

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:

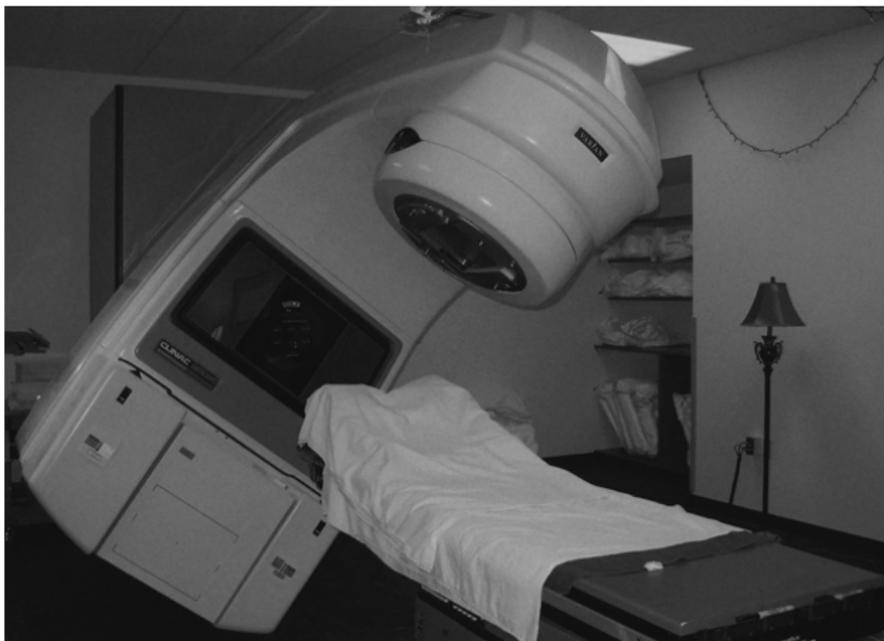
<http://www.informatik.uni-bremen.de/~cxl/lehre/ksgm.ss17>

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

- ▶ 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)
```

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)
- ▶ Werte, unveränderlich (val)
- ▶ while -Schleifen
- ▶ Rekursion
 - ▶ Endrekursion wird optimiert
- ▶ Typinferenz
 - ▶ Mehr als Java, weniger als Haskell
- ▶ Interaktive Auswertung

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

- ▶ Rekursion
 - ▶ Endrekursion wird optimiert
- ▶ Typinferenz
 - ▶ Mehr als Java, weniger als Haskell
- ▶ Interaktive Auswertung

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)  
}
```

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))))
```

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

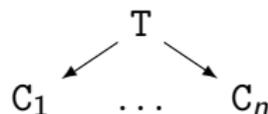
Implementierung algebraischer Datentypen

Haskell:

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

- ▶ Ein Typ T
- ▶ Konstruktoren erzeugen Datentyp

Scala:



- ▶ Varianten als **Subtypen**
- ▶ Problem und Vorteil:

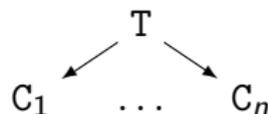
Implementierung algebraischer Datentypen

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

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`

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 $Z \rightarrow X \subseteq Z \rightarrow Y$, aber $X \rightarrow Z \not\subseteq Y \rightarrow Z$
 - ▶ Sondern $Y \rightarrow Z \subseteq X \rightarrow Z$

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 (Aspektororientierung)
- ▶ Nützlich zur Strukturierung:

thin interface + trait = 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
```

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

- ▶ 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

Die Übung am Donnerstag, 13.04.17, muss leider ausfallen!

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

Floyd-Hoare-Logik: Idee

- ▶ Was wird hier berechnet?

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

Floyd-Hoare-Logik: Idee

- ▶ Was wird hier berechnet? $p = n!$
- ▶ Wie können wir das **beweisen**?

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

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!}
```

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!}
```

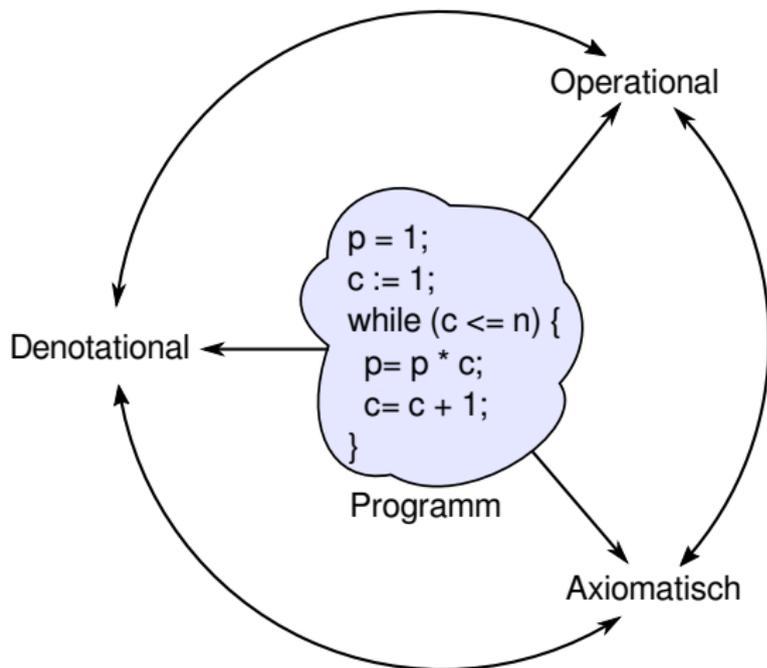
- ▶ Um Aussagen über ein Program zu beweisen, benötigen wir einen Formalismus (eine **Logik**), die es erlaubt, Zusicherungen über Werte von Variablen zu bestimmten Ausführungszeitpunkten (im Programm) **aufzuschreiben** und zu **beweisen**.
- ▶ Dazu müssen wir auch die **Bedeutung** (**Semantik**) des 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 eine partielle Funktion zwischen Systemzuständen) ab.
- ▶ **Axiomatische Semantik** beschreibt die Bedeutung eines Programmes durch Beweisregeln, mit denen 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 Programm.
- ▶ 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 ein wesentliches **Prinzip** und 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 (`==`, `!=`, `<=`, ...), boolesche 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 $b ::= \mathbf{0} \mid \mathbf{1} \mid a_1 == a_2 \mid a_1 != a_2$
 $\mid a_1 <= a_2 \mid !b \mid b_1 \&\& b_2 \mid b_1 \parallel b_2$

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

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

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 Zustände 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

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

- ▶ Für jedes Konstrukt der Programmiersprache gibt es eine Regel.

Regeln der Floyd-Hoare-Logik: Zuweisung

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

- ▶ 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:

$$x = 5 \\ \{x < 10\}$$

$$x = x + 1$$

Regeln der Floyd-Hoare-Logik: Zuweisung

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

- ▶ 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:

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

$$x = 5$$

$$\{x < 10\}$$

$$x = x + 1$$

Regeln der Floyd-Hoare-Logik: Zuweisung

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

- ▶ 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:

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

$$x = 5$$

$$\{x < 10\}$$

$$x = x + 1$$

$$\{x < 10\}$$

Regeln der Floyd-Hoare-Logik: Zuweisung

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

- ▶ 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

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

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

$$\frac{\vdash \{A\} \ c \ \{B\} \quad \vdash \{B\} \ \{c_s\} \ \{C\}}{\vdash \{A\} \ \{c \ c_s\} \ \{C\}}$$

- ▶ Hier wird ein Zwischenzustand B benötigt.

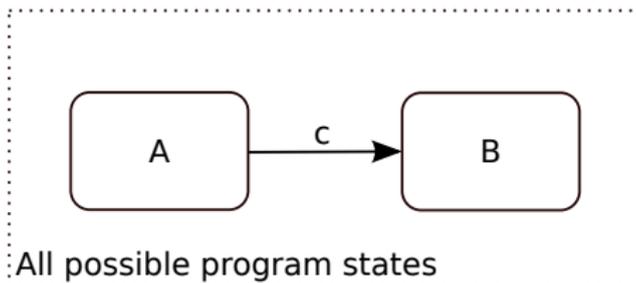
Regeln der Floyd-Hoare-Logik: Iteration

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

- ▶ 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

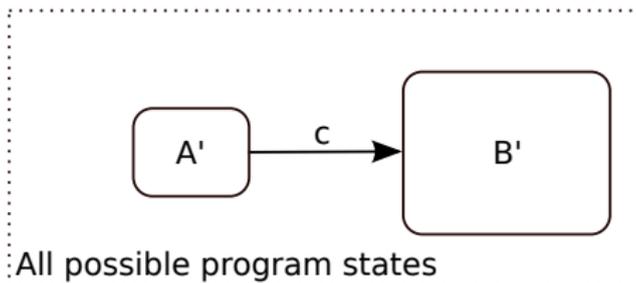
$$\frac{A' \implies A \quad \vdash \{A\} c \{B\} \quad B \implies B'}{\vdash \{A'\} c \{B'\}}$$



- ▶ $\models \{A\} c \{B\}$: 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. $P \implies Q$.

Regeln der Floyd-Hoare-Logik: Weakening

$$\frac{A' \implies A \quad \vdash \{A\} c \{B\} \quad B \implies B'}{\vdash \{A'\} c \{B'\}}$$



- ▶ $\models \{A\} c \{B\}$: 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. $P \implies Q$.
- ▶ Wir können A zu A' einschränken ($A' \subseteq A$ oder $A' \implies A$), oder B zu B' vergrößern ($B \subseteq B'$ oder $B \implies B'$), und erhalten $\models \{A'\} c \{B'\}$.

Überblick: die Regeln des Floyd-Hoare-Kalküls

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

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

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

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

$$\frac{A' \implies A \quad \vdash \{A\} c \{B\} \quad B \implies B'}{\vdash \{A'\} c \{B'\}}$$

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 bewiesen werden kann (aber sie stimmt), dann liegt das immer daran, dass eine **logische Aussage** (in einer Anwendung der Weakening-Regel) 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: $P \implies P_1$,
 $P_2 \implies P_3$, $P_3 \wedge x < n \implies P_4$,
 $P_3 \wedge \neg(x < n) \implies Q$.

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

```
// {x = 2Y}
```

```
// {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;
}
```

```
// {i2 ≤ a ∧ a < (i + 1)2}
```

Eine Handvoll Beispiele

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

```
// {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;
}
// {i2 ≤ 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 boolesche Ausdrücke, angereichert durch logische Variablen und Programmvariablen.
 - ▶ Wir können partielle Korrektheitsaussagen 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

- ▶ 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

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

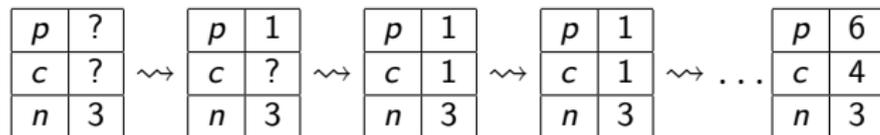
Wir betrachten einen Ausschnitt der Programmiersprache C (C0).

Ausbaustufe 1 kennt folgende Konstrukte:

- ▶ Typen: **int**;
- ▶ Ausdrücke: Variablen, Literale (für ganze Zahlen), arithmetische Operatoren (für ganze Zahlen), Relationen (`==`, `!=`, `<=`, ...), boolesche 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: $\Sigma = \mathbf{Loc} \rightarrow \mathbf{V}$
- 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 $b ::= \mathbf{0} \mid \mathbf{1} \mid a_1 == a_2 \mid a_1 != a_2$
 $\mid a_1 <= a_2 \mid !b \mid b_1 \&\& b_2 \mid b_1 \parallel b_2$

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

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

Eine Handvoll Beispiele

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

```
// {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;
}
// {i2 ≤ 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)

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

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)

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

Regeln:

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

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)

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

Regeln:

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

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

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

Operationale Semantik: Arithmetische Ausdrücke

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

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

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

Operationale Semantik: Arithmetische Ausdrücke

- **Aexp** $a ::= \mathbf{N} \mid \mathbf{Loc} \mid a_1 + a_2 \mid a_1 - a_2 \mid a_1 * a_2 \mid a_1 / a_2$
 $\langle a, \sigma \rangle \rightarrow_{\text{Aexp}} n \mid \perp$

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

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

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

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

Operationale Semantik: Arithmetische Ausdrücke

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

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

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

Operationale Semantik: Arithmetische Ausdrücke

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

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

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

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

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

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

Beispiel-Ableitungen

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

$$\overline{\langle (X + Y) * (X - Y), \sigma \rangle} \rightarrow_{Aexp}$$

Beispiel-Ableitungen

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

$$\frac{\overline{\langle X + Y, \sigma \rangle \rightarrow_{Aexp}} \quad \overline{\langle X - Y, \sigma \rangle \rightarrow_{Aexp}}}{\overline{\langle (X + Y) * (X - Y), \sigma \rangle \rightarrow_{Aexp}}}$$

Beispiel-Ableitungen

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

$$\frac{\frac{\langle X, \sigma \rangle \rightarrow_{Aexp} 6}{\langle X + Y, \sigma \rangle \rightarrow_{Aexp}} \quad \frac{}{\langle X - Y, \sigma \rangle \rightarrow_{Aexp}}}{\langle (X + Y) * (X - Y), \sigma \rangle \rightarrow_{Aexp}}$$

