Prof. Dr. Ralf Hinze
Cass Alexandru, M.Sc.
Alexander Dinges, M.Sc.
Sebastian Schloßer, M.Sc.

RPTU
Fachbereich Informatik
AG Programmiersprachen

Lösungshinweise/-vorschläge zum Übungsblatt 4: Grundlagen der Programmierung (WS 2024/25)

Klausuranmeldung Denken Sie daran, die Klausur und die Vorleistung im Prüfungsamt anzumelden!

Beweisbaum Werkzeug und Rekursions-Tutor Wir haben zwei digitale Hilfsmittel entwickelt, die Sie beim Lernen und Bearbeiten der Hausaufgaben unterstützen sollen. Das Beweisbaum Werkzeug erlaubt es, Beweisbäume für die Statische und Dynamische Semantik digital zu erstellen, wobei die Korrektheit der Bäume sichergestellt wird. Der Rekursions-Tutor hilft Ihnen dabei, die Auswertung von rekursiven Funktionsaufrufen besser nachzuvollziehen. Auch letztes Jahr standen den Studierenden teilweise diese Hilfsmittel zur Verfügung. Dabei hatte sich gezeigt, dass diejenigen, die im Zeitraum bis zur Probeklausur Beweisbäume ausschließlich digital erstellt haben, in der Probeklausur unter Klausurbedingungen auf Papier erheblich schlechter abgeschnitten haben als die anderen, die ihre Beweisbäume bereits in den Hausaufgaben ohne technische Hilfe erstellen mussten. Deshalb und weil wir auch dieses Jahr wieder evaluieren möchten, inwiefern die Hilfsmittel Ihren Lernerfolg unterstützt haben, stehen die Hilfsmittel nur zeitweise zur Verfügung und sich müssen zwischenzeitlich Hausaufgaben auch ohne die Hilfsmittel bearbeiten.

Der Evaluationszeitraum läuft vier Wochen lang, nämlich für die Bearbeitungszeit der Übungsblätter 4 bis 7. Bei zwei der Übungsblättern können Sie das Beweisbaum Werkzeug nutzen, bei den anderen beiden den Rekursions-Tutor. Für dieses Übungsblatt haben die Übungsgruppen 1, 3, 5, 7, 9 und 11 den Rekursions-Tutor (hilfreich bei Aufgabe 4), die Übungsgruppen 2, 4, 6, 8 und 10 das Beweisbaum Werkzeug (hilfreich bei Aufgabe 5). Dann wird jede Woche gewechselt. Der Zugriff startet jeweils mit Beginn der Übungsstunde, in der die Präsenzaufgaben behandelt werden, und endet mit der Abgabefrist. Mit Beginn der Weihnachtsferien werden beide Hilfsmittel allen Studierenden dauerhaft freigeschaltet.

Sie finden die Hilfsmittel im ExClaim-System auf der “GdP24” Seite. Dort erscheint oben auf der Seite ein Button “Beweisbaum-Tool” bzw. “Recursion Tutor”. Die Buttons sind nur in dem für Ihre Übungsgruppe bestimmten Nutzungszeitraum für das jeweilige Hilfsmittel sichtbar.

Wir bitten Sie, sich an die vorgegebene Nutzungszeiträume zu halten. Einerseits ist das für Sie selbst wichtig, damit Sie auf die Klausur optimal vorbereitet sind, andererseits würden Sie unsere Studie verfälschen, wenn Sie die vorgesehenen Nutzungszeiträume umgehen.

Aufgabe 1 Datentypen (Präsenzaufgabe)

Motivation: Um Programmieraufgaben im ExClaim-System testen zu können, müssen wir die darin verwendeten Typdefinitionen vorgeben. In dieser Aufgabe sollen Sie (rekursive) Varianten und Records wiederholen und selbst entsprechende Typen definieren, um Sachverhalte zu modellieren. Sie können sich an den Vorlesungsfolien 280 bis 372 sowie am Skript Kapitel 4.1 und 4.2 orientieren.

a) Varianten

1. Definieren Sie einen Variantentyp zur Modellierung von Tierarten, der die Ausprägungen `Hund`, `Katze` und `Maus` annehmen kann.

