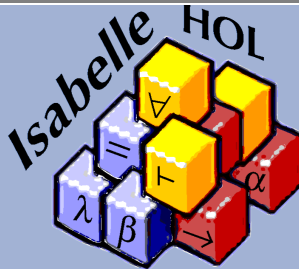
Karlsruhe Institute of Technology

Verifikation mit Isabelle

Joachim Breitner
BobKonf 2016
19. Februar 2016, Berlin

LEHRSTUHL PROGRAMMIERPARADIGMEN



Was ist Isabelle?

Karlsruhe Institute of Technology

Isabelle ist. . .

- ein interaktiver Theorembeweiser.

Was ist Isabelle?

Karlsruhe Institute of Technology

Isabelle ist. . .

- ein interaktiver Theorembeweiser.
- eine funktionale Programmiersprache
  mit algebraischen Datentypen, rekursiven Funktionen, Typklassen. . .

Was ist Isabelle?

Karlsruhe Institute of Technology

Isabelle ist. . .

- ein interaktiver Theorembeweiser.
- eine funktionale Programmiersprache
  mit algebraischen Datentypen, rekursiven Funktionen, Typklassen. . .
  aus der man Code in Haskell, Scala, Standard ML, OCaml generieren kann.

Was ist Isabelle?

KIT Karlsruhe Institute of Technology

Isabelle ist. . .

- ein interaktiver Theorembeweiser.
- eine funktionale Programmiersprache
  mit algebraischen Datentypen, rekursiven Funktionen, Typklassen. . .
  aus der man Code in Haskell, Scala, Standard ML, OCaml generieren kann.
- eine Entwicklungsumgebung für Programme *und* Beweise
  basierend auf jEdit.

Programmverifikation mit Isabelle – mehrere Ansätze

Karlsruhe Institute of Technology

- Haskell (o.ä.) → Isabelle

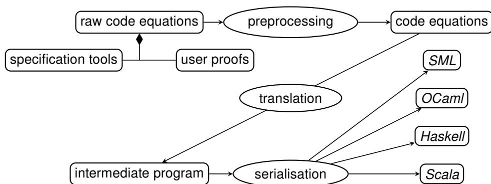
  . . . als *shallow embedding*:
  Haskell-Funktionen werden Isabelle-Funktionen (Haskabelle)

  . . . als *deep embedding*:
  Haskell-Syntax wird über einen Isabelle-Datentyp dargestellt.

  Beispiel: sel4

- Isabelle → Haskell (o.ä.) (Code-Generator)

  Beispiel: CoCon



Unser Plan für heute

- Einen einfachen Datentyp definieren (die üblichen Listen)
- Einfache Funktionen darüber implementieren (reverse)
- Daraus Haskell-Code generieren
- Etwas über unseren Code beweisen (reverse (reverse xs) == xs)
- Eine komplexere Datenstruktur implementieren (queues)
- Mit *program refinement* die Implementierung verbessern
- Sofern noch Zeit ist: Diese Datenstruktur verifzieren

Disclaimer

Karlsruhe Institute of Technology

Wir werden nur an der Oberfläche kratzen.

- Funktionale Programmierung wird weitgehend vorausgesetzt
- Wir werden nur eine Beweismethode nutzen,
  und die Aufgaben sind so ausgelegt, dass das geht
- Wir werden spezielle Klippen elegant umschiffen
  (z.B. nicht offensichtlich terminierende Programme)

Getting started

Karlsruhe Institute of Technology

Was bereits passiert sein sollte:

- Isabelle herunterladen (http://isabelle.in.tum.de/)
- Isabelle einmal starten (der erste Start dauert länger)

Was wir jetzt machen:

- Neue Datei `Queue.thy` anlegen.
- Datei-Header:

```
theory Queue
imports Main
begin
```

Funktionales Programmieren in Isabelle

Karlsruhe Institute of Technology

Wir setzen folgendes Haskell-Programm in Isabelle um:

```
data List a = N | C a (List a)

app :: List a -> List a -> List a
app N ys = ys
app (C x xs) ys = C x (app xs ys)

reverse :: List a -> List a
reverse N = N
reverse (C x xs) = app (reverse xs) (C x N)
```

Siehe das Syntax-Cheat-Sheet!

Code-Export mit Isabelle

Karlsruhe Institute of Technology

Um Haskell-Code zu erhalten, schreiben wir

export_code reverse **in** Haskell **file** "out/"

ans Ende der Datei.

Beachte:
Die Listen-Definition und app wurden automatisch auch exportiert.

Im Verzeichnis `out/` sollte dann eine Datei `Queue.hs` liegen,
die man z.B. mit `ghci` laden kann.

Beweise!

Beweismethode

Induktion + Simplifikation

Beweise!

Karlsruhe Institute of Technology

Beweismethode

Induktion + Simplifikation

Struktur (in unseren Fällen)

lemma *name*[simp]: "*ausdruck1 = ausdruck2*"
  apply (induction xs)
  apply auto
  done

Beweise!

Karlsruhe Institute of Technology

Beweismethode

Induktion + Simplifikation

Struktur (in unseren Fällen)

```
lemma name[simp]: "ausdruck1 = ausdruck2"
  apply (induction xs)
  apply auto
  done
```

Was wir wollen wir zeigen?

```
lemma reverse_reverse[simp]: "reverse (reverse xs) = xs"
```

Dies wird drei weitere Hilfslemmas benötigen! Welche?

