EIDI2-Klausur WS15/16

Diese Angabe ist kein offizielles Dokument der TU-München
Erstellt von Kevin Kappelmann, 09.03.2016
Musterlösung erhaltbar unter https://github.com/kappelmann/eidi2_rep_2016
Angabe und Musterlösungen ohne Gewähr

Sie haben 90 Minuten Zeit zur Bearbeitung der Aufgaben. Viel Erfolg!

1. Multiple Choice 7P

Sind die folgenden Behauptungen wahr oder falsch. Kreuzen Sie die richtige Antwort an.
Jede richtige Antwort: +0.5P
Keine Antwort: 0P
Jede falsche Antwort: -0.5P

Bei Antworten, die keine wahr oder falsch Antworten sind, geben sie die korrekte Antwort an. Hier gibt es keine Minuspunkte.

1.1. Verifikation

a) false ist die schwächste Zusicherung

b) x<0 ist eine stärkere Zusicherung als x<-1

c) Zum Beweis einer Programmiereigenschaft muss man an einem Knoten true herleiten

d) (A^B =>C)≡A=>(B=>C)

e) x≥y ist eine Schleifeninvariante für x=5; y=0; while y<x do x++; end

1.2. Ocaml

a) (f g) x wertet sich zu dem selben Wert aus wie f g x

b) Die durch let f x = let p a b = a+b in p x definierte Funktion f is vom Typ int → int

c) Die folgende OCaml Zeile ist fehlerfrei: match [1,2] with [x,y] → x+y | z → 0

d) Der Typ des Ausdrucks fold_left (fun a x → x a) ist …

e) Die Auswertung des Ausdrucks let rec x = 1 in let x x = x in x 2 liefert …

f) Der Typ des Ausdrucks fun x → x % x ist … (die Signatur des % Operators finden Sie im Anhang)

1.3. Verifikation

a) Die folgende Big-Step-Regel ist gültig

$$\frac{e \Rightarrow e'::e''}{(\text{match } e \text{ with } [] \rightarrow e1 \mid x::xs \rightarrow e2) = e2[e'/x, e''/xs]}$$

b) fun x → x 1 ist ein Wert

c) Ein Mini-OCaml Ausdruck ohne Funktionsapplikation terminiert immer

2. Weakest Precondition 10P

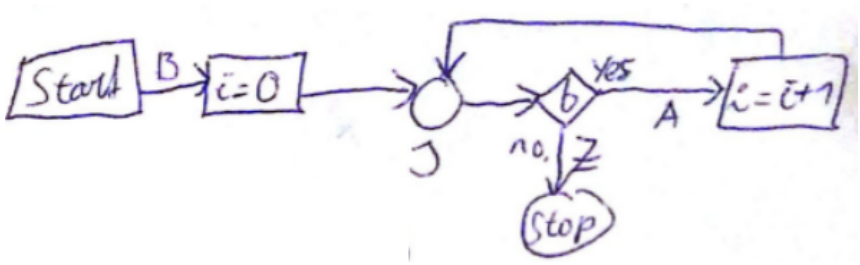
2.1. Berechne die schwächste Vorbedingung

a) WP[[i=i+1]](i=x^x=5) ≡ …

b) WP[[x<y]](x=2*y, x<2*y) ≡ …

2.2. Kontrollflussgraph

int i = 0;
while(b) { i=i+1;}



Geben sie für die Belegungen b und Z die schwächsten Vorbedingungen an

	A	I	B
b=false Z=false			
b=true Z=false			
b=i<17 Z=i>3			

3. Ocaml Sparse Vector 9P

Analog zu dünn besetzten Matrizen enthalten dünn besetzte Vektoren hauptsächlich Nullen, die man

nicht speichern will. Deshalb definieren wir einen Datentyp, der den Index beginnend mit 0 und den dazugehörigen Wert speichert.

Als Invariante soll gelten, dass nach jeder Operation nur Werte ungleich 0 in der Datenstruktur enthalten sind. z.B: add [1,1] [1,(-1)] = [] aber auch set 3 0 [3,1] = []

Implementieren Sie die folgenden Funktionen:

*type t = (int * int) list*
val empty : t
val set : int → int → t → t
val add : t → t → t
val mul : int → t → t
val sprod : t → t → int

Wobei empty den leeren Vektor darstellt.
set i v a setzt den Wert an Stelle i des Vektors a auf den Wert v
add a b Addition durch komponentenweise Summe, d.h. (a b c) + (d e f) = (a+d b+e c+f)
mul i r Skalarmultiplikation, d.h. i * (a b c) = (i*a i*b i*c)
sprod a b Standardskalarmultiplikation, d.h. <a,b>=$a_1*b_1+a_2*b_2+...+a_n*b_n$

4. Heavy Lifting 16P

Im folgenden wollen wir einen Funktor Lift definieren, dessen Anwendung ein Modul mit Signatur Base um nützliche Funktionen erweitert. Der Funktor soll auf beliebigen einfachen polymorphen zyklenfreie Datenstrukturen arbeiten.

