Korrekte Software: Grundlagen und Methoden
Vorlesung 1 vom 07.04.15: Einführung

Serge Autexier, Christoph Lüth

Universität Bremen

Sommersemester 2016

Organisatorisches

- Veranstalter:

Christoph Lüth	Serge Autexier
christoph.lueth@dfki.de	serge.autexier@dfki.de
MZH 4185, Tel. 59830	Cartesium 2.11, Tel. 59834

- Termine:
  - Vorlesung: Montag, 16 – 18, MZH 1460
  - Übung: Donnerstag, 14 – 16, MZH 1460
- Webseite:

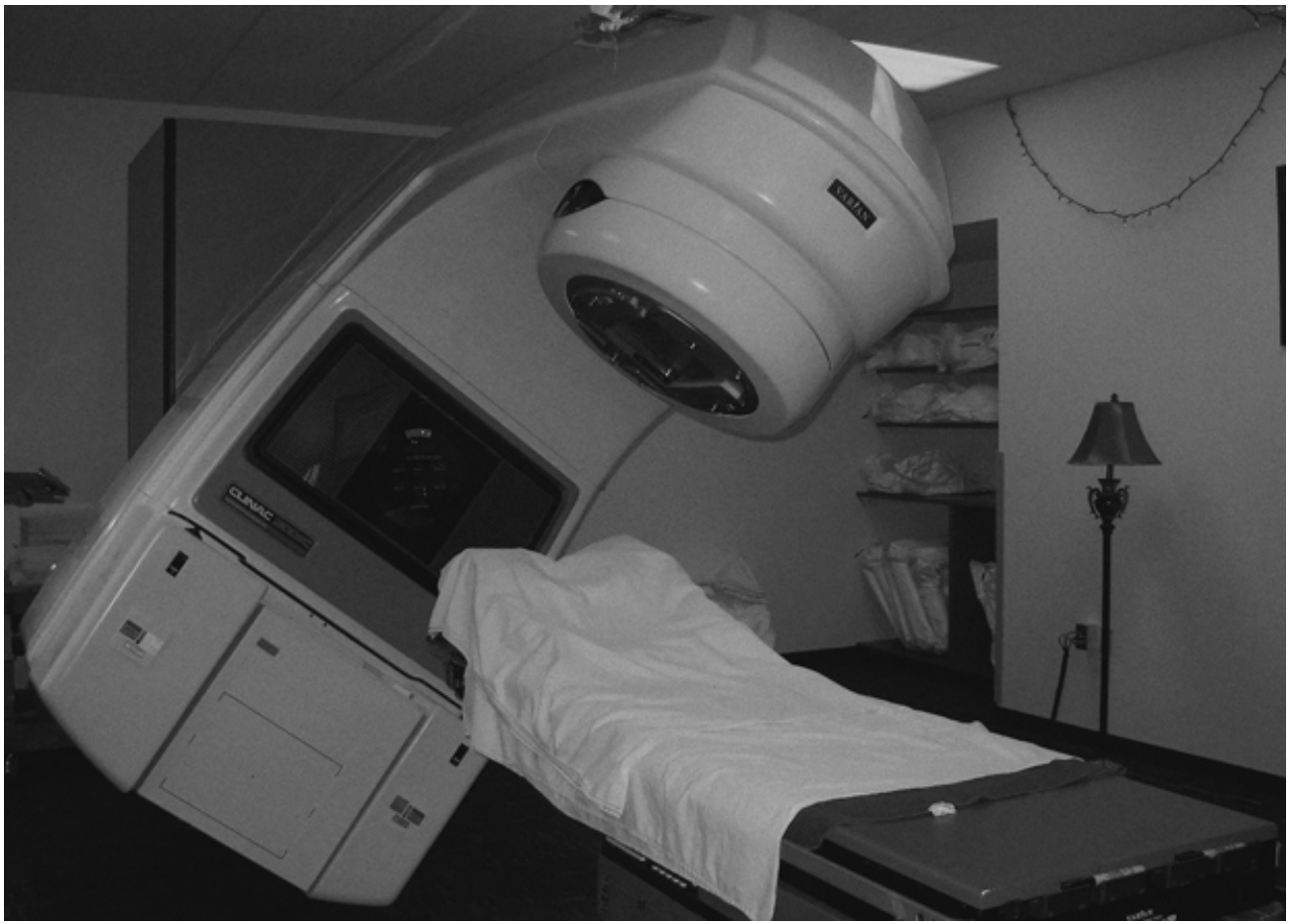
  http://www.informatik.uni-bremen.de/~cxl/lehre/ksgm.ss16

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. Reiche Datenstrukturen (Felder, `struct`)
  2. Funktion und Prozeduren (Modularität)
  3. Referenzen (Zeiger)
- Übungsbetrieb:
  - Betrachtete Programmiersprache: “C0” (erweiterte Untermenge von C)
  - Entwicklung eines Verifikationswerkzeugs in Scala
  - Beweise mit Isabelle (mächtiger Theorembeweiser)

Nächste Woche

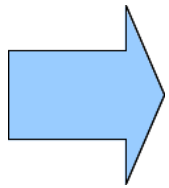
- Aussagenlogik
- Erstes Übungsblatt

Introduction to Scala

Based on the "Scala Training Course" by
Fredrik Vraalsen (fredrik@vraalsen.no) and
Alf Kristian Støyle (alf.kristian@gmail.com)
of scalaBin released under
Creative Commons Attribution 3.0 Unported license

Conciseness

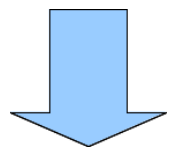
```
public class Person {
  private int age;
  private String name;

  public Person(int age, String
      name) {
    this.age = age;
    this.name = name;
  }

  public int getAge() {
    return this.age;
  }

  public void setAge(int age) {
    this.age = age;
  }

  public String getName() {
    return this.name;
  }

  public void setName(String
      name) {
    this.name = name;
```



```
class Person(var age: Int, var name:
    String)
```

Conciseness

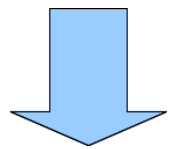
```
List<Person> persons = ...
List<Person> adults = new LinkedList<Person>();
List<Person> kids = new LinkedList<Person>();
for (Person person : persons) {
  if (person.getAge() < 18) {
    kids.add(person);
  } else {
    adults.add(person);
  }
}
```



```
val (kids, adults) = persons.partition (_.age < 18)
```

Conciseness

```
String s = "!em esreveR";
System.out.println(s.reverse());
```

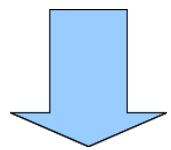


```
val s: java.lang.String = "!em esreveR"
println(s.reverse)
```

=> Reverse me!

Higher-Order

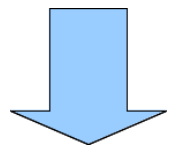
```
List<Person> persons = ...
List<Person> adults = new LinkedList<Person>();
List<Person> kids = new LinkedList<Person>();
for (Person person : persons) {
  if (person.getAge() < 18) {
    kids.add(person);
  } else {
    adults.add(person);
  }
}
```



```
val (kids, adults) = persons.partition (_.age < 18)
```

Java Interaction, Higher-Order

```
BufferedReader reader = null;
try {
  reader = new BufferedReader(new FileReader("f.txt"));
  System.out.println(reader.readLine());
} finally {
  if (reader != null) {
    try {
      reader.close();
    } catch (IOException e) {
      // Exception on close, ignore
    }
  }
}
```



```
using(new BufferedReader(new FileReader("f.txt"))) {
    reader => println(reader.readLine())
}
def using[A, B <: {def close(): Unit}] (closeable: B) (f: B =>
    A): A =
  try { f(closeable) } finally { closeable.close() }
```

```
val myList = List(1, 2, 3)
val res = (10 /: myList) (_+_)
```

=> ??

Scala

- Object oriented and functional
- Statically typed
- Java compatible
  - Compiles to Java bytecode (and CLR)
  - Existing libraries/frameworks
- Better Java

Topics

- Basic syntax
- REPL
- First class functions
- Pattern matching
- OO and traits
- Functional programming
- Higher-Order Functions
- Implicits
- (XML)

Basic Syntax

;

- Is optional (inferred)
- Except if multiple statements in a line

```
val s = "hello"
println(s)
```

```
val s = "hello"; println(s)
```

Type Definitions

Scala

s:String

i:Int

Java

String s

int i / Integer i

Variables

Scala

```
s:String
i:Int


val s = "Hello World"

var i = 1
private var k = 3
```

Java

```
String s
int i / Integer i


public final String s = "Hello
World";
public int i = 1;
private int j = 3;
```

Methods

Scala

```
def add(x: Int, y: Int): Int = {
    x + y
}

def add(x: Int, y: Int) = x + y

def doSomething(text: String) {
}
```

Java

```
public int add(int x, int y) {
    return x + y;
}

public void doSometing(String
    text) {
}
```

Methods

Scala

```
myObject.myMethod(1)
myObject myMethod(1)
myObject myMethod 1

myObject.myOtherMethod(1, 2)
myObject myOtherMethod(1, 2)

myObject.myMutatingMethod()
myObject.myMutatingMethod
// myObject myMutatingMethod
```

Java

```
myObject.myMethod(1);

myObject.myOtherMethod(1, 2);

myObject.myMutatingMethod()
```

Methods

Scala

```
override def toString = ...
```

Java

```
Override
public String toString() {...}
```

Classes And Constructors

Scala

```
class Person(val name: String)
```

Java

```
public class Person {
  private final String name;
  public Person(String name) {
    this.name = name;
  }
  public String getName() {
  return name;
  }
}
```

Traits (= Interface + Mixin)

Scala

```
trait Shape {
  def area: Double
}

class Circle extends Object
      with Shape
```

Java

```
interface Shape {
public double area();
}

public class Circle extends
     Object
        implements Shape
```

No “Static” in Scala

Scala

```
object PersonUtil {
 val AgeLimit = 18

 def countPersons(persons:
       List[Person]) = ...
}
```

Java

```
public class PersonUtil {
 public static final int
       AGE_LIMIT = 18;

 public static int
       countPersons(List<Person>
         persons) {
    ...
 }
}
```

if-then-else

Scala

```
if (foo) {
...
} else if (bar) {
...
} else {
...
}
```

Java

```
if (foo) {
...
} else if (bar) {
...
} else {
...
}
```

For-Loops

Scala

```
for (i <- 0 to 3) {
    ...
}

for (s <- args) println(s)
```

Java

```
for (int i = 0; i < 4; i++) {
    ...
}

for (String s : args) {
    System.out.println(s);
}
```

While-Loops

Scala

```
while (true) {
 ...
}
```

Java

```
while (true) {
 ...
}
```

Exceptions

Scala

```
throw new Exception("...")

try {
} catch {
  case e: IOException => ...
} finally {
}
```

Java

```
throw new Exception("...")

try {
} catch (IOException e) {
 ...
} finally {
}
```

Varargs

Scala

```
def foo(values: String*){ }

foo("bar", "baz")

val arr = Array("bar", "baz")
foo(arr: _*)
```

Java

```
public void foo(String ...
    values){ }

foo("bar", "baz");

String[] arr =
    new String[]{"bar", "baz"}
foo(arr);
```

(Almost) everything is an expression

```
val res = if (foo) x else y

val res = for (i <- 1 to 10) yield i      // List(1, ..., 10)

val res = try { x } catch { ...; y } finally { }  // x or y
```

Collections – List

Scala

```
val numbers = List(1, 2, 3)

val numbers = 1 :: 2 :: 3 :: Nil


numbers(0)
=> 1
```

Java

```
List<Integer> numbers =
    new ArrayList<Integer>();
numbers.add(1);
numbers.add(2);
numbers.add(3);


numbers.get(0);
=> 1
```

Collections – Map

Scala

```
var m = Map(1 -> "apple")
m += 2 -> "orange"


m(1)
=> "apple"
```

Java

```
Map<Int, String> m =
    new HashMap<Int, String>();
m.put(1, "apple");
m.put(2, "orange");

m.get(1);
=> apple
```

Generics

Scala

```
List[String]
```

Java

```
List<String>
```

Tuples

Scala

```
val tuple: Tuple2[Int, String] =
(1, "apple")

val quadruple =
(2, "orange", 0.5d, false)
```

Java

```
Pair<Integer, String> tuple =
    new Pair<Integer, String>(1,
    "apple")


... ;-)
```

Packages

Scala

```
package mypackage
...
```

Java

```
package mypackage;
...
```

Imports

Scala

```
import java.util.{List,
       ArrayList}


import java.io._

import java.sql.{Date => SDate}
```

Java

```
import java.util.List
import java.util.ArrayList

import java.io.*

???
```

Nice to Know

Scala

```
println("Hello")

val line = readLine()


sys.error("Bad")

1 + 1
1.+(1)

1 == new Object
1 eq new Object

"""A\sregex""".r

s"3 + 4 = ${3 + 4}" // "3 +
    4 = 7"
```

Java

```
System.out.println("Hello");

BufferedReader r = new BufferedReader(new
    InputStreamRead(System.in));
String line = r.readLine();

throw new RuntimeException("Bad")

new Integer(1).toInt() + new
    Integer(1).toInt();


new Integer(1).equals(new Object());
new Integer(1) == new Object();

java.util.regex.Pattern.compile("A\\sregex");

"3 + 4 = " + (3 + 4)
```

Topics

- Basic syntax
- REPL
- First class functions
- Pattern matching
- OO and traits
- Functional programming
- Higher-Order Functions
- Implicits
- (XML)

REPL - Read eval print loop

- Command line shell for on-the-fly execution of Scala statements
- bin/scala

IDE and Build Tools

- Scala IDE for Eclipse is the officially supported Platform by the creators of Scala.`http://scala-ide.org/`
- Scala Plugin for IDEA is very good too. (And IDEA is avaliable in a free edition)
- There used to be support for Netbeans, but that seems to be dead right now.

Build Tool

- SBT
  (Scala Build Tool) is an Mawen compatible build tool for Scala and Java
  `http://www.scala-sbt.org/`

First Class Functions

```
val even = Function[Int, Boolean] {
    def apply(i: Int) = i % 2 == 0
}

val even: (Int => Boolean) = (i: Int) => i % 2 == 0
val even = (i: Int) => i % 2 == 0

even.apply(42)        // true
even(13)              // false
```

First Class Functions

```
val numbers = List(1, 2, 3, 4, 5)

numbers.filter(even)                    // List(2, 4)

numbers.filter((i: Int) => i > 2)       // List(3, 4, 5)
numbers.filter(i => i > 2)              // List(3, 4, 5)
numbers.filter(_ > 2)                   // List(3, 4, 5)
```

Collections

```
numbers. filter (i => i > 2)                    // List(3, 4, 5)
numbers.find(i => i > 2)                        // Some(3)
numbers.exists(i => i > 2)                      // true
numbers. forall (i => i > 2)                    // false

numbers.map(i => i *2)                          // List(2, 4, 6, 8, 10)

numbers.foldLeft(0) { (a, b) => a + b }         // 15
```

Deferred execution - constructed example

```
helloButton.addActionListener(e =>
      println("Hello World!")
)
```

Closure

```
val people = List(Person("Alf"), Person("Fredrik"))

val name = "Fredrik"
val nameFilter = (p: Person) => p.name == name

people.filter(nameFilter) // Person("Fredrik")
```

Closures

```
val people = List(Person("Alf"), Person("Fredrik"))

var name = "Fredrik"
val nameFilter = (p: Person) => p.name == name

people.filter(nameFilter)    // Person("Fredrik")
name = "Alf"
people.filter(nameFilter)    // Person(Älf")
```

Pattern Matching

```
myObject match {
  case 1 => println("First was hit")
  case 2 => println("Second was Hit")
  case _ => println("Unknown")
}
```

Pattern Matching

```
myObject match {
  case i: Int => println("Found an int")
  case s: String => println("Found a String")
  case _ => println("Unknown")
}
```

Pattern Matching

```
myObject match {
  case i: Int => println("Found an int")
  case s: String => println("Found an String")
  case other => println("Unknown " + other)
}
```

Pattern Matching

```
myObject match {
  case i: Int if i == 1 => println("Found an int")
  case s: String => println("Found a String")
  case other => println("Unknown " + other)
}
```

Pattern Matching

```
val res = myObject match {
    case i: Int if i == 1 => "Found an int"
    case s: String => "Found a String"
    case other => "Unknown " + other
}
```

Pattern Matching

```
val res = myObject match {
    case (first, second) => second
    case (first, second, third) => third
}
```

Pattern Matching

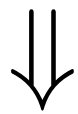
```
val mathedElement = list match {
  case List(firstElement, lastElement) => firstElement
  case List(firstElement, _*) => firstElement
  case _ => "failed"
}
```

Pattern Matching

```
def length( list : List [_]): Int =
  list match {
    case Nil => 0
    case head :: tail => 1 + length(tail)
  }
```

Pattern Matching

```
public static Integer getSecondOr0(List<Integer> list) {
    if (list != null && list.size() >= 2) {
        return list.get(1);
    } else {
        return 0;
    }
}
```



```
def second_or_0(list: List[Int]) = list match {
  case List(_, x, _*) => x
  case _ => 0
}
```

Case classes

- Class types that can be used in pattern matching
- Generated into your class:
  - `equals`
  - `hashCode`
  - `toString`

Case classes

```
abstract class Person(name: String)
case class Man(name: String) extends Person(name)
case class Woman(name: String, children: List[Person])
  extends Person(name)
```

Case Classes

```
p match {
  case Man(name) => println("Man with name " + name)
  case Woman(name, children) => println("Woman with name" +
    name + " and with " + children.size + " children")
}
```

Regular Expressions

```
val regex = """(\d+)(\w+)""".r

val myString = ...

val res: String = myString match {
  case regex(digits, word) => digits
  case _ => "None"
}
```

Regular Expressions

```
val regex = """(\d+)(\w+)""".r

val myString = ...

val res: Option[String] = myString match {
  case regex(digit, word) => Some(digit)
  case _ => None
}
```

Options

- Never NullPointerException again!
- Option has two possible values:
  - `Some(value)`
  - `None`

```
val someOption: Option[String] = Some("value")
val noOption:   Option[String] = None
```

Options

```
def getValue(s: Any): Option[String]


getValue(object) match {
  case Some(value) => println(value)
  case None => println("Nothing")
}


val result = getValue(object).getOrElse("Nothing")
```

Korrekte Software: Grundlagen und Methoden
Vorlesung 2 vom 10.04.16: Die Floyd-Hoare-Logik

Serge Autexier, Christoph Lüth

Universität Bremen

Sommersemester 2016

Fahrplan

- Einführung
- Die Floyd-Hoare-Logik
- Operationale Semantik
- Denotationale Semantik
- Äquivalenz der Semantiken
- Verifikation: Vorwärts oder Rückwärts?
- Korrektheit des Hoare-Kalküls
- Einführung in Isabelle/HOL
- Weitere Datentypen: Strukturen und Felder
- Funktionen und Prozeduren
- Referenzen und Zeiger
- Frame Conditions & Modification Clauses
- Ausblick und Rückblick

Idee

- Was wird hier berechnet?

