Einführung in die Informatik 2

Prof. Dr. Seidl, J. Kranz, N. Hartmann, J. Brunner
WS 2016/17

Übungsblatt 7
Abgabefrist: 19.12.2016

Aufgabe 7.1 (P) Unendlich faule Listen

Das bereits bekannte Konzept der Listen soll in dieser Aufgabe so erweitert werden, dass unendliche Listen von Werten erzeugt und verwendet werden können. Überlegen Sie, wie ein Typ `'a llist` aussehen müsste um eine unendliche Folge von Elementen des Typs `'a` darzustellen.

```
1 type 'a llist = (* ? *)
```

Betrachten Sie die folgenden Funktionen und entscheiden Sie, ob deren Implementierung für unendliche Listen möglich und sinnvoll ist. Falls ja, implementieren Sie die Funktion und verwenden Sie Endrekursion, wo dies möglich ist.

1. Die Funktion

   ```
   val lseq : int -> int llist
   ```

   die eine unendliche Folge von ganzen Zahlen erzeugt. Die Folge soll mit der übergebenen Zahl beginnen.

2. Die Funktion

   ```
   val lconst : int -> int llist
   ```

   die eine unendliche Folge der übergebenen Zahl erzeugt.

3. Die Funktion

   ```
   val lpowers2 : unit -> int llist
   ```

   die 2er-Potenzen (beginnend mit 1) erzeugt.

4. Die Funktion

   ```
   val lrng : int -> int llist
   ```

   die eine Folge zufälliger Werte erzeugt. Diese Werte sollen dabei zwischen 0 und dem übergebenen Maximalwert (inklusive) liegen. Eine zufällige Ganzzahl aus dem Interval $[0, n)$ lässt sich in OCaml mit der Funktion `Random.int n` erzeugen.

5. Die Funktion

   ```
   val lfib : unit -> int llist
   ```

   welche die Fibonacci-Folge (beginnend mit 0 und 1) erzeugt.

6. Die Funktion

   ```
   val lhd : 'a llist -> 'a
   ```

   die das erste Element (Head) der Liste zurückgibt.

7. Die Funktion

```
val ltl : 'a llist -> 'a llist
```

die den Rest der Liste (Tail) liefert.

8. Die Funktion

```
val ltake : int -> 'a llist -> 'a list
```

die für ein übergebenes n eine endliche Liste mit den ersten n Elementen liefert.

9. Die Funktion

```
val ldrop : int -> 'a llist -> 'a llist
```

die für ein übergebenes n die ersten n Elemente der Liste entfernt.

10. Die Funktion

```
val lappend : 'a llist -> 'a llist -> 'a llist
```

die zwei Listen aneinander hängt.

11. Die Funktion

```
val lnth : int -> 'a llist -> 'a
```

die das n-te Element der Liste zurückgibt.

12. Die Funktion

```
val lreverse : 'a llist -> 'a llist
```

welche die Reihenfolge der Elemente der Liste umdreht.

13. Die Funktion

```
val lfilter : ('a -> bool) -> 'a llist -> 'a llist
```

die eine neue Liste zurückgibt, die nur noch diejenigen Elemente enthält, die das übergebene Prädikat erfüllen.

14. Die Funktion

```
val lmap : ('a -> 'b) -> 'a llist -> 'b llist
```

die alle Elemente mit Hilfe der übergebenen Funktion transformiert und daraus eine neue Liste erzeugt.

15. Die Funktion

```
val linterleave : 'a llist -> 'a llist -> 'a llist
```

die ein Interleaving der beiden übergebenen Listen erzeugt. So soll zum Beispiel aus den beiden Listen `[1; 2; 3; 4; 5; 6; ...]` und `[20; 20; 20; 20; 20; 20; ...]` die neue Liste `[1; 20; 2; 20; 3; 20; 4; 20; 5; 20; 6; 20; ...]` entstehen.

16. Die Funktion

```
val linterleaven : 'a llist list -> 'a llist
```

die als eine Verallgemeinerung der Funktion `linterleave` zu verstehen ist und ein Interleaving einer beliebigen Anzahl von Listen erzeugen kann.

17. Die Funktion

```
val lrepeat : 'a list -> 'a llist
```

die eine Liste erzeugt, welche die Elemente der übergebenen (endlichen) Liste zyklisch wiederholt. Aus der Liste `[1; 2; 3]` wird also die unendliche Folge `[1; 2; 3; 1; 2; 3; 1; 2; 3; ...]` erzeugt.

