Hochschule RheinMain

Funktionale Programmierung

10 Lean

Prof. Dr. David Sabel
Sommersemester 2025

Stand der Folien: 18. Juni 2025

Was ist Lean?

- Funktionale Programmiersprache und Theorembeweiser
- Strikte funktionale Programmiersprache
- Abhängige Typen (dependent types)
- Theorie basiert auf Thierry Coquands Calculus of Constructions
- Mathematische Aussagen beweisen (indem man Programme schreibt!)
  Die Idee dabei ist der Curry-Howard-Isomorphismus, ungefähr:
  - Typ T = logische Aussagen
  - Programm vom Typ T = logische Aussage ist korrekt

  Existenz des Programms = Beweis der Aussage
- Man kann damit auch Aussagen über Programme beweisen, d.h. Verifikation von Programmen ist möglich

Einige Ressourcen zu Lean

- Webseite: `https://lean-lang.org/`
- The Lean Language Reference
  `https://lean-lang.org/doc/reference/latest`
- David Christiansen: Functional Programming in Lean.
  `https://lean-lang.org/functional_programming_in_lean`
- Burkhardt Renz: Funktionale Programmierung in Lean.
  Vorlesungsskript, TH Mittelhessen
  `https://esb-dev.github.io/fpl.html`

Lean und Haskell

Konzepte in Lean, die sehr ähnlich zu Haskell sind

- Syntax zum Teil ähnlich zu Haskell
- IO: Wie in Haskell gibt es eine `IO`-Monade, die Seiteneffekt-behaftete Funktionen trennt von reinen Funktionen
- Typklassen: Zur Überladung von Funktionen verwendet Lean Typklassen, die ähnlich zu Haskell's Typklassensystem sind

Historie von Lean

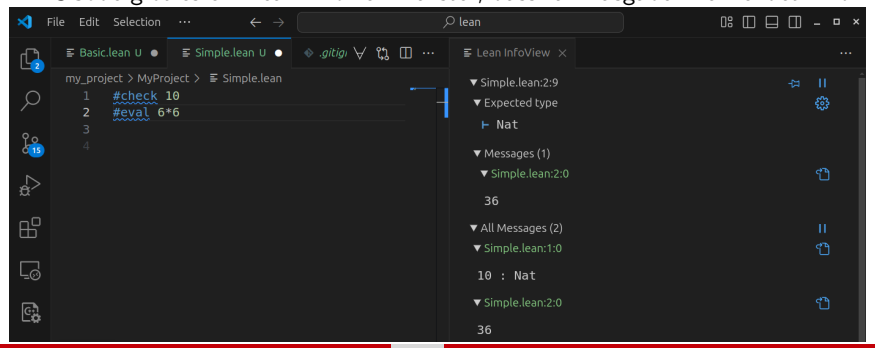
Hochschule RheinMain

- 2013 Leonardo de Moura (Microsoft Research) veröffentlicht Version 0.1
- 2017: Lean 3, von vielen Mathematikern verwendet
- Community programmiert Mathlib, die am Ende von Lean 3 aus über 1 Million Zeilen formalisierter Mathematik bestand.
- 2023: Lean 4 veröffentlicht. Wesentlich: Großteil von Lean 4 ist selbst in Lean 4 implementiert
- Mathlib in Lean 4: 1,5 Millionen Zeilen Code.

Installation

Hochschule RheinMain

- `https://leanprover-community.github.io/get_started.html`
- Offizielle Dokumentation:
  `https://docs.lean-lang.org/lean4/doc/quickstart.html`
- In VSCode gibt es ein Lean Infoview-Fenster, dass für Ausgaben verwendet wird:



EINFÜHRENDES ZU LEAN

Hochschule RheinMain

Auswerten

Hochschule RheinMain

Ein Ausdruck `Expr` kann durch die Quelltextanweisung

```
#eval Expr
```

ausgewertet werden.

