

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 für ein Programm, wenn alle Zusicherungen lokal konsistent sind?

- Die Zusicherung am Startknoten ist *true*.
- Für alle Anweisungen s mit Vorbedingung A und Nachbedingung B gilt:
$$A \implies \text{WP}[\![s]\!](B)$$
- Das Programm terminiert für alle Eingaben.
- Das Programm enthält keine Schleifen.

2. Welche der nachfolgenden Aussagen gilt?

- $x < 7 \implies x < 12 \vee y = 21$
- $x = 1 \wedge y = 2 \implies x > -2 \wedge y \geq x$
- $x = y \vee x = 3 \implies y = 3$
- $x = x - 1 \implies y > x \vee x = 21 \vee x = 2 \wedge y = 8$

3. Was ist eine hinreichende Bedingung für die Terminierung eines MiniJava Programms?

- Das Programm enthält keine Schleifen.
- Mindestens eine Schleife im Programm wird nur endlich oft durchlaufen.
- Im Programm kommt eine Variable r vor.
- Am Stopknoten lässt sich *true* beweisen.

4. Was gilt für den Datentyp `type 'a t = Lama of 'a t | Dromedar`?

- Er definiert zwei Konstruktoren.
- Er definiert ein Record.
- Er ist rekursiv.
- Er enthält eine Typ-Variable.

5. Was gilt für die Funktion `let rec f a b = if b then a else f a true`?

- Sie ist endrekursiv.
- Sie ist polymorph.
- Sie terminiert nicht.
- Sie gibt den Typ `unit` zurück.

Lösungsvorschlag 7

```
module ListIter = struct (0.5P)
  type 'a t = 'a list (1P)
  type 'a s = 'a list (1P)
  let init c = c (0.5P)
  let next = function [] -> None, [] | x::xs -> Some x, xs (1.5P)
end

module TreeIter = struct (0.5P)
  type 'a t = 'a tree (1P)
  type 'a s = 'a t list (1P)
  let init c = [c] (0.5P)
  let rec next = function [] -> None, [] (2.5P)
    | Leaf::xs -> next xs
    | Node (x, Leaf, Leaf)::xs -> Some x, xs
    | Node (x, l, r)::xs -> next (l::Node (x, Leaf, Leaf)::r::xs)
  end

module ExtIter (I : Iter) = struct (1P)
  include I (1P)
  let rec next_filtered f s = (2P)
    match next s with
    | None, s' -> None, s'
    | Some x, s' -> if f x then Some x, s' else next_filtered f s'
  let next_mapped f s = (2P)
    match next s with
    | None, s' -> None, s'
    | Some x, s' -> Some (f x), s'
  end

module PairIter (F : Iter) (S : Iter) = struct (1P)
  type ('a, 'b) t = 'a F.t * 'b S.t (1P)
  type ('a, 'b) s = 'a F.s * 'b S.s (1P)
  let init (c1, c2) = F.init c1, S.init c2 (0.5P)
  let next (s1, s2) = (2.5P)
    match F.next s1, S.next s2 with
    | (Some x1, s1'), (Some x2, s2') -> Some (x1, x2), (s1', s2')
    | _, _ -> None, (s1, s2)
  end
```

Korrekturhinweise:

- (4.5P), (5.5P), (6P) und (6P) für die Teilaufgaben 1, 2, 3 und 4.
- (-0.5P) wenn `end` fehlt.

• jeweils (-0.5P) wenn bei 6.4 Typen die Typvariablen fehlen.