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

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;
}
```

Idee

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

$\{1 \leq n\}$

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

$\{p = n!\}$

Idee

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

$\{1 \leq n\}$

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

$\{p = n!\}$

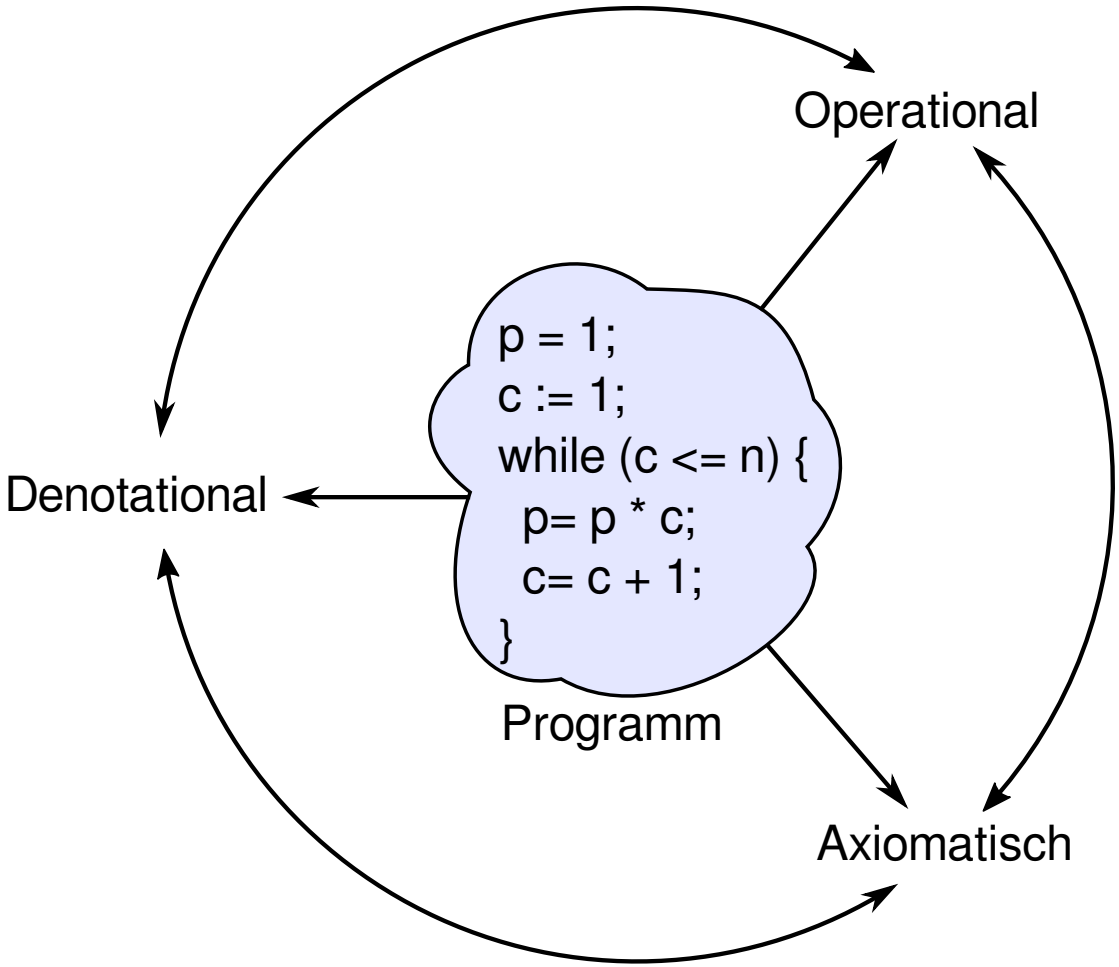
- Um Aussagen über ein Program zu beweisen, benötigen wir einen Formalismus (eine Logik), die es erlaubt, Zusicherungen über Werte von Variablen zu bestimmten Ausführungszeitpunkten (im Programm) aufzuschreiben und zu beweisen.
- Dazu müssen wir auch die Bedeutung (Semantik) des Programmes definieren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

Semantik von Programmiersprachen

Drei wesentliche Möglichkeiten:

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

Drei Semantiken — Eine Sicht



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

Floyd-Hoare-Logik

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

Unsere Programmiersprache

Wir betrachten einen Ausschnitt der Programmiersprache C (C0).

Ausbaustufe 1 kennt folgende Konstrukte:

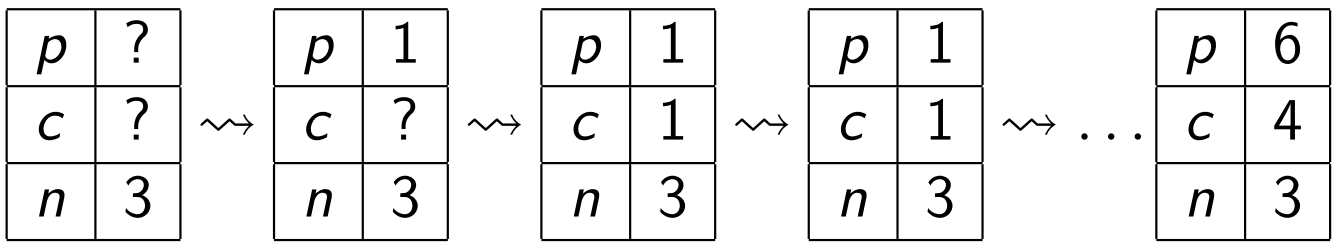
- Typen: **int**;
- Ausdrücke: Variablen, Literale (für ganze Zahlen), arithmetische Operatoren (für ganze Zahlen), Relationen (==, !=, <=, ...), boolsche Operatoren (&&, ||);
- Anweisungen:
  - Fallunterscheidung (**if**...**else**...), Iteration (**while**), Zuweisung, Blöcke;
  - Sequenzierung und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

C0: Ausdrücke und Anweisungen

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

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: $n = 3$



Systemzustände

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

Floyd-Hoare-Tripel

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

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

Totale Korrektheit ($\models [P]\, c\, [Q]$)

c ist total korrekt, wenn für alle Zustande σ, die P erfüllen:
die Ausführung von c mit σ in σ' terminiert, und σ' erfüllt Q.

- $\models \{\mathbf{1}\}$ **while**$(1)\{\ \}\, \{\mathbf{1}\}$ gilt
- $\models [\mathbf{1}]$ **while**$(1)\{\ \}\, [\mathbf{1}]$ gilt nicht

Regeln der Floyd-Hoare-Logik

- Die Floyd-Hoare-Logik erlaubt es, Zusicherungen der Form $\vdash \{P\}\, c\, \{Q\}$ syntaktisch herzuleiten.

- Der Kalkül der Logik besteht aus sechs Regeln der Form

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

- Für jedes Konstrukt der Programmiersprache gibt es eine Regel.

Regeln der Floyd-Hoare-Logik: Zuweisung

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

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

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

x = x+ 1

Regeln der Floyd-Hoare-Logik: Zuweisung

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

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

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

x = x+ 1

Regeln der Floyd-Hoare-Logik: Zuweisung

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

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

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

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

Regeln der Floyd-Hoare-Logik: Zuweisung

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

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

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

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

Regeln der Floyd-Hoare-Logik: Fallunterscheidung und Sequenzierung

$$\frac{\vdash \{A \,\&\&\, [\![b]\!]\}\, c_0\, \{B\} \qquad \vdash \{A \,\&\&\, \neg[\![b]\!]\}\, c_1\, \{B\}}{\vdash \{A\}\ \textbf{if}\ (b)\ c_0\ \textbf{else}\ c_1\, \{B\}}$$

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

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

- Hier wird ein Zwischenzustand B benötigt.

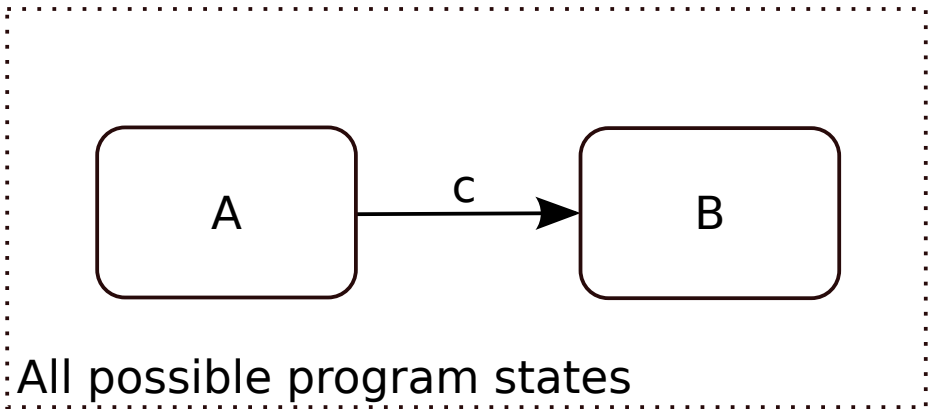
Regeln der Floyd-Hoare-Logik: Iteration

$$\frac{\vdash \{A \wedge [\![b]\!]\}\, c\, \{A\}}{\vdash \{A\}\ \textbf{while}(b)\ c\, \{A \wedge \neg [\![b]\!]\}}$$

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

Regeln der Floyd-Hoare-Logik: Weakening

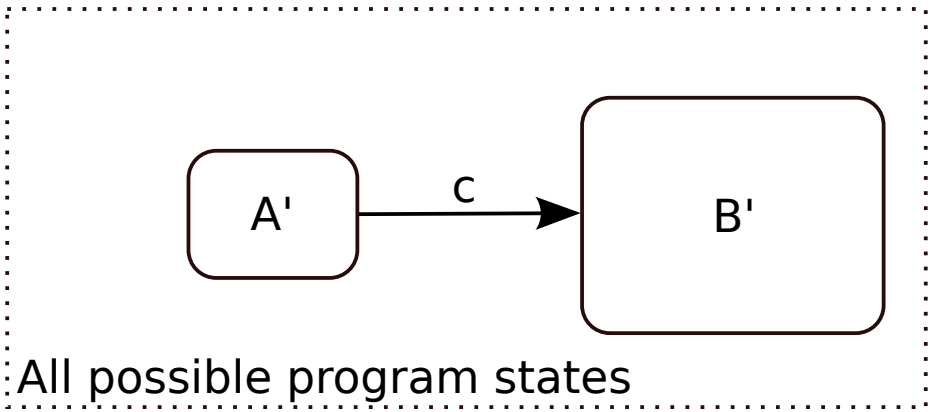
$$\frac{A' \longrightarrow A \qquad \vdash \{A\}\, c\, \{B\} \qquad B \longrightarrow B'}{\vdash \{A'\}\, c\, \{B'\}}$$



- $\models \{A\}\, c\, \{B\}$: Ausführung von c startet in Zustand, in dem A gilt, und endet (ggf) in Zustand, in dem B gilt.
- Zustandsprädikate beschreiben Mengen von Zuständen: $P \subseteq Q$ gdw. $P \longrightarrow Q$.

Regeln der Floyd-Hoare-Logik: Weakening

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



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

Überblick: die Regeln des Floyd-Hoare-Kalküls

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

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

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

$$\frac{\vdash \{A \wedge [\![b]\!]\}\, c\, \{A\}}{\vdash \{A\}\ \textbf{while}(b)\ c\, \{A \wedge \neg [\![b]\!]\}}$$

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

Eigenschaften der Floyd-Hoare-Logik

Korrektheit

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

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

Relative Vollständigkeit

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

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

Wie wir Floyd-Hoare-Beweise aufschreiben

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

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

Warum Verifikation?

Hier sind Varianten des Fakultätsbeispiels.
Welche sind korrekt?

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

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

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

Eine Handvoll Beispiele

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

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

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

Eine Handvoll Beispiele

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

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

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

Zusammenfassung

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

Korrekte Software: Grundlagen und Methoden
Vorlesung 3 vom 18.04.16: Operationale Semantik

Serge Autexier, Christoph Lüth

Universität Bremen

Sommersemester 2016

Fahrplan

- Einführung
- Die Floyd-Hoare-Logik
- Operationale Semantik
- Denotationale Semantik
- Äquivalenz der Semantiken
- Verifikation: Vorwärts oder Rückwärts?
- Korrektheit des Hoare-Kalküls
- Einführung in Isabelle/HOL
- Weitere Datentypen: Strukturen und Felder
- Funktionen und Prozeduren
- Referenzen und Zeiger
- Frame Conditions & Modification Clauses
- 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 (==, !=, <=, ...), boolsche Operatoren (&&, ||);
- Anweisungen:
  - Fallunterscheidung (**if**...**else**...), Iteration (**while**), Zuweisung, Blöcke;
  - Sequenzierung und leere Anweisung sind implizit

Semantik von C0

Systemzustände

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

C0: Ausdrücke und Anweisungen

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

Eine Handvoll Beispiele

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

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

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

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

Operationale Semantik: Arithmetische Ausdrücke

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

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

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

Operationale Semantik: Arithmetische Ausdrücke

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

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

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

Regeln

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

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)

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

Regeln

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

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

Operationale Semantik: Arithmetische Ausdrücke

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

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

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

Regeln

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

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

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

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

Operationale Semantik: Arithmetische Ausdrücke

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

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

Operationale Semantik: Arithmetische Ausdrücke

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

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

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

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

Operationale Semantik: Arithmetische Ausdrücke

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

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

Beispiel Ableitungen

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

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

Beispiel Ableitungen

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

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

Beispiel Ableitungen

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

$$\dfrac{\dfrac{\langle X, \sigma\rangle \to_{Aexp} 6}{\langle X + Y, \sigma\rangle \to_{Aexp}} \qquad \dfrac{}{\langle X - Y, \sigma\rangle \to_{Aexp}}}{\langle (X + Y) * (X - Y), \sigma\rangle \to_{Aexp}}$$

Beispiel Ableitungen

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

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

Beispiel Ableitungen

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

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

Beispiel Ableitungen

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

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

Beispiel Ableitungen

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

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

Beispiel Ableitungen

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

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

Beispiel Ableitungen

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

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

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

Beispiel Ableitungen

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

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

$$\dfrac{\dfrac{\langle X, \sigma\rangle \to_{Aexp} 6 \qquad \langle X, \sigma\rangle \to_{Aexp} 6}{\langle X * X, \sigma\rangle \to_{Aexp} 36}}{\langle (X * X) - (Y * Y), \sigma\rangle \to_{Aexp}}$$

Beispiel Ableitungen

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

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

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

Beispiel Ableitungen

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

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

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

Operationale Semantik: Boolesche Ausdrücken

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

Rules

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

Operationale Semantik: Boolesche Ausdrücken

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

Rules

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

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

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

Operationale Semantik: Boolesche Ausdrücken

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

Rules

$$\frac{\langle a_1, \sigma\rangle \to_{Aexp} n_1 \qquad \langle a_2, \sigma\rangle \to_{Aexp} n_2 \qquad n_i \neq \bot, n_1 \text{ kleiner/gleich } n_2}{\langle a_1 <= a_2, \sigma\rangle \to_{Bexp} 1}$$

$$\frac{\langle a_1, \sigma\rangle \to_{Aexp} n_1 \qquad \langle a_2, \sigma\rangle \to_{Aexp} n_2 \qquad n_i \neq \bot, n_1 \text{ größer als } n_2}{\langle a_1 <= a_2, \sigma\rangle \to_{Bexp} 0}$$

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

Operationale Semantik: Boolesche Ausdrücken

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

Rules

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

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

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

Operationale Semantik: Boolesche Ausdrücken

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

Rules

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

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

Operationale Semantik: Anweisungen

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

Regeln

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

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

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

Operationale Semantik: Anweisungen

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

Regeln

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

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

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

Definiere :

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

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

Operationale Semantik: Anweisungen

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

Regeln

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

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

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

$$\frac{\langle a, \sigma \rangle \to_{Aexp} \bot}{\langle X = a, \sigma \rangle \to_{Stmt} \bot}$$

Operationale Semantik: Anweisungen

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

Regeln

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

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

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

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

Operationale Semantik: Anweisungen

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

Regeln

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

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

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

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

Operationale Semantik: Anweisungen

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

Regeln

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

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

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

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

Beispiel

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

$\sigma(y) = 3$

Äquivalenz arithmetischer Ausdrücke

Gegeben zwei Aexp a_1 and a_2

- Sind sie gleich?

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

(X∗X) + 2∗X∗Y + (Y∗Y) und (X+Y) ∗ (X+Y)

- Wann sind sie gleich?

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

X∗X und 9∗X+22
X∗X und X∗X+1

Äquivalenz Boolscher Ausdrücke

Gegeben zwei Bexp-Ausdrücke b_1 and b_2

- Sind sie gleich?

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

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

Beweis an der Tafel

Zusammenfassung

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

Korrekte Software: Grundlagen und Methoden
Vorlesung 4 vom 25.04.16: Denotationale Semantik

Serge Autexier, Christoph Lüth

Universität Bremen

Sommersemester 2016

Beweisen

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

Definition

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

Ein einfaches Beispiel:

Lemma

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

Beweis an der Tafel

Fahrplan

- Einführung
- Die Floyd-Hoare-Logik
- Operationale Semantik
- Denotationale Semantik
- Äquivalenz der Semantiken
- Verifikation: Vorwärts oder Rückwärts?
- Korrektheit des Hoare-Kalküls
- Einführung in Isabelle/HOL
- Weitere Datentypen: Strukturen und Felder
- Funktionen und Prozeduren
- Referenzen und Zeiger
- Frame Conditions & Modification Clauses
- Ausblick und Rückblick

Überblick

- Kleinster Fixpunkt
- Denotationale Semantik für C0

Regeln und Regelinstanzen

Definition