Beispiel-Ableitungen

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

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

Beispiel-Ableitungen

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

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

Beispiel-Ableitungen

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

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

Beispiel-Ableitungen

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

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

Beispiel-Ableitungen

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

$$\frac{\langle X, \sigma \rangle \rightarrow_{Aexp} 6 \quad \langle Y, \sigma \rangle \rightarrow_{Aexp} 5}{\langle X + Y, \sigma \rangle \rightarrow_{Aexp} 11} \quad \frac{\langle X, \sigma \rangle \rightarrow_{Aexp} 6 \quad \langle Y, \sigma \rangle \rightarrow_{Aexp} 5}{\langle X - Y, \sigma \rangle \rightarrow_{Aexp} 1}$$

$$\langle (X + Y) * (X - Y), \sigma \rangle \rightarrow_{Aexp} 11$$

Beispiel-Ableitungen

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

$$\frac{\langle X, \sigma \rangle \rightarrow_{Aexp} 6 \quad \langle Y, \sigma \rangle \rightarrow_{Aexp} 5}{\langle X + Y, \sigma \rangle \rightarrow_{Aexp} 11} \quad \frac{\langle X, \sigma \rangle \rightarrow_{Aexp} 6 \quad \langle Y, \sigma \rangle \rightarrow_{Aexp} 5}{\langle X - Y, \sigma \rangle \rightarrow_{Aexp} 1}$$

$$\langle (X + Y) * (X - Y), \sigma \rangle \rightarrow_{Aexp} 11$$

$$\langle (X * X) - (Y * Y), \sigma \rangle \rightarrow_{Aexp}$$

Beispiel-Ableitungen

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

$$\frac{\langle X, \sigma \rangle \rightarrow_{Aexp} 6 \quad \langle Y, \sigma \rangle \rightarrow_{Aexp} 5}{\langle X + Y, \sigma \rangle \rightarrow_{Aexp} 11} \quad \frac{\langle X, \sigma \rangle \rightarrow_{Aexp} 6 \quad \langle Y, \sigma \rangle \rightarrow_{Aexp} 5}{\langle X - Y, \sigma \rangle \rightarrow_{Aexp} 1}$$

$$\langle (X + Y) * (X - Y), \sigma \rangle \rightarrow_{Aexp} 11$$

$$\frac{\langle X, \sigma \rangle \rightarrow_{Aexp} 6 \quad \langle X, \sigma \rangle \rightarrow_{Aexp} 6}{\langle X * X, \sigma \rangle \rightarrow_{Aexp} 36}$$

$$\langle (X * X) - (Y * Y), \sigma \rangle \rightarrow_{Aexp}$$

Beispiel-Ableitungen

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

$$\frac{\langle X, \sigma \rangle \rightarrow_{Aexp} 6 \quad \langle Y, \sigma \rangle \rightarrow_{Aexp} 5}{\langle X + Y, \sigma \rangle \rightarrow_{Aexp} 11} \quad \frac{\langle X, \sigma \rangle \rightarrow_{Aexp} 6 \quad \langle Y, \sigma \rangle \rightarrow_{Aexp} 5}{\langle X - Y, \sigma \rangle \rightarrow_{Aexp} 1}$$

$$\langle (X + Y) * (X - Y), \sigma \rangle \rightarrow_{Aexp} 11$$

$$\frac{\langle X, \sigma \rangle \rightarrow_{Aexp} 6 \quad \langle X, \sigma \rangle \rightarrow_{Aexp} 6}{\langle X * X, \sigma \rangle \rightarrow_{Aexp} 36} \quad \frac{\langle Y, \sigma \rangle \rightarrow_{Aexp} 5 \quad \langle Y, \sigma \rangle \rightarrow_{Aexp} 5}{\langle Y * Y, \sigma \rangle \rightarrow_{Aexp} 25}$$

$$\langle (X * X) - (Y * Y), \sigma \rangle \rightarrow_{Aexp}$$

Beispiel-Ableitungen

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

$$\frac{\langle X, \sigma \rangle \rightarrow_{Aexp} 6 \quad \langle Y, \sigma \rangle \rightarrow_{Aexp} 5}{\langle X + Y, \sigma \rangle \rightarrow_{Aexp} 11} \quad \frac{\langle X, \sigma \rangle \rightarrow_{Aexp} 6 \quad \langle Y, \sigma \rangle \rightarrow_{Aexp} 5}{\langle X - Y, \sigma \rangle \rightarrow_{Aexp} 1}$$

$$\langle (X + Y) * (X - Y), \sigma \rangle \rightarrow_{Aexp} 11$$

$$\frac{\langle X, \sigma \rangle \rightarrow_{Aexp} 6 \quad \langle X, \sigma \rangle \rightarrow_{Aexp} 6}{\langle X * X, \sigma \rangle \rightarrow_{Aexp} 36} \quad \frac{\langle Y, \sigma \rangle \rightarrow_{Aexp} 5 \quad \langle Y, \sigma \rangle \rightarrow_{Aexp} 5}{\langle Y * Y, \sigma \rangle \rightarrow_{Aexp} 25}$$

$$\langle (X * X) - (Y * Y), \sigma \rangle \rightarrow_{Aexp} 11$$

Operationale Semantik: Boolesche Ausdrücke

► **Bexp** $b ::= 0 \mid 1 \mid a_1 == a_2 \mid a_1 <= a_2 \mid !b \mid b_1 \&\& b_2 \mid b_1 \parallel b_2$

Regeln:

$$\langle b, \sigma \rangle \rightarrow_{Bexp} 1 \mid 0 \mid \perp$$

$$\overline{\langle \mathbf{1}, \sigma \rangle \rightarrow_{Bexp} \mathbf{1}}$$

$$\overline{\langle \mathbf{0}, \sigma \rangle \rightarrow_{Bexp} \mathbf{0}}$$

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

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

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

Operationale Semantik: Boolesche Ausdrücke

► **Bexp** $b ::= 0 \mid 1 \mid a_1 == a_2 \mid a_1 <= a_2 \mid !b \mid b_1 \&\& b_2 \mid b_1 \parallel b_2$

Regeln:

$$\langle b, \sigma \rangle \rightarrow_{Bexp} 1 \mid 0 \mid \perp$$

$$\frac{\langle b, \sigma \rangle \rightarrow_{Bexp} 1}{\langle !b, \sigma \rangle \rightarrow_{Bexp} 0}$$

$$\frac{\langle b, \sigma \rangle \rightarrow_{Bexp} 0}{\langle !b, \sigma \rangle \rightarrow_{Bexp} 1}$$

$$\frac{\langle b, \sigma \rangle \rightarrow_{Bexp} \perp}{\langle !b, \sigma \rangle \rightarrow_{Bexp} \perp}$$

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

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

Operationale Semantik: Boolesche Ausdrücke

► **Bexp** $b ::= 0 \mid 1 \mid a_1 == a_2 \mid a_1 <= a_2 \mid !b \mid b_1 \&\& b_2 \mid b_1 \parallel b_2$

Regeln:

$$\langle b, \sigma \rangle \rightarrow_{Bexp} 1 \mid 0 \mid \perp$$

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

wobei $t = 0$ wenn $t_1 = t_2 = 0$;

$t = 1$ wenn $t_1 = 1$ oder $(t_1 = 0$ und $t_2 = 1)$;

$t = \perp$ sonst

Operationale Semantik: Anweisungen

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

Beispiel:

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

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

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

Operationale Semantik: Anweisungen

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

Regeln:

Definiere :

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

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

Es gilt:

$$\begin{aligned} \forall \sigma, n, m, \forall X, Y . X \neq Y &\Rightarrow \sigma[n/X][m/Y] = \sigma[m/Y][n/X] \\ \forall \sigma, n, m, \forall X . \sigma[n/X][m/X] &= \sigma[m/X] \end{aligned}$$

Operationale Semantik: Anweisungen

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

Regeln:

$$\langle \{ \}, \sigma \rangle \rightarrow_{\text{Stmt}} \sigma$$

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

$$\frac{\langle a, \sigma \rangle \rightarrow_{\text{Aexp}} \perp}{\langle X = a, \sigma \rangle \rightarrow_{\text{Stmt}} \perp}$$

$$\frac{\langle c, \sigma \rangle \rightarrow_{\text{Stmt}} \sigma' \neq \perp \quad \langle \{c_s\}, \sigma' \rangle \rightarrow_{\text{Stmt}} \sigma'' \neq \perp}{\langle \{c \ c_s\}, \sigma \rangle \rightarrow_{\text{Stmt}} \sigma''}$$

$$\frac{\langle c, \sigma \rangle \rightarrow_{\text{Stmt}} \perp}{\langle \{c \ c_s\}, \sigma \rangle \rightarrow_{\text{Stmt}} \perp}$$

$$\frac{\langle c, \sigma \rangle \rightarrow_{\text{Stmt}} \sigma' \neq \perp \quad \langle \{c_s\}, \sigma' \rangle \rightarrow_{\text{Stmt}} \perp}{\langle \{c \ c_s\}, \sigma \rangle \rightarrow_{\text{Stmt}} \perp}$$

Operationale Semantik: Anweisungen

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

Regeln:

$$\langle \{ \}, \sigma \rangle \rightarrow_{\text{Stmt}} \sigma$$

$$\frac{\langle b, \sigma \rangle \rightarrow_{\text{Bexp}} 1 \quad \langle c_1, \sigma \rangle \rightarrow_{\text{Stmt}} \sigma'}{\langle \text{if} (b) c_1 \text{ else } c_2, \sigma \rangle \rightarrow_{\text{Stmt}} \sigma'}$$

$$\frac{\langle b, \sigma \rangle \rightarrow_{\text{Bexp}} 0 \quad \langle c_2, \sigma \rangle \rightarrow_{\text{Stmt}} \sigma'}{\langle \text{if} (b) c_1 \text{ else } c_2, \sigma \rangle \rightarrow_{\text{Stmt}} \sigma'}$$

$$\frac{\langle b, \sigma \rangle \rightarrow_{\text{Bexp}} \perp}{\langle \text{if} (b) c_1 \text{ else } c_2, \sigma \rangle \rightarrow_{\text{Stmt}} \perp}$$

Operationale Semantik: Anweisungen

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

Regeln:

$$\langle \{ \}, \sigma \rangle \rightarrow_{\text{Stmt}} \sigma$$

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

$$\frac{\langle b, \sigma \rangle \rightarrow_{\text{Bexp}} 1 \quad \langle c, \sigma \rangle \rightarrow_{\text{Stmt}} \sigma' \quad \langle \text{while} (b) c, \sigma' \rangle \rightarrow_{\text{Stmt}} \sigma''}{\langle \text{while} (b) c, \sigma \rangle \rightarrow_{\text{Stmt}} \sigma''}$$

$$\frac{\langle b, \sigma \rangle \rightarrow_{\text{Bexp}} 1 \quad \langle c, \sigma \rangle \rightarrow_{\text{Stmt}} \perp}{\langle \text{while} (b) c, \sigma \rangle \rightarrow_{\text{Stmt}} \perp}$$

$$\frac{\langle b, \sigma \rangle \rightarrow_{\text{Bexp}} \perp}{\langle \text{while} (b) c, \sigma \rangle \rightarrow_{\text{Stmt}} \perp}$$

Beispiel

```
x = 1;
while (y != 0) {
  y = y - 1;
  x = 2 * x;
}
// x = 2y
σ(y) = 3
```

Äquivalenz arithmetischer Ausdrücke

Gegeben zwei Aexp a_1 and a_2

- Sind sie gleich?

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

$$(X * X) + 2 * X * Y + (Y * Y) \quad \text{und} \quad (X + Y) * (X + Y)$$

- Wann sind sie gleich?

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

$$X * X \quad \text{und} \quad 9 * X + 22$$

$$X * X \quad \text{und} \quad X * X + 1$$

Äquivalenz Boolescher Ausdrücke

Gegeben zwei Bexp-Ausdrücke b_1 and b_2

- Sind sie gleich?

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

$A \ || \ (A \ \&\& \ B)$ und A

Beweisen

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

Definition

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

Ein einfaches Beispiel:

Lemma

Sei $w \equiv \mathbf{while} (b) c$ mit $b \in \mathbf{Bexp}$, $c \in \mathbf{Stmt}$.

Dann gilt: $w \sim \mathbf{if} (b) \{ c; w \} \mathbf{else} \{ \}$

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 $f : A \rightarrow A$ 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 $\frac{x_1 \dots x_n}{y}$, $n \geq 0$.

Die Anwendung einer Regel auf spezifische $a_1 \dots a_n$ ist eine Regelinstanz

- ▶ Betrachte folgende Regelmenge R

$$\frac{-}{2^2}$$

$$\frac{-}{2^3}$$

$$\frac{n \quad m}{n \cdot m}$$

- ▶ Regelinstanzen sind

$$\frac{-}{4}$$

$$\frac{-}{8}$$

$$\frac{4 \quad 8}{32}$$

$$\frac{4 \quad 4}{16}$$

$$\frac{16 \quad 32}{512}$$

$$\frac{3 \quad 5}{15}$$

...

Induktive Definierte Mengen

Definition

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

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

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

Beispiel

- ▶ Betrachte folgende Regelmenge R

$$\frac{-}{2^2}$$

$$\frac{-}{2^3}$$

$$\frac{n \quad m}{n \cdot m}$$

- ▶ 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

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

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

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

Definition (Abgeschlossen und Monoton)

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

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

Kleinsten 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 $\frac{x_1, \dots, x_k}{y} \in R$ und $x_1, \dots, x_k \subseteq A$.

