Karlsruher Institut für Technologie
Institut für Theoretische Informatik
Prof. Dr. B. Beckert
Thorsten Bormer

Formale Systeme, WS 2010/2011

Praxisaufgabe 2: Theorembeweiser KeY

Abgabe am 06.02.2011.

Für die vollständige Lösung dieser Praxisaufgabe – sie besteht aus den beiden Teilaufgaben 1 und 2 – erhalten Sie 1,5 Bonuspunkte für die Abschlussklausur (bitte beachten Sie die Erläuterung zu Bonuspunkten auf der Webseite zur Vorlesung).

A Informationen zum KeY-System

A.1 Allgemeines

Was ist KeY? Zusammen mit unseren Partnern an der Chalmers University of Technology in Göteborg wird an unserem Institut das KeY-System entwickelt. Es ist ein Softwareverifikationswerkzeug, mit dem die Übereinstimmung von Java Card-Software und ihrer Spezifikation formal bewiesen werden kann.

Die Logik, auf der das KeY-System basiert, ist eine dynamische Prädikatenlogik. In dieser dynamischen Logik ist die in der Vorlesung eingeführte Prädikatenlogik vollständig enthalten. Deshalb können wir KeY auch benutzen, um rein prädikatenlogische Probleme zu formulieren und zu beweisen.

Literatur zu KeY Auf der Internetseite der Vorlesung steht eine Einführung zu Beweisen von prädikatenlogischen Formeln mit KeY (`KeYIntro.pdf`). Bitte arbeiten Sie diese Einführung durch.

Für weitergehende Fragen bietet sich die Lektüre des KeY-Buches [BHS07] an und darin vor allem Kapitel 10, das eine tiefere Einführung in KeY bietet. Kapitel 2 des Buches erklärt fundiert die prädikatenlogischen Grundlagen, auf denen KeY basiert.

Installation Das KeY-System besitzt eine graphische Benutzeroberfläche und ist komplett in Java geschrieben, so dass es auf jeder Plattform, für die eine virtuelle Maschine für Java zur Verfügung steht, lauffähig ist.

Version 1.6 ist die zuletzt veröffentlichte Version und kann über die KeY-Homepage[1] bezogen werden. Wenn Sie die Software „Java Web Start“ installiert haben (auf fast allen modernen Systemen mit Java der Fall), können Sie KeY auch direkt aus dem Internetbrowser heraus starten, ohne etwas installieren zu müssen.

A.2 Hinweis zur Konfiguration von KeY

Die automatische Beweisbarkeit hängt jeweils von der Problemformulierung sowie von den verwendeten Beweisereinstellungen ab.

- Für Teilaufgabe 1 empfiehlt es sich, im Bereich unten links in KeY auf der Registerkarte „Proof Search Strategy“ die Einstellung **FOL** zu wählen.
- Für Teilaufgabe 2 wählen Sie am besten **Java DL** und belassen alle weiteren Schalter in ihrer Standardstellung.

[1] `http://www.key-project.org/download/key.html`

A.3 KeY-Problemdateien zu den Teilaufgaben

- Für Teilaufgabe 1 erstellen Sie bitte selbst eine KeY-Problemdatei.
- Für Teilaufgabe 2 finden Sie auf der Webseite der Vorlesung die Datei `removeDuplicates.key` zum Herunterladen vor. Diese müssen Sie vervollständigen (wie unten im Abschnitt zu Teilaufgabe 2 beschrieben).

A.4 Abgabe der Lösungen

Bitte speichern Sie die beiden Beweise in KeY als „.key.proof“-Dateien ab. Auf der Webseite der Vorlesung steht Ihnen ein Formular zur Verfügung, um die beiden Lösungen einzureichen.

Für unvollständige Beweise werden Punkte auch anteilig vergeben.