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

- Betrachte folgende Regelmenge R

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

- Regelinstanzen sind

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

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

Induktive Definierte Mengen

Definition

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

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

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

Beispiel

- Betrachte folgende Regelmenge R

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

- Was sind

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

Induktive Definierte Mengen

Definition

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

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

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

Definition (Abgeschlossen und Monoton)

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

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

Kleinster Fixpunkt Operator

Lemma

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

Lemma

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

(a) *A ist R-abgeschlossen,*

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

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

Beweis von Lemma (a).

A ist R-abgeschlossen:

Sei $\frac{x_1,\ldots,x_k}{y} \in R$ und $x_1,\ldots,x_k \subseteq A$. Da $A = \bigcup_{i\in\mathbb{N}} A_i$ gibt es ein l so dass $x_1,\ldots,x_k \subseteq A_l$. Also auch:
$y \in \hat{R}(A_l) = \hat{R}(\hat{R}^l(\emptyset)) = \hat{R}^{l+1}(\emptyset) = A_{l+1} \subseteq A$. □

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

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

□

Beweis von Lemma (c).

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

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

Basisfall $A_0 = \emptyset \subseteq B$

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

□

Kleinster Fixpunkt Operator

Definition

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

ist der kleinste Fixpunkt.

Kleinster Fixpunkt

- Betrachte folgende Regelmenge R

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

- Was sind

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

Kleinster Fixpunkt

- Betrachte folgende Regelmenge R

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

- Was sind

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

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

Denotationale Semantik - Motivation

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

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

- Denotationale Semantik: Eine Menge von Regeln, die ein Programm in eine $\underbrace{\text{partielle Funktion}}_{\text{Denotat}}$ von Zustand nach Zustand überführen

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

Denotationale Semantik - Motivation

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

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

or

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

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

Denotierte Funktionen

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

Denotat von **Aexp**

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

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

Denotat von **Bexp**

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

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

Denotat von **Bexp**

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

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

Denotat von **Stmt**

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

$$\begin{array}{rcll}
\mathcal{D}[\![x = a]\!] & = & \{(\sigma, \sigma(x \mapsto n) \mid \sigma \in \Sigma \wedge (\sigma, n) \in \mathcal{E}[\![a]\!]\} & \\
\mathcal{D}[\![\{c\ c_s\}]\!] & = & \mathcal{D}[\![c]\!] \circ \mathcal{D}[\![c_s]\!] & \text{Komposition von Relationen} \\
\mathcal{D}[\![\{\ \}]\!] & = & \textbf{Id} & \textbf{Id} := \{(\sigma, \sigma) | \sigma \in \Sigma\} \\
\mathcal{D}[\![\textbf{if}\ (b)\ c_0\ \textbf{else}\ c_1]\!] & = & \{(\sigma, \sigma') \mid (\sigma, 1) \in \mathcal{B}[\![b]\!] \wedge (\sigma, \sigma') \in \mathcal{D}[\![c_0]\!]\} & \\
 & & \cup \{(\sigma, \sigma') \mid (\sigma, 0) \in \mathcal{B}[\![b]\!] \wedge (\sigma, \sigma') \in \mathcal{D}[\![c_1]\!]\} &
\end{array}$$

Denotat von **Stmt**

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

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

Aber was ist

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

Denotationale Semantik für **while**

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

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

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

Denotationale Semantik für **while**

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

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

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

Denotationale Semantik für **while**

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

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

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

Denotationale Semantik von **while**

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

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

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

Denotationale Semantik von **while**

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

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

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

Denotationale Semantik von **while**

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

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

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

Denotationale Semantik von **while**

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

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

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

Denotationale Semantik von **while**

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

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

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

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

Denotationale Semantik von **while**

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

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

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

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

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

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

Denotation für **Stmt**

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

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

mit

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

Weitere Intuition zur Fixpunkt Konstruktion

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

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

- Beweis an der Tafel

Korrekte Software: Grundlagen und Methoden
Vorlesung 5 vom 2.05.16: Äquivalenz operationale und denotationale Semantik

Serge Autexier, Christoph Lüth

Universität Bremen

Sommersemester 2016

Operationale vs. denotationale Semantik

	Operational $\langle a, \sigma \rangle \to_{Aexp} n$	Denotational $\mathcal{E}[\![a]\!]$
m$\in$ **N**	$\langle m, \sigma \rangle \to_{Aexp} m$	$\{(\sigma, m) \mid \sigma \in \Sigma\}$
x$\in$ **Loc**	$\dfrac{x \in Dom(\sigma)}{\langle x, \sigma \rangle \to_{Aexp} \sigma(x)}$	$\{(\sigma, \sigma(x)) \mid \sigma \in \Sigma, x \in Dom(\sigma)\}$
$a_1 \circ a_2$	$\dfrac{\langle a_1, \sigma \rangle \to_{Aexp} n \quad \langle a_2, \sigma \rangle \to_{Aexp} m \quad n, m \neq \bot}{\langle a_1 \circ a_2, \sigma \rangle \to_{Aexp} n \circ^I m}$	$\{(\sigma, n \circ^I m) \mid \sigma \in \Sigma, (\sigma, n) \in \mathcal{E}[\![a_1]\!], (\sigma, m) \in \mathcal{E}[\![a_2]\!]\}$

$$\frac{\langle a_1, \sigma \rangle \to_{Aexp} n \quad \langle a_2, \sigma \rangle \to_{Aexp} m \quad n = \bot \text{ oder } m = \bot}{\langle a_1 \circ a_2, \sigma \rangle \to_{Aexp} \bot} \quad \circ \in \{+, \times, -\}$$

Operationale vs. denotationale Semantik

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

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

Equivalenz operationale und denotationale Semantik

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

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

$$\langle a, \sigma \rangle \to_{Aexp} \perp \Leftrightarrow \sigma \notin Dom(\mathcal{E}[\![a]\!])$$

- Beweis per struktureller Induktion über a.

Operationale vs. denotationale Semantik

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

Operationale vs. denotationale Semantik

	Operational $\langle a, \sigma\rangle \to_{Bexp} b$	Denotational $\mathcal{B}[\![b]\!]$
$a_0 == a_1$	$\dfrac{\langle a_0,\sigma\rangle \to_{Aexp} n \quad \langle a_1,\sigma\rangle \to_{Aexp} m \quad n,m \neq \bot \quad n = m}{\langle a_0 == a_1,\sigma\rangle \to_{Aexp} 1}$	$\{(\sigma,1) \mid \sigma \in \Sigma, (\sigma,n_0) \in \mathcal{E}[\![a_0]\!](\sigma), (\sigma,n_1) \in \mathcal{E}[\![a_1]\!], n_0 = n_1\} \cup$
	$\dfrac{\langle a_0,\sigma\rangle \to_{Aexp} n \quad \langle a_1,\sigma\rangle \to_{Aexp} m \quad n,m \neq \bot \quad n \neq m}{\langle a_0 == a_1,\sigma\rangle \to_{Aexp} 0}$	$\{(\sigma,0) \mid \sigma \in \Sigma, (\sigma,n_0) \in \mathcal{E}[\![a_0]\!](\sigma), (\sigma,n_1) \in \mathcal{E}[\![a_1]\!], n_0 \neq n_1\}$
	$\dfrac{\langle a_0,\sigma\rangle \to_{Aexp} n \quad \langle a_1,\sigma\rangle \to_{Aexp} m \quad n = \bot \text{ oder } m = \bot}{\langle a_0 == a_1,\sigma\rangle \to_{Aexp} \bot}$	

$<=$ analog

Operationale vs. denotationale Semantik

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

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

Operationale vs. denotationale Semantik

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

Operationale vs. denotationale Semantik

Operational $\langle c, \Sigma \rangle \to_{Stmt} \Sigma | \bot$ Denotational $\mathcal{D}[\![c]\!]$

$\underbrace{\textit{while}\ (b)\ c}_{w}$ $\quad \dfrac{\langle b, \sigma \rangle \to_{Bexp} 0}{\langle w, \sigma \rangle \to_{Stmt} \sigma} \quad \dfrac{\langle b, \sigma \rangle \to_{Bexp} \bot}{\langle w, \sigma \rangle \to_{Stmt} \bot}$ $\qquad fix(\Gamma)$

$$\frac{\langle b, \sigma \rangle \to_{Bexp} 1 \quad \langle c, \sigma \rangle \to_{Stmt} \sigma' \neq \bot \quad \langle w, \sigma' \rangle \to_{Stmt} \sigma''}{\langle w, \sigma \rangle \to_{Stmt} \sigma''}$$

$$\frac{\langle b, \sigma \rangle \to_{Bexp} 1 \quad \langle c, \sigma \rangle \to_{Stmt} \bot}{\langle w, \sigma \rangle \to_{Stmt} \bot}$$

mit

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

Equivalenz operationale und denotationale Semantik

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

$$\langle c, \sigma \rangle \rightarrow_{Stmt} \sigma' \Leftrightarrow (\sigma, \sigma') \in \mathcal{D}[\![c]\!]$$

$$\langle c, \sigma \rangle \rightarrow_{Stmt} \bot \Rightarrow \sigma \notin Dom(\mathcal{D}[\![c]\!])$$

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

Korrekte Software: Grundlagen und Methoden
Vorlesung 6 vom 09.05.16: Vorwärts und Rückwärts mit Floyd und Hoare.

Serge Autexier, Christoph Lüth

Universität Bremen

Sommersemester 2016

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:

  // $\{X = x \wedge Y = y\}$
  z = y ;
  // $\{X = x \wedge Y = z\}$
  y = x ;
  // $\{X = y \wedge Y = z\}$
  x = z ;
  // $\{X = y \wedge Y = x\}$

Idee

- Hier ist ein einfaches Programm:

  // $\{X = x \wedge Y = y\}$
  z = y ;
  // $\{X = x \wedge Y = z\}$
  y = x ;
  // $\{X = y \wedge Y = z\}$
  x = z ;
  // $\{X = y \wedge Y = x\}$

- Wir sehen:

  1. Die Verifikation erfolgt rückwärts (von hinten nach vorne).
  2. Die Verifikation kann berechnet werden.

Idee

- Hier ist ein einfaches Programm:

  // $\{X = x \wedge Y = y\}$
  z = y ;
  // $\{X = x \wedge Y = z\}$
  y = x ;
  // $\{X = y \wedge Y = z\}$
  x = z ;
  // $\{X = y \wedge Y = x\}$

- 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?

Vorwärts!

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

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

Vorwärts!

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

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

- 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.

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:

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

Vorwärtsverkettung

- Vorwärtsaxiom äquivalent zum Rückwärtsaxiom.
- In der Anwendung umständlicher.
- Vereinfachung benötigt Lemma:

$$\exists x.P(x) \wedge x = t \longleftrightarrow P(t)$$

- Vorteile?
  - Wir wollten doch sowieso die Anwendung automatisieren...

Schwächste Vorbedingung, stärkste Nachbedingung

- Prädikat P schwächer als Q wenn $Q \longrightarrow P$ (stärker wenn $P \longrightarrow Q$).
- Gegeben C0-Programm c, Prädikat P, dann ist
  - $\mathrm{wp}(c, P)$ die schwächste Vorbedingung Q so dass $\models \{Q\}\, c\, \{P\}$;
  - $\mathrm{sp}(P, c)$ die stärkste Nachbedingung Q so dass $\models \{P\}\, c\, \{Q\}$.

Schwächste Vorbedingung, stärkste Nachbedingung

- Prädikat P schwächer als Q wenn $Q \longrightarrow P$ (stärker wenn $P \longrightarrow Q$).
- Gegeben C0-Programm c, Prädikat P, dann ist
  - $\mathsf{wp}(c, P)$ die schwächste Vorbedingung Q so dass $\models \{Q\}\, c\, \{P\}$;
  - $\mathsf{sp}(P, c)$ die stärkste Nachbedingung Q so dass $\models \{P\}\, c\, \{Q\}$.
- Semantische Charakterisierung:

$$\begin{array}{lcl} \models \{P\}\, c\, \{Q\} & \longleftrightarrow & P \longrightarrow \mathsf{wp}(c, Q) \\ \models \{P\}\, c\, \{Q\} & \longleftrightarrow & \mathsf{sp}(P, c) \longrightarrow Q \end{array}$$

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

- Einfach für Programme ohne Schleifen:

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

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

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

  - Hilft auch nicht weiter...

Lösung: Annotierte Programme

- Wir helfen dem Rechner weiter und annotieren die Schleifeninvariante am Programm.
- Damit berechnen wir:
  - die approximative schwächste Vorbedingung $\mathsf{awp}(c, Q)$ zusammen mit einer Menge von Verifikationsbedingungen $\mathsf{wvc}(c, Q)$
  - oder die approximative stärkste Nachbedingung $\mathsf{asp}(P, c)$ zusammen mit einer Menge von Verifikationsbedingungen $\mathsf{svc}(P, c)$
- Es gilt:

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

Approximative schwächste Vorbedingung

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

Approximative stärkste Nachbedingung

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

$$
\begin{aligned}
\mathrm{svc}(P, \{\,\}) &\stackrel{def}{=} \emptyset \\
\mathrm{svc}(P, x = e) &\stackrel{def}{=} \emptyset \\
\mathrm{svc}(P, \{c\ c_s\}) &\stackrel{def}{=} \mathrm{svc}(P, c) \cup \mathrm{svc}(\mathrm{asp}(P, c), \{c_s\}) \\
\mathrm{svc}(P, \textbf{if}\ (b)\ c_0\ \textbf{else}\ c_1) &\stackrel{def}{=} \mathrm{svc}([\![b]\!] \wedge P, c_0) \cup \mathrm{svc}(\neg[\![b]\!] \wedge P, c_1) \\
\mathrm{svc}(P, /{**}\ \{q\}\ {*}/) &\stackrel{def}{=} \{P \longrightarrow [\![q]\!]\} \\
\mathrm{svc}(P, \textbf{while}\ b\ /{**}\ \textbf{inv}\ i\ {*}/\ c) &\stackrel{def}{=} \{P \longrightarrow [\![i]\!]\} \\
&\quad \cup \{\mathrm{asp}([\![b]\!] \wedge [\![i]\!], c) \longrightarrow [\![i]\!]\} \\
&\quad \cup\ \mathrm{svc}([\![b]\!] \wedge [\![i]\!], c)
\end{aligned}
$$

Beispiel: das Fakultätsprogramm

- In der Praxis sind Vor- und Nachbedingung gegeben, und nur die Verifikationsbedingungen relevant:

Sei F das annotierte Fakultätsprogramm:

```
// {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 vorwärts: $\mathrm{svc}(F, \top)$
- Berechnung rückwärts: $\mathrm{wvc}(\top, F)$

Zusammenfassung

- Die Zuweisungsregel gibt es“rückwärts” und “vorwärts”.
- Bis auf die Invarianten an Schleifen können wir Korrektheit automatisch prüfen.

Rückwärtsberechnung:

- Einfacher zu berechnen
- Führt zu großen Formeln
- Keine Möglichkeit, Zwischenzustände zu vereinfachen

Vorwärtsberechnung:

- Entspricht symbolischer Ausführung
- Umständlichere Berechnung der Verifikationsbedingungen
- Erlaubt zustandsbasierte Vereinfachung (z.B. Entfernen unerreichbarer Fälle)

- Die Generierung von Verifikationsbedingungen korrespondiert zur relativen Vollständigkeit der Floyd-Hoare-Logik — nächste Woche.

