Korrekte Software: Grundlagen und Methoden
Vorlesung 1 vom 06.04.17: Einführung

Serge Autexier, Christoph Lüth

Universität Bremen

Sommersemester 2017

Organisatorisches

- Veranstalter:

Christoph Lüth
christoph.lueth@dfki.de
MZH 4186, Tel. 59830

Serge Autexier
serge.autexier@dfki.de
Cartesium 2.11, Tel. 59834

- Termine:
  - Montag, 14 – 16, MZH 6210
  - Donnerstag, 14 – 16, MZH 1110
- Webseite:

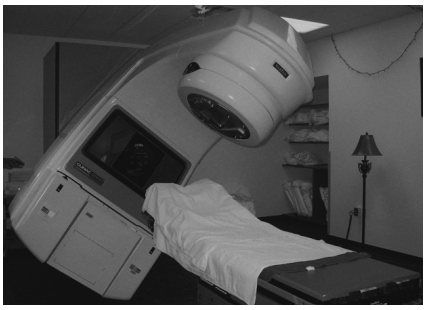
`http://www.informatik.uni-bremen.de/~cxl/lehre/ksgm.ss17`

Prüfungsformen

- 10 Übungsblätter (geplant)
- Prüfungsform 1:
  - Bearbeitung der Übungsblätter,
  - Fachgespräch,
  - Note aus den Übungsblättern.
- Prüfungsform 2:
  - Mind. ausreichende Bearbeitung der Übungsblätter (50%),
  - mündliche Prüfung,
  - Note aus der Prüfung.

Warum Korrekte Software?

Software-Disaster I: Therac-25



Bekannte Software-Disaster II: Ariane-5

Bekannte Software-Disaster III: Airbus A400M

Inhalt der Vorlesung

Themen

Korrekte Software im Lehrbuch:

- Spielzeugsprache
- Wenig Konstrukte
- Kleine Beispiele

Korrekte Software im Einsatz:

- Richtige Programmiersprache
- Mehr als nur ganze Zahlen
- Skalierbarkeit — wie können große Programme verifiziert werden?

Inhalt

- Grundlagen:
  - Der Hoare-Kalkül — Beweis der Korrektheit von Programmen
  - Bedeutung von Programmen: Semantik
- Erweiterung der Programmkonstrukte und des Hoare-Kalküls:
  1. Referenzen (Zeiger)
  2. Funktion und Prozeduren (Modularität)
  3. Reiche Datenstrukturen (Felder, `struct`)
- Übungsbetrieb:
  - Betrachtete Programmiersprache: "C0" (erweiterte Untermenge von C)
  - Entwicklung eines Verifikationswerkzeugs in Scala
  - Beweise mit Princess (automatischer Theorembeweiser)

Einige Worte zu Scala

Scala

- A scalable language
- Rein objektorientiert
- Funktional
- Eine "JVM-Sprache"
- Seit 2004 von Martin Odersky, EPFL Lausanne (`http://www.scala-lang.org/`).
- Seit 2011 kommerziell durch Lightbend Inc. (formerly Typesafe)

Scala am Beispiel: `01-GCD.scala`

Was sehen wir hier?

```
def gcdLoop(x: Long, y: Long): Long =
  {
    var a = x
    var b = y
    while (a ≠ 0) {
      val temp = a
      a = b % a
      b = temp
    }
    return b
  }

def gcd(x: Long, y: Long): Long =
  if (y == 0) x else gcd (y, x % y)
```

- Variablen, veränderlich (var)
  - *Mit Vorsicht benutzen!*
- Werte, unveränderlich (val)
- while -Schleifen
  - *Unnötig!*
- Rekursion
  - Endrekursion wird optimiert
- Typinferenz
  - Mehr als Java, weniger als Haskell
- Interaktive Auswertung

Scala am Beispiel: `02-Rational-1.scala`

Was sehen wir hier?

```
class Rational(n: Int, d: Int) {

  require(d ≠ 0)

  private val g = gcd(n.abs, d.abs)
  val numer = n / g
  val denom = d / g

  def this(n: Int) = this(n, 1)

  def add(that: Rational): Rational =
    new Rational(
      numer * that.denom + that.numer
        * denom,
      denom * that.denom
    )

  override def toString = numer +"/"+
    denom

  private def gcd(a: Int, b: Int):
    Int =
    if (b == 0) a else gcd(b, a % b)
}
```

- Klassenparameter
- Konstruktoren (this)
- Klassenvorbedingungen (require)
- private Werte und Methoden
- Methoden, Syntax für Methodenanwendung
- override (nicht optional)
- Overloading
- Operatoren
- Companion objects (object)

Algebraische Datentypen: `03-Expr.scala`

Was sehen wir hier?

```
abstract class Expr
  case class Var(name: String) extends
    Expr
  case class Number(num: Double)
    extends Expr
  case class UnOp(operator: String,
    arg: Expr) extends Expr
  case class BinOp(operator: String,
      left: Expr, right: Expr) extends
        Expr

def eval(expr: Expr): Double = expr
  match {
  case v: Var ⇒ 0     // Variables evaluate to
    0
  case Number(x) ⇒ x
  case BinOp("+", e1, e2) ⇒ eval(e1)
    + eval(e2)
  case BinOp("*", e1, e2) ⇒ eval(e1)
    * eval(e2)
  case UnOp("-", e) ⇒ - eval(e)
  }

val e = BinOp("*", Number(12),
          UnOp("-", BinOp("+",
            Number(2.3),
              Number(3.7))))
```

- case class erzeugt
  - Factory-Methode für Konstruktoren
  - Parameter als implizite val
  - abgeleitete Implementierung für toString , equals
  - ... und pattern matching (match)
- Pattern sind
  - case 4 => Literale
  - case C(4) => Konstruktoren
  - case C(x) => Variablen
  - case C(_) => Wildcards
  - case x: C => getypte pattern
  - case C(D(x: T, y), 4) => geschachtelt

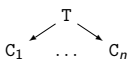
Implementierung algebraischer Datentypen

Haskell:

```
data T = C1 | ... | Cn
```

- Ein Typ `T`
- Konstruktoren erzeugen Datentyp

Scala:



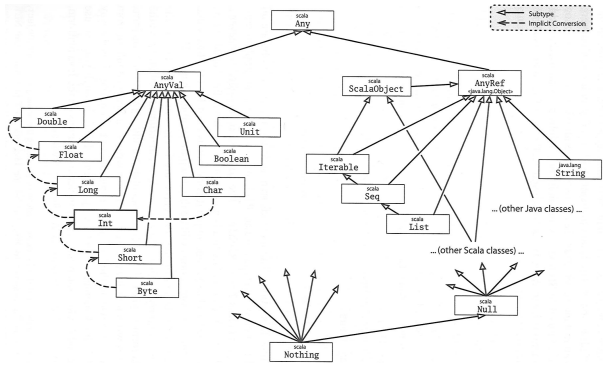
- Varianten als Subtypen
- Problem und Vorteil: Erweiterbarkeit
- sealed verhindert Erweiterung

Das Typsystem

Das Typsystem behebt mehrere Probleme von Java:

- Werte vs. Objekte
- Scala vs. Java
- NULL references

Vererbungshierarchie



Quelle: Odersky, Spoon, Venners: *Programming in Scala*

Parametrische Polymorphie

- Typparameter (wie in Haskell, Generics in Java), Bsp. List[T]
- Problem: Vererbung und Polymorphie
- Ziel: wenn $S < T$, dann List[S] < List[T]
- Does not work — `04-Ref.hs`
- Warum?
  - Funktionsraum nicht monoton im ersten Argument
  - Sei $X \subseteq Y$, dann $Z \longrightarrow X \subseteq Z \longrightarrow Y$, aber $X \longrightarrow Z \not\subseteq Y \longrightarrow Z$
  - Sondern $Y \longrightarrow Z \subseteq X \longrightarrow Z$

Typvarianz

class C[+T]
- Kovariant
- Wenn $S < T$, dann C[S] < C[T]
- Parametertyp T nur im Wertebereich von Methoden

class C[T]
- Rigide
- Kein Subtyping
- Parametertyp T kann beliebig verwendet werden

class C[-T]
- Kontravariant
- Wenn $S < T$, dann C[T] < C[S]
- Parametertyp T nur im Definitionsbereich von Methoden

Beispiel:

```
class Function[-S, +T] {
  def apply(x:S) : T
  }
```


Traits: `05-Funny.scala`

Was sehen wir hier?

- Trait (Mix-ins): abstrakte Klassen, Interfaces; Haskell: Typklassen
- „Abstrakte Klasse ohne Oberklasse“
- Unterschied zu Klassen:
  - Mehrfachvererbung möglich
  - Keine feste Oberklasse (super dynamisch gebunden)
  - Nützlich zur Strukturierung (Aspektorientierung)
- Nützlich zur Strukturierung:

$$\textit{thin interface} + \textit{trait} = \textit{rich interface}$$

Beispiel: `05-Ordered.scala`, `05-Rational.scala`

Komprehension mit for : `06-For.scala`

```
val l1 =
   List(1,2,3,4,5,6,7,8,9)
for { x<- l1;
      if (x % 2 == 0)
    } yield 2*x+1

val l1= List.range(1,9)
def half(x: Int):
   Option[Int] =
  if (x%2 == 0) Some (x/2)
     else None
for { x<- l1;
      y<- half(x)
    } yield y
```

- For-Schleife iteriert über Liste:
  - Generatoren, Filter, Result
- Für andere Datentypen: Option
- Für beliebige Datentypen T mit

```
def map[B](f:(A) ⇒
    B): T[B] =  ???
def flatMap[B](f:(A)⇒
    T[B]): T[B] = ???
```


Was wir ausgelassen haben...

- Gleichheit: == (final), equals (nicht final), eq (Referenzen)
- Implizite Parameter und Typkonversionen
- Stringinterpolation, XML
- Nebenläufigkeit (Aktoren, Futures)
- Typsichere Metaprogrammierung
- Das *simple build tool* sbt
- Scala-Plugin für IntelliJ
- Der JavaScript-Compiler scala.js

Scala Zusammenfassung

- Objekt-orientiert:
  - Veränderlicher, gekapselter Zustand
  - Subtypen und Vererbung
  - Klassen und Objekte
- Funktional:
  - Unveränderliche Werte
  - Parametrische und Ad-hoc Polymorphie
  - Funktionen höherer Ordnung
  - Hindley-Milner Typinferenz

Zusammenfassung

- Zum Lernen von Scala: 0. Übungsblatt
  - Keine Punkte, aber Kurzbewertung wenn gewünscht.
- Nächste Woche:
  - Reprise der Hoare-Logik
  - Semantik
  - Erste Gehversuche mit dem Analysewerkzeug

Korrekte Software: Grundlagen und Methoden
Vorlesung 2 vom 10.04.17: Die Floyd-Hoare-Logik

Serge Autexier, Christoph Lüth

Universität Bremen

Sommersemester 2017

Organisatorisches

Die Übung am Donnerstag, 13.04.17, muss leider ausfallen!

Fahrplan

Floyd-Hoare-Logik: Idee

- Was wird hier berechnet? $p = n!$
- Wie können wir das beweisen?
- Wir berechnen symbolisch, welche Werte Variablen über den Programmverlauf annehmen.

```
{1 ≤ n}
p= 1;
c= 1;
while (c <= n) {
  p := p * c;
  c := c + 1;
}
{p = n!}
```

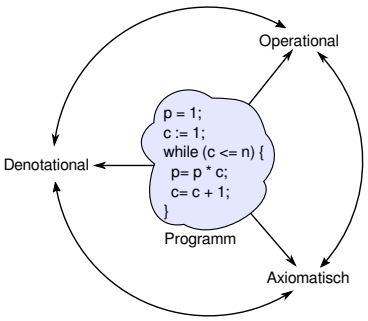
- Um Aussagen über ein Program zu beweisen, benötigen wir einen Formalismus (eine Logik), die es erlaubt, Zusicherungen über Werte von Variablen zu bestimmten Ausführungszeitpunkten (im Programm) aufzuschreiben und zu beweisen.
- Dazu müssen wir auch die Bedeutung (Semantik) des Programmes definieren — die Frage "Was tut das Programm" mathematisch exakt beantworten.

Semantik von Programmiersprachen

Drei wesentliche Möglichkeiten:

- Operationale Semantik beschreibt die Bedeutung eines Programmes, indem die Ausführung auf einer abstrakten Maschine beschrieben wird.
- Denotationale Semantik bildet jedes Programm auf ein mathematisches Objekt (meist ein partielle Funktion zwischen Systemzuständen) ab.
- Axiomatische Semantik beschreibt die Bedeutung eines Programmes durch Beweisregeln, mit welchem sich gültige Eigenschaften herleiten lassen. Das prominenteste Beispiel hierzu ist die Floyd-Hoare-Logik.

Drei Semantiken — Eine Sicht



- Jede Semantik ist eine Sicht auf das Program.
- Diese Semantiken sollten alle äquivalent sein. Wir müssen sie also in Beziehung setzen, und zeigen dass sie die gleiche Sicht ergeben.
- Für die axiomatische Semantik (die Floyd-Hoare-Logik) ist das die Frage der Korrektheit der Regeln.

Floyd-Hoare-Logik

- Grundbaustein der Floyd-Hoare-Logik sind Zusicherungen der Form $\{P\}\, c\, \{Q\}$ (Floyd-Hoare-Tripel), wobei P die Vorbedingung ist, c das Programm, und Q die Nachbedingung.
- Die Logik hat sowohl logische Variablen (zustandsfrei), und Programmvariablen (deren Wert sich über die Programmausführung ändert).
- Die Floyd-Hoare-Logik hat eine wesentliches Prinzip and einen Trick.
- Das Prinzip ist die Abstraktion vom Programmzustand durch eine logische Sprache; insbesondere wird die Zuweisung durch Substitution modelliert.
- Der Trick behandelt Schleifen: Iteration im Programm entspricht Rekursion in der Logik. Ein Beweis ist daher induktiv, und benötigt eine Induktionsannahme — eine Invariante.

Unsere Programmiersprache

Wir betrachten einen Ausschnitt der Programmiersprache C (C0).

Ausbaustufe 1 kennt folgende Konstrukte:

- Typen: **int**;
- Ausdrücke: Variablen, Literale (für ganze Zahlen), arithmetische Operatoren (für ganze Zahlen), Relationen (==, !=, <=, ...), boolsche Operatoren (&&, ||);
- Anweisungen:
  - Fallunterscheidung (**if**...**else**...), Iteration (**while**), Zuweisung, Blöcke;
  - Sequenzierung und leere Anweisung sind implizit

C0: Ausdrücke und Anweisungen

Aexp $a ::= \mathbf{N} \mid \mathbf{Loc} \mid a_1 + a_2 \mid a_1 - a_2 \mid a_1 * a_2 \mid a_1/a_2$
Bexp $b ::= \mathbf{0} \mid \mathbf{1} \mid a_1 == a_2 \mid a_1 != a_2$
$\mid a_1 <= a_2 \mid !b \mid b_1 \,\&\&\, b2 \mid b_1 \,||\, b_2$
Exp $e := \mathbf{Aexp} \mid \mathbf{Bexp}$
Stmt $c ::= \mathbf{Loc} = \mathbf{Exp};$
$\mid$ **if** (b) c_1 **else** c_2
$\mid$ **while** (b) c
$\mid \{c^*\}$

Floyd-Hoare-Tripel

Partielle Korrektheit ($\models \{P\}\, c\, \{Q\}$)
c ist partiell korrekt, wenn für alle Zustände σ, die P erfüllen:
wenn die Ausführung von c mit σ in σ' terminiert, **dann** erfüllt σ' Q

Totale Korrektheit ($\models [P]\, c\, [Q]$)
c ist total korrekt, wenn für alle Zustande σ, die P erfüllen:
die Ausführung von c mit σ in σ' terminiert, und σ' erfüllt Q.

- Folgendes gilt: $\models \{\mathbf{true}\}\ \mathbf{while}(1)\{\ \}\ \{\mathbf{true}\}$
- Folgendes gilt nicht: $\models [\mathbf{true}]\ \mathbf{while}(1)\{\ \}\ [\mathbf{true}]$

Regeln der Floyd-Hoare-Logik

- Die Floyd-Hoare-Logik erlaubt es, Zusicherungen der Form $\vdash \{P\}\, c\, \{Q\}$ syntaktisch herzuleiten.
- Der Kalkül der Logik besteht aus sechs Regeln der Form

$$\frac{\vdash \{P_1\}\, c_1\, \{Q_1\} \ldots \vdash \{P_n\}\, c_n\, \{Q_n\}}{\vdash \{P\}\, c\, \{Q\}}$$

- Für jedes Konstrukt der Programmiersprache gibt es eine Regel.

Regeln der Floyd-Hoare-Logik: Zuweisung

$$\frac{}{\vdash \{P[e/x]\}\, x = e\, \{P\}}$$

- Eine Zuweisung x=e ändert den Zustand so dass an der Stelle x jetzt der Wert von e steht. Damit nachher das Prädikat P gilt, muss also vorher das Prädikat gelten, wenn wir x durch e ersetzen.
- Es ist völlig normal (aber dennoch falsch) zu denken, die Substitution gehöre eigentlich in die Nachbedingung.
- Beispiele:

$\{5 < 10 \iff (x < 10)[5/x]\}$
x = 5
$\{x < 10\}$

$\{x < 9 \iff x + 1 < 10\}$
x = x+ 1
$\{x < 10\}$

Regeln der Floyd-Hoare-Logik: Fallunterscheidung und Sequenzierung

$$\frac{\vdash \{A \,\&\&\, b\}\, c_0\, \{B\} \qquad \vdash \{A \,\&\&\, \neg b\}\, c_1\, \{B\}}{\vdash \{A\}\ \mathbf{if}\ (b)\ c_0\ \mathbf{else}\ c_1\, \{B\}}$$

- In der Vorbedingung des **if**-Zweiges gilt die Bedingung b, und im **else**-Zweig gilt die Negation $\neg b$.
- Beide Zweige müssem mit derselben Nachbedingung enden.

$$\frac{\vdash \{A\}\, c\, \{B\} \qquad \vdash \{B\}\, \{c_s\}\, \{C\}}{\vdash \{A\}\, \{c\ c_s\}\, \{C\}}$$

- Hier wird ein Zwischenzustand B benötigt.

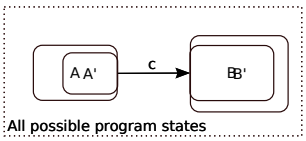
Regeln der Floyd-Hoare-Logik: Iteration

$$\frac{\vdash \{A \wedge b\}\, c\, \{A\}}{\vdash \{A\}\ \mathbf{while}(b)\ c\, \{A \wedge \neg b\}}$$

- Iteration korrespondiert zu Induktion.
- Bei (natürlicher) Induktion zeigen wir, dass die gleiche Eigenschaft P für 0 gilt, und dass wenn sie für $P(n)$ gilt, daraus folgt, dass sie für $P(n+1)$ gilt.
- Analog dazu benötigen wir hier eine Invariante A, die sowohl vor als auch nach dem Schleifenrumpf gilt.
- In der Vorbedingung des Schleifenrumpfes können wir die Schleifenbedingung b annehmen.
- Die Vorbedingung der Schleife ist die Invariante A, und die Nachbedingung der Schleife ist A und die Negation der Schleifenbedingung b.

Regeln der Floyd-Hoare-Logik: Weakening

$$\frac{A' \Longrightarrow A \qquad \vdash \{A\}\, c\, \{B\} \qquad B \Longrightarrow B'}{\vdash \{A'\}\, c\, \{B'\}}$$



- $\models \{A\}\, c\, \{B\}$: Ausführung von c startet in Zustand, in dem A gilt, und endet (ggf) in Zustand, in dem B gilt.
- Zustandsprädikate beschreiben Mengen von Zuständen: $P \subseteq Q$ gdw. $P \Longrightarrow Q$.
- Wir können A zu A' einschränken ($A' \subseteq A$ oder $A' \Longrightarrow A$), oder B zu B' vergrößern ($B \subseteq B'$ oder $B \Longrightarrow B'$), und erhalten $\models \{A'\}\, c\, \{B'\}$.

Überblick: die Regeln des Floyd-Hoare-Kalküls

$$\frac{}{\vdash \{P[e/x]\}\, x = e\, \{P\}}$$

$$\frac{}{\vdash \{A\}\, \{\}\, \{A\}} \qquad \frac{\vdash \{A\}\, c\, \{B\} \qquad \vdash \{B\}\, \{c_s\}\, \{C\}}{\vdash \{A\}\, \{c\ c_s\}\, \{C\}}$$

$$\frac{\vdash \{A \wedge b\}\, c_0\, \{B\} \qquad \vdash \{A \wedge \neg b\}\, c_1\, \{B\}}{\vdash \{A\}\ \mathbf{if}\ (b)\ c_0\ \mathbf{else}\ c_1\, \{B\}}$$

$$\frac{\vdash \{A \wedge b\}\, c\, \{A\}}{\vdash \{A\}\ \mathbf{while}(b)\ c\, \{A \wedge \neg b\}} \qquad \frac{A' \Longrightarrow A \qquad \vdash \{A\}\, c\, \{B\} \qquad B \Longrightarrow B'}{\vdash \{A'\}\, c\, \{B'\}}$$

Eigenschaften der Floyd-Hoare-Logik

Korrektheit

Wenn $\vdash \{P\}\, c\, \{Q\}$, dann $\models \{P\}\, c\, \{Q\}$

- Wenn wir eine Korrektheitsaussage herleiten können, dann gilt sie auch.
- Wird gezeigt, indem wir $\models \{P\}\, c\, \{Q\}$ durch die anderen Semantiken definieren, und zeigen, dass alle Regeln diese Gültigkeit erhalten.

Relative Vollständigkeit

Wenn $\models \{P\}\, c\, \{Q\}$, dann $\vdash \{P\}\, c\, \{Q\}$ (bis auf Weakening)

- Wenn eine Korrektheitsaussage nicht beweisen werden kann (aber sie stimmt), dann liegt das immer daran, dass eine logische Aussage (in einer Anwendung der Weakening-Regelx) nicht bewiesen werden kann.
- Das ist zu erwarten: alle interessanten Logiken sind unvollständig.

Wie wir Floyd-Hoare-Beweise aufschreiben

```
// {P}
// {P1}
x= e;
// {P2}
// {P3}
while (x< n) {
    // {P3 ∧ x < n}
    // {P4}
    z= a;
    // {P3}
    }
// {P3 ∧ ¬(x < n)}
// {Q}
```

- Beispiel zeigt: $\vdash \{P\}\, c\, \{Q\}$
- Programm wird mit gültigen Zusicherungen annotiert.
- Vor einer Zeile steht die Vorbedingung, danach die Nachbedingung.
- Implizite Anwendung der Sequenzenregel.
- Weakening wird notiert durch mehrere Zusicherungen, und muss bewiesen werden.
  - Im Beispiel: $P \Longrightarrow P_1$, $P_2 \Longrightarrow P_3$, $P_3 \wedge x < n \Longrightarrow P_4$, $P_3 \wedge \neg(x < n) \Longrightarrow Q$.

Warum Verifikation?