B Teilaufgabe 1: Der Fall „Tante Agathe“

Folgendes logische Puzzle ist Teil einer großen Benchmarkbibliothek für Beweiser. Die natürlichsprachige Formulierung lautet:

Tante Agathe wurde in ihrem Haus tot aufgefunden. Nach bisherigen Ermittlungen gilt Folgendes als sicher:

1. Im Haus lebten nur Agathe, ihr Butler und Onkel Charles.
2. Agathe wurde von einem Hausbewohner getötet.
3. Wer jemanden tötet, hasst sein Opfer.
4. Charles hasst niemanden, den Agathe hasste.
5. Der Täter ist niemals reicher als das Opfer.
6. Der Butler hasst alle, die nicht reicher als Agathe sind, sowie alle, die Agathe hasste.
7. Kein Bewohner des Hauses hasst(e) alle Hausbewohner.
8. Agathe hasste alle außer dem Butler.
9. Agathe war nicht der Butler.

Zur Formalisierung dieser Aussagen wählen wir die Signatur: $\Sigma_{Agathe} = (F, P, \alpha)$ mit

- $P = \{\text{kills}, \text{hates}, \text{richer}\}$
- $F = \{a, b, c\}$
- $\alpha(a) = \alpha(b) = \alpha(c) = 0, \qquad \alpha(\text{kills}) = \alpha(\text{hates}) = \alpha(\text{richer}) = 2$

Über diese Signatur lassen sich die gegebenen Indizien folgendermaßen formalisieren:

$$
\begin{array}{ll}
 & \forall x(x \doteq a \vee x \doteq b \vee x \doteq c) \\
\wedge & \exists x(kills(x,a)) \\
\wedge & \forall x\ \forall y\ (kills(x,y) \rightarrow hates(x,y)) \\
\wedge & \forall x\ (hates(a,x) \rightarrow \neg hates(c,x)) \\
\wedge & \forall x\ \forall y\ (kills(x,y) \rightarrow \neg richer(x,y)) \\
\wedge & \forall x\ ((\neg richer(x,a) \vee hates(a,x)) \rightarrow hates(b,x)) \\
\wedge & \forall x\ \exists y\ \neg hates(x,y) \\
\wedge & \forall x\ (\neg x \doteq b \rightarrow hates(a,x)) \\
\wedge & \neg a \doteq b
\end{array}
$$

- Formalisieren Sie nun auch noch die Aussage „Tante Agathe hat sich selbst umgebracht.“,
- schreiben Sie eine Problembeschreibungsdatei für den Beweiser KeY und
- beweisen Sie mit KeY, dass aus den Indizien folgt, dass Tante Agathe sich selbst umgebracht haben muss.

C Teilaufgabe 2: Abstrakte Datentypen

Aufgabe. In dieser Aufgabe betrachten wir den termerzeugten Datentyp „Liste von Zahlen“. Ziel ist es, eine Funktion zu definieren, die Duplikate aus einer (aufsteigend sortierten[2]) Liste entfernt, und formal ihre Korrektheit zu beweisen. Dies ist eine typische Fragestellung der *Programmverifikation*. Die Definition der Operation zum Entfernen der Duplikate in der Logik entspricht dabei einer Implementierung in einer funktionalen Programmiersprache.

Signatur. Wir betrachten zwei Sorten von Termen bzw. Elementen: ganze Zahlen (Sorte `int`) und Listen von ganzen Zahlen (Sorte `lst`). In der Signatur gibt es u.a.:

<u>die Konstruktoren</u>			
`nil`	:	$\to$ `lst`	
`myCons`	:	`int` $\times$ `lst` $\to$ `lst`	
<u>der Mutator</u>			
`remDup`	:	`lst` $\to$ `lst`	(entfernt die Duplikate aus der Liste)
<u>den Inspektor</u>			
`myCount`	:	`int` $\times$ `lst` $\to$ `int`	(wie oft kommt die Zahl vor in der Liste?)
<u>die Prädikate</u>			
`sorted`	:	`lst` $\to$ `bool`	(ist die Liste sortiert?)
`lowerBound`	:	`int` $\times$ `lst` $\to$ `bool`	(es gibt keine kleinere Zahl in der Liste)
`headEq`	:	`int` $\times$ `lst` $\to$ `bool`	(die Zahl entspricht dem Kopf der Liste)

