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

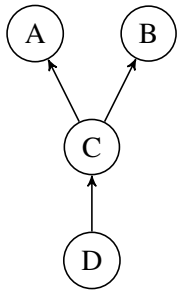
RPTU
Fachbereich Informatik
AG Programmiersprachen

Lösungshinweise/-vorschläge zum Übungsblatt 12: Grundlagen der Programmierung (WS 2023/24)

Aufgabe 1 Untertypen (Präsenzaufgabe)

Motivation: In dieser Aufgabe sollen Sie sich mit Untertypbeziehungen beschäftigen. Sie können sich an den Vorlesungsfolien 901 bis 969 sowie an den Kapiteln 8.1 und 8.2 im Skript orientieren.

Es gelten die Untertypbeziehungen $D \preccurlyeq C$, $C \preccurlyeq A$ und $C \preccurlyeq B$, die in der nebenstehenden Abbildung visualisiert sind.



In der folgenden Tabelle werden je zwei Typen in Relation gesetzt:

- $t_1 \prec t_2$ bedeutet, dass t_1 ein Untertyp von t_2 ist ($t_1 \preccurlyeq t_2$), nicht jedoch t_2 ein Untertyp von t_1.
- $t_1 \succ t_2$ bedeutet, dass t_2 ein Untertyp von t_1 ist ($t_2 \preccurlyeq t_1$), nicht jedoch t_1 ein Untertyp von t_2.
- $t_1 \parallel t_2$ bedeutet, dass t_1 und t_2 unvergleichbar sind, das heißt es gilt weder $t_1 \preccurlyeq t_2$ noch $t_2 \preccurlyeq t_1$.

Füllen Sie die Lücken in der Tabelle aus. In der ersten und dritten Spalte müssen Sie einen Typ eintragen, in der zweiten Spalte eine Relation ($\prec$ oder $\succ$ oder $\parallel$).

Zur Erinnerung: Die folgenden Deduktionsregeln gelten für die Relation »$\preccurlyeq$«:

$$\frac{t_1 \preccurlyeq t_2 \qquad t_2 \preccurlyeq t_3}{t_1 \preccurlyeq t_3} \qquad \frac{t_1 \preccurlyeq t_1' \qquad t_2 \preccurlyeq t_2'}{t_1 * t_2 \preccurlyeq t_1' * t_2'} \qquad \frac{t_1' \preccurlyeq t_1 \qquad t_2 \preccurlyeq t_2'}{t_1 \to t_2 \preccurlyeq t_1' \to t_2'}$$

t_1	Relation	t_2
D	$\boxed{\prec}$	A
$\boxed{A}$ oder $\boxed{B}$	$\succ$	C
$A * C$	$\prec$	$\boxed{A * A}$ oder $\boxed{A * B}$
$A * B$	$\boxed{\parallel}$	$B * A$
$C * D$	$\succ$	$\boxed{D * D}$
$A \rightarrow C$	$\boxed{\prec}$	$C \rightarrow C$
$B \rightarrow D$	$\boxed{\prec}$	$C \rightarrow C$
$C \rightarrow D$	$\succ$	$\boxed{A \rightarrow D}$ oder $\boxed{B \rightarrow D}$
$C \rightarrow A$	$\boxed{\succ}$	$B \rightarrow D$
$D \rightarrow A$	$\boxed{\parallel}$	$C \rightarrow B$

Aufgabe 2 Warteschlangen (Einreichaufgabe, 15 Punkte)

Schreiben Sie Ihre Lösungen in die Datei `Queues.fs` *aus der Vorlage* `Aufgabe-12-2.zip`.

Wir definieren eine Schnittstelle `IQueue`, um Warteschlangen zu beschreiben. Objekte, welche die Schnittstelle implementieren, müssen die Methoden `isEmpty`, `Add` und `Remove` definieren. Die Methode `isEmpty` gibt zurück, ob die Warteschlange leer ist. Mit Hilfe von `Add` soll ein Element in die Warteschlange eingefügt werden. Die Methode `Remove` entfernt das vorderste Element in einer nicht leeren Warteschlange und gibt es zurück. Falls die Warteschlange leer ist, wird `None` zurückgegeben.

```
type IQueue<'a> =
    interface
        abstract member isEmpty: Unit -> Bool
        abstract member Add: 'a -> Unit
        abstract member Remove: Unit -> Option<'a>
    end
```

a) Schreiben Sie eine Funktion `simpleQueue: Unit -> IQueue<'a>`, die ein `IQueue` Objekt erzeugt und zurückgibt. Verwenden Sie als Warteschlange nur eine Liste. Ein Element wird dann zur Warteschlange hinzugefügt, indem es ans Ende der Liste angehängt wird. Ein Element aus der Warteschlange zu entfernen wird dadurch realisiert, dass das erste Element aus der Liste entfernt wird.

```
let simpleQueue<'a> (): IQueue<'a> =
    let mutable xs = []
    {
        new IQueue<'a> with
            member self.isEmpty (): Bool =
                match xs with
                | [] -> true
                | _  -> false

            member self.Add (elem: 'a): Unit =
                xs <- xs @ [elem]

            member self.Remove (): Option<'a> =
                match xs with
                | [] -> None
                | y :: ys -> xs <- ys; Some y
    }
```

