

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

Serge Autexier, Christoph Lüth

Universität Bremen

Sommersemester 2016

18:10:54 2016-07-07

1 [66]



Organisatorisches

► Veranstalter:

Christoph Lüth
christoph.lueth@dfki.de
MZH 4185, Tel. 59830

Serge Autexier
serge.autexier@dfki.de
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/~cx1/lehre/ksgm.ss16>

Korrekte Software

2 [66]



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.

Korrekte Software

3 [66]



Warum Korrekte Software?

Korrekte Software

4 [66]



Software-Disaster I: Therac-25



Korrekte Software

5 [66]



Bekanntes Software-Disaster II: Ariane-5



Korrekte Software

6 [66]



Bekanntes Software-Disaster III: Airbus A400M



Korrekte Software

7 [66]



Inhalt der Vorlesung

Korrekte Software

8 [66]



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?

Korrekte Software

9 [66]



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**)

Korrekte Software

10 [66]



Nächste Woche

- ▶ Aussagenlogik
- ▶ Erstes Übungsblatt

Korrekte Software

11 [66]



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

Korrekte Software

12 [66]



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)
```

Korrekte Software

13 [66]



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
```

Korrekte Software

14 [66]



Conciseness

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



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

=> Reverse me!
```

Korrekte Software

15 [66]



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
```

Korrekte Software

16 [66]



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)
```



Variables

Scala
s:String
i:Int

Java
String s
int i / Integer i

```
val s = "Hello World"
var i = 1
private var k = 3
```

```
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 doSomething(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	Java
<code>println("Hello")</code>	<code>System.out.println("Hello");</code>
<code>val line = readLine()</code>	<code>BufferedReader r = new BufferedReader(new InputStreamRead(System.in)); String line = r.readLine();</code>
<code>sys.error("Bad")</code>	<code>throw new RuntimeException("Bad")</code>
<code>1 + 1</code> <code>1 .+(1)</code>	<code>new Integer(1).toInt() + new Integer(1).toInt();</code>
<code>1 == new Object</code> <code>1 eq new Object</code>	<code>new Integer(1).equals(new Object()); new Integer(1) == new Object();</code>
<code>""A\sregex"".r</code>	<code>java.util.regex.Pattern.compile("A\sregex");</code>
<code>s"3 + 4 = \${3 + 4}" // "3 + 4 = 7"</code>	<code>"3 + 4 = " + (3 + 4)</code>

Korrekte Software 41 [66]

Topics

- ▶ Basic syntax
 - ▶ REPL
 - ▶ First class functions
 - ▶ Pattern matching
 - ▶ OO and traits
 - ▶ Functional programming
 - ▶ Higher-Order Functions
 - ▶ Implicits
 - ▶ (XML)
- Korrekte Software 42 [66]

REPL - Read eval print loop

- ▶ Command line shell for on-the-fly execution of Scala statements
 - ▶ `bin/scala`
- Korrekte Software 43 [66]

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 available 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 Maven compatible build tool for Scala and Java <http://www.scala-sbt.org/>
- Korrekte Software 44 [66]

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
```

Korrekte Software 45 [66]

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)
```

Korrekte Software 46 [66]

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
```

Korrekte Software 47 [66]

Deferred execution - constructed example

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

Korrekte Software 48 [66]

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("Alf")
```



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? $p = n!$
 - ▶ Wie können wir das **beweisen**?
 - ▶ Wir berechnen symbolisch, welche Werte Variablen über den Programmverlauf annehmen.
- ```

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

```
- ▶ Um Aussagen über ein Programm 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.



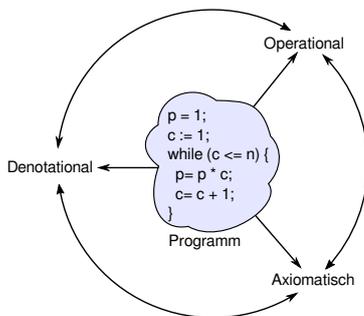
## Semantik von Programmiersprachen

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



## Floyd-Hoare-Logik

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



## Unsere Programmiersprache

Wir betrachten einen Ausschnitt der Programmiersprache **C** (**C0**).

Ausbaustufe 1 kennt folgende Konstrukte:

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



## C0: Ausdrücke und Anweisungen

```

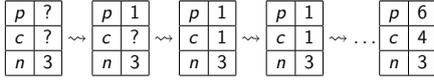
Aexp a ::= N | Loc | a1 + a2 | a1 - a2 | a1 * a2 | a1 / a2
Bexp b ::= 0 | 1 | a1 == a2 | a1 != a2
 | a1 <= a2 | !b | b1 && b2 | b1 || b2
Exp e ::= Aexp | Bexp
Stmnt c ::= Loc = Exp;
 | if (b) c1 else c2
 | while (b) c
 | {c*}

```



## 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 werden zu **Werten Val** (hier ganze Zahlen) aus.
- Adressen **Loc** sind hier Programmvariablen (Namen)
- Ein **Systemzustand** bildet Adressen auf Werte ab:  $\Sigma = \text{Loc} \rightarrow \text{Val}$
- 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  $\sigma$ , die  $P$  erfüllen: **wenn** die Ausführung von  $c$  mit  $\sigma$  in  $\sigma'$  terminiert, **dann** erfüllt  $\sigma' Q$

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

$c$  ist **total korrekt**, wenn für alle Zustände  $\sigma$ , die  $P$  erfüllen: die Ausführung von  $c$  mit  $\sigma$  in  $\sigma'$  terminiert, und  $\sigma'$  erfüllt  $Q$ .

- $\models \{1\} \text{while}(1)\{1\}$  gilt
- $\models [1] \text{while}(1)\{1\}$  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

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

- Für jedes Konstrukt der Programmiersprache gibt es eine Regel.



## Regeln der Floyd-Hoare-Logik: Zuweisung

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

- 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:

$$\frac{}{\vdash \{5 < 10 \leftrightarrow (x < 10)[x/5]\} x = 5 \{x < 10\}} \quad \frac{}{\vdash \{x < 9 \leftrightarrow x + 1 < 10\} x = x + 1 \{x < 10\}}$$



## Regeln der Floyd-Hoare-Logik: Fallunterscheidung und Sequenzierung

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

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

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

- Hier wird ein Zwischenzustand  $B$  benötigt.



## Regeln der Floyd-Hoare-Logik: Iteration

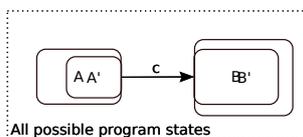
$$\frac{\vdash \{A \wedge [[b]]\} c \{A\}}{\vdash \{A\} \text{while}(b) c \{A \wedge \neg[[b]]\}}$$

- 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

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



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



## Überblick: die Regeln des Floyd-Hoare-Kalküls

$$\frac{}{\vdash \{P[[e]/X]\} x = e \{P\}} \quad \frac{}{\vdash \{A\} \{c\} \{A\}} \quad \frac{\vdash \{A\} c \{B\} \quad \vdash \{B\} \{c_s\} \{C\}}{\vdash \{A\} \{c c_s\} \{C\}} \quad \frac{\vdash \{A \wedge [[b]]\} c_0 \{B\} \quad \vdash \{A \wedge \neg[[b]]\} c_1 \{B\}}{\vdash \{A\} \text{if}(b) c_0 \text{else} c_1 \{B\}} \quad \frac{\vdash \{A \wedge [[b]]\} c \{A\}}{\vdash \{A\} \text{while}(b) c \{A \wedge \neg[[b]]\}} \quad \frac{A' \rightarrow A \quad \vdash \{A\} c \{B\} \quad B \rightarrow B'}{\vdash \{A'\} c \{B'\}}$$



## Eigenschaften der Floyd-Hoare-Logik

### Korrektheit

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

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

### Relative Vollständigkeit

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

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



## Warum Verifikation?

Hier sind Varianten des Fakultätsbeispiels.  
Welche sind korrekt?

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



## Eine Handvoll Beispiele

```
// {y = Y ∧ y ≥ 0} // {0 ≤ a}
x = 1; t = 1;
while (y != 0) { s = 1;
 y = y - 1; i = 0;
 x = 2 * x; while (s <= a) {
} t = t + 2;
// {x = 2^Y} s = s + t;
 i = i + 1;
 }
// {a ≥ 0 ∧ b ≥ 0} // {i^2 ≤ a ∧ a < (i+1)^2}
r = b; }
q = 0; while (b <= r) {
while (b <= r) { r = r - y;
 r = r - y; q = q + 1;
 q = q + 1; }
} // {a = b * q + r ∧ r < b}
```



## Zusammenfassung

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



# Korrekte Software: Grundlagen und Methoden

## Vorlesung 3 vom 18.04.16: Operationale Semantik

Serge Autexier, Christoph Lüth

Universität Bremen

Sommersemester 2016

18:10:57 2016-07-07

1 [18]



## 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

Korrekte Software

2 [18]



## 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

Korrekte Software

3 [18]



## 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 ( $=$ ,  $!$ ,  $<$ ,  $\dots$ ), boolesche Operatoren ( $\&\&$ ,  $\|\|$ );
- ▶ Anweisungen:
  - ▶ Fallunterscheidung (**if...else...**), Iteration (**while**), Zuweisung, Blöcke;
  - ▶ Sequenzierung und leere Anweisung sind implizit

Korrekte Software

4 [18]



## 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:  $\Sigma = \text{Loc} \rightarrow \text{Val}$
- ▶ Ein Programm bildet einen Anfangszustand **möglicherweise** auf einen Endzustand ab (wenn es **terminiert**).
- ▶ Zusicherungen sind Prädikate über dem Systemzustand.

