Einführung in die Informatik 2

Prof. Dr. H. Seidl, N. Hartmann, R. Vogler 24.03.2018

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 **120** Minuten.
- Prüfen Sie, ob Sie **12** Seiten erhalten haben.
- Sie können maximal **150** Punkte erreichen. Erreichen Sie mindestens **60** Punkte, bestehen Sie die Klausur.
- Als Hilfsmittel ist nur ein beidseitig beschriebenes A4-Blatt zugelassen.
- Es dürfen nur Funktionen aus den Modulen **Pervasives**, **List**, **String**, **Thread** und **Event** verwendet werden. Funktionen aus diesen Modulen dürfen ohne Angabe von Modulnamen verwendet werden.

vorzeitige Abgabe um ..... Hörsaal verlassen von ..... bis .....

1	2	3	4	5	6	7	8	Σ

.....
Erstkorrektor

.....
Zweitkorrektor

Aufgabe 1 Multiple-Choice [16 Punkte]

Kreuzen Sie in den folgenden Teilaufgaben jeweils die richtigen Antworten an. Es können pro Teilaufgabe keine, einige oder alle Antworten richtig sein. Die Teilaufgaben werden isoliert bewertet, Fehler in einer Teilaufgabe wirken sich also nicht auf die erreichbare Punktzahl bei anderen Teilaufgaben aus.

1. Was gilt, wenn zwischen einer Vorbedingung A und einer Nachbedingung B an einer Anweisung s die Beziehung $A \implies \mathsf{WP}[\![\mathtt{s}]\!](B)$ gilt?

   - ☐ $A \equiv B$.
   - ☒ Die Zusicherungen A und B sind lokal konsistent.
   - ☒ Wenn die Zusicherung A gilt, dann gilt auch B.
   - ☐ Das Programm erreicht die mit A und B annotierten Programmpunkte.

2. Welche der nachfolgenden Aussagen gilt?

   - ☐ $x > 1 \implies x > 0 \wedge y = 21$
   - ☒ $x < 3 \vee x^2 \geq 12 \implies x \neq 3$ stronger -> weaker
   - ☐ $x \neq y \wedge y < 3 \implies x > 0 \vee y < 0$
   - ☒ $x < 12 \wedge (y = x + 1 \vee y > 2) \wedge y * x \neq 2 \implies x - 1 \neq x$

3. Was gilt für die Funktion

```
let rec f n = if n > 1 then f (n-1) * f (n+1) else 1
```

   - ☐ Sie ist höherer Ordnung.
   - ☐ Sie ist endrekursiv.
   - ☒ Sie erzeugt einen Stackoverflow für die Eingabe 2.
   - ☒ Sie hat den Typ `int -> int`.

4. Was gilt für den Ausdruck `input_line (open_in "foo")` ?

   - ☒ Er öffnet die Datei `"foo"` zum Lesen.
   - ☐ Er gibt den Inhalt der Datei `"foo"` auf dem Terminal aus.
   - ☒ Er wirft eine Ausnahme, wenn die Datei `"foo"` leer ist. EOF
   - ☒ Er hat den Typ `string`.

5. Was gilt für den Ausdruck `let g = fun x -> (x 2, []) in g` ?

   - ☒ Er enthält eine polymorphe Funktion.
   - ☐ Er hat den Typ `int * 'a list`.
   - ☒ Er berechnet den Wert `fun x -> (x 2, [])`
   - ☐ Er enhält einen Syntaxfehler.

6. Was gilt für den Ausdruck `let rec f x = f x in f 1` ?

   - [ ] Er enthält Typfehler.
   - [x] Er terminiert nicht.
   - [ ] Er hat den Typ `int`.
   - [x] Er hat den Typ `'a`.

7. Was gilt für den Ausdruck `function Some (a,b) -> Some a | x -> x` ?

   - [ ] Er enhält Syntaxfehler.
   - [x] Er enhält Typfehler.
   - [ ] Er führt zu einer Warnung.
   - [ ] Er hat den Typ `('a * 'b) option -> 'a option`.

