Beweise CYP

1: Struktur

1.1 – Syntax von Beweisen

Reihenfolge von Beweisen
Eine Beweisdatei kann eine beliebige Anzahl von Beweisen enthalten. Beweise, die weiter unten in der Datei stehen, können die vorher bewiesenen Lemmas nutzen.

Lemma
Jedes Lemma startet mit der zu beweisenen Gleichung. Es kann benannt werden (optional)

```
Lemma <opt_name>: <lhs> .=. <rhs>
```

Beweisart
Im nächsten Schritt wird die Beweisart angegeben. Extensionalitäten werden hier auch genannt.

```
Proof by extensionality with <var>
```

Beweis
Der Beweis wird als eine Reihe von Gleichungen mit Erklärung der Umformungsschritte dargestellt. Dabei muss jede Gleichung in einer eigenen Zeile stehen und um ein Leerzeichen eingerückt sein.

```
                    <term a1>
(by <reason>)   .=. <term a2>
                     .
                     .
                     .
(by <reason>)   .=. <term an>

                    <term b1>
(by <reason>)   .=. <term b2>
                     .
                     .
                     .
(by <reason>)   .=. <term bn>
```

Gültigkeit des Beweises
- Der erste Term ist `<lhs>` aus dem Lemma, der letze Term ist `<rhs>` aus dem Lemma.
- Alle Umformungsschritte sind gültig. Ein Umformungsschritt `<term a> .=. <term b>` ist gültig, wenn `<term a>` zu `<term b>` umgeformt werden kann:
  > durch Gleichung aus der Hintergrundtheorie `(by def <func_name>)`
  > durch ein zuvor bewiesenes Lemma `(by <lemma_name>)`
  > durch die Induktionshypothese `(by IH)`
  > durch die Annahme (bei Fallanalyse) `(by Assumption)`

Abschluss des Beweises
Ein Beweis wird abgeschlossen durch

```
QED
```

1.2 – Beweisarten

Beweis durch Gleichheit

Gleichheitsbweise werden durch folgendes Schlagwort eingeleitet:

```
Proof
```

Strukturelle Induktion

Strukturelle Induktion wird wie folgt eingeleitet:

```
Proof by induction on <type> <var>
```

Dabei ist `<var>` die Induktionsvariable, `<type>` ist der Name des Datentypes der Induktionsvariable (z.B. Tree, List).
Für jeden Konstruktor `<cons>` des Datentypes `<type>` wird eine Zeile benötigt.

```
Case <con>
```

Danach wird das Subgoal (gemäß des Cases sind Variablen ersetzt) und eine mögliche Induktionshypothese angegeben (falls es sich nicht um den Basisfall handelt).

```
To show: <subgoal>
IH: <induction hypothesis>
```

Der eigentliche Beweis wird durch `Proof` eingeleitet, dann folgen eine Reihe von Umformungen und der Beweis wird mit `QED` abgeschlossen.

Rechnerische Induktion

Rechnerische Induktion über eine bestimmte Funktion `<func>` mit ihren formalen Parametern `<agrs>` wird wie folgt eingeleitet:

```
Proof by computation induction on <args> with <func>
```

Für jede definierende Gleichung von `<func>` benötigt man einen eigenen Beweis (entsprechend numeriert).

```
Case <number>
```

Danach wird das Subgoal und eine mögliche Induktionshypothese angegeben. Die Anzahl der nötigen Induktionshypothesen hängt von der Anzahl der rekursiven Aufrufe im akt. Case ab.

```
To show: <subgoal>
IH: <induction hypothesis>
```

Der eigentliche Beweis wird durch `Proof` eingeleitet, dann folgen eine Reihe von Umformungen und der Beweis wird mit `QED` abgeschlossen.

Fallanalyse

Fallanalyse wird bei nicht-rekursiven Funktionen verwendet, wenn der übergebene Parameter durch mehrere Konstruktoren `<cons>` generiert werden kann.

```
Proof by case analyses on <data_type> <var_name>
```

Für jeden Konstruktor `<cons>` wird ein eigener Beweis benötigt.

```
Case <cons>
```

Potentielle Variabeln werden wie folgt gesetzt:

```
Assumption: <varName> .=. <cons>
```

Der eigentliche Beweis wird durch `Proof` eingeleitet, dann folgen eine Reihe von Umformungen und der Beweis wird mit `QED` abgeschlossen.

