

# Computer-gestützte Beweisführung

## Klausur I

Heinrich-Heine-Universität Düsseldorf  
Wintersemester 2023/24

- Die Klausur dauert 90 Minuten. Sie besteht aus 8 Aufgaben. Pro Aufgabe sind 10 Punkte zu erwerben.
- Öffnen Sie die Klausur erst, wenn der Klausurbeginn angesagt wurde!
- Es sind keine Hilfsmittel (Taschenrechner, Mitschriften, Notizen, Skript, Handy, etc.) außer mechanischen Armbanduhren und Weckern zugelassen.
- Bitte prüfen Sie die folgenden Angaben und korrigieren Sie sie gegebenenfalls:

**Name:**

**Matrikelnr.:**

Bitte legen Sie Ihren Studierendenausweis und Ihren Lichtbildausweis vor sich auf das Pult, damit Ihre Identität während der Klausur geprüft werden kann.

- Bearbeiten Sie die Aufgaben jeweils auf den dafür vorgesehenen Seiten. Wenn Sie zusätzliche Blätter benötigen, geben Sie uns bitte ein Zeichen. Schreiben Sie auf zusätzliche Blätter Ihren Namen und welche Aufgabe Sie bearbeiten.
- Sie können einzelne Klausurbögen mit „bitte nicht werten“ kennzeichnen, aber nicht mitnehmen. Die Klausuraufgaben werden wir veröffentlichen.
- In allen Aufgaben meinen wir mit „definieren“ und „beweisen“ die Erstellung formaler Definitionen und Beweisterme im Calculus of Inductive Constructions. Beschränken Sie sich auf die Taktiken, die in der Vorlesung eingeführt wurden.

**Viel Erfolg!**

Aufgabe	1	2	3	4	5	6	7	8	$\Sigma$
Punkte									/80

# Spickzettel Teil I: Typisierungsregeln

Einfach getypter  $\lambda$ -Kalkül.

$$\text{Konst} \frac{}{\Gamma \vdash c : T} \quad \text{wenn } c \text{ eine Konstante von Typ } T \text{ ist}$$

$$\text{Var} \frac{}{\Gamma \vdash x : \Gamma(x)}$$

$$\text{Anw} \frac{\Gamma \vdash t : S \rightarrow T \quad \Gamma \vdash s : S}{\Gamma \vdash t s : T}$$

$$\text{Lam} \frac{\Gamma, x : S \vdash t : T}{\Gamma \vdash (\lambda x : S. t) : S \rightarrow T}$$

Calculus of Inductive Constructions.

$$\text{Konst} \frac{}{\Gamma \vdash c : T} \quad \text{wenn } c \text{ eine Konstante von Typ } T \text{ ist}$$

$$\text{Var} \frac{}{\Gamma \vdash x : \Gamma(x)}$$

$$\text{Anw} \frac{\Gamma \vdash t : \Pi x : S. T \quad \Gamma \vdash s : S}{\Gamma \vdash t s : T[x := s]}$$

$$\text{Lam} \frac{\Gamma, x : S \vdash t : T \quad \Gamma \vdash S : \text{Sort}_u}{\Gamma \vdash (\lambda x : S. t) : (\Pi x : S. T)}$$

$$\text{Sort} \frac{}{\Gamma \vdash \text{Sort}_u : \text{Sort}_{u+1}}$$

$$\text{Pi} \frac{\Gamma \vdash S : \text{Sort}_u \quad \Gamma, x : S \vdash T : \text{Sort}_v}{\Gamma \vdash \Pi x : S. T : \text{Sort}_{\text{imax}(u,v)}}$$

$$\text{DefGl} \frac{\Gamma \vdash t : T}{\Gamma \vdash t : S} \quad \text{wenn } T \equiv S$$

wobei

$$\text{imax}(u, v) = \begin{cases} 0 & \text{wenn } v = 0 \\ \max(u, v) & \text{sonst} \end{cases}$$

## Spickzettel Teil II: Wichtige Definitionen

### Logische Prädikate.

```
inductive And (p q : Prop) : Prop :=
| And.intro : p → q → And p q

inductive True : Prop :=
| True.intro : True

inductive Or (p q : Prop) :=
| Or.inl : p → Or p q
| Or.inr : q → Or p q

inductive False : Prop :=
[Induktiver Typ ohne Konstruktoren]

def Not (p : Prop) : Prop := p → False

inductive Iff (p q : Prop) : Prop :=
| Iff.intro : (p → q) → (q → p) → Iff p q

inductive Exists.{u} {X : Typeu} (p : X → Prop) : Prop :=
| Exists.intro : ∀ x : X. p x → Exists p

inductive Eq.{u} {X : Typeu} (x : X) : X → Prop :=
| rfl : Eq x x
```