Beispiel:

```
#eval 6*6
```

Simple.lean

```
36
```

Lean InfoView

Lean verwendet die **strikte** Auswertung:

→ Auwertung einer Funktionsanwendung $f\ a_1\ \ldots\ a_n$:

- Auswerten von a_1 bis a_n zu $f\ v_1\ \ldots\ v_n$
- Einsetzen von der Werte $v_1, \ldots, v_n$ in den Rumpf der Definition von f

Typen

Hochschule RheinMain

Typ berechnen und anzeigen mit:

`#check Expr`

Beispiel:

```
#check 6 * 6
```
Simple.lean

```
6 * 6 : Nat
```
Lean InfoView

Typen explizit angeben mit

Expr : Type

Beispiel:

```
#check (6 * 6 : Int)
```
Simple.lean

```
6 * 6 : Int
```
Lean InfoView

Funktionstypen: Funktionspfeil -> oder Unicode-Symbol

Funktionen definieren

Hochschule RheinMain

Schlüsselwort `def`, Syntax für Funktionsdefinitionen:

`def` *funktionsName Argument*$_1$ ... *Argument*$_n$:= *Rumpf*

Beispiel:

```
def verdopple n := 2*n
#check verdoppele
```
Simple.lean

```
verdopple (n : Nat) : Nat
```
Lean InfoView

Funktionsdefinition mit Typen

Hochschule RheinMain

Beispiel:

```
def verdopple' n := n+n
```
Simple.lean

```
typeclass instance problem is stuck,
  HAdd ?m.453 ?m.453 (?m.442 n)
```
Lean InfoView

Problem: Lean weiß nicht, welchen Typ er für `n` verwenden soll, da + überladen ist.

Aber:

```
def verdopple'' n := 0+n+n
#check verdopple''
```
Simple.lean

```
verdopple'' (n : Nat) : Nat
```
Lean InfoView

Funktioniert, da Lean für 0 den Typ `Nat` annimmt und damit den passenden Typ für + herleiten kann.

Funktionsdefinition mit Angabe der Typen

Hochschule RheinMain

Syntax:

`def` *funktionsName* `(`*Argument*$_1$ `:` *Typ*$_1$`)` ... `(`*Argument*$_n$ `:` *Typ*$_n$`)` `:` *Ergebnistyp* `:=` *Rumpf*

Beispiel:

```
def verdopple' (n : Nat) : Nat := n+n
#check verdopple'
```
Simple.lean

```
verdopple' (n : Nat) : Nat
```
Lean InfoView

Alternativ mit Abstraktion (in Lean geschrieben als `fun x => s` oder `λx => s`:

Beispiel

```
def verdopple''' : Nat -> Nat := fun n => n+n
#check verdopple'''
```
Simple.lean

```
verdopple''' : Nat -> Nat
```
Lean InfoView

Funktionen in Lean: Terminierung

Hochschule RheinMain

Funktionen in Lean müssen **total** und **terminierend** sein.

Grund: Konsistenz der Beweise erfordert dies

Beispiel

```
def nichtterminierend (n : Nat) : Nat :=
   nichtterminierend n
```

Simple.lean

```
fail to show termination for
  nichtterminierend
with errors
...
```

Lean InfoView

Die Terminierung kann auch händisch nachgewiesen werden (nicht für obige Funktion)

Andere Abhilfe: Schlüsselwort `partial`

```
partial def nichtterminierend (n : Nat) : Nat :=
   nichtterminierend n
```

Simple.lean

Dependent Types

Hochschule RheinMain

Abhängige Typen = Typen hängen von **Werten** ab,

Beispiel:

```
def jenachdem (w:Bool) : if w then Nat else Bool :=
 match w with
   | true => (0:Nat)
   | false => false
```

Simple.lean

- `if-then-else` auf Typebene!
- Funktion `jenachdem` liefert anderen Typ abhängig vom **Wert** der Eingabe

Namespaces

Hochschule RheinMain

- Namespaces werden zur Strukturierung verwendet
- Z.B. können gleichnamige Funktionen in unterschiedlichen Namespaces verwendet werden. (z.B. `map` für Listen und für Arrays
- Namespaces `List` und `Array`: Zugriff auf `map` über `List.map` und `Array.map`
- Mit open *Namensbereich* kann man den Namespace öffnen und damit direkt auf `map` zugreifen

