Verifikation mit Isabelle

(Handout)

Joachim Breitner*

19. Februar 2016
BobKonf 2016

Haskell-Code-Vorlage

```Haskell
module Queue where

import Prelude hiding (reverse)

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)

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)

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)
```

*breitner@kit.edu, http://www.joachim-breitner.de/

Isabelle-Syntax-Cheat-Sheet

Theorie-Kopf

```
theory Theoriename
imports Main Weitere Theorien
begin
...
end
```

Typen

Typvariablen beginnen mit ' (Apostroph), Typkonstruktoren werden post-fix geschrieben.

Auswahl an Typen:

- ('a × 'b) für Paare (x,y) (Eingabe: `\ti`)
  Selektoren fst und snd
- 'a ⇒ 'b für Funktionen (Eingabe: =>).
- 'a option, Konstruktoren: Some x und None

Terme

Funktionsanwendung wie in Haskell:
foo arg1 arg2 arg3.

Geklammert werden müssen:

- (if ... then ... else)
- (case ... of N ⇒ ... | C x xs ⇒ ...)

Algebraische Datentypen

```
datatype 'a 'b 'c typname
  = con1 "typ1a" "typ1b" ...
  | con2 "typ2a" "typ2b" ...
  ...
```

Funktionen

```
fun name :: "typ1 ⇒ typ3 ⇒ typ" where
  "name pat1a pat1b = rhs1"
 |"name pat2a pat2b = rhs2"
```

Patterns können wie üblich Variablen (xs) oder Konstrukturen (C x xs) sein.

Definitionen ohne Parameter gehen nicht mit **fun**, sondern mit

```
definition name :: "typ" where
  [simp]:"name = rhs"
```

Code-Export

```
export_code funktion1 funktion2 ...
  in Sprache file "Pfad"
```

Sprache: Haskell, SML, OCaml oder Scala.
Pfad: ein Verzeichnis bei Haskell, ein Dateiname sonst.

Lemmata

```
lemma Lemmaname[simp]: "Aussage"
  Beweis
```

Das [simp] ist optional, und sorgt dafür dass der Simplifier (z.B. auto, siehe unten) dieses Lemma, wenn es eine Gleichung ist, stets von links nach rechts anwendet.
Achtung: Kann zu Endlosschleifen führen.

Mit [code_unfold] wird die Gleichung bei der Codegenerierung angewandt.

Beweise

Grundform ohne Induktion:

```
apply auto
done
```

Grundform mit Induktion:

```
apply (induction xs)
apply auto
done
```

Optionen für auto:

- split: typ.split
  Wenn das Ziel eine Fallunterscheidung (case ... of) über Werte vom Typ typ enthält, zerlege das Ziel entsprechend.

Optionen für induction:

- arbitrary: ys
  Die Variable ys ist im induktiven Schritt beliebig.
- rule: foo.induct
  Verwende eine Induktionsvariante, die der Definition der Funktion foo entspricht.

Einen unfertigen Beweis erstmal ignorieren:

```
sorry
```