8. Wie kann der WP-Operator für eine MiniJava Anweisung `x = b ? e1 : e2` (für Bedingung b und Ausdrücke e_1, e_2) definiert werden, wenn der ternäre Operator `?:` die aus Java gewohnte Semantik haben soll?

   - [ ] $\mathsf{WP}[\![\texttt{x = } b \texttt{ ? } e_1 \texttt{ : } e_2]\!](B) \equiv (b \wedge B) \vee (\neg b \wedge \neg B)$
   - [ ] $\mathsf{WP}[\![\texttt{x = } b \texttt{ ? } e_1 \texttt{ : } e_2]\!](B) \equiv B[(e_1 = e_2)/x]$
   - [x] $\mathsf{WP}[\![\texttt{x = } b \texttt{ ? } e_1 \texttt{ : } e_2]\!](B) \equiv (b \implies B[e_1/x]) \wedge (\neg b \implies B[e_2/x])$
   - [ ] $\mathsf{WP}[\![\texttt{x = } b \texttt{ ? } e_1 \texttt{ : } e_2]\!](B) \equiv (x = e_1 \implies B) \wedge (x = e_2 \implies \neg B)$

Aufgabe 2 OCaml-Typen [10 Punkte]

1. Geben Sie einen Ausdruck mit dem Typ `'a -> 'a option list` .

   fun a -> [Some a]

---

2. Geben Sie einen Ausdruck mit dem Typ `('a * 'b -> 'c) -> 'b -> 'a -> 'c` .

   fun f b a -> f (a, b)

---

3. Welchen Typ hat der Ausdruck `fun x -> fun x -> x, x` ?

   'a -> 'b -> 'b * 'b

---

4. Welchen Typ hat der Ausdruck `fun x -> x (fun x -> x)` ?

   (('a -> 'a) -> 'b) -> 'b

Aufgabe 3 OCaml: Peano-Zahlen [12 Punkte]

Für den Umgang mit natürlichen Zahlen definieren wir den Typ

```
type nat = Z | S of nat
```

Dabei repräsentiert `Z` die Zahl 0 und `S n` den Nachfolger der Zahl n, also $n+1$. Wir gehen davon aus, dass `Z` saturierend ist, d.h. alle negativen Zahlen werden auf `Z` abgebildet. Implementieren Sie die folgenden Funktionen, wobei Sie für die Implementierung der Funktionen `add`, `mul`, `sub` und `leq` die Funktionen `of_int` und `to_int` **nicht** verwenden dürfen:

1. `of_int : int -> nat` wandelt eine Ganzzahl in eine natürliche Zahl um.
   Beispiel: `of_int 2 = S (S Z)`, `of_int (-2) = Z`.

2. `to_int : nat -> int` wandelt eine natürliche Zahl in eine Ganzzahl um.
   Beispiel: `to_int (S (S Z)) = 2`.

3. `add : nat -> nat -> nat` addiert zwei natürliche Zahlen.
   Beispiel: `add (S (S Z)) (S Z) = S (S (S Z))`.

4. `mul : nat -> nat -> nat` multipliziert zwei natürliche Zahlen.
   Beispiel: `mul (S (S Z)) Z = Z`.

5. `sub : nat -> nat -> nat` substrahiert zwei natürliche Zahlen.
   Beispiel: `sub (S (S Z)) (S Z) = S Z`, `sub (S Z) (S (S Z)) = Z`.

6. `leq : nat -> nat -> bool` vergleicht zwei natürliche Zahlen (less-or-equal).
   Beispiel: `leq (S (S Z)) (S Z) = false`, `leq (S Z) (S (S Z)) = true`.

Aufgabe 4 Big-Step Semantik, Äquivalenz [24 Punkte]

Es seien folgende Definitionen gegeben:

```
let rec map f l =
  match l with [] -> []
  | h::t -> f h :: map f t

let (%) a b c = a (b c)

let id x = x
```

1. [9,5 Punkte] Zeigen Sie, dass für beliebige Listen `l` die Äquivalenz

   ```
   map g (map f l) = l
   ```

   gilt, sofern die Funktionen `f` und `g` die Voraussetzung

   ```
   g % f = id
   ```

   erfüllen.