   Zusätzlich soll ein Variantentyp für Lebewesen definiert werden: Bei einem Lebewesen kann es sich entweder um ein Tier einer der oben genannten Tierarten handeln, oder um ein Lebewesen, das kein Tier ist.

```
type Tierart =
    | Hund
    | Katze
    | Maus

type Lebewesen =
    | KeinTier
    | Tier of Tierart
```

2. Schreiben Sie eine Funktion `eineMaus: Tierart -> Bool`, die prüft, ob es sich bei der übergebenen Tierart um eine Maus handelt.

```
let eineMaus (t: Tierart): Bool =
    match t with
    | Maus -> true
    | _ -> false
```

3. Schreiben Sie eine Funktion `eineKatze: Lebewesen -> Bool`, die prüft, ob es sich beim übergenenen Lebewesen um eine Katze handelt.

```
let eineKatze (l: Lebewesen): Bool =
    match l with
    | KeinTier -> false
    | Tier t ->
        match t with
        | Katze -> true
        | _ -> false

let eineKatze' (l: Lebewesen): Bool =
    match l with
    | Tier Katze -> true
    | _ -> false
```

4. Schreiben Sie eine Funktion `mindestensEinTier: Lebewesen -> Lebewesen -> Bool`, die prüft, ob es sich bei mindestens einem der beiden Argumente um ein Tier handelt.

```
let mindestensEinTier (l1: Lebewesen) (l2: Lebewesen): Bool =
    match (l1, l2) with
    | (KeinTier, KeinTier) -> false
    | _ -> true
```

b) Rekursive Varianten

1. Wiederholen Sie den Typ `Nats` für Listen natürlicher Zahlen

```
type Nats =
    | Nil
    | Cons of Nat * Nats
```

2. Schreiben Sie eine Funktion `findMax`, welche die größte Zahl in einer Liste natürlicher Zahlen zurückgibt.

```
let rec findMax (ns: Nats): Nat =
    match ns with
    | Nil -> 0N
    | Cons (x, xs) -> max x (findMax xs)
```

3. Schreiben Sie eine Funktion `plusOne`, die alle Zahlen in der Liste um eins erhöht.

```
let rec plusOne (ns: Nats): Nats =
    match ns with
    | Nil -> Nil
    | Cons (x, xs) -> Cons (x + 1, plusOne xs)
```

c) Records

1. Schreiben Sie einen Record-Typ, um Eigenschaften von Büchern zu speichern. Gesichert werden soll der Titel, der Name der Autorin/des Autors, das Veröffentlichungsjahr sowie die ISBN.

   Verwenden Sie den Record, um das Buch „Harry Potter and the Philosopher's Stone“ von „J. K. Rowling“ aus dem Jahr 1997 mit der ISBN 9780747532743 zu erfassen.

```
type Buch = {titel: String ; autor: String ; jahr: Nat ; isbn: Nat}
let harry = { titel="Harry Potter and the Philosopher's Stone"
            ; autor="J. K. Rowling" ; jahr = 1997N ; isbn = 9780747532743N }
```

2. Wo liegt der Vorteil gegenüber einem entsprechenden Quadrupel?

```
type Buch' = String * String * Nat * Nat
```

Hier müssen wir uns merken, welche Information an welcher Position steht. Durch die Angabe von Labels ist die Reihenfolge bei Records egal. Aussagekräftige Labels schaffen zusätzliche Klarheit beim Lesen und Schreiben des Codes.

Aufgabe 2 Modellierung von Datentypen (Einreichaufgabe, 3 Punkte)

Motivation: In dieser Aufgabe sollen Sie sich mit Variantentypen und Records beschäftigen. Sie können sich an den Vorlesungsfolien 280 bis 326 sowie am Skript Kapitel 4.1 und 4.2 orientieren.

Schreiben Sie Ihre Lösungen in die Datei `Mensa.fs` *aus der Vorlage* `Aufgabe-4-2.zip`.

Definieren Sie geeignete Datentypen, um einzelne Gerichte der Mensa zu modellieren. Relevant sind folgende Daten: Name des Gerichts (nutzen Sie hierfür den Typ `String`), Preis in Cents, Ausgabeort sowie, ob das Gericht vegan, vegetarisch oder nicht-vegetarisch ist. Ausgabeorte können Ausgabe 1, Ausgabe 2, Wok, Grill, Salatbüffet und Atrium sein. Verwenden Sie möglichst genaue Datentypen, es ist also nicht zulässig, alles als `String` darzustellen.

