Fakultät IV – Elektrotechnik und Informatik
MPGI 1 WS 2013/14
Pepper / Höger / Reicherdt / Zuber / Tutoren

Aufgabenblatt 7

letzte Aktualisierung: 27. 11, 18:05

Ausgabe: 27.11.2013
Abgabe: 05/06.12.2013

Thema: *Datentypen, Polymorphie*

Achtung: In diesem Aufgabenblatt erwarten wir von euch ein **Testmodul**, das Testfunktionen ähnlich der Hausaufgabe 3.2 des 5. Aufgabenblattes enthält. Es soll **alle** eure Module der Hausaufgabe importieren und deren Funktionen mit **geeigneten** Testwerten füttern und **nachvollziehbare** Ausgaben liefern.

1. Aufgabe: Datenstrukturen: Tutorien

1.1. (Tut) Produkttypen, Summentypen, induzierte Signatur

An einer Universität gibt es Studenten und verschiedene Arten von Lehrkräften. Um diese zu verwalten, sollen passende Datenstrukturen erstellt werden.

Deklariert und definiert mittels `DATA`

- einen Produkttypen `student`, der einen Studenten beschreibt (Matrikelnummer, Geburtsjahr, Fachsemester),
- und einen Summentypen `teacher`, der Lehrkräfte in den Ausprägungen Tutor (Stundenzahl u. -lohn), WiMi (Stundenzahl u. -lohn, Monatsbonus) und Professor (Monatslohn, Fachgebiet) beschreibt.

Notiert jeweils die induzierte Signatur und benennt ihre Komponenten.

1.2. (Tut) Rekursive Datenstrukturen

Das Prinzip der Rekursion ist nicht nur auf Funktionen, sondern auch auf Datenstrukturen anwendbar. Ein Beispiel für eine rekursive Datenstruktur ist die Liste oder Sequenz. Listen sind Folgen von gleichartigen Elementen.

In Opal werden Listen durch die Sorte `seq` beschrieben. `seq` ist ein rekursiver Datentyp, der sich aus der Möglichkeit der leeren Liste (Rekursionsabbruch) und dem Anhängen eines neuen Elementes an die Liste (Rekursionsvorschrift bzw. Konstruktionsvorschrift) zusammensetzt. Die Struktur `Seq` ist in Opal parametrisiert.

```
DATA seq == <>
            ::(ft:data, rt:seq)
```

Schreibt für die Datenstruktur `seq[nat]` eine Backtick-Funktion (‘), welche die Elemente der Liste der Reihenfolge nach ausgibt.

1.3. (Pflicht-Hausaufgabe) Produkttypen und Aufzählungstypen
Definiert eine Datenstruktur `tutorial`, die ein Fach mit einer Tutoriennummer, einer Lehrkraft und einer Liste von Studenten verknüpft. Benennt die Selektoren mit `subject`, `tutnr`, `teacher` und `students`. Verwendet für die verknüpften Personen die Datenstrukturen `teacher` und `seq[student]`, letzteres mit Hilfe der OPAL-eigenen Struktur `Seq`. Verwendet für das Fach einen Aufzählungstypen mit dem Namen `subject`, der mindestens die Fächer `MPGI1, TechGI1, LinA` beinhaltet.

Notiert die induzierten Signaturen (`tutorial, subject`) und benennt ihre Komponenten.

1.4. (Pflicht-Hausaufgabe) Average year of birth
Deklariert und definiert nun eine Funktion, die das durchschnittliche Geburtsjahr der Studenten in einem Tutorium (Input: `tutorial`) errechnet. Nutzt ggf. die BIBLIOTHECA OPALICA, um geeignete Listenfunktionen zu finden.
Erstellt dazu geeignete konstante Testwerte und gebt für eure Testwerte je die Tutoriennummer und das errechnete Durchschnittsalter aus.

Verwendet für die Aufgabe folgende Signaturen:

```
FUN averageYear : seq[tutorial] -> denotation
FUN averageYear : tutorial -> real
```

2. Aufgabe: Parametrisierte Strukturen – Priority Queue

Für viele Algorithmen und Abläufe müssen fortlaufend Daten gesammelt und nach Prioritäten gewichtet abgearbeitet werden. Wir wollen im folgenden eine **polymorphe** Struktur entwickeln, die diese Teilaufgabe für beliebige Anwendungsfälle übernehmen kann.

