12 Ein Blick in formale Programmiermethoden: Der Hoare Kalkül

Überblick

- Ziele
- Grundlagen
- Einfache Regeln
- Schleifen
- Funktionsaufruf
- Terminierung
- Verifikation und Programmentwicklung
- Ausblick

Literatur: Duden Informatik Stichworte Semantik und Verifikation u.v.a.m. D. Gries, The Science of Programming.

12.1 Ziele

„Testen kann die Anwesenheit aber nie die *Abwesenheit* von Fehlern beweisen“

Frage: Können wir *beweisbar* korrekte Programme schreiben?

Antwort

Ja: Wir brauchen nur eine formale Semantik der Programmiersprache und viel Grips

Nein: C++ (Java,C, ...) hat definitiv keine formale Semantik

Nein: Sind die Compiler verifiziert?

Nein: Die Spezifikation kann fehlerhaft sein

Nein: Soviel Grips haben wir nicht

Trotzdem

- SML
- C++ hat eine formal spezifizierbare Teilmenge
- Wir können eine saubere Entwurfssprache verwenden
- Teilaspekte einer Verifikation können helfen
- Formale Methoden erleichtern Entwurf

12.2 Grundlagen

12.2.1 Zustandsmenge

Der Programmzustand wird durch die Belegung aller Variablen festgelegt.

Beispiel:

Function ggt($a : \mathbb{Z}$, $b : \mathbb{Z}$): $\mathbb{Z}$
  $r : \mathbb{Z}$
  $r := b$
  while $r \neq 0$ **do**
    $r := a \bmod b$
    $a := b$;
    $b := r$
  od
  return a

Zustandsmenge ist $\{(a, b, r)\} = \mathbb{Z} \times \mathbb{Z} \times \mathbb{Z}$.

12.2.2 Zusicherungen

(engl. assertions)

Prädikatenlogische Ausdrücke über den Variablen definieren Teilmenge der Zustandsmenge M.

$\{Q\} \equiv$ „An dieser Programmstelle is Q“ wahr

Zustand$\in \{m \in M \mid Q\}$

Function ggt($a : \mathbb{Z}$, $b : \mathbb{Z}$): $\mathbb{Z}$
  $r : \mathbb{Z}$
  $\{a > 0 \wedge b > 0\}$
  $r := b$
  while $r \neq 0$ **do**
    $r := a \bmod b$
    $a := b$;
    $b := r$
  od
  $\{r = \text{ggt}(a, b)\}$
  return r

Allgemein

$\{P\}\, A\, \{Q\}$

Vorbedingung: Zusicherung, die vor einer Anweisungsfolge A gilt

Nachbedingung: Zusicherung, die nach einer Anweisungsfolge A gilt

relativ!

12.2.3 Vorgehensweise

- Formale (und zwar *axiomatische*) Semantik beschreibt für jedes statement der Programmiersprache die Wirkung auf Vor- und Nachbedingung
- Semantik der Datentypen nicht vergessen (z.B. Rechenregeln für Zahlen)
- Regeln beschreiben, wie sich Zusicherungen umformen lassen

Partielle Korrektheit

Aus globaler Vorbedingung folgt Nachbedingung falls das Programm terminiert

Totale Korrektheit

na?

12.3 Einfache Regeln

12.3.1 Sehr einfache Regeln

Leeranweisungsaxiom: $\{P\}$; $\{P\}$

Sequenz $\dfrac{\{P\}\, A\, \{Q\}\,, \{Q\}\, B\, \{R\}}{\{P\}\, A; B\, \{R\}}$

Vorbedingung stärken: $\dfrac{P' \Rightarrow P, \{P\}\, A\, \{Q\}}{\{P'\}\, A\, \{Q\}}$

Nachbedingung schwächen:
$\dfrac{\{P\}\, A\, \{Q\}\,, Q \Rightarrow Q'}{\{P\}\, A\, \{Q'\}}$

12.3.2 Zuweisungsaxiom

$$\{P[x \leftarrow e]\}\, x := e\, \{P\}$$

Die **Substitution** $[x \leftarrow e]$ steht für „ersetze Variable x im Prädikat P durch Ausdruck e“. (*Freie* Vorkommen von x: $(\forall x : x + 1 > x)[x \leftarrow 1] = \forall x : x + 1 > x)$

Beispiel:

$\{i = 42\}$
$\{i + 1 = 43\}$
$i := i + 1$
$\{i = 43\}$

Beispiel

Procedure swap($a : \mathbb{Z}$, $b : \mathbb{Z}$)
$\{a = a_0 \wedge b = b_0\}$ (* alte Werte merken *)
$\{a_0 + b_0 = a_0 + b_0 \wedge a = a_0 \wedge b = b_0\}$
$\{a + b = a_0 + b_0 \wedge a = a_0 \wedge b = b_0\}$
$\{a + b = a_0 + b_0 \wedge b = b_0\}$
$a := a + b$
$\{a = a_0 + b_0 \wedge b = b_0\}$
$\{a = a_0 + b_0 \wedge a - b = a_0 + b_0 - b_0\}$
$\{a = a_0 + b_0 \wedge a - b = a_0\}$
$b := a - b$
$\{a = a_0 + b_0 \wedge b = a_0\}$
$\{a - b = b_0 \wedge b = a_0\}$
$a := a - b$
$\{a = b_0 \wedge b = a_0\}$

Warum schlechtes Beispiel?