b) Implementieren Sie die Funktion `priorityQueue: Unit -> IQueue< QElem<'a> >`, die ein `IQueue` Objekt erzeugt und zurückgibt. Der Typ

```
type QElem<'a> =
    { priority: Nat
      value: 'a }
```

ist Ihnen von Blatt 5, Aufgabe 2, bekannt. Entsprechend dieser Aufgabe soll auch die Warteschlange intern funktionieren.

```
let priorityQueue<'a> (): IQueue<QElem<'a>> =
    let mutable xs = []
    {
        new IQueue<QElem<'a>> with
            member self.isEmpty (): Bool =
                match xs with
                | [] -> true
                | _  -> false

            member self.Add (elem: QElem<'a>): Unit =
                let rec helper (xs: List<QElem<'a>>)
                                    (elem: QElem<'a>): List<QElem<'a>> =
                    match xs with
                    | [] -> [elem]
                    | y::ys ->
                        if elem.priority <= y.priority then elem::xs
                        else y::(helper ys elem)
                xs <- helper xs elem

            member self.Remove (): Option<QElem<'a>> =
                match xs with
                | [] -> None
                | y :: ys -> xs <- ys; Some y
    }
```

c) Implementieren Sie die Funktion `advancedQueue: Unit -> IQueue<'a>`, die ebenfalls ein `IQueue` Objekt erzeugt und zurückgibt. Hier soll die Warteschlange allerdings durch zwei Listen `front` und `rear` abgebildet werden. Ein neues Element wird zur Warteschlange hinzugefügt, indem es einfach an den Anfang der `front` Liste eingefügt wird. Wir entfernen ein Element aus der Warteschlange, indem wir das erste Element der `rear` Liste entfernen und zurückgegeben. Wenn allerdings die `rear` Liste leer ist, müssen wir zuvor die Elemente der `front` Liste in umgekehrter Reihenfolge in `rear` schreiben und alle Einträge aus `front` entfernen. Sind sowohl `front` als auch `rear` leer, dann ist die Warteschlange insgesamt leer und es soll `None` zurückgegeben werden.

Der Vorteil hierbei ist, dass die Laufzeit der beiden Operationen besser ist als in der ersten Variante.

```
let advancedQueue<'a> (): IQueue<'a> =
    let mutable front = []
    let mutable rear  = []
    {
        new IQueue<'a> with
            member self.isEmpty (): Bool =
                match front, rear with
                | [], [] -> true
                | _ , _ -> false

            member self.Add (elem: 'a): Unit =
                front <- elem :: front

            member self.Remove (): Option<'a> =
                match rear with
                | [] ->
                    match front with
                    | [] -> None
                    | _  -> rear <- List.rev front
                            front <- []
                            self.Remove ()
                | b :: bs ->
                    rear <- bs
                    Some b
    }
```

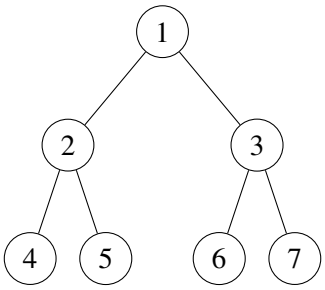
d) Implementieren Sie die Funktionen `enqueue: IQueue<'a> -> List<'a> -> Unit` und `dequeue: IQueue<'a> -> List<'a>`. Die Funktion `enqueue` fügt eine Liste von Elementen der Reihe nach in eine Warteschlange ein, während `dequeue` alle Elemente aus der Warteschlange entfernt und diese als Liste zurückgibt (das erste Element der Warteschlange befindet vorne in der Liste).

Beachten Sie, dass die Funktionen unabhängig von einer konkreten Implementierung der `IQueue` Schnittstelle sind. Wir können sowohl eine `simpleQueue` als auch eine `advancedQueue` verwenden, ohne `enqueue` oder `dequeue` anpassen zu müssen.

```
let rec enqueue (q: IQueue<'a>) (elems: List<'a>): Unit =
    List.iter q.Add elems

let rec dequeue (q: IQueue<'a>): List<'a> =
    match q.Remove () with
    | None -> []
    | Some e -> e :: dequeue q
```

e) Warteschlangen werden als Hilfsmittel in manchen Algorithmen benötigt. Ein Beispiel hierfür ist der Breitendurchlauf von Bäumen. Wir betrachten dazu den bekannten Typ für Binärbäume:




```
type Tree<'a> =
    | Empty
    | Node of Tree<'a> * 'a * Tree<'a>
```

Beim Breitendurchlauf werden die Elemente ebenenweise durchlaufen, in nebenstehendem Beispiel also 1, 2, 3, 4, 5, 6, 7.