2.1. (Tut) Der abstrakte Datentyp PriorityQueue Betrachtet folgenden Ausschnitt aus der Signatur der Struktur `PriorityQueue`! Welche Rolle spielen `entry` und `<=`, welches Ergebnis sollte `e {dequeue(pop(init enqueue 10 enqueue 2 enqueue 23))}` liefern?

```
SIGNATURE PriorityQueue[entry, <=]
SORT entry
FUN <= : entry ** entry -> bool

SORT priorityQueue
-- creates an empty queue
FUN init : priorityQueue
-- adds an element
FUN enqueue : priorityQueue ** entry -> priorityQueue
-- gets the element with the highest priority (if it exists)
FUN dequeue : priorityQueue -> option[entry]
-- removes the element with the highest priority, returns the remaining queue
FUN pop : priorityQueue -> priorityQueue
```

Hinweis:
```
SIGNATURE Option [data]
          TYPE option  == nil
                       avail( cont: data )
```

2.2. (Tut) PriorityQueue implementieren Intern wollen wir Opals `Seq[entry]` verwenden, um eine PriorityQueue zu repräsentieren. Indem wir `content` und `prio` nicht in der Signatur erwähnen, verhindern wir, dass die Stack-Funktionalität von Seq von außen nutzbar ist, und dadurch das Sortierungskriterium verletzt werden kann.

```
DATA priorityQueue ==
    prio(content :seq[entry])
```

Implementiert in der Vorgabestruktur die Funktionen `enqueue`, `dequeue` und `pop`!

2.3. (Tut) PriorityQueue testen Leider lassen sich parametrisierte Strukturen nicht direkt in oasys ausführen. Legt darum eine Struktur `PriorityQueueTest` an, in der ihr PriorityQueue mit `nat` und dessen `<=` instanziiert. Testet, ob `enqueue`, `dequeue` und `pop` wie erwartet zusammenspielen!

2.4. (Tut) dropWhile für PriorityQueue Schreibt eine Funktion höherer Ordnung `dropWhile : (entry -> bool) -> priorityQueue -> priorityQueue`. Diese soll, solange eine bestimmte Bedingung für das jeweils wichtigste Element gilt, dieses aus der Warteschlange schmeißen und die nächsten betrachten. Erfüllt ein wichstigstes Element die Bedingung nicht, bleibt die Queue unverändert.

3. Aufgabe: Finite Set

Nun soll es darum gehen, unter Rückgriff auf die in Opal eingebauten Sequences eine Struktur für endliche Mengen zu implementieren. Die Struktur ist mit einem Typ und einer Gleichheitsfunktion (Äquivalenzrelation) über dem Typ zu parametrisieren. Innerhalb unserer Mengen darf es jedes Element nur einmal geben. **Eure Struktur soll mit den vorgegebenen Tests in `FiniteSetTest` kompatibel sein und die dort erwarteten Ergebnisse liefern.**

3.1. (Pflicht-Hausaufgabe) Struktur anlegen Legt die parametrisierte Struktur `FiniteSet` an! Definiert einen Datentyp, der auf Sequenzen aufbaut, sowie folgende Funktionen:

- `{}` erstellt eine leere Menge.
- `{}?` prüft für eine Menge, ob es sich um die leere Menge handelt.
- `contains` prüft, ob eine Menge ein Objekt enthält. (Für spätere Aufgaben bietet es sich an, hier die Menge als ersten Parameter zu nehmen und Currying anzuwenden.)
- `<<` fügt einer Menge ein Element hinzu, falls dieses noch nicht enthalten ist, oder lässt die Menge unverändert.
- `‘` wandelt eine Menge unter Verwendung einer zu den Elementen passenden Backtick-Funktion in in eine `denotation` um, z.B. `‘(‘:nat->denotation)({} << 1:nat << 2)` ⇝ `"{1,2}"`

Nachdem ihr diese Aufgabe abgeschlossen habt, sollte die vorgegebene Struktur `FiniteSetTest` ausführbar sein und sinnvolle Ergebnisse liefern.

Hinweis: Für `contains` kann die Funktion `exist?` aus der Struktur `Seq` verwendet werden! Auch sonst dürft und sollt ihr hier auf Funktionen aus `Seq` zurückgreifen, wenn es euch das Leben einfacher macht.

3.2. (Pflicht-Hausaufgabe) Test-Struktur vervollständigen Testet eure Funktionen mithilfe von `FiniteSetTest`! Erweitert die Struktur so, dass alle Funktionen getestet werden!