Hinweis: Sie sollen keine Funktionen implementieren, sondern lediglich Typdefinitionen angeben. Zu dieser Aufgabe gibt es deshalb auch keine Testfälle!

```
type Ausgabe = | AusgabeEins | AusgabeZwei | Wok | Grill | Salatbueffet | Atrium
type Kostform = | Vegetarisch | Vegan | Nichtvegetarisch

type Gericht = { name: String; preis: Nat; ort: Ausgabe; kostform: Kostform }
```

Aufgabe 3 Listen natürlicher Zahlen (Einreichaufgabe, 6 Punkte)

Motivation: In dieser Aufgabe sollen Sie sich mit rekursiven Variantentypen beschäftigen. Sie können sich an den Vorlesungsfolien 332 bis 351 sowie am Skript Kapitel 4.2.2 orientieren.

Schreiben Sie Ihre Lösungen in die Datei `Nats.fs` *aus der Vorlage* `Aufgabe-4-3.zip`.

Bevor wir nächste Woche den in F# eingebauten Typ für Listen kennenlernen, arbeiten wir diese Woche zunächst mit einem selbst definierten Typ für Listen natürlicher Zahlen. Alles was wir dazu brauchen, sind rekursive Varianten und Tupel. Sie kennen den Typ bereits aus der Vorlesung:

```
type Nats = | Nil | Cons of Nat * Nats
```

Hinweis: Wenn Sie im Internet nach Teilen der Aufgabenstellung suchen, werden Sie vielleicht auf das `List` *F#-Modul stoßen, das allerdings mit dem in F# eingebauten Typ für Listen arbeitet. Dieses Modul werden wir auf einem späteren Übungsblatt vorstellen, hier dürfen Sie es in Ihrer Lösung jedoch* ***nicht*** *verwenden.*

Wir verwenden bei einigen Teilaufgaben die Beispielliste:

```
let ex = Cons (2N, Cons (4N, Cons (3N, Cons(4N, Cons(2N, Cons (1N, Nil))))))
```

a) Schreiben Sie eine Funktion `trace`, welche eine Funktion des Typs `Nat -> Nat` und einen Startwert des Typs `Nat` als Argument erhält, und die Funktion auf den Startwert, und danach sukzessive auf alle Zwischenergebnisse anwendet, bis sie `0N` oder `1N` produziert. Dabei sollen alle Zwischenergebnisse als Liste zurückgegeben werden. Wir schreiben der Lesbarheit halber die Listen hier mit einer vereinfachten Syntax.

Beispiele:

```
trace (fun x -> x - 1N) 5N = [5N;4N;3N;2N;1N]
trace (fun x -> x - 1N) 0N = [0N]
trace (fun x -> x - 1N) 1N = [1N]
trace (fun x -> x - 2N) 6N = [6N;4N;2N;0N]
trace (fun x -> if x % 2N = 0N then x / 2N else 3N * x + 1N) 5N = [5N; 16N; 8N; 4N; 2N; 1N]
```

```
let rec trace (f : Nat -> Nat) (start : Nat): Nats =
  Cons (start,
        if start = 0N || start = 1N then Nil
        else trace f (f start)
        )
```

Terminiert diese Funktion immer?

b) Schreiben Sie eine Funktion `isSortedBy: (lessEqual : Nat * Nat -> Bool ) -> Nats -> Bool`, die prüft, ob die gegebene Liste bezüglich der gegebenen Vergleichsfunktion *aufsteigend* sortiert ist.