Da $A = \bigcup_{i \in \mathbb{N}} A_i$ gibt es ein j so dass $x_1, \dots, x_k \subseteq A_j$.

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 $\frac{x_1, \dots, x_k}{y} \in R$ geben mit

$x_1, \dots, x_k \subseteq A_{n-1} \subseteq A$.

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.



Kleinsten Fixpunkt Operator

Definition

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

ist der **kleinste Fixpunkt**.

Kleinsten Fixpunkt

- ▶ Betrachte folgende Regelmenge R

$$\frac{-}{2^2}$$

$$\frac{-}{2^3}$$

$$\frac{n \quad m}{n \cdot m}$$

- ▶ Was sind

$$\hat{R}^1(\emptyset) = \hat{R}(\emptyset) = \{4, 8\}$$

$$\hat{R}^2(\emptyset) = ?$$

$$\hat{R}^3(\emptyset) = ?$$

$$\hat{R}^{i+1}(\emptyset) = ?$$

Kleinster Fixpunkt

- ▶ Betrachte folgende Regelmenge R

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

- ▶ Was sind

$$\hat{R}^1(\emptyset) = \hat{R}(\emptyset) = \{4, 8\}$$

$$\hat{R}^2(\emptyset) = ?$$

$$\hat{R}^3(\emptyset) = ?$$

$$\hat{R}^{i+1}(\emptyset) = ?$$

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

Denotationale Semantik - Motivation

▶ Operationale Semantik

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

$$\langle c, \sigma \rangle \rightarrow_{Stmnt} \sigma' \mid \perp$$

▶ Denotationale Semantik

Eine Menge von Regeln, die ein Programm in eine partielle Funktion von Zustand nach Zustand überführen

Denotat

$$\mathcal{C}[[c]] : \Sigma \rightarrow \Sigma$$

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**

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

Denotierte Funktionen

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

Denotat von **Aexp**

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

$$\mathcal{A}[[n]] = \{(\sigma, n) \mid \sigma \in \Sigma\}$$

$$\mathcal{A}[[x]] = \{(\sigma, \sigma(x)) \mid \sigma \in \Sigma, x \in \text{Dom}(\sigma)\}$$

$$\mathcal{A}[[a_0 + a_1]] = \{(\sigma, n_0 + n_1) \mid (\sigma, n_0) \in \mathcal{A}[[a_0]] \wedge (\sigma, n_1) \in \mathcal{A}[[a_1]]\}$$

$$\mathcal{A}[[a_0 - a_1]] = \{(\sigma, n_0 - n_1) \mid (\sigma, n_0) \in \mathcal{A}[[a_0]] \wedge (\sigma, n_1) \in \mathcal{A}[[a_1]]\}$$

$$\mathcal{A}[[a_0 * a_1]] = \{(\sigma, n_0 * n_1) \mid (\sigma, n_0) \in \mathcal{A}[[a_0]] \wedge (\sigma, n_1) \in \mathcal{A}[[a_1]]\}$$

$$\mathcal{A}[[a_0/a_1]] = \{(\sigma, n_0/n_1) \mid (\sigma, n_0) \in \mathcal{A}[[a_0]] \wedge (\sigma, n_1) \in \mathcal{A}[[a_1]] \wedge n_1 \neq 0\}$$

Denotat von **Bexp**

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

$$\mathcal{B}[[1]] = \{(\sigma, 1) \mid \sigma \in \Sigma\}$$

$$\mathcal{B}[[0]] = \{(\sigma, 0) \mid \sigma \in \Sigma\}$$

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

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

Denotat von **Bexp**

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

$$\begin{aligned} \mathcal{B}[[!b]] &= \{(\sigma, 1) \mid \sigma \in \Sigma, (\sigma, 0) \in \mathcal{B}[[b]]\} \\ &\cup \{(\sigma, 0) \mid \sigma \in \Sigma, (\sigma, 1) \in \mathcal{B}[[b]]\} \end{aligned}$$

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

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

Denotat von Stmt

$$\mathcal{C}[\cdot] : \mathbf{Stmt} \rightarrow (\Sigma \rightarrow \Sigma)$$

$$\mathcal{C}[x = a] = \{(\sigma, \sigma[n/x]) \mid \sigma \in \Sigma \wedge (\sigma, n) \in \mathcal{A}[a]\}$$

$$\mathcal{C}[\{c \ c_s\}] = \mathcal{C}[c_s] \circ \mathcal{C}[c] \quad \text{Komposition von Relationen}$$

$$\mathcal{C}[\{\}] = \mathbf{Id} \quad \mathbf{Id} := \{(\sigma, \sigma) \mid \sigma \in \Sigma\}$$

$$\begin{aligned} \mathcal{C}[\mathbf{if} (b) \ c_0 \ \mathbf{else} \ c_1] &= \{(\sigma, \sigma') \mid (\sigma, 1) \in \mathcal{B}[b] \wedge (\sigma, \sigma') \in \mathcal{C}[c_0]\} \\ &\cup \{(\sigma, \sigma') \mid (\sigma, 0) \in \mathcal{B}[b] \wedge (\sigma, \sigma') \in \mathcal{C}[c_1]\} \end{aligned}$$

Denotat von Stmt

$$\mathcal{C}[\cdot] : \mathbf{Stmt} \rightarrow (\Sigma \rightarrow \Sigma)$$

$$\mathcal{C}[x = a] = \{(\sigma, \sigma[n/x]) \mid \sigma \in \Sigma \wedge (\sigma, n) \in \mathcal{A}[a]\}$$

$$\mathcal{C}[\{c \ c_s\}] = \mathcal{C}[c_s] \circ \mathcal{C}[c] \quad \text{Komposition von Relationen}$$

$$\mathcal{C}[\{\}] = \mathbf{Id} \quad \mathbf{Id} := \{(\sigma, \sigma) \mid \sigma \in \Sigma\}$$

$$\begin{aligned} \mathcal{C}[\mathbf{if} (b) \ c_0 \ \mathbf{else} \ c_1] &= \{(\sigma, \sigma') \mid (\sigma, 1) \in \mathcal{B}[b] \wedge (\sigma, \sigma') \in \mathcal{C}[c_0]\} \\ &\cup \{(\sigma, \sigma') \mid (\sigma, 0) \in \mathcal{B}[b] \wedge (\sigma, \sigma') \in \mathcal{C}[c_1]\} \end{aligned}$$

Aber was ist

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

Denotationale Semantik für **while**

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

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

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

Denotationale Semantik für **while**

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

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

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

Denotationale Semantik für **while**

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

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

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

Denotationale Semantik von **while**

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

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

$$\mathcal{C}[[w]]_0 = \{(\sigma, \sigma) \mid (\sigma, 0) \in \mathcal{B}[[b]](\sigma)\}$$

Denotationale Semantik von **while**

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

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

$$\mathcal{C}[[w]]_0 = \{(\sigma, \sigma) \mid (\sigma, 0) \in \mathcal{B}[[b]](\sigma)\}$$

$$\mathcal{C}[[w]]_1 = \{(\sigma, \sigma') \mid \exists \sigma''. (\sigma, 1) \in \mathcal{B}[[b]] \wedge (\sigma, \sigma'') \in \mathcal{C}[[c]] \\ \wedge (\sigma'', \sigma') \in \mathcal{C}[[w]]_0\}$$

Denotationale Semantik von **while**

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

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

$$\mathcal{C}[[w]]_0 = \{(\sigma, \sigma) \mid (\sigma, 0) \in \mathcal{B}[[b]](\sigma)\}$$

$$\mathcal{C}[[w]]_1 = \{(\sigma, \sigma') \mid \exists \sigma''. (\sigma, 1) \in \mathcal{B}[[b]] \wedge (\sigma, \sigma'') \in \mathcal{C}[[c]] \\ \wedge (\sigma'', \sigma') \in \mathcal{C}[[w]]_0\}$$

$$\mathcal{C}[[w]]_2 = \{(\sigma, \sigma') \mid \exists \sigma''. (\sigma, 1) \in \mathcal{B}[[b]] \wedge (\sigma, \sigma'') \in \mathcal{C}[[c]] \\ \wedge (\sigma'', \sigma') \in \mathcal{C}[[w]]_1\}$$

\vdots

Denotationale Semantik von **while**

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

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

$$\mathcal{C}[[w]]_0 = \{(\sigma, \sigma) \mid (\sigma, 0) \in \mathcal{B}[[b]](\sigma)\}$$

$$\mathcal{C}[[w]]_1 = \{(\sigma, \sigma') \mid \exists \sigma''. (\sigma, 1) \in \mathcal{B}[[b]] \wedge (\sigma, \sigma'') \in \mathcal{C}[[c]] \\ \wedge (\sigma'', \sigma') \in \mathcal{C}[[w]]_0\}$$

$$\mathcal{C}[[w]]_2 = \{(\sigma, \sigma') \mid \exists \sigma''. (\sigma, 1) \in \mathcal{B}[[b]] \wedge (\sigma, \sigma'') \in \mathcal{C}[[c]] \\ \wedge (\sigma'', \sigma') \in \mathcal{C}[[w]]_1\}$$

\vdots

$$\mathcal{C}[[w]]_{i+1} = \{(\sigma, \sigma') \mid \exists \sigma''. (\sigma, 1) \in \mathcal{B}[[b]] \wedge (\sigma, \sigma'') \in \mathcal{C}[[c]] \\ \wedge (\sigma'', \sigma') \in \mathcal{C}[[w]]_i\}$$

Denotationale Semantik von **while**

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

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

$$\mathcal{C}[[w]]_0 = \{(\sigma, \sigma) \mid (\sigma, 0) \in \mathcal{B}[[b]](\sigma)\}$$

$$\mathcal{C}[[w]]_1 = \{(\sigma, \sigma') \mid \exists \sigma''. (\sigma, 1) \in \mathcal{B}[[b]] \wedge (\sigma, \sigma'') \in \mathcal{C}[[c]] \\ \wedge (\sigma'', \sigma') \in \mathcal{C}[[w]]_0\}$$

$$\mathcal{C}[[w]]_2 = \{(\sigma, \sigma') \mid \exists \sigma''. (\sigma, 1) \in \mathcal{B}[[b]] \wedge (\sigma, \sigma'') \in \mathcal{C}[[c]] \\ \wedge (\sigma'', \sigma') \in \mathcal{C}[[w]]_1\}$$

\vdots

$$\mathcal{C}[[w]]_{i+1} = \{(\sigma, \sigma') \mid \exists \sigma''. (\sigma, 1) \in \mathcal{B}[[b]] \wedge (\sigma, \sigma'') \in \mathcal{C}[[c]] \\ \wedge (\sigma'', \sigma') \in \mathcal{C}[[w]]_i\}$$

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

Denotationale Semantik von **while**

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

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

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

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

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

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

Denotation für **Stmt**

$$\mathcal{C}[\cdot] : \mathbf{Stmt} \rightarrow (\Sigma \rightarrow \Sigma)$$

$$\mathcal{C}[x = a] = \{(\sigma, \sigma[n/X]) \mid \sigma \in \Sigma \wedge (\sigma, n) \in \mathcal{A}[a]\}$$

$$\mathcal{C}[\{c \ c_s\}] = \mathcal{C}[c_s] \circ \mathcal{C}[c] \quad \text{Komposition von Relationen}$$

$$\mathcal{C}[\{\}] = \mathbf{Id} \quad \mathbf{Id} := \{(\sigma, \sigma) \mid \sigma \in \Sigma\}$$

$$\begin{aligned} \mathcal{C}[\mathbf{if} (b) \ c_0 \ \mathbf{else} \ c_1] &= \{(\sigma, \sigma') \mid (\sigma, 1) \in \mathcal{B}[b] \wedge (\sigma, \sigma') \in \mathcal{C}[c_0]\} \\ &\quad \cup \{(\sigma, \sigma') \mid (\sigma, 0) \in \mathcal{B}[b] \wedge (\sigma, \sigma') \in \mathcal{C}[c_1]\} \end{aligned}$$

$$\mathcal{C}[\mathbf{while} (b) \ c] = \mathit{fix}(\Gamma)$$

mit

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

Weitere Intuition zur Fixpunkt Konstruktion

- ▶ Sei $w \equiv \mathbf{while} (b) c$
- ▶ Zur Erinnerung: Wir haben begonnen mit $w \sim \mathbf{if} (b) \{ c w \} \mathbf{else} \{ \}$
- ▶ Dann müsste auch gelten