Aufgabe 5 Big-Step Semantik, Äquivalenz

[24 Punkte]

Es seien folgende Funktionsdefinitionen gegeben:

```

let rec nlen n l =
  match l with [] -> 0
  | h::t -> n + nlen n t

let rec fold_left f a l =
  match l with [] -> a
  | h::t -> fold_left f (f a h) t

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

let (+) a b = a + b

```

1. Zeigen Sie mithilfe der Äquivalenzumformungen, dass für beliebige l und n gilt:

$$nlen\ n\ l = fold_left\ (+)\ 0\ (map\ (\text{fun } _\ -> n)\ l)$$

2. Zeigen Sie mithilfe der Big-Step operationellen Semantik, dass der Ausdruck

$$\text{map}\ (\text{fun } a\ -> a * a)\ [nlen\ 2\ [3]]$$

zur Liste [4] auswertet. Regeln der Form $v \Rightarrow v$ dürfen weggelassen werden. Dazu seien bereits folgende Teile gegeben, die Sie in Ihrem Beweis direkt verwenden dürfen:

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

$$\pi_{nlen} = \text{GD} \frac{nlen = \text{fun } n\ l\ -> \text{match } l \text{ with } [] \ -> 0 \mid h::t \ -> n + nlen\ n\ t}{nlen \Rightarrow \text{fun } n\ l\ -> \text{match } l \text{ with } [] \ -> 0 \mid h::t \ -> n + nlen\ n\ t}$$

Aufgabe 3 OCaml: Balancierte Klammern

[12 Punkte]

Implementieren Sie eine Funktion `is_balanced : string -> bool` welche entscheidet, ob Klammern in der Eingabe balanciert sind. Klammern sind balanciert, wenn es nie mehr schließende als öffnende und am Ende genau gleich viele öffnende wie schließende Klammern gibt.

Beispiele:

```
false = is_balanced "()"
false = is_balanced "(a)b)"
true  = is_balanced "foo"
true  = is_balanced "a(b(c)d)e"
```

Für diese Aufgabe dürfen Sie annehmen, dass es eine Funktion `String.to_list : string -> char list` gibt, welche einen String in eine Liste seiner Zeichen umwandelt.

Lösungsvorschlag 3

```
let is_balanced s =
  let rec f c = function (1P)
    | [] -> c=0 (2P)
    | '(::xs -> f (c+1) xs (2P)
    | ')'::xs -> c>0 && f (c-1) xs (2+2P)
    | _::xs -> f c xs (2P)
  in f 0 (String.to_list s) (1P)
```

Korrekturhinweise:

- Wurde der Fall ")()" nicht korrekt erkannt, dann gab es
 - (-2P), wenn dies leicht zu beheben wäre (z.B. durch Einfügen eines $c > 0$).
 - (-4P), wenn die Implementierung dies so nicht abfangen könnte, z.B. wenn nur die Zahl der Klammern verglichen wird.
- (2P) für jeden der 4 Fälle.
- (1P) für die rekursive Hilfsfunktion und Patternmatching auf die Liste.
- (1P) für `String.to_list` im richtigen Kontext.

Aufgabe 2 OCaml-Typen

[10 Punkte]

1. Geben Sie einen Ausdruck mit dem Typ $('a \rightarrow 'b) \rightarrow 'a \rightarrow 'b$.

fun f x -> f x

2. Geben Sie einen Ausdruck mit dem Typ $('a \rightarrow 'b) \rightarrow ('c \rightarrow 'a) \rightarrow 'c \rightarrow 'b$.

fun f g x -> f (g x)

3. Welchen Typ hat der Ausdruck `fun xs -> (List.hd xs) 1`?

(int -> 'a) list -> 'a

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

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

Lösungsvorschlag 2

Korrekturhinweise:

- (2P), (2P), (3P) und (3P) für die Teilaufgaben 1, 2, 3 und 4.
- Sind nur kleinere Fehler in einem Typen, gibt es Teilpunkte.
- Wurde anstelle eines Ausdrucks eine Funktion mit entsprechendem Typ definiert, gibt es (1P) Abzug. Kam dies in beiden Teilaufgaben vor, so gab es beim zweiten mal volle Punkte (Folgefehler).

