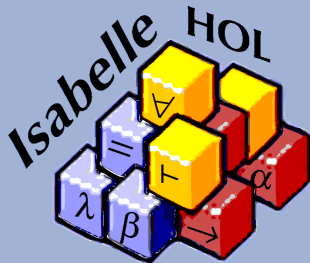


# Theorembeweiserpraktikum

## Anwendungen in der Sprachtechnologie

<http://pp.info.uni-karlsruhe.de/lehre/SS2010/tba/>  
Daniel Wasserrab

IPD Snelting, Lehrstuhl Programmierparadigmen



Bisher kennen wir *simp* als Termersetzungstaktik

Viele weitere automatische Taktiken für den *logical reasoner*

Unterscheiden sich in

- Regelmenge,
- ob nur auf ein oder alle Zwischenziele angewendet,
- ob Abbruch bei nichtgelösten Zielen oder Rückgabe an Benutzer,
- ob Simplifier mitverwendet oder nicht,
- ob mehr Zwischenziele erzeugt werden dürfen oder nicht

z.B. entspricht *simp\_all simp*, jedoch Termersetzungsregeln auf jedes Zwischenziel angewandt

Im Folgenden ein paar Taktiken vorgestellt

- nur “offensichtliche” logische Regeln
- nur auf oberstes Zwischenziel angewendet
- nach Vereinfachung Rückgabe an Benutzer
- *clarify* ohne Simplifier, *clarsimp* = *clarify* + *simp*
- kein Aufteilen in Zwischenziele

Oft verwendet um aktuelles Zwischenziel leichter lesbar zu machen

- mächtige Regelmenge
- nur auf oberstes Zwischenziel angewendet
- versucht Ziel komplett zu lösen, ansonsten Abbruch
- kein Simplifier

Schnellster *logical reasoner*, gut auch bei Quantoren  
von den Entwicklern empfohlene Taktik

- mächtige Regelmenge, da basierend auf *blast*
- auf alle Zwischenziele angewendet
- nach Vereinfachung Rückgabe an Benutzer, wenn nicht gelöst
- mit Simplifier
- Aufteilen in Zwischenziele

oft verwendete “ad-hoc”-Taktik

*force* wie *auto*, nur sehr aggressives Lösen und Aufteilen, deswegen anfällig für Endlosschleifen und Aufhängen

- große Regelmenge
- nur auf oberstes Zwischenziel angewendet
- versucht Ziel komplett zu lösen, ansonsten Abbruch
- Simplifier, *fast* nur *logical reasoner*

*fastsimp* gilt als langsamste Taktik, von Entwicklern als *legacy* bezeichnet in der Praxis allerdings oft gut brauchbare Taktik für Lösen eines Zwischenziels

- Simplifikationsregeln: möglich bei allen Taktiken mit `Simplifier`  
normalerweise `simp:Regelname`, z.B. `apply(auto simp:bla)`  
bei `simp` und `simp_all` stattdessen `add:`, z.B. `apply(simp add:bla)`  
für Einschränken der Regelmenge: `simp del:` bzw. `simp only:`  
`apply(fastsimp simp del:bla)` bzw. `apply(auto simp only:bla)`
- Deduktionsregeln: alle Taktiken mit `logical reasoner`  
Unterscheidung zwischen Introduktions-, Eliminations- und  
Destruktionsregeln
  - Einführung: `intro:`, z.B. `apply(blast intro:foo)`
  - Elimination: `elim:`, z.B. `apply(auto elim:foo)`
  - Destruktion: `dest:`, z.B. `apply(fastsimp dest:foo)`
- alles beliebig kombinierbar, z.B.  
`apply(auto dest:foo intro:bar simp:zip zap)`

## Teil VII

# Primitive und wechselseitige Rekursion



# Deklaration von Funktionen

Festlegen von Namen und Signatur einer Funktion: **Deklaration**  
dazu in Isabelle das Schlüsselwort **consts**:

```
consts length :: "'a list  $\Rightarrow$  nat"
```

Vorsicht! Gibt der Funktion noch keinerlei Semantik!