Beispiele:

```
#eval List.map (fun x => x + x) [1,2,3]
#eval map (fun x => x + x) [1,2,3]
open List
#eval map (fun x => x + x) [1,2,3]
```

Simple.lean

```
[2,4,6]
unknown identifier 'map'

[2,4,6]
```

Lean InfoView

Namespaces: Punkt-Notation

Hochschule RheinMain

Statt *Namensbereich.f Argument* darf man *Argument.f* schreiben,

wenn der Typ von *Argument* im *Namensbereich* definiert wurde.

Beispiel:

```
#eval [1,2,3].map (fun x => x + x)
```

Simple.lean

```
[2,4,6]
```

Lean InfoView

Funktionen dem Namespace hinzufügen:

```
def Nat.verdoppele (n: Nat) : Nat := n+n
#eval (5:Nat).verdoppele
```

Simple.lean

```
10
```

Lean InfoView

Hochschule RheinMain

EINGEBAUTE UND VORDEFINIERTE TYPEN

Zahlen

Hochschule RheinMain

`Nat`	Natürliche Zahlen mit 0
`Int`	Ganzzahlen
`Fin n`	Natürliche Zahlen kleiner `n`
`ISize`	Ganzzahlen beschränkter Länge (32 oder 64 Bit)
`USize`	Natürliche Zahlen beschränkter Länge (32 oder 64 Bit)
`BitVec`	Bit-Vektoren beschränkter Länge
`Float32`	Fließkommazahlen 32 Bit
`Float`	Fließkommazahlen 64 Bit

Zeichen und Zeichenketten

Hochschule RheinMain

`Char` Zeichen, in einfache Anführungsstriche

`String` Zeichenketten, in doppelte Anführungsstriche

Strings verhalten sich wie Listen, aber intern wird eine andere Darstellung benutzt.

Typen mit einem und keinem Wert

Hochschule RheinMain

`Unit`	Typ, der nur das Nulltupel `()` enhält
`Empty`	Typ, der keine Werte enthält

Wahrheitswerte

Hochschule RheinMain

`Bool` Wahrheitswerte `false` und `true`

Junktoren:

- `&&` Logisches Und
- `||` Logisches Oder
- `^^` Exklusives Oder
- `!` Negation (präfix)

Optionale Werte

Hochschule RheinMain

`Option` Konstruktoren sind `none` und `some`

analog zum `Maybe`-Typ in Haskell

Produkttypen: Paare und Tupel

Hochschule RheinMain

- Paare mit runden Klammern, Selektoren sind `fst` und `snd`
- Typ ist $(\alpha_1 \times \alpha_2)$ oder alternativ `Prod` α_1 α_2.
- n-stellige Tupel: `(e1,...,en,)` geschrieben, der Typ ist $(\alpha_1 \times \alpha_2 \times \ldots \times \alpha_n)$.
  (interne Darstellung: geschachtelte Paare, rechtsgeklammert)

Beispiel:

```
#eval Prod.fst (1,2,3)
#eval Prod.snd (1,2,3)
```

Simple.lean

```
1
(2,3)
```

Lean InfoView

Summentypen

Hochschule RheinMain

- Summentypen vereinigen Typen
  (Haskell: `Either a b` mit Konstruktoren `Left` und `Right`)
- Lean: Typ $\alpha \oplus \beta$ mit Konstruktoren `inl` und `inr`

Listen

Hochschule RheinMain

- Homogene Listen als Typ `List a` vorhanden für Listen mit Inhalt vom Typ `a`
- Konstruktoren sind `nil` und `cons` , die aber auch auch als `[]` für `nil` und `x::xs` für `(cons xs xs)` geschrieben werden.
- Auch `[1,2,3,4]` als Syntax erlaubt

Definition der Listen als induktiver Datentyp:

```
inductive List (a:Type) where
  | nil: List a
  | cons: a -> List a -> List a
```

Semantik ist induktiv: D.h. unendliche Listen sind nicht erlaubt!
Definition eines induktiven Datentyps erzeugt Namespace:

- Namespace ist der Typname
- Konstruktoren werden darunter definiert.