Beispiele:

```
isSortedBy (fun (m, n) -> m <= n) Nil = true
isSortedBy (fun (m, n) -> m <= n) (Cons (1N, Cons (2N, Nil))) = true
isSortedBy (fun (m, n) -> m < n) (Cons (1N, Cons (1N, Nil))) = false
isSortedBy (fun (m, n) -> m >= n) (Cons (2N, Cons (1N, Nil))) = true
isSortedBy (fun (m, n) -> m <= n) ex = false
```

```
let rec isSortedBy (lessEqual : Nat * Nat -> Bool ) (xs: Nats): Bool =
    match xs with
    | Nil -> true
    | Cons (_, Nil) -> true
    | Cons (x, Cons (y, xs)) -> lessEqual (x,y) && isSortedBy lessEqual (Cons (y,
     xs))
```

c) Schreiben Sie eine Funktion `exists: (Nat -> Bool) -> Nats -> Bool`, die ein Prädikat auf den natürlichen Zahlen (also eine Funktion, die eine natürliche Zahl nimmt und prüft, ob diese eine bestimmte Bedingung erfüllt) und eine Liste von natürlichen Zahlen nimmt und zurückgibt, ob irgendeine Zahl dieses erfüllt.

Beispiele:

```
exists p Nil = false
exists (fun n -> n%2=0) (Cons (2N, Cons (1N, Nil))) = true
exists (fun n -> n%2=0) (Cons (1N, Cons (3N, Nil))) = false
exists (fun n -> n>3) ex = true
```

```
let rec exists (p: Nat -> Bool) (xs: Nats): Bool =
    match xs with
    | Nil -> false
    | Cons (x, xs) -> p x || exists p xs
```

Aufgabe 4 Rekursion mit rekursiven Varianten (Einreichaufgabe, 6 Punkte)

Motivation: Als Teil einer vorlesungsbegleitenden Studie wollen wir untersuchen, welches mentale Modell von Rekursion die Studierenden haben. Bitte bearbeiten Sie diese Aufgabe daher gewissenhaft und ohne fremde Hilfe.

Hinweis: Wenn Sie in Übungsgruppe 1, 3, 5, 7, 9 oder 11 sind, dann steht Ihnen zur Bearbeitung dieser Aufgabe unser Rekursions-Tutor zur Verfügung, siehe Hinweis auf der ersten Seite.

Betrachten Sie die erneut die Definition des Typs `Nats` von Vorlesungsfolie 334 und Aufgabe 3. Betrachten Sie dann die nachfolgenden rekursiven Funktionen. Was ist das Ergebnis des jeweiligen Funktionsaufrufs? Zeigen Sie Ihre Vorgehensweise, indem Sie alle gemachten Schritte aufschreiben und gegebenenfalls erklären. **Das Endergebnis allein gibt keine Punkte! Erstellen Sie *keinen Beweisbaum*!**

Bearbeiten Sie diese Aufgabe auf Papier oder schreiben Sie Ihre Schritte digital in eine pdf-Datei.

Wir definieren eine Nats-Liste, die wir gleich verwenden:

```
let ns = Cons (3, Cons (1, Cons (2, Nil)))
```

a) Funktionsaufruf: `f ns`

```
let rec f (ns: Nats): Nats =
  match ns with
  | Nil          -> Cons (0, Nil)
  | Cons (x, xs) -> Cons (2 * x, f xs)
```

Ein möglicher sinnvoller Lösungsweg ist der folgende tabellarische Ansatz:

Schritt	ns	Ausdruck	Erklärung
1	Cons (3, Cons (1, Cons (2, Nil)))	Cons (2 * 3, f (Cons (1, Cons (2, Nil))))	f ns wird zu Cons (2 * 3, f (Cons (1, Cons (2, Nil)))) aufgelöst
2	Cons (1, Cons (2, Nil))	Cons (2 * 1, f (Cons (2, Nil)))	f (Cons (1, Cons (2, Nil))) wird zu Cons (2 * 1, f (Cons (2, Nil))) aufgelöst
3	Cons (2, Nil)	Cons (2 * 2, f Nil)	f (Cons (2, Nil)) wird zu Cons (2 * 2, f Nil) aufgelöst
4	Nil	Cons (0, Nil)	f Nil wird zu Cons (0, Nil) aufgelöst (Basisfall)
5		Cons (2 * 2, Cons (0, Nil))	Rückgabewert von f Nil in Schritt 3 einsetzen
6		Cons (2 * 1, Cons (2 * 2, Cons (0, Nil)))	Rückgabewert von f (Cons (2, Nil)) in Schritt 2 einsetzen
7		Cons (2 * 3, Cons (2 * 1, Cons (2 * 2, Cons (0, Nil))))	Rückgabewert von f (Cons (1, Cons (2, Nil))) in Schritt 1 einsetzen
8		Cons (6, Cons (2, Cons (4 Cons (0, Nil))))	Endergebnis

b) Funktionsaufruf: `g ns`

```
let rec h (n: Nat): Nat =
    if n = 0 then 1
    else 2 * h (n - 1)

let rec g (ns: Nats): Nat =
  match ns with
  | Nil          -> 1
  | Cons (x, xs) -> h x + g xs
```