12.3.3 Bedingung

$$\frac{\{P \wedge B\}\, I\, \{Q\}\,, \{P \wedge \neg B\}\, E\, \{Q\}}{\{P\}\, \textbf{if } B \textbf{ then } I \textbf{ else} E\, \{Q\}}$$

Beispiel: $y = |x|$

$\{\text{true}\}$
if $x > 0$ **then**
$\{x > 0\}$
$\{x = x \wedge x > 0\}$
$y := x$
$\{y = x \wedge x > 0\}$
$\{y = |x|\}$
else
$\{x \leq 0\}$
$\{-x = -x \wedge x \leq 0\}$
$y := -x$
$\{y = -x \wedge x \leq 0\}$
$\{y = |x|\}$
fi
$\{y = |x|\}$

12.4 Schleifen und Rekursion

12.4.1 While

$$\frac{\{I \wedge B\}\, L\, \{I\}}{\{I\}\, \textbf{while } B \textbf{ do } L\, \{I \wedge \neq B\}}$$

I heißt **Schleifeninvariante**

12.4.2 Beispiel: ggt

Function ggt($a : \mathbb{Z}$, $b : \mathbb{Z}$): $\mathbb{Z}$
  $r : \mathbb{Z}$
  $\{a > 0 \wedge b > 0\}$
  $\{a > 0 \wedge b > 0 \wedge a = a_0 \wedge b = b_0\}$
  $\{\mathcal{I} := \text{ggt}(a, b) = \text{ggt}(a_0, b_0)\}$
  while $b \neq 0$ **do**
    $\{\mathcal{I} \wedge b \neq 0\}$
    $\{\mathcal{I} \wedge b \neq 0 \wedge a \bmod b = a \bmod b\}$
    $r := a \bmod b$
    $\{\mathcal{I} \wedge r = a \bmod b\}$
    $\{\text{ggt}(b, r) = \text{ggt}(a_0, b_0)\}$
    $a := b$;
    $\{\text{ggt}(a, r) = \text{ggt}(a_0, b_0)\}$
    $b := r$
    $\{\text{ggt}(a, b) = \text{ggt}(a_0, b_0)\} \equiv \{\mathcal{I}\}$
  od
  $\{\mathcal{I}\}$
  $\{a = \text{ggt}(a, b)\}$
  return a

Parallele Zuweisung

Function ggt($a : \mathbb{Z}$, $b : \mathbb{Z}$): $\mathbb{Z}$
  $\{a > 0 \wedge b > 0 \wedge a = a_0 \wedge b = b_0\}$
  $\{\mathcal{I} := \text{ggt}(a, b) = \text{ggt}(a_0, b_0)\}$
  while $b \neq 0$ **do**
    $\{\mathcal{I} \wedge b \neq 0\}$
    $(a, b) := (b, a \bmod b)$
    $\{\text{ggt}(a, b) = \text{ggt}(a_0, b_0)\} \equiv \{\mathcal{I}\}$
  od
  $\{a = \text{ggt}(a, b)\}$
  return a

12.4.3 Beispiel: Binäre Suche

b: sorted **array** [1..n] **of** TotalOrder
x: TotalOrder
$\{b[1] \leq x < b[n]\}$
$\{b[1] \leq x < b[n] \wedge 1 < n \wedge 1 = 1 \wedge n = n\}$
$(i, j) := (1, n)$
$\{b[i] \leq x < b[j] \wedge 1 < n \wedge i = 1 \wedge j = n\}$
$\{\mathcal{I} := b[i] \leq x < b[j] \wedge 1 \leq i < j \leq n\}$
while $j \neq i + 1$ **do**
  $\{\mathcal{I} \wedge j \neq i + 1\} \Rightarrow \{\mathcal{I} \wedge i < \lfloor \frac{i+j}{2} \rfloor 2 < n\}$
  $m := \lfloor \frac{i+j}{2} \rfloor$
  $\{\mathcal{I} \wedge i < m < j\}$
  if $x < b[m]$ **then**
    $\{\mathcal{I} \wedge i < m < j \wedge x < b[m]\}$
    $\{b[i] \leq x \wedge 1 \leq i < m < n \wedge x < b[m]\}$
    $j := m$
    $\{b[i] \leq x \wedge 1 \leq i < j < n \wedge x < b[j]\} \equiv \{\mathcal{I}\}$
  else $\{\mathcal{I} \wedge i < m < j \wedge b[m] \leq x\}$
    $\{x < b[j] \wedge 1 \leq m < j \leq n \wedge b[m] \leq x\}$
    $i := m$
    $\{x < b[j] \wedge 1 \leq i < j \leq n \wedge b[i] \leq x\} \equiv \{\mathcal{I}\}$
od $\{\mathcal{I} \wedge j = i + 1\}$
$\{b[i] \leq x < b[i + 1] \wedge i < n\}$

