Einführung in die Programmiertechnik

Verifikation

Vermeidung von Fehlern

- Verwendung geeigneter Werkzeuge
  - Auswahl einer dem Problem angemessenen Programmiersprache
  - Verwendung eines zuverlässigen Compilers
  - Verwendung von Fehlerdiagnosewerkzeugen/-optionen
- *Erst überlegen, dann Programmieren*
  - Planung: Analyse, Design
- Klarer Programmierstil
  - Offensichtliche Korrektheit, leichte Lesbarkeit statt Trickserei

Zwischenbehauptungen

- Annahmen über die Belegung von Variablen, während Algorithmus läuft
  - Vorbedingungen (*preconditions*): Welche Forderungen werden an die Eingabe gestellt?
  - Nachbedingungen (*postconditions*): Welche Eigenschaften erfüllt das Ergebnis?
  - Invarianten (*invariants*): Welche Eigenschaften gelten dauerhaft?
- Zwischenbehauptungen (*assertions*): Bedingungen, die an einem bestimmten Punkt der Ausführung erwartet werden
  - geeignet zur Überprüfung von Vor- und Nachbedingungen
  - bedingt geeignet zur Überprüfung von Invarianten: Test muss an “strategischer” Stelle erfolgen
    - etwa: bei jedem Schleifendurchlauf

Zwischenbehauptungen (2)

- Python: assert-Anweisung
  - assert_stmt ::= "assert" expression ["," expression]
- Falls erster *Ausdruck* falsch ist, wird Ausnahme AssertionError ausgelöst (falls vorhanden, mit zweitem Ausdruck als Parameter)
  - assert alter >= 18, “muss volljaehrig sein”
- python -O übergeht alle Tests von Zwischenbehauptungen
  - “Optimierung”: Reduzierung der Laufzeit
  - geeignet nach Abschluss der Testphase
- Testen: Ausführung des Programms mit verschiedenen Eingaben
  - Zwischenbehauptungen werden nur für getestete Eingaben überprüft

Partielle Korrektheit

- Seinen P und Q Bedingungen, S ein Programm:
  - **{P} S {Q}** bedeutet: Falls Vorbedingung P erfüllt ist, ist nach Ablauf von S Nachbedingung Q erfüllt
  - Aussage gilt auch, wenn S nicht terminiert
    - {P} S {false} ist gleibedeutend mit: P terminiert nicht
- partielle Korrektheitsaussage (PCA, *partial correctness assertion*)
- Ziel: Aufstellung und Beweis von PCAs

PCAs: Ein Beispiel

```
{ N > 0 }
sum = 0
i = 0
while i < N:
    i = i+1
    sum = sum + i
{ sum = 1 + 2 + ... + N }
```

- Wenn bei Start des Programms die Konstante N > 0 ist, dann wird nach der Terminierung die Variable sum den Wert 0 + 1 + 2 + ... + N haben

Zerlegung durch Zwischenbehauptungen

- Zerlegung eines sequentiellen Programms in zwei Teile; Nachbedingung des ersten Teils ist Vorbedingung des zweiten

```
{ N > 0 }
sum = 0
i = 0
{ N > 0 and sum = 0 and i = 0 }
```

```
{ N > 0 and sum = 0 and i = 0 }
while i < N:
    i = i+1
    sum = sum + i
{ sum = 1 + 2 + ... + N }
```

Zerlegung durch Zwischenbehauptungen (2)

- zweiter Schritt: weitere Zerlegung des ersten Teils

{ N > 0 }
sum = 0
{ N > 0 **and** sum = 0 }

{ N > 0 **and** sum = 0 }
i = 0
{ N > 0 **and** sum = 0 **and** i = 0 }

Zerlegung durch Zwischenbehauptungen (3)

- Hintereinanderausführungsregel:

$$\frac{\{P\}S_1\{R\}, \{R\}S_2\{Q\}}{\{P\}S_1; S_2\{Q\}}$$

- Wahl einer geeigneten Zwischenbehauptung ist kritisch:
  - Ist R zu stark, lässt sich {P} S_1 {R} nicht beweisen
  - Ist R zu schwach, lässt sich {R} S_2 {Q} nicht beweisen

Zuweisungsregel

- Betrachten: {P} v = t {Q}
- Beispiel: Gilt {y < 2*(x+1)} x = x+1 {x > y-x}?
  - Nennen x' neuen Wert von x, entsprechend y'
  - Gilt also (1)?

$$y < 2(x+1) \implies x' > y' - x'(1)$$

  - y' = y (y wurde nicht zugewiesen)
  - x' = x+1
  - Einsetzen: x+1 > y - (x+1)

$$y < 2(x+1) \implies 2(x+1) > y \implies x+1 > y-(x+1)$$

Zuweisungsregel (2)

- Verallgemeinert: Q[v/t] ist Aussage Q, wobei v durch t ersetzt wurde

$$\frac{P \implies Q[v/t]}{\{P\}v := t\{Q\}}$$

- Beispiel: {true} x = 5 { x<10}
  - weil: true ⇒ (x < 10)[x/5], also true ⇒ (5 < 10)