Korrekte Software

5 [18]



## C0: Ausdrücke und Anweisungen

```
Aexp a ::= N | Loc | a1 + a2 | a1 - a2 | a1 * a2 | a1 / a2
Bexp b ::= 0 | 1 | a1 == a2 | a1 != a2
 | a1 <= a2 | !b | b1 && b2 | b1 || b2
Exp e ::= Aexp | Bexp
Stmt c ::= Loc = Exp;
 | if (b) c1 else c2
 | while (b) c
 | {c*}
```

Korrekte Software

6 [18]



## 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}
```

Korrekte Software

7 [18]



## Operationale Semantik: Arithmetische Ausdrücke

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

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

### Regeln

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

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

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

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

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

Korrekte Software

8 [18]



## Operationale Semantik: Arithmetische Ausdrücke

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

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

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

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



## Operationale Semantik: Arithmetische Ausdrücke

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

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



## Beispiel Ableitungen

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

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

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

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

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



## Operationale Semantik: Boolesche Ausdrücken

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

Rules

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

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

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

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

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



## Operationale Semantik: Anweisungen

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

Regeln

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

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

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

Definiere:

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

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

$$\frac{\langle a, \sigma \rangle \rightarrow_{Aexp} n \in \mathbf{N} \quad \langle a, \sigma \rangle \rightarrow_{Aexp} \perp \quad \langle c, \sigma \rangle \rightarrow_{Stmt} \sigma' \neq \perp \quad \langle \{c_s\}, \sigma' \rangle \rightarrow_{Stmt} \sigma'' \neq \perp}{\langle X = a, \sigma \rangle \rightarrow_{Stmt} \sigma[n/X] \quad \langle X = a, \sigma \rangle \rightarrow_{Stmt} \perp \quad \langle \{c \ c_s\}, \sigma \rangle \rightarrow_{Stmt} \sigma''}$$



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

## Beispiel $b_1 \&\& b_2, \sigma \rightarrow_{Bexp} t$

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

wobei  $x = 1$ ;  
 $t = \text{wenn } (t_1 \neq t_2 = 0) \{$   
 $t = 1 \text{ wenn } t_1 = 1 \text{ oder } (t_1 = 0 \text{ und } t_2 = 1);$   
 $t = \perp \text{ sonst;}$   
 $\}$   
 $// x = 2^y$   
 $\sigma(y) = 3$



## Äquivalenz arithmetischer Ausdrücke

Gegeben zwei Aexp  $a_1$  and  $a_2$

► Sind sie gleich?  $\langle a_1, \sigma \rangle \rightarrow_{Aexp} n_1 \quad \langle a_2, \sigma \rangle \rightarrow_{Aexp} n_2$

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

$$(X * X) + 2 * X * Y \text{ (if } (X \neq Y) \ \text{else } (X + Y)) * (X + Y)$$

► Wann sind sie gleich?  $\langle b, \sigma \rangle \rightarrow_{Bexp} \perp \quad \langle b, \sigma \rangle \rightarrow_{Bexp} 0$

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

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



## Äquivalenz Boolescher Ausdrücke

Gegeben zwei Bexp-Ausdrücke  $b_1$  and  $b_2$

► Sind sie gleich?

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

$$A \parallel (A \&\& B) \quad \text{und} \quad A$$



## Beweisen

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

### Definition

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

Ein einfaches Beispiel:

### Lemma

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

Beweis an der Tafel



## Zusammenfassung

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



Korrekte Software: Grundlagen und Methoden  
Vorlesung 4 vom 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

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

Ein einfaches Beispiel:

Lemma

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

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

- ▶ Kleinsten Fixpunkt
- ▶ Denotationale Semantik für C0



Regeln und Regelinstanzen

Definition

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

- ▶ Betrachte folgende Regelmengen

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

- ▶ Regelinstanzen sind

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

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



Induktive Definierte Mengen

Definition

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

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

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



Beispiel

- ▶ Betrachte folgende Regelmengen

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

- ▶ 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

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

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

Definition (Abgeschlossen und Monoton)

- ▶ Eine Menge  $S$  ist abgeschlossen unter  $R$  ( $R$ -abgeschlossen) gdw.

$$\hat{R}(S) \subseteq S$$

- ▶ Eine Operation  $f$  ist monoton gdw.

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



## Kleinster Fixpunkt Operator

### Lemma

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

### Lemma

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

- (a)  $A$  ist  $R$ -abgeschlossen,
- (b)  $\hat{R}(A) = A$ , und
- (c)  $A$  ist die kleinste  $R$ -abgeschlossene Menge.



### Beweis von Lemma (a).

$A$  ist  $R$ -abgeschlossen:

Sei  $\frac{x_1, \dots, x_k}{y} \in R$  und  $x_1, \dots, x_k \subseteq A$ . Da  $A = \bigcup_{i \in \mathbb{N}} A_i$ , gibt es ein  $l$  so dass  $x_1, \dots, x_k \subseteq A_l$ . 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  $\frac{x_1, \dots, x_k}{y} \in R$  geben mit  $x_1, \dots, x_k \subseteq A_{n-1} \subseteq A$ . Also ist  $y \in \hat{R}(A)$ . □



### Beweis von Lemma (c).

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

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

Basisfall  $A_0 = \emptyset \subseteq B$

Induktionsschritt Da  $B$   $R$ -abgeschlossen ist gilt:  $\hat{R}(B) \subseteq B$ .

Induktionsannahme:  $A_n \subseteq B$ .

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



## Kleinster Fixpunkt Operator

### Definition

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

ist der **kleinste Fixpunkt**.



## Kleinster Fixpunkt

► Betrachte folgende Regelmengen  $R$

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

► Was sind

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

$$\hat{R}^2 = ?$$

$$\hat{R}^3 = ?$$

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

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



## Denotationale Semantik - Motivation

► Operationale Semantik:

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

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

► Denotationale Semantik: Eine Menge von Regeln, die ein Programm in eine **partielle Funktion** von Zustand nach Zustand überführen

Denotat

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



## Denotationale Semantik - Motivation

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

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

or

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

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



## Denotierte Funktionen

► jeder  $a : \mathbf{Aexp}$  denotiert eine partielle Funktion  $\Sigma \rightarrow \mathbf{N}$

► jeder  $b : \mathbf{Bexp}$  denotiert eine partielle Funktion  $\Sigma \rightarrow \mathbf{T}$

► jedes  $c : \mathbf{Stmt}$  denotiert eine partielle Funktion  $\Sigma \rightarrow \Sigma$



## Denotat von Aexp

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

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



## Denotat von Bexp

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

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



## Denotat von Bexp

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

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



## Denotat von Stmt

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

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

Aber was ist

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



## Denotationale Semantik für while

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