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

- ▶ 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

- ▶ 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

Operationale vs. denotationale Semantik

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

Denotational $\mathcal{A}[[a]]$

$$m \in \mathbf{N} \quad \langle m, \sigma \rangle \rightarrow_{Aexp} m$$

$$\{(\sigma, m) \mid \sigma \in \Sigma\}$$

$$x \in \mathbf{Loc} \quad \frac{x \in \text{Dom}(\sigma)}{\langle x, \sigma \rangle \rightarrow_{Aexp} \sigma(x)}$$

$$\{(\sigma, \sigma(x)) \mid \sigma \in \Sigma, x \in \text{Dom}(\sigma)\}$$

$$a_1 \circ a_2 \quad \frac{\begin{array}{l} \langle a_1, \sigma \rangle \rightarrow_{Aexp} n \\ \langle a_2, \sigma \rangle \rightarrow_{Aexp} m \\ n, m \neq \perp \end{array}}{\langle a_1 \circ a_2, \sigma \rangle \rightarrow_{Aexp} n \circ^l m}$$

$$\{(\sigma, n \circ^l m) \mid \sigma \in \Sigma, (\sigma, n) \in \mathcal{A}[[a_1]], (\sigma, m) \in \mathcal{A}[[a_2]]\}$$

$$\frac{\begin{array}{l} \langle a_1, \sigma \rangle \rightarrow_{Aexp} n \\ \langle a_2, \sigma \rangle \rightarrow_{Aexp} m \\ n = \perp \text{ oder } m = \perp \end{array}}{\langle a_1 \circ a_2, \sigma \rangle \rightarrow_{Aexp} \perp}$$

$\circ \in \{+, *, -\}$

Operationale vs. denotationale Semantik

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

$$a_1/a_2 \quad \frac{\begin{array}{l} \langle a_1, \sigma \rangle \rightarrow_{Aexp} n \\ \langle a_2, \sigma \rangle \rightarrow_{Aexp} m \\ m \neq 0 \quad m, n \neq \perp \end{array}}{\langle a_1 \circ a_2, \sigma \rangle \rightarrow_{Aexp} n \circ^l m}$$

$$\frac{\begin{array}{l} \langle a_1, \sigma \rangle \rightarrow_{Aexp} n \\ \langle a_2, \sigma \rangle \rightarrow_{Aexp} m \\ n = \perp, m = \perp \text{ oder } m = 0 \end{array}}{\langle a_1/a_2, \sigma \rangle \rightarrow_{Aexp} \perp}$$

Denotational $\mathcal{A}[[a]]$

$$\{(\sigma, n/m) \mid \sigma \in \Sigma, (\sigma, n) \in \mathcal{A}[[a_1]], (\sigma, m) \in \mathcal{A}[[a_2]], m \neq 0\}$$

Äquivalenz operationale und denotationale Semantik

- ▶ Für alle $a \in \mathbf{Aexp}$, für alle $n \in \mathbf{N}$, für alle Zustände σ :

$$\langle a, \sigma \rangle \rightarrow_{Aexp} n \Leftrightarrow (\sigma, n) \in \mathcal{A}[[a]]$$

$$\langle a, \sigma \rangle \rightarrow_{Aexp} \perp \Leftrightarrow \sigma \notin Dom(\mathcal{A}[[a]])$$

- ▶ Beweis per struktureller Induktion über a .

Operationale vs. denotationale Semantik

Operational $\langle b, \sigma \rangle \rightarrow_{Bexp} 0|1$

Denotational $\mathcal{B}[[b]]$

1 $\langle 1, \sigma \rangle \rightarrow_{Bexp} 1$

$\{(\sigma, 1) \mid \sigma \in \Sigma\}$

0 $\langle 0, \sigma \rangle \rightarrow_{Bexp} 0$

$\{(\sigma, 0) \mid \sigma \in \Sigma\}$

Operationale vs. denotationale Semantik

Operat. $\langle b, \sigma \rangle \rightarrow_{Bexp} 0|1$

$$\langle a_0, \sigma \rangle \rightarrow_{Aexp} n$$

$$\langle a_1, \sigma \rangle \rightarrow_{Aexp} m$$

$$\frac{n, m \neq \perp \quad n = m}{\langle a_0 == a_1, \sigma \rangle \rightarrow_{Bexp} 1}$$

$$\langle a_0 == a_1, \sigma \rangle \rightarrow_{Bexp} 1$$

$$\langle a_0, \sigma \rangle \rightarrow_{Aexp} n$$

$$\langle a_1, \sigma \rangle \rightarrow_{Aexp} m$$

$$\frac{n, m \neq \perp \quad n \neq m}{\langle a_0 == a_1, \sigma \rangle \rightarrow_{Bexp} 0}$$

$$\langle a_0 == a_1, \sigma \rangle \rightarrow_{Bexp} 0$$

$$\langle a_0, \sigma \rangle \rightarrow_{Aexp} n$$

$$\langle a_1, \sigma \rangle \rightarrow_{Aexp} m$$

$$\frac{n = \perp \text{ oder } m = \perp}{\langle a_0 == a_1, \sigma \rangle \rightarrow_{Bexp} \perp}$$

$$\langle a_0 == a_1, \sigma \rangle \rightarrow_{Bexp} \perp$$

$a_1 \leq a_2$

Denotational $\mathcal{B}[[b]]$

$$\{(\sigma, 1) \mid \sigma \in \Sigma, \\ (\sigma, n_0) \in \mathcal{A}[[a_0]], \\ (\sigma, n_1) \in \mathcal{A}[[a_1]], \\ n_0 = n_1\}$$

\cup

$$\{(\sigma, 0) \mid \sigma \in \Sigma, \\ (\sigma, n_0) \in \mathcal{A}[[a_0]], \\ (\sigma, n_1) \in \mathcal{A}[[a_1]], \\ n_0 \neq n_1\}$$

analog

Operationale vs. denotationale Semantik

Operational $\langle a, \sigma \rangle \rightarrow_{Bexp} b$

Denotational $\mathcal{B}[[b]]$

$$b_1 \&\& b_0 \quad \frac{\langle b_1, \sigma \rangle \rightarrow_{Bexp} 0}{\langle b_1 \&\& b_2, \sigma \rangle \rightarrow 0}$$
$$\frac{\langle b_1, \sigma \rangle \rightarrow_{Bexp} 1 \quad \langle b_2, \sigma \rangle \rightarrow_{Bexp} b}{\langle b_1 \&\& b_2, \sigma \rangle \rightarrow b}$$
$$\frac{\langle b_1, \sigma \rangle \rightarrow_{Bexp} \perp}{\langle b_1 \&\& b_2, \sigma \rangle \rightarrow \perp}$$

$$\{(\sigma, 0) \mid (\sigma, 0) \in \mathcal{B}[[b_1]]\}$$

$$\{(\sigma, b) \mid (\sigma, 1) \in \mathcal{B}[[b_1]], (\sigma, b) \in \mathcal{B}[[b_2]]\}$$

$b_1 \parallel b_2$

analog

$!n$

...

Äquivalenz operationale und denotationale Semantik

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

$$\langle b, \sigma \rangle \rightarrow_{Bexp} t \Leftrightarrow (\sigma, t) \in \mathcal{B}[[b]]$$

$$\langle b, \sigma \rangle \rightarrow_{Bexp} \perp \Leftrightarrow \sigma \notin Dom(\mathcal{B}[[b]])$$

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

Operationale vs. denotationale Semantik

Operational

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

$\{c_1 \dots c_n\}$

$$\frac{\langle \{\}, \sigma \rangle \rightarrow_{Stmt} \sigma}{\langle \{c_1, \sigma\} \rightarrow_{Stmt} \sigma'' \neq \perp}$$
$$\frac{\langle \{c_2 \dots c_n\}, \sigma' \rangle \rightarrow_{Stmt} \sigma''}{\langle \{c_1 \dots c_n\}, \sigma \rangle \rightarrow_{Stmt} \sigma''}$$
$$\frac{\langle c_1, \sigma \rangle \rightarrow_{Stmt} \perp}{\langle \{c_1 \dots c_n\}, \sigma \rangle \rightarrow_{Stmt} \perp}$$

Denotational $C[[c]]$

$$B[[c_n]] \circ \dots \circ B[[c_1]] \circ Id$$

$x = a$

$$\frac{\langle a, \sigma \rangle \rightarrow_{Aexp} n}{\langle x = a, \sigma \rangle \rightarrow_{Stmt} \sigma[n/x]}$$
$$\frac{\langle a, \sigma \rangle \rightarrow_{Aexp} \perp}{\langle x = a, \sigma \rangle \rightarrow_{Stmt} \perp}$$

$$\{(\sigma, \sigma[n/X]) \mid (\sigma, n) \in \mathcal{A}[[a]]\}$$

Operationale vs. denotationale Semantik

Operational

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

if (b) c_0

$$\frac{\langle b, \sigma \rangle \rightarrow_{Bexp} 1 \quad \langle c_0, \sigma \rangle \rightarrow_{Stmt} \sigma'}{\langle c, \sigma \rangle \rightarrow_{Stmt} \sigma'}$$

$$\frac{\langle b, \sigma \rangle \rightarrow_{Bexp} \perp}{\langle c, \sigma \rangle \rightarrow_{Stmt} \perp}$$

else c_1

$$\frac{\langle c_1, \sigma \rangle \rightarrow_{Stmt} \sigma'}{\langle c, \sigma \rangle \rightarrow_{Stmt} \sigma'}$$

Denotational $\mathcal{C}[[c]]$

$$\{(\sigma, \sigma') \mid (\sigma, 1) \in \mathcal{B}[[b]], (\sigma, \sigma') \in \mathcal{C}[[c_0]]\}$$

$$\{(\sigma, \sigma') \mid (\sigma, 0) \in \mathcal{B}[[b]], (\sigma, \sigma') \in \mathcal{C}[[c_1]]\}$$

Operationale vs. denotationale Semantik

Operational $\langle c, \sigma \rangle \rightarrow_{Stmnt} \sigma' \mid \perp$

Denotational $\mathcal{C}[[c]]$

$$\underbrace{\text{while } (b) \text{ c}}_w \quad \frac{\langle b, \sigma \rangle \rightarrow_{Bexp} 0 \quad \langle b, \sigma \rangle \rightarrow_{Bexp} \perp}{\langle w, \sigma \rangle \rightarrow_{Stmnt} \sigma \quad \langle w, \sigma \rangle \rightarrow_{Stmnt} \perp} \quad \text{fix}(\Gamma)$$

$$\frac{\langle b, \sigma \rangle \rightarrow_{Bexp} 1 \quad \langle c, \sigma \rangle \rightarrow_{Stmnt} \sigma' \neq \perp \quad \langle w, \sigma' \rangle \rightarrow_{Stmnt} \sigma''}{\langle w, \sigma \rangle \rightarrow_{Stmnt} \sigma''}$$

$$\frac{\langle b, \sigma \rangle \rightarrow_{Bexp} 1 \quad \langle c, \sigma \rangle \rightarrow_{Stmnt} \perp}{\langle w, \sigma \rangle \rightarrow_{Stmnt} \perp}$$

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' :

$$\langle c, \sigma \rangle \rightarrow_{\mathbf{Stmnt}} \sigma' \Leftrightarrow (\sigma, \sigma') \in \mathcal{C}[[c]]$$

$$\langle c, \sigma \rangle \rightarrow_{\mathbf{Stmnt}} \perp \Rightarrow \sigma \notin \text{Dom}(\mathcal{C}[[c]])$$

- ▶ \Rightarrow Beweis per Induktion über die Ableitung in der operationalen Semantik
- ▶ \Leftarrow Beweis per struktureller Induktion über c (Verwendung der Äquivalenz für arithmetische und boolesche 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 $c \equiv \text{while}(1)\{\}$:
 $\mathcal{C}[[c]] = \emptyset$ aber $\langle c, \sigma \rangle \rightarrow_{\mathbf{Stmnt}} \perp$ 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

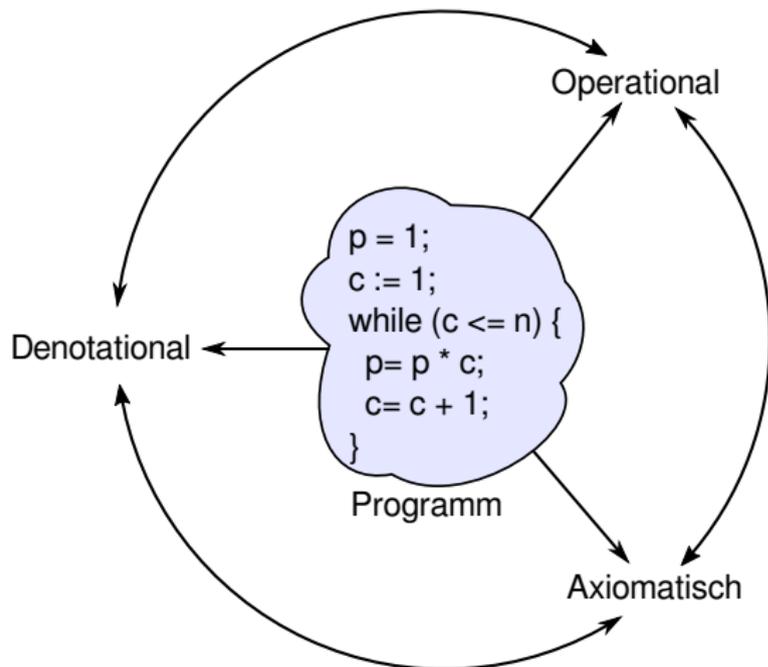
Fahrplan

- ▶ 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

Motivation

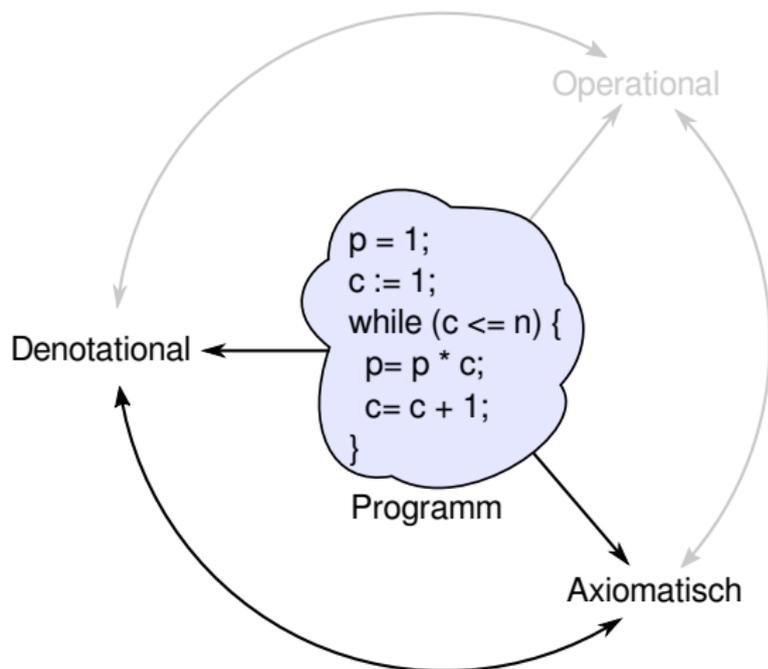
- ▶ In den letzten Wochen: **Semantik**
- ▶ Warum?
 - ▶ **Bedeutung** von Programmen **mathematisch** präzise fassen,
 - ▶ um insbesondere die über die Korrektheit von **Programmen** zu reden.
- ▶ Fragen:
 1. Was **bedeutet** es, wenn ein Programm **korrekt** ist?
 2. Wie können wir das **beweisen**?