Übliche Listenfunktionen sind vordefiniert.

Beispiele

Hochschule RheinMain

```
def mylist1 : List Nat := [1,2,3,4,5,6]
def mylist2 : List Nat := List.replicate 5 3
def mylist3 : List (List Nat) := [mylist1,mylist1,mylist2]
```

```
#eval mylist1
#eval List.length mylist1
#eval List.take 4 mylist1
#eval mylist1.take 4
#eval List.drop 4 mylist1
#eval mylist1.get (2: Fin 6)
#eval List.append mylist1 mylist2
#eval List.concat mylist1 50
#eval mylist1.reverse
```

Simple.lean

```
[1,2,3,4,5,6]
6
[1,2,3,4]
[1,2,3,4]
[5,6]
3
[1,2,3,4,5,6,3,3,3,3,3]
[1,2,3,4,5,6,50]
[6,5,4,3,2,1]
```

Lean InfoView

Beispiele (2)

Hochschule RheinMain

```
def mylist1 : List Nat := [1,2,3,4,5,6]
def mylist2 : List Nat := List.replicate 5 3
def mylist3 : List (List Nat) := [mylist1,mylist1,mylist2]
```

```
#eval mylist1.foldl (fun x y => x + y) 0
#eval mylist1.foldr (fun x y => x + y) 0
#eval mylist1.map (fun x => 2*x)
#eval mylist2.flatMap (fun x => [2*x])
#eval mylist1.filter (fun x => x > 5)
#eval mylist3.flatten
```

Simple.lean

```
21
21
[2,4,6,8,10,12]
[6,6,6,6,6]
[6]
[1,2,3,4,5,6,1,2,3,
 4,5,6,3,3,3,3,3]
```

Lean InfoView

Arrays

Hochschule RheinMain

- Typ `Array a` mit `a` Typ der Elemente
- `#[ e1, e2, ... en ]` erzeugt Array mit Elementen `e1`, `e2`, ..., `en`
- `ar[`i`]`: Zugriff auf das $i+1$-te Element von `ar`
- `ar[`i`:`j`]`: Zugriff auf das Unterarray von Index i bis j
  (`Subarray a` ist ein eigener Typ)

Subtypen

Hochschule RheinMain

Subtypen schränken einen bestehenden Typen mit einem Prädikat ein.

Syntax: `{ n : a // p n }`.

Der Subtyp enthält nur die Objekte `n` vom Typ a, die p erfüllen

Beispiel: Vorgängerfunktion für natürliche Zahlen ohne 0:

```
def pred (x : {n : Nat // n > 0 }) : Nat := x - 1
```

Elemente eines Subtypen: Paare $\langle e, b \rangle$ mit e Element des Typs und b der Beweis für $p\ e$

```
def ten : { n : Nat // n > 0 } := ⟨10, by simp⟩
```

`by simp`: eingebauter Simplifier, anschließend funktioniert

```
#check pred ten
#eval pred ten
```

Implizit: Konvertierung zum Basistyp, z.B. wenn `x - 1` in `pred` ausgewertet wird

Verzögerte Auswertung

Hochschule RheinMain

- Verzögerte Auswertung in strikten Sprachen ist möglich durch Verpacken von Ausdrücken als Abstraktionen:
  - anstelle von e verwende $\lambda x.e$
  - Auswertung von e erzwingen: Anwenden auf `()`
- Lean bietet angenehmere Methode:
  - Typ `Thunk a`
  - `Thunk.mk e` erzeugt die Abstraktion `fun (x:Unit) => e`
  - `Thunk.get` zum Zugriff auf `e`