Lösungsvorschlag 5

1. Wir müssen die allgemeinere (0.5P) Aussage

$\text{acc} + \text{nlen } n \ 1 = \text{fold_left } (+) \ \text{acc} (\text{map} (\text{fun } _{-} \rightarrow n) \ 1)$ (0.5P)

beweisen. Dazu machen wir eine Induktion über die Länge k von $l = []$ (0.5P):

- Induktionsanfang: $k = 0$, also $l = []$ (0.5P)

$$\begin{aligned}
 & \text{acc} + \text{nlen } n \ [] \\
 \stackrel{\text{nlen}}{=} & \text{acc} + \text{match } [] \ \text{with } [] \rightarrow 0 \mid h :: t \rightarrow n + \text{nlen } n \ t \\
 \stackrel{\text{match}}{=} & \text{acc} + 0 \\
 \stackrel{\text{arith}}{=} & \text{acc} \\
 \stackrel{\text{match}}{=} & \text{match } [] \ \text{with } [] \rightarrow \text{acc} \mid h :: t \rightarrow \text{fold_left } (+) ((+) \ \text{acc} \ h) \ t \\
 \stackrel{(+)}{=} & \text{fold_left } (+) \ \text{acc} \ []
 \end{aligned} \quad (1P)$$

$$\begin{aligned}
 \stackrel{\text{match}}{=} & \text{fold_left } (+) \ \text{acc} (\text{match } [] \ \text{with } [] \rightarrow [] \\
 & \mid h :: t \rightarrow (\text{fun } _{-} \rightarrow n) \ h :: \text{map} (\text{fun } _{-} \rightarrow n) \ t) \\
 \stackrel{\text{map}}{=} & \text{fold_left } (+) \ \text{acc} (\text{map} (\text{fun } _{-} \rightarrow n) \ [])
 \end{aligned} \quad (1P)$$

- Induktionsannahme: Aussage gilt für $l = xs$ mit Länge $k \geq 0$

- Induktionsschritt: $l = x :: xs$ mit Länge $k + 1$ (0.5P)

$$\begin{aligned}
 & \text{acc} + \text{nlen } n \ (x :: xs) \\
 \stackrel{\text{nlen}}{=} & \text{acc} + \text{match } x :: xs \ \text{with } [] \rightarrow 0 \mid h :: t \rightarrow n + \text{nlen } n \ t \\
 \stackrel{\text{match}}{=} & \text{acc} + n + \text{nlen } n \ xs \\
 \stackrel{IA}{=} & \text{fold_left } (+) (\text{acc} + n) (\text{map} (\text{fun } _{-} \rightarrow n) \ xs) \\
 \stackrel{(+)}{=} & \text{fold_left } (+) ((+) \ \text{acc} \ n) (\text{map} (\text{fun } _{-} \rightarrow n) \ xs) \\
 \stackrel{\text{match}}{=} & \text{match } n :: \text{map} (\text{fun } _{-} \rightarrow n) \ xs \ \text{with } [] \rightarrow \text{acc} \\
 & \mid h :: t \rightarrow \text{fold_left } (+) ((+) \ \text{acc} \ h) \ t \\
 \stackrel{f=1}{=} & \text{fold_left } (+) \ \text{acc} (n :: \text{map} (\text{fun } _{-} \rightarrow n) \ xs) \\
 \stackrel{\text{fun}}{=} & \text{fold_left } (+) \ \text{acc} ((\text{fun } _{-} \rightarrow n) \ x :: \text{map} (\text{fun } _{-} \rightarrow n) \ xs) \\
 \stackrel{\text{match}}{=} & \text{fold_left } (+) \ \text{acc} (\text{match } x :: xs \ \text{with } [] \rightarrow [] \\
 & \mid h :: t \rightarrow (\text{fun } _{-} \rightarrow n) \ h :: \text{map} (\text{fun } _{-} \rightarrow n) \ t) \\
 \stackrel{\text{map}}{=} & \text{fold_left } (+) \ \text{acc} (\text{map} (\text{fun } _{-} \rightarrow n) \ (x :: xs))
 \end{aligned} \quad (1P) \quad (1P) \quad (1P) \quad (1P)$$