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**.

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**.

Denotationale Semantik

- ▶ Denotat eines Ausdrucks (Programms) ist partielle Funktion:

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

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

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

- ▶ $f : A \rightarrow B$, dann (\perp steht für “undefiniert”):

$$\text{def}(f(x)) \iff f(x) \neq \perp$$

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

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

- ▶ Aber: $\models \{P\} c \{Q\} \stackrel{?}{\iff} \vdash \{P\} c \{Q\}$

Überblick: die Regeln des Floyd-Hoare-Kalküls

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

$$\frac{}{\vdash \{A\} \{\} \{A\}} \quad \frac{\vdash \{A\} c \{B\} \quad \vdash \{B\} \{c_s\} \{C\}}{\vdash \{A\} \{c \ c_s\} \{C\}}$$

$$\frac{\vdash \{A \wedge b\} c_0 \{B\} \quad \vdash \{A \wedge \neg b\} c_1 \{B\}}{\vdash \{A\} \text{if } (b) \ c_0 \ \text{else } c_1 \{B\}}$$

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

$$\frac{A' \implies A \quad \vdash \{A\} c \{B\} \quad B \implies B'}{\vdash \{A'\} c \{B'\}}$$

Korrektheit und Vollständigkeit

▶ **Korrektheit:** $\vdash \{P\} c \{Q\} \stackrel{?}{\implies} \models \{P\} c \{Q\}$

▶ Wir können nur gültige Eigenschaften von Programmen herleiten.

▶ **Vollständigkeit:** $\models \{P\} c \{Q\} \stackrel{?}{\implies} \vdash \{P\} c \{Q\}$

▶ 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 $wp(c, Q)$.
- ▶ 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

- ▶ 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

Vorwärts oder Rückwärts?

Idee

- ▶ Hier ist ein einfaches Programm:

```
// {X = x ∧ Y = y}
```

```
z = y;
```

```
y = x;
```

```
x = z;
```

```
// {X = y ∧ Y = x}
```

Idee

- ▶ Hier ist ein einfaches Programm:

```
// {X = x ∧ Y = y}
```

```
z = y;
```

```
y = x;
```

```
// {X = y ∧ Y = z}
```

```
x = z;
```

```
// {X = y ∧ Y = x}
```