'a t ist der Typ, der Datenstruktur, empty liefert eine leere Datenstruktur, insert fügt einen neuen Wert in die Datenstruktur ein und fold faltet die Funktion über die Datenstruktur.
Beachte: fold insert empty x = x

4.1. Funktorimplementierung

Implementiere den Funktor, wobei die Funktionen, die vom List-Modul bekannte Semantik besitzen sollen. Wandlen Sie die Datenstruktur nicht erst in eine Liste um, sondern verwenden sie z.B. B.fold

Signaturen

```
module type Base = sig
    type 'a t
    val empty : 'a t (*nullary*)
    val insert : 'a → 'a t → 'a t (*rec constr.*)
    val fold : ('a → 'b → 'b) → 'a t → 'b → 'b
end
module Lift (B:Base) : sig
    include Base
    val iter : ('a → unit) → 'a t → unit
    val map : ('a → 'b) → 'a t → 'b t
    val filter : ('a → bool) → 'a t → 'a t
    val append : 'a t → 'a t → 'a t
```

```
    val flatten : 'a t t → 'a t
    val to_list : 'a t → 'a list
    val from_list : 'a list → 'a t
end
```

Implementierung

```
module Lift (B: Base) = struct
    (*Hier kommt ihre Implementierung*)
end
```

4.2. Anwendung des Funktors

Geben Sie ein Base Modul für Listen an.

```
module List = Lift(struct
        (*Hier kommt ihr Code*)
    end)
```

4.3. (Bonus) Base Modul für binäre Suchbäume

Geben Sie ein Base Modul für binäre Suchbäume an.

```
module searchTree = Lift(struct
        (*Hier kommt ihr Code*)
    end)
```

5. ThreadedTree

Wir wollen nebenläufige Berechnungen auf Binärbäume durchführen. Die Daten befinden sich dabei nur in den Blättern des Baumes.

```
type 'a t = Leaf of 'a | Node of 'a t * 'a t
```

Schreiben Sie eine Funktion min welche das minimale Element eines Baumes liefert

```
val min 'a t → 'a
```

Dabei sollen innere Knoten für ihre Kinder Threads erstellen, auf das Ergebnis beider Teilbäume warten und dann dem Vaterknoten das Ergebnis mitteilen.

5.1. Alternative

Falls jemand nicht nochmal den ThreadedTree machen will, da wir ihn schon im Repetitorium zur Übung gemacht haben, kann man alternativ folgende Aufgabe bearbeiten:

Sie sind Übungsleitung eines Kurses an der TUM. Da Sie bereits Profi im funktionalen Programmieren sind, wollen Sie den Ablauf des Korrigierens der Klausuren automatisieren und schreiben sich deshalb ein Modul, das dies ermöglichen soll.
Sie erstellen hierzu folgende Signaturen:

```
type s = Student of string
(*p stellt den Datentyp für unsere Punkte dar*)
type p = float
type k = Klausur of s*(p list)
type g = Eins | Zwei | Drei | Vier | Fuenf
(*Die Notenskala weißt jeder Note ein oberes und unteres Punktelimit zu, wobei
 das erste p das obere und das zweite p das untere Punktelimit sein soll.*)
type n = Skala of (g*p*p) list

(*Berechnet die Summe der Punkte einer Klausur*)
val sum_k : k -> p

(*Ermittelt die Note für eine gegebene Punkteanzahl mit dem gegebenen Notenspiegel.
 Sollte keine passende Note gefunden werden, wird eine Exception geworfen*)
val get_grade : p -> n -> g

(*Berechnet die Note für eine Klausur mit dem gegebenen Notenspiegel*)
val grade_k : k -> n -> g

(*Berechnet für eine Liste von Klausuren mit gegebenen Notenspiegel
 die Liste von Noten für die Studenten.
 Da die Übungsleitung viele Tutoren besitzt, wird die Klausurenliste komplett
 parallelisiert benotet, d.h. die Note jeder Klausur wird immer von einem
 eigenen Thread parallel berechnet. Schnellere Threads sollen dabei nicht auf langsamere
 Threads warten müssen.*)
val eval : k list -> n -> (s*g) list
```

Implementieren sie die Funktionen des Moduls an Hand der gegebenen Signaturen und Kommentaren:

let sum_k = todo

let rec get_grade = todo

let grade_k = todo

let eval = todo

6. Beweise

```
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]
let rec rev1 = fun x → fun y → match x with [] → y
      | x::xs → rev1 xs (x::y)
```

Lemma 1: app x [] = x

Lemma 2: app (rev x) y = rev1 x y

Beweisen Sie aufeinander aufbauend die folgenden drei Behauptungen. Ergebnisse aus den vorherigen Aufgaben dürfen als gegeben betrachtet werden.
Geben Sie bei jedem Schritt die angewendete Regel an (Def, IA, IS, Lemma1,...)

1. rev1 (rev 1 x y) z = rev1 y (app x z)
2. rev (rev x) = x
3. (Bonus) rev (app (rev x) (rev y)) = app y x

Anhang

```
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
end

val ( % ) : ('a -> 'b) -> ('c -> 'a) -> 'c -> 'b
val id : 'a → 'a
val neg : ('a → bool) → 'a → bool
val flip : ('a → 'b → 'c) → 'b → 'a → 'c

module Thread : sig
    val create : ('a → 'b) → 'a → t
    val delay : float → unit
    val join : t → unit
end

module Event : sig
    type 'a channel
    val new_channel : unit → 'a channel
    val send : 'a channel → 'a → unit event
    val receive : 'a channel → 'a event
    val sync : 'a channel → 'a
    val select : 'a event list → 'a
    val choose : 'a event list → 'a event
    val wrap : 'a event → ('a → 'b) → 'b event
end
```