Amortized Queue

Wir ergänzen folgenden Code:

```
data AQueue a = AQueue (List a) (List a)

emptyQ :: AQueue a
emptyQ = AQueue N N

enqueue :: a -> AQueue a -> AQueue a
enqueue x (AQueue xs ys) = AQueue (C x xs) ys

dequeue :: AQueue a -> (Maybe a, AQueue a)
dequeue (AQueue N N) = (Nothing, AQueue N N)
dequeue (AQueue xs (C y ys)) = (Just y, AQueue xs ys)
dequeue (AQueue xs N) =
        case reverse xs of C y ys -> (Just y, AQueue N ys)
```

Der Code ist zu lahm

Karlsruhe Institute of Technology

Unsere Definition von reverse ist zwar verifikations-freundlich, aber nicht performant ($O(n^2)$).

Der Code ist zu lahm

Karlsruhe Institute of Technology

Unsere Definition von reverse ist zwar verifikations-freundlich, aber nicht performant ($O(n^2)$).

Besser ist:

```
fast_rev :: List a −> List a −> List a
fast_rev N ys = ys
fast_rev (C x xs) ys = fast_rev xs (C x ys)
```

Dies könnte man wie folgt nutzen:

```
reverse :: List a −> List a
reverse xs = fast_rev xs N
```

Schnellerer Code

Karlsruhe Institute of Technology

Aber wir wollen unsere Beweise zu reverse nicht ändern,
sondern nur die Implementierung von reverse in dequeue!

Schnellerer Code

Karlsruhe Institute of Technology

Aber wir wollen unsere Beweise zu reverse nicht ändern,
sondern nur die Implementierung von reverse in dequeue!

Lösung: Code refinement

1. Wir beweisen, wie sich fast_rev verhält:

```
lemma fast_rev_reverse[simp]: "fast_rev xs ys = app (reverse xs) ys"
Beweis
```

(Achtung: Induktionsbeweis geht nicht ohne weiteres durch.)

Schnellerer Code

Karlsruhe Institute of Technology

Aber wir wollen unsere Beweise zu reverse nicht ändern,
sondern nur die Implementierung von reverse in dequeue!

Lösung: Code refinement

1. Wir beweisen, wie sich fast_rev verhält:

   lemma fast_rev_reverse[simp]: "fast_rev xs ys = app (reverse xs) ys"
   Beweis

   (Achtung: Induktionsbeweis geht nicht ohne weiteres durch.)

2. Wir lassen den Code-Generator reverse durch fast_rev ersetzen:

   lemma [code_unfold]: "reverse xs = fast_rev xs N"
   Beweis

Wie beeinflusst das den generierten Code?

Queue verifizieren

Zuletzt möchten wir noch die Queue selbst verifizieren.

1. Wir suchen wir einen äquivalenten, aber einfacheren Datentypen (Performance ist egal!):

Queue verifizieren

Karlsruhe Institute of Technology

Zuletzt möchten wir noch die Queue selbst verifizieren.

1. Wir suchen wir einen äquivalenten, aber einfacheren Datentypen (Performance ist egal!): **Listen!**

Queue verifizieren

Karlsruhe Institute of Technology

Zuletzt möchten wir noch die Queue selbst verifizieren.

1. Wir suchen wir einen äquivalenten, aber einfacheren Datentypen (Performance ist egal!): **Listen!**
2. Wir implementieren emptyQ’, enqueue’ und dequeue’ auf ’a List, so dass sie “offensichtlich korrekt” sind.

Queue verifizieren

Karlsruhe Institute of Technology

Zuletzt möchten wir noch die Queue selbst verifizieren.

1. Wir suchen wir einen äquivalenten, aber einfacheren Datentypen (Performance ist egal!): **Listen!**
2. Wir implementieren emptyQ', enqueue' und dequeue' auf 'a List, so dass sie "offensichtlich korrekt" sind.
3. Wir beschreiben eine Queue als Liste:

```
fun list_of :: "'a queue ⇒ 'a List" where ...
```

Queue verifizieren

Karlsruhe Institute of Technology

Zuletzt möchten wir noch die Queue selbst verifizieren.

1. Wir suchen wir einen äquivalenten, aber einfacheren Datentypen (Performance ist egal!): **Listen!**
2. Wir implementieren emptyQ', enqueue' und dequeue' auf 'a List, so dass sie "offensichtlich korrekt" sind.
3. Wir beschreiben eine Queue als Liste:

```
fun list_of :: "'a queue ⇒ 'a List" where ...
```

4. Wir zeigen: Die alten Methoden machen „das selbe“ wie die neuen. Die Beweise über queue werden ggf. komplizierter:

```
apply (induction q rule: dequeue.induct)
apply (auto split: List.split)
done
```

Tipp: Man braucht ein Hilfslemma der Form (app xs ys = N) = (. . .).

Wie geht es weiter?

Karlsruhe Institute of Technology

- Isabelle lernen, z.B. mit
  Nipkow, Klein: *Concrete Semantics*
- Auf stackoverflow.com mit Tag `isabelle` bekommt man Hilfe.
- Den Code-Generator kennen lernen lernen, z.B. mit
  Haftmann, Bulwahn: *Code generation from Isabelle/HOL theories*
- Mehr auf http://isabelle.in.tum.de/documentation.html

Thank you for your attention.

Code examples inspired by, and code generation diagram taken from:
Florian Haftmann, Lukas Bulwahn, *Code generation from Isabelle/HOL theories*.
Tobias Nipkow, *Programming and Proving in Isabelle/HOL*.