Hier sind Varianten des Fakultätsbeispiels.
Welche sind korrekt?

```
// {1 ≤ n}
p = 1;
c = 1;
while (c<=n) {
  c = c+1;
  p = p*c;
}
// {p = n!}
```

```
// {1 ≤ n}
p = 1;
c = 1;
while (c<n) {
  c = c+1;
  p = p*c;
}
// {p = n!}
```

```
// {1 ≤ N ∧ n = N}
p = 1;
while (0<n) {
  p = p*n;
  n = n-1;
}
// {p = N!}
```


Eine Handvoll Beispiele

```
// {y = Y ∧ y ≥ 0}
x= 1;
while (y != 0) {
 y= y-1;
 x= 2*x;
}
// {x = 2^Y}
```

```
// {a ≥ 0 ∧ b ≥ 0}
r= a;
q= 0;
while (b <= r) {
  r= r-b;
  q= q+1;
  }
// {a = b * q + r ∧ 0 ≥ r ∧ r < b}
```

```
// {0 ≤ a}
t= 1;
s= 1;
i= 0;
while (s <= a) {
  t= t+ 2;
  s= s+ t;
  i= i+ 1;
  }
// {i^2 ≤ a ∧ a < (i+1)^2}
```


Zusammenfassung

- Floyd-Hoare-Logik zusammengefasst:
  - Die Logik abstrahiert über konkrete Systemzustände durch Zusicherungen (Hoare-Tripel $\models \{P\}\, c\, \{Q\}$).
  - Zusicherungen sind boolsche Ausdrücke, angereichert durch logische Variablen und Programmvariablen.
  - Wir können partielle Korrektheitsaussgen der Form $\models \{P\}\, c\, \{Q\}$ herleiten (oder totale, $\models [P]\, c\, [Q]$).
  - Zuweisungen werden durch Substitution modelliert, d.h. die Menge der gültigen Aussagen ändert sich.
  - Für Iterationen wird eine Invariante benötigt (die nicht hergeleitet werden kann).
- Die Korrektheit hängt sehr davon ab, wie exakt wir die Semantik der Programmiersprache beschreiben können.

Korrekte Software: Grundlagen und Methoden
Vorlesung 3 vom 20.04.17: Operationale Semantik

Serge Autexier, Christoph Lüth

Universität Bremen

Sommersemester 2017

Fahrplan

Zutaten

```
// GGT(A,B)
if (a == 0) r = b;
else {
  while (b != 0) {
   if (a <= b)
      b = b - a;
   else a = a - b;
  }
  r = a;
}
```

- Programme berechnen Werte
- Basierend auf
  - Werte sind Variablen zugewiesen
  - Evaluation von Ausdrücken
- Folgt dem Programmablauf

Unsere Programmiersprache

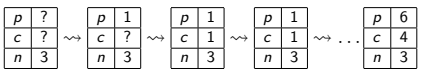
Wir betrachten einen Ausschnitt der Programmiersprache C (C0).

Ausbaustufe 1 kennt folgende Konstrukte:

- Typen: **int**;
- Ausdrücke: Variablen, Literale (für ganze Zahlen), arithmetische Operatoren (für ganze Zahlen), Relationen (==, !=, <=, ...), boolsche Operatoren (&&, ||);
- Anweisungen:
  - Fallunterscheidung (**if**...**else**...), Iteration (**while**), Zuweisung, Blöcke;
  - Sequenzierung und leere Anweisung sind implizit

Semantik von C0

- Die (operationale) Semantik einer imperativen Sprache wie C0 ist ein Zustandsübergang: das System hat einen impliziten Zustand, der durch Zuweisung von Werten an Adressen geändert werden kann.
- Konkretes Beispiel: $n = 3$




```
p = 1;
c = 1;
while(c <= n){
  p = p * c;
  c = c + 1; }
```

Systemzustände

- Ausdrücke werten zu Werten **V** (hier ganze Zahlen) aus.
- Adressen **Loc** sind hier Programmvariablen (Namen)
- Ein Systemzustand bildet Adressen auf Werte ab: $\Sigma = \mathbf{Loc} \rightharpoonup \mathbf{V}$
- Ein Programm bildet einen Anfangszustand möglicherweise auf einen Endzustand ab (wenn es terminiert).
- Zusicherungen sind Prädikate über dem Systemzustand.

C0: Ausdrücke und Anweisungen

Aexp $a ::= \mathbf{N} \mid \mathbf{Loc} \mid a_1 + a_2 \mid a_1 - a_2 \mid a_1 * a_2 \mid a_1/a_2$

Bexp $b ::= \mathbf{0} \mid \mathbf{1} \mid a_1 == a_2 \mid a_1 != a_2$
$\mid a_1 <= a_2 \mid !b \mid b_1 \,\&\&\, b2 \mid b_1 \,||\, b_2$

Exp $e := \mathbf{Aexp} \mid \mathbf{Bexp}$

Stmt $c ::=$ $\mathbf{Loc} = \mathbf{Exp};$
$\mid$ **if** (b) c_1 **else** c_2
$\mid$ **while** (b) c
$\mid \{c^*\}$

Eine Handvoll Beispiele

```
// {y = Y ∧ y ≥ 0}
x = 1;
while (y != 0) {
 y = y-1;
 x = 2*x;
}
// {x = 2^Y}
```

```
// {a ≥ 0 ∧ b ≥ 0}
r = b;
q = 0;
while (b <= r) {
   r = r-y;
   q = q+1;
   }
// {a = b*q + r ∧ r < b}
```

```
p = 1;
c = 1;
while (c<=n) {
   c = c+1;
   p = p*c;
}
// {p = n!}
```

```
// {0 ≤ a}
t = 1;
s = 1;
i = 0;
while (s <= a) {
   t = t + 2;
   s = s + t;
   i = i + 1;
   }
// {i^2 ≤ a ∧ a < (i+1)^2}
```


Operationale Semantik: Arithmetische Ausdrücke

Ein arithmetischer Ausdruck a wertet unter gegebenen Zustand σ zu einer ganzen Zahl n (Wert) aus oder zu einem Fehler $\perp$.

- **Aexp** $a ::= \mathbf{N} \mid \mathbf{Loc} \mid a_1 + a_2 \mid a_1 - a_2 \mid a_1 * a_2 \mid a_1 \,/\, a_2$
- Zustände bilden Adressen/Programmvariablen auf **Werte** ab (σ)

$$\langle a, \sigma\rangle \rightarrow_{Aexp} n|\perp$$

Regeln:

$$\frac{}{\langle n, \sigma\rangle \rightarrow_{Aexp} n}$$

$$\frac{X \in \mathbf{Loc}, X \in Dom(\sigma), \sigma(X) = v}{\langle X, \sigma\rangle \rightarrow_{Aexp} v} \qquad \frac{X \in \mathbf{Loc}, X \notin Dom(\sigma)}{\langle X, \sigma\rangle \rightarrow_{Aexp} \perp}$$

Operationale Semantik: Arithmetische Ausdrücke

- **Aexp** $a ::= \mathbf{N} \mid \mathbf{Loc} \mid a_1 + a_2 \mid a_1 - a_2 \mid a_1 * a_2 \mid a_1 \ / \ a_2$

$$\langle a, \sigma\rangle \to_{Aexp} n | \bot$$

$$\frac{\langle a_1, \sigma\rangle \to_{Aexp} n_1 \quad \langle a_2, \sigma\rangle \to_{Aexp} n_2 \quad n_i \in \mathbf{N}, n \text{ Summe } n_1 \text{ und } n_2}{\langle a_1 + a_2, \sigma\rangle \to_{Aexp} n}$$

$$\frac{\langle a_1, \sigma\rangle \to_{Aexp} n_1 \quad \langle a_2, \sigma\rangle \to_{Aexp} n_2 \quad \text{falls } n_1 = \bot \text{ oder } n_2 = \bot}{\langle a_1 + a_2, \sigma\rangle \to_{Aexp} \bot}$$

$$\frac{\langle a_1, \sigma\rangle \to_{Aexp} n_1 \quad \langle a_2, \sigma\rangle \to_{Aexp} n_2 \quad n_i \in \mathbf{N}, n \text{ Diff. } n_1 \text{ und } n_2}{\langle a_1 - a_2, \sigma\rangle \to_{Aexp} n}$$

$$\frac{\langle a_1, \sigma\rangle \to_{Aexp} n_1 \quad \langle a_2, \sigma\rangle \to_{Aexp} n_2 \quad \text{falls } n_1 = \bot \text{ oder } n_2 = \bot}{\langle a_1 - a_2, \sigma\rangle \to_{Aexp} \bot}$$

Operationale Semantik: Arithmetische Ausdrücke

- **Aexp** $a ::= \mathbf{N} \mid \mathbf{Loc} \mid a_1 + a_2 \mid a_1 - a_2 \mid a_1 * a_2 \mid a_1 \ / \ a_2$

$$\langle a, \sigma\rangle \to_{Aexp} n | \bot$$

$$\frac{\langle a_1, \sigma\rangle \to_{Aexp} n_1 \quad \langle a_2, \sigma\rangle \to_{Aexp} n_2 \quad n_i \in \mathbf{N}, n \text{ Produkt } n_1 \text{ und } n_2}{\langle a_1 * a_2, \sigma\rangle \to_{Aexp} n}$$

$$\frac{\langle a_1, \sigma\rangle \to_{Aexp} n_1 \quad \langle a_2, \sigma\rangle \to_{Aexp} n_2 \quad \text{falls } n_1 = \bot \text{ oder } n_2 = \bot}{\langle a_1 * a_2, \sigma\rangle \to_{Aexp} \bot}$$

$$\frac{\langle a_1, \sigma\rangle \to_{Aexp} n_1 \quad \langle a_2, \sigma\rangle \to_{Aexp} n_2 \quad n_i \in \mathbf{N}, n_2 \neq 0, n \text{ Quotient } n_1, n_2}{\langle a_1 / a_2, \sigma\rangle \to_{Aexp} n}$$

$$\frac{\langle a_1, \sigma\rangle \to_{Aexp} n_1 \quad \langle a_2, \sigma\rangle \to_{Aexp} n_2 \quad \text{falls } n_1 = \bot, n_2 = \bot \text{ oder } n_2 = 0}{\langle a_1 / a_2, \sigma\rangle \to_{Aexp} \bot}$$

Beispiel-Ableitungen

Sei $\sigma(X) = 6, \sigma(Y) = 5$.

$$\frac{\dfrac{\langle X, \sigma\rangle \to_{Aexp} 6 \quad \langle Y, \sigma\rangle \to_{Aexp} 5}{\langle X + Y, \sigma\rangle \to_{Aexp} 11} \quad \dfrac{\langle X, \sigma\rangle \to_{Aexp} 6 \quad \langle Y, \sigma\rangle \to_{Aexp} 5}{\langle X - Y, \sigma\rangle \to_{Aexp} 1}}{\langle (X + Y) * (X - Y), \sigma\rangle \to_{Aexp} 11}$$

$$\frac{\dfrac{\langle X, \sigma\rangle \to_{Aexp} 6 \quad \langle X, \sigma\rangle \to_{Aexp} 6}{\langle X * X, \sigma\rangle \to_{Aexp} 36} \quad \dfrac{\langle Y, \sigma\rangle \to_{Aexp} 5 \quad \langle Y, \sigma\rangle \to_{Aexp} 5}{\langle Y * Y, \sigma\rangle \to_{Aexp} 25}}{\langle (X * X) - (Y * Y), \sigma\rangle \to_{Aexp} 11}$$

Operationale Semantik: Boolesche Ausdrücke

- **Bexp** $b ::= 0 \mid 1 \mid a_1 == a_2 \mid a_1 \ <= \ a_2 \mid !b \mid b_1 \ \&\& \ b2 \mid b_1 \ || \ b_2$

Regeln:

$$\langle b, \sigma\rangle \to_{Bexp} 1 | 0 | \bot$$

$$\frac{}{\langle \mathbf{1}, \sigma\rangle \to_{Bexp} \mathbf{1}} \qquad \frac{}{\langle \mathbf{0}, \sigma\rangle \to_{Bexp} \mathbf{0}}$$

$$\frac{\langle a_1, \sigma\rangle \to_{Aexp} n_1 \quad \langle a_2, \sigma\rangle \to_{Aexp} n_2 \quad n_i \neq \bot, n_1 \text{ und } n_2 \text{ gleich}}{\langle a_1 == a_2, \sigma\rangle \to_{Bexp} 1}$$

$$\frac{\langle a_1, \sigma\rangle \to_{Aexp} n_1 \quad \langle a_2, \sigma\rangle \to_{Aexp} n_2 \quad n_i \neq \bot, n_1 \text{ und } n_2 \text{ ungleich}}{\langle a_1 == a_2, \sigma\rangle \to_{Bexp} 0}$$

$$\frac{\langle a_1, \sigma\rangle \to_{Aexp} n_1 \quad \langle a_2, \sigma\rangle \to_{Aexp} n_2 \quad n_1 = \bot \text{ or } n_2 = \bot}{\langle a_1 == a_2, \sigma\rangle \to_{Bexp} \bot}$$

Operationale Semantik: Boolesche Ausdrücke

- **Bexp** $b ::= 0 \mid 1 \mid a_1 == a_2 \mid a_1 \ <= \ a_2 \mid !b \mid b_1 \ \&\& \ b2 \mid b_1 \ || \ b_2$

Regeln:

$$\langle b, \sigma\rangle \to_{Bexp} 1 | 0 | \bot$$

$$\frac{\langle b, \sigma\rangle \to_{Bexp} 1}{\langle !b, \sigma\rangle \to_{Bexp} 0} \qquad \frac{\langle b, \sigma\rangle \to_{Bexp} 0}{\langle !b, \sigma\rangle \to_{Bexp} 1} \qquad \frac{\langle b, \sigma\rangle \to_{Bexp} \bot}{\langle !b, \sigma\rangle \to_{Bexp} \bot}$$

$$\frac{\langle b_1, \sigma\rangle \to_{Bexp} t_1 \quad \langle b_2, \sigma\rangle \to_{Bexp} t_2}{\langle b_1 \ \&\& \ b_2, \sigma\rangle \to_{Bexp} t}$$

wobei $t = 1$ wenn $t_1 = t_2 = 1$;
$t = 0$ wenn $t_1 = 0$ oder $(t_1 = 1$ und $t_2 = 0)$;
$t = \bot$ sonst

Operationale Semantik: Boolesche Ausdrücke

- **Bexp** $b ::= 0 \mid 1 \mid a_1 == a_2 \mid a_1 \ <= \ a_2 \mid !b \mid b_1 \ \&\& \ b2 \mid b_1 \ || \ b_2$

Regeln:

$$\langle b, \sigma\rangle \to_{Bexp} 1 | 0 | \bot$$

$$\frac{\langle b_1, \sigma\rangle \to_{Bexp} t_1 \quad \langle b_2, \sigma\rangle \to_{Bexp} t_2}{\langle b_1 \ || \ b_2, \sigma\rangle \to_{Bexp} t}$$

wobei $t = 0$ wenn $t_1 = t_2 = 0$;
$t = 1$ wenn $t_1 = 1$ oder $(t_1 = 0$ und $t_2 = 1)$;
$t = \bot$ sonst

Operationale Semantik: Anweisungen

- **Stmt** $c ::= \mathbf{Loc} = \mathbf{Exp};$ | $\{c^*\}$ | **if** (b) c_1 **else** c_2 | **while** (b) c

Beispiel:

$$\langle c, \sigma\rangle \to_{Stmt} \sigma' | \bot$$

$$\langle X = 5, \sigma\rangle \to_{Stmt} \sigma'$$

wobei $\sigma'(X) = 5$ und $\sigma'(Y) = \sigma(Y)$ für alle $Y \neq X$

Operationale Semantik: Anweisungen

- **Stmt** $c ::= \mathbf{Loc} = \mathbf{Exp};$ | $\{c^*\}$ | **if** (b) c_1 **else** c_2 | **while** (b) c

Regeln:

Definiere :

$$\sigma[m/X](Y) := \begin{cases} m & \text{if } X = Y \\ \sigma(Y) & \text{sonst} \end{cases}$$

$$\langle X = 5, \sigma\rangle \to_{Stmt} \sigma[5/X]$$

Es gilt:

$$\forall \sigma, n, m, \forall X, Y \ . \ X \neq Y \Rightarrow \sigma[n/X][m/Y] = \sigma[m/Y][n/X]$$
$$\forall \sigma, n, m, \forall X \ . \ \sigma[n/X][m/X] = \sigma[m/X]$$

Operationale Semantik: Anweisungen

- **Stmt** c ::= **Loc** = **Exp**; | $\{c^*\}$ | **if** (b) c_1 **else** c_2 | **while** (b) c

Regeln:

$$\langle \{\}, \sigma \rangle \to_{Stmt} \sigma$$

$$\frac{\langle a, \sigma \rangle \to_{Aexp} n \in \mathbf{N}}{\langle X = a, \sigma \rangle \to_{Stmt} \sigma[n/X]} \qquad \frac{\langle a, \sigma \rangle \to_{Aexp} \bot}{\langle X = a, \sigma \rangle \to_{Stmt} \bot}$$

$$\frac{\langle c, \sigma \rangle \to_{Stmt} \sigma' \neq \bot \qquad \langle \{c_s\}, \sigma' \rangle \to_{Stmt} \sigma'' \neq \bot}{\langle \{c\ c_s\}, \sigma \rangle \to_{Stmt} \sigma''}$$

$$\frac{\langle c, \sigma \rangle \to_{Stmt} \bot}{\langle \{c\ c_s\}, \sigma \rangle \to_{Stmt} \bot}$$

$$\frac{\langle c, \sigma \rangle \to_{Stmt} \sigma' \neq \bot \qquad \langle \{c_s\}, \sigma' \rangle \to_{Stmt} \bot}{\langle \{c\ c_s\}, \sigma \rangle \to_{Stmt} \bot}$$

Operationale Semantik: Anweisungen

- **Stmt** c ::= **Loc** = **Exp**; | $\{c^*\}$ | **if** (b) c_1 **else** c_2 | **while** (b) c

Regeln:

$$\langle \{\}, \sigma \rangle \to_{Stmt} \sigma$$

$$\frac{\langle b, \sigma \rangle \to_{Bexp} 1 \qquad \langle c_1, \sigma \rangle \to_{Stmt} \sigma'}{\langle \textbf{if}\ (\ b\ )\ c_1\ \textbf{else}\ c_2, \sigma \rangle \to_{Stmt} \sigma'}$$

$$\frac{\langle b, \sigma \rangle \to_{Bexp} 0 \qquad \langle c_2, \sigma \rangle \to_{Stmt} \sigma'}{\langle \textbf{if}\ (\ b\ )\ c_1\ \textbf{else}\ c_2, \sigma \rangle \to_{Stmt} \sigma'}$$

$$\frac{\langle b, \sigma \rangle \to_{Bexp} \bot}{\langle \textbf{if}\ (\ b\ )\ c_1\ \textbf{else}\ c_2, \sigma \rangle \to_{Stmt} \bot}$$

Operationale Semantik: Anweisungen

- **Stmt** c ::= **Loc** = **Exp**; | $\{c^*\}$ | **if** (b) c_1 **else** c_2 | **while** (b) c

Regeln:

$$\langle \{\}, \sigma \rangle \to_{Stmt} \sigma$$

$$\frac{\langle b, \sigma \rangle \to_{Bexp} 0}{\langle \textbf{while}\ (\ b\ )\ c, \sigma \rangle \to_{Stmt} \sigma}$$

$$\frac{\langle b, \sigma \rangle \to_{Bexp} 1 \qquad \langle c, \sigma \rangle \to_{Stmt} \sigma' \qquad \langle \textbf{while}\ (\ b\ )\ c, \sigma' \rangle \to_{Stmt} \sigma''}{\langle \textbf{while}\ (\ b\ )\ c, \sigma \rangle \to_{Stmt} \sigma''}$$

$$\frac{\langle b, \sigma \rangle \to_{Bexp} 1 \qquad \langle c, \sigma \rangle \to_{Stmt} \bot}{\langle \textbf{while}\ (\ b\ )\ c, \sigma \rangle \to_{Stmt} \bot} \qquad \frac{\langle b, \sigma \rangle \to_{Bexp} \bot}{\langle \textbf{while}\ (\ b\ )\ c, \sigma \rangle \to_{Stmt} \bot}$$

Beispiel

```
x = 1;
while (y != 0) {
 y = y - 1;
 x = 2 * x;
}
// x = 2^y
```

$\sigma(y) = 3$

Äquivalenz arithmetischer Ausdrücke

Gegeben zwei Aexp a_1 and a_2

- Sind sie gleich?

$$a_1 \sim_{Aexp} a_2 \text{ gdw } \forall \sigma, n. \langle a_1, \sigma \rangle \to_{Aexp} n \Leftrightarrow \langle a_2, \sigma \rangle \to_{Aexp} n$$

(X*X) + 2*X*Y + (Y*Y) und (X+Y) * (X+Y)

- Wann sind sie gleich?

$$\exists \sigma, n. \langle a_1, \sigma \rangle \to_{Aexp} n \Leftrightarrow \langle a_2, \sigma \rangle \to_{Aexp} n$$

X*X und 9*X+22
X*X und X*X+1

Äquivalenz Boolscher Ausdrücke

Gegeben zwei Bexp-Ausdrücke b_1 and b_2

- Sind sie gleich?

$$b_1 \sim_{Bexp} b_2 \text{ iff } \forall \sigma, b. \langle b_1, \sigma \rangle \to_{Bexp} b \Leftrightarrow \langle b_2, \sigma \rangle \to_{Bexp} b$$

A || (A && B) und A

Beweisen

Zwei Programme c_0, c_1 sind äquivalent gdw. sie die gleichen Zustandsveränderungen bewirken. Formal definieren wir

Definition

$$c_0 \sim c_1 \text{ iff } \forall \sigma, \sigma'. \langle c_0, \sigma \rangle \to_{Stmt} \sigma' \Leftrightarrow \langle c_1, \sigma \rangle \to_{Stmt} \sigma'$$

Ein einfaches Beispiel:

Lemma

Sei $w \equiv$ ***while*** (b) c *mit* $b \in$ ***Bexp***, $c \in$ ***Stmt***.
Dann gilt: $w \sim$ ***if*** (b) $\{c; w\}$ ***else*** $\{\}$

Beweis an der Tafel

Zusammenfassung

- Operationale Semantik als ein Mittel für Beschreibung der Semantik
- Auswertungsregeln arbeiten entlang der syntaktischen Struktur
- Werten Ausdrücke zu Werten aus und Programme zu Zuständen (zu gegebenen Zustand)
- Fragen zu Programmen: Gleichheit

Korrekte Software: Grundlagen und Methoden
Vorlesung 4 vom 24.04.17: Denotationale Semantik

Serge Autexier, Christoph Lüth

Universität Bremen

Sommersemester 2017

Fahrplan

- Einführung
- Die Floyd-Hoare-Logik
- Operationale Semantik
- Denotationale Semantik
- Äquivalenz der Operationalen und Denotationalen Semantik
- Korrektheit des Hoare-Kalküls
- Vorwärts und Rückwärts mit Floyd und Hoare
- Funktionen und Prozeduren
- Referenzen und Speichermodelle
- Verifikationsbedingungen Revisited
- Vorwärtsrechnung Revisited
- Programmsicherheit und Frame Conditions
- Ausblick und Rückblick

Überblick

- Kleinster Fixpunkt
- Denotationale Semantik für C0

Fixpunkt

- Sei $f : A \to A$ eine Funktion. Ein Fixpunkt von f ist ein $a \in A$, so dass $f(a) = a$.
- Beispiele
  - Fixpunkte von $f(x) = \sqrt{x}$ sind 0 und 1; ebenfalls für $f(x) = x^2$.
  - Für die Sortierfunktion ist

Regeln und Regelinstanzen

Definition

Sei R eine Menge von Regeln $\frac{x_1 \ldots x_n}{y}$, $n \geq 0$.
Die Anwendung einer Regel auf spezifische $a_1 \ldots a_n$ ist eine Regelinstanz

- Betrachte folgende Regelmenge R

$$\frac{-}{2^2} \qquad \frac{-}{2^3} \qquad \frac{n \quad m}{n \cdot m}$$

- Regelinstanzen sind

$$\frac{-}{4} \qquad \frac{-}{8} \qquad \frac{4 \quad 8}{32} \qquad \frac{4 \quad 4}{16}$$

$$\frac{16 \quad 32}{512} \qquad \frac{3 \quad 5}{15} \qquad \ldots$$

Induktive Definierte Mengen

Definition

Seit R eine Menge von Regelinstanzen und B eine Menge. Dann definieren wir

$$\hat{R}(B) = \{y \mid \exists x_1, \ldots, x_k \subseteq B.\ \frac{x_1, \ldots, x_k}{y} \in R\} \text{ und}$$

$$\hat{R}^0(B) = B \text{ und } \hat{R}^{i+1}(B) = \hat{R}(\hat{R}^i(B))$$

Beispiel

- Betrachte folgende Regelmenge R

$$\frac{-}{2^2} \qquad \frac{-}{2^3} \qquad \frac{n \quad m}{n \cdot m}$$

- Was sind

$$\hat{R}^1(\emptyset) = \hat{R}(\emptyset) = \{4, 8\}$$
$$\hat{R}^2(\emptyset) = ?$$
$$\hat{R}^3(\emptyset) = ?$$
$$\hat{R}^{i+1}(\emptyset) = ?$$

Induktive Definierte Mengen

Definition

Seit R eine Menge von Regelinstanzen und B eine Menge. Dann definieren wir

$$\hat{R}(B) = \{y \mid \exists x_1, \ldots, x_k \subseteq B.\ \frac{x_1, \ldots, x_k}{y} \in R\} \text{ und}$$

$$\hat{R}^0(B) = B \text{ und } \hat{R}^{i+1}(B) = \hat{R}(\hat{R}^i(B))$$

Definition (Abgeschlossen und Monoton)

- Eine Menge S is abgeschlossen unter R (R-abgeschlossen) gdw. $\hat{R}(S) \subseteq S$
- Eine Operation f ist monoton gdw.

$$\forall A, B. A \subseteq B \Rightarrow f(A) \subseteq f(B)$$

Kleinster Fixpunkt Operator

Lemma

Für jede Menge von Regelinstanzen R ist die induzierte Operation $\hat{R}$ monoton.