Verzögerte Auswertung: Beispiel

Hochschule RheinMain

```
#eval Thunk.get ((factorial 100) : Thunk Nat)

def leftIfGreaterZero (x:Nat) (y:Nat) : Nat :=
   if x > 0 then x else y

def lazyLeftIfGreaterZero (x:Nat) (y:Thunk Nat) : Nat :=
   if x > 0 then x else y.get

-- terminiert sofort
#eval lazyLeftIfGreaterZero 10 ((factorial 200000):Thunk Nat)

-- stack overflow;
#eval leftIfGreaterZero 10 (factorial 200000)
```


DEPENDENT TYPES

Hochschule RheinMain

Dependent Types

Hochschule RheinMain

- Nochmal: Abhängige Typen = Typen hängen von Werten ab
- Standardbeispiel: Vektoren, die ihre Länge im Typ kodieren
- Lean-Tutorials: Vektoren als Subtyp von Listen, der zusätzlich die Länge speichert
- Wir: Eigener Typ, d.h. direkte Definition

Vektoren

Hochschule RheinMain

```
inductive Vect (a : Type) : Nat -> Type where
  | nil : Vect a 0
  | cons : a -> Vect a n -> Vect a (n + 1)
```

Vect.lean

Zahl im Typ kodiert die Länge des Vektors
Vektor-Addition:

```
def Vect.add (x : Vect Nat n) (y: Vect Nat n) : Vect Nat n :=
  match x with
  | Vect.nil => Vect.nil
  | Vect.cons a as =>
       match y with
         | Vect.cons b bs => Vect.cons (a + b) (Vect.add as bs)
```

Vect.lean

Vektor-Addition

Hochschule RheinMain

```
def Vect.add (x : Vect Nat n) (y: Vect Nat n) : Vect Nat n
...
```

sichert zu:

- Nur Vektoren gleicher Länge können addiert werden
- Addition von Vektoren ungleicher Länge gibt einen Typfehler
- D.h. Fehler zur Compilezeit!

```
#check (Vect.cons 1
          Vect.nil)
       .add (Vect.cons 1
              (Vect.cons 2
                 Vect.nil))
```

Vect.lean

```
application type mismatch
  Vect.cons 1 (Vect.cons 2 Vect.nil)
argument
  Vect.cons 2 Vect.nil
has type
  Vect Nat (0 + 1) : Type
but is expected to have type
  Vect Nat 0 : Type
```

Lean InfoView

Head und Tail für nicht-leere Vektoren

Hochschule RheinMain

```
def Vect.head (x : Vect a (n+1)) : a :=
   match x with
   | Vect.cons a _ => a

def Vect.tail (x : Vect a (n+1)) : Vect a n :=
   match x with
   | Vect.cons _ as => as
```

Vect.lean

- Fälle für `nil` nicht möglich da $n + 1 > 0$
- Awendung auf leere Vektoren (Typ `Vect a 0`) ungetypt

Beispielaufrufe

Hochschule RheinMain

```
def emptyVect : Vect Nat 0 := Vect.nil
def myVect1 := Vect.cons 100 (Vect.cons 200 (Vect.cons 300 Vect.nil))
```

```
#eval myVect1.head

#eval myVect1.tail

#eval emptyVect.head
```

Vect.lean

```
100

Vect.cons 200 (Vect.cons 300 (Vect.nil))

application type mismatch
  emptyVect.head
argument
  emptyVect
has type
  Vect Nat 0 : Type
but is expected to have type
  Vect ?m.2992 (?m.2993 + 1) : Type
```

Lean InfoView

Replicate: Vector der Länge n erstellen

Hochschule RheinMain

- Länge des Vektors wird übergeben
- Man sieht: Typ hängt von Eingabe ab (= abhängiger Typ)

```
def Vect.replicate (n:Nat) (x:a) : (Vect a n) :=
  match n with
    | 0 => Vect.nil
    | Nat.succ m =>  Vect.cons x (Vect.replicate m x)
```


Append für Vektoren

Hochschule RheinMain

```
def Vect.append (xs:Vect a m) (ys:Vect a n) : (Vect a (n+m)) :=
 match xs with
   | Vect.nil => ys
   | Vect.cons z zs => Vect.cons z (zs.append ys)
```

