UNIVERSITÄT ZU LÜBECK
INSTITUTE FOR SOFTWARE ENGINEERING
AND PROGRAMMING LANGUAGES

Software Engineering im Wintersemester 2021/2022
Prof. Dr. Martin Leucker, Malte Schmitz, Stefan Benox, Julian Schulz, Benedikt Stepanek, Friederike Weilbeer, Tom Wetterich

Übungszettel 5 (Lösungsvorschlag)

08.12.2021

Abgabe bis Donnerstag, 25. November um 23:59 Uhr online im Moodle.

Aufgabe 5.1: Analyse einer algebraischen Spezifikation

5 Punkte, mittel

Betrachten Sie folgende algebraische Spezifikation:

```
spec Bool =
sort
    bool = true | false
ops
    not: (bool) bool
vars
    x: bool
axioms
    false ≠ true
end

spec MNat = Bool then
sort
    mnat = zero | succ(mnat)
ops
    add: (mnat, mnat) mnat
    mult: (mnat, mnat) mnat
    iszero: (mnat) bool
vars
    m, n: mnat
axioms
    succ(zero) ≠ zero
    succ(succ(zero)) ≠ zero
    succ(succ(succ(zero))) = zero
```

```
    add(zero, n) = n
    add(succ(m), n) = succ(add(m, n))

    mult(zero, n) = zero
    mult(succ(m), n) = add(mult(m, n), n)

    m = zero ⇒ iszero(m) = true
    m ≠ zero ⇒ iszero(m) = false
end
```

1. In der Algebraischen Spezifikation für `Bool` fehlen die Axiome für die Funktion `not`. `not` entspricht der logischen Negation. Stellen Sie Axiome für `not` auf.

   ▼ **Lösungsvorschlag**

   ```
   not(true) = false
   not(false) = true
   ```

2. Geben Sie für folgende Aussagen an, ob sie aus den Axiomen folgen oder nicht und begründen Sie Ihre Antwort.

   1. `succ(succ(succ(succ(zero)))) = succ(zero)`

      ▼ **Lösungsvorschlag**

      Ja, die Aussage folgt direkt aus dem Axiom `succ(succ(succ(zero))) = zero`.

   2. `mult(succ(succ(zero)), zero) = zero`

      ▼ **Lösungsvorschlag**

      Ja, denn es gilt `mult(succ(succ(zero)), zero) = add(add(zero, zero), zero) = zero`.

   3. `iszero(add(succ(succ(zero)), succ(zero))) = false`

      ▼ **Lösungsvorschlag**

Nein, denn es gilt `add(succ(succ(zero)), succ(zero)) = succ(succ(succ(zero))) = zero` und `iszero(zero) = true`.

3. Erläutern Sie, welches mathematische Konstrukt durch die Spezifikation `MNat` beschrieben wird.

▼ Lösungsvorschlag

Die Spezifikation `MNat` beschreibt einen Restklassenkörper $\mathbb{F}_3$ modulo 3.

Die ersten drei Axiome beschreiben einen Modulo-Zähler, indem sie festlegen, dass $\mathbf{3} = \mathbf{0}$, aber $\mathbf{1} \neq \mathbf{0}$ und $\mathbf{2} \neq \mathbf{0}$.

4. Entwickeln Sie ein Modell $\mathcal{M}$ für `MNat` mit der Trägermenge $\mathbb{N}_0$ für `mnat`. Dabei sei $\mathcal{B}$ ein Modell für `Bool`.

▼ Lösungsvorschlag

Ein Modell für eine algebraische Spezifikation besteht aus einer Trägermenge für jede Sorte, eine Übersetzung der Konstanten und Konstruktoren auf Elemente der Trägermenge und eine Implementierung der Operationen durch mathematische Funktionen. Wir verwenden die Trägermenge $\mathbb{N}_0$ für die Sorte `mnat`.

Das Modell $\mathcal{M}$ besteht aus der Konstanten $\mathtt{zero}^{\mathcal{M}} \in \mathbb{N}_0$, dem Konstruktor $\mathtt{succ}^{\mathcal{M}} \colon \mathbb{N}_0 \to \mathbb{N}_0$ und folgenden Operationen:

$$\begin{aligned} \mathtt{add}^{\mathcal{M}} &\colon \mathbb{N}_0 \times \mathbb{N}_0 \to \mathbb{N}_0 \\ \mathtt{mult}^{\mathcal{M}} &\colon \mathbb{N}_0 \times \mathbb{N}_0 \to \mathbb{N}_0 \\ \mathtt{iszero}^{\mathcal{M}} &\colon \mathbb{N}_0 \to \mathbb{B} \end{aligned}$$