Lemma

Sei $A_i = \hat{R}^i(\emptyset)$ für alle $i \in \mathbb{N}$ und $A = \bigcup_{i \in \mathbb{N}} A_i$. Dann gilt

(a) *A ist R-abgeschlossen,*

(b) *$\hat{R}(A) = A$, und*

(c) *A ist die kleinste R-abgeschlossene Menge.*

Beweis von Lemma (a).

A ist R-abgeschlossen:

Sei $\frac{x_1,\ldots,x_k}{y} \in R$ und $x_1, \ldots, x_k \subseteq A$.
Da $A = \bigcup_{i \in \mathbb{N}} A_i$ gibt es ein j so dass $x_1, \ldots, x_k \subseteq A_j$.
Also auch:

$$\begin{aligned} y \in \hat{R}(A_j) &= \hat{R}(\hat{R}^j(\emptyset)) \\ &= \hat{R}^{j+1}(\emptyset) \\ &= A_{j+1} \subseteq A. \end{aligned}$$

□

Beweis von Lemma (b): $\hat{R}(A) = A$.

- $\hat{R}(A) \subseteq A$:
  Da A R-abgeschlossen gilt auch $\hat{R}(A) \subseteq A$.
- $A \subseteq \hat{R}(A)$:
  Sei $y \in A$. Dann $\exists n > 0$. $y \in A_n$ und $y \notin \hat{R}(A_{n-1})$.
  Folglich muss es eine Regelinstanz $\frac{x_1,\ldots,x_k}{y} \in R$ geben mit $x_1, \ldots, x_k \subseteq A_{n-1} \subseteq A$.
  Also ist $y \in \hat{R}(A)$.

□

Beweis von Lemma (c).

A ist die kleinste R-abgeschlossene Menge, d.h. für jede R-abgeschlossene Menge B gilt $A \subseteq B$.

Beweis per Induktion über n dass gilt $A_n \subseteq B$:

- Basisfall:
  $A_0 = \emptyset \subseteq B$
- Induktionsschritt:
  Da B R-abgeschlossen ist gilt: $\hat{R}(B) \subseteq B$.
  Induktionsannahme: $A_n \subseteq B$.
  Dann gilt $A_{n+1} = \hat{R}(A_n) \subseteq \hat{R}(B) \subseteq B$ weil $\hat{R}$ monoton und B ist R-abgeschlossen.

□

Kleinster Fixpunkt Operator

Definition

$$fix(\hat{R}) = \bigcup_{n \in N} \hat{R}^n(\emptyset)$$

ist der kleinste Fixpunkt.

Kleinster Fixpunkt

- Betrachte folgende Regelmenge R

$$\frac{-}{2^2} \qquad \frac{-}{2^3} \qquad \frac{n \quad m}{n \cdot m}$$

- Was sind

$$\begin{aligned} \hat{R}^1(\emptyset) = \hat{R}(\emptyset) &= \{4, 8\} \\ \hat{R}^2(\emptyset) &= ? \\ \hat{R}^3(\emptyset) &= ? \\ \hat{R}^{i+1}(\emptyset) &= ? \end{aligned}$$

- Wie sieht $fix(\hat{R})$ aus?

Denotationale Semantik - Motivation

- Operationale Semantik
  Eine Menge von Regeln, die einen Zustand und ein Programm in einen neuen Zustand oder Fehler überführen

$$\langle c, \sigma \rangle \rightarrow_{Stmt} \sigma' | \perp$$

- Denotationale Semantik
  Eine Menge von Regeln, die ein Programm in eine partielle Funktion (Denotat) von Zustand nach Zustand überführen

$$\mathcal{C}[\![c]\!] : \Sigma \rightharpoonup \Sigma$$

Denotationale Semantik - Motivation

Zwei Programme sind äquivalent gdw. sie immer zum selben Zustand (oder Fehler) auswerten

$$c_0 \sim c_1 \text{ iff } (\forall \sigma, \sigma' . \langle c_0, \sigma \rangle \rightarrow_{Stmt} \sigma' \Leftrightarrow \langle c_1, \sigma \rangle \rightarrow_{Stmt} \sigma')$$

oder
Zwei Programme sind äquivalent gdw. sie die selbe partielle Funktion denotieren

$$c_0 \sim c_1 \text{ iff } \{(\sigma, \sigma') | \langle c_0, \sigma \rangle \rightarrow_{Stmt} \sigma'\} = \{(\sigma, \sigma') | \langle c_1, \sigma \rangle \rightarrow_{Stmt} \sigma'\}$$

Denotierte Funktionen

- jeder a :**Aexp** denotiert eine partielle Funktion $\Sigma \rightharpoonup \mathbf{N}$
- jeder b :**Bexp** denotiert eine partielle Funktion $\Sigma \rightharpoonup \mathbf{T}$
- jedes c :**Stmt** denotiert eine partielle Funktion $\Sigma \rightharpoonup \Sigma$

Denotat von **Aexp**

$$\mathcal{A}[\![a]\!] : \mathbf{Aexp} \to (\Sigma \rightharpoonup \mathbf{N})$$

$$\begin{aligned}
\mathcal{A}[\![n]\!] &= \{(\sigma, n) \mid \sigma \in \Sigma\} \\
\mathcal{A}[\![x]\!] &= \{(\sigma, \sigma(x)) \mid \sigma \in \Sigma, x \in \mathit{Dom}(\sigma)\} \\
\mathcal{A}[\![a_0 + a_1]\!] &= \{(\sigma, n_0 + n_1) \mid (\sigma, n_0) \in \mathcal{A}[\![a_0]\!] \wedge (\sigma, n_1) \in \mathcal{A}[\![a_1]\!]\} \\
\mathcal{A}[\![a_0 - a_1]\!] &= \{(\sigma, n_0 - n_1) \mid (\sigma, n_0) \in \mathcal{A}[\![a_0]\!] \wedge (\sigma, n_1) \in \mathcal{A}[\![a_1]\!]\} \\
\mathcal{A}[\![a_0 * a_1]\!] &= \{(\sigma, n_0 * n_1) \mid (\sigma, n_0) \in \mathcal{A}[\![a_0]\!] \wedge (\sigma, n_1) \in \mathcal{A}[\![a_1]\!]\} \\
\mathcal{A}[\![a_0 / a_1]\!] &= \{(\sigma, n_0 / n_1) \mid (\sigma, n_0) \in \mathcal{A}[\![a_0]\!] \wedge (\sigma, n_1) \in \mathcal{A}[\![a_1]\!] \wedge n_1 \neq 0\}
\end{aligned}$$

Denotat von **Bexp**

$$\mathcal{B}[\![a]\!] : \mathbf{Bexp} \to (\Sigma \rightharpoonup \mathbf{T})$$

$$\begin{aligned}
\mathcal{B}[\![1]\!] &= \{(\sigma, 1) \mid \sigma \in \Sigma\} \\
\mathcal{B}[\![0]\!] &= \{(\sigma, 0) \mid \sigma \in \Sigma\} \\
\mathcal{B}[\![a_0 == a_1]\!] &= \{(\sigma, 1) \mid \sigma \in \Sigma, (\sigma, n_0) \in \mathcal{A}[\![a_0]\!](\sigma), \\
&\qquad (\sigma, n_1) \in \mathcal{A}[\![a_1]\!], n_0 = n_1\} \\
&\quad \cup \{(\sigma, 0) \mid \sigma \in \Sigma, (\sigma, n_0) \in \mathcal{A}[\![a_0]\!](\sigma), \\
&\qquad (\sigma, n_1) \in \mathcal{A}[\![a_1]\!], n_0 \neq n_1\} \\
\mathcal{B}[\![a_0 <= a_1]\!] &= \{(\sigma, 1) \mid \sigma \in \Sigma, (\sigma, n_0) \in \mathcal{A}[\![a_0]\!](\sigma), \\
&\qquad (\sigma, n_1) \in \mathcal{A}[\![a_1]\!], n_0 \leq n_1\} \\
&\quad \cup \{(\sigma, 0) \mid \sigma \in \Sigma, (\sigma, n_0) \in \mathcal{A}[\![a_0]\!](\sigma), \\
&\qquad (\sigma, n_1) \in \mathcal{A}[\![a_1]\!], n_0 > n_1\}
\end{aligned}$$

Denotat von **Bexp**

$$\mathcal{B}[\![a]\!] : \mathbf{Bexp} \to (\Sigma \rightharpoonup \mathbf{T})$$

$$\begin{aligned}
\mathcal{B}[\![!b]\!] &= \{(\sigma, 1) \mid \sigma \in \Sigma, (\sigma, 0) \in \mathcal{B}[\![b]\!]\} \\
&\quad \cup \{(\sigma, 0) \mid \sigma \in \Sigma, (\sigma, 1) \in \mathcal{B}[\![b]\!]\} \\
\mathcal{B}[\![b_1 \;\&\&\; b2]\!] &= \{(\sigma, 0) \mid \sigma \in \Sigma, (\sigma, 0) \in \mathcal{B}[\![b_1]\!]\} \\
&\quad \cup \{(\sigma, t_2) \mid \sigma \in \Sigma, (\sigma, 1) \in \mathcal{B}[\![b_1]\!], (\sigma, t_2) \in \mathcal{B}[\![b_2]\!]\} \\
\mathcal{B}[\![b_1 \;||\; b_2]\!] &= \{(\sigma, 1) \mid \sigma \in \Sigma, (\sigma, 1) \in \mathcal{B}[\![b_1]\!]\} \\
&\quad \cup \{(\sigma, t_2) \mid \sigma \in \Sigma, (\sigma, 0) \in \mathcal{B}[\![b_1]\!], (\sigma, t_2) \in \mathcal{B}[\![b_2]\!]\}
\end{aligned}$$

Denotat von **Stmt**

$$\mathcal{C}[\![.]\!] : \mathbf{Stmt} \to (\Sigma \rightharpoonup \Sigma)$$

$$\begin{aligned}
\mathcal{C}[\![x = a]\!] &= \{(\sigma, \sigma[n/x]) \mid \sigma \in \Sigma \wedge (\sigma, n) \in \mathcal{A}[\![a]\!]\} \\
\mathcal{C}[\![\{c\ c_s\}]\!] &= \mathcal{C}[\![c_s]\!] \circ \mathcal{C}[\![c]\!] \quad \text{Komposition von Relationen} \\
\mathcal{C}[\![\{\,\}]\!] &= \mathbf{Id} \qquad \mathbf{Id} := \{(\sigma, \sigma) \mid \sigma \in \Sigma\} \\
\mathcal{C}[\![\textbf{if } (b)\ c_0 \textbf{ else } c_1]\!] &= \{(\sigma, \sigma') \mid (\sigma, 1) \in \mathcal{B}[\![b]\!] \wedge (\sigma, \sigma') \in \mathcal{C}[\![c_0]\!]\} \\
&\quad \cup \{(\sigma, \sigma') \mid (\sigma, 0) \in \mathcal{B}[\![b]\!] \wedge (\sigma, \sigma') \in \mathcal{C}[\![c_1]\!]\}
\end{aligned}$$

Aber was ist

$$\mathcal{C}[\![\textbf{while } (b)\ c]\!] = ??$$

Denotationale Semantik für **while**

Sei $w \equiv$ **while** (b) c (und $\sigma \in \Sigma$). Wir wissen bereits, dass gilt

$$w \sim \textbf{ if } (b) \{c\ w\} \textbf{ else } \{\}$$

$$\begin{aligned}
\mathcal{C}[\![w]\!] &= \{(\sigma, \sigma') \mid (\sigma, 1) \in \mathcal{B}[\![b]\!] \wedge (\sigma, \sigma') \in \mathcal{C}[\![\{c\ w\}]\!]\} \\
&\quad \cup \{(\sigma, \sigma) \mid (\sigma, 0) \in \mathcal{B}[\![b]\!]\} \\
&= \{(\sigma, \sigma') \mid (\sigma, 1) \in \mathcal{B}[\![b]\!] \wedge (\sigma, \sigma') \in \mathcal{C}[\![w]\!] \circ \mathcal{C}[\![c]\!]\} \\
&\quad \cup \{(\sigma, \sigma) \mid (\sigma, 0) \in \mathcal{B}[\![b]\!]\} \\
&= \{(\sigma, \sigma') \mid \exists \sigma''.\ (\sigma, 1) \in \mathcal{B}[\![b]\!] \wedge (\sigma, \sigma'') \in \mathcal{C}[\![c]\!] \wedge (\sigma'', \sigma') \in \mathcal{C}[\![w]\!]\} \\
&\quad \cup \{(\sigma, \sigma) \mid (\sigma, 0) \in \mathcal{B}[\![b]\!]\}
\end{aligned}$$

Denotationale Semantik von **while**

Sei $w \equiv$ **while** (b) c (und $\sigma \in \Sigma$). Wir wissen bereits, dass gilt

$$w = \textbf{ if } (b) \{c\ w\} \textbf{ else } \{\}$$

$$\begin{aligned}
\mathcal{C}[\![w]\!]_0 &= \{(\sigma, \sigma) \mid (\sigma, 0) \in \mathcal{B}[\![b]\!](\sigma)\} \\
\mathcal{C}[\![w]\!]_1 &= \{(\sigma, \sigma') \mid \exists \sigma''.\ (\sigma, 1) \in \mathcal{B}[\![b]\!] \wedge (\sigma, \sigma'') \in \mathcal{C}[\![c]\!] \\
&\qquad \wedge (\sigma'', \sigma') \in \mathcal{C}[\![w]\!]_0\} \\
\mathcal{C}[\![w]\!]_2 &= \{(\sigma, \sigma') \mid \exists \sigma''.\ (\sigma, 1) \in \mathcal{B}[\![b]\!] \wedge (\sigma, \sigma'') \in \mathcal{C}[\![c]\!] \\
&\qquad \wedge (\sigma'', \sigma') \in \mathcal{C}[\![w]\!]_1\} \\
&\vdots \\
\mathcal{C}[\![w]\!]_{i+1} &= \{(\sigma, \sigma') \mid \exists \sigma''.\ (\sigma, 1) \in \mathcal{B}[\![b]\!] \wedge (\sigma, \sigma'') \in \mathcal{C}[\![c]\!] \\
&\qquad \wedge (\sigma'', \sigma') \in \mathcal{C}[\![w]\!]_i\}
\end{aligned}$$

$$\begin{aligned}
\Gamma(\varphi) &= \{(\sigma, \sigma') \mid \exists \sigma''.\ \mathcal{B}[\![b]\!](\sigma) = 1 \wedge (\sigma, \sigma'') \in \mathcal{C}[\![c]\!] \wedge (\sigma'', \sigma') \in \varphi\} \\
&\quad \cup \{(\sigma, \sigma) \mid \mathcal{B}[\![b]\!](\sigma) = 0\}
\end{aligned}$$

Denotationale Semantik von **while**

Sei $w \equiv$ **while** (b) c (und $\sigma \in \Sigma$). Wir wissen bereits, dass gilt

$$w = \textbf{ if } (b) \{c\ w\} \textbf{ else } \{\}$$

$$\begin{aligned}
\Gamma(\psi) &= \{(\sigma, \sigma') \mid \exists \sigma''.\ (\sigma, 1) \in \mathcal{B}[\![b]\!] \wedge (\sigma, \sigma'') \in \mathcal{C}[\![c]\!] \wedge (\sigma'', \sigma') \in \psi\} \\
&\quad \cup \{(\sigma, \sigma) \mid (\sigma, 0) \in \mathcal{B}[\![b]\!]\}
\end{aligned}$$

Γ ist wie $\hat{R}$, wobei R definiert ist wie folgt:

$$\begin{aligned}
R &= \left\{ \frac{(\sigma'', \sigma')}{(\sigma, \sigma')} \mid (\sigma, 1) \in \mathcal{B}[\![b]\!] \wedge (\sigma, \sigma'') \in \mathcal{C}[\![c]\!] \right\} \\
&\quad \cup \left\{ \frac{}{(\sigma, \sigma)} \mid (\sigma, 0) \in \mathcal{B}[\![b]\!] \right\}
\end{aligned}$$

und die Semantik von w ist der Fixpunkt von Γ, d.h. $\mathcal{C}[\![w]\!] = \mathit{fix}(\Gamma)$

Denotation für **Stmt**

$$\mathcal{C}[\![.]\!] : \textbf{Stmt} \to (\Sigma \rightharpoonup \Sigma)$$

$$\begin{array}{rcl}
\mathcal{C}[\![x = a]\!] & = & \{(\sigma, \sigma[n/X]) \mid \sigma \in \Sigma \wedge (\sigma, n) \in \mathcal{A}[\![a]\!]\} \\
\mathcal{C}[\![\{c\ c_s\}]\!] & = & \mathcal{C}[\![c_s]\!] \circ \mathcal{C}[\![c]\!] \quad \text{Komposition von Relationen} \\
\mathcal{C}[\![\{\ \}]\!] & = & \textbf{Id} \qquad\qquad \textbf{Id} := \{(\sigma, \sigma) | \sigma \in \Sigma\} \\
\mathcal{C}[\![\textbf{if}\ (b)\ c_0\ \textbf{else}\ c_1]\!] & = & \{(\sigma, \sigma') \mid (\sigma, 1) \in \mathcal{B}[\![b]\!] \wedge (\sigma, \sigma') \in \mathcal{C}[\![c_0]\!]\} \\
 & & \cup \{(\sigma, \sigma') \mid (\sigma, 0) \in \mathcal{B}[\![b]\!] \wedge (\sigma, \sigma') \in \mathcal{C}[\![c_1]\!]\} \\
\mathcal{C}[\![\textbf{while}\ (b)\ c]\!] & = & \mathit{fix}(\Gamma)
\end{array}$$

mit

$$\begin{array}{rcl}
\Gamma(\psi) & = & \{(\sigma, \sigma') \mid (\sigma, 1) \in \mathcal{B}[\![b]\!] \wedge (\sigma, \sigma') \in \psi \circ \mathcal{C}[\![c]\!]\} \\
 & & \cup \{(\sigma, \sigma) \mid (\sigma, 0) \in \mathcal{B}[\![b]\!]\}
\end{array}$$

Weitere Intuition zur Fixpunkt Konstruktion

- Sei w ≡ **while** (b) c
- Zur Erinnerung: Wir haben begonnen mit w ∼ **if** (b) { c w } **else** {}
- Dann müsste auch gelten

$$\mathcal{C}[\![w]\!] \stackrel{!}{=} \mathcal{C}[\![\textbf{if}\ (b)\ \{\ c\ w\ \}\ \textbf{else}\ \{\}]\!]$$

- Beweis an der Tafel

Fahrplan

- Einführung
- Die Floyd-Hoare-Logik
- Operationale Semantik
- Denotationale Semantik
- Äquivalenz der Operationalen und Denotationalen Semantik
- Korrektheit des Hoare-Kalküls
- Vorwärts und Rückwärts mit Floyd und Hoare
- Funktionen und Prozeduren
- Referenzen und Speichermodelle
- Verifikationsbedingungen Revisited
- Vorwärtsrechnung Revisited
- Programmsicherheit und Frame Conditions
- Ausblick und Rückblick

Korrekte Software: Grundlagen und Methoden
Vorlesung 5 vom 04.05.17: Äquivalenz der Operationalen und Denotationalen Semantik

Serge Autexier, Christoph Lüth

Universität Bremen

Sommersemester 2017

Fahrplan

Operationale vs. denotationale Semantik

	Operational $\langle a, \sigma\rangle \to_{Aexp} n$	Denotational $\mathcal{A}[\![a]\!]$
m∈ **N**	$\langle m, \sigma\rangle \to_{Aexp} m$	$\{(\sigma, m) \mid \sigma \in \Sigma\}$
x∈ **Loc**	$\dfrac{x \in Dom(\sigma)}{\langle x, \sigma\rangle \to_{Aexp} \sigma(x)}$	$\{(\sigma, \sigma(x)) \mid \sigma \in \Sigma, x \in Dom(\sigma)\}$
$a_1 \circ a_2$	$\dfrac{\langle a_1, \sigma\rangle \to_{Aexp} n \quad \langle a_2, \sigma\rangle \to_{Aexp} m \quad n, m \neq \bot}{\langle a_1 \circ a_2, \sigma\rangle \to_{Aexp} n \circ' m}$	$\{(\sigma, n \circ' m) \mid \sigma \in \Sigma, (\sigma, n) \in \mathcal{A}[\![a_1]\!], (\sigma, m) \in \mathcal{A}[\![a_2]\!]\}$
	$\dfrac{\langle a_1, \sigma\rangle \to_{Aexp} n \quad \langle a_2, \sigma\rangle \to_{Aexp} m \quad n = \bot \text{ oder } m = \bot}{\langle a_1 \circ a_2, \sigma\rangle \to_{Aexp} \bot}$ $\circ \in \{+, *, -\}$	

Operationale vs. denotationale Semantik

	Operational $\langle a, \sigma\rangle \to_{Aexp} n$	Denotational $\mathcal{A}[\![a]\!]$
a_1/a_2	$\dfrac{\langle a_1, \sigma\rangle \to_{Aexp} n \quad \langle a_2, \sigma\rangle \to_{Aexp} m \quad m \neq 0 \quad m, n \neq \bot}{\langle a_1 \circ a_2, \sigma\rangle \to_{Aexp} n \circ' m}$	$\{(\sigma, n/m) \mid \sigma \in \Sigma, (\sigma, n) \in \mathcal{A}[\![a_1]\!], (\sigma, m) \in \mathcal{A}[\![a_2]\!], m \neq 0\}$
	$\dfrac{\langle a_1, \sigma\rangle \to_{Aexp} n \quad \langle a_2, \sigma\rangle \to_{Aexp} m \quad n = \bot, m = \bot \text{ oder } m = 0}{\langle a_1/a_2, \sigma\rangle \to_{Aexp} \bot}$	

Äquivalenz operationale und denotationale Semantik

- Für alle $a \in$ **Aexp**, für alle $n \in$ **N**, für alle Zustände σ:

$$\langle a, \sigma\rangle \to_{Aexp} n \Leftrightarrow (\sigma, n) \in \mathcal{A}[\![a]\!]$$
$$\langle a, \sigma\rangle \to_{Aexp} \bot \Leftrightarrow \sigma \notin Dom(\mathcal{A}[\![a]\!])$$

- Beweis per struktureller Induktion über a.