18. Die Funktion

```
val lprimes : unit -> int llist
```

die alle Primzahlen in aufsteigender Reihenfolge erzeugt.

19. Die Funktion

```
val lbinary : unit -> int list llist
```

die alle Wörter über dem Alphabet $\{0, 1\}$, also alle Folgen von 0 und 1 erzeugt. Ein Wort wird dabei als (endliche) `int list` repräsentiert. Das Wort 0010 kommt also irgendwo in der zurückgegebenen Liste `[... [0; 0; 1; 0]; ...]` vor.

20. Betrachten Sie die Funktion `val lmagic : unit -> float llist`. Was könnte diese Funktion erzeugen? *Hinweis: Probieren Sie es aus!*

```
let lmagic () =
  let rec lmagic_impl a b =
    let d = float_of_int a in
    let c = b +. ((16. ** (-.d)) *. ((4. /. (8. *. d +. 1.))
      -. (2. /. (8. *. d +. 4.)) -. (1. /. (8. *. d +. 5.))
      -. (1. /. (8. *. d +. 6.)))) in
    Cons ((int_of_float (c *. (10. ** (float_of_int a)))) mod 10,
      fun () -> lmagic_impl (a+1) c)
  in lmagic_impl 0 0.
```

Lösungsvorschlag 7.1

Die Lösung befindet sich in der Datei `ocaml/p7_sol.ml`.

Allgemeine Hinweise zur Hausaufgabenabgabe

Die Hausaufgabenabgabe bezüglich dieses Blattes erfolgt über das Ocaml-Abgabesystem. Sie erreichen es unter `https://vmnipkow3.in.tum.de`. Sie können hier Ihre Abgaben einstellen und erhalten Feedback, welche Tests zu welchen Aufgaben erfolgreich durchlaufen. Sie können Ihre Abgabe beliebig oft testen lassen. Bitte beachten Sie, dass Sie Ihre Abgabe stets als **eine einzige UTF8-kodierte Textdatei mit dem Namen *ha7.ml* hochladen müssen**. Die hochgeladene Datei muss ferner den Signaturen in *ha6.mli* entsprechen.

Hinweis zu Endrekursion

Wenn Sie zur Implementierung einer endrekursiven Funktion Funktionen aus der Standard-Bibliothek verwenden, müssen Sie unbedingt darauf achten, dass diese selbst endrekursiv sind. Es muss für Ihre Implementierung also gelten, dass sich die maximal nötige Stacktiefe durch eine Konstante begrenzen lässt.

Aufgabe 7.2 (H) Hyperfibonacci [3 Punkte]

Wir definieren die Hyperfibonacci-Funktion für $k \in \mathbb{N}$ und $n \in \mathbb{N}_0$:

$$hf(k,n) = \begin{cases} n & \texttt{für } n \leq k \\ \sum_{i=1}^{k} hf(k, n-i) & \texttt{sonst} \end{cases}$$

Implementieren Sie die Ocaml-Funktion `val hyperfib : int -> int -> int`, die die Hyperfibonacci-Funktion **endrekursiv** berechnet.

Lösungsvorschlag 7.2

Die Lösung befindet sich in der Datei `ocaml/ha7_sol.ml`.

Aufgabe 7.3 (H) Coco [2 Punkte + 5 Bonuspunkte]

Gegeben sei folgende Funktion:

$$f(n) = \begin{cases} f(\frac{n}{2}) & \texttt{für } n \equiv 0 \ (\texttt{mod } 2) \\ f(3n+1) & \texttt{sonst} \end{cases}$$

Bearbeiten Sie folgende Teilaufgaben.

1. Implementieren Sie die Ocaml-Funktion `val coco : int -> int`, die für ein gegebenes n die Anzahl der rekursiven Aufrufe von $f(n)$ berechnet, die nötig sind, bis erstmals f mit dem Parameter 1 aufgerufen wird. Ihre Implementierung muss **endrekursiv** sein.

2. *(Bonusaufgabe, keine Lösung)* Zeigen oder widerlegen Sie[1]: Die Berechnung der in 1. definierten Funktion terminiert für jedes $n \in \mathbb{N}$.

Lösungsvorschlag 7.3

Die Lösung befindet sich in der Datei `ocaml/ha7_sol.ml`.

Aufgabe 7.4 (H) Endrekursive Bäumchen [5 Punkte]

In dieser Aufgabe soll die bereits bekannte Ocaml-Funktion

```
val to_sorted_list : bin_tree -> int list,
```

die einen binären Baum in eine aufsteigend sortierte Liste umwandelt, **endrekursiv** implementiert werden. Die Funktion soll ferner eine Laufzeitkomplexität in $\mathcal{O}(n)$ für einen Baum, der n Zahlen enthält, aufweisen.

