FAU, Informatik 2, AUD-Team
aud@i2.cs.fau.de

8. Übung

Abgabe bis 26.06.2017, 10:00 Uhr

Einzelaufgabe 8.1: ADT — 33 EP

Gegeben seien die folgenden Gerüste der ADTs Nat (Repräsentation natürlicher Zahlen $\mathbb{N}_0^+$) und $NatStack$ (Stapel mit Nat-Elementen):

---

adt *Nat*
sorts *Nat*
ops

NaN:		$\mapsto$ *Nat*	*// „Not a Nat“ $\rightsquigarrow$ vergleichbar mit* `Double.NaN`
zero:		$\mapsto$ *Nat*	*// entspricht der Zahl „0“*
succ:	*Nat*	$\mapsto$ *Nat*	*// Nachfolger „$(x+1)$“ der Nat-Zahl x*
eq:	*Nat* $\times$ *Nat*	$\mapsto$ *Nat*	*// $succ(zero)$, falls die Operanden gleich sind; $zero$ sonst*
add:	*Nat* $\times$ *Nat*	$\mapsto$ *Nat*	*// Addition im Nat-Raum*
sub:	*Nat* $\times$ *Nat*	$\mapsto$ *Nat*	*// Subtraktion im Nat-Raum (z.B. „$42-666=0$“)*
mul:	*Nat* $\times$ *Nat*	$\mapsto$ *Nat*	*// Multiplikation im Nat-Raum*
div:	*Nat* $\times$ *Nat*	$\mapsto$ *Nat*	*// ganzzahlige Division im Nat-Raum (immer abgerundet)*
mod:	*Nat* $\times$ *Nat*	$\mapsto$ *Nat*	*// Rest bei Nat-Division (modulo, in Java: %)*

axs

succ(NaN) = NaN	*// **alle** Operationen mit NaN ergeben stets **wieder** NaN*
...	*// weitere Axiome aus Platzgründen weggelassen*

end *Nat*

---

adt *NatStack*
sorts *NatStack, Nat*
ops

empty:		$\mapsto$ *NatStack*	*// erzeugt einen neuen leeren Stapel*
push:	*NatStack* $\times$ *Nat*	$\mapsto$ *NatStack*	*// legt ein Nat **oben auf** den Stapel*
peek:	*NatStack*	$\mapsto$ *Nat*	*// gibt bei leerem Stapel NaN und bei // nicht-leerem Stapel das **„oberste“** Nat zurück*
pop:	*NatStack*	$\mapsto$ *NatStack*	*// gibt bei leerem Stapel $empty$ und bei nicht-leerem // Stapel den Stapel ohne das **„oberste“** Nat zurück*
put:	*Nat* $\times$ *NatStack*	$\mapsto$ *NatStack*	*// fügt ein Nat ganz **unten** in den Stapel ein*
get:	*NatStack*	$\mapsto$ *Nat*	*// gibt bei leerem Stapel NaN und bei // nicht-leerem Stapel das **„unterste“** Nat zurück*
poll:	*NatStack*	$\mapsto$ *NatStack*	*// gibt bei leerem Stapel $empty$ und bei nicht-leerem // Stapel den Stapel ohne das **„unterste“** Nat zurück*
stackMul:	*NatStack*	$\mapsto$ *Nat*	*// multipliziert alle Nat-Zahlen im $NatStack$*
stackPairOp:	*NatStack*	$\mapsto$ *NatStack*	*// multipliziert gleiche bzw. addiert ungleiche // benachbarte Zahlen*
nat2bin:	*Nat*	$\rightarrow$ *NatStack*	*// Binärdarstellung des Nats ($empty$ falls NaN)*
bin2nat:	*NatStack*	$\rightarrow$ *Nat*	*// Umkehrfunktion von nat2bin*

axs

... *// Axiome aus Platzgründen weggelassen bzw. sind zu ergänzen*

end *NatStack*

WICHTIG: In dieser Aufgabe stehen Ihnen für die Axiome des ADTs $NatStack$ **keine** anderen Datentypen (also **kein** `int`, `String` usw.), Konstanten (also auch **nicht** $false$, $null$, 0 usw.) oder Operationen (also **kein** $<$, $+$, $==$, $\neq$ usw.) zur Verfügung – auch **nicht** auf der „rechten Seite" der Axiome in „falls/sonst"-Bedingungen! Sie dürfen ausschließlich die vorgegebenen oder zu ergänzenden **adt**s mit deren **ops** verwenden!

