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

Serge Autexier, Christoph Lüth

Universität Bremen

Sommersemester 2017

Organisatorisches

- Veranstalter:

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

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

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

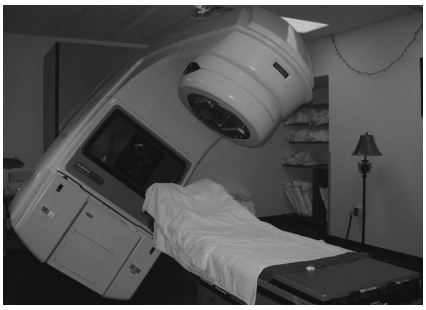
`http://www.informatik.uni-bremen.de/~cxl/lehre/ksgm.ss17`

Prüfungsformen

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

Warum Korrekte Software?

Software-Disaster I: Therac-25



Bekannte Software-Disaster II: Ariane-5

Bekannte Software-Disaster III: Airbus A400M

Inhalt der Vorlesung

Themen

Korrekte Software im Lehrbuch:

- Spielzeugsprache
- Wenig Konstrukte
- Kleine Beispiele

Korrekte Software im Einsatz:

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

Inhalt

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

Einige Worte zu Scala

Scala

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

Scala am Beispiel: `01-GCD.scala`

Was sehen wir hier?

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

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

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

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

Was sehen wir hier?

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

  require(d ≠ 0)

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

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

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

  override def toString = numer +"/"+
    denom

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

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

Algebraische Datentypen: `03-Expr.scala`

Was sehen wir hier?

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

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

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

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

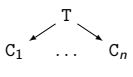
Implementierung algebraischer Datentypen

Haskell:

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

- Ein Typ `T`
- Konstruktoren erzeugen Datentyp

Scala:



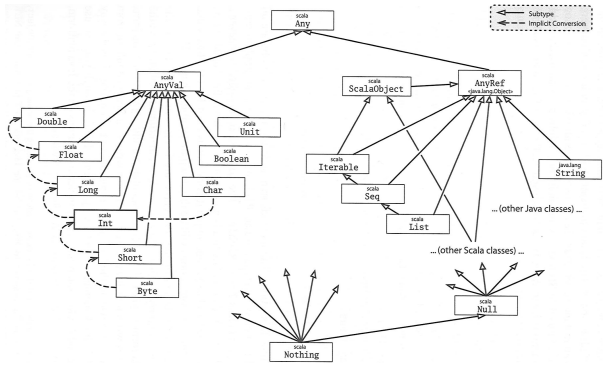
- Varianten als Subtypen
- Problem und Vorteil: Erweiterbarkeit
- sealed verhindert Erweiterung

Das Typsystem

Das Typsystem behebt mehrere Probleme von Java:

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

Vererbungshierarchie



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

Parametrische Polymorphie

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

Typvarianz

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

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

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

Beispiel:

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


Traits: `05-Funny.scala`

Was sehen wir hier?

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

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

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

Komprehension mit for : `06-For.scala`

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

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

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

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


Was wir ausgelassen haben...

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

Scala Zusammenfassung

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

Zusammenfassung

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