Rechnerübung zu Theorembeweiser und ihre Anwendungen

Prof. Dr.-Ing. Gregor Snelting
Dipl.-Inf. Univ. Daniel Wasserrab

Lehrstuhl Programmierparadigmen
IPD Snelting
Universität Karlsruhe (TH)

Teil VIII

Wechselseitige Rekursion

wechselseitige Rekursion

Probleme bei bisheriger Rekursion: was tun bei

- mehreren Datentypen, die sich gegenseitig
- Datentypen, die eine Liste ihres eigenen Typs

bei der Definition verwenden?

Beispiel: Bäume mit beliebigem Verzweigungsgrad
jeder Knoten verwaltet eine Liste von Nachfolgerbäumen
Datentyp einfach wie bisher definiert:

```
datatype 'a tree = Leaf 'a
  | Node 'a "'a tree list"
```

Datentyp wechselseitig rekursiv definiert für sich und Liste seiner Typen

wechselseitige Rekursion

Probleme bei bisheriger Rekursion: was tun bei

- mehreren Datentypen, die sich gegenseitig
- Datentypen, die eine Liste ihres eigenen Typs

bei der Definition verwenden?

Beispiel: Bäume mit beliebigem Verzweigungsgrad
jeder Knoten verwaltet eine Liste von Nachfolgerbäumen
Datentyp einfach wie bisher definiert:

```
datatype 'a tree = Leaf 'a
  | Node 'a "'a tree list"
```

Datentyp wechselseitig rekursiv definiert für sich und Liste seiner Typen

wechselseitige Rekursion

Wollen Höhenfunktion für solche Bäume definieren

Ansatz:

primrec *height :: "'a tree ⇒ nat"*

where *"height (Leaf l) = 1"*
| "height (Node n ts) = ?

wechselseitige Rekursion

Wollen Höhenfunktion für solche Bäume definieren
Ansatz:

```
primrec height :: "'a tree ⇒ nat"
```

```
where "height (Leaf l) = 1"
| "height (Node n ts) = heights ts + 1"
```

brauchen Definition der Höhe für Liste von Bäumen!

wechselseitige Rekursion

Wollen Höhenfunktion für solche Bäume definieren
Ansatz:

```
primrec height :: "'a tree ⇒ nat"
  and heights :: "'a tree list ⇒ nat"

  where "height (Leaf l) = 1"
  | "height (Node n ts) = heights ts + 1"

  | "heights [] = 0"
  | "heights (t#ts) = max (height t) (heights ts)"
```

brauchen Definition der Höhe für Liste von Bäumen!
dazu gleichzeitige Definition der Funktion *height* für *'a tree*
und *heights* für *'a tree list*
sowohl **primrec** als auch **fun** verwendbar

Beweisen mit wechselseitig rekursiv definierten Datentypen

Wir wollen nun zeigen, dass die Höhe jedes Baums größer als 1 ist

1. Versuch:

lemma

```
      "height t > 0"
 apply(induct t)
```

Beweisen mit wechselseitig rekursiv definierten Datentypen

Wir wollen nun zeigen, dass die Höhe jedes Baums größer als 1 ist

1. Versuch:

lemma

```
      "height t > 0"
 apply(induct t)
```

komisches subgoal: `?P2.0 []`

Beweisen mit wechselseitig rekursiv definierten Datentypen

Wir wollen nun zeigen, dass die Höhe jedes Baums größer als 1 ist
1. Versuch:

lemma

```
      "height t > 0"
```

apply`(induct t)`

komisches subgoal: `?P2.0 []`

wir haben keine Aussage über die Höhe von Baumlisten!