a) Ergänzen Sie den ADT $NatStack$ um die Axiome der Operationen put, get und $poll$ so, dass der $NatStack$ wie eine *Deque* von beiden Seiten bearbeitet werden kann. Führen Sie dazu die Operationen put / get / $poll$ rekursiv und **nur** auf $empty$ / $push$ / $peek$ / pop zurück.

b) Ergänzen Sie $NatStack$ nun um die Axiome der Operation $stackMul$ gemäß Kommentar. *Hinweis*: Aufgrund der Art wie put / get / $poll$ vorangehend definiert wurden, genügt hier die Spezifikation der Wirkung von $stackMul$ auf die beiden *Primärkonstruktoren* ...

c) Ergänzen Sie den ADT $NatStack$ um die Axiome der Operation $stackPairOp$ gemäß Kommentar bzw. folgendem Beispiel: Die Zahlen im Stack werden paarweise von „oben nach unten" betrachtet und genau dann durch ihr Produkt ersetzt, wenn sie gleich sind ($\rightsquigarrow$ Quadratzahl) – andernfalls durch ihre Summe. Alle Einschränkungen gelten weiterhin!

$$\textbf{Beispiel:} \left\{ \begin{array}{c} \triangleright\ 47_{(Nat)} \\ \triangleright\ 11_{(Nat)} \\ \blacktriangleright\ 11_{(Nat)} \\ \blacktriangleright\ 11_{(Nat)} \\ \triangleright\ 11_{(Nat)} \\ \triangleright\ 655_{(Nat)} \\ \lozenge\ 42_{(Nat)} \end{array} \right\} \overset{stackPairOp}{\longmapsto} \left\{ \begin{array}{c} \triangleright\ 58_{(Nat)} \\ \blacktriangleright\ 121_{(Nat)} \\ \triangleright\ 666_{(Nat)} \\ \lozenge\ 42_{(Nat)} \end{array} \right\}$$

Zur Vereinfachung wurden die Nat-Zahlen hier mit Ziffern ausgedrückt, d.h. z.B. $3_{(Nat)}$ steht für $succ(succ(succ(zero)))$

d) Geben Sie die Axiome der Operation $nat2bin$ an, die für eine Nat-Zahl ihre Binärdarstellung im *Zweierkomplement* als $NatStack$ wie im folgenden Beispiel erzeugt:

$$\textbf{Beispiel: } 42_{(10)} \overset{nat2bin}{\longmapsto} \left\{ \begin{array}{c} \underline{zero} \\ succ(zero) \\ zero \\ succ(zero) \\ zero \\ succ(zero) \\ zero \end{array} \right\} = 010101\underline{0}_{(2)}$$

e) Ergänzen Sie die Axiome für die Operation `bin2nat`, die aus einer Binärdarstellung im Stapel die zugehörige Nat-Zahl berechnet. Alle von $zero$ verschiedenen Nats werden dabei als 1-Bit interpretiert. Da es keine negativen Nat gibt, soll im Falle einer negativen Binärzahl NaN zurückgegeben werden. Das *niederwertigste* Bit ist wie bei $nat2bin$ das **„oberste"**.

f) Implementieren Sie eine Klasse `NatStackOp`, die *ausschließlich* die Operationen `stackMul`, `stackPairOp`, `nat2bin` und `bin2nat` sowie `put`, `get` und `poll` bereitstellt. Hier dürfen Sie ausnahmsweise die Java-Operatoren „==", „`&&`" und „`||`" sowie „`if`" und „`return`" verwenden, ansonsten aber *nichts* aus der Java-API.

Geben Sie Ihre Lösung als `ADT.pdf` sowie `NatStackOp.java` über EST ab.

