a t -> 'a c
val count : 'a c -> int
val merge : ('a -> 'b -> 'c) -> 'a c -> 'b c -> 'c c
```

wobei

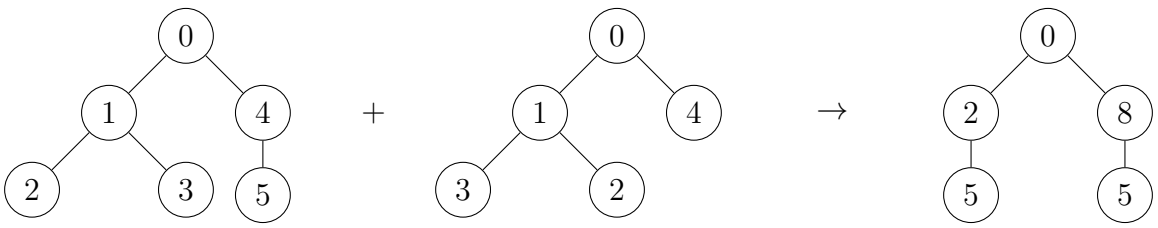
- `compress t` komprimiert den Baum:



  Knoten mit nur einem Nachfolger im rechten Baum stehen für den `Eq`-Konstruktor. Blätter sind nicht dargestellt.

- `count x` liefert die Anzahl aller Werte. Für das vorhergehende Beispiel gilt also `count (compress t) = 9`.

- `merge f a b` fügt zwei Bäume zusammen. Intuitiv bedeutet dies, dass die dekomprimierte Version von a und b übereinander gelegt werden und die sich überschneidenen inneren Knoten durch `f` ihren neuen Wert bekommen. Innere Knoten die nur in einem der Bäume vorkommen, werden unverändert übernommen, d.h. für Blätter gilt `merge f t CLeaf = merge f Cleaf t = t`. Achten Sie darauf, dass das Ergebnis wieder ein komprimierter Baum ist! Beispiel für `merge (+)`:



Aufgabe 4. OCaml: Threads [10 Punkte]

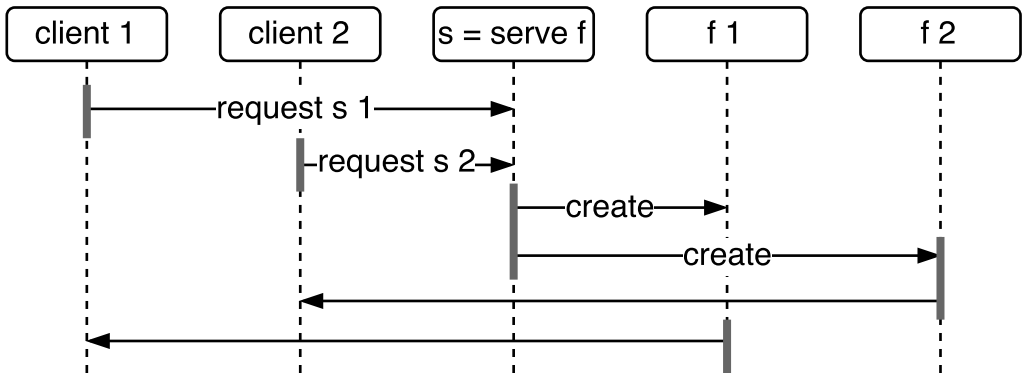
Implementieren Sie folgendes Modul

```
module Server : sig
  type ('a, 'b) t
  val serve : ('a -> 'b) -> ('a, 'b) t
  val request : ('a, 'b) t -> 'a -> 'b
end
```

wobei

- `('a, 'b) t` den Typ eines Servers darstellt, der Anfragen vom Typ `'a` entgegen nimmt und Ergebnisse vom Typ `'b` liefert.
- `serve f` einen Thread startet, der Anfragen mit der Funktion `f` beantwortet. Für die Bearbeitung einer Anfrage soll dazu jeweils wieder ein eigener Thread gestartet werden.
- `request s a` dem Server `s` eine Anfrage `a` schickt und auf dessen Antwort wartet.

Achten Sie darauf, dass Anfragen transaktionell bearbeitet werden – das Ergebnis also invariant gegenüber der Auswertungsreihenfolge von Threads ist. Werden z.B. `request s 1` und `request s 2` von unterschiedlichen Threads aus aufgerufen, so muss sichergestellt werden, dass jeder Request die richtige Antwort bekommt.



Implementierungen ohne Threads geben keine Punkte.

Aufgabe 5. MiniOCaml-Beweise [13 Punkte]

1. Zeigen Sie mithilfe der Big-Step-Semantik

$$\frac{\texttt{e terminiert}}{\texttt{map (fun x->x) e} \Rightarrow \texttt{e}}$$

für

```
let rec map = fun f -> fun a -> match a with
      [] -> []
    | x::xs -> f x :: map f xs
```

Gehen Sie nur von wohlgetypten Ausdrücken aus. Benutzungen von $v \Rightarrow v$ können Sie weglassen. Desweiteren können Sie folgende Setzungen verwenden:

$\pi_{\texttt{match}} \coloneqq$ `match a with [] -> [] | x::xs -> f x :: map f xs`
$\pi_{\texttt{map}} \coloneqq$ `fun f -> fun a -> match a with [] -> [] | x::xs -> f x :: map f xs`

2. Gegeben seien die *stets terminierenden* Funktionen