Die Konstante $\mathtt{zero}^{\mathcal{M}}$ definieren wir als die Zahl $\mathbf{0} \in \mathbb{N}_0$:

$$\mathtt{zero}^{\mathcal{M}} = 0$$

Die Konstruktorfunktion $\mathtt{succ}^{\mathcal{M}}$ sei gegeben durch

$$\mathtt{succ}^{\mathcal{M}}(n) = n + 1 \bmod 3$$

Schließlich seien die Operationen gegeben durch

$$\mathtt{add}^{\mathcal{M}}(n, m) = (n + m) \bmod 3$$
$$\mathtt{mult}^{\mathcal{M}}(n, m) = (n \cdot m) \bmod 3$$
$$\mathtt{iszero}^{\mathcal{M}}(n) = \begin{cases} \mathtt{true}^{\mathcal{B}} & \text{falls } n = 0 \\ \mathtt{false}^{\mathcal{B}} & \text{sonst} \end{cases}$$

5. Diskutieren Sie, ob für Ihr Modell das Erzeugungsprinzip gilt.

▼ **Lösungsvorschlag**

Nein. Es lassen sich nur die Zahlen 0, 1 und 2 aus $\mathbb{N}_0$ erzeugen. Entsprechend lassen sich nicht alle Elemente aus der Trägermenge erzeugen.

Aufgabe 5.2: Mengen als algebraische Datenstruktur

4 Punkte, mittel

Im Folgenden ist eine algebraische Datenstruktur zur Verwaltung von Mengen gegeben:

```
spec NatSet = Bool and Nat then

sorts
    natSet = emptySet | add(natSet, nat)

ops
    isEmpty: (natSet) bool,
    contains: (natSet, nat) bool,
    union: (natSet, natSet) natSet,
    intersect: (natSet, natSet) natSet

vars
    s, s': natSet,
    n, n': nat

axioms
    isEmpty(emptySet) = true
    isEmpty(add(s, n)) = false

    contains(emptySet, n) = false
    contains(add(s, n), n) = true
    n ≠ n' ⇒ contains(add(s, n), n') = contains(s, n')
```

```
        union(emptySet, s) = s
        union(add(s, n), s') = add(union(s,s'), n)

        intersect(emptySet, s') = emptySet
        intersect(s, emptySet) = emptySet
        intersect(add(s, n), add(s', n)) = add(intersect(s, s'), n)
        n ≠ n' ⇒ intersect(add(s, n), add(s', n')) =
                union(intersect(s, add(s', n')), intersect(add(s, n), s'))
end
```

1. Beschreiben Sie in eigenen Worten, was die Funktion `intersect` macht und wie die einzelnen Axiome das Verhalten von `intersect` festlegen.

▼ Lösungsvorschlag

`intersect` berechnet den Schnitt zweier Mengen. Die ersten beiden Axiome legen fest, dass der Schnitt mit der leeren Menge die leere Menge ist. Das dritte Axiom stellt sicher, dass ein Element genau dann im Schnitt enthalten ist, wenn es in beiden Mengen enthalten ist. Wenn ein Element in beiden Mengen enthalten ist, aber nicht jeweils als letztes eingefügt wurde, dann greift das dritte Axiom nicht. Dafür nutzen wir das vierte Axiom: Wir entfernen jeweils das als letztes hinzugefügte Element aus je einer Menge, berechnen den Schnitt erneut und vereinigen die Ergebnisse. Wir durchsuchen so für das zuletzt eingefügte Element in der ersten Menge die gesamte zweite Menge und umgekehrt.

2. Entwickeln Sie ein Modell $\mathcal{A}$, welches das Erzeugungsprinzip erfüllt. Geben Sie auch die Trägermenge an. Nehmen Sie dabei an, dass $\mathcal{B}$ ein geeignetes Modell für `Bool` und $\mathcal{N}$ für `Nat` ist.

▼ Lösungsvorschlag

Wir verwenden als Trägermenge die Menge aller endlichen Teilmengen der Menge der natürlichen Zahlen $\mathbf{natSet}^{\mathcal{A}} = \{m \subset \mathbb{N} \mid |m| < \infty\}$.

Für die Konstante $\mathbf{emptySet}^{\mathcal{A}} \in \mathbf{natSet}^{\mathcal{A}}$ wählen wir

$$\mathbf{emptySet}^{\mathcal{A}} = \emptyset$$

Die Konstruktorfunktion $\mathbf{add}^{\mathcal{A}} : \mathbf{natSet}^{\mathcal{A}} \times \mathbb{N} \to \mathbf{natSet}^{\mathcal{A}}$ sei gegeben durch