Gruppenaufgabe 8.2: wp-Kalkül **16 GP**

Die Methode `cs(n)` soll Partialsummen einer Collatz-ähnlichen Folge für $n \geq 1$ berechnen:

```
long cs(long n) {
    long r = 0, i = 0;
    while (i < n) {
        i++;
        if (i % 2 == 0) {
            r += i;
        } else {
            r += 3 * i + 1;
        }
    }
    return r;
}
```

Sie dürfen im Folgenden vereinfachend annehmen, dass es keinen Überlauf geben kann. Anstelle der mathematischen Modulo-Funktion $(x \mathit{ mod } y)$ dürfen Sie in Ihrem Beweis vereinfachend die Java-Schreibweise `x % y` benutzen.

Wichtig: *Geben Sie Ihre Beweise möglichst ausführlich an, d.h. einzelne Zwischenschritte (Umformungen) müssen nachvollziehbar sein.*

Beweisen Sie *formal* mittels *wp-Kalkül* die totale Korrektheit von `cs(n)` hinsichtlich:

$$\forall n \geq 1 : \texttt{cs(n)} \equiv (n + n \mathit{ mod } 2) \cdot (n + 1)$$

Überlegen Sie zunächst, welches der folgenden Prädikate eine zum Beweisen der Korrektheit der Methode sinnvolle Schleifeninvariante für die Schleife in `cs(n)` darstellt:

- $r = ((i + 1) + (i + 1) \mathit{ mod } 2) \cdot (i + 2) \land 0 \leq i + 1 \leq n \land 1 \leq n$
- $r = (i + i \mathit{ mod } 2) \cdot (i + 1) \land 0 \leq i \leq n \land 1 \leq n$
- $r = (n + n \mathit{ mod } 2) \cdot (n + 1) \land 0 \leq i \leq n \land 1 \leq n$
- $cs(n) = (i + i \mathit{ mod } 2) \cdot (i + 1) \land 0 \leq i \leq n \land 1 \leq n$

a) Weisen Sie *formal* mittels *wp-Kalkül* nach, dass die von Ihnen gewählte Invariante unmittelbar vor dem ersten Betreten des Schleifenrumpfs in `cs(n)` gilt.

b) Weisen Sie *formal* mittels *wp-Kalkül* nach, dass die von Ihnen gewählte Invariante nach jedem Durchlaufen des Schleifenrumpfs in `cs(n)` gilt.

c) Weisen Sie *formal* mittels *wp-Kalkül* nach, dass nach dem letzten Schleifendurchlauf in `cs(n)` die Nachbedingung $r \equiv (n + n \mathit{ mod } 2) \cdot (n + 1)$ erfüllt ist.

d) Geben Sie eine geeignete Schleifenvariante V für die Schleife in `cs(n)` an und begründen Sie kurz, warum Ihr Term V tatsächlich eine Schleifenvariante ist.

Geben Sie Ihre Lösung als `CollatzSumWP.pdf` über EST ab.

FAU, Informatik 2, AUD-Team
aud@i2.cs.fau.de

Gruppenaufgabe 8.3: Induktionsbeweis **11 GP**

Gegeben sei folgendes rekursives Programmfragment in der Sprache Java:

```
long doss(int a, int z) { // Diffs Of Squares Sums
    if (a == 0 && z == 0) {
        return 0;
    } else if (a == 0) {
        return doss(0, z - 1) - z * z;
    } else if (z == 0) {
        return a * a + doss(a - 1, 0);
    } else {
        return a * a + doss(a - 1, z - 1) - z * z;
    }
}
```

a) Beweisen Sie *formal* mittels vollständiger Induktion, dass die Methode `doss(a,z)` für beliebige $a, z \geq 0$ die Differenz der Summe der ersten a bzw. z Quadrate berechnet:

$$\forall a, z \in \mathbb{N}_0 : \texttt{doss(a,z)} \equiv \frac{a(a+1)(2a+1) - z(z+1)(2z+1)}{6}$$

Sie dürfen dabei annehmen, dass es *keinen* Überlauf im Rückgabewert gibt.

b) Geben Sie eine geeignete Terminierungsfunktion an und skizzieren Sie den Nachweis der Terminierung kurz.

Geben Sie Ihre Lösung als `Induktion.pdf` über EST ab.

33 EP + 27 GP = 60 Punkte