### Eliminationsregeln.

```
And.left : Π{p q : Prop}. And p q → p   And.right : Π{p q : Prop}. And p q → p
Or.elim : Π{p q r : Prop}. Or p q → (p → r) → (q → r) → r
False.elim : Π{p : Prop}. False → p
Iff.mp : ∀{p q : Prop}. Iff p q → (p → q)   Iff.mpr : ∀{p q : Prop}. Iff p q → (q → p)
Exists.elim.{u} : Π{X : Typeu} {p : X → Prop} {q : Prop}.
  Exists p → (∀ x : X. p x → q) → q
Eq.subst : Π{X : Type} {x y : X} {p : X → Prop}. Eq x y → p x → p y
```

### Induktive Typen.

```
inductive Bool : Type :=
| false : Bool
| true : Bool

inductive Nat : Type :=
| zero : Nat
| succ : Nat → Nat

inductive List (X : Type) : Type :=
| nil : List X
| cons : X → List X → List X
```

**Aufgabe 1** (Einfach getypter  $\lambda$ -Kalkül). Sind die folgenden Terme wohlgetypt im einfach getypten  $\lambda$ -Kalkül? Wenn nein, begründen Sie kurz. Wenn ja, geben Sie den Typ des Terms an und ergänzen Sie den Typ jeder durch  $\lambda$  gebundenen Variable. (Sie brauchen hier keine Typderivation aufschreiben.) Es seien  $A$  und  $B$  Basistypen und  $f : A \rightarrow B$ ,  $g : B \rightarrow A$  und  $a : A$  seien Konstanten.

(a)  $(\lambda x. g (g x)) (f a)$

(b)  $(\lambda y x. y x) g$





**Aufgabe 2** (Typderivation). Erstellen Sie eine Typderivation für  $\lambda x : \mathbf{Prop}. \mathbf{Not} \ x$  im Calculus of Inductive Constructions, wobei  $\mathbf{Not} : \mathbf{Prop} \rightarrow \mathbf{Prop}$  eine Konstante ist. Der Spickzettel am Anfang der Klausur kann hier hilfreich sein.







**Aufgabe 3** (Induktive Typen und Pattern-Matching).

- (a) Definieren Sie eine „kleiner-gleich“ Funktion  $kg : \text{Nat} \rightarrow \text{Nat} \rightarrow \text{Bool}$ , die zu zwei natürlichen Zahlen genau dann `true` zurückgibt, wenn die erste Zahl kleiner als oder gleich der zweiten ist. (Achtung: Der Rückgabewert soll `Bool` sein, nicht `Prop`!)
- (b) Definieren Sie eine Funktion  $\text{sortiert} : \text{List Nat} \rightarrow \text{Bool}$ , die genau dann `true` zurückgibt, wenn eine gegebene Liste von natürlichen Zahlen aufsteigend sortiert ist. Die Funktion `sortiert` soll also für `[1, 1, 33, 230]` `true` zurückgeben, aber für `[4, 2, 7]` `false`.

Als gegeben können Sie dabei annehmen und verwenden:

- die Funktion `kg` aus Teil (a)
- eine Funktion  $\text{and} : \text{Bool} \rightarrow \text{Bool} \rightarrow \text{Bool}$ , die genau dann `true` zurückgibt, wenn ihre beiden Argumente `true` sind.

Tipp: Unterscheiden Sie drei Fälle: die leere Liste, eine Liste mit einem Element, und eine Liste mit mindestens zwei Elementen.





**Aufgabe 4** (Grundlegende Beweise). Beweisen Sie folgende Theoreme. Der Spickzettel am Anfang der Klausur kann hier hilfreich sein.

(a)

**theorem** false\_iff\_not\_true : False  $\leftrightarrow$   $\neg$ True := ...

(b)

**theorem** not\_implies\_of\_and\_not (p q : Prop) :

(p  $\wedge$   $\neg$ q)  $\rightarrow$   $\neg$ (p  $\rightarrow$  q) := ...