Korrekte Software: Grundlagen und Methoden
Vorlesung 7 vom 12.05.16: Korrektheit der Floyd-Hoare-Logik

Serge Autexier, Christoph Lüth

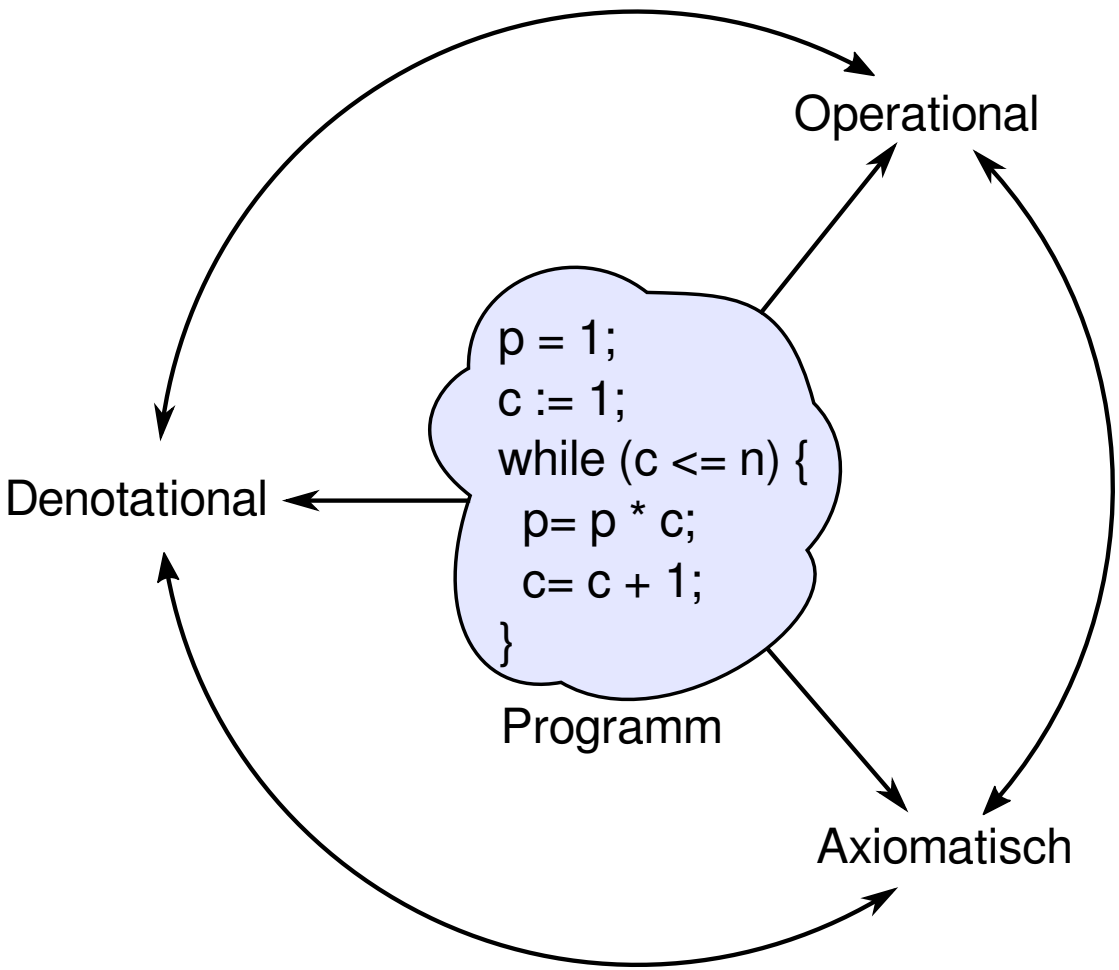
Universität Bremen

Sommersemester 2016

Fahrplan

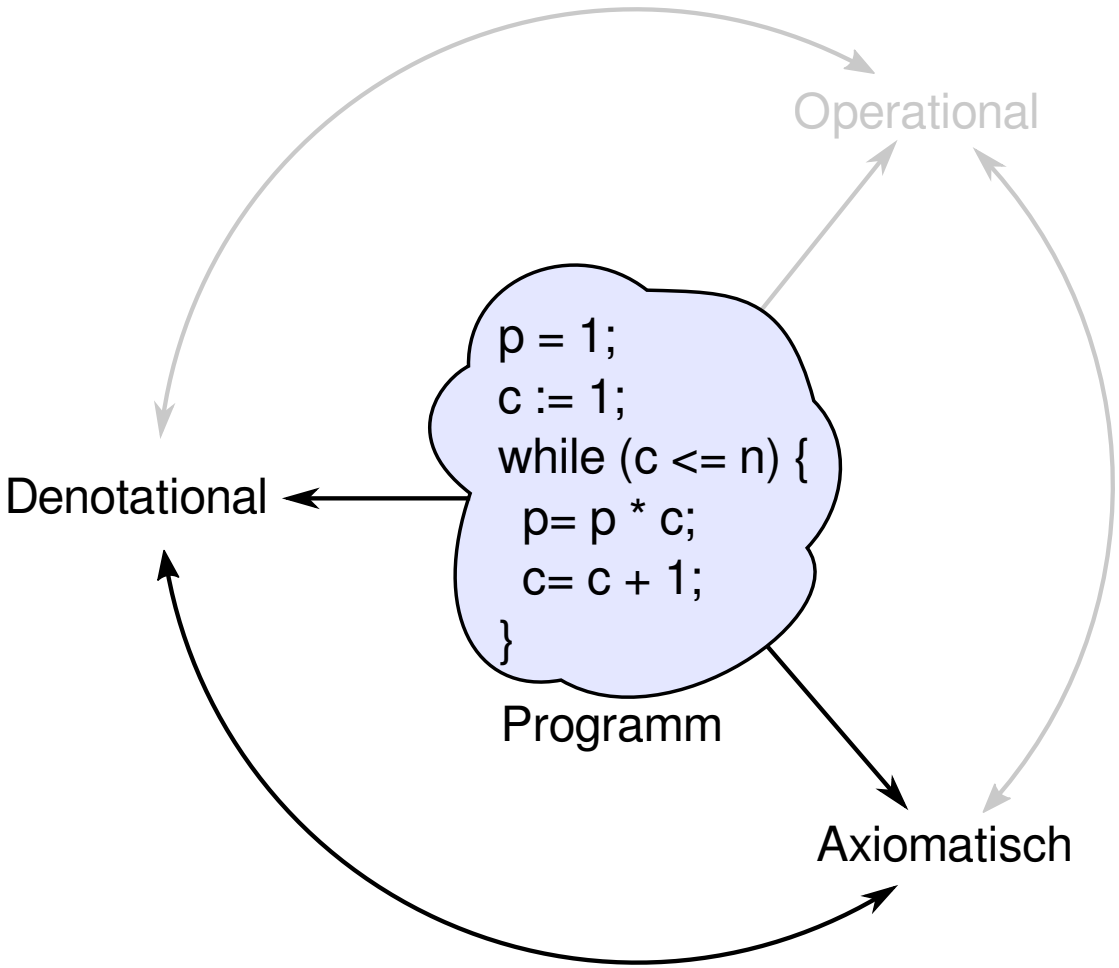
- Einführung
- Die Floyd-Hoare-Logik
- Operationale Semantik
- Denotationale Semantik
- Äquivalenz der Semantiken
- Verifikation: Vorwärts oder Rückwärts?
- Korrektheit des Hoare-Kalküls
- Einführung in Isabelle/HOL
- Weitere Datentypen: Strukturen und Felder
- Funktionen und Prozeduren
- Referenzen und Zeiger
- Frame Conditions & Modification Clauses
- Ausblick und Rückblick

Motivation



- 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.

Motivation



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

Denotionale Semantik

- Denotat eines Ausdrucks (Programms) ist partielle Funktion:

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

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

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

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

$$\mathrm{def}(f(x)) \longleftrightarrow f(x) \neq \perp$$

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)

Bezug zur Semantik?

Hoare-Tripel und denotationale Semantik

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

Gültigkeit von Hoare-Tripeln

$$\models \{P\}\, c\, \{Q\} \longleftrightarrow \forall \sigma \in \Sigma.\, \sigma \models P \wedge \mathrm{def}(\mathcal{D}[\![c]\!](\sigma)) \longrightarrow \mathcal{D}[\![c]\!]\sigma \models Q$$

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

Überblick: die Regeln des Floyd-Hoare-Kalküls

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

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

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

$$\frac{\vdash \{A \wedge [\![b]\!]\}\, c\, \{A\}}{\vdash \{A\}\ \textbf{while}(b)\ c\, \{A \wedge \neg [\![b]\!]\}}$$

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

Korrektheit und Vollständigkeit

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

Korrektheit der Floyd-Hoare-Logik

Floyd-Hoare-Logik ist korrekt.

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

Beweis:

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

Vollständigkeit der Floyd-Hoare-Logik

Floyd-Hoare-Logik ist vollständig modulo weakening.

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

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

Zusammenfassung

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

Korrekte Software: Grundlagen und Methoden
Vorlesung 8 vom 19.05.16: Einführung zu Isabelle

Serge Autexier, Christoph Lüth

Universität Bremen

Sommersemester 2016

Fahrplan

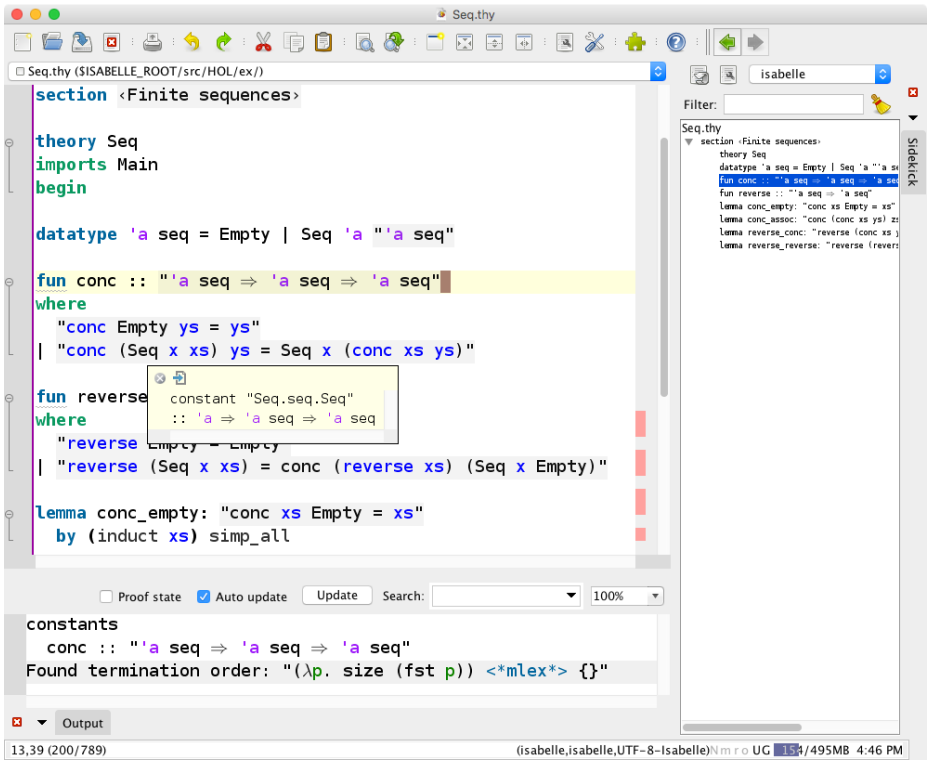
- Einführung
- Die Floyd-Hoare-Logik
- Operationale Semantik
- Denotationale Semantik
- Äquivalenz der Semantiken
- Verifikation: Vorwärts oder Rückwärts?
- Korrektheit des Hoare-Kalküls
- Einführung in Isabelle/HOL
- Weitere Datentypen: Strukturen und Felder
- Funktionen und Prozeduren
- Referenzen und Zeiger
- Frame Conditions & Modification Clauses
- Ausblick und Rückblick

Motivation

- Verwendung des interaktiven Theorembeweisers Isabelle/HOL, um anfallende Beweisverpflichtungen über C0-Software (und kommende Erweiterungen) zu beweisen.

Isabelle/HOL

- Ist ein interaktiver Theorembeweiser
- Website: `isabelle.in.tum.de`
- Basiert auf Logik HOL
- Umfangreiche Automatisierungen für Beweissuche
- High-level Syntax für Modellierung und Beweiskonstruktion
- Gute Editor-Integration (jEdit) ≈ IDE für Isabelle Theorien und Beweise
- Im Reiter “Documentation”: Prog-prove, Tutorial



HOL Formeln

- HOL is ein getypte Logik höherer Ordnung (ähnlich zu funktionalen Programmiersprachen)
  - Basistypen: nat, bool, int
  - Typkonstruktoren: list , set
  - Funktionstyp: =>
  - Typvariablen: 'a 'b 'c
- Typdeklarationen:
  - op + :: nat => nat => nat
  - <= :: nat => nat => bool
  - exp2 :: nat => nat

Terme und Formeln

- Terme:
  - Infix Notation a = b, a ~= b, a <= b, a + b usw.
  - Wenn f :: t1 => t2 und t :: t1 dann ist
    f t :: t2
  - Formeln sind Terme vom Typ bool

```
True :: bool, False :: bool
not :: bool => bool                    ~, \<not>
& :: bool => bool => bool              \<and>
| :: bool => bool => bool              \<or>
--> :: bool => bool => bool            \<longrightarrow>
= :: 'a => 'a => bool
ALL x . P                              \<forall> x . P
EX x . P                               \<exists> x . P
```

Beweiszustände

$$\bigwedge x_1 \ldots x_n.\text{assumptions} \Longrightarrow \text{conclusion}$$

$$\bigwedge x, y, z. \overbrace{[\![x \leq y; y \leq z]\!]} \Longrightarrow x \leq z$$

$$\bigwedge x, y, z. \overbrace{x \leq y \Longrightarrow y \leq z} \Longrightarrow x \leq z$$

Theorien

Dateiname: T.thy

```
theory T  (* Name muss mit Dateiname uebereinstimmen *)

imports Main (* in Main ist alles drin, was man
                so braucht / erst mal *)
begin
(* ... Definitionen, Theoreme, Beweise *)
end
```

Datentypen

```
datatype 'a list = Nil | Cons 'a "'a list"
```

- Listen von Objekten vom Typ 'a
- Nil hat als Notation auch []
- Cons x xs hat als Notation auch x#xs Erzeugt Induktionsregeln (für Beweise)

$$\frac{\text{P Nil} \qquad \bigwedge x, xs.\text{P xs} \Longrightarrow \text{P (Cons x xs)}}{\text{ALL I . P I}}$$

Konstanten

```
definition eins :: nat where "eins = Suc 0"

definition zweierliste :: "'a => 'a => 'a list" where
"zweierliste x y = x#y#[]"
```

- Erzeugt entsprechende Konstanten, aber keine Simplifikationsregeln

Funktionen

```
fun div2 :: "nat => nat" where
"div2 0 = 0" |
"div2 (Suc 0) = 0" |
"div2 (Suc (Suc n)) = Suc (div2 n)"
```

- Beweis der Terminierung automatisch (falls Fehlschlag, muss man korrigieren oder selber helfen)
- Erzeugt spezielle Induktionsregel

$$\frac{\text{P 0} \qquad \text{P (Suc 0)} \qquad \bigwedge n.\text{P n} \Longrightarrow \text{P (Suc (Suc n))}}{\text{ALL n . P n}}$$

- Name: div2.induct

Konstanten / Funktionen / Prädikate

```
fun div2 :: "nat => nat" where
"div2 0 = 0" |
"div2 (Suc 0) = 0" |
"div2 (Suc (Suc n)) = Suc (div2 n)"
```

- Beweis der Terminierung automatisch (falls Fehlschlag, muss man korrigieren oder selber helfen)
- Erzeugt spezielle Induktionsregel

$$\frac{\text{P 0} \qquad \text{P (Suc 0)} \qquad \bigwedge n.\text{P n} \Longrightarrow \text{P (Suc (Suc n))}}{\text{ALL n . P n}}$$

- Name: div2.induct

Theoreme und Beweise

```
lemma rev_app: "rev (app xs ys) = app (rev ys) (rev xs)"
```

- Beweiszustand
- Ein oder mehrere Unterziele
- Beweisskript bearbeitet immer das erste Unterziele
- Anwendung einer Taktik oder Regel mittels apply

Automatisierungen / Beweismethoden

- Es gibt keine vollständige Beweisverfahren für Higher-Order-Logik (HOL), aber Teile lassen sich automatisieren
- Simplifikation: simp
  - Wendet alle verwendbaren Simplifikationsregeln an
    - Datatypdefinitionen, Funktionsdefinitions (auch primrec), keine Konstanten Definitionen
    - Theoreme nur wenn sie mit [simp] gekennzeichnet sind.
  - Keywords:

```
add: <list-of-theorem-names>
del: <list-of-theorem-names>
only: <list-of-theorem-names>
```

- Etwas mehr Automatisierung: auto

Automatisierung

- Arithmetik: arith (eingebaut in simp, auto)

```
lemma "[| ~ (m < n); m < n + (1::nat)|] ==> m = n"

lemma "m ~= (n::nat) ==> (m < n | n < m)"
```

- Noch etwas mehr fastforce (auch Quantoren)

```
lemma "[|\<forall> xs \<in> A .
          \<exists> ys . xs = ys @ ys;
        us \<in> A |]
    ==> \<exists> n . length us = n + n"
```

- Noch etwas mehr: blast
- Sehr viel mehr: sledgehammer

Darüberhinaus...