□

$\text{APP}, \frac{\pi_{nlen}}{\text{OP}}$	$\text{PM} \frac{\text{match } [] \text{ with } [] \rightarrow 0 \mid h :: t \rightarrow 2 + nlen \ 2 \ t \Rightarrow 0}{\frac{nlen \ 2 \ [] \Rightarrow 0}{\frac{2 + nlen \ 2 \ [] \Rightarrow 2}{\frac{\text{match } [3] \text{ with } [] \rightarrow 0 \mid h :: t \rightarrow 2 + nlen \ 2 \ t \Rightarrow 2}{\frac{nlen \ 2 \ [3] \Rightarrow 2}{\frac{[nlen \ 2 \ [3]] \Rightarrow [2]}{\text{LI}}}}}}}$
$\text{APP}, \frac{\pi_{nlen}}{\text{OP}}$	$\text{PM} \frac{\text{match } [] \text{ with } [] \rightarrow 0 \mid h :: t \rightarrow (\text{fun } a \rightarrow a * a) \ h : \text{map } (\text{fun } a \rightarrow a * a) \ t \Rightarrow []}{\frac{\text{map } (\text{fun } a \rightarrow a * a) \ t \Rightarrow []}{\frac{\text{map } (\text{fun } a \rightarrow a * a) \ 2 : \text{map } (\text{fun } a \rightarrow a * a) \ [] \Rightarrow [4]}{\frac{\text{map } (\text{fun } a \rightarrow a * a) \ 2 : \text{map } (\text{fun } a \rightarrow a * a) \ [] \Rightarrow [4]}{\frac{\text{match } [2] \text{ with } [] \rightarrow [] \mid h :: t \rightarrow (\text{fun } a \rightarrow a * a) \ h : \text{map } (\text{fun } a \rightarrow a * a) \ t \Rightarrow [4]}{\frac{\text{map } (\text{fun } a \rightarrow a * a) \ [nlen \ 2 \ [3]] \Rightarrow [4]}{\text{APP}, \frac{\pi_{map}}{\text{PI}}}}}}}$
$\text{APP}, \frac{\pi_{nlen}}{\text{OP}}$	$\text{PM} \frac{\text{match } [] \text{ with } [] \rightarrow 0 \mid h :: t \rightarrow (\text{fun } a \rightarrow a * a) \ h : \text{map } (\text{fun } a \rightarrow a * a) \ t \Rightarrow []}{\frac{\text{map } (\text{fun } a \rightarrow a * a) \ t \Rightarrow []}{\frac{\text{map } (\text{fun } a \rightarrow a * a) \ 2 : \text{map } (\text{fun } a \rightarrow a * a) \ [] \Rightarrow [4]}{\frac{\text{map } (\text{fun } a \rightarrow a * a) \ 2 : \text{map } (\text{fun } a \rightarrow a * a) \ [] \Rightarrow [4]}{\frac{\text{match } [2] \text{ with } [] \rightarrow [] \mid h :: t \rightarrow (\text{fun } a \rightarrow a * a) \ h : \text{map } (\text{fun } a \rightarrow a * a) \ t \Rightarrow [4]}{\frac{\text{map } (\text{fun } a \rightarrow a * a) \ [nlen \ 2 \ [3]] \Rightarrow [4]}{\text{APP}, \frac{\pi_{map}}{\text{PI}}}}}}}$

Lösungsvorschlag 4

Wir wählen die Schleifeninvariante $I := x = 3^i * i! \wedge y = 3i \wedge i \geq 0$

(2P+2P+1P).

$$\begin{aligned} & \text{WP}[\text{write}(x)](Z) \\ & \equiv \text{WP}[\text{write}(x)](x = 3^n * n!) \\ & \equiv x = 3^n * n! \quad \equiv: H \end{aligned} \quad (\text{Def. 1P})$$

$$\begin{aligned} & \text{WP}[\text{i} = \text{i} + 1](I) \\ & \equiv \text{WP}[\text{i} = \text{i} + 1](x = 3^i * i! \wedge y = 3i \wedge i \geq 0) \\ & \equiv x = 3^{i+1} * (i+1)! \wedge y = 3(i+1) \wedge i+1 \geq 0 \quad \equiv: G \end{aligned} \quad (\text{Def. 1P})$$