```
let rec length = fun x -> match x
    with [] -> 0
    | x::xs -> 1 + length xs
let rec app = fun x -> fun y -> match x
    with [] -> y
    | x::xs -> x :: app xs y
let rec rev = fun x -> match x
    with [] -> []
    | x::xs -> app (rev xs) [x]
```

Beweisen Sie nun aufeinander aufbauend (das Ergebnis der jeweils vorherigen Aufgabe kann als gegeben betrachtet werden):

(a) `length (app x y) = length x + length y`

(b) `length (rev x) = length x`

Geben Sie für jeden Schritt die verwendete Regel an (z.B. Def. `app`, IA, IS usw.)!

Aufgabe 6. OCaml: Functors [13 Punkte]

Im Folgenden soll ein Funktor `Memo` implementiert werden. Als Argument nimmt er ein Modul `A` mit der Signatur:

```
module type Hashable = sig
  type t
  val hash : t -> int
end
```

Der Funktor `Memo` dient der Memoisierung von Funktionen `A.t -> 'a`. Dabei soll sichergestellt werden, dass die Funktion für jedes Argument nur einmal ausgeführt wird. Dazu werden Paare von bereits berechneten Argumenten und zugehörigem Ergebnis der Funktionsanwendung in einem Suchbaum `'a tree` gespeichert. Dieser ist definiert durch:

```
type 'a tree = Node of int * (A.t * 'a) list * 'a tree * 'a tree | Empty
```

Für die Suche im Suchbaum wird der Hashwert des Arguments verwendet, welcher mit `A.hash` berechnet wird. Beachten Sie, dass mehrere Argumente auf den selben `int`-Wert gehasht werden können!

Der Zustand setzt sich aus der Funktion selbst und dem Suchbaum mit den bisherigen Auswertungen zusammen:

```
type 'a s = (A.t -> 'a) * 'a tree
```

1. Implementieren Sie nun:

   ```
   module Memo (A : Hashable) : sig
     type 'a s (* state for results of type 'a *)
     val create : (A.t -> 'a) -> 'a s
     val eval : A.t -> 'a s -> 'a * 'a s (* result and new state *)
     val (%>) : ('a s -> 'a * 'a s) -> ('a -> 'a s -> 'b) -> 'a s -> 'b
   end = struct
     type 'a tree = Node of int * (A.t * 'a) list * 'a tree * 'a tree
                  | Empty
     type 'a s = (A.t -> 'a) * 'a tree

     (* Code Aufgabe 1 *)
   end
   ```

   Zur Erläuterung:

   - `create f` liefert den leeren Zustand (noch keine Auswertungen) für die Funktion `f`.
   - `eval a x` nimmt das Argument `a` und den aktuellen Zustand `x` und liefert das Ergebnis und den neuen Zustand.
   - `f %> g` komponiert die beiden Funktionen so, dass zuerst `f` auf einem Zustand ausgeführt wird, woraufhin das Ergebnis sowie der neue Zustand `g` übergeben wird. Beispiel:
     `(eval 8 %> (fun a -> eval 9 %> (fun b _ -> a+b))) (create fib)`.

2. Implementieren Sie ein Modul `Int` das die Signatur `Hashable` für Integer erfüllt.

```
module Int : Hashable = struct
  (* Code Aufgabe 2 *)
end
```

3. Implementieren Sie einen Funktor `Tuple` der die Signatur `Hashable` für Tupel erfüllt.

```
module Tuple (A : Hashable) (B : Hashable) : Hashable = struct
  (* Code Aufgabe 3 *)
end
```

4. Implementieren Sie (unter Verwendung von `Memo`) einen Funktor `Memo2` der Funktionen mit zwei Argumenten (`A.t -> B.t -> 'a`) unterstützt.

```
module Memo2 (A : Hashable) (B : Hashable) : sig
  type 'a s
  val create : (A.t -> B.t -> 'a) -> 'a s
  val eval : A.t -> B.t -> 'a s -> 'a * 'a s
  val (%>) : ('a s -> 'a * 'a s) -> ('a -> 'a s -> 'b) -> 'a s -> 'b
end = struct
  (* Code Aufgabe 4 *)
end
```

Anhang

Funktionen die als gegeben betrachtet werden dürfen (alle anderen müssen definiert werden):

```
val ( % ) : ('a -> 'b) -> ('c -> 'a) -> 'c -> 'b
val id : 'a -> 'a
val flip : ('a -> 'b -> 'c) -> 'b -> 'a -> 'c
val neg : ('a -> bool) -> 'a -> bool
val const : 'a -> 'b -> 'a
val curry : ('a * 'b -> 'c) -> 'a -> 'b -> 'c
val uncurry : ('a -> 'b -> 'c) -> 'a * 'b -> 'c
val fst : 'a * 'b -> 'a
val snd : 'a * 'b -> 'b
module List : sig
  val cons : 'a -> 'a list -> 'a list
  val map : ('a -> 'b) -> 'a list -> 'b list
  val fold_left : ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a
  val fold_right : ('a -> 'b -> 'b) -> 'a list -> 'b -> 'b
  val find : ('a -> bool) -> 'a list -> 'a option
end
```

Weiterhin dürfen die Module `Thread` und `Event` komplett als gegeben betrachtet werden. Hilfestellung zur Signatur der wichtigsten Funktionen:

```
module Thread : sig
  type t
  val create : ('a -> 'b) -> 'a -> t
  (* ... *)
end
module Event : sig
  type 'a channel
  val new_channel : unit -> 'a channel
  type 'a event
  val send : 'a channel -> 'a -> unit event
  val receive : 'a channel -> 'a event
  val sync : 'a event -> 'a
  (* ... *)
end
```