2. [14,5 Punkte] Zeigen Sie mithilfe der Big-Step operationellen Semantik, dass der Ausdruck

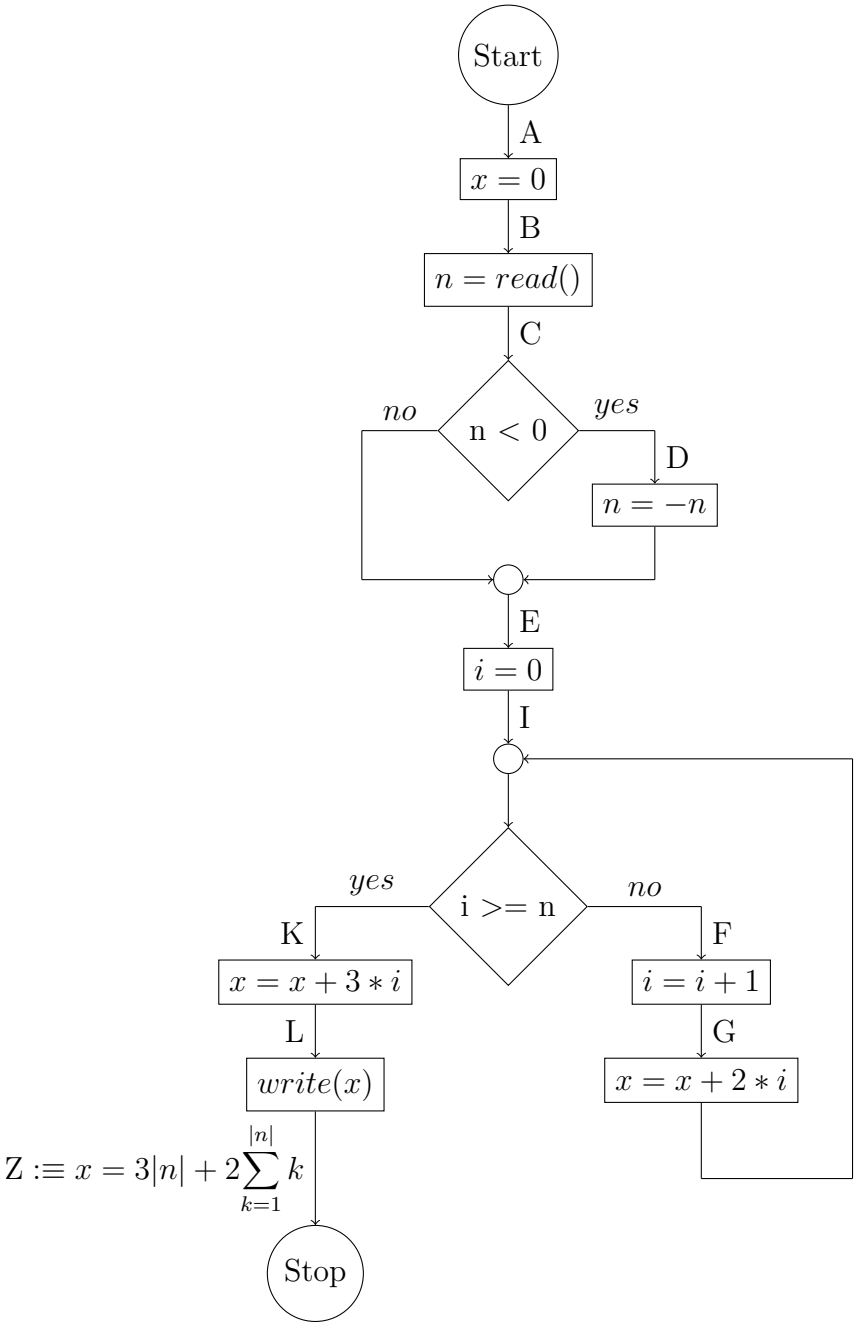
   ```
   map (id (fun x -> x + 1)) [2;3]
   ```

   zur Liste `[3;4]` auswertet. Dazu sei bereits folgender Teil gegeben, den Sie in Ihrem Beweis direkt verwenden dürfen:

$$\pi_{map} = \frac{}{\texttt{map} = \texttt{fun f l -> match l with [] -> [] | h::t -> f h::map f t}}\,\text{GD}\;\frac{}{\texttt{map} \Rightarrow \texttt{fun f l -> match l with [] -> [] | h::t -> f h::map f t}}$$