Wird in den Übungen verwendet, um Funktionen einzuführen, die sie selbst  
noch definieren (also mit Semantik versehen) sollen

Selbst bitte vermeiden! Immer gleich Deklaration und Definition

Viele Datentypen mit Selbstbezug, z.B.

- natürliche Zahl (ungleich 0) ist Nachfolger einer natürlichen Zahl
- nichtleere Liste ist Liste mit zusätzlichem Kopfelement
- nichtleere Menge ist Menge mit einem zusätzlichen Element

Viele Datentypen mit Selbstbezug, z.B.

- natürliche Zahl (ungleich 0) ist Nachfolger einer natürlichen Zahl
- nichtleere Liste ist Liste mit zusätzlichem Kopfelement
- nichtleere Menge ist Menge mit einem zusätzlichen Element

Formalisierung in Isabelle/HOL am Bsp. natürliche Zahlen:

**datatype** *nat* = 0 / Suc *nat*

Also Konstruktoren von *nat*: 0 und Suc (Präfix)

# Parametertypen

verschiedene Typen in Containerdatentypen: Parametertyp 'a'  
kann bei Verwendung entsprechen initialisiert werden (muss aber nicht)

verschiedene Typen in Containerdatentypen: Parametertyp 'a  
kann bei Verwendung entsprechen initialisiert werden (muss aber nicht)

Beispiel: Listen mit Typparameter

```
datatype 'a list =  
  Nil    ("[]")  
  | Cons 'a "'a list"    (infix "#" 65)
```

Konstruktoren von list: [] und # (Infix) (x#[ ] = [x])

verschiedene Typen in Containerdatentypen: Parametertyp 'a  
kann bei Verwendung entsprechen initialisiert werden (muss aber nicht)

Beispiel: Listen mit Typparameter

```
datatype 'a list =  
  Nil    ("[]")  
  | Cons 'a "'a list"    (infix "#" 65)
```

Konstruktoren von list: [] und # (Infix) ( $x\#[] = [x]$ )

Funktionsdeklaration kann jetzt z.B. so aussehen:

```
consts foo :: "nat list  $\Rightarrow$  bool"  
consts bar :: "nat  $\Rightarrow$  bool list  $\Rightarrow$  nat"  
consts zip :: "'a list  $\Rightarrow$  'a"
```

**Definition** von Funktionen über rekursive Datentypen: **primrec** kombiniert Deklaration und Definition (frühere **consts** löschen!)  
ein Parameter der Funktion muss in seine Konstruktoren aufgeteilt werden  
Beispiel:

```
primrec length :: "'a list  $\Rightarrow$  nat"  
  where "length [] = 0"  
  | "length (x#xs) = Suc(length xs)"
```

```
primrec tl :: "'a list  $\Rightarrow$  'a list"  
  where "tl [] = []"  
  | "tl (x#xs) = xs"
```

Es müssen nicht alle Konstruktoren spezifiziert werden:

```
primrec hd :: "'a list ⇒ 'a"  
  where "hd(x#xs) = x"
```

```
primrec last :: "'a list ⇒ 'a"  
  where "last(x#xs) = (if xs=[] then x else last xs)"
```

Aussagen über nicht-enthaltene Konstruktoren wie z.B.  $hd([])$  kaum möglich, da nichinterpretierte Funktionssymbole, d.h. nicht vereinfachbar



# @ und rev

weiterer Infixoperator: @ hängt Listen zusammen

Beispiel:  $[0,4]@[2] = [0,4,2]$

# @ und rev

weiterer Infixoperator: @ hängt Listen zusammen

Beispiel:  $[0,4]@[2] = [0,4,2]$

rev dreht Listen um, also  $rev [0,4,2] = [2,4,0]$

Wie lautet die entsprechende Deklaration/Definition?