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

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



## Denotationale Semantik von while

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

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

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

$$\Gamma(\varphi) = \{(\sigma, \sigma') \mid \exists \sigma''. \mathcal{B}[b](\sigma) = \text{true} \wedge (\sigma, \sigma'') \in \mathcal{D}[c] \wedge (\sigma'', \sigma') \in \varphi\} \cup \{(\sigma, \sigma) \mid \mathcal{B}[b](\sigma) = \text{false}\}$$



## Denotationale Semantik von while

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

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

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

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

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

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



## Denotation für Stmt

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

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

mit

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



## Weitere Intuition zur Fixpunkt Konstruktion

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

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

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

Denotational  $\mathcal{E}[a]$

$$\begin{array}{l}
 m \in \mathbf{N} \quad \langle m, \sigma \rangle \rightarrow_{Aexp} m \quad \{(\sigma, m) \mid \sigma \in \Sigma\} \\
 x \in \mathbf{Loc} \quad \frac{x \in \text{Dom}(\sigma)}{\langle x, \sigma \rangle \rightarrow_{Aexp} \sigma(x)} \quad \{(\sigma, \sigma(x)) \mid \sigma \in \Sigma, x \in \text{Dom}(\sigma)\} \\
 a_1 \circ a_2 \quad \frac{\langle a_1, \sigma \rangle \rightarrow_{Aexp} n \quad \langle a_2, \sigma \rangle \rightarrow_{Aexp} m \quad n, m \neq \perp}{\langle a_1 \circ a_2, \sigma \rangle \rightarrow_{Aexp} n \circ^l m} \quad \{(\sigma, n \circ^l m) \mid \sigma \in \Sigma, (\sigma, n) \in \mathcal{E}[a_1], (\sigma, m) \in \mathcal{E}[a_2]\} \\
 \frac{\langle a_1, \sigma \rangle \rightarrow_{Aexp} n \quad \langle a_2, \sigma \rangle \rightarrow_{Aexp} m \quad n = \perp \text{ oder } m = \perp}{\langle a_1 \circ a_2, \sigma \rangle \rightarrow_{Aexp} \perp} \quad \circ \in \{+, \times, -\}
 \end{array}$$



Operationale vs. denotationale Semantik

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

Denotational  $\mathcal{E}[a]$

$$\begin{array}{l}
 a_1 / a_2 \quad \frac{\langle a_1, \sigma \rangle \rightarrow_{Aexp} n \quad \langle a_2, \sigma \rangle \rightarrow_{Aexp} m \quad m \neq 0 \quad m, n \neq \perp}{\langle a_1 / a_2, \sigma \rangle \rightarrow_{Aexp} n \circ^l m} \quad \{(\sigma, n/m) \mid \sigma \in \Sigma, (\sigma, n) \in \mathcal{E}[a_1], (\sigma, m) \in \mathcal{E}[a_2], m \neq 0\} \\
 \frac{\langle a_1, \sigma \rangle \rightarrow_{Aexp} n \quad \langle a_2, \sigma \rangle \rightarrow_{Aexp} m \quad n = \perp, m = \perp \text{ oder } m = 0}{\langle a_1 / a_2, \sigma \rangle \rightarrow_{Aexp} \perp}
 \end{array}$$



Äquivalenz operationale und denotationale Semantik

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

$$\begin{array}{l}
 \langle a, \sigma \rangle \rightarrow_{Aexp} n \Leftrightarrow (\sigma, n) \in \mathcal{E}[a] \\
 \langle a, \sigma \rangle \rightarrow_{Aexp} \perp \Leftrightarrow \sigma \notin \text{Dom}(\mathcal{E}[a])
 \end{array}$$

► Beweis per struktureller Induktion über  $a$ .



Operationale vs. denotationale Semantik

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

Denotational  $\mathcal{B}[b]$

$$\begin{array}{l}
 1 \quad \langle 1, \sigma \rangle \rightarrow_{Bexp} 1 \quad \{(\sigma, 1) \mid \sigma \in \Sigma\} \\
 0 \quad \langle 0, \sigma \rangle \rightarrow_{Bexp} 0 \quad \{(\sigma, 0) \mid \sigma \in \Sigma\}
 \end{array}$$



Operationale vs. denotationale Semantik

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

Denotational  $\mathcal{B}[b]$

$$\begin{array}{l}
 a_0 == a_1 \quad \frac{\langle a_0, \sigma \rangle \rightarrow_{Aexp} n \quad \langle a_1, \sigma \rangle \rightarrow_{Aexp} m \quad n, m \neq \perp \quad n = m}{\langle a_0 == a_1, \sigma \rangle \rightarrow_{Aexp} 1} \quad \frac{\langle a_0, \sigma \rangle \rightarrow_{Aexp} n \quad \langle a_1, \sigma \rangle \rightarrow_{Aexp} m \quad n, m \neq \perp \quad n \neq m}{\langle a_0 == a_1, \sigma \rangle \rightarrow_{Aexp} 0} \\
 \frac{\langle a_0, \sigma \rangle \rightarrow_{Aexp} n \quad \langle a_1, \sigma \rangle \rightarrow_{Aexp} m \quad n = \perp \text{ oder } m = \perp}{\langle a_0 == a_1, \sigma \rangle \rightarrow_{Aexp} \perp}
 \end{array}$$

$\leq$  analog



Operationale vs. denotationale Semantik

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

Denotational  $\mathcal{B}[b]$

$$\begin{array}{l}
 b_1 \&\&b_0 \quad \frac{\langle b_1, \sigma \rangle \rightarrow_{Bexp} 0 \quad \langle b_1 \&\&b_2, \sigma \rangle \rightarrow 0}{\langle b_1, \sigma \rangle \rightarrow_{Bexp} 1} \quad \frac{\langle b_1, \sigma \rangle \rightarrow_{Bexp} 1 \quad \langle b_2, \sigma \rangle \rightarrow_{Bexp} b}{\langle b_1 \&\&b_2, \sigma \rangle \rightarrow b} \quad \{(\sigma, 0) \mid (\sigma, 0) \in \mathcal{B}[b_1]\} \\
 \frac{\langle b_1, \sigma \rangle \rightarrow_{Bexp} \perp}{\langle b_1 \&\&b_2, \sigma \rangle \rightarrow \perp} \quad \{(\sigma, b) \mid (\sigma, 1) \in \mathcal{B}[b_1], (\sigma, b) \in \mathcal{B}[b_2]\} \\
 b_1 || b_2 \quad \text{analog} \\
 !n \quad \dots
 \end{array}$$



Äquivalenz operationale und denotationale Semantik

► Für alle  $b \in \mathbf{Bexp}$ , für alle  $t \in \mathbf{B}$ , für alle Zustände  $\sigma$ :

$$\begin{array}{l}
 \langle b, \sigma \rangle \rightarrow_{Bexp} t \Leftrightarrow (\sigma, t) \in \mathcal{B}[b] \\
 \langle b, \sigma \rangle \rightarrow_{Bexp} \perp \Leftrightarrow \sigma \notin \text{Dom}(\mathcal{B}[b])
 \end{array}$$

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



## Operationale vs. denotationale Semantik

Operational  $\langle a, \sigma \rangle \rightarrow_{Stmt} c$       Denotational  $\mathcal{D}[c]$

$$\{c_1 \dots c_n\} \quad \frac{\langle c_1, \sigma \rangle \rightarrow_{Stmt} \sigma'' \neq \perp \quad \langle \{c_2 \dots c_n\}, \sigma' \rangle \rightarrow_{Stmt} \sigma''}{\langle \{c_1 \dots c_n\}, \sigma \rangle \rightarrow_{Stmt} \sigma''} \quad \mathcal{B}[c_n] \circ \dots \circ \mathcal{B}[c_1] \circ Id$$

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

$$x = a \quad \frac{\langle a, \sigma \rangle \rightarrow_{Aexp} n}{\langle x = a, \sigma \rangle \rightarrow_{Stmt} \sigma[n/x]} \quad \{(\sigma, \sigma[n/x]) \mid (\sigma, n) \in \mathcal{E}[a]\}$$

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



## Operationale vs. denotationale Semantik

Operational  $\langle a, \sigma \rangle \rightarrow_{Stmt} c$       Denotational  $\mathcal{D}[c]$

$$\text{if } (b) \ c_0 \quad \frac{\langle b, \sigma \rangle \rightarrow_{Bexp} 1 \quad \langle c_0, \sigma \rangle \rightarrow_{Stmt} \sigma'}{\langle c, \sigma \rangle \rightarrow_{Stmt} \sigma'} \quad \{(\sigma, \sigma') \mid (\sigma, 1) \in \mathcal{B}[b], (\sigma, \sigma') \in \mathcal{D}[c_0]\}$$

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

$$\text{else } c_1 \quad \frac{\langle b, \sigma \rangle \rightarrow_{Bexp} 0 \quad \langle c_1, \sigma \rangle \rightarrow_{Stmt} \sigma'}{\langle c, \sigma \rangle \rightarrow_{Stmt} \sigma'} \quad \{(\sigma, \sigma') \mid (\sigma, 0) \in \mathcal{B}[b], (\sigma, \sigma') \in \mathcal{D}[c_1]\}$$



## Operationale vs. denotationale Semantik

Operational  $\langle c, \Sigma \rangle \rightarrow_{Stmt} \Sigma \mid \perp$       Denotational  $\mathcal{D}[c]$

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

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

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

mit

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



## Äquivalenz operationale und denotationale Semantik

► Für alle  $c \in \mathbf{Stmt}$ , für alle Zustände  $\sigma, \sigma'$ :

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

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

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



Serge Autexier, Christoph Lüth

Universität Bremen

Sommersemester 2016



## Idee

- Hier ist ein einfaches Programm:

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

- Wir sehen:

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

- Muss das so sein? Ist das immer so?



## Vorwärts!

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

$$\vdash \{P[\llbracket e \rrbracket / X]\} x = e \{P\}$$

- Alternative Zuweisungsregel (nach Floyd):

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

- $FV(P)$  sind die **freien** Variablen in  $P$ .
- Jetzt ist die Vorbedingung offen — Regel kann vorwärts angewandt werden.



## Vorwärtsverkettung

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

- Ein einfaches Beispiel (nach Mike Gordon):