Lösungsvorschlag 7.4

Die Lösung befindet sich in der Datei `ocaml/ha7_sol.ml`.

Aufgabe 7.5 (H) MiniML-Interpreter [10 Punkte + 5 Bonuspunkte]

In dieser Aufgabe soll ein Interpreter für eine fiktive Sprache *MiniML* in Ocaml programmiert werden. Dazu ist folgender Typ zur Repräsentation von MiniML-Programmen gegeben:

```
type variable = { name : string }
and immediate =
    Bool of bool
  | Integer of int
and expression =
    Variable of variable
  | Immediate of immediate
  | BinOperation of binop
  | UnOperation of unop
  | Binding of { is_rec : bool; var : variable; expr : expression; inner
  ↪  : expression }
  | Func of { param : variable; body : expression }
  | Closure of { param : variable; body : expression }
  | FuncApp of { f : expression; param_act : expression }
  | IfThenElse of { cond: expression; then_branch: expression;
  ↪  else_branch: expression}
and binop_op =
    Addition
  | Subtraction
  | Multiplication
```

[1] Aus Zeitgründen können nur für Tutoren direkt verständliche Lösungen bewertet werden; für den Beweis werden Integer als beliebig große Zahlen angenommen.

```
  | CompareEq
  | CompareLess
  | And
  | Or
and binop = binop_op * expression * expression
and unop_op =
    Negate
  | Not
and unop = unop_op * expression
and program = expression
[@@deriving show]
```

Ziel der Aufgabe ist es, eine Funktion

```
val evaluate : int list -> program -> immediate,
```

zu programmieren, die aus einer Liste von Parametern und einem MiniML-Programm das Ergebnis der Programmausführung berechnet. Gehen Sie vor wie folgt.

1. Betrachten Sie obigen Typen für MiniML-Programme. Welche Vor- bzw. Nachteile hat es, Ausdrücke unabhängig vom Typen als `expression` darzustellen? Argumentieren Sie insbesondere, welchen Einfluss dies auf den Zeitpunkt hat, zu dem Programmierfehler auffallen.
2. Implementieren Sie die `evaluate`-Funktion so weit, dass sie mit Variablen, Konstanten, sowie binären und unären Operationen umgehen kann. Parameter in der Parameterliste sollen im Programm in den Variablen `"@i"` zur Verfügung stehen, wobei i der Index des Parameters (beginnend bei 0) ist.
3. Fügen Sie nun `let`-Bindings, Funktionen, Funktionsaufrufe und bedingte Ausführung hinzu. Sie müssen *Shadowing* hier noch nicht korrekt behandeln; jede Variable darf im gesamten Programm nur einfach vorkommen.
4. Implementieren Sie Unterstützung für rekursive Funktionen.
5. *(Bonusaufgabe)* Fügen Sie schließlich Unterstützung für *Shadowing*, partielle Applikation, Funktionen höherer Ordnung und *Closures* hinzu.

   Hinweis: Closures kommen im Eingabeprogramme nicht vor. Sie können verwendet werden, um Closure-Objekte im bereits teilweise ausgewerteten Programm bzw. einem Zustand zu repräsentieren, werden also ggf. aus Funktionen und dem jeweiligen Kontext erzeugt.

Hinweis: Nicht alle MiniML-Programme sind wohlgeformt. Nutzen Sie die Funktion `val failwith : string -> 'a`, um bei einem Fehler Ihre Interpretation mit einer Nachricht abzubrechen, wenn eine Programmausführung keinen Sinn macht. Nutzen Sie diese Funktion auch, falls Sie Teilaufgaben abgeben oder testen möchten; so können Sie Fälle modellieren, die in der jeweiligen Teilaufgabe noch offen sind.

Lösungsvorschlag 7.5

1. Mittels `expression`-Objekten lassen sich typ-inkorrekte MiniML-Programm repräsentieren. Dies fällt erst auf, wenn das entsprechende Programm ausgeführt wird.

Eine bessere, wenn auch etwas komplizierte Repräsentation von Programmen weist dagegen schon zur Übersetzungszeit auf einige oder alle Typfehler hin; letzteres ist z.B. bei Ocaml der Fall.

Die übrigen Lösungen befindet sich in der Datei `ocaml/ha7_sol.ml`.