Korrekte Software: Grundlagen und Methoden SoSe 17

Serge Autexier
Christoph Lüth

5. Übungsblatt

Ausgabe: 11.05.17
Abgabe: 18.05.17

In diesem Aufgabenblatt wollen wir eine Erweiterung von `cat`, dem C0-Analysewerkzeug, implementieren, welche Beweise im Floyd-Hoare-Stil prüfen kann. Das benötigte Rahmenwerk finden Sie im Git-Repository der Veranstaltung. **Lösen Sie beide Aufgaben, so dass alle Tests (sbt test) durchlaufen.**

5.1 *Check This Proof!* *15 Punkte*

Die Beweise werden als *Annotationen* (speziell formatierte Kommentare, die nur für `cat` und nicht für den Compiler eine Bedeutung tragen) am Programm vermerkt. Abb. 1 zeigt als Beispiel das vollständig annotierte

```
void quotrem(int x, int y)
/** pre  x ≥ 0 && y > 0;
    post x == q*y+ r && 0 ≤ r && r < y;    */
{
 int q;
 int r;

 /** { x ≥ 0 && y > 0 } */
 /** { x == x && 0 ≤ x && y > 0 } */
 r= x;
 /** { x== r && 0 ≤ r && y > 0 } */
 /** { x== 0*y+ r && 0 ≤ r } */
 q= 0;
 /** { x== q*y+r && 0 ≤ r } */
 while (r ≥ y) {
    /** { x== q*y+ r && 0 ≤ r && r ≥ y } */
    /** { x== q*y+ r && y ≤ r } */
    /** { x== (q+ 1)*y+ (r- y) && 0 ≤ r-y } */
    r= r- y;
    /** { x== (q+1)*y+ r && 0 ≤ r } */
    q= q+1;
    /** { x== q*y+ r && 0 ≤ r } */
    }
 /** { x == q*y+ r && 0 ≤ r && !(r ≥ y) } */
 /** { x == q*y+ r && 0 ≤ r && r < y } */
}
```

Abbildung 1: Vollständig annotiertes Programm zur Berechnung des ganzzahligen Teilers

Programm zur Berechnung des ganzzahligen Teilers.

Wir wollen jetzt unser Analysewerkzeug so erweitern, dass es prüft, ob diese Annotationen den Regeln des Floyd-Hoare-Kalküls entsprechen. Grob gesprechen heißt das:

- Stehen zwei Annotationen `/** { P } */` und `/** { Q } */` direkt hintereinander, entspricht dieses der Anwendung der Weakening-Regel.
- Eine Anweisung (Zuweisung, Fallunterscheidung, Schleife) zwischen zwei Annotationen entspricht der jeweiligen Regel für die Zuweisung.

- Zwei Anweisungen direkt hintereinander sind nicht erlaubt.

Die Spezifikationen werden intern durch den Datentypen Term repräsentiert. Eine Funktion

```
sem(ctxt: Env, sp: Expr): Term
```

übersetzt einen Ausdruck der abstrakten Syntax (Expr) in einen Term. Für Terme gibt es ferner die Methode

```
abstract class Term {
  subst(x: Identifier, t: Term): Term
}
```

die in einem Term jedes Vorkommen des Bezeichners x durch den Term t ersetzt.

Kern des Beweisprüfers (proof checker) sind zwei wechselseitig rekursive Funktionen, welche für eine einzelne Anweisung, oder für eine Sequenz von Anweisungen, prüft, ob die Abfolge von annotierten Spezifikationen und Anweisungen den Regel des Floyd-Hoare-Kalküls entspricht:

```
checkTriple(ctxt: Env, pre: Spec, s: Stmt, post: Spec) : List[Term]
checkTripleStmts(ctxt: Env, pre: Spec, s: List[Stmt], post: Spec): List[Term]
```

Der Rückgabewert ist eine Liste von Termen, welche *Beweisverpflichtungen* sind, die sich aus der Anwendung der Weakening-Regel ergeben und das Programm regelkonform ist. Ansonsten wird ein Fehler geworfen.

Insgesamt implementieren wir den Beweisprüfer als eine Funktion welche für eine gegebene Liste von Funktionsdefinitionen eine Liste aus Beweispverpflichtungen mit dem Namen der Funktion und der Liste der Beweisverpflichtungen zurückgibt (Res[List[ProofObligation]]).

Der Aufruf auf der Kommandozeile für die Verifikation des Beispielprogramms aus Abbildung 1 lautet

```
sbt "run --proofcheck --fun= quotrem src/test/examples/test-proofcheck-quotrem.c"
```

5.2 *Vereinfachungen* *5 Punkte*

Die entstehenden Beweisverpflichtungen sind zum großen Teil sehr trivial. Wir wollen sie daher etwas vereinfachen, indem wir eine Funktion

```
simplify(t: Term): Term
```

implementieren, die Terme wie folgt vereinfacht:

1. Mit dem Extraktor Implies wird eine Implikation in Prämisse prem und Konklusion conc wie folgt zerlegt:[1]

   ```
   t match { case Implies(prem, conc) ⇒ ... }
   ```

   Prämissen und Konklusion werden jetzt mit den folgende Schritten vereinfacht:

   (a) Ein Term t in konjunktiver Form $t = t_1 \wedge t_2 \wedge \ldots \wedge t_n$ wird mit der Funktion

   ```
   conjoints(t: Term): Option[List[Term]]
   ```

   in eine Liste von Konjunktion dekomponiert. Die einzelnen Konjunkte werden wie folgt vereinfacht:

   i. Explizite Negationen auflösen, d.h. aus `!a == b` wird $a \neq b$ usw.;
   ii. Die Relationen größer und größer-gleich werden durch kleiner und kleiner-gleich ersetzt;
   iii. Reflexive Gleichungen und Ungleichungen (d.h. `a == a`, $a \leq a$) werden durch trueT ersetzt;

   (b) Danach wird die Liste als ganzes vereinfacht, indem Vorkommen von trueT gestrichen, und mehrfache auftretende Konjunkte durch ein einzelnes ersetzt werden.

   (c) Optional können jetzt noch Vereinfachungen wie $A \leq B \wedge B \leq A$ zu $A = B$ implementiert werden.

2. Die vereinfachte Liste von Konjunktion der Konklusion wird vereinfacht, indem alle Konjunkte, die auch in den Prämissen vorkommen, gestrichen werden.

3. Danach wird aus den Listen von Konjunkten der Prämisse und Konklusion wieder eine Implikation (Funktion implies aus `Logic.scala`). Ist die Liste der Konjunkte der Konklusion leer, wurde die Beweisverpflichtung gezeigt, und kann entfallen.

Der Aufruf auf der Kommandozeile für die Verifikation des Beispielprogramms aus Abbildung 1 inklusive Simplifikation lautet

```
sbt "run --proofcheck --simp --fun=quotrem src/test/examples/test-proofcheck-quotrem.c"
```

[1] Ähnliche Extraktoren And, Or, Not, Equals, NotEquals, Less, LEssEquals, Greater, GreaterEquals für Konjunktion, Disjunction usw. finden sich in `Logic.scala`.