Operationale vs. denotationale Semantik

	Operational $\langle b, \sigma\rangle \to_{Bexp} 0\vert 1$	Denotational $\mathcal{B}[\![b]\!]$
1	$\langle 1, \sigma\rangle \to_{Bexp} 1$	$\{(\sigma, 1) \mid \sigma \in \Sigma\}$
0	$\langle 0, \sigma\rangle \to_{Bexp} 0$	$\{(\sigma, 0) \mid \sigma \in \Sigma\}$

Operationale vs. denotationale Semantik

	Operat. $\langle b, \sigma\rangle \to_{Bexp} 0\vert 1$	Denotational $\mathcal{B}[\![b]\!]$
$a_0 == a_1$	$\dfrac{\langle a_0, \sigma\rangle \to_{Aexp} n \quad \langle a_1, \sigma\rangle \to_{Aexp} m \quad n, m \neq \bot \quad n = m}{\langle a_0 == a_1, \sigma\rangle \to_{Bexp} 1}$ $\dfrac{\langle a_0, \sigma\rangle \to_{Aexp} n \quad \langle a_1, \sigma\rangle \to_{Aexp} m \quad n, m \neq \bot \quad n \neq m}{\langle a_0 == a_1, \sigma\rangle \to_{Bexp} 0}$ $\dfrac{\langle a_0, \sigma\rangle \to_{Aexp} n \quad \langle a_1, \sigma\rangle \to_{Aexp} m \quad n = \bot \text{ oder } m = \bot}{\langle a_0 == a_1, \sigma\rangle \to_{Bexp} \bot}$	$\{(\sigma, 1) \mid \sigma \in \Sigma, (\sigma, n_0) \in \mathcal{A}[\![a_0]\!], (\sigma, n_1) \in \mathcal{A}[\![a_1]\!], n_0 = n_1\}$ $\cup$ $\{(\sigma, 0) \mid \sigma \in \Sigma, (\sigma, n_0) \in \mathcal{A}[\![a_0]\!], (\sigma, n_1) \in \mathcal{A}[\![a_1]\!], n_0 \neq n_1\}$
$a1 <= a2$	analog	

Operationale vs. denotationale Semantik

	Operational $\langle a, \sigma\rangle \to_{Bexp} b$	Denotational $\mathcal{B}[\![b]\!]$
$b_1 \&\& b_0$	$\dfrac{\langle b_1, \sigma\rangle \to_{Bexp} 0}{\langle b_1 \&\& b_2, \sigma\rangle \to 0}$ $\dfrac{\langle b_1, \sigma\rangle \to_{Bexp} 1 \quad \langle b_2, \sigma\rangle \to_{Bexp} b}{\langle b_1 \&\& b_2, \sigma\rangle \to b}$ $\dfrac{\langle b_1, \sigma\rangle \to_{Bexp} \bot}{\langle b_1 \&\& b_2, \sigma\rangle \to \bot}$	$\{(\sigma, 0) \mid (\sigma, 0) \in \mathcal{B}[\![b_1]\!]\}$ $\{(\sigma, b) \mid (\sigma, 1) \in \mathcal{B}[\![b_1]\!], (\sigma, b) \in \mathcal{B}[\![b_2]\!]\}$
$b_1 \vert\vert b_2$	analog	
$!n$	...	

Äquivalenz operationale und denotationale Semantik

- Für alle $b \in \mathbf{Bexp}$, für alle $t \in \mathbf{B}$, for alle Zustände σ:

$$\langle b, \sigma \rangle \to_{Bexp} t \Leftrightarrow (\sigma, t) \in \mathcal{B}[\![b]\!]$$
$$\langle b, \sigma \rangle \to_{Bexp} \bot \Leftrightarrow \sigma \notin Dom(\mathcal{B}[\![b]\!])$$

- Beweis per struktureller Induktion über b (unter Verwendung der Äquivalenz für AExp).

Operationale vs. denotationale Semantik

	Operational $\langle c, \sigma \rangle \to_{Stmt} \sigma' \mid \bot$	Denotational $\mathcal{C}[\![c]\!]$
$\{c_1 \dots c_n\}$	$\dfrac{}{\langle \{\}, \sigma \rangle \to_{Stmt} \sigma}$ $\dfrac{\langle c_1, \sigma \rangle \to_{Stmt} \sigma'' \neq \bot \quad \langle \{c_2 \dots c_n\}, \sigma' \rangle \to_{Stmt} \sigma''}{\langle \{c_1 \dots c_n\}, \sigma \rangle \to_{Stmt} \sigma''}$ $\dfrac{\langle c_1, \sigma \rangle \to_{Stmt} \bot}{\langle \{c_1 \dots c_n\}, \sigma \rangle \to_{Stmt} \bot}$	$\mathcal{B}[\![c_n]\!] \circ \dots \mathcal{B}[\![c_1]\!] \circ Id$
$x = a$	$\dfrac{\langle a, \sigma \rangle \to_{Aexp} n}{\langle x = a, \sigma \rangle \to_{Stmt} \sigma[n/x]}$ $\dfrac{\langle a, \sigma \rangle \to_{Aexp} \bot}{\langle x = a, \sigma \rangle \to_{Stmt} \bot}$	$\{(\sigma, \sigma[n/X]) \mid (\sigma, n) \in \mathcal{A}[\![a]\!]\}$

Operationale vs. denotationale Semantik

	Operational $\langle c, \sigma \rangle \to_{Stmt} \sigma' \mid \bot$	Denotational $\mathcal{C}[\![c]\!]$
if (b) c_0	$\dfrac{\langle b, \sigma \rangle \to_{Bexp} 1 \quad \langle c_0, \sigma \rangle \to_{Stmt} \sigma'}{\langle c, \sigma \rangle \to_{Stmt} \sigma'}$ $\dfrac{\langle b, \sigma \rangle \to_{Bexp} \bot}{\langle c, \sigma \rangle \to_{Stmt} \bot}$	$\{(\sigma, \sigma') \mid (\sigma, 1) \in \mathcal{B}[\![b]\!], (\sigma, \sigma') \in \mathcal{C}[\![c_0]\!]\}$
else c_1	$\dfrac{\langle b, \sigma \rangle \to_{Bexp} 0 \quad \langle c_1, \sigma \rangle \to_{Stmt} \sigma'}{\langle c, \sigma \rangle \to_{Stmt} \sigma'}$	$\{(\sigma, \sigma') \mid (\sigma, 0) \in \mathcal{B}[\![b]\!], (\sigma, \sigma') \in \mathcal{C}[\![c_1]\!]\}$

Operationale vs. denotationale Semantik

	Operational $\langle c, \sigma \rangle \to_{Stmt} \sigma' \mid \bot$	Denotational $\mathcal{C}[\![c]\!]$
$\underbrace{\textit{while } (b) \ c}_{w}$	$\dfrac{\langle b, \sigma \rangle \to_{Bexp} 0}{\langle w, \sigma \rangle \to_{Stmt} \sigma} \quad \dfrac{\langle b, \sigma \rangle \to_{Bexp} \bot}{\langle w, \sigma \rangle \to_{Stmt} \bot}$ $\dfrac{\langle b, \sigma \rangle \to_{Bexp} 1 \quad \langle c, \sigma \rangle \to_{Stmt} \sigma' \neq \bot \quad \langle w, \sigma' \rangle \to_{Stmt} \sigma''}{\langle w, \sigma \rangle \to_{Stmt} \sigma''}$ $\dfrac{\langle b, \sigma \rangle \to_{Bexp} 1 \quad \langle c, \sigma \rangle \to_{Stmt} \bot}{\langle w, \sigma \rangle \to_{Stmt} \bot}$	$fix(\Gamma)$

mit

$$\Gamma(\varphi) = \{(\sigma, \sigma') \mid (\sigma, 1) \in \mathcal{B}[\![b]\!], (\sigma, \sigma') \in \varphi \circ \mathcal{C}[\![c]\!]\} \cup \{(\sigma, \sigma) \mid (\sigma, 0) \in \mathcal{B}[\![b]\!]\}$$

Äquivalenz operationale und denotationale Semantik

- Für alle $c \in \mathbf{Stmt}$, für alle Zustände σ, σ':

$$\langle c, \sigma \rangle \to_{Stmt} \sigma' \Leftrightarrow (\sigma, \sigma') \in \mathcal{C}[\![c]\!]$$
$$\langle c, \sigma \rangle \to_{Stmt} \bot \Rightarrow \sigma \notin Dom(\mathcal{C}[\![c]\!])$$

- $\Rightarrow$ Beweis per Induktion über die Ableitung in der operationalen Semantik
- $\Leftarrow$ Beweis per struktureller Induktion über c (Verwendung der Äquivalenz für arithmetische und boolsche Ausdrücke). Für die While-Schleife Rückgriff auf Definition des Fixpunkts und Induktion über die Teilmengen $\Gamma^i(\emptyset)$ des Fixpunkts.
  - Gegenbeispiel für $\Leftarrow$ in der zweiten Aussage: wähle $c \equiv \textit{while}(1)\{\}$: $\mathcal{C}[\![c]\!] = \emptyset$ aber $\langle c, \sigma \rangle \to_{Stmt} \bot$ gilt nicht (sondern?).

Korrekte Software: Grundlagen und Methoden
Vorlesung 6 vom 14.05.17: Korrektheit der Floyd-Hoare-Logik

Serge Autexier, Christoph Lüth

Universität Bremen

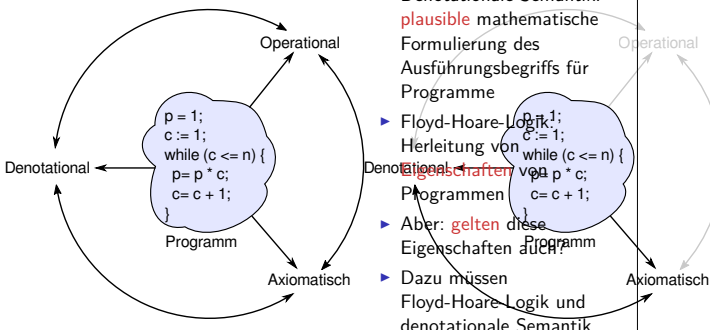
Sommersemester 2017

Fahrplan

Motivation

- In den letzten Wochen: **Semantik**
- Warum?
  - Bedeutung von Programmen mathematisch präzise fassen,
  - um insbesondere die über die Korrektheit von Programmen zu reden.
- Fragen:
  1. Was bedeutet es, wenn ein Programm korrekt ist?
  2. Wie können wir das beweisen?

Was gibt es heute?



- Denotationale Semantik: plausible mathematische Formulierung des Ausführungsbegriffs für Programme
- Floyd-Hoare-Logik: Herleitung von Eigenschaften von Programmen
- Aber: gelten diese Eigenschaften auch?
- Dazu müssen Floyd-Hoare-Logik und denotationale Semantik übereinstimmen.

Denotionale Semantik

- Denotat eines Ausdrucks (Programms) ist partielle Funktion:

$$\mathcal{A}[\![-]\!] : \mathbf{Aexp} \to \Sigma \rightharpoonup \mathbf{N}$$

$$\mathcal{B}[\![-]\!] : \mathbf{Bexp} \to \Sigma \rightharpoonup \mathbf{T}$$

$$\mathcal{C}[\![-]\!] : \mathbf{Stmt} \to \Sigma \rightharpoonup \Sigma$$

- $f : A \rightharpoonup B$, dann ($\bot$ steht für "undefiniert"):

$$\mathrm{def}(f(x)) \iff f(x) \neq \bot$$

Floyd-Hoare-Tripel: Gültigkeit und Herleitbarkeit

$P, Q \in \mathbf{Bexp}, c \in \mathbf{Stmt}$

$\models \{P\}\, c\, \{Q\}$ "Hoare-Tripel gilt" (semantisch)

$\vdash \{P\}\, c\, \{Q\}$ "Hoare-Tripel herleitbar" (syntaktisch)

Remember:

Partielle Korrektheit ($\models \{P\}\, c\, \{Q\}$)

c ist partiell korrekt, wenn für alle Zustände σ, die P erfüllen:
wenn die Ausführung von c mit σ in σ' terminiert, **dann** erfüllt σ' Q

Bezug zur Semantik?

Hoare-Tripel und denotationale Semantik

- Mit der denotationalen Semantik können wir die Gültigkeit von Hoare-Tripeln formal definieren.
- Notation: für $P \in \mathbf{Bexp}$, $\sigma \models P \iff \mathcal{B}[\![P]\!](\sigma) = 1$

Gültigkeit von Hoare-Tripeln

$$\models \{P\}\, c\, \{Q\} \iff \forall \sigma \in \Sigma.\, \sigma \models P \wedge \mathrm{def}(\mathcal{C}[\![c]\!](\sigma)) \implies \mathcal{C}[\![c]\!]\sigma \models Q$$

- Aber: $\models \{P\}\, c\, \{Q\} \quad \overset{?}{\leftrightsquigarrow} \quad \vdash \{P\}\, c\, \{Q\}$

Überblick: die Regeln des Floyd-Hoare-Kalküls

$$\frac{}{\vdash \{P[e/x]\}\, x = e\, \{P\}}$$

$$\frac{}{\vdash \{A\}\, \{\}\, \{A\}} \qquad \frac{\vdash \{A\}\, c\, \{B\} \qquad \vdash \{B\}\, \{c_s\}\, \{C\}}{\vdash \{A\}\, \{c\ c_s\}\, \{C\}}$$

$$\frac{\vdash \{A \wedge b\}\, c_0\, \{B\} \qquad \vdash \{A \wedge \neg b\}\, c_1\, \{B\}}{\vdash \{A\}\ \textbf{if}\ (b)\ c_0\ \textbf{else}\ c_1\, \{B\}}$$

$$\frac{\vdash \{A \wedge b\}\, c\, \{A\}}{\vdash \{A\}\ \textbf{while}(b)\ c\, \{A \wedge \neg b\}} \qquad \frac{A' \implies A \qquad \vdash \{A\}\, c\, \{B\} \qquad B \implies B'}{\vdash \{A'\}\, c\, \{B'\}}$$

Korrektheit und Vollständigkeit

- Korrektheit: $\vdash \{P\}\, c\, \{Q\} \stackrel{?}{\Longrightarrow} \models \{P\}\, c\, \{Q\}$
  - Wir können nur gültige Eigenschaften von Programmen herleiten.
- Vollständigkeit: $\models \{P\}\, c\, \{Q\} \stackrel{?}{\Longrightarrow} \vdash \{P\}\, c\, \{Q\}$
  - Wir können alle gültigen Eigenschaften auch herleiten.

Korrektheit der Floyd-Hoare-Logik

Floyd-Hoare-Logik ist korrekt.

Wenn $\vdash \{P\}\, c\, \{Q\}$, dann $\models \{P\}\, c\, \{Q\}$.

Beweis:

- Durch strukturelle Induktion über der Herleitung von $\vdash \{P\}\, c\, \{Q\}$
- Bsp: Sequenz, Zuweisung, Weakening, While.

Vollständigkeit der Floyd-Hoare-Logik

Floyd-Hoare-Logik ist vollständig modulo weakening.

Wenn $\models \{P\}\, c\, \{Q\}$, dann $\vdash \{P\}\, c\, \{Q\}$ bis auf die Bedingungen der Weakening-Regel.

- Beweis durch Konstruktion der schwächsten Vorbedingung $\mathrm{wp}(c, Q)$.
- Wenn wir eine gültige Zusicherung nicht herleiten können, liegt das nur daran, dass wir eine Beweisverpflichtung nicht beweisen können.
- Logik erster Stufe ist unvollständig, also können wir gar nicht besser werden.

Zusammenfassung

- Die Gültigkeit von Hoare-Tripeln ist ein semantisches Konzept, und über die denotationale Semantik definiert.
- Das Verhältnis von denotationaler Semantik zur Floyd-Hoare-Logik ist also die Frage nach Korrektheit und Vollständigkeit.
- Floyd-Hoare-Logik ist korrekt, wir können nur gültige Zusicherungen herleiten.
- Floyd-Hoare-Logik ist vollständig bis auf das Weakening.

Korrekte Software: Grundlagen und Methoden
Vorlesung 7 vom 18.05.17: Vorwärts und Rückwärts mit Floyd und Hoare

Serge Autexier, Christoph Lüth

Universität Bremen

Sommersemester 2017

Fahrplan

Vorwärts oder Rückwärts?

Idee

- Hier ist ein einfaches Programm:

```
// {X = x ∧ Y = y}
z = y;
// {X = x ∧ Y = z}
y = x;
// {X = y ∧ Y = z}
x = z;
// {X = y ∧ Y = x}
```

- Wir sehen:
  1. Die Verifikation erfolgt rückwärts (von hinten nach vorne).
  2. Die Verifikation kann berechnet werden.
- Muss das so sein? Ist das immer so?

Rückwärtsanwendung der Regeln

- Zuweisungsregel kann rückwärts angewandt werden, weil die Nachbedingung eine offene Variable ist — P passt auf jede beliebige Nachbedingung.

$$\frac{}{\vdash \{P[e/x]\}\, x = e\, \{P\}}$$

- Was ist mit den anderen Regeln? Nur **while** macht Probleme!

$$\frac{}{\vdash \{A\}\, \{\}\, \{A\}} \qquad \frac{\vdash \{A \wedge b\}\, c_0\, \{B\} \qquad \vdash \{A \wedge \neg b\}\, c_1\, \{B\}}{\vdash \{A\}\ \textbf{if}\ (b)\ c_0\ \textbf{else}\ c_1\, \{B\}}$$

$$\frac{\vdash \{A\}\, c\, \{B\} \qquad \vdash \{B\}\, \{c_s\}\, \{C\}}{\vdash \{A\}\, \{c\ c_s\}\, \{C\}} \qquad \frac{\vdash \{A \wedge b\}\, c\, \{A\}}{\vdash \{A\}\ \textbf{while}(b)\ c\, \{A \wedge \neg b\}}$$

$$\frac{A' \Longrightarrow A \qquad \vdash \{A\}\, c\, \{B\} \qquad B \Longrightarrow B'}{\vdash \{A'\}\, c\, \{B'\}}$$

Vorwärts?

- Alternative Zuweisungsregel (nach Floyd):

$$\frac{V \notin FV(P)}{\vdash \{P\}\, x = e\, \{\exists V.x = e[V/x] \wedge P[V/x]\}}$$

  - $FV(P)$ sind die freien Variablen in P.
- Jetzt ist die Vorbedingung offen — Regel kann vorwärts angewandt werden
- Gilt auch für die anderen Regeln

Vorwärtsverkettung

$$\frac{V \notin FV(P)}{\vdash \{P\}\, x = e\, \{\exists V.x = e[V/x] \wedge P[V/x]\}}$$

- Ein einfaches Beispiel (nach Mike Gordon):

```
// {x = 1}
x= x+1;
// {∃V.x = x + 1[V/x] ∧ (x = 1)[V/x]}
```

- Vereinfachung der Nachbedingung:

$$\begin{aligned} & \exists V.x = (x+1)[V/x] \wedge (x = 1)[V/x] \\ \Longleftrightarrow\ & \exists V.x = (V+1) \wedge (V = 1) \\ \Longleftrightarrow\ & x = 1 + 1 \\ \Longleftrightarrow\ & x = 2 \end{aligned}$$

Vorwärtsverkettung

- Vorwärtsaxiom äquivalent zum Rückwärtsaxiom.
- In der Anwendung umständlicher.
- Vereinfachung benötigt Lemma: $\exists x.P(x) \wedge x = t \Longleftrightarrow P(t)$
- Vorteile?
  - Wir wollten doch sowieso die Anwendung automatisieren...
  - Wir stellen die Frage erstmal zurück

Zwischenfazit: Der Floyd-Hoare-Kalkül ist symmetrisch

Es gibt zwei Zuweisungsregeln, eine für die Rückwärtsanwendung von Regeln, eine für die Vorwärtsanwendung

Schwächste Vorbedingungen

Berechnung von Vorbedingungen

- Die Rückwärtsrechnung von einer gegebenen Nachbedingung entspricht der Berechnung einer Vorbedingung
- Gegeben C0-Programm c, Prädikat P, dann ist
  - $\mathrm{wp}(c, P)$ die schwächste Vorbedingung Q so dass $\models \{Q\}\, c\, \{P\}$;
  - $\mathrm{sp}(P, c)$ die stärkste Nachbedingung Q so dass $\models \{P\}\, c\, \{Q\}$
- Prädikat P schwächer als Q wenn $Q \Longrightarrow P$ (stärker wenn $P \Longrightarrow Q$).
- Semantische Charakterisierung:

$$\begin{array}{rcl} \models \{P\}\, c\, \{Q\} & \iff & P \Longrightarrow \mathrm{wp}(c, Q) \\ \models \{P\}\, c\, \{Q\} & \iff & \mathrm{sp}(P, c) \Longrightarrow Q \end{array}$$

Berechnung von $\mathrm{wp}(c, Q)$

- Einfach für Programme ohne Schleifen:

$$\begin{array}{rcl} \mathrm{wp}(\{\,\}, P) & \stackrel{def}{=} & P \\ \mathrm{wp}(x = e, P) & \stackrel{def}{=} & P[e/x] \\ \mathrm{wp}(\{c\ c_s\}, P) & \stackrel{def}{=} & \mathrm{wp}(c, \mathrm{wp}(\{c_s\}, P)) \\ \mathrm{wp}(\textbf{if}\ (b)\ c_0\ \textbf{else}\ c_1, P) & \stackrel{def}{=} & (b \wedge \mathrm{wp}(c_0, P)) \vee (\neg b \wedge \mathrm{wp}(c_1, P)) \end{array}$$

- Für Schleifen: nicht entscheidbar.
  - "Cannot in general compute a finite formula" (Gordon)
- Wir können rekursive Formulierung angeben:

$$\mathrm{wp}(\textbf{while}\ (b)\ \{c\}, P) \stackrel{def}{=} (\neg b \wedge P) \vee (b \wedge \mathrm{wp}(c, \mathrm{wp}(\textbf{while}\ (b)\ \{c\}, P)))$$

  - Hilft auch nicht weiter...