- Zeigt wie sich die Vektorlängen addieren
- Vektorlängen zur Compilezeit voraussagbar
- Beachte: Mit `Vect a (m+n)` als Ergebnistyp kann Lean die Funktion so nicht typisieren, da der Typchecker die Gleichheit $m + n = n + m$ nicht selbst herleiten kann

Append für Vektoren (2)

Hochschule RheinMain

Folgendes funktioniert

```
def Vect.append (xs:Vect a m) (ys:Vect a n) : (Vect a (n+m)) ...

def Vect.append2 (xs:Vect a m) (ys:Vect a n) : (Vect a (m+n)) :=
 Nat.add_comm m n  ▶  xs.append ys
```

Dabei wird dem Lean-System gesagt, dass er ein bewiesenes Gesetz (Kommutativität der Addition für natürliche Zahlen) beim Typcheck verwenden soll.
D.h. hier werden Logik und Programmierung vermischt.

Reverse für Vektoren

Hochschule RheinMain

```
def Vect.reverse (xs:Vect a n) : (Vect a n) :=
 match n, xs with
   | 0, Vect.nil => Vect.nil
   | u+1, Vect.cons y ys =>
      Nat.add_comm 1 u ▶ ys.append (Vect.cons y Vect.nil)
```

- Auch hier muss man dem Typchecker helfen
- Hier werden mehrere Terme auf einmal gematched

Typsicherer Zugriff auf ein Vektorelement

Hochschule RheinMain

```
def Vect.get (x: Vect Nat n) (i:Fin n) : Nat :=
 match i, x with
  | ⟨0,_⟩, Vect.cons y ys => y
  | ⟨j+1,h⟩, Vect.cons _ ys => ys.get ⟨j,Nat.le_of_succ_le_succ h⟩
```

- `Fin n` sichert zu, dass $i \in \{0, \ldots, n-1\}$
- Der Beweis, dass $j = i - 1$ immer noch in `Fin n`, wird hier mithilfe des Beweises h, dass $i = j + 1$ auch `Fin n` stammt, erbracht

Typsicherer Zugriff auf ein Vektorelement (2)

Hochschule RheinMain

Beispiele

```
def myVect2 := Vect.replicate 5 2
def four: Fin 5 := ⟨ 4, by simp ⟩

#check myVect1
#check myVect2
#eval myVect2.get four
#check myVect1.get four
```

Vect.lean

```
myVect1 : Vect Nat (0 + 1 + 1 + 1)
Vect Nat 5
2
application type mismatch
  myVect1.get four
argument
  four
has type
  Fin 5 : Type
but is expected to have type
  Fin (0 + 1 + 1 + 1) : Type
```

Lean InfoView

VERIFIKATION MIT LEAN

Hochschule RheinMain

Verifikation mit Lean

Hochschule RheinMain

- Lean ist auch ein interaktiver Theorembeweiser
- Mathlib: Versuch große Teile der Mathematik zu formalisieren
- Auch Verifikation von Programmen möglich
- Wir betrachten einige Beispiele

Interaktives Theorembeweisen in Lean

Hochschule RheinMain

- Man möchte eine Aussage zeigen
- Lean unterstützt beim der Schrittweisen Beweis
- Man wendet in jedem Schritt sogenannte Beweistaktiken an
- Übersicht über Taktiken `https://www.ma.imperial.ac.uk/~buzzard/xena/formalising-mathematics-2024/Part_C/Part_C.html`
- Es gibt auch automatische Taktiken in Lean

Gesetze auf Listen

Hochschule RheinMain

- Gleichungen für Listenverarbeitende Funktionen können in Lean mit struktureller Induktion bewiesen werden
- Beachte: Das geht so nur, da die Listen endlich sind (in Haskell bräuchte man andere Methoden)
- Wir betrachten zwei Gesetze auf Listen
- Wir benutzen einen eigenen Listen-Typ: Dann können wir keinen Theoreme aus der List-Bibliothek verwenden