$$\mathtt{add}^{\mathcal{A}}(s, n) = s \cup \{n\}$$

Die Operationen

$$\begin{aligned}
\mathtt{isEmpty}^{\mathcal{A}}&: \mathtt{natSet}^{\mathcal{A}} \to \mathbb{B} \\
\mathtt{contains}^{\mathcal{A}}&: \mathtt{natSet}^{\mathcal{A}} \times \mathbb{N} \to \mathbb{B} \\
\mathtt{union}^{\mathcal{A}}&: \mathtt{natSet}^{\mathcal{A}} \times \mathtt{natSet}^{\mathcal{A}} \to \mathtt{natSet}^{\mathcal{A}} \\
\mathtt{intersect}^{\mathcal{A}}&: \mathtt{natSet}^{\mathcal{A}} \times \mathtt{natSet}^{\mathcal{A}} \to \mathtt{natSet}^{\mathcal{A}}
\end{aligned}$$

seien gegeben durch

$$\begin{aligned}
\mathtt{isEmpty}^{\mathcal{A}}(s) &= (s = \emptyset) \\
\mathtt{contains}^{\mathcal{A}}(s, n) &= n \in s \\
\mathtt{union}^{\mathcal{A}}(s, s') &= s \cup s' \\
\mathtt{intersect}^{\mathcal{A}}(s, s') &= s \cap s'
\end{aligned}$$

Ergänzende Erläuterungen

Die Einschränkung der Trägermenge $\mathtt{natSet}^{\mathcal{A}}$ auf endliche Teilmengen ist wichtig, denn sonst könnten wir das Erzeugungsprinzip nicht nachweisen, denn die Menge der erzeugbaren Elemente ist definiert als die Menge aller Element, die *in endlich vielen Schritten* durch iterierte Anwendung der Operationen aus den Konstanten erzeugt werden können.

Es sind auch andere Trägermengen möglich. So könnte man zum Beispiel auch ein Modell $\mathcal{B}$ definieren mit der Trägermenge $\mathtt{natSet}^{\mathcal{B}} = \mathbb{N}^*$, also allen endlichen Sequenzen natürlicher Zahlen.

Für Sequenzen nutzen wir folgende Notation: Eine Sequenz

$$w = \langle w_0, w_1, \ldots, w_{n-1} \rangle \in \mathbb{N}^*$$

der Länge $|w| = n$ besteht aus n Zahlen. Die leere Sequenz ist entsprechend $\langle\rangle \in \mathbb{N}^*$. Mit

$$w^{(1)} = \langle w_1, w_2, \ldots, w_{n-1} \rangle$$

sei die Sequenz w ohne das erste Element bezeichnet. Entsprechend gilt $|w^{(1)}| = |w| - 1$. Sei

$$v = \langle v_0, v_1, \ldots, v_{m-1} \rangle \in \mathbb{N}^*$$

eine weitere Sequenz der Länge $|v| = m$. Dann ist die Konkatenation

$$v \mathbin{\&} w = \langle v_0, v_1, \dots, v_{m-1}, w_0, w_1, \dots, w_{n-1} \rangle$$

eine weitere Sequenz $v \mathbin{\&} w \in \mathbb{N}^*$ der Länge $|v \mathbin{\&} w| = n + m$.

Die Konstanten, Konstruktoren und Operatoren könnte man dann wie folgt definieren:

$$\begin{aligned}
\mathtt{emptySet}^{\mathcal{B}} &= \langle\rangle \\
\mathtt{add}^{\mathcal{B}}(s, n) &= \langle n \rangle \mathbin{\&} s \\
\mathtt{isEmpty}^{\mathcal{B}}(s) &= s = \langle\rangle \\
\mathtt{contains}^{\mathcal{B}}(s, n) &= \exists\, 0 < i \leq |s| \colon s_i = n \\
\mathtt{union}^{\mathcal{B}}(s, s') &= \begin{cases} s' \text{ wenn } s = \langle\rangle \\ \langle s_0 \rangle \mathbin{\&} \mathtt{union}^{\mathcal{B}}(s^{(1)}, s') \text{ sonst} \end{cases} \\
\mathtt{intersect}^{\mathcal{B}}(s, s') &= \begin{cases} \langle\rangle \text{ wenn } s = \langle\rangle \text{ oder } s' = \langle\rangle \\ \langle s_0 \rangle \mathbin{\&} \mathtt{intersect}^{\mathcal{B}}(s^{(1)}, s'^{(1)}) \text{ wenn } s_0 = s'_0 \\ \mathtt{union}^{\mathcal{B}}(\mathtt{intersect}^{\mathcal{B}}(s^{(1)}, s'), \mathtt{intersect}^{\mathcal{B}}(s, s' \end{cases}
\end{aligned}$$

Sowohl $\mathcal{A}$ als auch $\mathcal{B}$ sind Modelle und beide erfüllen das Erzeugungsprinzip. Das Modell $\mathcal{B}$ ist allerdings nicht nur deutlich komplizierter, sondern es speichert auch zusätzliche Informationen, die für eine Menge irrelevant sind: Durch die Verwendung einer Sequenz speichern wir nicht nur, welche Elemente enthalten sind, sondern auch, in welcher Reihenfolge und wie oft diese eingefügt wurden. Alle diese zusätzlichen Informationen sind zur Erfüllung der Axiome nicht notwendig.