Ein möglicher sinnvoller Lösungsweg ist der folgende tabellarische Ansatz:

Schritt	ns	Ausdruck	Erklärung
1	Cons (3, Cons (1, Cons (2, Nil)))	h 3 + g (Cons (1, Cons (2, Nil)))	g ns wird zu h 3 + g (Cons (1, Cons (2, Nil))) aufgelöst
2	Cons (1, Cons (2, Nil))	h 1 + g (Cons (2, Nil))	g (Cons (1, Cons (2, Nil))) wird zu h 1 + g (Cons (2, Nil)) aufgelöst
3	Cons (2, Nil)	h 2 + g Nil	g (Cons (2, Nil)) wird zu h 2 + g Nil aufgelöst
4	Nil	1	g Nil wird zu 1 aufgelöst (Basisfall)
5		h 2 + 1	Rückgabewert von g Nil in Schritt 3 einsetzen
6		h 1 + (h 2 + 1)	Rückgabewert von g (Cons (2, Nil)) in Schritt 2 einsetzen
7		h 3 + (h 1 + (h 2 + 1))	Rückgabewert von g (Cons (1, Cons (2, Nil))) in Schritt 1 einsetzen
8		2 * (2 * 2) + (2 + (2 * 2 + 1))	Aufrufe von h ausrechnen und einsetzen
9		15	Endergebnis

c) Erklären Sie in ein bis zwei Sätzen, warum die Funktion g aus Aufgabenteil b) für Nats-Listen von natürlichen Zahlen immer terminiert.

Wichtige Aspekte:

- Liste wird mit jedem Rekursionsschritt um ein Element verkürzt
- Die leere Liste ist der Basisfall, der so immer erreicht wird
- Die Hilfsfunktion `h` terminiert ebenfalls immer, da in jedem Rekursionsschritt `n` um 1 verringert wird, bis der Basisfall `n = 0` erreicht ist

Aufgabe 5 Dynamische und Statische Semantik (Einreichaufgabe, 7 Punkte)

Motivation: In dieser Aufgabe sollen Sie die Semantikregeln rückwärts anwenden: Im ersten Teil ist die linke Seite (der Ausdruck) unbekannt, aber die rechte Seite (der Wert) vorgegeben. Im zweiten Teil arbeiten Sie wieder in der gewohnten Richtung, der Ausdruck ist nun bekannt und es ist der dazugehörige Typ gesucht. Sie benötigen die Regeln der Vorlesungsfolien 109-110, 138-139, 143-144 und 182-183.

Hinweis: Wenn Sie in Übungsgruppe 2, 4, 6, 8 oder 10 sind, dann steht Ihnen zur Bearbeitung dieser Aufgabe unser Beweisbaum Werkzeug zur Verfügung, siehe Hinweis auf der ersten Seite. Sie müssen das Sprachfeature "Wertedefinitionen & Funktionen" aktivieren. Nach dem Einloggen im ExClaim-System führt dieser Link direkt zum vorbereiteten Baum für den ersten Teil dieser Aufgabe. Wenn Sie möchten, können Sie einen Screenshot des erstellten Beweisbaumes in Ihre Abgabe mit einbinden. Falls Sie bereits Erfahrung mit LaTeX haben, dürfen Sie die entsprechende Exportfunktion nutzen und eine kompilierte pdf-Datei abgeben. Der LaTeX-Quellcode oder die durch den Button "Baum speichern" erzeugte json-Datei wird jedoch nicht als Abgabe anerkannt! Alternativ können Sie die Abgabe wie gewohnt auf Papier aufschreiben.