```
// {x = 1}
x = x + 1;
// {\exists V.x = \llbracket x + 1 \rrbracket[V/x] \wedge (x = 1)[V/x]}
```

- Vereinfachung der Nachbedingung:

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



## Vorwärtsverkettung

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

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

- Vorteile?

- Wir wollten doch sowieso die Anwendung automatisieren...



## Schwächste Vorbedingung, stärkste Nachbedingung

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

$$\begin{aligned} \models \{P\} c \{Q\} &\leftrightarrow P \rightarrow wp(c, Q) \\ \models \{P\} c \{Q\} &\leftrightarrow sp(P, c) \rightarrow Q \end{aligned}$$



## Berechnung von $wp(c, Q)$

- Einfach für Programme ohne Schleifen:

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

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

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

- 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  $awp(c, Q)$  zusammen mit einer Menge von **Verifikationsbedingungen**  $wvc(c, Q)$
  - oder die **approximative** stärkste Nachbedingung  $asp(P, c)$  zusammen mit einer Menge von **Verifikationsbedingungen**  $svc(P, c)$
- Es gilt:

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



## Approximative schwächste Vorbedingung

|                                                                     |                                                                                                                                                                                                                                           |
|---------------------------------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| $\text{awp}(\{\}, P)$                                               | $\stackrel{\text{def}}{=} P$                                                                                                                                                                                                              |
| $\text{awp}(x = e, P)$                                              | $\stackrel{\text{def}}{=} P[\llbracket e \rrbracket / x]$                                                                                                                                                                                 |
| $\text{awp}(\{c \ c_s\}, P)$                                        | $\stackrel{\text{def}}{=} \text{awp}(c, \text{awp}(\{c_s\}, P))$                                                                                                                                                                          |
| $\text{awp}(\text{if } (b) \ c_0 \ \text{else } \ c_1, P)$          | $\stackrel{\text{def}}{=} (b \wedge \text{awp}(c_0, P)) \vee (\neg b \wedge \text{awp}(c_1, P))$                                                                                                                                          |
| $\text{awp}(/\!/\ \{q\} \ */ , P)$                                  | $\stackrel{\text{def}}{=} \llbracket q \rrbracket$                                                                                                                                                                                        |
| $\text{awp}(\text{while } (b) \ /\!/\ \text{inv } \ i \ */ \ c, P)$ | $\stackrel{\text{def}}{=} \llbracket i \rrbracket$                                                                                                                                                                                        |
| $\text{wvc}(\{\}, P)$                                               | $\stackrel{\text{def}}{=} \emptyset$                                                                                                                                                                                                      |
| $\text{wvc}(x = e, P)$                                              | $\stackrel{\text{def}}{=} \emptyset$                                                                                                                                                                                                      |
| $\text{wvc}(\{c \ c_s\}, P)$                                        | $\stackrel{\text{def}}{=} \text{wvc}(c, \text{awp}(\{c_s\}, P)) \cup \text{wvc}(\{c_s\}, P)$                                                                                                                                              |
| $\text{wvc}(\text{if } (b) \ c_0 \ \text{else } \ c_1, P)$          | $\stackrel{\text{def}}{=} \text{wvc}(c_0, P) \cup \text{wvc}(c_1, P)$                                                                                                                                                                     |
| $\text{wvc}(/\!/\ \{q\} \ */ , P)$                                  | $\stackrel{\text{def}}{=} \{ \llbracket q \rrbracket \rightarrow P \}$                                                                                                                                                                    |
| $\text{wvc}(\text{while } b \ /\!/\ \text{inv } \ i \ */ \ c, P)$   | $\stackrel{\text{def}}{=} \text{wvc}(c, \llbracket i \rrbracket)$<br>$\cup \{ \llbracket i \rrbracket \wedge b \rightarrow \text{awp}(c, \llbracket i \rrbracket) \}$<br>$\cup \{ \llbracket i \rrbracket \wedge \neg b \rightarrow P \}$ |



## Approximative stärkste Nachbedingung

|                                                                     |                                                                                                                                                                                                                                                                                  |
|---------------------------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| $\text{asp}(P, \{\})$                                               | $\stackrel{\text{def}}{=} P$                                                                                                                                                                                                                                                     |
| $\text{asp}(P, x = e)$                                              | $\stackrel{\text{def}}{=} \exists V. x = \llbracket e \rrbracket[V/x] \wedge P[V/x]$                                                                                                                                                                                             |
| $\text{asp}(P, \{c \ c_s\})$                                        | $\stackrel{\text{def}}{=} \text{asp}(\text{awp}(P, c), c_s)$                                                                                                                                                                                                                     |
| $\text{asp}(P, \text{if } (b) \ c_0 \ \text{else } \ c_1, P)$       | $\stackrel{\text{def}}{=} \text{asp}(\llbracket b \rrbracket \wedge P, c_0) \vee \text{asp}(\neg \llbracket b \rrbracket \wedge P, c_1)$                                                                                                                                         |
| $\text{asp}(P, /\!/\ \{q\} \ */ )$                                  | $\stackrel{\text{def}}{=} \llbracket q \rrbracket$                                                                                                                                                                                                                               |
| $\text{asp}(P, \text{while } (b) \ /\!/\ \text{inv } \ i \ */ \ c)$ | $\stackrel{\text{def}}{=} \llbracket i \rrbracket \wedge \neg \llbracket b \rrbracket$                                                                                                                                                                                           |
| $\text{svc}(P, \{\})$                                               | $\stackrel{\text{def}}{=} \emptyset$                                                                                                                                                                                                                                             |
| $\text{svc}(P, x = e)$                                              | $\stackrel{\text{def}}{=} \emptyset$                                                                                                                                                                                                                                             |
| $\text{svc}(P, \{c \ c_s\})$                                        | $\stackrel{\text{def}}{=} \text{svc}(P, c) \cup \text{svc}(\text{awp}(P, c), \{c_s\})$                                                                                                                                                                                           |
| $\text{svc}(P, \text{if } (b) \ c_0 \ \text{else } \ c_1)$          | $\stackrel{\text{def}}{=} \text{svc}(\llbracket b \rrbracket \wedge P, c_0) \cup \text{svc}(\neg \llbracket b \rrbracket \wedge P, c_1)$                                                                                                                                         |
| $\text{svc}(P, /\!/\ \{q\} \ */ )$                                  | $\stackrel{\text{def}}{=} \{ P \rightarrow \llbracket q \rrbracket \}$                                                                                                                                                                                                           |
| $\text{svc}(P, \text{while } b \ /\!/\ \text{inv } \ i \ */ \ c)$   | $\stackrel{\text{def}}{=} \{ P \rightarrow \llbracket i \rrbracket \}$<br>$\cup \{ \text{asp}(\llbracket b \rrbracket \wedge \llbracket i \rrbracket, c) \rightarrow \llbracket i \rrbracket \}$<br>$\cup \text{svc}(\llbracket b \rrbracket \wedge \llbracket i \rrbracket, c)$ |



## 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:  $\text{svc}(F, \top)$
- ▶ Berechnung rückwärts:  $\text{wvc}(\top, F)$



## 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.



Serge Autexier, Christoph Lüth

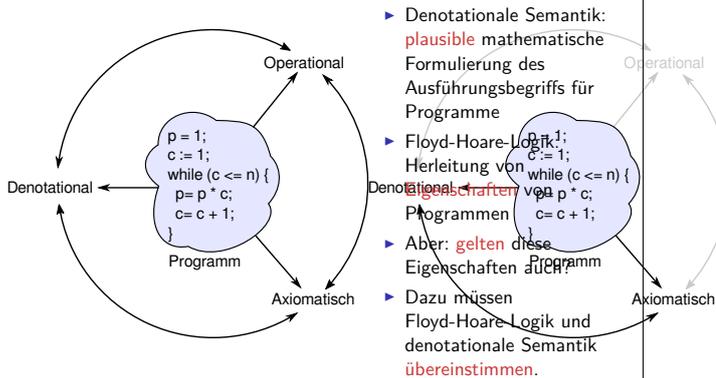
Universität Bremen

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

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

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

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

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

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

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

## Floyd-Hoare-Tripel: Gültigkeit und Herleitbarkeit

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

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

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

Bezug zur Semantik?

## Hoare-Tripel und denotationale Semantik

- ▶ Mit der denotationalen Semantik können wir die Gültigkeit von Hoare-Tripeln **formal** definieren.

- ▶ Notation: für  $P \in \mathbf{Bexp}, \sigma \models P \iff \mathcal{B}[P](\sigma) = 1$

### Gültigkeit von Hoare-Tripeln

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

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

## Überblick: die Regeln des Floyd-Hoare-Kalküls

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

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

$$\frac{\vdash \{A \wedge [[b]]\} c_0 \{B\} \quad \vdash \{A \wedge \neg[[b]]\} c_1 \{B\}}{\vdash \{A\} \text{if } (b) \text{ c}_0 \text{ else } c_1 \{B\}}$$

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

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

## Korrektheit und Vollständigkeit

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

- ▶ Wir können nur gültige Eigenschaften von Programmen herleiten.

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

- ▶ Wir können alle gültigen Eigenschaften auch herleiten.

## Korrektheit der Floyd-Hoare-Logik

Floyd-Hoare-Logik ist korrekt.

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

Beweis:

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



## Vollständigkeit der Floyd-Hoare-Logik

Floyd-Hoare-Logik ist vollständig modulo weakening.

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

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



## Zusammenfassung

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



Korrekte Software: Grundlagen und Methoden  
Vorlesung 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



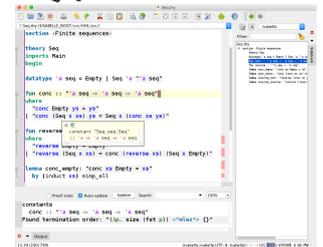
## Motivation

- ▶ 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](http://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

$$\bigwedge x_1 \dots x_n. \text{assumptions} \implies \text{conclusion}$$

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

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



## 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)

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

Korrekte Software

9 [21]



## 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

Korrekte Software

10 [21]



## 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

$$\frac{P \text{ 0} \quad P (\text{Suc } 0) \quad \bigwedge n. P \text{ n} \implies P (\text{Suc } (\text{Suc } n))}{\text{ALL n. } P \text{ n}}$$

- ▶ Name: div2.induct

Korrekte Software

11 [21]



## 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

$$\frac{P \text{ 0} \quad P (\text{Suc } 0) \quad \bigwedge n. P \text{ n} \implies P (\text{Suc } (\text{Suc } n))}{\text{ALL n. } P \text{ n}}$$

- ▶ Name: div2.induct

Korrekte Software

12 [21]



## 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

Korrekte Software

13 [21]



## 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, Funktionsdefinitionen (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

Korrekte Software

14 [21]



## 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

Korrekte Software

15 [21]



## 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"

Korrekte Software

16 [21]



## 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

$$\frac{P\ 0 \quad \bigwedge n. P\ (\text{div2}\ n) \implies P\ n}{\text{ALL}\ n.\ P\ n}$$

- ▶ 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)

$$\frac{\Gamma \implies ?P \quad \Gamma \implies ?Q}{\Gamma \implies ?P \wedge ?Q} \text{conjI} \qquad \frac{\Gamma, ?P, ?Q \implies G}{\Gamma, ?P \wedge ?Q \implies G} \text{conjE}$$

- ▶ **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

$$\frac{\Gamma, A \implies B}{\Gamma \implies A \rightarrow B} \text{impl} \qquad \frac{\bigwedge x. [\Gamma \implies (?Px)]}{\Gamma \implies \forall x. (?Px)} \text{allI}$$



## Regeln für Gleichheit

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

- ▶ 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

18.11.02 2016-07-07

1 [18]



## 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

Korrekte Software

2 [18]



## Motivation

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

Korrekte Software

3 [18]



## 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;
```

