Formale Grundlagen der Informatik 3

Prof. Dr. Christoph Walther / Visar Januzaj, Nathan Wasser
Technische Universität Darmstadt — Wintersemester 2011/12

Praktische Übung 2

Diese Übung wird in der **KW 49 (05.12.–09.12.2011)** testiert. Reichen Sie dazu Ihre mit ✓`eriFun` 3.4 erstellte `.vf`-Lösungsdatei bis spätestens **Sonntag, 04.12.2011 23:59 Uhr** online im Kursportal ein. Beachten Sie bitte bei der Abgabe die dort angegebenen Hinweise. Später eintreffende Lösungen können nicht akzeptiert werden.

Es gelten die gleichen Testattermine bzw. Praktikumsgruppen G1-G90 wie in der letzten Testatwoche. Falls Sie und Ihre Gruppenmitglieder einen anderen Testattermin haben wollen, dann melden Sie sich in eine der leeren Gruppen um. Wählen Sie dabei bitte **nur** eine **leere** Gruppe aus, oder sprechen Sie die Mitglieder einer besetzen Gruppe an, ob sie mit Ihnen tauschen wollen. Die Ummeldung ist bis **Mittwoch, 30.11.2011 12:00 Uhr** freigeschaltet. Danach ist keine Um-/Anmeldung mehr möglich.

Bei Fragen und Problemen können Sie neben den Poolbetreuungszeiten der Tutoren auch das Forum zur Veranstaltung nutzen.

Der Fachbereich Informatik misst der Einhaltung der Grundregeln der wissenschaftlichen Ethik großen Wert bei. Zu diesen gehört auch die strikte Verfolgung von Plagiarismus. Mit der Abgabe einer Lösung bestätigen Sie, dass Sie der alleinige Autor des gesamten Materials sind. Bei Unklarheiten zu diesem Thema finden Sie weiterführende Informationen unter
`http://www.informatik.tu-darmstadt.de/Plagiarism`
oder sprechen Sie Ihren Betreuer an.

Vorbemerkung

Lesen Sie sich zunächst die Aufgaben durch. Bevor Sie mit der Lösung der Aufgaben beginnen, lesen Sie die Tipps, um systematisch zu einer Lösung zu gelangen.

Praktische Übung 2.1 (Addition und Multiplikation von Bytes)

In dieser Aufgabe geht es um die Verifikation der Korrektheit einer arithmetisch-logischen Einheit (ALU). Wir beschränken uns hierbei auf den Korrektheitsnachweis der Addition und Multiplikation von Bytes.

Sie finden in der Datei `Praxis3.vf` bereits die benötigten Datenstrukturen `bit` zur Repräsentation von Bits und `byte` zur Repräsentation von Bytes. Beachten Sie, dass wir bei der Modellierung der ALU von einer festen Bitbreite abstrahieren, indem wir die Algorithmen für Bytes beliebiger Länge formulieren. Bei einem nicht-leeren Byte x haben wir direkten Zugriff auf das „least significant bit“ $lsb(x)$. Mit $rshift(x)$ erhält man das Byte, das durch Wegstreichen des LSB aus x entsteht.

byte-inc ist eine Prozedur zum Inkrementieren eines Bytes um 1. Mit *byte2nat*[1] wird ein Byte in eine natürliche Zahl umgewandelt. Die Addition und Multiplikation auf Bytes wird durch die (noch unvollständigen) Prozeduren *byte-add* und *byte-mult* implementiert.

(a) Markieren Sie die Prozedur *byte-add* und wählen Sie im Kontextmenü *Change Procedure*. Vervollständigen Sie die Implementierung an den durch ⋆ gekennzeichneten Stellen.

[1] „byte to nat“, also von Bytes zu natürlichen Zahlen

Zur Funktionsweise: Wie bei der „schriftlichen Addition“ werden zunächst die least significant bits und dann (ggf. mit Übertrag) die höherwertigen Bits addiert. Im Unterschied zu der üblichen Hardware-Implementierung wird der Übertrag nicht sofort berücksichtigt — es gibt keinen entsprechenden formalen Parameter in *byte-add* —, sondern erst im Nachhinein.

(b) Beweisen Sie die Korrektheit der Byte-Addition, d. h. das Lemma „`byte-add is correct`“.

(c) Markieren Sie die Prozedur *byte-mult* und wählen Sie im Kontextmenü *Change Procedure*. Vervollständigen Sie die Implementierung an den durch $\star$ gekennzeichneten Stellen.

Zur Funktionsweise: Wie bei der „schriftlichen Multiplikation“ wird zunächst das least significant bit von x mit y multipliziert und dazu das nach links geschobene Produkt der höherwertigen Bits von x mit y addiert (mittels *byte-add*). Bei der Multiplikation von $x := 101$ mit $y := 1011$ erhält man beispielsweise folgendes Schema:

$$\begin{array}{rr} & 1011 \\ + & 0 \\ + & 101100 \\ \hline = & 110111 \end{array}$$

(d) Beweisen Sie die Korrektheit der Byte-Multiplikation, d. h. das Lemma „`byte-mult is correct`“.