Rückwärtsbeweis

- Beispiel: Vertauschen zweier Variablen ohne Hilfsvariable

  {x = A **and** y = B}

  x = x - y

  y = x + y

  x = y - x

  { x = B **and** y = A}

- Ziel: Finden einer Zwischenbehauptung R, so dass

  {x = A **and** y = B}

  x = x - y

  y = x + y

  {R}
- sowie

  {R}

  x = y - x

  { x = B **and** y = A}
- R möglichst schwach, so dass immernoch
  - R ⇒ {x = B **and** y=A}[x/y-x], also
  - R ⇒ {y-x = B **and** y=A}
  - Lösungsansatz: R ≡ {y-x = B **and** y=A}

- Gesucht nun Zwischenbehauptung R2, so dass

  {x = A **and** y = B}

  x = x - y

  {R2}
- sowie

  {R2}

  y = x + y

  {y-x = B and y=A}
- Wieder R2 möglichst schwach:
  - R2 ⇒ {y-x = B **and** y=A}[y/x+y], also
  - R2 ⇒ {(x+y)-x = B **and** x+y=A}
  - Lösungsansatz: R2 ≡ {y = B **and** x+y=A}
- R2 folgt aus Zuweisungsregel:
  - {x=A **and** y = B} ⇒ {y=B **and** x+y=A}[x/x-y]

Rückwärtsbeweis (4)

- Idee: Aufstellen von Zwischenbedingungen von hinten
  - mit jeweils schwächster akzeptabler Vorbedingung
  - *weakest precondition*

if-then-else-Regel

$$\frac{\{P \wedge B\}S_1\{Q\}, \{P \wedge \neg B\}S_2\{Q\}}{\{P\}if\,B : S_1 else : S_2\{Q\}}$$

- Beispiel:

{ x = A}
if x > 0:
    z = A
else:
    z = -A
{ z = |A| }

- Zu zeigen:
  - { x = A **and** x > 0 } z = x { z = |A|}, und
  - { x = A **and not** x > 0} z = -x { z = |A| }

Einarmige Alternative und Abschwächungsregel

- if B: S
- Rückführung auf if-then-else-Regel durch *pass*

```
if B:
    S
else:
    pass
```

- Leeranweisung: Abschwächungsregel
  - abgeleitet aus {P} x=x {Q}
-
-
-
-

$$\frac{P \implies Q}{\{P\}\ pass\{Q\}}$$

Einarmige Alternative

- if-then-Regel

$$\frac{\{P \wedge B\}S\{Q\}, P \wedge \neg B \implies Q}{\{P\}if\,B : S\{Q\}}$$

- Beispiel:

{x = A}
if x < 0:
    x = -x
{x = |A|}

Invarianten und while-Schleifen

- while B: S
- Gesucht: Invarianten von S
  - {I} S {I}
- Für Schleifen stärkere Vorbedingung akzeptabel
- Def: Sei B ein boolescher Ausdruck und S eine Anweisung. I heißt **Invariante bezüglich B**, falls {I and B} S {I}
- Beispiel: ggT(x,y) = N Invariante bezüglich x ≠ y von

```
if x > y:
        x = x - y
else:
        y = y-x
```

Schleifenregel

$$\frac{\{I \wedge B\}\ S\{I\}}{\{I\}\ whileB : S\{I \wedge \neg B\}}$$

- while-Schleifen werden oft mit der Schleifeninvariante *annotiert*

```
i = sum = 0
while i < N:    {sum = Σ_{k=0}^{i} k}
    i = i+1
    sum = sum + i
```

Starke und schwache Invarianten

- Für jede Schleife kann man beliebig viele Invarianten finden
  - beispielsweise {true}, {false}
  - Ist Invariante zu schwach, kann man Korrektheit des Gesamtprogramms nicht beweisen
  - Ist Invariante zu stark, kann man Invariante nicht beweisen
- Strategie:
  1. Aufstellen einer denkbaren Invariante I
  2. Beweis der Voraussetzung der Schleifenregel: {I **and** B} S {I}
  3. Einsetzen von { I } als Zwischenbehauptung vor der Schleife
  4. Einsetzen von { I **and not** B } als Zwischenbehauptung nach der Schleife

Andere Programmkonstrukte

- Andere Schleifen: Rückführung auf while-Schleife
- Unterprogramme: Annahme von PCAs unabhängig vom Rufer
  - Beweis der Vorbedingung für jeden Aufruf
  - u.U. nicht ausreichend: Nachbedingung soll spezifisch für Parameter sein
    - separate Beweise pro Aufruf
  - Rekursive Aufrufe: Beweis durch Induktion über Rekursionstiefe
- Ausnahme: z.B. Rückführung auf bedingte Anweisungen
  - manuelle Transformation i.d.R. nicht praktikabel

Programmverifizierer

- Verifikation zu großen Teilen mechanisch
  - Rückwärtsschließen
  - Umformulierung des Programms
  - Überprüfen der korrekten Anwendung von Schlußregeln
  - Aufstellung von Schleifeninvarianten schwer automatisierbar
- Programmverifizierer: Programme zur Durchführung von Verifikationen
  - oft integriert mit Theorembeweiser, um Implikationen zu beweisen oder Formeln zu vereinfachen
- Beispiel: Gumms *NPPV* (*New Paltz Program Verifier*)