Wir beschränken uns nur auf „kanonische“ Interpretationen, in denen die Sorte `int` der Menge $\mathbb{Z}$ der ganzen Zahlen und `lst` der Menge aller (endlichen) Listen von ganzen Zahlen entspricht. Die Menge der Listen wird von den beiden Funktionen `nil` und `myCons` erzeugt. Die Signatur beinhaltet außerdem die üblichen Funktionen und Prädikate auf `int`, die ebenfalls kanonisch interpretiert werden. Die Variablen n und m seien im Folgenden stets vom Typ `int` und l vom Typ `lst`.

Induktionsregel. Da wir nur bestimmte Interpretationen betrachten, können wir den in der Vorlesung vorgestellten Sequenzenkalkül um weitere Regelschemata erweitern. Um Aussagen über Listen beweisen zu können, nehmen wir das Regelschema für die strukturelle Induktion über Listen hinzu:

$$\frac{\begin{array}{lcl}\Gamma & \Rightarrow & \{l/\texttt{nil}\}\varphi,\ \Delta \\ \Gamma & \Rightarrow & \forall n \forall l(\varphi \to \{l/\texttt{myCons}(n,l)\}\varphi),\ \Delta\end{array}}{\begin{array}{lcl}\Gamma & \Rightarrow & \forall l\ \varphi,\ \Delta\end{array}}\ \text{(listInduction)}$$

Diese Regel ist korrekt, da die Sorte `lst` von den beiden Konstruktoren `nil` und `myCons` termerzeugt wird, also jedes Element der Sorte als Term über diesen Funktionen dargestellt werden kann.

[2]Diese Eigenschaft erlaubt eine effiziente Implementation der Duplikatssuche. Dies muss auch in Ihrer Definition der entsprechenden Operation (s.u.) ausgenutzt werden.

Datentypaxiome. Die Funktionen `remDup`, `myCount` und die Prädikate `sorted`, `lowerBound`, `headEq` sind durch folgende Axiome (partiell) festgelegt:

$$
\begin{array}{rlcll}
\forall n & \texttt{lowerBound}(n, \texttt{nil}) & & & \text{(lowerBoundNil)} \\
\forall n \forall m \forall l & \big(\texttt{lowerBound}(n, \texttt{myCons}(m, l)) & \leftrightarrow & n \leqslant m \wedge \texttt{lowerBound}(n, l)\big) & \text{(lowerBoundCons)} \\
 & \texttt{sorted}(\texttt{nil}) & & & \text{(sortedNil)} \\
\forall n \forall l & \big(\texttt{sorted}(\texttt{myCons}(n, l)) & \leftrightarrow & \texttt{lowerBound}(n, l) \wedge \texttt{sorted}(l)\big) & \text{(sortedCons)} \\
\forall n & \texttt{myCount}(n, \texttt{nil}) & \doteq & 0 & \text{(countNil)} \\
\forall n \forall m \forall l & \texttt{myCount}(n, \texttt{myCons}(m, l)) & \doteq & \texttt{myCount}(n, l) + & \\
 & & & (\texttt{if } n \doteq m \texttt{ then 1 else 0})^3 & \text{(countCons)} \\
\forall n & \neg\texttt{headEq}(n, \texttt{nil}) & & & \text{(headEqNil)} \\
\forall n \forall m \forall l & \big(\texttt{headEq}(n, \texttt{myCons}(m, l))) & \leftrightarrow & (n = m)\big) & \text{(headEqCons)} \\
 & \texttt{remDup}(\texttt{nil}) & \doteq & \texttt{nil} & \text{(remDupNil)} \\
\forall n \forall l & \texttt{remDup}(\texttt{myCons}(n, l))) & \doteq & \ldots \text{ (benutzen Sie } \texttt{headEq}) & \text{(remDupCons)}
\end{array}
$$

C.1 Vervollständigung der Formalisierung des Datentyps

Auf der Seite zur Vorlesung finden Sie eine KeY-Datei, in der die obigen Axiome als Sequenzenkalkülregeln formalisiert sind (die z.T. automatisch angewendet werden). Vervollständigen Sie die mit „. . . “ markierte Stelle.