Praktische Übung 2.2 (Bytes und natürliche Zahlen)

Analog zu der Prozedur *byte2nat* gibt es in der Datei `Praxis3.vf` auch eine Prozedur *nat2byte*, die eine natürliche Zahl in die Byte-Darstellung umwandelt.

(a) Beweisen Sie, dass die Prozedur *nat2byte* eine injektive Funktion berechnet.

(b) Beweisen Sie, dass $\mathit{byte2nat}(\mathit{nat2byte}(x)) = x$ für alle natürlichen Zahlen x gilt.

(c) Überlegen Sie sich, warum die Prozedur *byte2nat* keine injektive Funktion berechnet und damit die Gleichung $\mathit{nat2byte}(\mathit{byte2nat}(x)) = x$ nicht für alle Bytes x gilt.

(d) Markieren Sie die Prozedur *wfbyte*[2] und wählen Sie im Kontextmenü *Change Procedure*. Implementieren Sie die Prozedur so, dass die von `byte2nat` berechnete Funktion auf der Teilmenge aller Bytes x, für die $\mathit{wfbyte}(x)$ gilt, injektiv ist. Zeigen Sie dies, indem Sie das Lemma „`byte2nat is injective (wfbyte)`“ beweisen. Das Hilfslemma „`dbl(x)` $\neq$ `succ(dbl(y))`“ ist hierbei sehr hilfreich und sollte daher zuerst bewiesen werden.

(e) Zeigen Sie, dass $\mathit{wfbyte}(\mathit{nat2byte}(x))$ für alle natürlichen Zahlen x gilt.

(f) Weisen Sie nun nach, dass die Gleichung $\mathit{nat2byte}(\mathit{byte2nat}(x)) = x$ für alle Bytes x mit $\mathit{wfbyte}(x)$ gilt.

[2] Die Abkürzung steht für „well-formed byte“.

Allgemeine Tipps

- **Analysieren Sie fehlgeschlagene Beweisversuche, um auf die benötigten Hilfslemmata zu kommen.** Siehe dazu den nachfolgenden Hinweis „Systematisches Erfinden von Lemmata“.
- Es sind keine zusätzlichen Hilfsprozeduren für die Beweise nötig.
- Verwirren Sie sich nicht durch „auf Vorrat“ erfundene Lemmata in der Hoffnung, irgendwann durch Zufall ein nützliches Lemma zu erraten.
- Wenn Sie ein neues Hilfslemma formulieren, prüfen Sie, ob seine Aussage tatsächlich „neu“ ist. Mit einer Umformulierung eines anderen Lemmas drehen Sie sich nur im Kreis und kommen nicht vorwärts. Prüfen Sie vor einem Beweisversuch außerdem mittels des Interpreters oder des Disprovers, ob die Aussage plausibel ist, d. h. dass es kein „offensichtliches“ Gegenbeispiel gibt.

  Bedenken Sie, dass eine Aussage, deren Wahrheit für Sie offensichtlich ist, nicht notwendigerweise von ✓`eriFun` erkannt wird. Auf den Einführungsfolien wurde dies mit den Lemmata „$n+0 = n$“ und „$k <> \emptyset = k$“ demonstriert. Solche Aussagen sind für uns „offensichtlich wahr“. Das System kann aber diese Wahrheiten nur dann in Beweisen verwenden, wenn diese zuvor als ✓`eriFun`-Lemmata formuliert und auch verifiziert wurden. Anders gesagt, es ist allein Ihre Aufgabe, nützliche Hilfssätze zu erfinden (selbst wenn deren Aussagen für Sie trivial sind). Sind alle erforderlichen Lemmata vorhanden, so ist es dann die Aufgabe des Systems, die Beweise zu berechnen.
- Wenn Sie während des Beweisversuchs für ein Lemma A ein Lemma B formuliert und verifiziert haben, so können Sie am besten testen, ob Lemma B tatsächlich weiterhilft, indem Sie den Beweisbaum von Lemma A an der Wurzel abschneiden und dann *Verify* für Lemma A erneut aufrufen. Die Verwendung von Lemma B im neu berechneten Beweis von Lemma A ist ein gutes Zeichen (aber keine Garantie) für die Nützlichkeit von Lemma B. Ist jedoch zusätzlich das Beweisziel einfacher als zuvor, so ist Lemma B bestimmt nützlich. Wird Lemma B dagegen nicht verwendet, so ist es entweder unbrauchbar oder es fehlen noch weitere Hilfslemmata.
- Lassen Sie sich nicht durch fehlgeschlagene Beweisversuche entmutigen! Um nicht die Orientierung zu verlieren, sollten Sie konsequent darauf achten, aussagekräftige Namen für Ihre Lemmata zu wählen. Sortieren Sie Ihre Lemmata auch geeignet ein: Lemma B sollte *nach* Lemma A einsortiert werden, wenn A im Beweis von B verwendet wird.[3] Besteht zwischen zwei Lemmata keine Abhängigkeit, ist eine chronologische Einsortierung empfehlenswert: Das zuerst bewiesene Lemma kommt zuerst. Dies erleichtert Ihnen auch die Erläuterung Ihrer Lösung beim Testatgespräch.