- Fallunterscheidung: case_tac

```
apply (case_tac xs)
```

- Induktion: induct

```
apply (induct xs)
apply (induct xs :rule div2.induct)
```

- Zwischenziele einführen: subgoal_tac

lemma "[| A --> B; B --> C |] ==> A --> C"

Konstanten / Funktionen / Prädikate

```
fun forever :: "nat ⇒ nat" where
"forever 0 = 1" |
"forever (Suc n) = forever (div2 n)"
```

- Beweis der Terminierung automatisch (falls Fehlschlag, muss man korrigieren oder selber helfen)
- Erzeugt spezielle Induktionsregel

$$\frac{\text{P 0} \qquad \bigwedge n.\text{P (div2 n)} \Longrightarrow \text{P n}}{\text{ALL n . P n}}$$

- Name: div2.induct

Einzelne Regeln

- Manchmal helfen die Taktiken nicht, oder machen zu viel, und man muss einzelne Beweisschritte eingeben.
- Basisbeweisschritte sind Kalkülregeln (ähnlich wie Operationale/Axiomatische Semantik)

$$\frac{\Gamma \Longrightarrow ?P \qquad \Gamma \Longrightarrow ?Q}{\Gamma \Longrightarrow ?P \wedge ?Q}\, conjI \qquad\qquad \frac{\Gamma, ?P, ?Q \Longrightarrow G}{\Gamma, ?P \wedge ?Q \Longrightarrow G}\, conjE$$

- rule: match Conclusion und wendet Regel rückwärts an (Einführungsregeln)
- erule: match Conclusion und eine Assumption, wendet Regel an (Eliminationsregeln)
- drule: match eine Assumption, wendet Regel an und löscht verwendete Assumption
- frule: wie drule ohne das Assumption gelöscht wird.

Weitere Einführungsregeln

$$\frac{\Gamma, A \Longrightarrow B}{\Gamma \Longrightarrow A \to B}\, impI \qquad \frac{\bigwedge x.[\Gamma \Longrightarrow (?Px)]}{\Gamma \Longrightarrow \forall x.(?Px)}\, allI$$

Regeln für Gleichheit

$$\frac{\Gamma; s = t \Longrightarrow (Ps)}{\Gamma; s = t \Longrightarrow (Pt)}\,subst \qquad \frac{\Gamma; s = t \Longrightarrow (Pt)}{\Gamma; s = t \Longrightarrow (Ps)}\,subst$$

- subst, ssubst

- Parameter vorgeben: apply (rule ssubst [where t="(f x)" and s="x"])

Theoreme finden

- Theoreme sind in Lemmata oder Definitionen in importierten Theorien von Main
- Im Reiter “query” im Eingabefeld “find” kann nach Theorem gesucht werden
- Verwende Patterns um nach Struktur zu suchen (Wildcard _)
  - "_ + x = x''
- Weitere Beispiele im Tutorial auf S.34

Korrekte Software: Grundlagen und Methoden
Vorlesung 9 vom 23.05.16: Weitere Datentypen: Strukturen und Felder

Serge Autexier, Christoph Lüth

Universität Bremen

Sommersemester 2016

Fahrplan

- Einführung
- Die Floyd-Hoare-Logik
- Operationale Semantik
- Denotationale Semantik
- Äquivalenz der Semantiken
- Verifikation: Vorwärts oder Rückwärts?
- Korrektheit des Hoare-Kalküls
- Einführung in Isabelle/HOL
- Weitere Datentypen: Strukturen und Felder
- Funktionen und Prozeduren
- Referenzen und Zeiger
- Frame Conditions & Modification Clauses
- Ausblick und Rückblick

Motivation

- Weitere Basisdatentypen von C (arrays, strings und structs)
- Noch rein funktional, keine 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;
```

Strings

```
char hallo[5] = {'h', 'a', 'l', 'l', 'o', \0 }

char hallo[] = "hallo";

printf(hallo[4]);   /* liefert 'o' */
```

Struct

```
struct Vorlesung {
  char dozenten[2][30];
  char titel[30];
  int cp;
} ksgm;

struct Vorlesung ksgm;

int i = 0;
char name1[] = "Serge Autexier";
while (i < strlen(name1)) {
  ksgm.dozenten[0][i] = name1[i];
  i = i + 1;
}
char name2[] = "Christoph Lueth";
i = 0;
while (i < strlen(name2)) {
  ksgm.dozenten[1][i] = name2[i];
  i = i + 1;
}
```

Rekursive Struct

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

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

int i = 9;
while (i>0) {
 struct Liste next;
 next.kopf = i;
 next.rest = start;
 i = i - 1;
 start = next;
}
```

Ausdrücke

Location Expressions $\mathbf{Lexp} ::= \mathbf{Loc} \mid \mathbf{Lexp}\ [a] \mid \mathbf{Lexp}\ .\ \mathbf{name}$

Aexp $a ::= \mathbf{N} \mid \mathbf{Lexp} \mid a_1 + a_2 \mid a_1 - a_2 \mid a_1 * a_2 \mid a_1 / a_2 \mid \mathit{strlen}(\mathit{Exp})$

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

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

ExpList $el := e\ (, el)?$

Statements

Type type ::= *int* | *char* | *struct* **name** $\{$*puredecl*$*\}$

Decl *decl* ::= *puredecl*
| *type* **Loc**[] = $\{$*el*$\}$;

puredecl ::= *type* **Loc**;
| *type* **Loc**[**N**];

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

Werte und Zustände

Container **Cont** ::= **Loc** | **Cont** [**N**] |**Cont** . **name**

Werte sind die kleinste Menge **V** für die gilt

- **N**, **B**, **C** sind Teilmengen von **V** **(V_B)**

Zustände sind partielle Funktionen $\sigma : \mathbf{Cont} \rightharpoonup \mathbf{V}$ so dass gilt

- $\forall c, c' \in Dom(\sigma).c$ ist kein Präfix von c' und umgekehrt.
- if $c[i]c' \in Dom(\sigma)$ then $\forall 0 \leq j \leq i.\exists c_j.c[j]c_j \in Dom(\sigma)$

Zustandprojektion Sei $u \in \mathbf{Cont}$ und σ ein Zustand: Wir definieren die Projektion von σ auf u durch

$$\sigma_{|u} := \{(v, n) | (uv, n) \in \sigma\}$$

Beispiel

Programm

```
struct A {
 int c[2];
 struct B {
  char name[20];
 } b;
};

struct A x[] = {
 {{1,2},
  {{'n','a','m','e','1','\0'}}
 },
 {{3,4},
  {{'n','a','m','e','2','\0'}}
 }
};
```

Zustand

$$
\begin{aligned}
x.[0].c[0] &\to 1 \\
x.[0].c[1] &\to 2 \\
x.[0].b.name[0] &\to \text{'}n\text{'} \\
x.[0].b.name[1] &\to \text{'}a\text{'} \\
x.[0].b.name[2] &\to \text{'}m\text{'} \\
x.[0].b.name[3] &\to \text{'}e\text{'} \\
x.[0].b.name[4] &\to \text{'}1\text{'} \\
x.[0].b.name[5] &\to \text{'}\backslash 0\text{'}
\end{aligned}
\qquad
\begin{aligned}
x.[1].c[0] &\to 3 \\
x.[1].c[1] &\to 4 \\
x.[1].b.name[0] &\to \text{'}n\text{'} \\
x.[1].b.name[1] &\to \text{'}a\text{'} \\
x.[1].b.name[2] &\to \text{'}m\text{'} \\
x.[1].b.name[3] &\to \text{'}e\text{'} \\
x.[1].b.name[4] &\to \text{'}2\text{'} \\
x.[1].b.name[5] &\to \text{'}\backslash 0\text{'}
\end{aligned}
$$

Auswertung von Lexp zu Cont

$$\frac{x \in \mathbf{Loc}}{\langle x, \sigma \rangle \to_{Lexp} x}$$

$$\frac{\langle lexp, \sigma \rangle \to_{Lexp} c \qquad \langle a, \sigma \rangle \to_{Aexp} i}{\langle lexp[a], \sigma \rangle \to_{Lexp} c[i]}$$

$$\frac{\langle lexp, \sigma \rangle \to_{Lexp} c}{\langle lexp.name, \sigma \rangle \to_{Lexp} c.name}$$

Aexp: Operationale Semantik

$$\frac{\langle lexp, \sigma \rangle \rightarrow_{Lexp} c \qquad c \in Dom(\sigma)}{\langle lexp, \sigma \rangle \rightarrow_{Aexp} \sigma(c)}$$

$$\frac{\langle lexp, \sigma \rangle \rightarrow_{Lexp} c \qquad c \notin Dom(\sigma)}{\langle lexp, \sigma \rangle \rightarrow_{Aexp} \bot}$$

$$\frac{\begin{array}{c}\langle str, \sigma \rangle \rightarrow_{Lexp} s :: char[n], \\ l = min(\{n+1\} \cup \{m | m < n, s[m] =' \backslash 0', s[0..m-1] \neq' \backslash 0'\}\end{array}}{\langle strlen(str), \sigma \rangle \rightarrow_{Aexp} l}$$

Operationale Semantic: Zuweisungen

$$\frac{\langle lexp, \sigma\rangle \rightarrow_{Lexp} c \qquad \sigma(c) :: \tau \qquad \langle exp, \sigma\rangle \rightarrow e :: \tau}{\langle lexp = exp, \sigma\rangle \rightarrow_{Stmt} \sigma[e/c]}$$

$$\begin{array}{lll} \text{Stmt } c & ::= & \text{decl} \\ & | & \textbf{Lexp} = \textbf{Exp}; \\ & | & \textbf{if } (\ b\ )\ c_1\ \textbf{ else }\ c_2 \\ & | & \textbf{while } (\ b\ )\ c \\ & | & \{c^*\} \end{array}$$

Denotationale Semantik

- Denotation für **Lexp**

$$\begin{aligned}\mathcal{L}[\![x]\!] &= \{(\sigma, x) | \sigma \in \Sigma\} \\ \mathcal{L}[\![lexp[a]]\!] &= \{(\sigma, l[i]) | (\sigma, l) \in \mathcal{L}[\![lexp]\!], (\sigma, i) \in \mathcal{E}[\![a]\!]\} \\ \mathcal{L}[\![lexp.name]\!] &= \{(\sigma, l.name) | (\sigma, l) \in \mathcal{L}[\![lexp]\!]\}\end{aligned}$$

- Denotation für **Zuweisungen**

$$\mathcal{D}[\![lexp = exp]\!] = \{(\sigma, \sigma[e/c]) | (\sigma, c) \in \mathcal{L}[\![lexp]\!], (\sigma, e) \in \mathcal{E}[\![exp]\!]\}$$

Hoare-Regel

- Vor- Nachbedingungen von Hoare-Regeln müssen auch Gleichungen über Container Werte haben

- Nicht unbedingt alle, aber alle die gebraucht werden

Beispiel

```
int a[3];
/** { 1 } */
/** { 3 = 3 and 3 = 3 } */
a[2] = 3;
/** { a[2] = 3 and a[2] = 3 } */
/** { 4 = 4 and a[2] = 3 and 4 * a[2] = 12 } */
a[1] = 4;
/** { a[1] = 4 and a[2] = 3 and a[1] * a[2] = 12 } */
/** { 5 = 5 and a[1] = 4 and a[2] = 3 and
      5 * a[1] * a[2] = 60 } */
a[0] = 5;
/** { a[0] = 5 and a[1] = 4 and a[2] = 3 and
      a[0] * a[1] * a[2] = 60 } */
```

Beispiel

```
int a[3];
/** { true } */
/** { 2 = 2 and 3 = 3 and 3 = 3 } */
int i = 2;
/** { i = 2 and 3 = 3 and 3 = 3 } */
a[i] = 3;
/** { i = 2 and a[i] = 3 and a[i] = 3 } */
/** { 1 = 1 and 4 = 4 and a[2] = 3 and 4 * a[2] = 12 } */
i = 1;
/** { i = 1 and 4 = 4 and a[2] = 3 and 4 * a[2] = 12 } */
a[i] = 4;
/** { i = 1 and a[i] = 4 and a[2] = 3 and
      a[i] * a[2] = 12 } */
/** { 0 = 0 and a[1] = 4 and a[2] = 3 and
      a[1] * a[2] = 12 } */
i = 0;
/** { i = 0 and a[1] = 4 and a[2] = 3 and
      a[1] * a[2] = 12 } */
/** { i = 0 and 5 = 5 and a[1] = 4 and a[2] = 3 and
      5 * a[1] * a[2] = 60 } */
a[i] = 5;
/** { i = 0 and a[i] = 5 and a[1] = 4 and a[2] = 3 and
      a[i] * a[1] * a[2] = 60 } */
/** { i = 0 and a[i] = 5 and a[1] = 4 and a[2] = 3 and
      a[0] * a[1] * a[2] = 60 } */
```

Korrekte Software: Grundlagen und Methoden
Vorlesung 10 vom 30.05.16: Funktionen und Prozeduren

Serge Autexier, Christoph Lüth

Universität Bremen

Sommersemester 2016

Fahrplan

- Einführung
- Die Floyd-Hoare-Logik
- Operationale Semantik
- Denotationale Semantik
- Äquivalenz der Semantiken
- Verifikation: Vorwärts oder Rückwärts?
- Korrektheit des Hoare-Kalküls
- Einführung in Isabelle/HOL
- Weitere Datentypen: Strukturen und Felder
- Funktionen und Prozeduren
- Referenzen und Zeiger
- Frame Conditions & Modification Clauses
- 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)
- Wir brauchen:
  1. Von Anweisungen zu Funktionen: Deklarationen und Parameter
  2. Semantik von Funktionsdefinition und Funktionsaufruf
  3. Spezifikation von Funktionen
  4. Beweisregeln für Funktionsdefinition und Funktionsaufruf

Von Anweisungen zu Funktionen

- Erweiterung unserer Kernsprache:

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

- **Type**, **Init** (Initialisierer) s. letzte Vorlesung
- **FunSpec** später
- Abstrakte Syntax (vereinfacht, konkrete Syntax mischt **Type** und *Id*)

Rückgabewerte

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

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

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

Rule 14.7 (required)

A function shall have a single point of exit at the end of the function.

  - Nicht immer möglich, unübersichtlicher Code...

- Lösung 2: Erweiterung der Semantik von $\Sigma \rightharpoonup \Sigma$ zu $\Sigma \rightharpoonup (\Sigma + \Sigma \times \mathbf{V})$

Erweiterte Semantik

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

Erweiterte Semantik

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

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

Semantik von Anweisungen

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

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

$$\mathcal{D}[\![\textbf{while}\ (b)\ c]\!] = \mathit{fix}(\Gamma)$$

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

Semantik von Funktionsdefinitionen

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

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

$$\mathcal{D}_{fd}[\![f(t_1\ p_1, t_2\ p_2, \ldots, t_n\ p_n)\ blk]\!] =$$
$$\lambda v_1, \ldots, v_n. \{(\sigma, (\sigma', v)) \mid$$
$$(\sigma, (\sigma', v)) \in \mathcal{D}_{blk}[\![blk]\!] \circ_S \{(\sigma, \sigma[p_1 \mapsto v_1, \ldots, p_n \mapsto v_n])\}\}$$

- 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

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

Blöcke bestehen aus Deklarationen und einer Anweisung:

$$\mathcal{D}_{blk}[\![decls\ stmts]\!] = \mathcal{D}[\![stmts]\!] \circ_S \mathcal{D}_d[\![decls]\!]$$
$$\mathcal{D}_d[\![t\ i]\!] = \{(\sigma, \sigma[i \mapsto \bot])\}$$
$$\mathcal{D}_d[\![t\ i\ = init]\!] = \{(\sigma, \sigma[i \mapsto \mathcal{E}_{init}[\![init]\!]\}$$

- Verallgemeinerung auf Sequenz von Deklarationen
- $\mathcal{E}_{init}[\![.]\!]$ ist das Denotat von Initialisierungen

Funktionsaufrufe

- Aufruf einer Funktion: $f(t_1, \ldots, t_n)$:
  - Auswertung der Argumente $t_1, \ldots, t_n$
  - Einsetzen in die Semantik $\mathcal{D}_{fd}[\![f]\!]$
- Was ist mit Seiteneffekten?
  - Erst mal gar nichts...
- Call by name, call by value, call by reference...?
  - C kennt nur call by value (C-Standard 99, §6.9.1. (10))
  - Arrays werden als Referenzen übergeben (deshalb betrachten wir heute keine Arrays als Funktionsparameter).

Funktionsaufrufe

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

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

- Das Environment ist zusätzlicher Parameter für alle Definitionen
- Damit:

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

  - Aufruf einer nicht-definierten Funktion f oder mit falschen Anzahl n von Parametern ist nicht definiert
  - Wird durch statische Analyse verhindert

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:

$$\textbf{FunSpec} ::= /** \ \textbf{pre}\ \textbf{Bexp}\ \textbf{post}\ \textbf{Bexp}\ */$$

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

Semantik von Spezifikationen

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

$$\mathcal{B}_{sp}[\![.]\!] : Env \rightharpoonup (\Sigma \times (\Sigma \times \mathbf{V}_U)) \to \mathbf{T}$$

$$\mathcal{E}_{sp}[\![.]\!] : Env \rightharpoonup (\Sigma \times (\Sigma \times \mathbf{V}_U)) \to \mathbf{V}$$

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

$$\ldots$$

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

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

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

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

Gültigkeit von Spezifikationen

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

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

  - Γ enthält globale Definitionen, insbesondere andere Funktionen.
  - Vgl. $\models \{P\}\, c\, \{Q\}$ für Hoare-Tripel

- Aber wie beweisen wir das? $\longrightarrow$ Nächste Vorlesung
  - Die Grenzen des Hoare-Kalküls sind erreicht.