1.3 – Templates

Beweisart	Template

Equality Proof	`Lemma <optional name>: <left_side> .=. <right_side>`
	`Proof`
	`  <left_side>`
	`(by <rule>) .=. ...`
	`(by <rule>) .=. <subgoal>`
	
	`  <right_side>`
	`(by <rule>) .=. <...>`
	`(by <rule>) .=. <subgoal>`
	`QED`

Structural Induction	`Lemma <optional_name>: <left_side> .=. <right_side>`
	
	`Proof by induction on <data_type> <variable_name>`
	
	`Case <base_case>`
	`To show: <lemma_for_base_case>`
	
	`Proof`
	`...`
	`QED`
	
	`Case <other_case>`
	`To show: <lemma_for_other_case>`
	`IH: <induction_hypothesis>`
	
	`Proof`
	`...`
	`QED`
	`QED`

Computational Induction	`Lemma <optional_name>: <left_side> .=. <right_side>`
	
	`Proof by computation induction on <variables> with <function_name>`
	
	`Case 1`
	`To show: <lemma_for_recursion_base>`
	
	`Proof`
	`...`
	`QED`
	
	`Case 2`
	`To show: <lemma_for_next_equation>`
	`IH: <induction_hypothesis>`
	
	`Proof`
	`...`
	`QED`
	`QED`

Case Analysis	`Proof by case analysis on <data_type> <variable_name>`
	
	`Case <constructor>`
	`Assumption: <variable_name> .=. <constructor>`
	
	`Proof`
	`...`
	`QED`
	
	`Case <other_constructor>`
	`...`
	
	`QED`

2: Beispiele

2.1 – Basis Beweise

Listen

- Äquivalenz von Cons- und Verknüpfungs-Operator

```
 Lemma con: (x : xs) .=. [x] ++ xs
   Proof
                     (x : xs)
     (by def ++) .=. x:([]++xs)
     (by def ++) .=. (x:[])++xs
   QED
```

- Neutralität der leeren Liste

```
Lemma appNil: xs ++ [] .=. xs
  Proof by induction on List xs
  Case []
    To show: [] ++ [] .=. []
    Proof
                     [] ++ []
      (by def ++) .=. []
    QED
  Case (x:xs)
    To show: (x:xs) ++ [] .=. (x:xs)
    IH: xs ++ [] .=. xs
    Proof
                     (x:xs) ++ []
      (by def ++) .=. x : (xs ++ [])
      (by IH)     .=. x : xs
    QED
QED
```

- Assozitivität des Verknüpfungs-Operators

```
Lemma assoc_list: xs ++ (ys ++ zs) .=. (xs ++ ys) ++ zs
  Proof by induction on List xs
  Case []
    To show: [] ++ (ys ++ zs) .=. ([] ++ ys) ++ zs
    Proof
                     [] ++ (ys ++ zs)
      (by def ++) .=. ys ++ zs

                     ([] ++ ys) ++ zs
      (by def ++) .=. ys ++ zs
    QED
  Case (x:xs)
  To show: (x:xs) ++ (ys ++ zs) .=. ((x:xs) ++ ys) ++ zs
    IH: xs ++ (ys ++ zs) .=. (xs ++ ys) ++ zs
    Proof
                     (x:xs) ++ (ys ++ zs)
      (by def ++) .=. x : (xs ++ (ys ++ zs))
      (by IH)     .=. x : ((xs ++ ys) ++ zs)

                     ((x:xs) ++ ys) ++ zs
      (by def ++) .=. (x : (xs ++ ys)) ++ zs
      (by def ++) .=. x : ((xs ++ ys)) ++ zs
    QED
QED
```

Bool

- Doppelte Negation

```
Lemma: not . not .=. id
Proof by extensionality with x
  To show: (not . not) x .=. id x
  Proof by induction on Bool x
    Case False
      To show: (not . not) False .=. id False
      Proof
            (not . not) False
      (by def .) .=. not (not False)
      (by def not) .=. not True
      (by def not) .=. False
      (by def id) .=. id False
      QED
    Case True
      To show: (not . not) True .=. id True
      Proof
            (not . not) True
      (by def .) .=. not (not True)
      (by def not) .=. not False
      (by def not) .=. True
      (by def id) .=. id True
      QED
  QED
QED
```