Um einen Baum der Breite nach zu durchlaufen, wenden wir folgenden Algorithmus an:

1. Der Baum wird als Ganzes in die Warteschlange eingefügt.
2. Das erste Element wird aus der Warteschlange entfernt. Wenn wir `None` erhalten haben, ist die Warteschlange leer und wir sind fertig.
3. Wenn es sich bei dem entfernten Element um ein Blatt (ein Element `Empty`) handelt, gehe zurück zu Punkt 2.
4. Wenn es sich bei dem entfernten Element um einen Knoten (ein Element `Node`) handelt, füge den linken und rechten Teilbaum zur Warteschlange hinzu. Das im Knoten enthaltene Element ist das vorderste Element der Rückgabeliste. Um den Rest der Rückgabeliste zu berechnen, gehe zurück zu Punkt 2.

Schreiben Sie eine Funktion `bft: IQueue<Tree<'a>> -> List<'a>` (für *breadth-first traversal*), die als Argument einen Baum in einer Warteschlange erwartet (Punkt 1 müssen Sie also nicht implementieren) und diesen der Breite nach durchläuft, um die gefundenen Elemente in einer Liste zurückzugeben.

```
// Breadth-first traversal
let rec bft (q: IQueue<Tree<'a>>): List<'a>  =
    match q.Remove () with
    | None -> []
    | Some root ->
        match root with
        | Empty -> bft q
        | Node (left, x, right) ->
            q.Add left
            q.Add right
            x :: (bft q)
```

Aufgabe 3 Untertypen (Trainingsaufgabe)

Motivation: In dieser Aufgabe sollen Sie die Regeln der statischen Semantik von Schnittstellen und Untertypen einüben. Sie können sich an den Vorlesungsfolien 899 bis 967 sowie an den Kapiteln 8.1 und 8.2 im Skript orientieren.

```
type A =
    interface
        abstract member f: Unit -> Nat
    end

type B =
    interface
        inherit A
        abstract member g: Nat -> String
    end
```

```
type C =
    interface
        inherit A
        abstract member h: String -> Nat
    end

type D =
    interface
        inherit C
        abstract member i: Nat -> Unit
    end
```

Verwenden Sie die Schnittstellentypdefinitionen von oben, um den Typ der folgenden Ausdrücke mit einem vollständigen Beweisbaum anzugeben. Benutzen Sie die Regeln der **statischen Semantik** aus der Vorlesung.

a) `fun (s : B * D) -> ((snd s) :> C).h ((fst s).g 1N)`

Verwende $\Sigma := \{\texttt{s} \mapsto B * D\}$

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\Sigma \vdash \texttt{s} : B * D}}{\Sigma \vdash \texttt{snd s} : D} \quad \dfrac{}{D \leqslant C}}{\Sigma \vdash \texttt{(snd s) :> C} : C}}{\Sigma \vdash \texttt{((snd s) :> C).h} : String \to Nat} \quad \dfrac{\dfrac{\dfrac{\dfrac{}{\Sigma \vdash \texttt{s} : B * D}}{\Sigma \vdash \texttt{fst s} : B}}{\Sigma \vdash \texttt{(fst s).g} : Nat \to String} \quad \dfrac{}{\Sigma \vdash \texttt{1N : Nat}}}{\Sigma \vdash \texttt{(fst s).g 1N} : String}}{\Sigma \vdash \texttt{((snd s) :> C).h ((fst s).g 1N)} : Nat}}{\emptyset \vdash \texttt{fun (s : B * D) -> ((snd s) :> C).h ((fst s).g 1N)} : B * D \to Nat}$$

b) `fun (s: A -> D) -> (fun (b: B) -> (s b).f ())`

Verwende $\Sigma := \{\texttt{s} \mapsto A \to D, b \mapsto B\}$

$$\dfrac{\dfrac{\dfrac{}{\Sigma \vdash \texttt{s} : A \to D} \quad \dfrac{\dfrac{\dfrac{}{B \leqslant A} \quad \dfrac{\dfrac{}{D \leqslant C} \quad \dfrac{}{C \leqslant A}}{D \leqslant A}}{\Sigma \vdash A \to D \leqslant B \to A}}{\Sigma \vdash \texttt{s} : B \to A} \quad \dfrac{}{\Sigma \vdash \texttt{b} : B}}{\Sigma \vdash \texttt{s b} : A} \quad \dfrac{}{\Sigma \vdash \texttt{()} : Unit}}{\Sigma \vdash \texttt{(s b).f ()} : Nat}}{\{\texttt{s} \mapsto A \to D\} \vdash \texttt{fun (b: B) -> (s b).f ()} : B \to Nat}}{\emptyset \vdash \texttt{fun (s: A -> D) -> (fun (b: B) -> (s b).f ())} : (A \to D) \to (B \to Nat)}$$