Aufgabe 5 Verifikation: Weakest Preconditions [24 Punkte]

Zeigen Sie mithilfe des WP-Kalküls, dass die Zusicherung Z für alle Ausführungen des folgenden Programms gilt. Die Terminierung des Programms muss nicht gezeigt werden.



Aufgabe 6 OCaml: Selbstorganisierende assoziative Listen [23 Punkte]

In dieser Aufgabe wollen wir die Suche in assoziativen Listen beschleunigen.

1. [6 Punkte] Die erste Strategie dafür bewegt bei jeder Anfrage das Element an den Anfang der Liste. Der Aufruf `find1 "baz" [("foo", -3); ("bar", 4); ("baz", 5)]` liefert

   ```
   Some 5, [("baz", 5); ("foo", -3); ("bar", 4)]
   ```

   d.h. den optionalen Wert des angefragten Schlüssels zusammen mit der neuen assoziativen Liste. Implementieren Sie eine Funktion `val find1 : 'a -> ('a * 'b) list -> 'b option * ('a * 'b) list` mit dieser Eigenschaft sodass die Laufzeit proportional zur Position des Schlüssels ist.

2. [17 Punkte] Die zweite Strategie berücksichtigt die Häufigkeit mit der ein Schlüssel bisher angefragt wurde. Deshalb erweitern wir den Typ folgendermaßen:

   ```
   type ('a,'b) mfu = (int * ('a * 'b)) list
   ```

   wobei `int` die Anzahl der bisherigen Anfragen darstellt. In der Liste sollen die Einträge absteigend nach der Anzahl der Anfragen sortiert abgelegt sein. Eine solche Liste ist zum Beispiel

   ```
   let xs = [(3, ("foo", -3)); (2, ("bar", 4)); (1, ("baz", 5))]
   ```

   Dazu implementieren wir folgende Funktionen:

   (a) `val init : ('a * 'b) list -> ('a * 'b) mfu` welche eine assoziative Liste um die Häufigkeiten 0 erweitert;

   (b) `val find2 : 'a -> ('a * 'b) mfu -> 'b option * ('a * 'b) mfu` welche für einen Schlüssel und eine gewichtete assoziative Liste `xs` den zugehörigen Wert (falls vorhanden) und zusätzlich die neue Liste `xs'` liefert. `xs'` soll sich von `xs` nur im Eintrag für den angefragten Schlüssel unterscheiden. Für diesen soll das entsprechende Gewicht inkrementiert werden und die Position innerhalb der Liste angepasst werden. Die Laufzeit der Funktion soll dabei proportional zur Position des Schlüssels in `xs` sein.

   Zum Beispiel liefert `find2 "baz" xs` als Ergebnis das Paar

   ```
   Some 5, [(3, ("foo", -3)); (2, ("baz", 5)); (2, ("bar", 4))]
   ```

Aufgabe 7 OCaml: Module/Funktoren [18 Punkte]

Zur Beschreibung eines Typen sei folgende Signatur gegeben:

```
module type S = sig
  type t
  val size : t -> int
  val show : t -> string
end
```

Implementieren Sie die Funktoren

1. [4 Punkte] `Pair` für Paare
2. [7 Punkte] `List` für Listen
3. [7 Punkte] `Either` für zwei Alternativen (unterschieden durch die Konstruktoren `A` und `B`)

welche Module der Signatur `S` als Argumente haben und deren Ergebnis wiederum die Signatur `S` erfüllt. Dabei sollen die Funktionen `size` und `show` auf die entsprechenden Funktionen der Eingabe-Module zurückgreifen und der Typ `t` sich aus den Typen der Eingabe-Module zusammensetzen.