Eigener Listentyp

Hochschule RheinMain

```
import Mathlib

-- Eigener ListenTyp

inductive MyList (a:Type) where
  | myNil: MyList a
  | myCons: a -> MyList a -> MyList a

-- namespace MyList öffnen
open MyList
```

Verify.lean

Append und Reverse

Hochschule RheinMain

```
-- append-Funktion für MyList-Listen

def app (xs: MyList a) (ys: MyList a) : MyList a :=
 match xs with
  | myNil => ys
  | myCons a as => myCons a (app as ys)

-- reverse-Funktion für MyList-Listen

def rev (xs:MyList a) : MyList a :=
   match xs with
     | myNil => myNil
     | myCons a as => app (rev as) (myCons a myNil)
```

Verify.lean

Assoziativität von append

Hochschule RheinMain

Ziel: `app (app xs ys) zs = app xs (app ys zs)`
Formulierung als (benamtes) Theorem in Lean:

```
theorem app_associative:
  app (app xs ys) zs = app xs (app ys zs) := by
```

Verify.lean

Das aktuelle Beweisziel wird im InfoView gezeigt:

```
1 goal
α : Type
xs ys zs : MyList α
⊢ app (app xs ys) zs = app xs (app ys zs)
```

Lean InfoView

Assoziativität von append

Hochschule RheinMain

Ziel: `app (app xs ys) zs = app xs (app ys zs)`
Wir verwenden Induktion über `xs`:

```
theorem app_associative:
  app (app xs ys) zs = app xs (app ys zs) := by
  induction xs
```

Verify.lean

```
unsolved goals
  case myNil ...
  case myCons ...
```

Lean InfoView

Assoziativität von append

Hochschule RheinMain

Ziel: `app (app xs ys) zs = app xs (app ys zs)`
Wir verwenden Induktion über `xs`:

```
theorem app_associative:  app (app xs ys) zs = app xs (app ys zs) := by
  induction xs with
    | myNil          => rw [app]
                        rw [app]
    | myCons a as ih => rw [app]
                        rw [app]
                        rw [ih]
                        rw [app]
```

Verify.lean

- Taktik `rw[h]` wendet bereits bewiesene Gleichungen auf das Ziel an.
- `rw` stammt von „rewrite“
- `ih` ist die Induktionshypothese
- Beweis: `rw` um Definition von app und Induktionshypothese anzuwenden

Weiteres Gesetz

Hochschule RheinMain

- Ziel: `rev (rev xs) = xs`
- Direkter Beweis scheitert
- Hilfslemmas erforderlich

Hilfslemmas (1)

Hochschule RheinMain

```
lemma app_nil : app xs myNil = xs :=  by
  induction xs with
    | myNil          => rw[app]
    | myCons a as ih => rw[app]
                        rw[ih]
```

Verify.lean

Hilfslemmas (2)

Hochschule RheinMain

Ziel: `rev (rev xs) = xs`

```
lemma rev_append : rev (app xs ys) = app (rev ys) (rev xs)
   := by induction xs with
       | myNil          => rw[rev]
                           rw[app]
                           rw[app_nil]
       | myCons a as ih => rw[app]
                           rw[rev]
                           rw[ih]
                           rw[app_associative]
                           rw[rev]
```

Verify.lean

Hier verwenden wir unser Theorem zur Assoziativität von append und das vorherige Hilfslemma

Zweimal Reverse ändert nichts

Hochschule RheinMain

```
theorem rev_rev: rev (rev xs) = xs := by
  induction xs with
  | myNil          => rw[rev]
                      rw[rev]
  | myCons a as ih => rw[rev]
                      rw[rev_append]
                      rw[ih]
                      rw[rev]
                      rw[rev]
                      rw[app]
                      rw[app]
                      rw[app]
```

Fazit

- Mit Lean kann man funktional Programmieren
- Aber auch: Interaktiv Beweisen und Verifizieren
- Mehr Lean?
  → Veranstaltung Clever Coden mit Lean von Bodo Igler besuchen!