Beweisen mit wechselseitig rekursiv definierten Datentypen

Wir wollen nun zeigen, dass die Höhe jedes Baums größer als 1 ist

2. Versuch:

lemma

```
"height t > 0" and "heights ts ≥ 0"
```

apply`(induct t)`

Beweisen mit wechselseitig rekursiv definierten Datentypen

Wir wollen nun zeigen, dass die Höhe jedes Baums größer als 1 ist
2. Versuch:

lemma

"height t > 0" **and** *"heights ts* $\geq$ *0"*

apply(*induct t* **and** *ts*)

müssen beide Parameter getrennt durch **and** angeben

Beweisen mit wechselseitig rekursiv definierten Datentypen

Wir wollen nun zeigen, dass die Höhe jedes Baums größer als 1 ist
2. Versuch:

lemma

"height t > 0" **and** *"heights ts* $\geq$ *0"*

apply(*induct t* **and** *ts*)

Problem: wegen generischem Elementyp *'a* werden *t* und *ts* unterschiedliche Typen zugeordnet! *t::"'a tree", ts::"'b tree list"*

Beweisen mit wechselseitig rekursiv definierten Datentypen

Wir wollen nun zeigen, dass die Höhe jedes Baums größer als 1 ist

3. Versuch:

lemma fixes *t::"'a tree"* **and** *ts::"'a tree list"*
shows *"height t > 0"* **and** *"heights ts* $\geq$ *0"*
apply(*induct t* **and** *ts*) **apply** *auto* **done**

Beweisen mit wechselseitig rekursiv definierten Datentypen

Wir wollen nun zeigen, dass die Höhe jedes Baums größer als 1 ist

4. Versuch: Variante mit *fun*

lemma fixes *t::"'a tree"* **and** *ts::"'a tree list"*
shows *"height t > 0"* **and** *"heights ts* $\geq$ *0"*
apply(*induct rule:height_heights.induct*) **apply** *auto* **done**

fun definiert Induktionsregel für wechselseitige Rekursion:

height_heights.induct

wechselseitige Rekursion und **arbitrary**

Lemma für wechselseitige Rekursion so viele "Teillemmas"
wie Datentypen rekursiv definiert
Damit kann dann wechselseitige Induktion angewandt werden
was passiert jedoch, wenn ein "Teillemma" nur gezeigt werden kann, wenn Induktionshypothese bestimmte Variablen allquantifizieren muss?
Kennen allgemein schon die Lösung: **arbitrary**

```
lemma "P t" "Q a t ts"
apply(induct t and ts)
```

wechselseitige Rekursion und **arbitrary**

Lemma für wechselseitige Rekursion so viele "Teillemmas"
wie Datentypen rekursiv definiert
Damit kann dann wechselseitige Induktion angewandt werden
was passiert jedoch, wenn ein "Teillemma" nur gezeigt werden kann, wenn Induktionshypothese bestimmte Variablen allquantifizieren muss?
Kennen allgemein schon die Lösung: **arbitrary**

lemma *"P t" "Q a t ts"*
apply(*induct t* **and** *ts*)

`Q` braucht `a` in Induktionshypothese quantifiziert, also in einem **arbitrary**

wechselseitige Rekursion und **arbitrary**

Lemma für wechselseitige Rekursion so viele "Teillemmas"
wie Datentypen rekursiv definiert
Damit kann dann wechselseitige Induktion angewandt werden
was passiert jedoch, wenn ein "Teillemma" nur gezeigt werden kann, wenn Induktionshypothese bestimmte Variablen allquantifizieren muss?
Kennen allgemein schon die Lösung: **arbitrary**

```
lemma "P t" "Q a t ts"
apply(induct t and ts arbitrary: and a)
```

auch hinter **arbitrary** die zu quantifizierenden Variablen für jedes Lemma mit **and** trennen

Induktiv wechselseitig

Wechselseitigkeit auch bei induktiven Definitionen funktioniert analog zu wechselseitiger Rekursion

Beispiel:

```
inductive even :: "nat ⇒ bool"
      and odd :: "nat ⇒ bool"
  where "even 0"
  | "odd n ⟹ even (Suc n)"
  | "even n ⟹ odd (Suc n)"
```