Zusammenfassung

- Funktionen sind zentrales Modularisierungskonzept
- Wir müssen Funktionen modular verifizieren können
- Semantik von Deklarationen und Parameter — straightforward
- Semantik von Rückgabewerten — Erweiterung der Semantik
- Funktionsaufrufe — Environment, um Funktionsbezeichnern eine Semantik zu geben
  - C kennt nur call by value
- Spezifikation von Funktionen: Vor-/Nachzustand statt logischer Variablen

Korrekte Software: Grundlagen und Methoden
Vorlesung 11 vom 06.06.16: Funktionen und Prozeduren

Serge Autexier, Christoph Lüth

Universität Bremen

Sommersemester 2016

Fahrplan

- Einführung
- Die Floyd-Hoare-Logik
- Operationale Semantik
- Denotationale Semantik
- Äquivalenz der Semantiken
- Verifikation: Vorwärts oder Rückwärts?
- Korrektheit des Hoare-Kalküls
- Einführung in Isabelle/HOL
- Weitere Datentypen: Strukturen und Felder
- Funktionen und Prozeduren
- Referenzen und Zeiger
- Frame Conditions & Modification Clauses
- 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)
- 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

Motivation

- Funktionen sind zentrales Modularisierungskonzept
- Wir müssen Funktionen modular verifizieren können
- Semantik von Deklarationen und Parameter — straightforward
- Semantik von Rückgabewerten — Erweiterung der Semantik
- Funktionsaufrufe — Environment, um Funktionsbezeichnern eine Semantik zu geben
  - C kennt nur call by value
- Spezifikation von Funktionen: Vor-/Nachzustand statt logischer Variablen

Hoare-Kalkül für Funktionspezifikationen

$$\begin{aligned}
\textbf{FunDef} &::= \textbf{Type}\ \mathit{Id}(\textbf{Param}^*)\ \textbf{FunSpec}^+\ \textbf{Blk} \\
\textbf{Param} &::= \textbf{Type}\ \mathit{Id} \\
\textbf{FunSpec} &::= /{*}{*}\ \textbf{pre}\ \textbf{Bexpr}\ \textbf{post}\ \textbf{Bexpr}\ {*}/ \\
\textbf{Blk} &::= \{\textbf{Decl}^*\ \textbf{Stmt}\} \\
\textbf{Decl} &::= \textbf{Type}\ \mathit{Id}\ = \textbf{Init} \mid \textbf{Type}\ \mathit{Id}
\end{aligned}$$

- Hoare-Tripel:

$$\{P\}\ c\ \{Q_L \mid Q_G\}$$

Beispiel

```
int factorial(int x)
/** pre x >= 0
    post \result = \old(x)! */ {
 int r = 0;
 if (x = 0) {
    return 1;
 }
 else {
    r = factorial(x - 1);
 }
 return r * x;
}
```

$$\frac{}{\tau_0\ f(\tau_1\ v_1, \ldots, \tau_n\ v_n)\ \texttt{/**}\ \text{pre}\ P\ \text{post}\ Q\ \texttt{*/}\ c}$$

Beispiel

```
int factorial(int x)
/** pre x >= 0
    post \result = \old(x)! */ {
 int r = 0;
 if (x = 0) {
    return 1;
 }
 else {
    r = factorial(x - 1);
 }
 return r * x;
}
```

$$\frac{\{P'\}c\{Q|Q\}}{\tau_0\ f(\tau_1\ v_1, \ldots, \tau_n\ v_n)\ /^{**}\ \text{pre}\ P\ \text{post}\ Q\ ^*/\ c}$$

Beispiel

```
int factorial(int x)
/** pre x >= 0
    post \result = \old(x)! */ {
 int r = 0;
 if (x = 0) {
    return 1;
 }
 else {
    r = factorial(x - 1);
 }
 return r * x;
}
```

$$\frac{P \longrightarrow P'_{\backslash old(\$) \to \$} \qquad \{P'\} c \{Q | Q\}}{\tau_0\ f(\tau_1\ v_1, \ldots, \tau_n\ v_n)\ \texttt{/**}\ \text{pre}\ P\ \text{post}\ Q\ \texttt{*/}\ c}$$

Beispiel

```
int factorial(int x)
/** pre x >= 0
    post \result = \old(x)! */ {
 int r = 0;
 if (x = 0) {
    return 1;
 }
 else {
    r = factorial(x - 1);
 }
 return r * x;
}
```

$$\frac{x >= 0 \longrightarrow P'_{\backslash old(\$) \to \$} \quad \{P'\}c\{\backslash result = \backslash old(x)! \mid \backslash result = \backslash old(x)!\}}{int\ factorial(int\ x)\ /^{**}\ \text{pre}\ x >= 0\ \text{post}\ \backslash result = \backslash old(x)!\ */\ c}$$

Hoare-Kalkül mit return

$$\frac{}{\{P\}\{\}\{P|Q\}}$$

$$\frac{\{P\}c\{Q_1'|Q_2\} \qquad \{Q_1'\}cs\{Q_1|Q_2\}}{\{P\}c\ cs\{Q_1|Q_2\}}$$

$$\frac{}{\{Q_1[e/l]\}l = e\{Q_1|Q_2\}}$$

$$\frac{\{P \wedge b\}c_1\{Q_1|Q_2\} \qquad \{P \wedge \neg b\}c_2\{Q_1|Q_2\}}{\{P\}\text{if } b\ c_1 \text{ else } c_2\{Q_1|Q_2\}}$$

While, Weakening

$$\frac{\{P \wedge b\} c \{P | Q\}}{\{P\}\text{while } (b)\ c\{P \wedge \neg(b) | Q\}}$$

$$\frac{P \longrightarrow P' \qquad \{P'\} c \{Q_1' | Q_2'\} \qquad Q_1' \longrightarrow Q_1 \qquad Q_2' \longrightarrow Q_2}{\{P\} c \{Q_1 | Q_2\}}$$

Hoare-Kalkül mit return

$$\frac{}{\{Q[^{e}/_{\backslash result}]\}\, return\ e\, \{P|Q\}}$$

$$\frac{Q \text{ enthält kein } \backslash result}{\{Q\}\, return \{P|Q\}}$$

Funktionsaufruf

$$\frac{\begin{array}{c}(V_1 = e_1 \wedge \ldots V_n = e_n \wedge Q_1) \longrightarrow f.pre(e_1, \ldots e_n) \\ P := f.post(e_1, \ldots, e_n)_{\backslash result \to I, \backslash old(v_i) \to V_i}\end{array}}{\begin{array}{c}\{(V_1 = e_1 \wedge \ldots V_n = e_n \wedge Q_1 \wedge P)[f(e_1, \ldots, e_n)/I]\} \\ I = f(e_1, \ldots, e_n) \\ \{V_1 = e_1 \wedge \ldots V_n = e_n \wedge Q_1 \wedge P | Q_2\}\end{array}}$$

f mit formalen Parametern $v_1, \ldots, v_n$; $V_1, \ldots V_n$ logische Variablen

```
int factorial(int x)
/** pre x >= 0
    post \result = \old(x)! */ {
/** { COND: x >= 0 ---> x >= 0 ∧ x! = x!
      x >= 0 ∧ x! = \old(x)! } */
 int r = 0;
/** {x >= 0 ∧ x! = \old(x)!} */
 if (x = 0) {
/** {x >= 0 ∧ x = 0 ∧ x! = \old(x)!} */
/** {x >= 0 ∧ x = 0 ∧ 1 = \old(x)!} */
    return 1;
/** { r * x = \old(i)! |  \result = \old(x)!} */
 }
 else { ... }
/** { r * x = \old(i)! | \result =\old(i)! } */
 return r * x;
/** {\result =\old(i)! | \result =\old(i)!} */
```

```
 else {
/** {x >= 0 ∧ ¬(x = 0) ∧ x! = \old(x)!} */
/** {COND: x >= 0 ∧ ¬(x = 0) ∧
            factorial(x−1) * x = \old(x)! ∧ V = x −−> x − 1 >= 0
     x >= 0 ∧ ¬(x = 0) ∧ factorial(x−1) * x = \old(x)! ∧
     V = x ∧ factorial(x−1) = (V−1)!} */
    r = factorial(x − 1);
/** {x >= 0 ∧ ¬(x = 0) ∧ r * x = \old(x)! ∧
     V = x ∧ r = (V−1)!} */
/** { x >= 0 ∧ ¬(x = 0) ∧ r * x = \old(x)! } */
/** { r * x = \old(i)! } */
 }
/** { r * x = \old(i)! | \result =\old(i)! } */
 return r * x;
/** {\result = \old(i)! | \result =\old(i)!} */
}
```

Approximative schwächste Vorbedingung (Revisited)

$$
\begin{aligned}
\mathrm{awp}(\Gamma, \{\,\}, P) &\stackrel{def}{=} P \\
\mathrm{awp}(\Gamma, l = f(e_1, \ldots, e_n), P) &\stackrel{def}{=} P[F([\![e_1]\!], \ldots, [\![e_n]\!])/[\![l]\!]] \\
\text{mit } post(\Gamma!f) &= (\forall v_1, \ldots, v_n.\,\textbf{result} = F(v_1, \ldots, v_n)) \\
\mathrm{awp}(\Gamma, l = e, P) &\stackrel{def}{=} P[[\![e]\!]/[\![l]\!]] \\
\mathrm{awp}(\Gamma, \{c\ c_s\}, P) &\stackrel{def}{=} \mathrm{awp}(\Gamma, c, \mathrm{awp}(\{c_s\}, P)) \\
\mathrm{awp}(\Gamma, \textbf{if}\ (b)\ \{c_0\}\ \textbf{else}\ \{c_1\}, P) &\stackrel{def}{=} (b \wedge \mathrm{awp}(\Gamma, c_0, P)) \\
&\quad \vee (\neg b \wedge \mathrm{awp}(\Gamma, c_1, P)) \\
\mathrm{awp}(\Gamma, /{**}\ \{q\}\ {*}/, P) &\stackrel{def}{=} [\![q]\!] \\
\mathrm{awp}(\Gamma, \textbf{while}\ (b)\ /{**}\ \textbf{inv}\ i\ {*}/\ c, P) &\stackrel{def}{=} [\![i]\!] \\
\mathrm{awp}(\Gamma, \textbf{return}\ e, P) &\stackrel{def}{=} post(\Gamma)[[\![e]\!]/\textbf{result}] \\
\mathrm{awp}(\Gamma, \textbf{return}, P) &\stackrel{def}{=} post(\Gamma)
\end{aligned}
$$

Approximative schwächste Vorbedingung (Revisited)

$$
\begin{aligned}
\mathsf{wvc}(\Gamma, \{\,\}, P) &\stackrel{def}{=} \emptyset \\
\mathsf{wvc}(\Gamma, x = e, P) &\stackrel{def}{=} \emptyset \\
\mathsf{wvc}(\Gamma, x = f(e_1, \ldots, e_n), P) &\stackrel{def}{=} P \longrightarrow \mathit{pre}(\Gamma!f)([\![e_1]\!], \ldots, [\![e_n]\!]) \\
\mathsf{wvc}(\Gamma, \{c\ c_s\}, P) &\stackrel{def}{=} \mathsf{wvc}(\Gamma, c, \mathsf{awp}(\{c_s\}, P)) \\
&\quad \cup \mathsf{wvc}(\Gamma, \{c_s\}, , P) \\
\mathsf{wvc}(\Gamma, \textbf{if}\ (b)\ c_0\ \textbf{else}\ c_1, P) &\stackrel{def}{=} \mathsf{wvc}(\Gamma, c_0, P) \cup \mathsf{wvc}(\Gamma, c_1, P) \\
\mathsf{wvc}(\Gamma, /{**}\ \{q\}\ {*}/, P) &\stackrel{def}{=} \{[\![q]\!] \longrightarrow P\} \\
\mathsf{wvc}(\Gamma, \textbf{while}\ (b)\ /{**}\ \textbf{inv}\ i\ {*}/\ c, P) &\stackrel{def}{=} \mathsf{wvc}(\Gamma, c, [\![i]\!]) \\
&\quad \cup \{[\![i]\!] \wedge b \longrightarrow \mathsf{awp}(\Gamma, c, [\![i]\!])\} \\
&\quad \cup \{[\![i]\!] \wedge \neg b \longrightarrow P\} \\
\mathsf{wvc}(\Gamma, \textbf{return}\ e, P) &\stackrel{def}{=} \emptyset
\end{aligned}
$$

Korrekte Software: Grundlagen und Methoden
Vorlesung 12 vom 09.06.16: Referenzen und Speichermodelle

Serge Autexier, Christoph Lüth

Universität Bremen

Sommersemester 2016

Organisatorisches

Die Vorlesung am Montag, 13.06.2016 fällt wegen der 10-Jahres-Feier des DFKI Bremen aus.

Besucht unseren Tag der offenen Tür am Dienstag, 14.06.2016 (Robert-Hooke-Straße 1, hinter dem Fallturm).

Fahrplan

- Einführung
- Die Floyd-Hoare-Logik
- Operationale Semantik
- Denotationale Semantik
- Äquivalenz der Semantiken
- Verifikation: Vorwärts oder Rückwärts?
- Korrektheit des Hoare-Kalküls
- Einführung in Isabelle/HOL
- Weitere Datentypen: Strukturen und Felder
- Funktionen und Prozeduren
- Referenzen und Zeiger
- Frame Conditions & Modification Clauses
- Ausblick und Rückblick

Motivation

- Bisher: Zustand ist **Loc** $\rightharpoonup$ **Val**
  - **Loc** — symbolische Zustände (*Locations*)
  - **Val** — Basisdatentypen
- Grenzen: keine Referenzen
  - Damit auch kein *call by reference*
  - Funktion können nur globale Seiteneffekte haben
  - Was wäre C ohne Pointer?

Refenzen in C

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

Erweiterung des Zustandmodells

- Erweiterung von Zustand und Werten:

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

- 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:

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

$$\begin{aligned}
\mathit{read}(\mathit{upd}(\sigma, l, v), l) &= v \\
l \neq m \longrightarrow \mathit{read}(\mathit{upd}(\sigma, l, v), m) &= \mathit{read}(\sigma, m) \\
\mathit{upd}(\mathit{upd}(\sigma, l, v), l, w) &= \mathit{upd}(\sigma, l, w) \\
l \neq m \longrightarrow \mathit{upd}(\mathit{upd}(\sigma, l, v), m, w) &= \mathit{upd}(\mathit{upd}(\sigma, m, w), l, v)
\end{aligned}$$

- Diese Gleichungen sind vollständig.