C.2 Die zu beweisende Aussage

Beweisen Sie mit KeY, dass die Funktion `remDup` aus einer aufsteigend sortierten Liste von ganzen Zahlen die Duplikate korrekt entfernt:

$$\forall l\,(\texttt{sorted}(l) \rightarrow \forall n\,(\texttt{myCount}(n, l) \geq 1 \rightarrow \texttt{myCount}(n, \texttt{remDup}(l)) = 1))$$

Diese Aussage sichert nur zu, dass bereits in der Ausgangsliste existierende Elemente korrekt behandelt werden. Über Elemente, die nicht in der ursprünglichen Liste vorhanden waren, wird hier zur Vereinfachung der Aufgabe keine Aussage gemacht.

C.3 Hilfreiche Lemmata

Mit einer einzigen Induktion kann die Aussage leider nicht bewiesen werden. Sie werden im Laufe des Beweises an Punkte kommen, an denen Lemmata benötigt werden, die selbst wieder durch Induktion zu beweisen sind. Benutzen Sie die (eingebaute) Regel `cut`, um ein Lemma einzufügen:

$$\frac{\overbrace{\Gamma, \varphi \;\Rightarrow\; \Delta}^{(A)} \qquad \overbrace{\Gamma \;\Rightarrow\; \varphi, \Delta}^{(B)}}{\Gamma \;\Rightarrow\; \Delta}\ (\mathsf{cut})$$

`cut` teilt den Beweis in zwei Äste auf:

[3] $\texttt{if } \phi \texttt{ then } t_1 \texttt{ else } t_2$ ist ein sog. *bedingter Term* (engl. conditional term).

$$\mathrm{val}_{D,I,\beta}(\texttt{if } \phi \texttt{ then } t_1 \texttt{ else } t_2) := \begin{cases} \mathrm{val}_{D,I,\beta}(t_1) \text{ wenn } \mathrm{val}_{D,I,\beta}(\phi) = W \\ \mathrm{val}_{D,I,\beta}(t_2) \text{ wenn } \mathrm{val}_{D,I,\beta}(\phi) = F \end{cases}$$

- Ast A: Hier steht die eingefügte Formel φ *links* des Sequenzenpfeiles. Damit steht sie auf diesem Ast als weitere Voraussetzung zur Verfügung und kann benutzt werden, um das ursprüngliche Ziel zu schließen (Verwenden des Lemmas).
- Ast B: Hier steht φ *rechts* des Sequenzenpfeiles. Auf diesem Ast ist φ also das Beweisziel und muss gezeigt werden (Beweis des Lemmas).

Im folgenden sind einige hilfreiche Lemmata aufgelistet:

Lemma 1 $\forall l \forall n \, \texttt{headEq}(n, l) \rightarrow \texttt{myCount}(n, l) \geq 1$

Lemma 2 macht eine Aussage über den Wertebereich von `myCount`.

Lemma 3 Eine Zahl, die eine untere Schranke der Werte einer sortierten Liste ist und nicht dem Kopf der Liste entspricht, ist selbst kein Element der Liste.

Lemma 4 Eine untere Schranke der Werte einer Liste ist kleiner oder gleich dem Kopf der Liste.

Literatur

[BHS07] Bernhard Beckert, Reiner Hähnle, and Peter H. Schmitt, editors. *Verification of Object-Oriented Software: The KeY Approach.* LNCS 4334. Springer-Verlag, 2007.