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}
```

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.

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.

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

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?

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

$$\frac{\vdash \{A\} c \{B\} \quad \vdash \{B\} \{c_s\} \{C\}}{\vdash \{A\} \{c c_s\} \{C\}} \quad \frac{\vdash \{A \wedge b\} c \{A\}}{\vdash \{A\} \mathbf{while}(b) c \{A \wedge \neg b\}}$$

$$\frac{A' \implies A \quad \vdash \{A\} c \{B\} \quad B \implies B'}{\vdash \{A'\} c \{B'\}}$$

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!

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

$$\frac{\vdash \{A\} c \{B\} \quad \vdash \{B\} \{c_s\} \{C\}}{\vdash \{A\} \{c c_s\} \{C\}} \quad \frac{\vdash \{A \wedge b\} c \{A\}}{\vdash \{A\} \mathbf{while}(b) c \{A \wedge \neg b\}}$$

$$\frac{A' \implies A \quad \vdash \{A\} c \{B\} \quad B \implies B'}{\vdash \{A'\} c \{B'\}}$$

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;

// { $\exists V. x = x + 1[V/x] \wedge (x = 1)[V/x]$ }

- ▶ Vereinfachung der Nachbedingung:

$$\exists V. x = (x + 1)[V/x] \wedge (x = 1)[V/x]$$

$$\iff \exists V. x = (V + 1) \wedge (V = 1)$$

$$\iff x = 1 + 1$$

$$\iff x = 2$$

Vorwärtsverkettung

- ▶ Vorwärtsaxiom äquivalent zum Rückwärtsaxiom.
- ▶ In der Anwendung **umständlicher**.
- ▶ Vereinfachung benötigt Lemma: $\exists x.P(x) \wedge x = t \iff P(t)$
- ▶ 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
 - ▶ $wp(c, P)$ die **schwächste Vorbedingung** Q so dass $\models \{Q\} c \{P\}$;
- ▶ Prädikat P **schwächer** als Q wenn $Q \implies P$ (**stärker** wenn $P \implies Q$).

Berechnung von Vorbedingungen

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

$$\models \{P\} c \{Q\} \iff P \implies \text{wp}(c, Q)$$

Berechnung von Vorbedingungen

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

$$\begin{aligned}\models \{P\} c \{Q\} &\iff P \implies \text{wp}(c, Q) \\ \models \{P\} c \{Q\} &\iff \text{sp}(P, c) \implies Q\end{aligned}$$

Berechnung von $wp(c, Q)$

- ▶ Einfach für Programme ohne Schleifen:

$$wp(\{\}, P) \stackrel{def}{=} P$$

$$wp(x = e, P) \stackrel{def}{=} P[e/x]$$

$$wp(\{c \ c_s\}, P) \stackrel{def}{=} wp(c, wp(\{c_s\}, P))$$

$$wp(\mathbf{if} (b) \ c_0 \ \mathbf{else} \ c_1, P) \stackrel{def}{=} (b \wedge wp(c_0, P)) \vee (\neg b \wedge wp(c_1, P))$$

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

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

- ▶ 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 $\text{awp}(c, Q)$ zusammen mit einer Menge von **Verifikationsbedingungen** $\text{wvc}(c, Q)$
- ▶ Die Verifikationsbedingungen treten dort auf, wo die Weakening-Regel angewandt werden muss.
- ▶ Es gilt:

$$\bigwedge \text{wvc}(c, Q) \implies \models \{\text{awp}(c, Q)\} c \{Q\}$$

Lösung: Annotierte Programme

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

$$\begin{aligned}\bigwedge \text{wvc}(c, Q) &\implies \models \{ \text{awp}(c, Q) \} c \{ Q \} \\ \bigwedge \text{svc}(P, c) &\implies \models \{ P \} c \{ \text{asp}(P, c) \}\end{aligned}$$

Approximative schwächste Vorbedingung

- Für die **while**-Schleife:

$$\text{awp}(\mathbf{while}(b) \ / ** \ \mathbf{inv} \ i \ */ \ c, P) \stackrel{\text{def}}{=} i$$

$$\begin{aligned} \text{wvc}(\mathbf{while}(b) \ / ** \ \mathbf{inv} \ i \ */ \ c, P) &\stackrel{\text{def}}{=} \text{wvc}(c, i) \\ &\cup \{i \wedge b \implies \text{awp}(c, i)\} \\ &\cup \{i \wedge \neg b \implies P\} \end{aligned}$$

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

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

$$\frac{A \wedge b \implies C \quad \vdash \{C\} c \{A\} \quad A \wedge \neg b \implies B}{\vdash \{A\} \mathbf{while}(b) c \{B\}} \quad (2)$$

Überblick: Approximative schwächste Vorbedingung

$$\text{awp}(\{\}, P) \stackrel{\text{def}}{=} P$$

$$\text{awp}(x = e, P) \stackrel{\text{def}}{=} P[e/x]$$

$$\text{awp}(\{c \ c_s\}, P) \stackrel{\text{def}}{=} \text{awp}(c, \text{awp}(\{c_s\}, P))$$

$$\text{awp}(\text{if } (b) \ c_0 \ \text{else } \ c_1, P) \stackrel{\text{def}}{=} (b \wedge \text{awp}(c_0, P)) \vee (\neg b \wedge \text{awp}(c_1, P))$$

$$\text{awp}(/** \{q\} */ , P) \stackrel{\text{def}}{=} q$$

$$\text{awp}(\text{while } (b) \ /** \ \text{inv } \ i \ */ \ c, P) \stackrel{\text{def}}{=} i$$

$$\text{wvc}(\{\}, P) \stackrel{\text{def}}{=} \emptyset$$

$$\text{wvc}(x = e, P) \stackrel{\text{def}}{=} \emptyset$$

$$\text{wvc}(\{c \ c_s\}, P) \stackrel{\text{def}}{=} \text{wvc}(c, \text{awp}(\{c_s\}, P)) \cup \text{wvc}(\{c_s\}, P)$$

$$\text{wvc}(\text{if } (b) \ c_0 \ \text{else } \ c_1, P) \stackrel{\text{def}}{=} \text{wvc}(c_0, P) \cup \text{wvc}(c_1, P)$$

$$\text{wvc}(/** \{q\} */ , P) \stackrel{\text{def}}{=} \{q \implies P\}$$

$$\text{wvc}(\text{while } (b) \ /** \ \text{inv } \ i \ */ \ c, P) \stackrel{\text{def}}{=} \text{wvc}(c, i) \cup \{i \wedge b \implies \text{awp}(c, i)\} \\ \cup \{i \wedge \neg b \implies P\}$$

Alternative Schreibweise: AWP

$$\overline{P \leftarrow_{awp} \{ \}, P}$$

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

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

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

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

Alternative Schreibweise: WVC

$$\overline{\{\}, P \rightarrow_{wvc} \emptyset}$$

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

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

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

$$\overline{/** \{q\} */ , P \rightarrow_{wvc} \{q \implies P\}}$$

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

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:
 $wvc(F, p == fac(N))$

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

- ▶ 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

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:

$$\mathbf{FunDef} ::= \mathbf{Type} \mathit{Id}(\mathbf{Param}^*) \mathbf{FunSpec}^+ \mathbf{Blk}$$
$$\mathbf{Param} ::= \mathbf{Type} \mathit{Id}$$
$$\mathbf{Blk} ::= \{\mathbf{Decl}^* \mathbf{Stmt}\}$$
$$\mathbf{Decl} ::= \mathbf{Type} \mathit{Id} = \mathbf{Aexp} \mid \mathbf{Type} \mathit{Id}$$
$$\mathbf{Aexp} ::= \dots \mid \mathit{Id}(\mathbf{Aexp}^*)$$
$$\mathbf{Stmt} ::= \dots \mid \mathit{Id}(\mathbf{Aexp}^*) \mid \mathbf{return} (\mathbf{Aexp})?$$

- ▶ **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 $\Sigma \rightarrow \Sigma$ zu $\Sigma \rightarrow (\Sigma + \Sigma \times \mathbf{V})$

Erweiterte Semantik

- ▶ Denotat einer Anweisung: $\Sigma \mapsto (\Sigma + \Sigma \times \mathbf{V})$
- ▶ Abbildung von Ausgangszustand Σ auf:
 - ▶ Sequentieller Folgezustand, oder
 - ▶ Rückgabewert und Rückgabeszustand
- ▶ Was ist mit **void**?

Erweiterte Semantik

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

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

Semantik von Anweisungen

$$\mathcal{C}[\cdot] : \mathbf{Stmt} \rightarrow \Sigma \rightarrow (\Sigma + \Sigma \times \mathbf{V}_U)$$

$$\mathcal{C}[x = e] = \{(\sigma, \sigma[a/x]) \mid (\sigma, a) \in \mathcal{A}[e]\}$$

$$\mathcal{C}[\{c \ c_s\}] = \mathcal{C}[c_s] \cdot_s \mathcal{C}[c] \quad \text{Komposition wie oben}$$

$$\mathcal{C}[\{\}] = \mathbf{Id} \quad \mathbf{Id} := \{(\sigma, \sigma) \mid \sigma \in \Sigma\}$$

$$\begin{aligned} \mathcal{C}[\mathbf{if} (b) \ c_0 \ \mathbf{else} \ c_1] &= \{(\sigma, \tau) \mid (\sigma, 1) \in \mathcal{B}[b] \wedge (\sigma, \tau) \in \mathcal{C}[c_0]\} \\ &\quad \cup \{(\sigma, \tau) \mid (\sigma, 0) \in \mathcal{B}[b] \wedge (\sigma, \tau) \in \mathcal{C}[c_1]\} \\ &\quad \text{mit } \tau \in \Sigma \cup (\Sigma \times \mathbf{V}_U) \end{aligned}$$

$$\mathcal{C}[\mathbf{return} \ e] = \{(\sigma, (\sigma, a)) \mid (\sigma, a) \in \mathcal{A}[e]\}$$

$$\mathcal{C}[\mathbf{return}] = \{(\sigma, (\sigma, *))\}$$

$$\mathcal{C}[\mathbf{while} (b) \ c] = \mathit{fix}(\Gamma)$$

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

Semantik von Funktionsdefinitionen

$$\mathcal{D}_{fd}[\cdot] : \mathbf{FunDef} \rightarrow \mathbf{V}^n \rightarrow \Sigma \rightarrow \Sigma \times \mathbf{V}_U$$

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

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

- ▶ 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

$$\mathcal{D}_{blk}[\cdot] : \mathbf{Blk} \rightarrow \Sigma \rightarrow (\Sigma + \Sigma \times \mathbf{V}_U)$$

$$\mathcal{D}_d[\cdot] : \mathbf{Decl} \rightarrow \Sigma \rightarrow (\Sigma + \Sigma \times \mathbf{V}_U)$$

Blöcke bestehen aus Deklarationen und einer Anweisung:

$$\mathcal{D}_{blk}[\mathit{decls} \ \mathit{stmts}] = \mathcal{C}[\mathit{stmts}] \cdot_S \mathcal{D}_d[\mathit{decls}]$$

$$\mathcal{D}_d[\mathit{t} \ \mathit{i}] = \{(\sigma, \sigma[\perp/\mathit{i}])\}$$

$$\mathcal{D}_d[\mathit{t} \ \mathit{i} = \mathit{init}] = \{(\sigma, \sigma[\mathcal{A}_{init}[\mathit{init}]/\mathit{i}])\}$$

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

Funktionsaufrufe

- ▶ Aufruf einer Funktion: $f(t_1, \dots, t_n)$:
 - ▶ Auswertung der Argumente t_1, \dots, 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):

$$\begin{aligned} Env &= Id \rightarrow \llbracket \mathbf{FunDef} \rrbracket \\ &= Id \rightarrow \mathbf{V}^N \rightarrow \Sigma \rightarrow (\Sigma \times \mathbf{V}_u) \end{aligned}$$

- ▶ Das Environment ist **zusätzlicher Parameter** für alle Definitionen

Semantik von Funktionsaufrufen

$$\mathcal{A}[[f(t_1, \dots, t_n)]]\Gamma = \{(\sigma, \nu) \mid \exists \sigma', \nu. (\sigma, (\sigma', \nu)) \in \Gamma(f)(\nu_1, \dots, \nu_n) \\ \wedge (\sigma, \nu_i) \in \mathcal{A}[[t_i]]\Gamma\}$$

$$\mathcal{C}[[f(t_1, \dots, t_n)]]\Gamma = \{(\sigma, \sigma') \mid \exists \sigma'. (\sigma, (\sigma', *)) \in \Gamma(f)(\nu_1, \dots, \nu_n) \\ \wedge (\sigma, \nu_i) \in \mathcal{A}[[t_i]]\Gamma\}$$

- ▶ 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 $\mathcal{A}[[f(t_1, \dots, t_n)]]$ ignoriert Endzustand
- ▶ Aufruf von Prozedur $\mathcal{C}[[f(t_1, \dots, t_n)]]$ ignoriert Rückgabewert

Semantik von Funktionsaufrufen

$$\mathcal{A}[\![f(t_1, \dots, t_n)]\!] \Gamma = \{(\sigma, \nu) \mid \exists \sigma', \nu. (\sigma, (\sigma', \nu)) \in \Gamma(f)(\nu_1, \dots, \nu_n) \wedge (\sigma, \nu_i) \in \mathcal{A}[\![t_i]\!] \Gamma\}$$

$$\mathcal{C}[\![f(t_1, \dots, t_n)]\!] \Gamma = \{(\sigma, \sigma') \mid \exists \sigma'. (\sigma, (\sigma', *)) \in \Gamma(f)(\nu_1, \dots, \nu_n) \wedge (\sigma, \nu_i) \in \mathcal{A}[\![t_i]\!] \Gamma\}$$

$$\mathcal{C}[\![x = f(t_1, \dots, t_n)]\!] \Gamma = \{(\sigma, \sigma'[v/x]) \mid \exists \sigma', \nu. (\sigma, (\sigma', \nu)) \in \Gamma(f)(\nu_1, \dots, \nu_n) \wedge (\sigma, \nu_i) \in \mathcal{A}[\![t_i]\!] \Gamma\}$$

- ▶ 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 $\mathcal{A}[\![f(t_1, \dots, t_n)]\!] \Gamma$ ignoriert Endzustand
- ▶ Aufruf von Prozedur $\mathcal{C}[\![f(t_1, \dots, t_n)]\!] \Gamma$ 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; $\Sigma \rightarrow \mathbf{T}$

Nachbedingung **post** sp; $\Sigma \times (\Sigma \times \mathbf{V}_U) \rightarrow \mathbf{T}$

\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 $\mathcal{B}[\![\cdot]\!]$ und $\mathcal{A}[\![\cdot]\!]$
 - ▶ Ausdrücke können in Vor- oder Nachzustand ausgewertet werden.
 - ▶ \backslash **result** kann nicht in Funktionen vom Typ **void** auftreten.

$$\mathcal{B}_{sp}[\![\cdot]\!] : Env \rightarrow \mathbf{Bexp} \rightarrow (\Sigma \times (\Sigma \times \mathbf{V}_U)) \rightarrow \mathbf{T}$$

$$\mathcal{A}_{sp}[\![\cdot]\!] : Env \rightarrow \mathbf{Aexp} \rightarrow (\Sigma \times (\Sigma \times \mathbf{V}_U)) \rightarrow \mathbf{V}$$

$$\mathcal{B}_{sp}[\![!b]\!] \Gamma = \{((\sigma, (\sigma', \nu)), 1) \mid ((\sigma, (\sigma', \nu)), 0) \in \mathcal{B}_{sp}[\![b]\!] \Gamma\} \\ \cup \{((\sigma, (\sigma', \nu)), 0) \mid ((\sigma, (\sigma', \nu)), 1) \in \mathcal{B}_{sp}[\![b]\!] \Gamma\}$$

...

$$\mathcal{B}_{sp}[\![\backslash \mathbf{old}(e)]\!] \Gamma = \{((\sigma, (\sigma', \nu)), b) \mid (\sigma, b) \in \mathcal{B}[\![e]\!] \Gamma\}$$

$$\mathcal{A}_{sp}[\![\backslash \mathbf{old}(e)]\!] \Gamma = \{((\sigma, (\sigma', \nu)), a) \mid (\sigma, a) \in \mathcal{A}[\![e]\!] \Gamma\}$$

$$\mathcal{A}_{sp}[\![\backslash \mathbf{result}]\!] \Gamma = \{((\sigma, (\sigma, \nu)), \nu)\}$$

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

Gültigkeit von Spezifikationen

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

$$\mathbf{pre} \ p \ \mathbf{post} \ q \models FunDef$$

$$\iff \forall v_1, \dots, v_n. \mathcal{D}_{fd}[\![FunDef]\!] \ \Gamma \in \mathcal{B}_{sp}[\![\mathbf{pre} \ p \ \mathbf{post} \ q]\!] \ \Gamma$$

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

Gültigkeit von Spezifikationen

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

$$\mathbf{pre} \ p \ \mathbf{post} \ q \models FunDef$$

$$\iff \forall v_1, \dots, v_n. \mathcal{D}_{fd}[\![FunDef]\!] \ \Gamma \in \mathcal{B}_{sp}[\![\mathbf{pre} \ p \ \mathbf{post} \ q]\!] \ \Gamma$$

- ▶ 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

$$\mathcal{C}[\cdot] : \mathbf{Stmt} \rightarrow \Sigma \rightarrow (\Sigma + \Sigma \times \mathbf{V}_U)$$

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.
- ▶ $\Gamma(f) = \forall x_1, \dots, x_n. (P, Q)$, für Funktion $f(x_1, \dots, x_n)$ 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

$$\frac{}{\Gamma \vdash \{Q\} \text{ return } \{P|Q\}}$$

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

- ▶ 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

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

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

Erweiterung des Floyd-Hoare-Kalküls: Aufruf

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

- ▶ 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, \dots, y_m sind die als $\backslash\mathbf{old}(y_j)$ in Q auftretenden Variablen
- ▶ Y_1, \dots, Y_m dürfen nicht in P oder Q enthalten sein
- ▶ Im ersten Fall (Aufruf als Prozedur) enthält Q kein $\backslash\mathbf{result}$

Erweiterter Floyd-Hoare-Kalkül I

$$\frac{}{\Gamma \vdash \{P\} \{\} \{P|Q_R\}} \quad \frac{\Gamma \vdash \{P\} c \{R|Q_R\} \quad \Gamma \vdash \{R\} c s \{Q|Q_R\}}{\Gamma \vdash \{P\} c c s \{Q|Q_R\}}$$

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

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

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

Erweiterter Floyd-Hoare-Kalkül II

$$\frac{}{\Gamma \vdash \{Q\} \text{ return } \{P|Q\}} \quad \frac{}{\Gamma \vdash \{Q[e/\text{result}]\} \text{ return } e \{P|Q\}}$$

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

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

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

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:

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

- ▶ Berechnung von **awp** und **wvc**:

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

Approximative schwächste Vorbedingung (Revisited)

$$\text{awp}(\Gamma, \{ \}, Q, Q_R) \stackrel{\text{def}}{=} Q$$

$$\text{awp}(\Gamma, l = f(e_1, \dots, e_n), Q, Q_R) \stackrel{\text{def}}{=} P[e_i/x_i]$$

mit $\Gamma(f) = \forall x_1, \dots, x_n. (P, R)$

$$\text{awp}(\Gamma, f(e_1, \dots, e_n), Q, Q_R) \stackrel{\text{def}}{=} P[e_i/x_i]$$

mit $\Gamma(f) = \forall x_1, \dots, x_n. (P, R)$

$$\text{awp}(\Gamma, l = e, Q, Q_R) \stackrel{\text{def}}{=} P[e/l]$$

$$\text{awp}(\Gamma, \{c \ c_s\}, Q, Q_R) \stackrel{\text{def}}{=} \text{awp}(\Gamma, c, \text{awp}(\{c_s\}, Q, Q_R), Q_R)$$

$$\text{awp}(\Gamma, \text{if } (b) \{c_0\} \text{ else } \{c_1\}, Q, Q_R) \stackrel{\text{def}}{=} (b \ \&\& \ \text{awp}(\Gamma, c_0, Q, Q_R)) \ || \ (!b \ \&\& \ \text{awp}(\Gamma, c_1, Q, Q_R))$$