**Aufgabe 5** (Induktive Prädikate). Gegeben sei das folgende induktive Prädikat, das  $\leq$  auf den natürlichen Zahlen repräsentiert:

```
inductive LE : Nat → Nat → Prop
| refl : ∀(n : Nat). LE n n
| step : ∀{n m : Nat}. LE n m → LE n (succ m)
```

(a) Beweisen Sie, dass  $0 \leq 2$ :

```
theorem zero_le_two : LE zero (succ (succ zero)) := ...
```

(b) Beweisen Sie, dass  $0 \leq n$  für jedes  $n$  gilt:

```
theorem zero_le (n : Nat) : LE zero n := ...
```







**Aufgabe 6** (Gleichungen). Gegeben seien die folgenden Funktionen `pred`, die eine natürliche Zahl auf ihren Vorgänger abbildet (und 0 auf 0), und `sub`, die zwei natürliche Zahlen subtrahiert (oder 0 zurückgibt, wenn die erste Zahl kleiner ist als die zweite):

```
def pred : Nat → Nat
| zero ⇒ zero
| succ x ⇒ x

def sub : Nat → Nat → Nat
| x, zero ⇒ x
| x, succ y ⇒ pred (sub x y)
```

Beweisen Sie, dass `sub` immer 0 zurückgibt, wenn das erste Argument 0 ist:

```
theorem zero_sub (y : Nat) : sub zero y = zero := ...
```

Tipp: Verwenden Sie die Definitionen von `sub` und `pred` und gebrauchen Sie Induktion.





**Aufgabe 7** (Rekursoren). Definieren Sie die folgenden Funktionen mithilfe von Rekursoren statt Pattern-Matching. Geben Sie auch die impliziten Argumente der Rekursoren an.

(a)

```
def or : Bool → Bool → Bool
| false, false ⇒ false
| true, false ⇒ true
| false, true ⇒ true
| true, true ⇒ true
```

Verwenden Sie hier den Rekursor von Bool:

```
Bool.rec.{u} : Π{motive : Bool → Sortu}.
  motive false → motive true → Πt : Bool. motive t
```

(b)

```
def fak : Nat → Nat
| zero ⇒ 1
| succ n ⇒ fak n * (succ n)
```

Verwenden Sie hier den Rekursor von Nat:

```
Nat.rec.{u} : Π{motive : Nat → Sortu}.
  motive zero →
  (Πn : Nat. motive n → motive (succ n)) →
  Πt : Nat. motive t
```







**Aufgabe 8** (Anwendungen).

- (a) Der folgende Compiler, den wir in der Vorlesung verifiziert haben, setzt arithmetische Ausdrücke in Instruktionen für eine Stapelmaschine um:

```
def comp : AExp → List Instr
| num n ⇒ [LOADI n]
| var x ⇒ [LOAD x]
| plus a b ⇒ comp a ++ comp b ++ [ADD]
```

Für welchen arithmetischen Ausdruck liefert der Compiler die folgenden Instruktionen?

```
[LOADI 1, LOADI 2, LOAD "x", ADD, ADD]
```

- (b) Geben Sie einen Syntaxbaum für den Satz „Sofia lacht“ und eine Schritt-für-Schritt-Berechnung von  $\llbracket \text{Sofia lacht} \rrbracket$  an (in  $\beta$ -reduzierter Form). Wenn nicht eindeutig, geben Sie an, welche syntaktische Struktur gemeint ist, z.B.  $\llbracket \text{Sofia (EN)} \rrbracket$  statt  $\llbracket \text{Sofia} \rrbracket$ . Verwenden Sie dabei die folgenden Regeln:

$\mathbf{EN} \longrightarrow \text{Sofia}$	$\llbracket \mathbf{EN} \rrbracket := (\text{sofia} : \text{Entität})$
$\mathbf{IV} \longrightarrow \text{lacht}$	$\llbracket \mathbf{IV} \rrbracket := (\text{lacht} : \text{Entität} \rightarrow \text{Prop})$
$\mathbf{NP} \longrightarrow \mathbf{EN}$	$\llbracket \mathbf{NP} \rrbracket := \lambda v : \text{Entität} \rightarrow \text{Prop}. v \llbracket \mathbf{EN} \rrbracket$
$\mathbf{VP} \longrightarrow \mathbf{IV}$	$\llbracket \mathbf{VP} \rrbracket := \llbracket \mathbf{IV} \rrbracket$
$\mathbf{S} \longrightarrow \mathbf{NP} \mathbf{VP}$	$\llbracket \mathbf{S} \rrbracket := \llbracket \mathbf{NP} \rrbracket \llbracket \mathbf{VP} \rrbracket$

- (c) Teilbarkeit ist definiert als:

```
def teilt (a b : Nat) : Prop :=  $\exists x : \text{Nat}. a * x = b$ 
```

Beweisen Sie:

```
theorem kein_vielfaches (a b c : Nat) (hab :  $\neg \text{teilt } a b$ ) :  $\neg (a * c = b)$  := ...
```