Lösung: Annotierte Programme

- Wir helfen dem Rechner weiter und annotieren die Schleifeninvariante am Programm.
- Damit berechnen wir:
  - die approximative schwächste Vorbedingung $\mathrm{awp}(c, Q)$ zusammen mit einer Menge von Verifikationsbedingungen $\mathrm{wvc}(c, Q)$
  - oder die approximative stärkste Nachbedingung $\mathrm{asp}(P, c)$ zusammen mit einer Menge von Verifikationsbedingungen $\mathrm{svc}(P, c)$
- Die Verifikationsbedingungen treten dort auf, wo die Weakening-Regel angewandt werden muss.
- Es gilt:

$$\begin{array}{rcl} \bigwedge \mathrm{wvc}(c, Q) & \Longrightarrow & \models \{\mathrm{awp}(c, Q)\}\, c\, \{Q\} \\ \bigwedge \mathrm{svc}(P, c) & \Longrightarrow & \models \{P\}\, c\, \{\mathrm{asp}(P, c)\} \end{array}$$

Approximative schwächste Vorbedingung

- Für die **while**-Schleife:

$$\begin{array}{rcl} \mathrm{awp}(\textbf{while}\ (b)\ /{**}\ \textbf{inv}\ i\ {*}/\ c, P) & \stackrel{def}{=} & i \\ \mathrm{wvc}(\textbf{while}\ (b)\ /{**}\ \textbf{inv}\ i\ {*}/\ c, P) & \stackrel{def}{=} & \mathrm{wvc}(c, i) \\ & & \cup \{i \wedge b \Longrightarrow \mathrm{awp}(c, i)\} \\ & & \cup \{i \wedge \neg b \Longrightarrow P\} \end{array}$$

- Entspricht der **while**-Regel (1) mit Weakening (2):

$$\frac{\vdash \{A \wedge b\}\, c\, \{A\}}{\vdash \{A\}\ \textbf{while}(b)\ c\, \{A \wedge \neg b\}} \quad (1)$$

$$\frac{A \wedge b \Longrightarrow C \qquad \vdash \{C\}\, c\, \{A\} \qquad A \wedge \neg b \Longrightarrow B}{\vdash \{A\}\ \textbf{while}(b)\ c\, \{B\}} \quad (2)$$

Überblick: Approximative schwächste Vorbedingung

$$\begin{array}{rcl} \mathrm{awp}(\{\,\}, P) & \stackrel{def}{=} & P \\ \mathrm{awp}(x = e, P) & \stackrel{def}{=} & P[e/x] \\ \mathrm{awp}(\{c\ c_s\}, P) & \stackrel{def}{=} & \mathrm{awp}(c, \mathrm{awp}(\{c_s\}, P)) \\ \mathrm{awp}(\textbf{if}\ (b)\ c_0\ \textbf{else}\ c_1, P) & \stackrel{def}{=} & (b \wedge \mathrm{awp}(c_0, P)) \vee (\neg b \wedge \mathrm{awp}(c_1, P)) \\ \mathrm{awp}(/{**}\ \{q\}\ {*}/, P) & \stackrel{def}{=} & q \\ \mathrm{awp}(\textbf{while}\ (b)\ /{**}\ \textbf{inv}\ i\ {*}/\ c, P) & \stackrel{def}{=} & i \end{array}$$

$$\begin{array}{rcl} \mathrm{wvc}(\{\,\}, P) & \stackrel{def}{=} & \emptyset \\ \mathrm{wvc}(x = e, P) & \stackrel{def}{=} & \emptyset \\ \mathrm{wvc}(\{c\ c_s\}, P) & \stackrel{def}{=} & \mathrm{wvc}(c, \mathrm{awp}(\{c_s\}, P)) \cup \mathrm{wvc}(\{c_s\}, P) \\ \mathrm{wvc}(\textbf{if}\ (b)\ c_0\ \textbf{else}\ c_1, P) & \stackrel{def}{=} & \mathrm{wvc}(c_0, P) \cup \mathrm{wvc}(c_1, P) \\ \mathrm{wvc}(/{**}\ \{q\}\ {*}/, P) & \stackrel{def}{=} & \{q \Longrightarrow P\} \\ \mathrm{wvc}(\textbf{while}\ (b)\ /{**}\ \textbf{inv}\ i\ {*}/\ c, P) & \stackrel{def}{=} & \mathrm{wvc}(c, i) \cup \{i \wedge b \Longrightarrow \mathrm{awp}(c, i)\} \\ & & \cup \{i \wedge \neg b \Longrightarrow P\} \end{array}$$

Alternative Schreibweise: AWP

$$\frac{}{P \leftarrow_{awp} \{\,\}, P} \qquad \frac{}{P[e/x] \leftarrow_{awp} x = e, P}$$

$$\frac{P_c \leftarrow_{awp} c, P_{c_s} \qquad P_{c_s} \leftarrow_{awp} \{c_s\}, P}{P_c \leftarrow_{awp} \{c\ c_s\}, P}$$

$$\frac{P_{c_0} \leftarrow_{awp} c_0, P \qquad P_{c_1} \leftarrow_{awp} c_1, P}{(b \wedge P_{c_0}) \vee (\neg b \wedge P_{c_1}) \leftarrow_{awp} \textbf{if}\ (b)\ c_0\ \textbf{else}\ c_1, P}$$

$$\frac{}{i \leftarrow_{awp} \textbf{while}\ (b)\ /{**}\ \textbf{inv}\ i\ {*}/\ c, P}$$

Alternative Schreibweise: WVC

$$\frac{}{\{\,\}, P \rightarrow_{wvc} \emptyset} \qquad \frac{}{x = e, P \rightarrow_{wvc} \emptyset}$$

$$\frac{P_{c_s} \leftarrow_{awp} c_s, P \qquad c, P_{c_s} \rightarrow_{wvc} VC_c \qquad \{c_s\}, P \rightarrow_{wvc} VC_{c_s}}{\{c\ c_s\}, P \rightarrow_{wvc} VC_c \cup VC_{c_s}}$$

$$\frac{c_0, P \rightarrow_{wvc} VC_{c_0} \qquad c_1, P \rightarrow_{wvc} VC_{c_1}}{\textbf{if}\ (b)\ c_0\ \textbf{else}\ c_1, P \rightarrow_{wvc} VC_{c_0} \cup VC_{c_1}}$$

$$\frac{}{/{**}\ \{q\}\ {*}/, P \rightarrow_{wvc} \{q \Longrightarrow P\}}$$

$$\frac{c, i \rightarrow_{wvc} VC_c \qquad P_c \leftarrow_{awp} c, i}{\textbf{while}\ (b)\ /{**}\ \textbf{inv}\ i\ {*}/\ c, P \rightarrow_{wvc} VC_c \cup \{i \wedge b \Longrightarrow P_c, i \wedge \neg b \Longrightarrow P\}}$$

Beispiel: das Fakultätsprogramm

- In der Praxis sind Vor- und Nachbedingung gegeben, und nur die Verifikationsbedingungen relevant.
- Sei F das annotierte Fakultätsprogramm:

```
int c, n, p;

/** { 0 <= n } */
p= 1;
c= 1;
while (c <= n) /** inv p == fac(c-1) && c-1 <= n; */ {
  p = p * c;
  c = c + 1;
  }
/** { p == fac(n) } */
```

- Berechnung der Verifikationsbedingungen zur Nachbedingung: $\mathrm{wvc}(F, p == \mathit{fac}(N))$

Zusammenfassung

- Die Regeln des Floyd-Hoare-Kalküls sind symmetrisch: Die Zuweisungsregel gibt es"rückwärts" und "vorwärts".
- Bis auf die Invarianten an Schleifen können wir Korrektheit automatisch prüfen.
- Wir annotieren daher die Invarianten an Schleifen, und können dann die schwächste Vorbedingung und Verifikationsbedingungen automatisch berechnen.
  - Davon sind die Verifikationsbedingungen das interessante.
- Die Generierung von Verifikationsbedingungen korrespondiert zur relativen Vollständigkeit der Floyd-Hoare-Logik.

Korrekte Software: Grundlagen und Methoden
Vorlesung 8 vom 22.05.17: Funktionen und Prozeduren

Serge Autexier, Christoph Lüth

Universität Bremen

Sommersemester 2017

Fahrplan

Funktionen & Prozeduren

- Funktionen sind das zentrale Modularisierungskonzept von C
  - Kleinste Einheit
  - NB. Prozeduren sind nur Funktionen vom Typ **void**
  - Auch in den meisten anderen Sprachen, meist mit Zustandsverkapselung (Methoden)
- Wie behandeln wir Funktionen?

Modellierung und Spezifikation von Funktionen

Wir brauchen:

(1) Von Anweisungen zu Funktionen: Deklarationen und Parameter

(2) Semantik von Funktionsdefinition und Funktionsaufruf

(3) Spezifikation von Funktionen

(4) Beweisregeln für Funktionsdefinition und Funktionsaufruf

Von Anweisungen zu Funktionen

- Erweiterung unserer Kernsprache:

$$\begin{aligned}
\textbf{FunDef} &::= \textbf{Type}\ \mathit{Id}(\textbf{Param}^*)\ \textbf{FunSpec}^+\ \textbf{Blk}\\
\textbf{Param} &::= \textbf{Type}\ \mathit{Id}\\
\textbf{Blk} &::= \{\textbf{Decl}^*\ \textbf{Stmt}\}\\
\textbf{Decl} &::= \textbf{Type}\ \mathit{Id}\ = \textbf{Aexp} \mid \textbf{Type}\ \mathit{Id}\\
\\
\textbf{Aexp} &::= \ldots \mid \mathit{Id}(\textbf{Aexp}^*)\\
\textbf{Stmt} &::= \ldots \mid \mathit{Id}(\textbf{Aexp}^*) \mid \textbf{return}\ (\textbf{Aexp})?
\end{aligned}$$

- **Type**: zur Zeit nur **int**; Initialisierer: konstanter Ausdruck
- **FunSpec** später
- Vereinfachte Syntax (konkrete Syntax mischt **Type** und *Id*, Kommata bei Argumenten, ...)

Rückgabewerte

- Problem: **return** bricht sequentiellen Kontrollfluss:

```
if (x == 0) return -1;
y = y / x;      // Wird nicht immer erreicht
```

- Lösung 1: verbieten!
  - MISRA-C (Guidelines for the use of the C language in critical systems):

> **Rule 14.7 (required)**
> A function shall have a single point of exit at the end of the function.

  - Nicht immer möglich, unübersichtlicher Code...
- Lösung 2: Erweiterung der Semantik von $\Sigma \rightharpoonup \Sigma$ zu $\Sigma \rightharpoonup (\Sigma + \Sigma \times \mathbf{V})$

Erweiterte Semantik

- Denotat einer Anweisung: $\Sigma \rightharpoonup (\Sigma + \Sigma \times \mathbf{V})$ $\Sigma \rightharpoonup (\Sigma + \Sigma \times \mathbf{V}_U)$
- Abbildung von Ausgangszustand Σ auf:
  - Sequentieller Folgezustand, oder
  - Rückgabewert und Rückgabezustand
- Was ist mit **void**?
  - Erweiterte Werte: $\mathbf{V}_U \stackrel{def}{=} \mathbf{V} + \{*\}$
- Komposition zweier Anweisungen $f, g : \Sigma \rightharpoonup (\Sigma + \Sigma \times \mathbf{V}_U)$:

$$g \cdot_S f(\sigma) \stackrel{def}{=} \begin{cases} g(\sigma') & f(\sigma) = \sigma' \\ (\sigma', v) & f(\sigma) = (\sigma', v) \end{cases}$$

Semantik von Anweisungen

$$\mathcal{C}[\![.]\!] : \textbf{Stmt} \to \Sigma \rightharpoonup (\Sigma + \Sigma \times \mathbf{V}_U)$$

$$\begin{aligned}
\mathcal{C}[\![x = e]\!] &= \{(\sigma, \sigma[a/x]) \mid (\sigma, a) \in \mathcal{A}[\![e]\!]\}\\
\mathcal{C}[\![\{c\ c_s\}]\!] &= \mathcal{C}[\![c_s]\!] \cdot_S \mathcal{C}[\![c]\!] \qquad \text{Komposition wie oben}\\
\mathcal{C}[\![\{\,\}]\!] &= \mathbf{Id} \qquad\qquad \mathbf{Id} := \{(\sigma, \sigma) \mid \sigma \in \Sigma\}\\
\mathcal{C}[\![\textbf{if}\ (b)\ c_0\ \textbf{else}\ c_1]\!] &= \{(\sigma, \tau) \mid (\sigma, 1) \in \mathcal{B}[\![b]\!] \wedge (\sigma, \tau) \in \mathcal{C}[\![c_0]\!]\}\\
&\quad \cup \{(\sigma, \tau) \mid (\sigma, 0) \in \mathcal{B}[\![b]\!] \wedge (\sigma, \tau) \in \mathcal{C}[\![c_1]\!]\}\\
&\quad \text{mit } \tau \in \Sigma \cup (\Sigma \times \mathbf{V}_U)\\
\mathcal{C}[\![\textbf{return}\ e]\!] &= \{(\sigma, (\sigma, a)) \mid (\sigma, a) \in \mathcal{A}[\![e]\!]\}\\
\mathcal{C}[\![\textbf{return}]\!] &= \{(\sigma, (\sigma, *))\}
\end{aligned}$$

$$\begin{aligned}
\mathcal{C}[\![\textbf{while}\ (b)\ c]\!] &= \mathit{fix}(\Gamma)\\
\Gamma(\psi) &\stackrel{def}{=} \{(\sigma, \tau) \mid (\sigma, 1) \in \mathcal{B}[\![b]\!] \wedge (\sigma, \tau) \in \psi \cdot_S \mathcal{C}[\![c]\!]\}\\
&\quad \cup \{(\sigma, \sigma) \mid (\sigma, 0) \in \mathcal{B}[\![b]\!]\}
\end{aligned}$$

Semantik von Funktionsdefinitionen

$$\mathcal{D}_{fd}[\![.]\!] : \textbf{FunDef} \to \mathbf{V}^n \rightharpoonup \Sigma \rightharpoonup \Sigma \times \mathbf{V}_U$$

Das Denotat einer Funktion ist eine Anweisung, die über den tatsächlichen Werten für die Funktionsargumente parametriert ist.

$$\begin{aligned}&\mathcal{D}_{fd}[\![f(t_1\ p_1, t_2\ p_2, \ldots, t_n\ p_n)\ blk]\!] = \\ &\quad \lambda v_1, \ldots, v_n.\, \{(\sigma, (\sigma', v)) \mid \\ &\qquad (\sigma, (\sigma', v)) \in \mathcal{D}_{blk}[\![blk]\!] \cdot_S \{(\sigma, \sigma[v_1/p_1, \ldots, v_n/p_n])\}\}\end{aligned}$$

- Die Funktionsargumente sind lokale Deklarationen, die mit den Aufrufwerten initialisiert werden.
  - Insbesondere können sie lokal in der Funktion verändert werden.
- Von $\mathcal{D}_{blk}[\![blk]\!]$ sind nur Rückgabezustände interessant.

Semantik von Blöcken und Deklarationen

$$\begin{aligned}\mathcal{D}_{blk}[\![.]\!] &: \textbf{Blk} \rightharpoonup \Sigma \rightharpoonup (\Sigma + \Sigma \times \mathbf{V}_U) \\ \mathcal{D}_{d}[\![.]\!] &: \textbf{Decl} \rightharpoonup \Sigma \rightharpoonup (\Sigma + \Sigma \times \mathbf{V}_U)\end{aligned}$$

Blöcke bestehen aus Deklarationen und einer Anweisung:

$$\begin{aligned}\mathcal{D}_{blk}[\![decls\ stmts]\!] &= \mathcal{C}[\![stmts]\!] \cdot_S \mathcal{D}_d[\![decls]\!] \\ \mathcal{D}_d[\![t\ i]\!] &= \{(\sigma, \sigma[\bot/i])\} \\ \mathcal{D}_d[\![t\ i\ = init]\!] &= \{(\sigma, \sigma[\mathcal{A}_{init}[\![init]\!]/i])\}\end{aligned}$$

- Verallgemeinerung auf Sequenz von Deklarationen
- $\mathcal{A}_{init}[\![.]\!]$ ist das Denotat von Initialisierungen:
  - Nur für konstante Ausdrücke, daher nicht zustandsabhängig

Funktionsaufrufe

- Aufruf einer Funktion: $f(t_1, \ldots, t_n)$:
  - Auswertung der Argumente $t_1, \ldots, t_n$
  - Einsetzen in die Semantik $\mathcal{D}_{fd}[\![f]\!]$
- Was ist mit Seiteneffekten?
  - Erst mal gar nichts, unsere Sprache hat noch keine ...
- Call by name, call by value, call by reference...?
  - C kennt nur call by value (C-Standard 99, §6.9.1. (10))

Funktionsaufrufe

- Um eine Funktion f aufzurufen, müssen wir (statisch!) die Semantik der Definition von f dem Bezeichner f zuordnen.
- Deshalb brauchen wir eine Umgebung (Environment):

$$\begin{aligned}Env &= Id \rightharpoonup [\![\textbf{FunDef}]\!] \\ &= Id \rightharpoonup \mathbf{V}^N \rightharpoonup \Sigma \rightharpoonup (\Sigma \times \mathbf{V}_u)\end{aligned}$$

- Das Environment ist zusätzlicher Parameter für alle Definitionen

Semantik von Funktionsaufrufen

$$\begin{aligned}\mathcal{A}[\![f(t_1, \ldots, t_n)]\!]\Gamma &= \{(\sigma, v) \mid \exists \sigma', v.\, (\sigma, (\sigma', v)) \in \Gamma(f)(v_1, \ldots v_n) \\ &\qquad \wedge (\sigma, v_i) \in \mathcal{A}[\![t_i]\!]\Gamma\} \\ \mathcal{C}[\![f(t_1, \ldots, t_n)]\!]\Gamma &= \{(\sigma, \sigma') \mid \exists \sigma'.\, (\sigma, (\sigma', *)) \in \Gamma(f)(v_1, \ldots v_n) \\ &\qquad \wedge (\sigma, v_i) \in \mathcal{A}[\![t_i]\!]\Gamma\} \\ \mathcal{C}[\![x = f(t_1, \ldots, t_n)]\!]\Gamma &= \{(\sigma, \sigma'[v/x]) \mid \exists \sigma', v.\, (\sigma, (\sigma', v)) \in \Gamma(f)(v_1, \ldots v_n) \\ &\qquad \wedge (\sigma, v_i) \in \mathcal{A}[\![t_i]\!]\Gamma\}\end{aligned}$$

- Aufruf einer nicht-definierten Funktion f oder mit falscher Anzahl n von Parametern ist nicht definiert
  - Muss durch statische Analyse verhindert werden
- Aufruf von Funktion $\mathcal{A}[\![f(t_1, \ldots, t_n)]\!]$ ignoriert Endzustand
- Aufruf von Prozedur $\mathcal{C}[\![f(t_1, \ldots, t_n)]\!]$ ignoriert Rückgabewert
- Besser: Kombination mit Zuweisung

Spezifikation von Funktionen

- Wir spezifizieren Funktionen durch Vor- und Nachbedingungen
  - Ähnlich den Hoare-Tripeln, aber vereinfachte Syntax
  - Behavioural specification, angelehnt an JML, OCL, ACSL (Frama-C)
- Syntaktisch:

FunSpec ::= /** **pre** **Bexp** **post** **Bexp** */

Vorbedingung	**pre** sp;	$\Sigma \to \mathbf{T}$
Nachbedingung	**post** sp;	$\Sigma \times (\Sigma \times \mathbf{V}_U) \to \mathbf{T}$
	\old(e)	Wert von e im Vorzustand
	\result	Rückgabewert der Funktion

Semantik von Spezifikationen

- Vorbedingung: Auswertung als $\mathcal{B}[\![sp]\!]\,\Gamma$ über dem Vorzustand
- Nachbedingung: Erweiterung von $\mathcal{B}[\![.]\!]$ und $\mathcal{A}[\![.]\!]$
  - Ausdrücke können in Vor- oder Nachzustand ausgewertet werden.
  - **result**kann nicht in Funktionen vom Typ **void** auftreten.

$$\begin{aligned}\mathcal{B}_{sp}[\![.]\!] &: Env \to \textbf{Bexp} \rightharpoonup (\Sigma \times (\Sigma \times \mathbf{V}_U)) \to \mathbf{T} \\ \mathcal{A}_{sp}[\![.]\!] &: Env \to \textbf{Aexp} \rightharpoonup (\Sigma \times (\Sigma \times \mathbf{V}_U)) \to \mathbf{V} \\ \mathcal{B}_{sp}[\![!b]\!]\,\Gamma &= \{((\sigma, (\sigma', v)), 1) \mid ((\sigma, (\sigma', v)), 0) \in \mathcal{B}_{sp}[\![b]\!]\Gamma\} \\ &\quad \cup \{((\sigma, (\sigma', v)), 0) \mid ((\sigma, (\sigma', v)), 1) \in \mathcal{B}_{sp}[\![b]\!]\Gamma\} \\ &\ldots \\ \mathcal{B}_{sp}[\![\backslash\textbf{old}(e)]\!]\,\Gamma &= \{((\sigma, (\sigma', v)), b) \mid (\sigma, b) \in \mathcal{B}[\![e]\!]\,\Gamma\} \\ \mathcal{A}_{sp}[\![\backslash\textbf{old}(e)]\!]\,\Gamma &= \{((\sigma, (\sigma', v)), a) \mid (\sigma, a) \in \mathcal{A}[\![e]\!]\,\Gamma\} \\ \mathcal{A}_{sp}[\![\backslash\textbf{result}]\!]\,\Gamma &= \{((\sigma, (\sigma, v)), v)\}\end{aligned}$$

$$\mathcal{B}_{sp}[\![\textbf{pre}\ p\ \textbf{post}\ q]\!]\,\Gamma = \{(\sigma, (\sigma', v)) \mid \sigma \in \mathcal{B}[\![p]\!]\,\Gamma \wedge (\sigma', (\sigma, v)) \in \mathcal{B}_{sp}[\![p]\!]\,\Gamma\}$$

Gültigkeit von Spezifikationen

- Die Semantik von Spezifikationen erlaubt uns die Definition der semantischen Gültigkeit.

$$\begin{aligned}&\textbf{pre}\ p\ \textbf{post}\ q \models FunDef \\ &\quad \Longleftrightarrow \forall v_1, \ldots, v_n.\, \mathcal{D}_{fd}[\![FunDef]\!]\,\Gamma \in \mathcal{B}_{sp}[\![\textbf{pre}\ p\ \textbf{post}\ q]\!]\,\Gamma\end{aligned}$$

  - Γ enthält globale Definitionen, insbesondere andere Funktionen.
- Wie passt das zu $\models \{P\}\, c\, \{Q\}$ für Hoare-Tripel?
- Wie beweisen wir das? Erweiterung des Hoare-Kalküls

Erweiterung des Floyd-Hoare-Kalküls

$$\mathcal{C}[\![.]\!] : \mathbf{Stmt} \to \Sigma \rightharpoonup (\Sigma + \Sigma \times \mathbf{V}_U)$$

Hoare-Tripel: zusätzliche Spezifikation für Rückgabewert.

Partielle Korrektheit ($\models \{P\}\, c\, \{Q|Q_R\}$)

c ist partiell korrekt, wenn für alle Zustände σ, die P erfüllen:

- die Ausführung von c mit σ in σ' regulär terminiert, so dass σ' die Spezifikation Q erfüllt,
- oder die Ausführung von c in σ' mit dem Rückgabewert v terminiert, so dass (σ', v) die Rückgabespezifikation Q_R erfüllt.

$$\models \{P\}\, c\, \{Q|Q_R\} \iff$$
$$\forall \sigma.\, \sigma \models \mathcal{B}[\![P]\!]\Gamma \implies \exists \sigma'.\, (\sigma, \sigma') \in \mathcal{C}[\![c]\!]\Gamma \wedge \sigma' \models \mathcal{B}[\![Q]\!]\Gamma$$
$$\vee$$
$$\exists \sigma', v.\, (\sigma, (\sigma', v)) \in \mathcal{C}[\![c]\!]\Gamma \wedge (\sigma', v) \models \mathcal{B}[\![Q_R]\!]\Gamma$$

Kontext

- Wir benötigen ferner einen Kontext Γ, der Funktionsbezeichnern ihre Spezifikation (Vor/Nachbedingung) zuordnet.
- $\Gamma(f) = \forall x_1, \ldots, x_n.\, (P, Q)$, für Funktion $f(x_1, \ldots, x_n)$ mit Vorbedingung P und Nachbedingung Q.
- Notation: $\Gamma \models \{P\}\, c\, \{Q|Q_R\}$ und $\Gamma \vdash \{P\}\, c\, \{Q|Q_R\}$
- Korrektheit gilt immer nur im Kontext, dadurch kann jede Funktion separat verifiziert werden (Modularität)

Erweiterung des Floyd-Hoare-Kalküls: return

$$\frac{}{\Gamma \vdash \{Q\}\ \mathbf{return}\ \{P|Q\}} \qquad \frac{}{\Gamma \vdash \{Q[e/\backslash\mathbf{result}]\}\ \mathbf{return}\ e\ \{P|Q\}}$$

- Bei **return** wird die Rückgabespezifikation Q zur Vorbedingung, die reguläre Nachfolgespezifikation wird ignoriert, da die Ausführung von **return** kein Nachfolgezustand hat.
- **return** ohne Argument darf nur bei einer Nachbedingung Q auftreten, die kein \result enthält.
- Bei **return** mit Argument ersetzt der Rückgabewert den \result in der Rückgabespezifikation.