Axiomatisches Speichermodell

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

$$\begin{aligned} \mathit{empty} &: \Sigma \\ \mathit{fresh} &: \Sigma \to \mathbf{Loc} \\ \mathit{rem} &: \Sigma \to \mathbf{Loc} \to \Sigma \end{aligned}$$

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

$$\begin{aligned} \mathit{dom}(\sigma) &= \{l \mid \exists v.\, \mathit{read}(\sigma, l) = v\} \\ \mathit{dom}(\mathit{empty}) &= \emptyset \end{aligned}$$

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

$$\begin{aligned} \mathit{fresh}(\sigma) &\notin \mathit{dom}(\sigma) \\ \mathit{dom}(\mathit{rem}(\sigma, l)) &= \mathit{dom}(\sigma) \setminus \{l\} \\ l \neq m \longrightarrow \mathit{read}(\mathit{rem}(\sigma, l), m) &= \mathit{read}(\sigma, m) \end{aligned}$$

Zeigerarithmetik

- Erklärt noch keine Zeigerarithmetik — dazu:

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

  - Wir betrachten keine Differenz von Zeigern

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

Erweiterung der Semantik

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

Erweiterung der Semantik: Lexp

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

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

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

Erweiterung der Semantik: Aexp(1)

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

$$\begin{aligned}
\mathcal{E}[\![n]\!]\ \Gamma &= \{(\sigma, n) \mid \sigma \in \Sigma\} \quad \text{für } n \in \mathbf{N} \\
\mathcal{E}[\![e]\!]\ \Gamma &= \{(\sigma, \mathit{read}(\sigma, l)) \mid (\sigma, l) \in \mathcal{L}[\![e]\!]\ \Gamma\} \\
&\qquad e \equiv x \mid \mathit{lexp}[a] \mid \mathit{lexp}.n \mid *e,\ \mathit{type}(\Gamma, e) \text{ kein Array-Typ} \\
\mathcal{E}[\![e]\!]\ \Gamma &= \{(\sigma, l) \mid (\sigma, l) \in \mathcal{L}[\![e]\!]\ \Gamma\} \\
&\qquad e \equiv x \mid \mathit{lexp}[a] \mid \mathit{lexp}.n \mid *e,\ \mathit{type}(\Gamma, e) \text{ Array-Typ} \\
\mathcal{E}[\![\&e]\!]\ \Gamma &= \{(\sigma, l) \mid (\sigma, l) \in \mathcal{L}[\![e]\!]\ \Gamma\} \\
\mathcal{E}[\![p + e]\!]\ \Gamma &= \{(\sigma, \mathit{add}(l, n \cdot \mathit{sizeof}(\tau))) \mid (\sigma, l) \in \mathcal{L}[\![p]\!]\ \Gamma \wedge (\sigma, n) \in \mathcal{E}[\![e]\!]\ \Gamma\} \\
&\qquad \mathit{type}(\Gamma, p) = *\tau,\ \mathit{type}(\Gamma, a_1) \text{ Integer-Typ} \\
\mathcal{E}[\![e + p]\!]\ \Gamma &= \{(\sigma, \mathit{add}(l, n \cdot \mathit{sizeof}(\tau))) \mid (\sigma, n) \in \mathcal{E}[\![e]\!]\ \Gamma \wedge (\sigma, l) \in \mathcal{L}[\![p]\!]\ \Gamma\} \\
&\qquad \mathit{type}(\Gamma, e) \text{ Integer-Typ und } \mathit{type}(\Gamma, p) = *\tau
\end{aligned}$$

Erweiterung der Semantik: Aexp(2)

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

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

für a_0, a_1 arithmetische Typen

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

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

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

Übersicht: Typen in C

int, **char**	Integer-Typ	arithmet. Typen	
float, **double**	Fließkomma-Typ		skalare Typen
* t	Pointer-Typ		
t[i]	Array-Typ		
struct t {...}	Struktur-Typen		
struct t, t[]	unvollständige Typen		

Hoare-Triple

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

- $P, Q, R : \Sigma \to \mathit{Bool}$ explizite Zustandsprädikate
- Übersetzung $[\![.]\!]$ von logischen Formeln in Zustandsprädikate
- Beispiel:

$$[\![x > 0]\!]\ \Gamma = \lambda\sigma.\, \mathit{read}(\sigma, \Gamma!x) > 0$$

- Für kürzere Regeln: "Lifting" von Boolschen Operationen:

$$P \wedge Q \stackrel{def}{=} \lambda\sigma.\, P(\sigma) \wedge Q(\sigma)$$

$$\neg P \stackrel{def}{=} \lambda\sigma.\, \neg P(\sigma)$$

$$P \longrightarrow Q \stackrel{def}{=} \lambda\sigma.\, P(\sigma) \longrightarrow Q(\sigma)$$

Regeln des Hoare-Kalküls

$$\frac{}{\Gamma \vdash \{\lambda\sigma.Q(upd(\sigma, [\![l]\!]\Gamma, [\![e]\!]\Gamma))\}\, l = e\, \{Q|R\}}$$

$$\frac{}{\Gamma \vdash \{P\}\,\{\}\,\{P|R\}} \qquad \frac{\Gamma \vdash \{P\}\, c\, \{Q_1|R\} \qquad \Gamma \vdash \{Q_1\}\,\{c_s\}\,\{Q_2|R\}}{\Gamma \vdash \{P\}\,\{c\ c_s\}\,\{Q_2|R\}}$$

$$\frac{\Gamma \vdash \{\ P \wedge [\![b]\!]\Gamma\ \}\, c_0\, \{Q|R\} \qquad \Gamma \vdash \{\ P \wedge \neg[\![b]\!]\Gamma\ \}\, c_1\, \{Q|R\}}{\Gamma \vdash \{P\}\ \textbf{if}\ (b)\ c_0\ \textbf{else}\ c_1\, \{Q|R\}}$$

$$\frac{\Gamma \vdash \{\ P \wedge [\![b]\!]\Gamma\ \}\, c\, \{Q|R\}}{\Gamma \vdash \{P\}\ \textbf{while}(b)\ c\, \{\ Q \wedge \neg[\![b]\!]\Gamma\ |R\}}$$

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

Ein kurzes Beispiel

```
void foo(){
int x, y, *z; /* Locations: l, m, n */

/** \s. read(upd(upd(upd(s, n, l), l, 0),
          read(upd(upd(s, n, l), l, 0), n), 5), l) = 5 */
z= &x;
/** \s. read(upd(upd(s, l, 0),
                 read(upd(s, l, 0), n), 5), l) = 5 */ (3)
x= 0;
/** \s. read(upd(s, read(s, n), 5), l) = 5) */ (2)
*z= 5;
/** \s. read(s, l) = 5 */                          (1)
/** \s. read(upd(s, m, read(s, l)), m) = 5 */
y= x;
/** \s. read(s, m) = 5
/** { y == 5 } */
```

Ein kurzes Beispiel

- An der Stelle (1) können wir direkt vereinfachen
- An den Stellen (2) und (3) ist keine Zwischenvereinfachung mehr möglich
- Die finale Vorbedingung wird wie folgt vereinfacht:

$$read(upd(upd(upd(\sigma, n, I), I, 0), \underline{read(upd(}upd(\sigma, n, I), \underline{I, 0), n)}, 5), I) = 5$$

$$read(upd(upd(upd(\sigma, n, I), I, 0)\underline{read(upd(\sigma, n, I), n)}, 5), I) = 5$$

$$\underline{read(upd(}upd(upd(\sigma, n, I), I, 0), \underline{I, 5), I)} = 5$$

$$5 = 5$$

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 Zustandsprädikaten.
  - Zuweisung wird zu Zustandsupdate.
  - Problem: Vereinfachung von Zuständen benötigt Gleichheit/Ungleichheit von Referenzen
- Nächsten Donnerstag: Gleichheit und Ungleichheit über Loc, Generierung von Vorbedingungen, Definiertheit

Korrekte Software: Grundlagen und Methoden
Vorlesung 13 vom 16.06.16: Referenzen und Speichermodelle

Serge Autexier, Christoph Lüth

Universität Bremen

Sommersemester 2016

Fahrplan

- Einführung
- Die Floyd-Hoare-Logik
- Operationale Semantik
- Denotationale Semantik
- Äquivalenz der Semantiken
- Verifikation: Vorwärts oder Rückwärts?
- Korrektheit des Hoare-Kalküls
- Einführung in Isabelle/HOL
- Weitere Datentypen: Strukturen und Felder
- Funktionen und Prozeduren
- Referenzen und Zeiger
- Frame Conditions & Modification Clauses
- Ausblick und Rückblick

Motivation: Ein kurzes Beispiel

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

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

Probleme

1. Gleichheit und Ungleichheit von Pointern:

$$read(S, \Gamma!x) \stackrel{?}{=} read(S, \Gamma!y)$$

2. Aliasing — unterschiedliche Referenzen auf das gleiche Objekt

3. Gültigkeit von Pointer und Undefiniertheit

Hoare-Regeln für Deklarationen

- Erste Näherung: skalare Typen

$$\frac{}{\Gamma \vdash \{P\}\,\{\,\}\,\{P|R\}}$$

$$\frac{\Gamma, (x : I) \vdash \{P\}\,\{ds\}\,\{Q|R\}}{\Gamma \vdash \{\lambda S.\, I = \mathit{fresh}(S) \wedge P(\mathit{upd}(S, I, \mathit{init}_t))\}\,\{\quad x;\ ds\}\,\{Q|R\}}$$

- init_t ist initialer Wert für Typen t (unbestimmt)
- Aus Definition von *fresh* folgt direkt:

$$\forall I.\, I \in \mathit{dom}(\sigma) \longrightarrow I \neq \mathit{fresh}(\sigma)$$

Getypter Speicher

- Die Operation $\mathit{fresh}(S)$ erzeugt einen frischen Speicherplatz
  - Nebenannahme: es gibt immer frischen Speicher
- Um Strukturen und Felder anzulegen, benötigen wir eine getypte Version.

$$\mathit{sizeof}(b) = 1 \qquad t \text{ ist skalarer Typ}$$

$$\mathit{sizeof}(\textbf{struct } \{\}) = 0$$

$$\mathit{sizeof}(\textbf{struct } \{\mathsf{t}\ \mathsf{i}\,;\ \ \mathsf{flds}\ \}) = \mathit{sizeof}(t) + \mathit{sizeof}(\textbf{struct } \{\ \mathsf{flds}\ \})$$

$$\mathit{sizeof}(\mathsf{t}\ \ \mathsf{id}\,[\mathsf{as}]) = \mathit{sizeof}(t) \cdot \mathit{as}$$

- Damit:

$$\mathit{fresh} : \Sigma \to \textbf{Type} \to \textbf{Loc}$$

$$\mathit{fresh}(\sigma, t) = l \longleftrightarrow \forall 0 \leq i \leq \mathit{sizeof}(t).\ \mathit{add}(l, i) \notin \mathit{dom}(\sigma)$$

- Behandelt einfaches Aliasing

Erweiterungen

- Speicher wird erweitert, indem frischen Lokationen ein indeterminierter Wert zugewiesen wird. Damit sind diese Lokationen gültig, aber nicht sinnvoll lesbar.
- Um den Speicher um strukturierte Typen zu erweitern:

$$
\begin{aligned}
ext &: \Sigma \to \mathbf{Loc} \to \mathbf{Type} \to \Sigma \\
ext(\sigma, l, t) &= upd(\sigma, l, init_t) \qquad t \text{ ist skalarer Typ} \\
ext(\sigma, l, \mathbf{struct}\ \{\}) &= \sigma \\
ext(\sigma, l, \mathbf{struct}\ \{\ \mathsf{t}\ \ \mathsf{i}\,;\ \ \mathsf{flds}\,\}) &= ext(ext(\sigma, l, t), add(l, sizeof(t)), \\
&\qquad\qquad \mathbf{struct}\ \{\,\mathsf{flds}\,\}) \\
ext(\sigma, l, \mathsf{t}\ \ \mathsf{id}\,[0]\,) &= \sigma \\
ext(\sigma, l, \mathsf{t}\ \ \mathsf{id}\,[\mathsf{n}]) &= ext(ext(\sigma, l, t), add(l, sizeof(t)), \mathsf{t}\ \ \mathsf{id}\,[\mathsf{n}-1])
\end{aligned}
$$

Erweiterte Hoare-Regeln für Deklarationen

$$\frac{}{\Gamma \vdash \{P\}\,\{\,\}\,\{P|R\}}$$

$$\frac{\Gamma, (x : I) \vdash \{P\}\,\{ds\}\,\{Q|R\}}{\Gamma \vdash \{\lambda S.\, I = \mathit{fresh}(S, t) \wedge P(\mathit{ext}(S, I, t))\}\,\{t\ x;\ ds\}\,\{Q|R\}}$$

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 abgestürtzt 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`.

Neue Annotationen

- valid (l): l ist eine gültige Lokation

$$[\![\text{valid (l)}]\!]\Gamma \stackrel{def}{=} \lambda S.\, \{add([\![l]\!]\Gamma, x) \mid 0 \leq x < sizeof(\mathbf{Type}(l))\} \subset dom(S)$$

- array (l, n): l ist eine gültige Lokation für ein Feld der Größe n.

$$[\![\text{array (a, n)}]\!]\Gamma \stackrel{def}{=} \lambda S.\, \{add([\![a]\!]\Gamma, x) \mid 0 \leq x < n * sizeof(\mathbf{Type}(a))\} \subset dom(S)$$

- separated (a, m, b, n): Felder a[m] und b[n] sind disjunkt.

$$[\![\text{separated (a, m, b, n)}]\!]\Gamma \stackrel{def}{=} (\{add([\![a]\!]\Gamma, x) \mid 0 \leq x < m * sizeof(\mathbf{Type}(a))\} \cap \{add([\![b]\!]\Gamma, x) \mid 0 \leq x < n * sizeof(\mathbf{Type}(b))\}) = \emptyset$$

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):

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

- Problem: gilt mit Pointern nur eingeschränkt, da c eventuell Teile des Zustands verändert, über den R Aussagen macht.

Modification Sets

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

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

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

Semantik mit Modification Sets

- Hoare-Tripel mit Modification Sets:

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

- wobei $\sigma \cong_L \tau \longleftrightarrow \forall l \in dom(\sigma) \cup dom(\tau) \setminus L.\, \sigma(l) = \tau(l)$

- oder alternativ $\sigma \cong_L \tau \longleftrightarrow \forall l.\, \sigma(l) \neq \tau(l) \longrightarrow l \in L$

Regeln mit Modification Sets

- Regeln werden mit Modification Set annotiert:

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

- Modification Set wird durchgereicht, aber:

$$\frac{\begin{array}{l}\Gamma, \Lambda \vdash \{\lambda\sigma.[\![l]\!]\Gamma \in dom(\sigma) \wedge [\![l]\!]\Gamma \in \Lambda \wedge Q(upd(\sigma, [\![l]\!]\Gamma, [\![e]\!]\Gamma))\} \\ \quad l = e \\ \quad \{Q|R\}\end{array}}{}$$

Das Beispiel vom Anfang

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

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

Brauchen wir **pre** x != y?

Swap (Annahme: &x != &y)

```
int swap(int *x, int *y) {
  /** { &x!=&y, *y == \old(*y), *x == \old(*x) } */
  int z;
  /** { &x!=&y, &z!=x, &z!=y, *y == \old(*y), *x == \old(*x) } */
  /** Beweis:
      [read(s,x)!=read(s,y), read(s,read(s,y)) == read(sold,read(sold,y)),
       read(s,z) == read(sold,read(sold,x))] upd(s,z,read(s,read(s,x)))
     <==> read(s,x)!=read(s,y), read(s,read(s,y)) == read(sold,read(sold,y)),
          read(s,read(s,x)) == read(sold,read(sold,x))
          : Da z!=read(s,x), z!=read(s,y) */
  z = *x;
  /** { &x!=&y, &z!=x, &z!=y, *y == \old(*y), z == \old(*x) } */
  /** { &x!=&y, *y == \old(*y), z == \old(*x) } */
  /** Beweis:
      [read(s,x)!=read(s,y), read(s,read(s,x)) == read(sold,read(sold,y)),
       read(s,z) == read(sold,read(sold,x))] upd(s,read(s,x),read(s,read(s,y)))
     <==> read(s,x)!=read(s,y), read(s,read(s,y)) == read(sold,read(sold,y)),
          read(s,z) == read(sold,read(sold,x))
        : Da z!=read(s,x), x!=read(s,x) und y!=read(s,x)  */
  *x = *y;
  /** { &x!=&y, *x == \old(*y), z == \old(*x) } */
  /** Beweis:
      [read(s,read(s,x)) == read(sold,read(sold,y)),read(s,read(s,y)) ==
       read(sold,read(sold,x))] upd(s,read(s,y),read(s,z))
     <==> read(s,read(s,x)) == read(sold,read(sold,y)), read(s,z) ==
          read(sold,read(sold,x)) : Da &x!=&y (read(s,x)!=read(s,y)) */
  *y = z;
  /** { &x!=&y, *x == \old(*y), *y == \old(*x) } */
  /** { *x == \old(*y), *y == \old(*x) } */
}
```

Swap (Annahme: &x == &y)

```
int swap(int *x, int *y) {
  /** { &x==&y, *x == \old(*x), *y == \old(*y) } */
  int z;
  /** { &x==&y, *x == \old(*x), *y == \old(*y) } */
  /** Beweis:
      [read(s,x) == read(s,y),
      read(s,z) == read(sold,read(sold,x)),
      read(s,z) == read(sold,read(sold,y))] upd(s,z,read(s,read(s,x)))
     <==> read(s,x)!=read(s,y), read(s,read(s,x)) == read(sold,read(sold,x)),
          read(s,read(s,y)) == read(sold,read(sold,y))
          -- da z!=&x, z != &y */
  z = *x;
  /** { &x==&y, z == \old(*x), z == \old(*y) } */
  /** Beweis:
     [read(s,x) == read(s,y),
      read(s,z) == read(sold,read(sold,x)),
      read(s,z) == read(sold,read(sold,y))](upd(s, read(s, x),read(s,read(s,y))))
   <==> read(s,x) == read(s,y), read(s,z) == read(sold,read(sold,y)),
          read(s,z) == read(sold,read(sold,x)) -- da z != &x, z != &y */
  *x = *y;
  /** { &x==&y, z == \old(*x), z == \old(*y) } */
  /** Beweis:
      [read(s, x) == read(s,y), read(s, read(s,x)) == read(sold,read(sold,x)),
       read(s, read(s, y)) == read(sold,read(sold,y))] (upd(s,read(s,y),read(s,z)))
     <==> read(s,x) == read(s,y), read(s,z) == read(sold,read(sold,x)),
          read(s,z) == read(sold,read(sold,y)) : Da read(s,x)==read(s,y) */
  *y = z;
  /** { &x==&y, *x == \old(*y), *y == \old(*x) } */
}
```

Zusammenfassung

- Herleitung von Gleichheit, Ungleichheit und Validität von Pointern ist schwierig.
- Dazu: kürzere Beschreibung des Zustands, Separation Logic
- Der Zustand ist immer noch sehr groß.
  - Wir können inbesondere keine Beweisverpflichtung zwischendurch erledigen.
- Dazu: Vorwärtsrechnung.

Korrekte Software: Grundlagen und Methoden
Vorlesung 14 vom 23.06.16: VCG Revisited

Serge Autexier, Christoph Lüth

Universität Bremen

Sommersemester 2016

Motivation

- Rückwärtsrechnung: es entstehen viele indeterminierte Zwischenzustände, über die wir nichts sagen können.
- Bsp. *swap*: Validität von $*y$ in der letzten Anweisung.
- Dadurch können Beweisverpflichtungen nicht direkt bewiesen werden.
- Die den Zustand beschreibenden Ausdrücke werden immer größer.