Beispiel:

```
module Int = struct
  type t = int
  let size x = 1
  let show = string_of_int
end
module IPair = Pair (Int) (Int)
module IList = List (Int)
module IEither = Either (Int) (Int)
module LPair = Pair (Int) (IList)
```

Für Ihre Implementierungen sollte nun folgendes gelten:

```
IPair.show (1,2) = "(1, 2)"
IPair.size (5,4) = 2
IList.show [1;2;3] = "[1; 2; 3]"
IList.size [4;2;5] = 3
IEither.show (IEither.A 1) = "(A 1)"
IEither.size (IEither.B 3) = 1
LPair.show (1,[2;3]) = "(1, [2; 3])"
LPair.size (4,[2;5]) = 3
```

Hinweis: Sie können die Funktion `String.concat : string -> string list -> string` *nutzen (siehe Anhang).*

Aufgabe 8 OCaml (Threads): P2P-Server [23 Punkte]

In dieser Aufgabe möchten wir ein Peer-to-Peer-Netzwerk simulieren. Jeder Nutzer kann für einen Schlüssel Daten bereitstellen. Ein Broker verwaltet dabei wer für welchen Schlüssel Daten anbietet und leitet Anfragen weiter. Wir definieren einen Typ

```
type ('a,'b) t = Publish of 'a * 'b channel
               | Request of 'a * 'b event option channel
```

wobei `Publish` aus dem Schlüssel und einem Channel besteht auf dem die Daten fortlaufend geschickt werden, und `Request` aus dem Schlüssel und einem Channel besteht auf dem ein Event zum Empfangen der Daten zurückgeschickt wird, falls jemand dazu Daten veröffentlicht hat.
Implementieren Sie die folgenden Funktionen:

1. [11 Punkte] `val broker : unit -> ('a,'b) t channel` wobei `broker ()` einen neuen Broker startet, der Anfragen für Schlüssel vom Typ `'a` mit Daten vom Typ `'b` verwaltet.

2. [7 Punkte] `val publish : ('a,'b) t channel -> 'a -> 'b -> unit` wobei `publish b k v` einen Thread startet, der die Daten `v` dauerhaft bereitstellt und dies beim Broker `b` unter dem Schlüssel `k` veröffentlicht.

3. [5 Punkte] `val request : ('a,'b) t channel -> 'a -> 'b option` wobei `request b k` bei einem Broker `b` den Schlüssel `k` anfragt und das Ergebnis empfängt (sofern vorhanden).

Hinweis: Sie dürfen für Ihre Lösung annehmen, dass jeder Schlüssel nur einmal veröffentlicht wird.

Anhang

Big-Step Semantik

Axiome: $v \Rightarrow v$ für jeden Wert v

Tupel: $$\text{T}\ \frac{e_1 \Rightarrow v_1 \quad \ldots \quad e_k \Rightarrow v_k}{(e_1, \ldots, e_k) \Rightarrow (v_1, \ldots, v_k)}$$

Listen: $$\text{L}\ \frac{e_1 \Rightarrow v_1 \quad e_2 \Rightarrow v_2}{e_1 :: e_2 \Rightarrow v_1 :: v_2}$$

Globale Definitionen: $$\text{GD}\ \frac{f = e \quad e \Rightarrow v}{f \Rightarrow v}$$

Lokale Definitionen: $$\text{LD}\ \frac{e_1 \Rightarrow v_1 \quad e_0[v_1/x] \Rightarrow v_0}{\texttt{let}\ x\ \texttt{=}\ e_1\ \texttt{in}\ e_0 \Rightarrow v_0}$$

Funktionsaufrufe: $$\textsc{App}\ \frac{e_1 \Rightarrow \texttt{fun}\ x\ \texttt{->}\ e_0 \quad e_2 \Rightarrow v_2 \quad e_0[v_2/x] \Rightarrow v_0}{e_1\ e_2 \Rightarrow v_0}$$