$$\begin{aligned} & \text{WP}[x = y * x](G) \\ & \equiv \text{WP}[x = y * x](x = 3^{i+1} * (i+1)! \wedge y = 3(i+1) \wedge i+1 \geq 0) \\ & \equiv y * x = 3^{i+1} * (i+1)! \wedge y = 3(i+1) \wedge i+1 \geq 0 \quad (\text{Def. 1P}) \\ & \Leftarrow x = \frac{3^{i+1} * (i+1)!}{y} \wedge y = 3(i+1) \wedge i \geq 0 \quad (\text{Aufl. 1P}) \\ & \equiv x = \frac{3^{i+1} * (i+1)!}{3(i+1)} \wedge y = 3(i+1) \wedge i \geq 0 \quad (\text{Eins. 1P}) \\ & \equiv x = 3^i * i! \wedge y = 3(i+1) \wedge i \geq 0 \quad \equiv: F \quad (\text{Simp. 1P}) \end{aligned}$$

$$\begin{aligned} & \text{WP}[y = y + 3](F) \\ & \equiv \text{WP}[y = y + 3](x = 3^i * i! \wedge y = 3(i+1) \wedge i \geq 0) \\ & \equiv x = 3^i * i! \wedge y + 3 = 3(i+1) \wedge i \geq 0 \quad (\text{Def. 1P}) \\ & \equiv x = 3^i * i! \wedge y = 3i \wedge i \geq 0 \quad \equiv: E \quad (\text{Simp. 1P}) \end{aligned}$$

$$\begin{aligned} & \text{WP}[\text{i} == \text{n}](E, H) \\ & \equiv \text{WP}[\text{i} == \text{n}](x = 3^i * i! \wedge y = 3i \wedge i \geq 0, x = 3^n * n!) \\ & \equiv (i \neq n \wedge x = 3^i * i! \wedge y = 3i \wedge i \geq 0) \vee (i = n \wedge x = 3^n * n!) \quad (\text{Def. 1P}) \\ & \equiv (i \neq n \wedge x = 3^i * i! \wedge y = 3i \wedge i \geq 0) \vee (i = n \wedge x = 3^i * i!) \quad (\text{Subst. 1P}) \\ & \Leftarrow (i \neq n \wedge x = 3^i * i! \wedge y = 3i \wedge i \geq 0) \vee (i = n \wedge x = 3^i * i! \wedge y = 3i \wedge i \geq 0) \quad (\text{Verst. 2P}) \\ & \equiv x = 3^i * i! \wedge y = 3i \wedge i \geq 0 \wedge (i \neq n \vee i = n) \\ & \equiv x = 3^i * i! \wedge y = 3i \wedge i \geq 0 \quad \equiv: I \quad (\text{I zeigen 1P}) \end{aligned}$$

$$\begin{aligned} & \text{WP}[y = 0](I) \\ & \equiv \text{WP}[y = 0](x = 3^i * i! \wedge y = 3i \wedge i \geq 0) \\ & \equiv x = 3^i * i! \wedge 0 = 3i \wedge i \geq 0 \quad \equiv: D \quad (\text{Def. 1P}) \end{aligned}$$

$$\begin{aligned} & \text{WP}[x = 1](D) \\ & \equiv \text{WP}[x = 1](x = 3^i * i! \wedge 0 = 3i \wedge i \geq 0) \\ & \equiv 1 = 3^i * i! \wedge 0 = 3i \wedge i \geq 0 \quad \equiv: C \quad (\text{Def. 1P}) \end{aligned}$$

$$\begin{aligned}
 & \text{WP}[i = 0](C) \\
 & \equiv \text{WP}[i = 0](1 = 3^i * 0 \wedge 0 = 3i \wedge i \geq 0) \\
 & \equiv 1 = 3^0 * 0 \wedge 0 = 3 * 0 \wedge 0 \geq 0 \\
 & \equiv \text{true} \equiv B
 \end{aligned}
 \quad (\text{Def. } 1P)$$

$$\begin{aligned}
 & \text{WP}[n = \text{read}()](B) \\
 & \equiv \text{WP}[n = \text{read}()](\text{true}) \\
 & \equiv \text{true} \equiv A
 \end{aligned}
 \quad (\text{Def. } 1P)$$