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

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

Approximative stärkste Nachbedingung (revisited)

$$\mathrm{asp}(\Gamma, \Lambda, P, c) \qquad \mathrm{svc}(\Gamma, \Lambda, P, c)$$

- Γ ist das environment
- Λ ist der modification set
- $P : \Sigma \to \mathbf{T}$ ist die Vorbedingung (vor c)
- c ist ein Statement
- $\mathrm{svc}(\Gamma, \Lambda, P, c)$ sind die Verifikationsbedingungen
- $\mathrm{asp}(\Gamma, \Lambda, P, c) : \Sigma \to \mathbf{T}$ gilt nach c, wenn:

(i) vorher P gilt,

(ii) c terminiert, und

(iii) die Verifikationsbedingungen $\mathrm{svc}(\Gamma, \Lambda, P, c)$ gelten:

$$\mathrm{svc}(\Gamma, \Lambda, P, c) \longrightarrow \models \{P\}\, c\, \{\mathrm{asp}(\Gamma, \Lambda, P, c)\}$$

Approximative stärkste Nachbedingung

$$
\begin{aligned}
\mathrm{asp}(\Gamma, \Lambda, P, \{\,\}) &\stackrel{def}{=} P \\
\mathrm{asp}(\Gamma, \Lambda, P, \{c\ c_s\}) &\stackrel{def}{=} \mathrm{asp}(\Gamma, \Lambda, \mathrm{asp}(\Gamma, \Lambda, P, c), \{c_s\}) \\
\mathrm{asp}(\Gamma, \Lambda, P, l = e) &\stackrel{def}{=} \lambda S.\, \exists S_0.\, S = upd(S_0, [\![l]\!]^\Gamma_{S_0}, [\![e]\!]^\Gamma_{S_0}) \wedge P(S_0 \\
\mathrm{asp}(\Gamma, \Lambda, P, l = f(e_1, \ldots, e_n)) &\stackrel{def}{=} \\
&\lambda S.\, \exists S_0.\, S = upd(S_0, [\![l]\!]^\Gamma_{S_0}, F([\![e_1]\!]^\Gamma_{S_0}, \ldots, [\![e_n]\!]^\Gamma_{S_0})) \wedge P(S_0 \\
&\text{mit } post(\Gamma!f) \equiv (\forall v_1, \ldots, v_n.\, \textbf{result} = F(v_1, \ldots, v_n) \\
\mathrm{asp}(\Gamma, \Lambda, P, f(e_1, \ldots, e_n)) &\stackrel{def}{=} \lambda S.\, \exists S_0.\, Q([\![e_1]\!]^\Gamma_{S_0}, \ldots, [\![e_n]\!]^\Gamma_{S_0})(S_0, S) \\
&\text{mit } post(\Gamma!f) \equiv (\forall v_1, \ldots, v_n.\, Q(v_1, \ldots, v_n) \\
\mathrm{asp}(\Gamma, \Lambda, P, \textbf{if}\ (b)\ c_0\ \textbf{else}\ c_1) &\stackrel{def}{=} [\![b]\!]^\Gamma \wedge \mathrm{asp}(\Gamma, \Lambda, P, c_0) \\
&\quad \vee (\neg [\![b]\!]^\Gamma \wedge \mathrm{asp}(\Gamma, \Lambda, c_1, P)) \\
\mathrm{asp}(\Gamma, \Lambda, P, /{**}\ \{q\}\ {*}/, P) &\stackrel{def}{=} [\![q]\!]^\Gamma \\
\mathrm{asp}(\Gamma, \Lambda, P, \textbf{while}\ (b)\ /{**}\ \textbf{inv}\ \ i\ {*}/\ c, P) &\stackrel{def}{=} [\![i]\!]^\Gamma \wedge \neg([\![b]\!]^\Gamma) \\
\mathrm{asp}(\Gamma, \Lambda, P, \textbf{return}\ e) &\stackrel{def}{=} \lambda S.\, post(\Gamma)[[\![e]\!]^\Gamma_S / \textbf{result}]\ S \\
\mathrm{asp}(\Gamma, \Lambda, P, \textbf{return}) &\stackrel{def}{=} post(\Gamma)
\end{aligned}
$$

ASP: Sonderregeln

$$\mathsf{asp}(\Gamma, \Lambda, \lambda S.\, \exists S_0.\, S = f(S_0) \wedge P(S_0), l = e) \stackrel{def}{=}$$
$$\lambda S.\, \exists S_0.\, S = upd(f(S_0), [\![l]\!]^{\Gamma}_{f(S_0)}, [\![e]\!]^{\Gamma}_{f(S_0)}) \wedge P(S_0)$$

$$\mathsf{asp}(\Gamma, \Lambda, \lambda S.\, \exists S_0.\, S = f(S_0) \wedge P(S_0), l = g(e_1, \ldots, e_n)) \stackrel{def}{=}$$
$$\lambda S.\, \exists S_0.\, S = upd(f(S_0), [\![l]\!]^{\Gamma}_{f(S_0)}, G([\![e_1]\!]^{\Gamma}_{f(S_0)}, \ldots, [\![e_n]\!]^{\Gamma}_{f(S_0)})) \wedge P(S_0)$$
$$\text{mit } post(\Gamma, \Lambda!g) \equiv (\forall v_1, \ldots, v_n.\, \textbf{result} = G(v_1, \ldots, v_n))$$

$$\mathsf{asp}(\Gamma, \Lambda, \lambda S.\, \exists S_0.\, S = f(S_0) \wedge P(S_0), p(e_1, \ldots, e_n)) \stackrel{def}{=}$$
$$\lambda S.\, \exists S_0.\, Q([\![e_1]\!]^{\Gamma}_{f(S_0)}, \ldots, [\![e_n]\!]^{\Gamma}_{f(S_0)})(f(S_0), S)$$
$$\text{mit } post(\Gamma, \Lambda!f) \equiv (\forall v_1, \ldots, v_n.\, Q(v_1, \ldots, v_n)$$

ASP: Weitere Anmerkungen

- $\mathrm{asp}(\Gamma, \Lambda, P, c)$ ist vom Typ $\Sigma \to \mathbf{T}$.
- Boolsche Operatoren sind geliftet:

$$[\![i]\!]^\Gamma \wedge \neg([\![b]\!]^\Gamma) \equiv \lambda S.\, [\![i]\!]^\Gamma_S \wedge (\neg([\![b]\!]^\Gamma_S))$$

- Zusatzbedingungen:
  1. Für alle Zuweisungsregeln:
     $[\![l]\!]^\Gamma_S$ ist im modification set und eine gültige Lokation
  2. Für die Zuweisungsregel mit Funktionsaufruf:
     modification set von f ist leer $(mod(\Gamma!f) = \emptyset)$

Verifikationsbedingungen

$$
\begin{aligned}
\mathrm{svc}(\Gamma, \Lambda, P, \{\,\}) &\stackrel{def}{=} \emptyset \\
\mathrm{svc}(\Gamma, \Lambda, P, \{c\ c_s\}) &\stackrel{def}{=} \mathrm{svc}(\Gamma, \Lambda, P, c) \\
&\quad \cup \mathrm{svc}(\Gamma, \Lambda, \mathrm{asp}(\Gamma, \Lambda, P, c), \{c_s\}) \\
\mathrm{svc}(\Gamma, P, I = e) &\stackrel{def}{=} \{\ \forall S.\, P(S) \longrightarrow \mathit{valid}(S, [\![I]\!]^\Gamma_S), \\
&\quad\ \forall S.\, P(S) \longrightarrow [\![I]\!]^\Gamma_S \in \Lambda\,\} \\
\mathrm{svc}(\Gamma, \Lambda, P, I = f(e_1, \ldots, e_n)) &\stackrel{def}{=} P \longrightarrow \mathit{pre}(\Gamma!f)([\![e_1]\!]^\Gamma_S, \ldots, [\![e_n]\!]^\Gamma) \\
&\quad \cup \{\forall S.\, P(S) \longrightarrow \mathit{valid}(S, [\![I]\!]^\Gamma_S), \\
&\quad\ \forall S.\, P(S) \longrightarrow [\![I]\!]^\Gamma_S \in \Lambda\,\} \\
\mathrm{svc}(\Gamma, \Lambda, P, \textbf{if}\ (b)\ c_0\ \textbf{else}\ c_1) &\stackrel{def}{=} \mathrm{svc}(\Gamma, \Lambda, [\![b]\!]^\Gamma \wedge P, c_0) \\
&\quad \cup\ \mathrm{svc}(\Gamma, \Lambda, \neg[\![b]\!]^\Gamma \wedge P, c_1) \\
\mathrm{svc}(\Gamma, \Lambda, P, /{**}\ \{q\}\ {*}/) &\stackrel{def}{=} \{\forall S.\, P(S) \longrightarrow [\![q]\!]^\Gamma_S\} \\
\mathrm{svc}(\Gamma, \Lambda, P, \textbf{while}\ (b)\ /{**}\ \textbf{inv}\ \ i\ {*}/\ c) &\stackrel{def}{=} \\
&\quad \{\forall S.\ \mathrm{asp}(\Gamma, \Lambda, [\![b]\!]^\Gamma \wedge [\![i]\!]^\Gamma, c)(S) \longrightarrow [\![i]\!]^\Gamma_S(S)\} \\
&\quad \cup \{\forall S.\, P(S) \longrightarrow [\![i]\!]^\Gamma_S\} \cup\ \mathrm{svc}(\Gamma, \Lambda, [\![b]\!]^\Gamma \wedge [\![i]\!]^\Gamma, c) \\
\mathrm{svc}(\Gamma, \Lambda, P, \textbf{return}\ e) &\stackrel{def}{=} \{\forall S.\, P(S) \longrightarrow \mathit{post}(\Gamma)[[\![e]\!]^\Gamma_S/\textbf{return}](S)\}
\end{aligned}
$$

Beispiel

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

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

Beispiel

```
int max(int a[], int a_len)
/** pre \array(a, a_len);
    post forall int i; 0 <= i && i < a_len --> a[i]<= \resu
 */
{
 int x;
 int r;

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

Korrekte Software: Grundlagen und Methoden
Vorlesung 15 vom 30.06.16: Separation Logic
Slides courtesy of Rajeev Goré, ANU, Australia

Serge Autexier, Christoph Lüth

Universität Bremen

Sommersemester 2016

Axiom Schemas

$$p_1 * p_2 \Leftrightarrow p_2 * p_1$$

$$(p_1 * p_2) * p_3 \Leftrightarrow p_1 * (p_2 * p_3)$$

$$p * \mathbf{emp} \Leftrightarrow p$$

$$(p_1 \vee p_2) * q \Leftrightarrow (p_1 * q) \vee p_2 * q)$$

$$(p_1 \wedge p_2) * q \Leftrightarrow (p_1 * q) \wedge p_2 * q)$$

$$(\exists x.p) * q \Leftrightarrow \exists x.(p * q) \qquad \text{when } x \text{ not free in } q$$

$$(\forall x.p) * q \Leftrightarrow \forall x.(p * q) \qquad \text{when } x \text{ not free in } q$$

Unsound axiom schemas

$$p \Rightarrow p * p \qquad \text{(Contraction)}$$

$$p * p \Rightarrow p \qquad \text{(Weakening)}$$

More valid axiom schemas

$$p_1 \wedge p_2 \Rightarrow p_1 * p_2 \quad \text{when } p_1 \text{ or } p_2 \text{ pure}$$

$$p_1 * p_2 \Rightarrow p_1 \wedge p_2 \quad \text{when } p_1 \text{ and } p_2 \text{ pure}$$

$$(p \wedge q) * r \Rightarrow p \wedge (q * r) \quad \text{when } p \text{ pure}$$

Pure Expressions

An expression e is pure, if it does neither contain $\mapsto$, $\rightsquigarrow$ *nor* **emp**.

Showing $x = y$

$$(6)\{x = x_1 \wedge x \mapsto v * y = y_1 \wedge y \mapsto v\} x := [x]; y = y$$
$$\{x = v \wedge x_1 \mapsto v * y = v \wedge y_1 \mapsto v\}$$

$$x = v \wedge x_1 \mapsto v * y = v \wedge y_1 \mapsto v$$
$$\Rightarrow x = v * x_1 \mapsto v * y = v \wedge y_1 \mapsto v \qquad x = v \text{ pure}$$
$$\Rightarrow x = v * x_1 \mapsto v * y = v * y_1 \mapsto v \qquad y = v \text{ pure}$$
$$\Rightarrow (x = v * y = v) * x_1 \mapsto v * y_1 \mapsto v$$
$$\Rightarrow (x = v \wedge y = v) * x_1 \mapsto v * y_1 \mapsto v x = v \text{ and } y = v \text{ pure}$$
$$\Rightarrow x = y * x_1 \mapsto v * y_1 \mapsto v$$

Mutation

$$\{e \mapsto -\}[e] := e'\{e \mapsto e'\}$$

Example $[x] := \mathsf{cpns}(3, 4)$

St	Hp		$x := \mathsf{cpns}(3,4)$	St	Hp	
$x = 20$	20	21		$x = 20$	20	21
$*1$	2			3	4	

Axiom Instance

$$\{x \mapsto 20, 21\}[x] := \mathit{cons}(3, 4)\{x \mapsto 3, 4\}$$

Mutation (backwards)

$$\{e \mapsto - * (e \mapsto e' \mathrel{-\!\!*} p)\}[e] := e'\{p\}$$

Example $[x] := \mathsf{cpns}(3, 4)$

St	Hp		$x := \mathsf{cpns}(3,4)$	St	Hp	
$x = 20$	20	21		$x = 20$	20	21
$*1$	2			3	4	

Axiom Instance

$$\{x \mapsto 20, 21 * (x \mapsto 3, 4 \mathrel{-\!\!*} x \mapsto 3 \wedge x + 1 \mapsto 4)\}[x] := \mathit{cons}(3, 4)\{x \mapsto 3 \wedge x$$

Summary

- Separation logic is the method to really handle point structures
- Can also handle function and procedure calls.
- Needs to be adapted for *C*

Korrekte Software: Grundlagen und Methoden
Vorlesung 16 vom 07.07.16: Rückblick & Ausblick

Serge Autexier, Christoph Lüth

Universität Bremen

Sommersemester 2016

Fahrplan

- Einführung
- Die Floyd-Hoare-Logik
- Operationale Semantik
- Denotationale Semantik
- Äquivalenz der Semantiken
- Verifikation: Vorwärts oder Rückwärts?
- Korrektheit des Hoare-Kalküls
- Einführung in Isabelle/HOL
- Weitere Datentypen: Strukturen und Felder
- Funktionen und Prozeduren
- Referenzen und Zeiger
- Frame Conditions & Modification Clauses
- Ausblick und Rückblick

Was gibt's heute?

- Rückblick
- Ausblick
- Feedback

Rückblick

Semantik

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

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)
- Zusammenhang mit denotationaler/operationaler Semantik
- VCG: Schwächste Vorbedingung und stärkste Nachbedingung

Erweiterung der Programmiersprache

- Für jede Erweiterung:
  - Wie modellieren wir semantisch?
  - Wie ändern sich die Regeln der Logik?
- Strukturen und Felder
  - Lokationen, **Lexp**, strukturierte Werte
  - Erweiterte Substitution in Zuweisungsregel
- Prozeduren und Funktionen
  - Modellierung von **return**: Erweiterung zu $\Sigma \rightharpoonup \Sigma \times \mathbf{V}_U$
  - Spezifikation von Funktionen durch Vor-/Nachbedingungen
  - Spezifikation der Funktionen muss im Kontext stehen

Erweiterung der Programmiersprache

- Zeiger und Referenzen
  - Lokationen nicht mehr symbolisch (Variablennamen), sondern abstrakt $\Sigma = \mathbf{Loc} \rightharpoonup \mathbf{Val}, \mathbf{Val} = \mathbf{N} + \mathbf{C} + \mathbf{Loc}$
  - Zustand wird als abstrakter Datentyp mit Operationen *Read* und *Upd* modelliert
  - Zuweisung nicht mehr mit Substitution/Ersetzung, sondern explizit durch *Upd*
  - Spezifikationen sind Zustandsprädikate
- 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
  $\longrightarrow$ Umständlich zu modellieren, Effekt zweitrangig

- Implementationsabhängiges, unspezifiziertes und undefiniertes Verhalten
  $\longrightarrow$ Genauere Unterscheidung in der Semantik

Kontrollstrukturen:

- **switch** $\longrightarrow$ Ist im allgemeinen Fall ein **goto**

- **goto**, setjmp/longjmp
  $\longrightarrow$ Tiefe Änderung der Semantik (*continuations*)

Die Sprache C: Was haben wir ausgelassen?

Typen:

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

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 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)
     - Werkzeuge: `absint`
  2. Halbautomatisch: Korrektheitsannotationen, Überprüfung automatisch (nicht immer sound, aber vollständig)
     - Werkzeuge: JML (ESC/Java, Krakatao; Java), Boogie und Why (generisches VCG), Frama-C (C), VCC (C), Spark (ADA)
  3. Interaktiv: Einbettung der Sprache in interaktiven Theorembeweiser (Isabelle, Coq)
     - Beispiele: L4.verified, VeriSoft, SAMS

Feedback

Deine Meinung zählt

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

Tschüß!