[3] Das bedeutet nicht, dass B unbedingt *direkt* nach A einzusortieren ist.

Spezial-Tipp: Systematisches Erfinden von Lemmata

Beim Beweisen von Aussagen kommt es häufig vor, dass man auf Hilfslemmata zurückgreifen muss. Diese Hilfslemmata sind in der Regel zu Beginn noch nicht bekannt (bzw. noch nicht formuliert), sondern entstehen erst im Verlauf des Beweises. Das Verfahren der *Generalisierung* ist eine systematische Weise, um benötigte Hilfslemmata zu (er)finden.[4] Wir beschreiben das Verfahren anhand eines Beispiels in ✓`eriFun`:

Wie in der Präsenzübung 1.1 definieren wir Prozeduren „+“ und „∗“ zur Addition bzw. Multiplikation natürlicher Zahlen. Anschließend beweisen wir die Assoziativität und Kommutativität von „+“. Wenn wir nun versuchen, die Assoziativität von „∗“, also das Lemma

$$\texttt{lemma * is associative <=}\ \forall x, y, z : \mathbb{N}$$
$$(\texttt{x} * \texttt{y}) * \texttt{z} = \texttt{x} * (\texttt{y} * \texttt{z})\ ,$$

zu beweisen, bleibt der Induktionsschritt mit der Beweisverpflichtung, also dem Goalterm,

$$(^{-}(x) * y + y) * z = (^{-}(x) * y) * z + y * z$$

stecken. Bei genauerem Hinsehen stellen wir fest, dass dies eine Instanz des Distributivgesetzes für die Multiplikation ist:

$$\forall a, b, c : \mathbb{N}.\ (a + b) * c = a * c + b * c$$

Der Begriff *Instanz* bedeutet, dass wir durch Anwendung der Substitution $\{a/^{-}(x) * y,\ b/y,\ c/z\}$ auf das allgemeine Distributivgesetz genau den Spezialfall erhalten, der im ursprünglichen Beweis als Goalterm übrig geblieben war.

Um das Lemma `* is associative` zu beweisen, kann man also zunächst einen Beweisversuch starten und dann den verbleibenden Goalterm analysieren, ob er sich zu einem Hilfslemma verallgemeinern lässt. Dazu sind Teilterme zu finden, die man durch eine neue Variable ersetzen kann, ohne dadurch die Aussage falsch werden zu lassen. Im obigen Beispiel ist $^{-}(x) * y$ solch ein Teilterm:

$$(\underbrace{^{-}(x) * y}_{a} + y) * z = \underbrace{(^{-}(x) * y)}_{a} * z + y * z \quad \rightsquigarrow \quad (a + y) * z = a * z + y * z$$

Fügt man diese generalisierte Aussage als Lemma hinzu und beweist sie, so kann der ursprüngliche Beweis von `* is associative` mit einer einfachen *Simplification* vollendet werden; ✓`eriFun` findet selbst die benötigte Substitution.

Machen Sie sich anhand dieses Beispiels mit dem Verfahren der Generalisierung vertraut. Sie werden es sicher zur Lösung der folgenden Aufgaben gewinnbringend einsetzen können. Passen Sie aber auf, dass Sie keine Übergeneralisierungen produzieren, wie folgendes Beispiel zeigt: Für einen Sortieralgorithmus *sort* gilt $sort(sort(k)) = sort(k)$. Mit der Generalisierung

$$sort(\underbrace{sort(k)}_{l}) = \underbrace{sort(k)}_{l} \quad \rightsquigarrow \quad sort(l) = l$$

erhält man die Aussage $sort(l) = l$, die jedoch nicht für alle Listen l gilt. In diesem Fall spricht man von einer *Übergeneralisierung*, d. h. die Generalsierung $sort(l) = l$ ist eine Übergeneralisierung von $sort(sort(k)) = sort(k)$. Es ist daher empfehlenswert, Generalisierungen vor einem Verifikationsversuch mittels *Disprove Lemma*, des Interpreters oder des *Refute*-Kommandos zu testen. Findet man dabei ein Gegenbeispiel, so liegt eine Übergeneralisierung vor.

[4] Im Allgemeinen lässt sich leider nicht jedes Hilfslemma durch Generalisierung finden.