Korrekte Software

4 [18]



## Strings

```
char hallo[5] = {'h', 'a', 'l', 'l', 'o', '\0'};
char hallo[] = "hallo";
printf(hallo[4]); /* liefert 'o' */
```

Korrekte Software

5 [18]



## 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;
}
ksgm.cp = 6;
```

Korrekte Software

6 [18]



## 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;
}
```

Korrekte Software

7 [18]



## Ausdrücke

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

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

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

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

$\text{ExpList } el ::= e (, el)?$

Korrekte Software

8 [18]



## 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) c1 else c2
 | 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<sub>B</sub>)

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

►  $\forall c, c' \in \text{Dom}(\sigma). c$  ist kein Präfix von  $c'$  und umgekehrt.

► if  $c[i]c' \in \text{Dom}(\sigma)$  then  $\forall 0 \leq j \leq i. \exists c_j. c[j]c_j \in \text{Dom}(\sigma)$

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

$$\sigma|_u := \{(v, n) \mid (uv, n) \in \sigma\}$$



## Beispiel

Programm

Zustand

```

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'}
};

```

|                                    |                                    |
|------------------------------------|------------------------------------|
| $x.[0].c[0] \rightarrow 1$         | $x.[1].c[0] \rightarrow 3$         |
| $x.[0].c[1] \rightarrow 2$         | $x.[1].c[1] \rightarrow 4$         |
| $x.[0].b.name[0] \rightarrow 'n'$  | $x.[1].b.name[0] \rightarrow 'n'$  |
| $x.[0].b.name[1] \rightarrow 'a'$  | $x.[1].b.name[1] \rightarrow 'a'$  |
| $x.[0].b.name[2] \rightarrow 'm'$  | $x.[1].b.name[2] \rightarrow 'm'$  |
| $x.[0].b.name[3] \rightarrow 'e'$  | $x.[1].b.name[3] \rightarrow 'e'$  |
| $x.[0].b.name[4] \rightarrow '1'$  | $x.[1].b.name[4] \rightarrow '2'$  |
| $x.[0].b.name[5] \rightarrow '\0'$ | $x.[1].b.name[5] \rightarrow '\0'$ |



## Auswertung von Lexp zu Cont

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

$$\frac{\langle \text{lexp}, \sigma \rangle \rightarrow_{\text{Lexp}} c \quad \langle a, \sigma \rangle \rightarrow_{\text{Aexp}} i}{\langle \text{lexp}[a], \sigma \rangle \rightarrow_{\text{Lexp}} c[i]}$$

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



## Aexp: Operationale Semantik

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

$$\frac{\langle \text{lexp}, \sigma \rangle \rightarrow_{\text{Lexp}} c \quad c \notin \text{Dom}(\sigma)}{\langle \text{lexp}, \sigma \rangle \rightarrow_{\text{Aexp}} \perp}$$

$$\frac{\langle \text{str}, \sigma \rangle \rightarrow_{\text{Lexp}} s :: \text{char}[n], \quad l = \min(\{n+1\} \cup \{m \mid m < n, s[m] = '\0', s[0..m-1] \neq '\0'\})}{\langle \text{strlen}(\text{str}), \sigma \rangle \rightarrow_{\text{Aexp}} l}$$



## Operationale Semantic: Zuweisungen

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

```

Stmt c ::= decl
 | Lexp = Exp;
 | if (b) c1 else c2
 | while (b) c
 | {c*}

```



## Denotationale Semantik

► Denotation für **Lexp**

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

$$\mathcal{L}[\text{lexp}[a]] = \{(\sigma, l[i]) \mid (\sigma, l) \in \mathcal{L}[\text{lexp}], (\sigma, i) \in \mathcal{E}[a]\}$$

$$\mathcal{L}[\text{lexp.name}] = \{(\sigma, l.name) \mid (\sigma, l) \in \mathcal{L}[\text{lexp}]\}$$

► Denotation für **Zuweisungen**

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



## 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 } */
```



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:

**FunDef** ::= *Id*(**Param**<sup>\*</sup>) **FunSpec**<sup>+</sup> **Blk**

**Param** ::= **Type** *Id*

**Blk** ::= {**Decl**<sup>\*</sup> **Stmt**}

**Decl** ::= **Type** *Id* = **Init** | **Type** *Id*

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



## Erweiterte Semantik

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

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



## Semantik von Anweisungen

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

$\mathcal{D}[x = e] = \{(\sigma, \sigma(c \mapsto a)) \mid (\sigma, c) \in \mathcal{L}[x], (\sigma, a) \in \mathcal{E}[e]\}$

$\mathcal{D}[\{c\}] = \mathcal{D}[c] \circ \mathcal{D}[c]$  Komposition wie oben

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

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

$\mathcal{D}[\text{return } e] = \{(\sigma, (\sigma, a)) \mid (\sigma, a) \in \mathcal{E}[e]\}$

$\mathcal{D}[\text{return}] = \{(\sigma, (\sigma, *))\}$

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

$\Gamma(\psi) \stackrel{\text{def}}{=} \{(\sigma, \tau) \mid (\sigma, 1) \in \mathcal{B}[b] \wedge (\sigma, \tau) \in \psi \circ \mathcal{D}[c]\} \cup \{(\sigma, \sigma) \mid (\sigma, 0) \in \mathcal{B}[b]\}$



## Semantik von Funktionsdefinitionen

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

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

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

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



## Semantik von Blöcken und Deklarationen

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

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

Blöcke bestehen aus Deklarationen und einer Anweisung:

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

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

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

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



## Funktionsaufrufe

- ▶ Aufruf einer Funktion:  $f(t_1, \dots, t_n)$ :
  - ▶ Auswertung der Argumente  $t_1, \dots, t_n$
  - ▶ Einsetzen in die Semantik  $\mathcal{D}_{fd}[f]$
- ▶ Was ist mit **Seiteneffekten**?
  - ▶ Erst mal gar nichts...
- ▶ 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):

$$Env = Id \rightarrow [\mathbf{FunDef}]$$

$$= Id \rightarrow \mathbf{V}^N \rightarrow \Sigma \rightarrow (\Sigma \times \mathbf{V}_U)$$

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

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

- ▶ 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:

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

|               |                              |                                                                     |
|---------------|------------------------------|---------------------------------------------------------------------|
| Vorbedingung  | <b>pre</b> sp;               | $\Sigma \rightarrow \mathbf{T}$                                     |
| Nachbedingung | <b>post</b> sp;              | $\Sigma \times (\Sigma \times \mathbf{V}_U) \rightarrow \mathbf{T}$ |
|               | $\backslash \mathbf{old}(e)$ | Wert von $e$ im <b>Vorzustand</b>                                   |
|               | $\backslash \mathbf{result}$ | <b>Rückgabewert</b> der Funktion                                    |



## Semantik von Spezifikationen

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

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

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

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

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

$$\dots$$

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

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

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

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



## Gültigkeit von Spezifikationen

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

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

$$\iff \forall \nu_1, \dots, \nu_n. \mathcal{D}_{fd}[\mathbf{FunDef}] \Gamma \in \mathcal{B}_{sp}[\mathbf{pre} \ p \ \mathbf{post} \ q] \Gamma$$

- ▶  $\Gamma$  enthält globale Definitionen, insbesondere andere Funktionen.
- ▶ Vgl.  $\models \{P\} c \{Q\}$  für Hoare-Triple
- ▶ Aber wie **beweisen** wir das? → 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



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 Funktionsspezifikationen

**FunDef** ::= Type Id(Param\*) FunSpec<sup>+</sup> Blk  
**Param** ::= Type Id  
**FunSpec** ::= /\*\* pre Bexpr post Bexpr \*/  
**Blk** ::= {Decl\* Stmt}  
**Decl** ::= Type Id = Init | Type Id

- ▶ 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;
}
```

$$\frac{P \rightarrow P'_{\text{old}(s) \rightarrow s} \quad \{P'\} c \{Q \mid Q'\} \quad \{P'\} c \{\text{result} = \text{old}(x)! \mid \text{result} = \text{old}(x)!\}}{\tau_0 f(\tau_1 v_1, \dots, \tau_n v_n) \text{ /** pre } P \text{ post } Q \text{ */ } / c \text{ int factorial(int x) /** pre } x \geq 0 \text{ post } \text{result} = \text{old}(x)! \text{ */}}$$