Weitere Korrekturhinweise:

- In dieser Aufgabe gibt es maximal (22P).
- Eine bis auf $i \geq 0$ korrekte Lösung gibt (21P).
- Ungültige oder zusätzliche Aussagen in der Invariante geben keinen direkten Punkt abzug.
- Ein $i \leq n$ (oder ähnliches) führt aber zu $\forall n. 0 \leq n \equiv \text{false}$ im letzten Schritt und gibt dort dann (-0.5P) bzw. (-1P).
- Die Definition des WP-Operators richtig einsetzen gibt (1P).
 - Ist die Aussage so einfach, z.B. da die entsprechende Variable gar nicht vorkommt, gibt es nur (0.5P).
 - Wird bei der Verzweigung lediglich $i \neq n \wedge E \vee i = n \wedge H$ angegeben, gibt es in der Regel (0.5P).
- Werden Umformungen oder Vereinfachungen in einem anderen (späteren) Schritt durchgeführt gibt es die Punkte dort.
 - Die Punkte sind dann meist als (+1P) angegeben.
- Werden sehr viele Umformungen und Vereinfachungen in einem Schritt durchgeführt, so gab es die Punkte in der Regel nicht.
 - z.B. $\text{WP}[i == n](E, H) \Leftarrow I$ gibt dann gar keine Punkte.
- Macht man in einem Schritt einen Fehler, so gibt es die entsprechenden Punkte nicht, alle nachfolgenden Schritte geben dann aber wieder Punkte, wenn sie (ausgehend aus der falschen Aussage) richtig durchgeführt werden.
- Werden fehlerhafte Aussagen korrekt umgeformt und vereinfacht, gibt es dennoch die Punkte.
- Ist die gesamte Lösung (schon vom Lösungsansatz her) grundlegend unsinnig¹, so wurden zusätzliche Punkte abgezogen.

¹Wir entschuldigen uns für den groben Kommentar, der sich dann möglicherweise an der Lösung befindet!

Lösungsvorschlag 6

```
(* 6.1 *)
let rec atoms a = function Atom e -> if e = a then 1 else 0
| Bond bs -> fold_left (fun s (b,c) -> s + c * atoms a b) 0 bs

(* 6.2 Variante: atoms *)
let rec elems = function Atom e -> [e]
| Bond bs -> map (elems % fst) bs |> flatten

let is_balanced { reacts; prods } =
  let cnt e l = fold_left (fun s (c,b) -> s + c * atoms e b) 0 l in
  (reacts @ prods) |> map (elems % snd) |> flatten
  |> for_all (fun e -> cnt e reacts = cnt e prods)

(* 6.2 Variante: assoc list *)
let is_balanced { reacts; prods } =
  let rec aux acc (c,b) = match b with Atom e ->
    let n,rest = try assoc e acc, remove_assoc e acc
    with _ -> 0,acc in (e,n+c)::rest
  | Bond bs -> fold_left (fun acc (b,c') -> aux acc (c*c',b)) acc bs
  in
  let cnts = fold_left aux [] reacts in
  let cnts = fold_left (fun acc (c,b) -> aux acc (-c,b)) cnts prods in
  for_all (fun (_,c) -> c = 0) cnts
```

Korrekturhinweise:

- Teilaufgabe 1 gibt maximal (10P).
 - je (1P) für matchen auf **Atom** und **Bond**.
 - (2P) für das Prüfen auf das Element im **Atom** Fall.
 - (6P) für die Rekursion über die Liste im **Bond** Fall, davon (2P) für die korrekte Verarbeitung des Index.
- Teilaufgabe 2 gibt maximal (14P).
 - kein genaues Schema, da Lösungen sehr verschieden.
 - (1P) für den korrekten Record-Zugriff.
 - (5-6P) für das extrahieren der Atome (falls dieser Ansatz).
 - (5-6P) für korrekten Umgang mit assoziativen Listen (falls dieser Ansatz)