$$\text{awp}(\Gamma, /** \{q\} */, Q, Q_R) \stackrel{\text{def}}{=} q$$

$$\text{awp}(\Gamma, \text{while } (b) /** \text{inv } i */ c, Q, Q_R) \stackrel{\text{def}}{=} i$$

$$\text{awp}(\Gamma, \text{return } e, Q, Q_R) \stackrel{\text{def}}{=} Q_R[e/ \backslash \text{result}]$$

$$\text{awp}(\Gamma, \text{return}, Q, Q_R) \stackrel{\text{def}}{=} Q_R$$

Approximative Verifikationsbedingungen (Revisited)

$$\text{wvc}(\Gamma, \{ \}, Q, Q_R) \stackrel{\text{def}}{=} \emptyset$$

$$\text{wvc}(\Gamma, x = e, Q, Q_R) \stackrel{\text{def}}{=} \emptyset$$

$$\text{wvc}(\Gamma, x = f(e_1, \dots, e_n), Q, Q_R) \stackrel{\text{def}}{=} \{R[e_i/x_i][x/\text{result}] \implies Q\}$$

mit $\Gamma(f) = \forall x_1, \dots, x_n. (P, R)$

$$\text{wvc}(\Gamma, f(e_1, \dots, e_n), Q, Q_R) \stackrel{\text{def}}{=} \{R[e_i/x_i] \implies Q\}$$

mit $\Gamma(f) = \forall x_1, \dots, x_n. (P, R)$

$$\text{wvc}(\Gamma, \{c \ c_s\}, Q, Q_R) \stackrel{\text{def}}{=} \text{wvc}(\Gamma, c, \text{awp}(\{c_s\}, Q, Q_R), Q_R) \cup \text{wvc}(\Gamma, \{c_s\}, Q, Q_R)$$

$$\text{wvc}(\Gamma, \text{if } (b) \ c_0 \ \text{else } \ c_1, Q, Q_R) \stackrel{\text{def}}{=} \text{wvc}(\Gamma, c_0, Q, Q_R) \cup \text{wvc}(\Gamma, c_1, Q, Q_R)$$

$$\text{wvc}(\Gamma, /** \{q\} */, Q, Q_R) \stackrel{\text{def}}{=} \{q \implies Q\}$$

$$\text{wvc}(\Gamma, \text{while } (b) \ /** \text{inv } i \ */ \ c, Q, Q_R) \stackrel{\text{def}}{=} \text{wvc}(\Gamma, c, i, Q_R) \cup \{i \wedge b \implies \text{awp}(\Gamma, c, i, Q_R)\} \cup \{i \wedge \neg b \implies Q\}$$

$$\text{wvc}(\Gamma, \text{return } e, Q, Q_R) \stackrel{\text{def}}{=} \emptyset$$

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

- ▶ 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

Motivation

- ▶ Weitere Basisdatentypen von C (arrays und structs)
- ▶ Noch rein funktional, keine Pointer
 - ▶ Damit auch kein *call by reference*
 - ▶ Funktionen 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**.

Referenzen 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 Zustandsmodells

- ▶ Erweiterung von Zustand und Werten:

$$\Sigma = \mathbf{Loc} \rightarrow \mathbf{V} \quad \mathbf{V} = \mathbf{N} + \mathbf{Loc}$$

- ▶ 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:

$$\text{read} : \Sigma \rightarrow \mathbf{Loc} \rightarrow \mathbf{V}$$

$$\text{upd} : \Sigma \rightarrow \mathbf{Loc} \rightarrow \mathbf{V} \rightarrow \Sigma$$

$$\text{read}(\text{upd}(\sigma, l, v), l) = v$$

$$l \neq m \implies \text{read}(\text{upd}(\sigma, l, v), m) = \text{read}(\sigma, m)$$

$$\text{upd}(\text{upd}(\sigma, l, v), l, w) = \text{upd}(\sigma, l, w)$$

$$l \neq m \implies \text{upd}(\text{upd}(\sigma, l, v), m, w) = \text{upd}(\text{upd}(\sigma, m, w), l, v)$$

- ▶ Diese Gleichungen sind **vollständig**.

Axiomatisches Speichermodell

- ▶ Es gibt einen **leeren** Speicher, und neue (“frische”) Adressen:

$$\text{empty} : \Sigma$$

$$\text{fresh} : \Sigma \rightarrow \mathbf{Loc}$$

$$\text{rem} : \Sigma \rightarrow \mathbf{Loc} \rightarrow \Sigma$$

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

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

$$\text{dom}(\text{empty}) = \emptyset$$

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

$$\text{fresh}(\sigma) \notin \text{dom}(\sigma)$$

$$\text{dom}(\text{rem}(\sigma, l)) = \text{dom}(\sigma) \setminus \{l\}$$

$$l \neq m \implies \text{read}(\text{rem}(\sigma, l), m) = \text{read}(\sigma, m)$$

Zeigerarithmetik

- ▶ Erklärt noch keine Zeigerarithmetik — dazu:

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

- ▶ Wir betrachten keine **Differenz** von Zeigern

$$add(l, 0) = l$$

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

Erweiterung der Semantik

- ▶ Problem: **L**oc haben unterschiedliche Semantik auf der linken oder rechten Seite einer Zuweisung.
 - ▶ $x = x+1$ — Links: Adresse der Variablen, rechts: Wert an dieser Adresse
- ▶ Lösung: “Except when it is (...) the operand of the unary & operator, the left operand of the . operator or an assignment 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 brauchen wir eine **Umgebung** (Environment):

$$\begin{aligned} Env &= Id \rightarrow \llbracket \mathbf{FunDef} \rrbracket \\ &= Id \rightarrow \mathbf{V}^N \rightarrow \Sigma \rightarrow (\Sigma \times \mathbf{V}_u) \end{aligned}$$

- ▶ Diese muss erweitert werden für Variablen:

$$Env = Id \rightarrow (\llbracket \mathbf{FunDef} \rrbracket \uplus \mathbf{Loc})$$

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

Ausdrücke

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

Lexp $l ::= Id \mid l [a] \mid l.Id \mid * a$

Aexp $a ::= \mathbf{N} \mid l \mid \&l \mid a_1 + a_2 \mid a_1 - a_2 \mid$
 $a_1 * a_2 \mid a_1/a_2 \mid Id(a^*)$

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

Exp $e ::= a \mid b$

Statements

Type ::= **PointerType** | **BasicType** | **StructType** | **ArrayType**

BasicType ::= *int*

StructType ::= *struct name* {**Puredecl***}

ArrayType ::= **Type**[*n?*]

PointerType ::= **Type***

ExtDecl ::= **LogSpec** | **Decl** | **FunDef**

Decl ::= **Type** *Id* (= *e* | {*el*})*;

FunDef ::= **Type** *Id*(**PureDecl***) **FunSpec**⁺ **Blk**

Blk ::= {**Decl*** **Stmt**}

Puredecl ::= **Type** *Id*

Stmt ::= **Lexp** = **Exp**; | **if** (*b*) *c*₁ **else** *c*₂

| **while** (*b*) *c* | {*c**}

| *Id*(**AExp***) | **return** (**AExp**?)

Erweiterung der Semantik: Lexp

$$\mathcal{L}[\![-]\!] : Env \rightarrow \mathbf{Lexp} \rightarrow \Sigma \rightarrow \mathbf{Loc}$$

$$\mathcal{L}[x] \Gamma = \{(\sigma, \Gamma!x) \mid \sigma \in \Sigma\}$$

$$\mathcal{L}[lexp[a]] \Gamma = \{(\sigma, add(l, i \cdot sizeof(\tau))) \mid (\sigma, l) \in \mathcal{L}[lexp] \Gamma, (\sigma, i) \in \mathcal{A}[a] \Gamma\}$$

$type(\Gamma, lexp) = \tau$ ist der Basistyp des Feldes

$$\mathcal{L}[lexp.f] \Gamma = \{(\sigma, l.f) \mid (\sigma, add(l, fld_off(\tau, f))) \in \mathcal{L}[lexp] \Gamma\}$$

$type(\Gamma, lexp) = \tau$ ist der Typ der Struktur

$$\mathcal{L}[*e] \Gamma = \mathcal{A}[e] \Gamma$$

- ▶ $type(\Gamma, e)$ 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)

$$\mathcal{A}[-] : Env \rightarrow \mathbf{Aexp} \rightarrow \Sigma \rightarrow \mathbf{V}$$

$$\mathcal{A}[n] \Gamma = \{(\sigma, n) \mid \sigma \in \Sigma\} \quad \text{für } n \in \mathbf{N}$$

$$\mathcal{A}[e] \Gamma = \{(\sigma, \text{read}(\sigma, l)) \mid (\sigma, l) \in \mathcal{L}[e] \Gamma\}$$

e is **LExp** und $\text{type}(\Gamma, e)$ kein Array-Typ

$$\mathcal{A}[e] \Gamma = \{(\sigma, l) \mid (\sigma, l) \in \mathcal{L}[e] \Gamma\}$$

e is **LExp** und $\text{type}(\Gamma, e)$ Array-Typ

$$\mathcal{A}[\&e] \Gamma = \{(\sigma, l) \mid (\sigma, l) \in \mathcal{L}[e] \Gamma\}$$

$$\mathcal{A}[p + e] \Gamma = \{(\sigma, \text{add}(l, n \cdot \text{sizeof}(\tau))) \mid (\sigma, l) \in \mathcal{L}[p] \Gamma \wedge (\sigma, n) \in \mathcal{A}[e] \Gamma\}$$

$\text{type}(\Gamma, p) = * \tau$, $\text{type}(\Gamma, e)$ Integer-Typ

$$\mathcal{A}[e + p] \Gamma = \mathcal{A}[p + e] \Gamma$$

$\text{type}(\Gamma, e)$ Integer-Typ und $\text{type}(\Gamma, p) = * \tau$

Erweiterung der Semantik: Aexp(2)

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

$$\mathcal{A}[a_0 + a_1] \Gamma = \{(\sigma, n_0 + n_1) \mid (\sigma, n_0) \in \mathcal{A}[a_0] \Gamma \wedge (\sigma, n_1) \in \mathcal{A}[a_1] \Gamma\}$$

für a_0, a_1 arithmetische Typen

$$\mathcal{A}[a_0 - a_1] \Gamma = \{(\sigma, n_0 - n_1) \mid (\sigma, n_0) \in \mathcal{A}[a_0] \Gamma \wedge (\sigma, n_1) \in \mathcal{A}[a_1] \Gamma\}$$

$$\mathcal{A}[a_0 * a_1] \Gamma = \{(\sigma, n_0 * n_1) \mid (\sigma, n_0) \in \mathcal{A}[a_0] \Gamma \wedge (\sigma, n_1) \in \mathcal{A}[a_1] \Gamma\}$$

$$\mathcal{A}[a_0/a_1] \Gamma = \{(\sigma, n_0/n_1) \mid (\sigma, n_0) \in \mathcal{A}[a_0] \Gamma \wedge (\sigma, n_1) \in \mathcal{A}[a_1] \Gamma \\ \wedge n_1 \neq 0\}$$

Explizite Zustandsprädikate

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

Bexp ::= ... (wie vorher)

Aexp ::= *read*(**St**, **Lexp**) | **N** | **Lexp** | **&Lexp** | ... | **\old**(*e*) | ...

St ::= *StateVar* | *upd*(**St**, **Aexp**, **Bexp**)

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

$$\mathcal{B}_{sp}[\cdot] : Env \rightarrow \mathbf{Bexp} \rightarrow (\Sigma \times (\Sigma \times \mathbf{V}_U)) \rightarrow \mathbf{T}$$

$$\mathcal{A}_{sp}[\cdot] : Env \rightarrow \mathbf{Aexp} \rightarrow (\Sigma \times (\Sigma \times \mathbf{V}_U)) \rightarrow \mathbf{V}$$

- ▶ 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

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

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

¹Außer l ist ein Array-Typ.

Formal: Konversion in Zustandsprädikate

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

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

$$l.id^\dagger = l^\dagger.id$$

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

$$*l^\dagger = l^\#$$

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

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

$$n^\# = n$$

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

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

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

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

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

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

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
 - ▶ Dereferenzierung 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 $\backslash\text{valid}(x)$

$\backslash\text{valid}(*x)$ 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.
- ▶ $\backslash\text{array}(a, n)$ bedeutet: a ist ein Feld der Länge n , d.h.

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

- ▶ Validität kann abgeleitet werden:

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

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

- ▶ 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

- ▶ 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

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

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

$l.id^{\dagger} = l^{\dagger}.id$

$l[e]^{\dagger} = l^{\dagger}[e^{\#}]$

$*l^{\dagger} = l^{\#}$

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

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

$n^{\#} = n$

$v^{\#} = v$ (v 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**:

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

Approximative schwächste Vorbedingung (Revisited)

$$\text{awp}(\Gamma, \{ \}, Q, Q_R) \stackrel{\text{def}}{=} Q$$

$$\text{awp}(\Gamma, l = f(e_1, \dots, e_n), Q, Q_R) \stackrel{\text{def}}{=} P[e_i/x_i]^\#$$

mit $\Gamma(f) = \forall x_1, \dots, x_n. (P, R)$

$$\text{awp}(\Gamma, f(e_1, \dots, e_n), Q, Q_R) \stackrel{\text{def}}{=} P[e_i/x_i]^\#$$

mit $\Gamma(f) = \forall x_1, \dots, x_n. (P, R)$

$$\text{awp}(\Gamma, l = e, Q, Q_R) \stackrel{\text{def}}{=} Q[\text{upd}(\sigma, l^\dagger, e^\#)/\sigma]$$

$$\text{awp}(\Gamma, \{c \ c_s\}, Q, Q_R) \stackrel{\text{def}}{=} \text{awp}(\Gamma, c, \text{awp}(\{c_s\}, Q, Q_R), Q_R)$$

$$\text{awp}(\Gamma, \text{if } (b) \{c_0\} \text{ else } \{c_1\}, Q, Q_R) \stackrel{\text{def}}{=} (b^\# \ \&\& \ \text{awp}(\Gamma, c_0, Q, Q_R)) \ || \ (! b^\# \ \&\& \ \text{awp}(\Gamma, c_1, Q, Q_R))$$

$$\text{awp}(\Gamma, /** \{q\} */ , Q, Q_R) \stackrel{\text{def}}{=} q$$

$$\text{awp}(\Gamma, \left[\begin{array}{l} \text{while } (b) \\ /** \text{inv } i */ \\ c \end{array} \right], Q, Q_R) \stackrel{\text{def}}{=} i$$

$$\text{awp}(\Gamma, \text{return } e, Q, Q_R) \stackrel{\text{def}}{=} Q_R[e^\# / \backslash \text{result}]$$

$$\text{awp}(\Gamma, \text{return}, Q, Q_R) \stackrel{\text{def}}{=} Q_R$$

Approximative Verifikationsbedingungen (Revisited)

$$\text{wvc}(\Gamma, \{ \}, Q, Q_R) \stackrel{\text{def}}{=} \emptyset$$

$$\text{wvc}(\Gamma, x = e, Q, Q_R) \stackrel{\text{def}}{=} \emptyset$$

$$\text{wvc}(\Gamma, x = f(e_1, \dots, e_n), Q, Q_R) \stackrel{\text{def}}{=} \{ (R[e_i/x_i][x/\text{result}])^\# \implies Q \}$$

mit $\Gamma(f) = \forall x_1, \dots, x_n. (P, R)$

$$\text{wvc}(\Gamma, f(e_1, \dots, e_n), Q, Q_R) \stackrel{\text{def}}{=} \{ (R[e_i/x_i])^\# \implies Q \}$$

mit $\Gamma(f) = \forall x_1, \dots, x_n. (P, R)$

$$\text{wvc}(\Gamma, \{c \ c_s\}, Q, Q_R) \stackrel{\text{def}}{=} \text{wvc}(\Gamma, c, \text{awp}(\Gamma, \{c_s\}, Q, Q_R), Q_R) \cup \text{wvc}(\Gamma, \{c_s\}, Q, Q_R)$$

$$\text{wvc}(\Gamma, \text{if } (b) \ c_0 \ \text{else } c_1, Q, Q_R) \stackrel{\text{def}}{=} \text{wvc}(\Gamma, c_0, Q, Q_R) \cup \text{wvc}(\Gamma, c_1, Q, Q_R)$$

$$\text{wvc}(\Gamma, /** \{q\} */, Q, Q_R) \stackrel{\text{def}}{=} \{q \implies Q\}$$

$$\text{wvc}(\Gamma, \text{while } (b) \ /** \text{inv } i \ */ \ c, Q, Q_R) \stackrel{\text{def}}{=} \text{wvc}(\Gamma, c, i) \cup \{i \wedge b^\# \implies \text{awp}(\Gamma, c, i, Q_R)\} \cup \{i \wedge !b^\# \implies Q\}$$

$$\text{wvc}(\Gamma, \text{return } e, Q, Q_R) \stackrel{\text{def}}{=} \emptyset$$

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(*y); */
/** post \old(*x) == *y
        && \old(*y) == x; */
{
  int z;

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

$$G = [\text{swap} \mid\!\!\rightarrow \forall \text{forall } x, y. (P = \backslash \text{valid}(*x) \ \&\& \ \backslash \text{valid}(*y), \\ Q = \backslash \text{old}(*x) = *y \ \&\& \ \backslash \text{old}(*y) = x)]$$
$$Q\# = \{ \backslash \text{old}(*x) = \text{read}(s, \text{read}(s, y)) \ \&\& \ \backslash \text{old}(*y) = \text{read}(s, \text{read}(s, x)) \}$$

$$(A) \text{awp}(G, z = *x; *x = *y; *y = z, Q\#, Q\#)$$
$$= \text{awp}(G, z = *x, \text{awp}(G, *x = *y, \text{awp}(G, *y = z, Q\#, Q\#), Q\#), Q\#)$$

// Siehe Einzelberechnungen unten

$$= \{ \backslash \text{old}(*x) = \text{read}(s, \text{read}(s, x)) \ \&\& \ \backslash \text{old}(*y) = \text{read}(s, \text{read}(s, y)) \}$$
$$(A.1) \text{awp}(G, *y = z, Q\#, Q\#)$$
$$= \{ \text{let } s_3 = \text{Upd}(s_2, \text{read}(s_2, y), \text{read}(s_2, z));$$
$$\text{in } \backslash \text{old}(*x) = \text{read}(s_3, \text{read}(s_3, y)) \ \&\& \ \backslash \text{old}(*y) = \text{read}(s_3, \text{read}(s_3, x)) \}$$

Da: $\text{read}(s_3, y) = \text{read}(\text{Upd}(s_2, \text{read}(s_2, y), \text{read}(s_2, z)), y)$

$$= \text{read}(s_3, y) \quad // \text{ da } y \neq \text{read}(s_2, y)$$

Also: $\text{read}(s_3, \text{read}(s_3, y)) = \text{read}(s_3, \text{read}(s_2, y)) = \text{read}(s_2, z)$

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) $\text{read}(s1, y) \neq \text{read}(s1, x)$:

Dann: $\text{read}(s3, \text{read}(s1, x)) = \text{read}(s2, \text{read}(s1, x))$

Also: $\text{read}(s2, \text{read}(s1, x)) = \text{read}(s1, \text{read}(s1, y))$

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: $\text{read}(s1, y) \neq$ (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) $\text{read}(s1, y) = \text{read}(s1, x)$:

Dann war auch: $\text{read}(s, y) = \text{read}(s, x)$:

Dann: $\text{read}(s2, \text{read}(s2, y)) = \text{read}(s1, \text{read}(s1, y))$

Dann: $\text{read}(s3, \text{read}(s1, x)) = \text{read}(s2, z)$
 $= \text{read}(s, \text{read}(s, x))$
 $= \text{read}(s, \text{read}(s, y))$

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) $\text{wvc}(G, \text{swap}) =$

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) $P\# = (\backslash\text{valid}(*x) \ \&\& \ \backslash\text{valid}(*y))\#$
 $= \backslash\text{valid}(\text{read}(s, \text{read}(s, x))) \ \&\& \ \backslash\text{valid}(\text{read}(s, \text{read}(s, y)))$

(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] <
```

```
        {
```

```
            if (a[j]> x) x= a[j];
```

```
            j= j+1;
```

```
        }
```

```
    return x;
```

```
}
```