## Hoare-Kalkül mit return

$$\frac{\overline{\{P\}\{P \mid Q\}}}{\frac{\{P\}c\{Q'_1 \mid Q_2\} \quad \{Q'_1\}c\{Q_1 \mid Q_2\}}{\{P\}c \text{ cs}\{Q_1 \mid Q_2\}}}$$

$$\frac{\overline{\{Q_1[e/l]\}l = e\{Q_1 \mid Q_2\}}}{\frac{\{P \wedge b\}c_1\{Q_1 \mid Q_2\} \quad \{P \wedge \neg b\}c_2\{Q_1 \mid Q_2\}}{\{P\} \text{ if } b \text{ c}_1 \text{ else } c_2\{Q_1 \mid Q_2\}}}$$



## While, Weakening

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

$$\frac{P \rightarrow P' \quad \{P'\}c\{Q'_1 \mid Q'_2\} \quad Q'_1 \rightarrow Q_1 \quad Q'_2 \rightarrow Q_2}{\{P\}c\{Q_1 \mid Q_2\}}$$



## Hoare-Kalkül mit return

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

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



## Funktionsaufruf

$$\frac{(V_1 = e_1 \wedge \dots \wedge V_n = e_n \wedge Q_1) \longrightarrow f.pre(e_1, \dots, e_n)}{P := f.post(e_1, \dots, e_n) \backslash result \rightarrow l, \backslash old(v_i) \rightarrow V_i}$$

$$\frac{\{(V_1 = e_1 \wedge \dots \wedge V_n = e_n \wedge Q_1 \wedge P)\{f(e_1, \dots, e_n)/l\}\}}{l = f(e_1, \dots, e_n)}$$

$$\{V_1 = e_1 \wedge \dots \wedge V_n = e_n \wedge Q_1 \wedge P|Q_2\}$$

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



```
int factorial(int x)
/** pre x >= 0
 post \result = \old(x)! */ {
/** { COND: x >= 0 \longrightarrow x >= 0 \wedge x! = x!
 x >= 0 \wedge x! = \old(x)! } */
int r = 0;
/** { x >= 0 \wedge x! = \old(x)! } */
if (x = 0) {
/** { x >= 0 \wedge x = 0 \wedge x! = \old(x)! } */
/** { x >= 0 \wedge x = 0 \wedge 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 \wedge \neg(x = 0) \wedge x! = \old(x)! } */
/** { COND: x >= 0 \wedge \neg(x = 0) \wedge
 factorial(x-1) * x = \old(x)! \wedge \forall x \longrightarrow x - 1 >= 0
 x >= 0 \wedge \neg(x = 0) \wedge factorial(x-1) * x = \old(x)! \wedge
 V = x \wedge factorial(x-1) = (V-1)! } */
r = factorial(x - 1);
/** { x >= 0 \wedge \neg(x = 0) \wedge r * x = \old(x)! \wedge
 V = x \wedge r = (V-1)! } */
/** { x >= 0 \wedge \neg(x = 0) \wedge 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)

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

$$\text{awp}(\Gamma, l = f(e_1, \dots, e_n), P) \stackrel{\text{def}}{=} P[F(\llbracket e_1 \rrbracket, \dots, \llbracket e_n \rrbracket) / l]$$

mit  $\text{post}(\Gamma!f) = (\forall v_1, \dots, v_n. \text{result} = F(v_1, \dots, v_n))$

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

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

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

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

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

$$\text{awp}(\Gamma, \text{return } e, P) \stackrel{\text{def}}{=} \text{post}(\Gamma)[\llbracket e \rrbracket / \text{result}]$$

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



## Approximative schwächste Vorbedingung (Revisited)

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

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

$$\text{wvc}(\Gamma, x = f(e_1, \dots, e_n), P) \stackrel{\text{def}}{=} P \longrightarrow \text{pre}(\Gamma!f)(\llbracket e_1 \rrbracket, \dots, \llbracket e_n \rrbracket)$$

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

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

$$\text{wvc}(\Gamma, /** \{q\} */ , P) \stackrel{\text{def}}{=} \{\llbracket q \rrbracket \longrightarrow P\}$$

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

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



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

Serge Autexier, Christoph Lüth

Universität Bremen

Sommersemester 2016

18.11.04 2016-07-07

1 [19]



## 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).

Korrekte Software

2 [19]



## 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

Korrekte Software

3 [19]



## Motivation

- ▶ Bisher: Zustand ist **Loc**  $\rightarrow$  **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?

Korrekte Software

4 [19]



## Referenzen in C

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

Korrekte Software

5 [19]



## Erweiterung des Zustandsmodells

- ▶ Erweiterung von Zustand und Werten:

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

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

Korrekte Software

6 [19]



## Axiomatisches Zustandsmodell

- ▶ Der Zustand ist ein abstrakter Datentyp  $\Sigma$  mit zwei Operationen und folgenden Gleichungen:

$$\mathit{read} : \Sigma \rightarrow \mathbf{Loc} \rightarrow \mathbf{Val}$$

$$\mathit{upd} : \Sigma \rightarrow \mathbf{Loc} \rightarrow \mathbf{Val} \rightarrow \Sigma$$

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

$$l \neq m \rightarrow \mathit{read}(\mathit{upd}(\sigma, l, v), m) = \mathit{read}(\sigma, m)$$

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

$$l \neq m \rightarrow \mathit{upd}(\mathit{upd}(\sigma, l, v), m, w) = \mathit{upd}(\mathit{upd}(\sigma, m, w), l, v)$$

- ▶ Diese Gleichungen sind **vollständig**.

Korrekte Software

7 [19]



## Axiomatisches Speichermodell

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

$$\mathit{empty} : \Sigma$$

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

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

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

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

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

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

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

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

$$l \neq m \rightarrow \mathit{read}(\mathit{rem}(\sigma, l), m) = \mathit{read}(\sigma, m)$$

Korrekte Software

8 [19]



## Zeigerarithmetik

- ▶ Erklärt noch keine Zeigerarithmetik — dazu:

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

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

$$\begin{aligned} add(l, 0) &= l \\ add(add(l, a), b) &= add(l, a + b) \end{aligned}$$



## Erweiterung der Semantik

- ▶ Problem: **Loc** haben unterschiedliche Semantik auf der linken oder rechten Seite einer Zuweisung.

- ▶  $x = x+1$  — Links: Adresse der Variablen, rechts: Wert an dieser Adresse

- ▶ Lösung: "Except when it is ( . . . ) the operand of the unary & operator, the left operand of the . operator or an assignment operator, an lvalue that does not have array type is converted to the value stored in the designated object (and is no longer an lvalue)" *C99 Standard*, §6.3.2.1 (2)



## Erweiterung der Semantik: Lexp

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

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

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

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

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

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

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

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



## Erweiterung der Semantik: Aexp(1)

$$\mathcal{E}[-] : \text{Env} \rightarrow \text{Aexp} \rightarrow \Sigma \rightarrow \text{Val}$$

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

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

$$e \equiv x \mid lexp[a] \mid lexp.n \mid *e, \quad type(\Gamma, e) \text{ kein Array-Typ}$$

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

$$e \equiv x \mid lexp[a] \mid lexp.n \mid *e, \quad type(\Gamma, e) \text{ Array-Typ}$$

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

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

$type(\Gamma, p) = * \tau, \quad type(\Gamma, a_1) \text{ Integer-Typ}$

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

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



## Erweiterung der Semantik: Aexp(2)

$$\mathcal{E}[-] : \text{Env} \rightarrow \text{Aexp} \rightarrow \Sigma \rightarrow \text{Val}$$

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

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

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



## Hoare-Triple

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

- ▶  $P, Q, R : \Sigma \rightarrow \text{Bool}$  **explizite** Zustandsprädikate
- ▶ Übersetzung  $[\cdot]$  von logischen Formeln in Zustandsprädikate
- ▶ Beispiel:

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

- ▶ Für kürzere Regeln: "Lifting" von Booleschen Operationen:

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

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

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



## Regeln des Hoare-Kalküls

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

$$\frac{\Gamma \vdash \{P\} c \{Q_1\}R \quad \Gamma \vdash \{Q_1\} \{c_s\} \{Q_2\}R}{\Gamma \vdash \{P\} \{c\} \{Q_2\}R}$$

$$\frac{\Gamma \vdash \{P \wedge [b] \Gamma\} c_0 \{Q\}R \quad \Gamma \vdash \{P \wedge \neg [b] \Gamma\} c_1 \{Q\}R}{\Gamma \vdash \{P\} \text{if } (b) \text{ else } c_1 \{Q\}R}$$

$$\frac{\Gamma \vdash \{P \wedge [b] \Gamma\} c \{Q\}R}{\Gamma \vdash \{P\} \text{while}(b) c \{Q \wedge \neg [b] \Gamma\} R}$$

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



## 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 */
/** \s. read(upd(s, m, read(s, l)), m) = 5 */ (1)
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:

$$\begin{aligned} & \text{read}(\text{upd}(\text{upd}(\text{upd}(\sigma, n, l), l, 0), \text{read}(\text{upd}(\text{upd}(\sigma, n, l), l, 0), n), 5), l) = 5 \\ & \text{read}(\text{upd}(\text{upd}(\text{upd}(\sigma, n, l), l, 0) \text{read}(\text{upd}(\sigma, n, l), n), 5), l) = 5 \\ & \text{read}(\text{upd}(\text{upd}(\text{upd}(\sigma, n, l), l, 0), l, 5), l) = 5 \\ & 5 = 5 \end{aligned}$$


## 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



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

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

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

$$\forall I. I \in dom(\sigma) \rightarrow I \neq fresh(\sigma)$$

int



## Getypter Speicher

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

$$\begin{aligned} sizeof(b) &= 1 && t \text{ ist skalarer Typ} \\ sizeof(\mathbf{struct} \{ \}) &= 0 \\ sizeof(\mathbf{struct} \{ t \ i; \text{flds} \}) &= sizeof(t) + sizeof(\mathbf{struct} \{ \text{flds} \}) \\ sizeof(t \ \text{id}[as]) &= sizeof(t) \cdot as \end{aligned}$$

- ▶ Damit:

$$\begin{aligned} fresh : \Sigma \rightarrow \mathbf{Type} \rightarrow \mathbf{Loc} \\ fresh(\sigma, t) = I \iff \forall 0 \leq i \leq sizeof(t). add(I, i) \notin dom(\sigma) \end{aligned}$$

- ▶ 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:

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



## Erweiterte Hoare-Regeln für Deklarationen

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



## 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 abgestürzt wäre.

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



## Neue Annotationen

- ▶ valid(l): l ist eine **gültige** Lokation

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

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

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

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

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



## 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**

**Mexp** ::= Loc | Mexp [\*] | Mexp [i : j] | Mexp . name

- ▶ Mexp sind Lexp, die auch **Teile** von Feldern bezeichnen.

- ▶ Semantik:  $\llbracket - \rrbracket : Env \rightarrow \mathbf{Mexp} \rightarrow \Sigma \rightarrow \mathbb{P}(\mathbf{Loc})$

- ▶ Modification Sets werden in die Hoare-Tripel **integriert**.



## Semantik mit Modification Sets

- ▶ Hoare-Tripel mit Modification Sets:

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

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

- ▶ oder alternativ  $\sigma \cong_L \tau \iff \forall l. \sigma(l) \neq \tau(l) \implies l \in L$



## Regeln mit Modification Sets

- ▶ Regeln werden mit Modification Set annotiert:

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

- ▶ Modification Set wird durchgereicht, aber:

$$\frac{\Gamma, \Lambda \vdash \{ \lambda \sigma. \llbracket l \rrbracket \Gamma \in \text{dom}(\sigma) \wedge \llbracket l \rrbracket \Gamma \in \Lambda \wedge Q(\text{upd}(\sigma, \llbracket l \rrbracket \Gamma, \llbracket e \rrbracket \Gamma)) \}}{l = e \quad \{Q \mid R\}}$$



## 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: $\&x \neq \&y$ )