Finden Sie einen Ausdruck e, der gemäß den Regeln der Dynamischen Semantik aus der Vorlesung zu folgendem Wert auswertet:

$$\langle \{a \mapsto 5\}, x, a + x \rangle$$

Geben Sie einen Beweisbaum mit den Regeln der Dynamischen Semantik an, der

$$\emptyset \vdash e \Downarrow \langle \{a \mapsto 5\}, x, a + x \rangle$$

zeigt. Bestimmen Sie anschließend den Typ t Ihres Ausdrucks e und geben Sie einen Beweisbaum mit den Regeln der Statischen Semantik an, der

$$\emptyset \vdash e : t$$

zeigt.

Ausdruck $e :=$ `let a = 5 in fun x → a + x` Link zum Baum

$$\dfrac{\dfrac{}{\emptyset \vdash 5 \Downarrow 5}}{\emptyset \vdash \texttt{let a = 5} \Downarrow \{\texttt{a} \mapsto 5\}} \quad \dfrac{}{\{\texttt{a} \mapsto 5\} \vdash \texttt{fun x → a + x} \Downarrow \langle \{\texttt{a} \mapsto 5\}, \texttt{x}, \texttt{a + x} \rangle}$$
$$\emptyset \vdash \texttt{let a = 5 in fun x → a + x} \Downarrow \langle \{\texttt{a} \mapsto 5\}, \texttt{x}, \texttt{a + x} \rangle$$

Typ $t :=$ Nat → Nat Link zum Baum

$$\dfrac{\dfrac{\dfrac{}{\emptyset \vdash 5 : \text{Nat}}}{\emptyset \vdash \texttt{let a = 5} : \{\texttt{a} \mapsto \text{Nat}\}} \quad \dfrac{\dfrac{}{\{\texttt{a} \mapsto \text{Nat}, \texttt{x} \mapsto \text{Nat}\} \vdash \texttt{a} : \text{Nat}} \quad \dfrac{}{\{\texttt{a} \mapsto \text{Nat}, \texttt{x} \mapsto \text{Nat}\} \vdash \texttt{x} : \text{Nat}}}{\{\texttt{a} \mapsto \text{Nat}, \texttt{x} \mapsto \text{Nat}\} \vdash \texttt{a + x} : \text{Nat}}}{\{\texttt{a} \mapsto \text{Nat}\} \vdash \texttt{fun (x: Nat) → a + x} : \text{Nat} \to \text{Nat}}$$
$$\emptyset \vdash \texttt{let a = 5 in fun (x: Nat) → a + x} : \text{Nat} \to \text{Nat}$$

Hinweis: Es sind auch andere Ausdrücke möglich, die `5` lässt sich beispielsweise durch `2 + 3` ersetzen, dann werden die Bäume jedoch größer.

Aufgabe 6 Statische Semantik von rekursiven Funktionen (Trainingsaufgabe)

Wir betrachten die rekursive Funktionsdefinition

```
let rec f  (x: Bool): Bool = (if x then x else f (not x))
```

bezüglich der Signatur $\Sigma := \{\texttt{not} \mapsto \text{Bool} \to \text{Bool}\}$. Geben Sie einen vollständigen Beweisbaum an.

Definiere aus Platzgründen $\Sigma_1 := \{\texttt{not} \mapsto \text{Bool} \to \text{Bool}, \texttt{f} \mapsto \text{Bool} \to \text{Bool}, \texttt{x} \mapsto \text{Bool}\}$.

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\Sigma_1 \vdash \texttt{x} : \text{Bool}} \quad \dfrac{}{\Sigma_1 \vdash \texttt{x} : \text{Bool}} \quad \dfrac{\dfrac{}{\Sigma_1 \vdash \texttt{f} : \text{Bool} \to \text{Bool}} \quad \dfrac{\dfrac{}{\Sigma_1 \vdash \texttt{not} : \text{Bool} \to \text{Bool}} \quad \dfrac{}{\Sigma_1 \vdash \texttt{x} : \text{Bool}}}{\Sigma_1 \vdash \texttt{not x} : \text{Bool}}}{\Sigma_1 \vdash \texttt{f (not x)} : \text{Bool}}}{\Sigma_1 \vdash \texttt{if x then x else f (not x)} : \text{Bool}}}{\Sigma \vdash \texttt{let rec f (x: Bool): Bool = if x then x else f (not x)} : \{\texttt{f} \mapsto \text{Bool} \to \text{Bool}\}}$$