Fazit

- ▶ Der Hoare-Kalkül ist viel Schreibarbeit
- ▶ 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

- ▶ 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

Es geht Vorwärts.

- ▶ Verifikation nach dem Hoare-Kalkül mit Zeigern:
 - ▶ Viel Schreiarbeit.
- ▶ 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

$$(-)_s^\dagger : \mathbf{St} \rightarrow \mathbf{Lexp} \rightarrow \mathbf{Lexp}$$

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

$$l.id_s^\dagger = l_s^\dagger.id$$

$$l[e]_s^\dagger = l_s^\dagger[e_s^\#]$$

$$*l_s^\dagger = l_s^\#$$

$$(-)_s^\# : \mathbf{St} \rightarrow \mathbf{Aexp} \rightarrow \mathbf{Aexp}$$

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

$$n_s^\# = n$$

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

$$\&e_s^\# = e_s^\dagger$$

$$(e_1 + e_2)_s^\# = e_{1s}^\# + e_{2s}^\#$$

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

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

$$x^\dagger = x_\sigma^\dagger$$

$$e^\# = e_\sigma^\#$$

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

Vorwärts!

- ▶ Alternative Zuweisungsregel (nach Floyd):

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

- ▶ $FV(P)$ 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** $\text{asp}(\Gamma, P, c)$ zusammen mit einer Menge von **Verifikationsbedingungen** $\text{svc}(\Gamma, P, c)$
- ▶ Es gilt:

$$\bigwedge_{p_i \in \text{svc}(\Gamma, P, c)} \forall \sigma. p_i \implies \models \{P\} c \{\text{asp}(\Gamma, P, c) \mid \text{asp}(\Gamma, P, c)\}$$

- ▶ 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:

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

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

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

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

Approximative stärkste Nachbedingung

$$\text{asp}(\Gamma, P, \{\}, Q) \stackrel{\text{def}}{=} P$$

$$\text{asp}(\Gamma, P, x = e, Q) \stackrel{\text{def}}{=} \exists S. P[S/\sigma] \ \&\& \ \sigma == \text{upd}(S, x_S^\dagger, e_S^\#)$$

$$\text{asp}(\Gamma, P, \{\text{return } e; c_S\}, Q) \stackrel{\text{def}}{=} P$$

$$\text{asp}(\Gamma, P, \{\text{return}; c_S\}, Q) \stackrel{\text{def}}{=} P$$

$$\text{asp}(\Gamma, P, \{c; c_S\}, Q) \stackrel{\text{def}}{=} \text{asp}(\Gamma, \text{asp}(\Gamma, P, c, Q), c_S, Q)$$

$$\text{asp}(\Gamma, P, \text{if } (b) \ c_0 \ \text{else } c_1, P, Q) \stackrel{\text{def}}{=} \text{asp}(\Gamma, b^\# \ \&\& \ P, c_0, Q) \\ \parallel \text{asp}(\Gamma, !(b^\#) \ \&\& \ P, c_1, Q)$$

$$\text{asp}(\Gamma, P, /** \{q\} */, Q) \stackrel{\text{def}}{=} q^\#$$

$$\text{asp}(\Gamma, P, \text{while } (b) \ /** \text{inv } i */ \ c, Q) \stackrel{\text{def}}{=} i^\# \ \&\& \ !b^\#$$

$$\text{asp}(\Gamma, P, f(e_1, \dots, e_n), Q) \stackrel{\text{def}}{=} \exists S. P[S/\sigma] \ \&\& \ R_2[(e_i)_S^\# / x_i]$$

$$\text{asp}(\Gamma, P, l = f(e_1, \dots, e_n), Q) \stackrel{\text{def}}{=} \exists S. P[S/\sigma] \ \&\& \\ R_2[(e_i)_S^\# / x_i][l_S^\dagger / \backslash \text{result}]$$

$$\text{mit } \Gamma(f) = \forall x_1, \dots, x_n. (R_1, R_2)$$

Verifikationsbedingungen

$$\text{svc}(\Gamma, P, \{ \}, Q) \stackrel{\text{def}}{=} \emptyset$$

$$\text{svc}(\Gamma, P, x = e, Q) \stackrel{\text{def}}{=} \emptyset$$

$$\text{svc}(\Gamma, P, \{\mathbf{return} \ e; \ c_s\}, Q) \stackrel{\text{def}}{=} \{P \Longrightarrow Q[e^\# / \backslash \mathbf{result}]\}$$

$$\text{svc}(\Gamma, P, \{\mathbf{return}; \ c_s\}, Q) \stackrel{\text{def}}{=} \{P \Longrightarrow Q\}$$

$$\text{svc}(\Gamma, P, \{c \ c_s\}, Q) \stackrel{\text{def}}{=} \text{svc}(\Gamma, P, c, Q) \cup \text{svc}(\Gamma, \text{asp}(\Gamma, P, c, Q), \{c_s\}, Q)$$

$$\text{svc}(\Gamma, P, \mathbf{if} \ (b) \ c_0 \ \mathbf{else} \ c_1, Q) \stackrel{\text{def}}{=} \text{svc}(\Gamma, b^\# \ \&\& \ P, c_0, Q) \cup \text{svc}(\Gamma, \neg(b^\#) \ \&\& \ P, c_1, Q)$$

$$\text{svc}(\Gamma, P, /** \ \{q\} \ */ , Q) \stackrel{\text{def}}{=} \{P \Longrightarrow q^\#\}$$

$$\text{svc}(P, \mathbf{while} \ b \ /** \ \mathbf{inv} \ i \ */ \ c, Q) \stackrel{\text{def}}{=} \{P \Longrightarrow i^\#\} \cup \{\text{asp}(\Gamma, b^\# \ \&\& \ i^\#, c, Q) \Longrightarrow i^\#\} \cup \text{svc}(\Gamma, b^\# \ \&\& \ i^\#, c, Q)$$

$$\text{svc}(\Gamma, P, f(e_1, \dots, e_n), Q) \stackrel{\text{def}}{=} \{P \Longrightarrow R_1[e_i^\# / x_i]\}$$

$$\text{svc}(\Gamma, P, l = f(e_1, \dots, e_n), Q) \stackrel{\text{def}}{=} \{P \Longrightarrow R_1[e_i^\# / x_i][l^\dagger / \backslash \mathbf{result}]\}$$

mit $\Gamma(f) = \forall x_1, \dots, x_n. (R_1, R_2)$

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]<
```

```
        {
```

```
            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:
 - ▶ Defferenzierung 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 $\backslash\text{valid}(x)$ und $\backslash\text{array}(a, n)$

$\backslash\text{valid}(x)$ für x Pointer-Typ $\iff *x$ ist definiert.

$\backslash\text{array}(a, n)$ 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:

$$\backslash\text{valid}(x, S) \stackrel{\text{def}}{=} \exists v. \text{read}(S, \text{read}(S, x)) = v$$

$$\backslash\text{array}(a, n, S) \stackrel{\text{def}}{=} \forall i. 0 \leq i < n \implies \backslash\text{valid}(a[i], S)$$

Wie beweisen wir Validität?

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

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

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

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

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

Program Safety

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

$$safe(x, S) = \emptyset$$

$$safe(*x, S) = \{\backslash\mathbf{valid}(x, S)\}$$

$$safe(a[i]) = \{\backslash\mathbf{array}(a, n) \ \&\& \ 0 \leq i \ \&\& \ i < n\}$$

$$safe(x + y, S) = safe(x, S) \cup safe(y, S)$$

$$safe(x/y, S) = \{y \neq 0\}$$

...

- ▶ Nicht ganz exakt: $y \neq 0$ muss im Zustand S ausgewertet werden,
 - ▶ D.h. $P \implies y \neq 0$ 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**

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

- ▶ Mexp sind Lexp, die auch **Teile** von Feldern bezeichnen.
- ▶ Semantik: $\llbracket - \rrbracket : Env \rightarrow \mathbf{Mexp} \rightarrow \Sigma \rightarrow \mathbb{P}(\mathbf{Loc})$
- ▶ 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

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

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):

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

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 $\langle c, \sigma \rangle \rightarrow \sigma'$
- ▶ Denotational — Partielle Funktion $\llbracket c \rrbracket : \Sigma \rightarrow \Sigma$
- ▶ 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 $\Sigma \rightarrow \Sigma \times \mathbf{V}_U$
 - ▶ 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
 $\Sigma = \mathbf{Loc} \rightarrow \mathbf{V}, \mathbf{V} = \mathbf{N} + \mathbf{C} + \mathbf{Loc}$
 - ▶ 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!