```
int swap(int *x, int *y) {
 /** { $\&x \neq \&y$, $*y = \text{old}(*y)$, $*x = \text{old}(*x)$ } */
 int z;
 /** { $\&x \neq \&y$, $\&z = x$, $\&z = y$, $*y = \text{old}(*y)$, $*x = \text{old}(*x)$ } */
 /** 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;
 /** { $\&x \neq \&y$, $\&z = x$, $\&z = y$, $*y = \text{old}(*y)$, $z = \text{old}(*x)$ } */
 /** { $\&x \neq \&y$, $*y = \text{old}(*y)$, $z = \text{old}(*x)$ } */
 /** 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;
 /** { $\&x \neq \&y$, $*x = \text{old}(*y)$, $z = \text{old}(*x)$ } */
 /** 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 $\&x \neq \&y$ (read(s,x)!=read(s,y)) */
 *y = z;
 /** { $\&x \neq \&y$, $*x = \text{old}(*y)$, $*y = \text{old}(*x)$ } */
 /** { $*x = \text{old}(*y)$, $*y = \text{old}(*x)$ } */
}
```

Korrekte Software

17 [19]



## Swap (Annahme: $\&x == \&y$ )

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

Korrekte Software

18 [19]



## 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 insbesondere keine Beweisverpflichtung zwischendurch erledigen.
- ▶ Dazu: Vorwärtsrechnung.

Korrekte Software

19 [19]



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

Serge Autexier, Christoph Lüth

Universität Bremen

Sommersemester 2016

18.11.06.2016-07-07

1 | 9



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;
}
```

Korrekte Software

2 | 9



Approximative stärkste Nachbedingung (revisited)

$$\text{asp}(\Gamma, \Lambda, P, c) \quad \text{svc}(\Gamma, \Lambda, P, c)$$

- ▶  $\Gamma$  ist das **environment**
- ▶  $\Lambda$  ist der **modification set**
- ▶  $P : \Sigma \rightarrow \mathbf{T}$  ist die Vorbedingung (vor  $c$ )
- ▶  $c$  ist ein Statement
- ▶  $\text{svc}(\Gamma, \Lambda, P, c)$  sind die **Verifikationsbedingungen**
- ▶  $\text{asp}(\Gamma, \Lambda, P, c) : \Sigma \rightarrow \mathbf{T}$  gilt **nach**  $c$ , wenn:
  - vorher  $P$  gilt,
  - $c$  terminiert, und
  - die Verifikationsbedingungen  $\text{svc}(\Gamma, \Lambda, P, c)$  gelten:
 
$$\text{svc}(\Gamma, \Lambda, P, c) \longrightarrow \models \{P\} c \{\text{asp}(\Gamma, \Lambda, P, c)\}$$

Korrekte Software

3 | 9



Approximative stärkste Nachbedingung

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

Korrekte Software

4 | 9



ASP: Sonderregeln

$$\begin{aligned} \text{asp}(\Gamma, \Lambda, \lambda S. \exists S_0. S = f(S_0) \wedge P(S_0), l = e) &\stackrel{\text{def}}{=} \lambda S. \exists S_0. S = \text{upd}(f(S_0), \llbracket l \rrbracket_{S_0}^r, \llbracket e \rrbracket_{S_0}^r) \wedge P(S_0) \\ \text{asp}(\Gamma, \Lambda, \lambda S. \exists S_0. S = f(S_0) \wedge P(S_0), l = g(e_1, \dots, e_n)) &\stackrel{\text{def}}{=} \lambda S. \exists S_0. S = \text{upd}(f(S_0), \llbracket l \rrbracket_{S_0}^r, G(\llbracket e_1 \rrbracket_{S_0}^r, \dots, \llbracket e_n \rrbracket_{S_0}^r)) \wedge P(S_0) \\ &\quad \text{mit } \text{post}(\Gamma!g) \equiv (\forall v_1, \dots, v_n. \text{result} = G(v_1, \dots, v_n)) \\ \text{asp}(\Gamma, \Lambda, \lambda S. \exists S_0. S = f(S_0) \wedge P(S_0), p(e_1, \dots, e_n)) &\stackrel{\text{def}}{=} \lambda S. \exists S_0. Q(\llbracket e_1 \rrbracket_{S_0}^r, \dots, \llbracket e_n \rrbracket_{S_0}^r)(f(S_0), S) \\ &\quad \text{mit } \text{post}(\Gamma!f) \equiv (\forall v_1, \dots, v_n. Q(v_1, \dots, v_n)) \end{aligned}$$

Korrekte Software

5 | 9



ASP: Weitere Anmerkungen

- ▶  $\text{asp}(\Gamma, \Lambda, P, c)$  ist vom Typ  $\Sigma \rightarrow \mathbf{T}$ .
- ▶ Boolesche Operatoren sind **geliftet**:

$$\llbracket i \rrbracket^r \wedge \neg(\llbracket b \rrbracket^r) \equiv \lambda S. \llbracket i \rrbracket_S^r \wedge \neg(\llbracket b \rrbracket_S^r)$$