Erweiterung des Floyd-Hoare-Kalküls: Spezifikation

$$\frac{P \implies P'[y_i/\backslash\mathbf{old}(y_i)] \quad \Gamma[f \mapsto \forall x_1, \ldots, x_n.\, (P, Q)] \vdash \{P'\}\ blk\ \{Q|Q\}}{\Gamma \vdash f(x_1, \ldots, x_n)/{**}\ \text{pre}\ P\ \text{post}\ Q\ {*}/\ \{ds\ blk\}}$$

- Die Parameter x_i werden per Konvention nur als x_i referenziert, aber es ist immer der Wert im Vorzustand gemeint (eigentlich \old(x_i)).
- Variablen unterhalb von \old(y) werden bei der Substitution (Zuweisungsregel) nicht ersetzt!
- \old(y) wird beim Weakening von der Vorbedingung P ersetzt

Erweiterung des Floyd-Hoare-Kalküls: Aufruf

$$\frac{\Gamma(f) = \forall x_1, \ldots, x_n.\, (P, Q), f \text{ vom Typ } \mathbf{void}}{\Gamma \vdash \{Y_j = y_j \,\&\&\, P[t_i/x_i]\}\ f(t_1, \ldots, t_n)\ \{Q[t_i/x_i][Y_j/\backslash\mathbf{old}(y_j)]|Q_R\}}$$

$$\frac{\Gamma(f) = \forall x_1, \ldots, x_n.\, (P, Q)}{\Gamma \vdash \{Y_j = y_j \,\&\&\, P[t_i/x_i]\}\ x = f(t_1, \ldots, t_n)\ \{Q[t_i/x_i][Y_j/\backslash\mathbf{old}(y_j)][x/\backslash\mathbf{result}]|Q_R\}}$$

- Γ muss f mit der Vor-/Nachbedingung P, Q enthalten
- In P und Q werden Parameter x_i durch Argumente t_i ersetzt.
- $y_1, \ldots, y_m$ sind die als \old(y_j) in Q auftretenden Variablen
- $Y_1, \ldots, Y_m$ dürfen nicht in P oder Q enthalten sein
- Im ersten Fall (Aufruf als Prozedur) enthält Q kein \result

Erweiterter Floyd-Hoare-Kalkül I

$$\frac{}{\Gamma \vdash \{P\}\,\{}\,\{P|Q_R\}} \qquad \frac{\Gamma \vdash \{P\}\, c\, \{R|Q_R\} \quad \Gamma \vdash \{R\}\, cs\, \{Q|Q_R\}}{\Gamma \vdash \{P\}\, c\ cs\, \{Q|Q_R\}}$$

$$\frac{}{\Gamma \vdash \{Q[e/x]\}\, x = e\, \{Q|Q_R\}} \qquad \frac{\Gamma \vdash \{P \,\&\&\, b\}\, c\, \{P|Q_R\}}{\Gamma \vdash \{P\}\ \mathbf{while}\ (b)\ c\, \{P \,\&\&\, !\, b|Q_R\}}$$

$$\frac{\Gamma \vdash \{P \,\&\&\, b\}\, c_1\, \{Q|Q_R\} \quad \Gamma \vdash \{P \,\&\&\, !\, b\}\, c_2\, \{Q|Q_R\}}{\Gamma \vdash \{P\}\ \mathbf{if}\ (b)\ c_1\ \mathbf{else}\ c_2\, \{Q|Q_R\}}$$

$$\frac{P \longrightarrow P' \quad \Gamma \vdash \{P'\}\, c\, \{Q'|R'\} \quad Q' \longrightarrow Q \quad R' \longrightarrow R}{\Gamma \vdash \{P\}\, c\, \{Q|R\}}$$

Erweiterter Floyd-Hoare-Kalkül II

$$\frac{}{\Gamma \vdash \{Q\}\ \mathbf{return}\ \{P|Q\}} \qquad \frac{}{\Gamma \vdash \{Q[e/\backslash\mathbf{result}]\}\ \mathbf{return}\ e\ \{P|Q\}}$$

$$\frac{\Gamma[f \mapsto (P, Q)] \vdash \{X_i = x_i \,\&\&\, Y_j = y_j \,\&\&\, P\}\ blk\ \{Q[X_i/x_i][y_j/\backslash\mathbf{old}(y_j)]|Q[X_i/x_i][y_j/\backslash\mathbf{old}(y_j)]\}}{\Gamma \vdash f(x_1, \ldots, x_n)/{**}\ \text{pre}\ P\ \text{post}\ Q\ {**}/\ \{ds\ blk\}}$$

$$\frac{\Gamma(f) = \forall x_1, \ldots, x_n.\, (P, Q), f \text{ vom Typ } \mathbf{void}}{\Gamma \vdash \{Y_j = y_j \,\&\&\, P[t_i/x_i]\}\ f(t_1, \ldots, t_n)\ \{Q[t_i/x_i][Y_j/\backslash\mathbf{old}(y_j)]|Q_R\}}$$

$$\frac{\Gamma(f) = \forall x_1, \ldots, x_n.\, (P, Q)}{\Gamma \vdash \{Y_j = y_j \,\&\&\, P[t_i/x_i]\}\ x = f(t_1, \ldots, t_n)\ \{Q[t_i/x_i][Y_j/\backslash\mathbf{old}(y_j)][x/\backslash\mathbf{result}]|Q_R\}}$$

Beispiel: die Fakultätsfunktion, rekursiv

```
int fac(int x)
/** pre x >= 0
    post \result = factorial(x) */
{
 int r = 0;

 if (x == 0) { return 1; }
 r = fac(x- 1);
 return r * x;
}
```

Approximative schwächste Vorbedingung

- Erweiterung zu $\text{awp}(\Gamma, c, Q, Q_R)$ und $\text{wvc}(\Gamma, c, Q, Q_R)$ analog zu der Erweiterung der Floyd-Hoare-Regeln.
  - Es werden der Kontext Γ und eine Rückgabespezifikation Q_R benötigt.
- Es gilt:

$$\bigwedge \text{wvc}(\Gamma, c, Q, Q_R) \Longrightarrow \Gamma \models \{\text{awp}(c, Q, Q_R)\}\, c\, \{Q|Q_R\}$$

- Berechnung von awp und wvc:

$$\begin{aligned}
&\text{awp}(\Gamma, f(x_1, \ldots, x_n)/{*}{*}\ \textbf{pre}\ P\ \textbf{post}\ Q\ {*}/\ \{ds\ blk\}) \stackrel{def}{=} \text{awp}(\Gamma', blk, Q, Q)\\
&\text{wvc}(\Gamma, f(x_1, \ldots, x_n)/{*}{*}\ \textbf{pre}\ P\ \textbf{post}\ Q\ {*}/\ \{ds\ blk\}) \stackrel{def}{=}\\
&\qquad \{P \Longrightarrow \text{awp}(\Gamma', blk, Q, Q)[y_j/\backslash\textbf{old}(y_j)]\} \cup \text{wvc}(\Gamma', blk, Q, Q)\\
&\qquad \Gamma' \stackrel{def}{=} \Gamma[f \mapsto \forall x_1, \ldots, x_n.\, (P, Q)]
\end{aligned}$$

Approximative schwächste Vorbedingung (Revisited)

$$\begin{aligned}
\text{awp}(\Gamma, \{\}, Q, Q_R) &\stackrel{def}{=} Q\\
\text{awp}(\Gamma, l = f(e_1, \ldots, e_n), Q, Q_R) &\stackrel{def}{=} P[e_i/x_i]\\
&\quad \text{mit } \Gamma(f) = \forall x_1, \ldots, x_n.\, (P, R)\\
\text{awp}(\Gamma, f(e_1, \ldots, e_n), Q, Q_R) &\stackrel{def}{=} P[e_i/x_i]\\
&\quad \text{mit } \Gamma(f) = \forall x_1, \ldots, x_n.\, (P, R)\\
\text{awp}(\Gamma, l = e, Q, Q_R) &\stackrel{def}{=} P[e/l]\\
\text{awp}(\Gamma, \{c\ c_s\}, Q, Q_R) &\stackrel{def}{=} \text{awp}(\Gamma, c, \text{awp}(\{c_s\}, Q, Q_R), Q_R)\\
\text{awp}(\Gamma, \textbf{if}\ (b)\ \{c_0\}\ \textbf{else}\ \{c_1\}, Q, Q_R) &\stackrel{def}{=} (b\ \&\&\ \text{awp}(\Gamma, c_0, Q, Q_R))\\
&\quad \|\ (!\, b\ \&\&\ \text{awp}(\Gamma, c_1, Q, Q_R))\\
\text{awp}(\Gamma, /{*}{*}\ \{q\}\ {*}/, Q, Q_R) &\stackrel{def}{=} q\\
\text{awp}(\Gamma, \textbf{while}\ (b)\ /{*}{*}\ \textbf{inv}\ i\ {*}/\ c, Q, Q_R) &\stackrel{def}{=} i\\
\text{awp}(\Gamma, \textbf{return}\ e, Q, Q_R) &\stackrel{def}{=} Q_R[e/\backslash\textbf{result}]\\
\text{awp}(\Gamma, \textbf{return}, Q, Q_R) &\stackrel{def}{=} Q_R
\end{aligned}$$

Approximative Verifikationsbedingungen (Revisited)

$$\begin{aligned}
\text{wvc}(\Gamma, \{\}, Q, Q_R) &\stackrel{def}{=} \emptyset\\
\text{wvc}(\Gamma, x = e, Q, Q_R) &\stackrel{def}{=} \emptyset\\
\text{wvc}(\Gamma, x = f(e_1, \ldots, e_n), Q, Q_R) &\stackrel{def}{=} \{R[e_i/x_i][x/\backslash\textbf{result}] \Longrightarrow Q\}\\
&\quad \text{mit } \Gamma(f) = \forall x_1, \ldots, x_n.\, (P, R)\\
\text{wvc}(\Gamma, f(e_1, \ldots, e_n), Q, Q_R) &\stackrel{def}{=} \{R[e_i/x_i] \Longrightarrow Q\}\\
&\quad \text{mit } \Gamma(f) = \forall x_1, \ldots, x_n.\, (P, R)\\
\text{wvc}(\Gamma, \{c\ c_s\}, Q, Q_R) &\stackrel{def}{=} \text{wvc}(\Gamma, c, \text{awp}(\{c_s\}, Q, Q_R), Q_R)\\
&\quad \cup\ \text{wvc}(\Gamma, \{c_s\}, , Q, Q_R)\\
\text{wvc}(\Gamma, \textbf{if}\ (b)\ c_0\ \textbf{else}\ c_1, Q, Q_R) &\stackrel{def}{=} \text{wvc}(\Gamma, c_0, Q, Q_R)\\
&\quad \cup\ \text{wvc}(\Gamma, c_1, Q, Q_R)\\
\text{wvc}(\Gamma, /{*}{*}\ \{q\}\ {*}/, Q, Q_R) &\stackrel{def}{=} \{q \Longrightarrow Q\}\\
\text{wvc}(\Gamma, \textbf{while}\ (b)\ /{*}{*}\ \textbf{inv}\ i\ {*}/\ c, Q, Q_R) &\stackrel{def}{=} \text{wvc}(\Gamma, c, i, Q_R)\\
&\quad \cup \{i \wedge b \Longrightarrow \text{awp}(\Gamma, c, i, Q_R)\}\\
&\quad \cup \{i \wedge \neg b \Longrightarrow Q\}\\
\text{wvc}(\Gamma, \textbf{return}\ e, Q, Q_R) &\stackrel{def}{=} \emptyset
\end{aligned}$$

Zusammenfassung

- Funktionen sind zentrales Modularisierungskonzept
- Wir müssen Funktionen modular verifizieren können
- Erweiterung der Semantik:
  - Semantik von Deklarationen und Parameter — straightforward
  - Semantik von Rückgabewerten — Erweiterung der Semantik
  - Funktionsaufrufe — Environment, um Funktionsbezeichnern eine Semantik zu geben
- Erweiterung der Spezifikationen:
  - Spezifikation von Funktionen: Vor-/Nachzustand statt logischer Variablen
- Erweiterung des Hoare-Kalküls:
  - Environment, um andere Funktionen zu nutzen
  - Gesonderte Nachbedingung für Rückgabewert/Endzustand

Korrekte Software: Grundlagen und Methoden
Vorlesung 9 vom 01.06.17: Referenzen und Speichermodelle

Serge Autexier, Christoph Lüth

Universität Bremen

Sommersemester 2017

Fahrplan

Motivation

- Weitere Basisdatentypen von C (arrays und structs)
- Noch rein funktional, keine Pointer
  - Damit auch kein *call by reference*
  - Funktion können nur globale Seiteneffekte haben
  - Was wäre C ohne Pointer?

Arrays

```
int a[1][2];

bool b[][] = { {1, 0},
               {1, 1},
               {0, 0} }; /* Ergibt Array [3][2] */

printf(b[2][1]);  /* liefert '0' */

int six[6] = {1,2,3,4,5,6};

// Allgemeine Form

typ name[groesse1][groesse2]...[groesseN] =
    { ... }
    x;
```


Struct

```
struct Point {
  int x;
  int y;
};

struct Point a = { 1, 2};
struct Point b;

b.x = a.x;
b.y = a.y;
```


Rekursive Struct

Rekursion nur über Pointer möglich:

```
struct Liste {
  int kopf;
  struct Liste *rest;
} start;

start.kopf = 10; /* start.rest bleibt undefiniert */
```

struct Liste *rest ist ein incomplete type.

Refenzen in C

- Pointer in C ("pointer type"):
  - Schwach getypt (**void** * kompatibel mit allen Zeigertypen)
  - Eingeschränkte Zeigerarithmetik (Addition, Subtraktion)
  - Felder werden durch Zeigerarithmetik implementiert
- Pointer sind *first-class-values*
- C-Standard läßt das Speichermodell relativ offen
  - Repräsentation von Objekten

Erweiterung des Zustandmodells

- Erweiterung von Zustand und Werten:

$$\Sigma = \mathbf{Loc} \rightharpoonup \mathbf{V} \qquad \mathbf{V} = \mathbf{N} + \mathbf{Loc}$$

- Was ist **Loc**?
  - Locations (Speicheradressen)
  - Man kann **Loc** axiomatisch oder modellbasiert beschreiben.

Axiomatisches Zustandsmodell

- Der Zustand ist ein abstrakter Datentyp Σ mit zwei Operationen und folgenden Gleichungen:

$$read : \Sigma \to \mathbf{Loc} \rightharpoonup \mathbf{V}$$
$$upd : \Sigma \to \mathbf{Loc} \to \mathbf{V} \rightharpoonup \Sigma$$

$$read(upd(\sigma, l, v), l) = v$$
$$l \neq m \implies read(upd(\sigma, l, v), m) = read(\sigma, m)$$
$$upd(upd(\sigma, l, v), l, w) = upd(\sigma, l, w)$$
$$l \neq m \implies upd(upd(\sigma, l, v), m, w) = upd(upd(\sigma, m, w), l, v)$$

- Diese Gleichungen sind vollständig.

Axiomatisches Speichermodell

- Es gibt einen leeren Speicher, und neue ("frische") Adressen:

$$empty : \Sigma$$
$$fresh : \Sigma \to \mathbf{Loc}$$
$$rem : \Sigma \to \mathbf{Loc} \to \Sigma$$

  - *fresh* modelliert Allokation, *rem* modelliert Deallokation
- *dom* beschreibt den Definitionsbereich:

$$dom(\sigma) = \{l \mid \exists v.\, read(\sigma, l) = v\}$$
$$dom(empty) = \emptyset$$

- Eigenschaften von *empty*, *fresh* und *rem*:

$$fresh(\sigma) \notin dom(\sigma)$$
$$dom(rem(\sigma, l)) = dom(\sigma) \setminus \{l\}$$
$$l \neq m \implies read(rem(\sigma, l), m) = read(\sigma, m)$$

Zeigerarithmetik

- Erklärt noch keine Zeigerarithmetik — dazu:

$$add : \mathbf{Loc} \to \mathbb{Z} \to \mathbf{Loc}$$

  - Wir betrachten keine Differenz von Zeigern

$$add(l, 0) = l$$
$$add(add(l, a), b) = add(l, a + b)$$

Erweiterung der Semantik

- Problem: **Loc** haben unterschiedliche Semantik auf der linken oder rechten Seite einer Zuweisung.
  - x = x+1 — Links: Addresse der Variablen, rechts: Wert an dieser Adresse
- Lösung: "Except when it is (...) the operand of the unary & oprerator, the left operand of the . operator or an assigment operator, an lvalue that does not have array type is converted to the value stored in the designated object (and is no longer an lvalue)"
  C99 Standard, §6.3.2.1 (2)

Umgebung

- Für Funktionen brauchten wir eine Umgebung (Environment):

$$Env = Id \rightharpoonup [\![\mathbf{FunDef}]\!]$$
$$= Id \rightharpoonup \mathbf{V}^N \rightharpoonup \Sigma \rightharpoonup (\Sigma \times \mathbf{V}_u)$$

- Diese muss erweitert werden für Variablen:

$$Env = Id \rightharpoonup ([\![\mathbf{FunDef}]\!] \uplus \mathbf{Loc})$$

  - Insbesondere: gleicher Namensraum für Funktionen und Variablen (*C99 Standard*, §6.2.3)

Ausdrücke

Syntaktische Klasse von Ausdrücken, die eine Location bezeichnen (**Lexp**):

$$\begin{aligned}
\textbf{Lexp}\ l &::= Id \mid l\ [a] \mid l.Id \mid *a \\
\textbf{Aexp}\ a &::= \mathbf{N} \mid l \mid \&l \mid a_1 + a_2 \mid a_1 - a_2 \mid \\
&\quad a_1 * a_2 \mid a_1 / a_2 \mid Id(a^*) \\
\textbf{Bexp}\ b &::= \mathbf{0} \mid \mathbf{1} \mid a_1 == a_2 \mid a_1 != a_2 \mid \\
&\quad a_1 <= a_2 \mid !b \mid b_1 \,\&\&\, b2 \mid b_1 \,||\, b_2 \\
\textbf{Exp}\ e &:= a \mid b
\end{aligned}$$

Statements

$$\begin{aligned}
\textbf{Type} &::= \textbf{PointerType} \mid \textbf{BasicType} \mid \textbf{StructType} \mid \textbf{ArrayType} \\
\textbf{BasicType} &::= int \\
\textbf{StructType} &::= struct\ \textbf{name}\ \{\textbf{Puredecl}*\} \\
\textbf{ArrayType} &::= \textbf{Type}[\textbf{n}?] \\
\textbf{PointerType} &::= \textbf{Type}* \\
\textbf{ExtDecl} &::= \textbf{LogSpec} \mid \textbf{Decl} \mid \textbf{FunDef} \\
\textbf{Decl} &::= \textbf{Type}\ Id\ (=\ e|\{el\})?; \\
\textbf{FunDef} &::= \textbf{Type}\ Id(\textbf{PureDecl}^*)\ \textbf{FunSpec}^+\ \textbf{Blk} \\
\textbf{Blk} &::= \{\textbf{Decl}^*\ \textbf{Stmt}\} \\
\textbf{Puredecl} &::= \textbf{Type}\ Id \\
\textbf{Stmt} &::= \textbf{Lexp} = \textbf{Exp};\ |\ \textbf{if}\ (\ b\ )\ c_1\ \textbf{else}\ c_2 \\
&\quad |\ \textbf{while}\ (\ b\ )\ c\ |\{c^*\} \\
&\quad |Id(\textbf{AExp}*)\ |\ \textbf{return}\ (\textbf{AExp}?)
\end{aligned}$$

Erweiterung der Semantik: Lexp

$$\mathcal{L}[\![-]\!] : Env \to \textbf{Lexp} \to \Sigma \rightharpoonup \textbf{Loc}$$

$$\begin{aligned}
\mathcal{L}[\![x]\!]\ \Gamma &= \{(\sigma, \Gamma!x) \mid \sigma \in \Sigma\} \\
\mathcal{L}[\![lexp[a]]\!]\ \Gamma &= \{(\sigma, add(l, i \cdot sizeof(\tau))) \mid (\sigma, l) \in \mathcal{L}[\![lexp]\!]\ \Gamma, (\sigma, i) \in \mathcal{A}[\![a]\!]\ \Gamma\} \\
&\quad type(\Gamma, lexp) = \tau \text{ ist der Basistyp des Feldes} \\
\mathcal{L}[\![lexp.f]\!]\ \Gamma &= \{(\sigma, l.f) \mid (\sigma, add(l, fld_off(\tau, f))) \in \mathcal{L}[\![lexp]\!]\ \Gamma\} \\
&\quad type(\Gamma, lexp) = \tau \text{ ist der Typ der Struktur} \\
\mathcal{L}[\![*e]\!]\ \Gamma &= \mathcal{A}[\![e]\!]\ \Gamma
\end{aligned}$$

- $type(\Gamma, e)$ ist der Typ eines Ausdrucks
- $fld\_off(\tau, f)$ ist der Offset des Feldes f in der Struktur τ
- $sizeof(\tau)$ ist die Größe von Objekten des Typs τ

Erweiterung der Semantik: Aexp(1)

$$\mathcal{A}[\![-]\!] : \mathit{Env} \to \mathbf{Aexp} \to \Sigma \rightharpoonup \mathbf{V}$$

$$\begin{aligned}
\mathcal{A}[\![n]\!]\ \Gamma &= \{(\sigma, n) \mid \sigma \in \Sigma\} \quad \text{für } n \in \mathbf{N} \\
\mathcal{A}[\![e]\!]\ \Gamma &= \{(\sigma, \mathit{read}(\sigma, l)) \mid (\sigma, l) \in \mathcal{L}[\![e]\!]\ \Gamma\} \\
&\quad e \text{ is } \mathbf{LExp} \text{ und } \mathit{type}(\Gamma, e) \text{ kein Array-Typ} \\
\mathcal{A}[\![e]\!]\ \Gamma &= \{(\sigma, l) \mid (\sigma, l) \in \mathcal{L}[\![e]\!]\ \Gamma\} \\
&\quad e \text{ is } \mathbf{LExp} \text{ und } \mathit{type}(\Gamma, e) \text{ Array-Typ} \\
\mathcal{A}[\![\&e]\!]\ \Gamma &= \{(\sigma, l) \mid (\sigma, l) \in \mathcal{L}[\![e]\!]\ \Gamma\} \\
\mathcal{A}[\![p + e]\!]\ \Gamma &= \{(\sigma, \mathit{add}(l, n \cdot \mathit{sizeof}(\tau)))) \mid (\sigma, l) \in \mathcal{L}[\![p]\!]\ \Gamma \wedge (\sigma, n) \in \mathcal{A}[\![e]\!]\ \Gamma\} \\
&\quad \mathit{type}(\Gamma, p) = * \tau,\ \mathit{type}(\Gamma, e) \text{ Integer-Typ} \\
\mathcal{A}[\![e + p]\!]\ \Gamma &= \mathcal{A}[\![p + e]\!]\Gamma \\
&\quad \mathit{type}(\Gamma, e) \text{ Integer-Typ und } \mathit{type}(\Gamma, p) = * \tau
\end{aligned}$$

Erweiterung der Semantik: Aexp(2)

$$\mathcal{A}[\![-]\!] : \mathit{Env} \to \mathbf{Aexp} \to \Sigma \rightharpoonup \mathbf{V}$$

$$\begin{aligned}
\mathcal{A}[\![a_0 + a_1]\!]\ \Gamma &= \{(\sigma, n_0 + n_1 \mid (\sigma, n_0) \in \mathcal{A}[\![a_0]\!]\ \Gamma \wedge (\sigma, n_1) \in \mathcal{A}[\![a_1]\!]\ \Gamma\} \\
&\quad \text{für } a_0, a_1 \text{ arithmetische Typen} \\
\mathcal{A}[\![a_0 - a_1]\!]\ \Gamma &= \{(\sigma, n_0 - n_1) \mid (\sigma, n_0) \in \mathcal{A}[\![a_0]\!]\ \Gamma \wedge (\sigma, n_1) \in \mathcal{A}[\![a_1]\!]\ \Gamma\} \\
\mathcal{A}[\![a_0 * a_1]\!]\ \Gamma &= \{(\sigma, n_0 * n_1) \mid (\sigma, n_0) \in \mathcal{A}[\![a_0]\!]\ \Gamma \wedge (\sigma, n_1) \in \mathcal{A}[\![a_1]\!]\ \Gamma\} \\
\mathcal{A}[\![a_0 / a_1]\!]\ \Gamma &= \{(\sigma, n_0 / n_1) \mid (\sigma, n_0) \in \mathcal{A}[\![a_0]\!]\ \Gamma \wedge (\sigma, n_1) \in \mathcal{A}[\![a_1]\!]\ \Gamma \\
&\quad \wedge n_1 \neq 0\}
\end{aligned}$$

Explizite Zustandsprädikate

- Erweiterung der **Aexp** um *read*, neue Sorte **St** mit Operation *upd*:

$$\begin{aligned}
\mathbf{Bexp} &::= \ldots \text{(wie vorher)} \\
\mathbf{Aexp} &::= \mathit{read}(\mathbf{St}, \mathbf{Lexp}) \mid \mathbf{N} \mid \mathbf{Lexp} \mid \&\mathbf{Lexp} \mid \ldots \mid \backslash\mathbf{old}(e) \mid \ldots \\
\mathbf{St} &::= \mathit{StateVar} \mid \mathit{upd}(\mathbf{St}, \mathbf{Aexp}, \mathbf{Bexp})
\end{aligned}$$

- Zustandsvariablen *StateVar*: Aktueller Zustand σ, Vorzustand ρ
- Damit Semantik:

$$\begin{aligned}
\mathcal{B}_{sp}[\![.]\!] &: \mathit{Env} \to \mathbf{Bexp} \rightharpoonup (\Sigma \times (\Sigma \times \mathbf{V}_U)) \to \mathbf{T} \\
\mathcal{A}_{sp}[\![.]\!] &: \mathit{Env} \to \mathbf{Aexp} \rightharpoonup (\Sigma \times (\Sigma \times \mathbf{V}_U)) \to \mathbf{V}
\end{aligned}$$

- Explizite Zustandsprädikate enthalten kein $*$ oder &

Hoare-Triple

$$\Gamma \models \{P\}\, c\, \{Q|R\}$$

- P, Q, R sind explizite Zustandsprädikate
- Deklarationen (**Decl**) allozieren für jede Variable eine Location, und ordnen diese in der Umgebung zu.
- Restriktion: keine dynamische Allokation von Variablen (malloc und Freunde)
- Gültigkeit wie vorher

Floyd-Hoare-Kalkül mit expliziten Zustandsprädikaten

$$\overline{\Gamma \vdash \{Q[\mathit{upd}(\sigma, x, e)/\sigma]\}\, x = e\, \{Q|R\}}$$

- Ein **Lexp** l auf der rechten Seite e wird durch $\mathit{read}(\sigma, l)$ ersetzt.[1]
- & dient lediglich dazu, diese Konversion zu verhindern.
- $*$ erzwingt diese Konversion, auch auf der linken Seite x.
- Beispiel: $*a = *\&b;$.