Interpolationssuche

b: sorted **array** [1..n] **of** TotalOrder
x: TotalOrder
$(i, j) := (1, n)$
$\{\mathcal{I} := b[i] \leq x < b[j] \wedge 1 \leq i < j \leq n\}$
while $j \neq i + 1$ **do**
  $\{\mathcal{I} \wedge j \neq i + 1\}$
  $m := i + \max(1, \left\lfloor (j - i) \frac{x - b[i]}{b[j] - b[i]} \right\rfloor)$
  $\{\mathcal{I} \wedge i < m < j\}$
  if $x < b[m]$ **then**
    $j := m$
  else
    $i := m$
od $\{b[i] \leq x < b[i + 1] \wedge i < n\}$

12.5 Funktionsaufruf

Gegeben sei eine *seiteneffektfreie* Funktion:

Function $f(x)$
  $\{P\}$ (* Vorbedingung *)
  A
  $\{Q\}$ (* Nachbedingung *)
  return y

$$\frac{\{P\}\ A\ \{Q\}}{\{P' \wedge P[x \leftarrow e]\}\ b := f(e)\ \{P' \wedge Q[x, y \leftarrow e, b]\}}$$

In P' darf b nicht vorkommen.

12.5.1 Beispiel: Rekursiver ggt

Function ggtF$((a,b) : \mathbb{Z}^2)$: $\mathbb{Z}$
  $\{a > 0 \wedge b > 0\}$
  $r := a \bmod b$
  $\{r = a \bmod b \wedge a > 0 \wedge b > 0\}$
  if $r = 0$ **then**
    $\{r = 0 \wedge r = a \bmod b \wedge a > 0 \wedge b > 0\}$
    $\{b = \text{ggt}(a,b)\}$
    $y := b$
    $\{y = \text{ggt}(a,b)\}$
  else
    $\{r \neq 0 \wedge r = a \bmod b \wedge a > 0 \wedge b > 0\}$
    $\{\text{ggt}(a,b) = \text{ggt}(b,r) \wedge b > 0 \wedge r > 0\}$
    $y :=$ ggtF$((b,r))$
    $\{\text{ggt}(a,b) = \text{ggt}(b,r) \wedge y = \text{ggt}(b,r)\}$
    $\{y = \text{ggt}(a,b)\}$
  $\{y = \text{ggt}(a,b)\}$
  return y

12.6 Terminierung

Programm A **total korrekt** gdw. A partiell korrekt und immer terminiert.

(Warum nicht völlig allgemein?)

Wie weist man Terminierung nach?

Oft ausreichend: **Schleifenvariante**

Funktion t : Zustand $\to \mathbb{N}$

while B **do**
  $\{t > 0\}$
  A
  {t hat sich echt verkleinert}

Beispiele

ggt: b

binäre Suche: $j - i$

12.7 Verifikation und Programmentwicklung

- Vor- und Nachbedingungen sind Spezifikationshilfsmittel (→Programming by contract).
- Tatsächliches Testen von assertions deckt Fehler auf
- Schleifeninvarianten sind oft „Essenz“ eines Algorithmus. Vorher ausdenken!
- Verifikation für kritische Programmteile (Atomkraftwerk, ...)
- und für „tricky“ Code.

12.7.1 Vorgehensweise

- Programm und Beweis Hand-in-Hand entwickeln.
- Formalismus bei unkritischen Teilen zurückdrängen
- Geignete Abkürzungen
- Verifikation von hinten nach vorne.
- Seiteffektfreie Funktionen bevorzugen
- Möglichst schwache Schleifenbedingung.
- Invarianten sind Verallgemeinerung der Postcondition (Luft aus Ballon ablassen.)

12.8 Blick über den Tellerrand

12.8.1 Methodik

- Saubere logische Grundlagen
- Allgemeinere Programme. (Zuweisung an Arrays, Seiteneffekte, Ein- Ausgabe ...)
- Leistungsfähige Spezifikationssprachen (z.B. „Z“).
- Semantik von Programmiersprachen
- Verifikation größerer Programme

12.8.2 Werkzeuge

- Funktionale Sprachen erleichtern Verifikation
- Logische Programmiersprachen
- Auch imperative Sprachen lassen sich säubern
- Unterstützung für Assertion-Checking
  - Quantifizierte Ausdrücke
  - OO-Unterstützung
- Interaktive Verifikation
- Automatische Theorembeweiser
  - Übernehmen Routineaufgaben
  - Überprüfen Beweise

Frohe Weihnachten und ein gutes neues Jahr