- ▶ Zusatzbedingungen:

- Für alle Zuweisungsregeln:  $\llbracket l \rrbracket_S^r$  ist im modification set und eine **gültige** Lokation
- Für die Zuweisungsregel mit Funktionsaufruf: modification set von  $f$  ist leer ( $\text{mod}(\Gamma!f) = \emptyset$ )

Korrekte Software

6 | 9



Verifikationsbedingungen

$$\begin{aligned} \text{svc}(\Gamma, \Lambda, P, \{\}) &\stackrel{\text{def}}{=} \emptyset \\ \text{svc}(\Gamma, \Lambda, P, \{c\ c_s\}) &\stackrel{\text{def}}{=} \text{svc}(\Gamma, \Lambda, P, c) \\ &\quad \cup \text{svc}(\Gamma, \Lambda, \text{asp}(\Gamma, \Lambda, P, c), \{c_s\}) \\ \text{svc}(\Gamma, \Lambda, P, l = e) &\stackrel{\text{def}}{=} \{ \forall S. P(S) \longrightarrow \text{valid}(S, \llbracket l \rrbracket_S^r), \\ &\quad \forall S. P(S) \longrightarrow \llbracket l \rrbracket_S^r \in \Lambda \} \\ \text{svc}(\Gamma, \Lambda, P, l = f(e_1, \dots, e_n)) &\stackrel{\text{def}}{=} P \longrightarrow \text{pre}(\Gamma!f)(\llbracket e_1 \rrbracket_S^r, \dots, \llbracket e_n \rrbracket_S^r) \\ &\quad \cup \{ \forall S. P(S) \longrightarrow \text{valid}(S, \llbracket l \rrbracket_S^r), \\ &\quad \forall S. P(S) \longrightarrow \llbracket l \rrbracket_S^r \in \Lambda \} \\ \text{svc}(\Gamma, \Lambda, P, \text{if } (b) \ c_0 \ \text{else } \ c_1) &\stackrel{\text{def}}{=} \text{svc}(\Gamma, \Lambda, \llbracket b \rrbracket^r \wedge P, c_0) \\ &\quad \cup \text{svc}(\Gamma, \Lambda, \neg \llbracket b \rrbracket^r \wedge P, c_1) \\ \text{svc}(\Gamma, \Lambda, P, \text{while } (b) \ \text{while } \ \text{inv } \ i \ \text{*/ } \ c) &\stackrel{\text{def}}{=} \{ \forall S. P(S) \longrightarrow \llbracket i \rrbracket_S^r \} \\ \text{svc}(\Gamma, \Lambda, P, \text{while } (b) \ \text{while } \ \text{inv } \ i \ \text{*/ } \ c, P) &\stackrel{\text{def}}{=} \{ \forall S. \text{asp}(\Gamma, \Lambda, \llbracket b \rrbracket^r \wedge \llbracket i \rrbracket^r, c)(S) \longrightarrow \llbracket i \rrbracket_S^r(S) \} \\ &\quad \cup \{ \forall S. P(S) \longrightarrow \llbracket l \rrbracket_S^r \} \cup \text{svc}(\Gamma, \Lambda, \llbracket b \rrbracket^r \wedge \llbracket i \rrbracket^r, c) \\ \text{svc}(\Gamma, \Lambda, P, \text{return } e) &\stackrel{\text{def}}{=} \{ \forall S. P(S) \longrightarrow \text{post}(\Gamma) \llbracket \llbracket e \rrbracket_S^r / \text{result} \rrbracket(S) \} \end{aligned}$$

Korrekte Software

7 | 9



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;
}
```

Korrekte Software

8 | 9



## Beispiel

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

Serge Autexier, Christoph Lüth

Universität Bremen

Sommersemester 2016



### Hoare Logic: syntax, semantics and calculus

| Syntax                                               | Semantics                                                                        | Calculus          |
|------------------------------------------------------|----------------------------------------------------------------------------------|-------------------|
| $\neg \wedge \vee \Rightarrow \exists$               | FOL                                                                              | N/A               |
| $= + - \leq \dots$                                   | Arithmetic                                                                       | N/A               |
| <code>:= ; while</code><br><code>if then else</code> | State maps variables to values (no pointers)                                     | N/A               |
| $\{P\}S\{Q\}$                                        | if initial state satisfies $P$ and $S$ terminates then final state satisfies $Q$ | 6 Inference Rules |

### Axiom Schemas

$$\begin{aligned}
 p_1 * p_2 &\Leftrightarrow p_2 * p_1 \\
 (p_1 * p_2) * p_3 &\Leftrightarrow p_1 * (p_2 * p_3) \\
 p * \text{emp} &\Leftrightarrow p \\
 (p_1 \vee p_2) * q &\Leftrightarrow (p_1 * q) \vee p_2 * q \\
 (p_1 \wedge p_2) * q &\Leftrightarrow (p_1 * q) \wedge p_2 * q \\
 (\exists x. p) * q &\Leftrightarrow \exists x. (p * q) && \text{when } x \text{ not free in } q \\
 (\forall x. p) * q &\Leftrightarrow \forall x. (p * q) && \text{when } x \text{ not free in } q
 \end{aligned}$$

#### Unsound axiom schemas

$$\begin{aligned}
 p &\Rightarrow p * p && \text{(Contraction)} \\
 p * p &\Rightarrow p && \text{(Weakening)}
 \end{aligned}$$



### More valid axiom schemas

$$\begin{aligned}
 p_1 \wedge p_2 &\Rightarrow p_1 * p_2 && \text{when } p_1 \text{ or } p_2 \text{ pure} \\
 p_1 * p_2 &\Rightarrow p_1 \wedge p_2 && \text{when } p_1 \text{ and } p_2 \text{ pure} \\
 (p \wedge q) * r &\Rightarrow p \wedge (q * r) && \text{when } p \text{ pure}
 \end{aligned}$$

#### Pure Expressions

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



### Showing $x = y$

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

$$\begin{aligned}
 x = v \wedge x_1 \mapsto v * y = v \wedge y_1 \mapsto v \\
 \Rightarrow x = v * x_1 \mapsto v * y = v \wedge y_1 \mapsto v && x = v \text{ pure} \\
 \Rightarrow x = v * x_1 \mapsto v * y = v * y_1 \mapsto v && y = v \text{ pure} \\
 \Rightarrow (x = v * y = v) * x_1 \mapsto v * y_1 \mapsto v \\
 \Rightarrow (x = v \wedge y = v) * x_1 \mapsto v * y_1 \mapsto v \wedge x = v \text{ and } y = v \text{ pure} \\
 \Rightarrow x = y * x_1 \mapsto v * y_1 \mapsto v
 \end{aligned}$$



### Mutation

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

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

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

#### Axiom Instance

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



### Mutation (backwards)

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

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

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

#### Axiom Instance

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



### 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

18.11.08 2016-07-07

1 [18]



## 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**

Korrekte Software

2 [18]



## Was gibt's heute?

- ▶ Rückblick
- ▶ Ausblick
- ▶ Feedback

Korrekte Software

3 [18]



## Rückblick

Korrekte Software

4 [18]



## Semantik

- ▶ Operational — Auswertungsrelation  $\langle c, \sigma \rangle \rightarrow \sigma'$
- ▶ Denotational — Partielle Funktion  $\llbracket c \rrbracket : \Sigma \rightarrow \Sigma$
- ▶ Axiomatisch — Floyd-Hoare-Logik
- ▶ Äquivalenz von operationaler und denotationaler Semantik
- ▶ Welche Semantik wofür?

Korrekte Software

5 [18]



## 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

Korrekte Software

6 [18]



## 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  $\Sigma \rightarrow \Sigma \times \mathbf{V}_U$
  - ▶ Spezifikation von Funktionen durch Vor-/Nachbedingungen
  - ▶ Spezifikation der Funktionen muss im Kontext stehen

Korrekte Software

7 [18]



## Erweiterung der Programmiersprache

- ▶ Zeiger und Referenzen
  - ▶ Lokationen nicht mehr symbolisch (Variablenamen), sondern abstrakt  $\Sigma = \mathbf{Loc} \rightarrow \mathbf{Val}, \mathbf{Val} = \mathbf{N} + \mathbf{C} + \mathbf{Loc}$
  - ▶ 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

Korrekte Software

8 [18]



# Ausblick



## Die Sprache C: Was haben wir ausgelassen?

### Semantik:

- ▶ Nichtdeterministische Semantik: Seiteneffekte, Sequence Points  
→ Umständlich zu modellieren, Effekt zweitrangig
- ▶ Implementationsabhängiges, un spezifiziertes und undefiniertes Verhalten  
→ Genauere Unterscheidung in der Semantik

### Kontrollstrukturen:

- ▶ **switch** → Ist im allgemeinen Fall ein **goto**
- ▶ **goto**, setjmp/longjmp  
→ Tiefe Änderung der Semantik (*continuations*)



## Die Sprache C: Was haben wir ausgelassen?

### Typen:

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



## 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!