[1] Außer l ist ein Array-Typ.

Formal: Konversion in Zustandsprädikate

$$\begin{aligned}
(-)^{\dagger} &: \mathbf{Lexp} \to \mathbf{Lexp} \\
v^{\dagger} &= v \quad (v \text{ Variable}) \\
l.id^{\dagger} &= l^{\dagger}.id \\
l[e]^{\dagger} &= l^{\dagger}[e^{\#}] \\
*l^{\dagger} &= l^{\#}
\end{aligned}$$

$$\begin{aligned}
(-)^{\#} &: \mathbf{Aexp} \to \mathbf{Aexp} \\
e^{\#} &= \mathit{read}(\sigma, e^{\dagger}) \quad (e \in \mathbf{Lexp}) \\
n^{\#} &= n \\
v^{\#} &= v \quad (v \text{ logische Variable}) \\
\&e^{\#} &= e^{\dagger} \\
e_1 + e_2{}^{\#} &= e_1{}^{\#} + e_2{}^{\#} \\
\backslash\mathbf{result}^{\#} &= \backslash\mathbf{result} \\
\backslash\mathbf{old}(e)^{\#} &= \backslash\mathbf{old}(e)
\end{aligned}$$

$$\overline{\Gamma \vdash \{Q[\mathit{upd}(\sigma, x^{\dagger}, e^{\#})/\sigma]\}\, x = e\, \{Q|R\}}$$

Zwei kurze Beispiele

```
void foo(){
 int x, y, z;
 /** { True } */
 z= x;
 x= 0;
 z= 5;
 y= x;
 /** { y == 0 } */
}
```

```
void foo(){
 int x, y, *z;
 /** { True } */
 z= &x;
 x= 0;
 *z= 5;
 y= x;
 /** { y == 5 } */
```


Weiteres Beispiel: Strukturen

```
struct Point {
  int x;
  int y;
 };
struct Point a = { 1, 2};
struct Point b;
b.x = a.x;
b.y = a.y;
{ b.x == a. x }
```

Weitere Beispiele: Felder

```
#include <limits.h>
#define N 10
int a[N];
int findmax()
  /** post \forall int i; 0 <= i && i < 10
                          ==> a[i] <= \result; */
{
  int x; int j;

  x= INT_MIN; j= 0;
  while (j< N) {
    if (a[j]> x) x= a[j];
    j= j+1;
    }
  return x;
}
```

Voller Beweis auf der Webseite (Quellen, `findmax-annotated.c`)

Felder und Zeiger revisited

- In C sind Zeiger und Felder schwach spezifiziert
- Insbesondere:
  - a[j]= *(a+j) für a Array-Typ
  - Derefenzierung von *x nur definiert, wenn x "gültig" ist (d.h. auf ein Objekt zeigt) *C99 Standard*, §6.5.3.2(4)
- Bisher in den Hoare-Regeln ignoriert — partielle Korrektheit.
- Ist das sinnvoll? Nein, bekannte Fehlerquelle

Spezifikation von Zeigern und Feldern

Das Prädikat \valid(x)

\valid($*x$) für x Pointer-Typ $\iff$ $*x$ ist definiert.

- Felder als Parameter werden Zeigern konvertiert, deshalb müssen wir spezifizieren können, dass ein Zeiger "in Wirklichkeit" ein Feld ist.
- **\array**(a, n) bedeutet: a ist ein Feld der Länge n, d.h.

$$\textbf{\array}(a, n) \iff (\forall i.\, 0 \leq i < n \implies \textbf{\valid}(a[i]))$$

- Validität kann abgeleitet werden:

$$\frac{\texttt{x= \&e}}{\texttt{\valid(*x)}} \qquad \frac{\texttt{\array(a, n)} \quad 0 \leq i \quad i < n}{\texttt{\valid(a[i])}}$$

Zusammenfassung

- Um Referenzen (Pointer) in C behandeln zu können, benötigen wir ein Zustandsmodell
- Referenzen werden zu Werten wie Zahlen oder Zeichen.
  - Arrays und Strukturen sind keine first-class values.
  - Großes Problem: aliasing
- Erweiterung der Semantik und der Hoare-Tripel nötig:
  - Vor/Nachbedingungen werden zu expliziten Zustandsprädikaten.
  - Zuweisung wird zu Zustandsupdate.
  - Problem:
    - Zustände werden sehr groß
    - Rückwärtsrechnung erzeugt schnell sehr große „unbestimmte" Zustände, die nicht vereinfacht werden können
  - Daher: Verifikationsbedingungen berechnen

Korrekte Software: Grundlagen und Methoden
Vorlesung 10 vom 12.06.17: Verifikationsbedingungen Revisited

Serge Autexier, Christoph Lüth

Universität Bremen

Sommersemester 2017

Fahrplan

Heute

- Der Hoare-Kalkül ist viel Schreibarbeit
- Deshalb haben wir Verifikationsbedingungen berechnet:
  - Approximative schwächste Vorbedingung
  - Approximative stärkste Nachbedingung
- Mit Zeigern ist rückwärts nicht das beste ...

Formal: Konversion in Zustandsprädikate

$$(-)^\dagger : \mathbf{Lexp} \to \mathbf{Lexp}$$

$$v^\dagger = v \quad (v \text{ Variable})$$

$$I.id^\dagger = I^\dagger.id$$

$$I[e]^\dagger = I^\dagger[e^\#]$$

$$*I^\dagger = I^\#$$

$$(-)^\# : \mathbf{Aexp} \to \mathbf{Aexp}$$

$$e^\# = read(\sigma, e^\dagger) \quad (e \in \mathbf{Lexp})$$

$$n^\# = n$$

$$v^\# = v \quad (v \text{ logische Variable})$$

$$\&e^\# = e^\dagger$$

$$e_1 + e_2{}^\# = e_1{}^\# + e_2{}^\#$$

$$\backslash\mathbf{result}^\# = \backslash\mathbf{result}$$

$$\backslash\mathbf{old}(e)^\# = \backslash\mathbf{old}(e)$$

$$\frac{}{\Gamma \vdash \{Q[upd(\sigma, x^\dagger, e^\#)/\sigma]\}\, x = e\, \{Q|R\}}$$

Approximative schwächste Vorbedingung

- Für die Berechnung der approximativen schwächsten Vorbedingung (AWP) und der Verifikationsbedingungen (WVC) müssen zwei Anpassungen vorgenommen werden:
  - Sowohl AWP als auch WVC berechnen symbolische Zustandsprädikate.
  - Die Zuweisungsregel muss angepasst werden.
- Berechnung von awp und wvc:

$$\begin{aligned}
&\mathrm{awp}(\Gamma, f(x_1,\ldots,x_n)/{*}{*}\ \mathbf{pre}\ P\ \mathbf{post}\ Q\ {*}/\ \{ds\ blk\}) \\
&\quad \stackrel{def}{=} \mathrm{awp}(\Gamma', blk, Q^\#, Q^\#) \\
&\mathrm{wvc}(\Gamma, f(x_1,\ldots,x_n)/{*}{*}\ \mathbf{pre}\ P\ \mathbf{post}\ Q\ {*}/\ \{ds\ blk\}) \\
&\quad \stackrel{def}{=} \{P^\# \Longrightarrow \mathrm{awp}(\Gamma', blk, Q^\#, Q^\#)[e_j{}^\#/\backslash\mathbf{old}(e_j)]\} \\
&\qquad \cup \mathrm{wvc}(\Gamma', blk, Q^\#, Q^\#) \\
&\Gamma' \stackrel{def}{=} \Gamma[f \mapsto \forall x_1,\ldots,x_n.\,(P,Q)]
\end{aligned}$$

Approximative schwächste Vorbedingung (Revisited)

$$\begin{aligned}
\mathrm{awp}(\Gamma, \{\}, Q, Q_R) &\stackrel{def}{=} Q \\
\mathrm{awp}(\Gamma, l = f(e_1,\ldots,e_n), Q, Q_R) &\stackrel{def}{=} P[e_i/x_i]^\# \\
&\quad \text{mit } \Gamma(f) = \forall x_1,\ldots,x_n.\,(P,R) \\
\mathrm{awp}(\Gamma, f(e_1,\ldots,e_n), Q, Q_R) &\stackrel{def}{=} P[e_i/x_i]^\# \\
&\quad \text{mit } \Gamma(f) = \forall x_1,\ldots,x_n.\,(P,R) \\
\mathrm{awp}(\Gamma, l = e, Q, Q_R) &\stackrel{def}{=} Q[upd(\sigma, l^\dagger, e^\#)/\sigma] \\
\mathrm{awp}(\Gamma, \{c\ c_s\}, Q, Q_R) &\stackrel{def}{=} \mathrm{awp}(\Gamma, c, \mathrm{awp}(\{c_s\}, Q, Q_R), Q_R) \\
\mathrm{awp}(\Gamma, \mathbf{if}\ (b)\ \{c_0\}\ \mathbf{else}\ \{c_1\}, Q, Q_R) &\stackrel{def}{=} (\ b^\#\ \&\&\ \mathrm{awp}(\Gamma, c_0, Q, Q_R)) \\
&\quad \|\ (\ !\,b^\#\ \&\&\ \mathrm{awp}(\Gamma, c_1, Q, Q_R)) \\
\mathrm{awp}(\Gamma, /{*}{*}\ \{q\}\ {*}/, Q, Q_R) &\stackrel{def}{=} q \\
\mathrm{awp}(\Gamma, \begin{bmatrix} \mathbf{while}\ (b) \\ /{*}{*}\ \mathbf{inv}\ i\ {*}/ \\ c \end{bmatrix}, Q, Q_R) &\stackrel{def}{=} i \\
\mathrm{awp}(\Gamma, \mathbf{return}\ e, Q, Q_R) &\stackrel{def}{=} Q_R[e^\#/\backslash\mathbf{result}] \\
\mathrm{awp}(\Gamma, \mathbf{return}, Q, Q_R) &\stackrel{def}{=} Q_R
\end{aligned}$$

Approximative Verifikationsbedingungen (Revisited)

$$\begin{aligned}
\mathrm{wvc}(\Gamma, \{\}, Q, Q_R) &\stackrel{def}{=} \emptyset \\
\mathrm{wvc}(\Gamma, x = e, Q, Q_R) &\stackrel{def}{=} \emptyset \\
\mathrm{wvc}(\Gamma, x = f(e_1,\ldots,e_n), Q, Q_R) &\stackrel{def}{=} \{\ (R[e_i/x_i][x/\backslash\mathbf{result}])^\# \Longrightarrow Q\} \\
&\quad \text{mit } \Gamma(f) = \forall x_1,\ldots,x_n.\,(P,R) \\
\mathrm{wvc}(\Gamma, f(e_1,\ldots,e_n), Q, Q_R) &\stackrel{def}{=} \{\ (R[e_i/x_i])^\# \Longrightarrow Q\} \\
&\quad \text{mit } \Gamma(f) = \forall x_1,\ldots,x_n.\,(P,R) \\
\mathrm{wvc}(\Gamma, \{c\ c_s\}, Q, Q_R) &\stackrel{def}{=} \mathrm{wvc}(\Gamma, c, \mathrm{awp}(\Gamma, \{c_s\}, Q, Q_R), Q_R) \\
&\quad \cup\ \mathrm{wvc}(\Gamma, \{c_s\}, Q, Q_R) \\
\mathrm{wvc}(\Gamma, \mathbf{if}\ (b)\ c_0\ \mathbf{else}\ c_1, Q, Q_R) &\stackrel{def}{=} \mathrm{wvc}(\Gamma, c_0, Q, Q_R) \\
&\quad \cup\ \mathrm{wvc}(\Gamma, c_1, Q, Q_R) \\
\mathrm{wvc}(\Gamma, /{*}{*}\ \{q\}\ {*}/, Q, Q_R) &\stackrel{def}{=} \{q \Longrightarrow Q\} \\
\mathrm{wvc}(\Gamma, \mathbf{while}\ (b)\ /{*}{*}\ \mathbf{inv}\ i\ {*}/\ c, Q, Q_R) &\stackrel{def}{=} \mathrm{wvc}(\Gamma, c, i) \\
&\quad \cup \{i \wedge b^\# \Longrightarrow \mathrm{awp}(\Gamma, c, i, Q_R)\} \\
&\quad \cup \{i \wedge\ !\,b^\# \Longrightarrow Q\} \\
\mathrm{wvc}(\Gamma, \mathbf{return}\ e, Q, Q_R) &\stackrel{def}{=} \emptyset
\end{aligned}$$

Beispiel: swap

```
void swap (int *x, int *y)
/** pre  \valid(*x);
    pre  \valid(*x); */
/** post \old(*x) == *y
         && \old(*y) == *x; */
{
 int z;

 z= *x;
 *x= *y;
 *y= z;
}
```

swap I

```
void swap (int *x, int *y)
/** pre  \valid(*x);
    pre  \valid(*x); */
/** post \old(*x) == *y
         && \old(*y) == x; */
{
 int z;

 z= *x;
 *x= *y;
 *y= z;
}

G = [swap |--> \forall x,y. (P = \valid(*x) && \valid(*y),
                             Q = \old(*x) == *y && \old(*y) == x)]

Q# = {\old(*x) == read(s, read(s, y)) && \old(*y) == read(s, read(s, x))}

*****************************************************************************
(A) awp(G, z = *x; *x = *y; *y = z, Q#, Q#)
= awp(G, z = *x, awp(G, *x = *y, awp(G, *y = z, Q#, Q#), Q#), Q#)
// Siehe Einzelberechnungen unten
= { \old(*x) == read(s, read(s, x)) && \old(*y) == read(s, read(s, y)) }


(A.1) awp(G, *y = z, Q#, Q#)
= { let s3=Upd(s2, read(s2, y), read(s2, z));
    in \old(*x) == read(s3, read(s3, y)) && \old(*y) == read(s3, read(s3, x)) }
Da: read(s3, y) = read(Upd(s2, read(s2, y), read(s2, z)), y)
                = read(s3, y) // da y != read(s2, y)
Also: read(s3, read(s3, y)) = read(s3, read(s2, y)) = read(s2, z)
```


swap II

```
= { let s3=Upd(s2, read(s2, y), read(s2, z));
    in \old(*x) == read(s2, z) && \old(*y) == read(s3, read(s3, x)) }
= Q1

(A.2) awp(G, *x = *y, Q1, Q#)
= { let s2=Upd(s1, read(s1, x), read(s1, read(s1, y));
    let s3=Upd(s2, read(s2, y), read(s2, z));
    in \old(*x) == read(s2, z) && \old(*y) == read(s3, read(s3, x)) }
Da: read(s3, x) = read(s2, x) // x != read(s2, y)
    read(s2, x) = read(s1, x) // x != read(s1, x)
    read(s2, z) = read(s1, z) // z != read(s1, x)

= { let s2=Upd(s1, read(s1, x), read(s1, read(s1, y));
    let s3=Upd(s2, read(s2, y), read(s2, z));
    in \old(*x) == read(s1, z) && \old(*y) == read(s3, read(s1, x)) }
= Q2

(A.3) awp(G, z = *x, Q2, Q#)
= { let s1=Upd(s, z, read(s, read(s, x)));
    let s2=Upd(s1, read(s1, x), read(s1, read(s1, y));
    let s3=Upd(s2, read(s2, y), read(s2, z));
    in \old(*x) == read(s1, z) && \old(*y) == read(s3, read(s1, x)) }
Da: read(s1, z) = read(s, read(s, x))
= { let s1=Upd(s, z, read(s, read(s, x)));
    let s2=Upd(s1, read(s1, x), read(s1, read(s1, y));
    let s3=Upd(s2, read(s2, y), read(s2, z));
    in \old(*x) == read(s, read(s, x)) && \old(*y) == read(s3, read(s1, x)) }
Es gilt: read(s2, read(s1, x)) = read(s1, read(s1, x))
```


swap III

```
** Fallunterscheidung **

1) read(s1, y) != read(s1, x):
Dann: read(s3, read(s1, x)) = read(s2, read(s1, x))
Also: read(s2, read(s1, x)) = read(s1, read(s1, y)
Folgt:
  = { let s1=Upd(s, z, read(s, read(s, x)));
    let s2=Upd(s1, read(s1, x), read(s1, read(s1, y));
    let s3=Upd(s2, read(s2, y), read(s2, z));
    in \old(*x) == read(s, read(s, x)) && \old(*y) == read(s1, read(s1, y)) }
Da ausserdem: read(s1, y) != (lokale Variable sind von aussen nicht sichtbar)
Folgt:
  = { let s1=Upd(s, z, read(s, read(s, x)));
    let s2=Upd(s1, read(s1, x), read(s1, read(s1, y));
    let s3=Upd(s2, read(s2, y), read(s2, z));
    in \old(*x) == read(s, read(s, x)) && \old(*y) == read(s, read(s, y)) }

2) read(s1, y) == read(s1, x):
Dann war auch: read(s, y) == read(s, x):
Dann: read(s2, read(s2, y)) = read(s1, read(s1, y)
Dann: read(s3, read(s1, x)) = read(s2, z)
                            = read(s, read(s, x))
                            = read(s, read(s, y))
Folgt (auch wie in 1)
  = { let s1=Upd(s, z, read(s, read(s, x)));
    let s2=Upd(s1, read(s1, x), read(s1, read(s1, y)));
    let s3=Upd(s2, read(s2, y), read(s2, z));
    in \old(*x) == read(s, read(s, x)) && \old(*y) == read(s, read(s, y)) }

*****************************************************************************
(B) wvc(G, swap) =
```


swap IV

```
      { P# ==> awp(G, z = *x; *x = *y; *y = z, Q#, Q#)[e_i/\old(e_i)] }
  U wvc(G,  z = *x; *x = *y; *y = z, Q#, Q#)
=   { P# ==> (\old(*x) == read(s, read(s, x)) && \old(*y) == read(s, read(s, y)))
             [read(s, read(s, x))/\old(*x), read(s, read(s, y))/\old(*y)] }
  U wvc(G,  z = *x; *x = *y; *y = z, Q#, Q#)
=   { P# ==> (read(s, read(s, x)) == read(s, read(s, x)) &&
              read(s, read(s, y)) == read(s, read(s, y))) }
  U wvc(G,  z = *x; *x = *y; *y = z, Q#, Q#)
= { True } U wvc(G,  z = *x; *x = *y; *y = z, Q#, Q#)
(Aus B.2 folgt)
= { True }

(B.1) P# = (\valid(*x) && \valid(*y))#
         = \valid(read(s, read(s, x))) && \valid(read(s, read(s, y)))

(B.2) wvc(G,  z = *x; *x = *y; *y = z, Q#, Q#)
=   wvc(G, z = *x, awp(G, *x = *y; *y = z,  Q#, Q#))
  U wvc(G, *x = *y, awp(G, *y = z,  Q#, Q#))
  U wvc(G, *y = z, awp(G, {}, Q#, Q#))
  U wvc(G, {}, Q#, Q#)
=   wvc(G, z = *x, awp(G, *x = *y; *y = z,  Q#, Q#)) [A.2]
  U wvc(G, *x = *y, awp(G, *y = z,  Q#, Q#)) [A.1]
  U wvc(G, *y = z, Q#)
  U {}
Durch (A.1), (A.2)
=   wvc(G, z = *x, Q2)
  U wvc(G, *x = *y, Q1)
  U wvc(G, *y = z, Q#)
= {}
```


Beispiel: `findmax` revisited

```
#include <limits.h>

int findmax(int a[], int a_len)
  /** pre  \array(a, a_len); */
  /** post \forall int i; 0 <= i && i < a_len
                          --> a[i] <= \result; */
{
  int x; int j;

  x= INT_MIN; j= 0;
  while (j< a_len)
    /* /\**  */ inv \forall int i; 0 <= i && i < j--> a[i]<= x && j <= 10; */
    {
    if (a[j]> x) x= a[j];
    j= j+1;
    }
  return x;
}
```


Fazit

- Der Hoare-Kalkül ist viel Schreibarbeitet
- Deshalb haben wir Verifikationsbedingungen berechnet:
  - Approximative schwächste Vorbedingung
- Als nächstes: Approximative stärkste Nachbedingung

Korrekte Software: Grundlagen und Methoden
Vorlesung 11 vom 19.06.17: Vorwärtsrechnung Revisited

Serge Autexier, Christoph Lüth

Universität Bremen

Sommersemester 2017

Fahrplan

Es geht Vorwärts.

- Verifikation nach dem Hoare-Kalkül mit Zeigern:
  - Viel Schreibarbeit.
- Berechnung von Verifikationsbedingungen:
  - Besser, aber:
  - Es entstehen viele "unbestimmte" Zwischenzustände, die nicht vereinfacht werden können.
- Daher heute Vorwärtsrechnung:
  - Die Vorwärtsregel nach Floyd (für explizite Zustandsprädikate)
  - Berechnung der stärksten Nachbedingung

Vorwärts?

- Wie kann eine Vorwärtsregel aussehen?
- Alt:

$$\frac{V \notin FV(P)}{\vdash \{P\}\, x = e\, \{\exists V.P[V/x] \,\&\&\, x = e[V/x]\}}$$

- Jetzt: Explizite Zustandsprädikate
- Nachbedingung: $\exists S.\, P[S/\sigma] \,\&\&\, \sigma = upd(S, x^\dagger, e^\#)$
  - S ist der Vorzustand
  - Aber: x und e müssen im Vorzustand S ausgewertet werden!
- Daher nötig: Zustand als zusätzlicher Parameter für $-^\dagger$ und $-^\#$

Formal: Konversion in Zustandsprädikate

$$(-)^\dagger_s : \mathbf{St} \to \mathbf{Lexp} \to \mathbf{Lexp} \qquad (-)^\#_s : \mathbf{St} \to \mathbf{Aexp} \to \mathbf{Aexp}$$

$$v^\dagger_s = v \quad (v \text{ Variable}) \qquad e^\#_s = read(s, e^\dagger_s) \quad (e \in \mathbf{Lexp})$$

$$l.id^\dagger_s = l^\dagger_s.id \qquad n^\#_s = n$$

$$l[e]^\dagger_s = l^\dagger_s[e^\#_s] \qquad v^\#_s = v \quad (v \text{ logische Variable})$$

$$*l^\dagger_s = l^\#_s \qquad \&e^\#_s = e^\dagger_s$$

$$(e_1 + e_2)^\#_s = e_1{}^\#_s + e_2{}^\#_s$$

$$\backslash\mathbf{result}^\#_s = \backslash\mathbf{result}$$

$$\backslash\mathbf{old}(e)^\#_s = e^\#_\rho$$

$$x^\dagger = x^\dagger_\sigma \qquad e^\# = e^\#_\sigma$$

- σ aktueller Zustand
- ρ initialer Zustand der gerade analysierten Funktion.