Funktionsaufrufe mit mehreren Argumenten: $$\textsc{App}',\ \frac{e_0 \Rightarrow \texttt{fun}\ x_1\ \ldots\ x_k\ \texttt{->}\ e \quad e_1 \Rightarrow v_1\ \ldots\ e_k \Rightarrow v_k \quad e[v_1/x_1\ \ldots\ v_k/x_k] \Rightarrow v}{e_0\ e_1\ \ldots\ e_k \Rightarrow v}$$

Pattern Matching: $$\text{PM}\ \frac{e_0 \Rightarrow v' \equiv p_i[v_1/x_1, \ldots, v_k/x_k] \quad e_i[v_1/x_1, \ldots, v_k/x_k] \Rightarrow v}{\texttt{match}\ e_0\ \texttt{with}\ p_1\ \texttt{->}\ e_1\ \texttt{|}\ \ldots\ \texttt{|}\ p_m\ \texttt{->}\ e_m \Rightarrow v}$$

— sofern v' auf keines der Muster $p_1, \ldots, p_{i-1}$ passt

Eingebaute Operatoren: $$\textsc{Op}\ \frac{e_1 \Rightarrow v_1 \quad e_2 \Rightarrow v_2 \quad v_1\ \texttt{op}\ v_2 \Rightarrow v}{e_1\ \texttt{op}\ e_2 \Rightarrow v}$$

— Unäre Operatoren werden analog behandelt.

Substitutionslemma

$$\frac{e_1 = e_2}{e[e_1/x] = e[e_2/x]}$$

OCaml Referenz

Modul List

Signatur	Erklärung
`val append : 'a list -> 'a list -> 'a list`	Concatenate two lists. Same as the infix operator `@`.
`val map : ('a -> 'b) -> 'a list -> 'b list`	`map f [a1; ...; an]` applies function `f` to `a1, ..., an`, and builds the list `[f a1; ...; f an]` with the results returned by `f`.
`val fold_left : ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a`	`fold_left f a [b1; ...; bn]` is `f (... (f (f a b1) b2) ...) bn`.
`val fold_right : ('a -> 'b -> 'b) -> 'a list -> 'b -> 'b`	`fold_right f [a1; ...; an] b` is `f a1 (f a2 (... (f an b) ...))`.
`val filter : ('a -> bool) -> 'a list -> 'a list`	`filter p l` returns all the elements of the list `l` that satisfy the predicate `p`. The order of the elements in the input list is preserved.
`val assoc : 'a -> ('a * 'b) list -> 'b`	`assoc a l` returns the value associated with key `a` in the list of pairs `l`. That is, `assoc a [ ...; (a,b); ...] = b` if `(a,b)` is the leftmost binding of `a` in list `l`. Raise `Not_found` if there is no value associated with a in the list `l`.
`val assoc_opt : 'a -> ('a * 'b) list -> 'b option`	Same as `assoc`, but returns `Some b`, or `None` if there is no binding.
`val remove_assoc : 'a -> ('a * 'b) list -> ('a * 'b) list`	`remove_assoc a l` returns the list of pairs `l` without the first pair with key `a`, if any.
`val mem_assoc : 'a -> ('a * 'b) list -> bool`	Same as `assoc`, but simply return true if a binding exists, and false if no bindings exist for the given key.

String

`val concat : string -> string list -> string`	`concat sep sl` concatenates the list of strings `sl`, inserting the separator string `sep` between each.

Modul Thread und Event

`val create : ('a -> 'b) -> 'a -> t`	`create funct arg` creates a new thread of control, in which the function application `funct arg` is executed concurrently with the other threads of the program.
`val send : 'a channel -> 'a -> unit event`	`send c x` sends a value `x` over the channel `c`. It returns an event that occurs as soon as value is received.
`val receive : 'a channel -> 'a event`	`receive c` returns an event that occurs as soon as a value is received from the channel.
`val sync : 'a event -> 'a`	`sync e` waites for a single event `e` to occur.
`val select : 'a event list -> 'a`	`select l` waites for any event in `l` to occur. The list may contain events that already occurred.
`val new_channel : unit -> 'a channel`	`new_channel ()` creates a new channel.