3. Weisen Sie für Ihr Modell das Erzeugungsprinzip nach.

▼ **Lösungsvorschlag**

Damit das Erzeugungsprinzip gilt muss sich jede beliebige endliche Menge natürlicher Zahlen durch die Konstruktoren der Datenstruktur `NatSet` erzeugt werden kann.

Sei $M = \{m_1, m_2, \dots, m_k\} \in T$ eine solche beliebige Menge aus der Trägermenge. Dann kann M erzeugt werden, in dem beginnend mit $\mathtt{emptySet}^{\mathcal{A}}$ der Konstruktor $\mathtt{add}^{\mathcal{A}}$ mit jedem Element der Menge aufgerufen wird:

$$M = \mathtt{add}^{\mathcal{A}}(\dots \mathtt{add}^{\mathcal{A}}(\mathtt{add}^{\mathcal{A}}(\mathtt{emptySet}^{\mathcal{A}}, m_1), m_2), \dots, m_k).$$

4. Zeigen Sie, dass das Axiom `union(add(s, n), s') = add(union(s,s'), n)` durch ihr Modell $\mathcal{A}$ erfüllt ist.

▼ **Lösungsvorschlag**

Wir wollen zeigen, dass

$$\mathtt{union}^{\mathcal{A}}(\mathtt{add}^{\mathcal{A}}(s,n),s') = \mathtt{add}^{\mathcal{A}}(\mathtt{union}^{\mathcal{A}}(s,s'),n)$$

für zwei beliebige Mengen $s, s' \in T$ und eine beliebige natürliche Zahl $n \in \mathbb{N}$. Wir setzen dazu ein:

$$\begin{aligned}
& \mathtt{union}^{\mathcal{A}}(\mathtt{add}^{\mathcal{A}}(s,n),s') \\
= \; & \mathtt{add}^{\mathcal{A}}(s,n) \cup s' \\
= \; & (s \cup \{n\}) \cup s' \\
= \; & (s \cup s') \cup \{n\} \\
= \; & \mathtt{union}^{\mathcal{A}}(s,s') \cup \{n\} \\
= \; & \mathtt{add}^{\mathcal{A}}(\mathtt{union}^{\mathcal{A}}(s,s'),n)
\end{aligned}$$

Aufgabe 5.3: Arrays als algebraische Datenstruktur

3 Punkte, schwer

Spezifizieren Sie dynamisch wachsende Arrays für natürliche Zahlen in einer algebraischen Datenstruktur `NatArray`. Sie können dabei auf die Datenstrukturen `Nat` und `Bool` zurückgreifen, die Ihnen aus der Vorlesung bekannt sind. Zu Beginn sind alle Stellen des Arrays undefiniert. Das Array soll über folgende Funktionalitäten verfügen:

- Mit der Funktion `set` kann an einer bestimmten Position eine neue Zahl gesetzt werden.
- `defined` soll feststellen ob der Wert an einer bestimmten Position definiert (also gesetzt) ist.

- `get` soll den aktuellen Wert an einer Position zurückgeben, wenn dieser definiert ist.
- `remove` soll den Eintrag an einer bestimmten Position als undefiniert setzen.
- `merge` soll zwei Arrays in einem vereinigen. Verfügen beide Arrays an der gleichen Stelle über einen Eintrag, soll der des ersten Arrays bevorzugt werden.
- `sum` soll die Summe über allen Einträgen des Arrays berechnen.

▼ **Lösungsvorschlag**

```
spec NatArray = Bool and Nat then
sorts
    natArray = empty | set(natArray, nat, nat)
ops
    defined: (natArray, nat) bool,
    get: (natArray, nat) nat,
    remove: (natArray, nat) natArray,
    merge: (natArray, natArray) natArray
    sum: (natArray) nat
vars
    a, a': natArray,
    v, n, n' ,d: nat
axioms
    defined(empty, n) = false
    defined(set(a, n, v), n) = true
    n ≠ n' ⇒ defined(set(a, n, v), n') = defined(a, n')

    get(set(a, n, v), n) = v
    n ≠ n' ⇒ get(set(a, n, v), n') = get(a, n')

    remove(empty, n) = empty
    remove(set(a, n, v), n) = remove(a, n)
    n ≠ n' ⇒ remove(set(a, n, v), n') = set(remove(a, n'), n, v)

    merge(empty, a) = a
    merge(set(a, n, v), a') = set(merge(a, a'), n, v)

    sum(empty) = zero
    sum(set(a, n, v)) = add(sum(remove(a, n)), v)
end
```