# @ und rev

weiterer Infixoperator: @ hängt Listen zusammen

Beispiel:  $[0,4]@[2] = [0,4,2]$

rev dreht Listen um, also  $rev [0,4,2] = [2,4,0]$

Wie lautet die entsprechende Deklaration/Definition?

```
primrec rev :: "'a list  $\Rightarrow$  'a list"
```

```
  where "rev [] = []"
```

```
  | "rev(x#xs) = rev(xs) @ [x]"
```

Beweise über rekursive Datentypen mittels **struktureller Induktion**  
d.h. Induktion über Konstruktoren des Datentyps

In Isabelle/HOL:

**lemma** *hd\_Cons\_tl*: " $xs \neq [] \implies hd\ xs \# tl\ xs = xs$ "

**apply** (*induct xs*)

**apply** *auto*

**done**

wendet strukturelle Induktion mit Datentypkonstruktoren von *xs* an  
automatische Taktik beendet Beweis

Problem: zu spezielle Induktionshypothesen

**lemma** " $(\text{rev } xs = \text{rev } ys) = (xs = ys)$ "

Induktion auf  $xs$  ermöglicht Lösen des  $[\ ]$ -Falles  
bei Induktionsschritt bleibt:

$\wedge a \ xs.$

$$(\text{rev } xs = \text{rev } ys) = (xs = ys) \implies$$

$$(\text{rev } (a \# xs) = \text{rev } ys) = (a \# xs = ys)$$

# Probleme mit Induktion

Problem: zu spezielle Induktionshypothesen

**lemma** " $(\text{rev } xs = \text{rev } ys) = (xs = ys)$ "

Induktion auf  $xs$  ermöglicht Lösen des  $[\ ]$ -Falles  
bei Induktionsschritt bleibt:

$\wedge a \ xs.$

$$(\text{rev } xs = \text{rev } ys) = (xs = ys) \implies$$

$$(\text{rev } (a \# xs) = \text{rev } ys) = (a \# xs = ys)$$

nicht lösbar!

$ys$  kann nicht gleich  $xs$  und  $a \# xs$  sein!

# Probleme mit Induktion

**Idee:**  $ys$  muss im Induktionsschritt freie Variable sein!

**Lösung:**  $ys$  nach *arbitrary* Schlüsselwort in Induktionsanweisung  
damit Induktionsschritt für  $ys$  meta-allquantifiziert:

**apply**(*induct xs arbitrary:ys*)

# Probleme mit Induktion

**Idee:**  $ys$  muss im Induktionsschritt freie Variable sein!

**Lösung:**  $ys$  nach *arbitrary* Schlüsselwort in Induktionsanweisung  
damit Induktionsschritt für  $ys$  meta-allquantifiziert:

**apply**(*induct xs arbitrary:ys*)

Resultiert in Induktionsschritt:

$\wedge a \ xs \ ys.$

$$(\wedge ys. (rev \ xs = rev \ ys) = (xs = ys)) \implies$$

$$(rev \ (a \ # \ xs) = rev \ ys) = (a \ # \ xs = ys)$$



# Probleme mit Induktion

**Idee:**  $ys$  muss im Induktionsschritt freie Variable sein!

**Lösung:**  $ys$  nach *arbitrary* Schlüsselwort in Induktionsanweisung  
damit Induktionsschritt für  $ys$  meta-allquantifiziert:

**apply**(*induct xs arbitrary:ys*)

Resultiert in Induktionsschritt:

$\wedge a \ xs \ ys.$

$$(\wedge ys. (rev \ xs = rev \ ys) = (xs = ys)) \implies$$

$$(rev \ (a \ # \ xs) = rev \ ys) = (a \ # \ xs = ys)$$

Heuristiken für (bisher scheiternde) Induktionen:

- alle freien Variablen (außer Induktionsvariable) mit *arbitrary*
- Induktion immer über das Argument, über das die Funktion rekursiv definiert ist
- Ziele durch Ersetzen von Konstanten durch Variablen generalisieren

# 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?

# 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  $\Rightarrow$  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  $\Rightarrow$  nat"  
  and heights :: "'a tree list  $\Rightarrow$  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 ≥ 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 ≥ 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 ≥ 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 ≥ 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 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**



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