Vorwärts!

- Alternative Zuweisungsregel (nach Floyd):

$$\frac{S \notin FV(P)}{\vdash \{P\}\, x = e\, \{\exists S.\, P[S/\sigma] \,\&\&\, \sigma == upd(S, x^\dagger_S, e^\#_S)\}}$$

  - $FV(P)$ sind die freien Variablen in P.
- Jetzt ist die Vorbedingung offen — Regel kann vorwärts angewandt werden
- Gilt auch für die anderen Regeln

Das übliche Beispiel I

```
void foo(){
 int x, y, *z;
 /** { True }  */
 z= &x;
 /** \exists S. s= upd(s, z+_S, &x#_S) */
 /** \exists S. s= upd(S, z, x) */
 x= 0;
 /** \exists S'. (\exists S. s= upd(S, z, x))[S'/s] && \sigma= upd(S', x+_S', 0#_S') */
 /** \exists S'. (\exists S. S'= upd(S, z, x)) && \sigma= upd(S', x, 0) */
 /* With \exists x. x= t && P(x) <==> P(t) we get: */
 /** \exists S. \sigma= upd(upd(S, z, x), x, 0) */
 *z= 5;
 /** \exists S'. (\exists S. S'= upd(upd(S, z, x), x, 0)) && \sigma= upd(S', (*z)+_S', 5#_S') */
 /** \exists S. \sigma= upd(upd(upd(S, z, x), x, 0),
                           read(upd(upd(S, z, x), x, 0), z),    /* this rewrites to x *
                           5) */
 /** \exists S. \sigma= upd(upd(upd(S, z, x), x, 0), x, 5) */
 /** \exists S. \sigma= upd(upd(S, z, x), x, 5) */
 y= x;
 /** \exists S'. (\exists S. S'= upd(upd(S, z, x), x, 5)) && \sigma= upd(S', y+_S', x#_S') */
 /** \exists S. \sigma= upd(upd(upd(S, z, x), x, 5),
                           y,
                           read(upd(upd(S, z, x), x, 5), x)) /* this rewrites to 5 */
 /** \exists S. \sigma= upd(upd(upd(S, z, x), x, 5), y, 5) */
 /** \exists S. \sigma= upd(upd(upd(S, z, x), x, 5), y, 5) && read(\sigma, y) == 5 */
}
```


Berechnung der stärksten Nachbedingung

- Analog zur schwächsten Vorbedingung berechnen wir die approximative stärkste Nachbedingung $\mathrm{asp}(\Gamma, P, c)$ zusammen mit einer Menge von Verifikationsbedingungen $\mathrm{svc}(\Gamma, P, c)$
- Es gilt:

$$\bigwedge_{p_i \in \mathrm{svc}(\Gamma,P,c)} \forall \sigma.\, p_i \Longrightarrow \models \{P\}\, c\, \{\mathrm{asp}(\Gamma, P, c) | \, \mathrm{asp}(\Gamma, P, c)\}$$

- Zu beachten:
  - **return** beendet den Kontrollfluss
  - Bei Funktionsaufruf: Auswertung der Argumente im Vorzustand

Berechnung von *awp* und *wvc*

- Ausgehend von Spezifikation mit Vor- und Nachbedingung:

  $\mathrm{asp}(\Gamma, f(x_1,\ldots,x_n)/{**}\ \textbf{pre}\ P\ \textbf{post}\ Q\ {*}/\ \{ds\ blk\}) \stackrel{def}{=} \mathrm{asp}(\Gamma', P^{\#}, blk, Q^{\#})$

  $\mathrm{svc}(\Gamma, f(x_1,\ldots,x_n)/{**}\ \textbf{pre}\ P\ \textbf{post}\ Q\ {*}/\ \{ds\ blk\}) \stackrel{def}{=} \mathrm{svc}(\Gamma', P^{\#}, blk, Q^{\#})$

  $\Gamma' \stackrel{def}{=} \Gamma[f \mapsto \forall x_1,\ldots,x_n.\,(P,Q)]$

- Für diese Form muss jede Funktion mit einem **return** enden.
- Die Verifikationsbedingungen sind implizit über σ und ρ allquantifiziert

Approximative stärkste Nachbedingung

$$
\begin{array}{rcl}
\mathrm{asp}(\Gamma,P,\{\,\},Q) & \stackrel{def}{=} & P \\
\mathrm{asp}(\Gamma,P,x=e,Q) & \stackrel{def}{=} & \exists S.P[S/\sigma]\ \&\&\ \sigma == upd(S,x_S^{\dagger},e_S^{\#}) \\
\mathrm{asp}(\Gamma,P,\{\textbf{return}\ e;\ c_s\},Q) & \stackrel{def}{=} & P \\
\mathrm{asp}(\Gamma,P,\{\textbf{return};\ c_s\},Q) & \stackrel{def}{=} & P \\
\mathrm{asp}(\Gamma,P,\{c;\ c_s\},Q) & \stackrel{def}{=} & \mathrm{asp}(\Gamma,\mathrm{asp}(\Gamma,P,c,Q),c_s,Q) \\
\mathrm{asp}(\Gamma,P,\textbf{if}\ (b)\ c_0\ \textbf{else}\ c_1,P,Q) & \stackrel{def}{=} & \mathrm{asp}(\Gamma,b^{\#}\ \&\&\ P,c_0,Q) \\
 & & \|\ \mathrm{asp}(\Gamma,!(b^{\#})\ \&\&\ P,c_1,Q) \\
\mathrm{asp}(\Gamma,P,/{**}\ \{q\}\ {*}/,Q) & \stackrel{def}{=} & q^{\#} \\
\mathrm{asp}(\Gamma,P,\textbf{while}\ (b)\ /{**}\ \textbf{inv}\ i\ {*}/\ c,Q) & \stackrel{def}{=} & i^{\#}\ \&\&\ !\,b^{\#} \\
\mathrm{asp}(\Gamma,P,f(e_1,\ldots,e_n),Q) & \stackrel{def}{=} & \exists S.\,P[S/\sigma]\ \&\&\ R_2[(e_i)_S^{\#}/x_i] \\
\mathrm{asp}(\Gamma,P,l=f(e_1,\ldots,e_n),Q) & \stackrel{def}{=} & \exists S.\ \ P[S/\sigma]\ \&\& \\
 & & \quad R_2[(e_i)_S^{\#}/x_i][l_S^{\dagger}/\backslash\textbf{result}] \\
 & & \text{mit } \Gamma(f)=\forall x_1,\ldots,x_n.\,(R_1,R_2)
\end{array}
$$

Verifikationsbedingungen

$$
\begin{array}{rcl}
\mathrm{svc}(\Gamma,P,\{\,\},Q) & \stackrel{def}{=} & \emptyset \\
\mathrm{svc}(\Gamma,P,x=e,Q) & \stackrel{def}{=} & \emptyset \\
\mathrm{svc}(\Gamma,P,\{\textbf{return}\ e;\ c_s\},Q) & \stackrel{def}{=} & \{P \Longrightarrow Q[e^{\#}/\backslash\textbf{result}]\} \\
\mathrm{svc}(\Gamma,P,\{\textbf{return};\ c_s\},Q) & \stackrel{def}{=} & \{P \Longrightarrow Q\} \\
\mathrm{svc}(\Gamma,P,\{c\ c_s\},Q) & \stackrel{def}{=} & \mathrm{svc}(\Gamma,P,c,Q) \cup \\
 & & \mathrm{svc}(\Gamma,\mathrm{asp}(\Gamma,P,c,Q),\{c_s\},Q) \\
\mathrm{svc}(\Gamma,P,\textbf{if}\ (b)\ c_0\ \textbf{else}\ c_1,Q) & \stackrel{def}{=} & \mathrm{svc}(\Gamma,b^{\#}\ \&\&\ P,c_0,Q) \\
 & & \cup\,\mathrm{svc}(!(b^{\#})\ \&\&\ P,c_1,Q) \\
\mathrm{svc}(\Gamma,P,/{**}\ \{q\}\ {*}/,Q) & \stackrel{def}{=} & \{P \Longrightarrow q^{\#}\} \\
\mathrm{svc}(P,\textbf{while}\ b\ /{**}\ \textbf{inv}\ i\ {*}/\ c,Q) & \stackrel{def}{=} & \{P \Longrightarrow i^{\#}\} \\
 & & \cup\,\{\mathrm{asp}(\Gamma,b^{\#}\ \&\&\ i^{\#},c,Q) \Longrightarrow i^{\#}\} \\
 & & \cup\,\mathrm{svc}(\Gamma,b^{\#}\ \&\&\ i^{\#},c,Q) \\
\mathrm{svc}(\Gamma,P,f(e_1,\ldots,e_n),Q) & \stackrel{def}{=} & \{P \Longrightarrow R_1[e_i{}^{\#}/x_i]\} \\
\mathrm{svc}(\Gamma,P,l=f(e_1,\ldots,e_n),Q) & \stackrel{def}{=} & \{P \Longrightarrow R_1[e_i{}^{\#}/x_i][l^{\dagger}/\backslash\textbf{result}]\} \\
 & & \text{mit } \Gamma(f)=\forall x_1,\ldots,x_n.\,(R_1,R_2)
\end{array}
$$

Beispiel: `findmax` revisited

```
#include <limits.h>

int findmax(int a[], int a_len)
  /** pre  \array(a, a_len); */
  /** post \forall int i; 0 <= i && i < a_len
                          --> a[i] <= \result; */
{
  int x; int j;

  x= INT_MIN; j= 0;
  while (j< a_len)
    /* /\**  */ inv \forall int i; 0 <= i && i < j--> a[i]<= x &&
    {
    if (a[j]> x) x= a[j];
    j= j+1;
    }
  return x;
}
```

Korrekte Software: Grundlagen und Methoden
Vorlesung 12 vom 26.06.17: Programmsicherheit und Frame Conditions

Serge Autexier, Christoph Lüth

Universität Bremen

Sommersemester 2017

Fahrplan

- Einführung
- Die Floyd-Hoare-Logik
- Operationale Semantik
- Denotationale Semantik
- Äquivalenz der Operationalen und Denotationalen Semantik
- Korrektheit des Hoare-Kalküls
- Vorwärts und Rückwärts mit Floyd und Hoare
- Funktionen und Prozeduren
- Referenzen und Speichermodelle
- Verifikationsbedingungen Revisited
- Vorwärtsrechnung Revisited
- Programmsicherheit und Frame Conditions
- Ausblick und Rückblick

Heute im Angebot

- Programmsicherheit Revisited:
  - Deferenzierung von Pointer und Arrays
  - Division durch 0
- Frame Conditions
  - Was ist das und wozu braucht man das?
  - Wie könnte man das in unserer Sprache behandeln?
  - Modification sets

Zur Erinnerung: Totale Korrektheit

- Partielle Korrektheit: wenn das Programm terminiert, erfüllt es die Nachbedingung.

Wie sinnvoll ist diese Aussage?

Mein Programm wäre richtig gewesen, wenn es nicht vorher abgestürzt wäre.

- Wir wollen mindestens ausschließen, dass Laufzeitfehler ("undefined behaviour" *C99 Standard*, §3.4.3) auftreten.
- Problem: wenn Pointer als Parameter übergeben werden, müssen sie dereferenzierbar sein.
- Dazu neue Annotationen: \valid() und \array()

Spezifikation von Zeigern und Feldern

Die Prädikate \valid(x) und \array(a, n)

\valid(x) für x Pointer-Typ $\iff$ $*$x ist definiert.
\array(a, n) für a Pointer-Typ $\iff$ a ist ein Feld der Länge n.

- Abhängig vom Zustand (warum?)
- Felder als Parameter werden Zeigern konvertiert, deshalb müssen wir spezifizieren können, dass ein Zeiger "in Wirklichkeit" ein Feld ist.
- Formale Definition:

$$\valid(x, S) \stackrel{def}{=} \exists v.\, read(S, read(S, x)) = v$$
$$\array(a, n, S) \stackrel{def}{=} \forall i.\, 0 \leq i < n \Longrightarrow \valid(a[i], S)$$

Wie beweisen wir Validität?

- Konvention: $\valid(x) \stackrel{def}{=} \valid(x, \sigma)$
- Herleitung von Validität:

$$\frac{\text{x= \&e}}{\valid(x, S)} \qquad \frac{S1 = upd(S, x, y) \quad \text{mit } y \in \mathbf{Loc}, y \neq read(S, y')}{\valid(x, S1)}$$

$$\frac{\valid(x, S) \;\&\&\; S1 = upd(S, y, l) \;\&\&\; y \neq x}{\valid(x, S1)}$$

$$\frac{\valid(y, S) \;\&\&\; S1 = upd(S, x, read(S, y))}{\valid(x, S1)}$$

$$\frac{\array(a, n, S) \quad 0 \leq i \quad i < n}{\valid(a[i], S)}$$

Program Safety

- Hier: Dereferenzierungen definiert, keine Division durch 0
- Formal als Verifikationsbedingungen

$$\begin{aligned} safe(x, S) &= \emptyset \\ safe(*x, S) &= \{\valid(x, S)\} \\ safe(a[i]) &= \{\array(a, n) \;\&\&\; 0 <= i \;\&\&\; i < n\} \\ safe(x + y, S) &= safe(x, S) \cup safe(y, S) \\ safe(x/y, S) &= \{y\,!=0\} \\ &\ldots \end{aligned}$$

- Nicht ganz exakt: $y\,!=0$ muss im Zustand S ausgewertet werden,
  - D.h. $P \Longrightarrow y\,!=0$ mit P Vorbedingung
  - Dazu muss *safe* in die Definition von asp/awp eingebunden werden
- Valid-Analysen bleiben rein schematisch
- Division durch 0 und Arrayzugriffe benötigen Auswertung

Funktionsparameter und Frame Conditions

- Problem: Funktionen können beliebige Änderungen im Speicher vornehmen.

```
int x, y, z;

z= x+ y;
swap(&x, &y);
/** { z = \old(x)+ \old(y) } */
```

- Vor/Nach dem Funktionsaufruf (hier swap) muss die Nachbedingung/Vorbedingung noch gelten.

Frame Rule

- Konstanzregel (Rule of Constancy):

$$\frac{\vdash \{P\}\, c\, \{Q\}}{\vdash \{P \wedge R\}\, c\, \{Q \wedge R\}}$$

- Nebenbedingung: keine in c veränderte Variable tritt in R auf
- Problem: gilt mit Pointern nur eingeschränkt, da c eventuell ohne direkte Zuweisung Teile des Zustands verändert, über den R Aussagen macht.

Modification Sets

- Idee: Spezifiziere, welcher Teil des Zustands verändert werden darf.
  - ... denn wir können nicht spezifizieren, was gleich bleibt.
- Syntax: modifies **Mexp**

$$\textbf{Mexp} ::= \textbf{Loc} \mid \textbf{Mexp}\ [*] \mid \textbf{Mexp}\ [i : j] \mid \textbf{Mexp}\ .\ \textbf{name}$$

  - Mexp sind Lexp, die auch Teile von Feldern bezeichnen.
- Semantik: $[\![-]\!] : \mathit{Env} \to \textbf{Mexp} \to \Sigma \to \mathbb{P}(\textbf{Loc})$
- Modification Sets werden in die Hoare-Tripel integriert.

Erweiterung der Hoare-Tripel

- Hoare-Tripel mit Modification Sets:

$$\Lambda \models \{P\}\, c\, \{Q\} \iff \forall \sigma.\, P(\sigma) \wedge \exists \sigma'.\, \sigma' = c(\sigma) \implies Q(\sigma') \wedge \sigma \cong_\Lambda \sigma'$$

- wobei $\sigma \cong_L \tau$: Zustände σ und τ sind gleich bis auf die Adressen in L

$$\sigma \cong_L \tau \iff \forall l \in \mathit{dom}(\sigma) \cup \mathit{dom}(\tau) \setminus L.\, \sigma(l) = \tau(l)$$

bzw.

$$\sigma \cong_L \tau \iff \forall l.\, \sigma(l) \neq \tau(l) \implies l \in L$$

Erweiterung der Regeln

- Regeln werden mit Modification Set annotiert:

$$\Gamma, \Lambda \vdash \{P\}\, c\, \{Q_1 | Q_2\}$$

- Modification Set wird durchgereicht
- Zuweisungsregel wird ergänzt (vorwärts/rückwärts):

$$\frac{S \notin FV(P)}{\Gamma, \Lambda \vdash \{P\}\, x = e\, \{\exists S.\, P[S/\sigma]\ \&\&\ x_S^\dagger \in \Lambda\ \&\&\ \sigma == \mathit{upd}(S, x_S^\dagger, e_S^\#)\}}$$

$$\frac{}{\Gamma, \Lambda \vdash \{Q[\mathit{upd}(\sigma, x_S^\dagger)/\sigma]\ \&\&\ x_\sigma^\dagger \in \Lambda\}\, x = e\, \{Q\}}$$

Zusammenfassung

- Programmsicherheit kann als zusätzliche Verifikationsbedingung formuliert werden
  - Nachteil: teilweise komplexe Verifikationsbedingungen
  - Vorteil: semantische Integrität
- Frame Rule: spezifiziert unveränderte Teile des Zustands
  - Essentiell für Skalierbarkeit
  - Bei Zeigern rein syntaktische Analyse (freie Variablen) nicht ausreichend, daher modification sets
  - Spezifizieren veränderlichen Teil des Zustandes
  - Werden bei Zuweisungen geprüft

Korrekte Software: Grundlagen und Methoden
Vorlesung 13 vom 05.07.17: Rückblick & Ausblick

Serge Autexier, Christoph Lüth

Universität Bremen

Sommersemester 2017

Fahrplan

- Einführung
- Die Floyd-Hoare-Logik
- Operationale Semantik
- Denotationale Semantik
- Äquivalenz der Operationalen und Denotationalen Semantik
- Korrektheit des Hoare-Kalküls
- Vorwärts und Rückwärts mit Floyd und Hoare
- Funktionen und Prozeduren
- Referenzen und Speichermodelle
- Verifikationsbedingungen Revisited
- Vorwärtsrechnung Revisited
- Programmsicherheit und Frame Conditions
- Ausblick und Rückblick

Was gibt's heute?

- Rückblick
- Ausblick
- Feedback

Rückblick

Semantik

- Operational — Auswertungsrelation $\langle c, \sigma\rangle \to \sigma'$
- Denotational — Partielle Funktion $[\![c]\!] : \Sigma \rightharpoonup \Sigma$
- Axiomatisch — Floyd-Hoare-Logik
- Welche Semantik wofür?
- Beweis: Äquivalenz von operationaler und denotationaler Semantik

Floyd-Hoare-Logik

- Floyd-Hoare-Logik: partiell und total
- $\vdash \{P\}\, c\, \{Q\}$ vs. $\models \{P\}\, c\, \{Q\}$: Vollständigkeit, Korrektheit
- Die sechs Basisregeln
- Zuweisungsregel: vorwärts (Floyd) vs. rückwärts (Hoare)
- VCG: Schwächste Vorbedingung und stärkste Nachbedingung
- Beweis: Korrektheit und Vollständigkeit der Floyd-Hoare-Logik

Erweiterung der Programmiersprache

- Für jede Erweiterung:
  - Wie modellieren wir semantisch?
  - Wie ändern sich die Regeln der Logik?
- Prozeduren und Funktionen
  - Modellierung von **return**: Erweiterung zu $\Sigma \rightharpoonup \Sigma \times \mathbf{V}_U$
  - Spezifikation von Funktionen durch Vor-/Nachbedingungen
  - Spezifikation der Funktionen muss im Kontext stehen

Erweiterung der Programmiersprache

- Strukturen, Felder, Referenzen
  - Lokationen, **Lexp**, strukturierte Werte
  - Lokationen nicht mehr symbolisch (Variablennamen), sondern abstrakt $\Sigma = \mathbf{Loc} \rightharpoonup \mathbf{V}, \mathbf{V} = \mathbf{N} + \mathbf{C} + \mathbf{Loc}$
  - Zustand als abstrakter Datentyp mit Operationen *read* und *upd*
  - Zuweisung nicht mehr mit Substitution, sondern explizit durch *upd*
  - Spezifikationen sind explizite Zustandsprädikate
  - Konversionen $(-)^{\dagger}, (-)^{\#}$

Erweiterung der Programmiersprache

- Programmsicherheit
  - Keine Division durch 0
  - Keine illegale Dereferenzierung (einschließlich Felder)
  - Dazu: \valid
- Frame Conditions und Modification Sets
  - Frame Problem: welcher Teil des Zustands bleibt gleich?
  - Mit Zeigern: modification sets — Spezifikation des veränderlichen Teils

Ausblick

Die Sprache C: Was haben wir ausgelassen?

Semantik:

- Nichtdeterministische Semantik: Seiteneffekte, Sequence Points
  ⟶ Umständlich zu modellieren, Effekt zweitrangig
- Implementationsabhängiges, unspezifiziertes und undefiniertes Verhalten
  ⟶ Genauere Unterscheidung in der Semantik

Kontrollstrukturen:

- **switch** ⟶ Ist im allgemeinen Fall ein **goto**
- **goto**, setjmp/longjmp
  ⟶ Allgemeinfall: tiefe Änderung der Semantik (*continuations*)

Die Sprache C: Was haben wir ausgelassen?

Typen:

- Funktionszeiger ⟶ Für "saubere" Benutzung gut zu modellieren
- Weitere Typen: **short**/**long int**, **double**/**float**, wchar_t, und Typkonversionen ⟶ Fleißarbeit
- Fließkommazahlen ⟶ Spezifikation nicht einfach
- **union** ⟶ Kompliziert das Speichermodell
- **volatile** ⟶ Bricht read/update-Gleichungen
- **typedef** ⟶ Ärgernis für Lexer/Parser, sonst harmlos

Die Sprache C: Was haben wir ausgelassen?

Für realistische C-Programme:

- Compiler-Erweiterungen (`gcc`, `clang`)
- Büchereien (Standardbücherei, Posix, . . .)
- Nebenläufigkeit

Wie modelliert man Java?

- Die Kernsprache ist ähnlich zu C0.
- Java hat erschwerend:
  - dynamische Bindung,
  - Klassen mit gekapselten Zustand und Invarianten,
  - Nebenläufigkeit, und
  - Reflektion.
- Java hat dafür aber
  - ein einfacheres Speichermodell, und
  - eine wohldefinierte Ausführungsumgebung (die JVM).

Wie modelliert man C++?

- Sehr vorsichtig (konservativ)
- Viele Features, fehlende formale Semantik, . . .
- Mehrfachvererbung theoretisch anspruchsvoll
- Es gibt keine Formalismen/Werkzeuge, die C++ voll unterstützen
- Ansätze: Übersetzung nach C/LLVM, Behandlung dort

Wie modelliert man PHP?

Gar nicht.

Korrekte Software in der Industrie

- Meist in speziellen Anwendungsgebieten: Luft-/Raumfahrt, Automotive, sicherheitskritische Systeme, Betriebssysteme
- Ansätze:
  1. Vollautomatisch: statische Analyse (Abstrakte Interpretation) für spezielle Aspekte: Freiheit von Ausnahmen und Unter/Überläufen, Programmsicherheit, Laufzeitverhalten (WCET) (nicht immer korrekt, meist vollständig)
     - Werkzeuge: `absint`
  2. Halbautomatisch: Korrektheitsannotationen, Überprüfung automatisch
     - Werkzeuge: Spark (ADA), Frama-C (C), JML (ESC/Java, Krakatao; Java), Boogie und Why (generisches VCG), VCC (C)
  3. Interaktiv: Einbettung der Sprache in interaktiven Theorembeweiser (Isabelle, Coq)
     - Beispiele: L4.verified, CompCert, SAMS

Feedback

Deine Meinung zählt

- Was war gut, was nicht?
- Arbeitsaufwand?
- Mehr Theorie oder mehr Praxis?
- Mehr oder weniger Scala?

Tschüß!