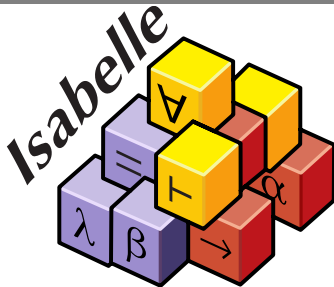


Theorembeweiserpraktikum

Anwendungen in der Sprachtechnologie

LEHRSTUHL PROGRAMMIERPARADIGMEN



Teil VIII

Quick and Dirty: apply-Skripte Oder: Wie es wirklich geht!

Isabelle arbeitet im Wesentlichen mit 3 Modi

theory mode: Deklarationen, Definitionen, Lemmas

state mode: Zwischenaussagen, Fixes, etc.

prove mode: Anwendung von Beweistaktiken

Befehle schalten zwischen den Modi hin und her

Beispiele:

definition: *theory mode* \rightarrow *theory mode*

lemma: *theory mode* \rightarrow *prove mode*

proof: *prove mode* \rightarrow *state mode*

assume: *state mode* \rightarrow *state mode*

show: *state mode* \rightarrow *prove mode*

by: *prove mode* \rightarrow *state mode* | *theory mode*

qed: *state mode* \rightarrow *state mode* | *theory mode*

apply: *prove mode* \rightarrow *prove mode*

verändert das aktuelle Beweisziel (`subgoal`) durch Anwendung der gegebenen *Taktik(en)*

Beispiel:

lemma `conjCommutes`: " $A \wedge B \implies B \wedge A$ "

apply (`rule conjI`)

apply (`erule conjE`)

apply `assumption`

apply (`erule conjE`)

apply `assumption`

done

done: *prove mode* \rightarrow *state mode* | *theory mode*

beendet einen Beweis

- Beweisziel** ist immer das aktuell zu zeigende Ziel unter `subgoal`.
- aktuelle Fakten** sind die gesammelten Fakten unter `this`.
- gegebene ...** sind die Parameter der Taktik.

manuelle Taktiken

- **(minus)**: Fügt die aktuellen Fakten dem Beweisziel hinzu.
 - fact**: Setzt aus den gegebenen Fakten das Beweisziel zusammen (modulo Unifikation und schematischen Typ- und Termvariablen).
- assumption**: Löst das Beweisziel, wenn eine passende Annahme vorhanden ist.
- this**: Wendet die aktuellen Fakten der Reihe nach als Regel auf das Beweisziel an.
- rule**: Wendet die gegebene(n) Regel(n) auf das Beweisziel an. Die aktuellen Fakten werden verwendet, um die Annahmen der Regel zu instanzieren.

- unfold:** Ersetzt die gegebenen Definitionen in allen Beweiszielen.
- fold:** Kollabiert die gegebenen Definitionen in allen Beweiszielen.
- insert:** Fügt die gegebenen Fakten in alle Beweisziele ein.
- erule:** Wendet die gegebene Regel als Eliminationsregel an.
- drule:** Wendet die gegebene Regel als Destruktionsregel an.
- frule:** Wie *drule*, aber hält die verwendete Annahme im Beweisziel.
- intro:** Wendet die gegebenen Regeln **wiederholt** als Introduktionsregel an.
- elim:** Wendet die gegebenen Regeln **wiederholt** als Eliminationsregel an.

Introduktion: Operator steht in der Konklusion (Standardname ... *I*)
„Was brauche ich, damit die Formel gilt?“

Beispiel: *conjI*: $\llbracket P; Q \rrbracket \Longrightarrow P \wedge Q$

Was passiert bei *rule*?

Konklusionen der Regel und des Beweisziels werden unifiziert. Jede Prämisse der Regel wird als neues Beweisziel hinzugefügt.

Elimination: Operator steht in der ersten Prämisse (Standardname ... *E*)
„Was kann ich aus der Formel folgern?“

Beispiel: *conjE*: $\llbracket P \wedge Q; \llbracket P; Q \rrbracket \Longrightarrow R \rrbracket \Longrightarrow R$

Was passiert bei *erule*?

Konklusionen der Regel und des Beweisziels werden unifiziert. Dann wird die erste Prämisse der Regel mit der ersten passenden Prämisse des Beweisziels unifiziert. Die übrigen Prämissen der Regel werden als neue Beweisziele hinzugefügt.

Destruktion: Operator steht in der ersten Prämisse
„Ich benötige eine schwächere Aussage.“

Beispiel: *conjunct1*: $P \wedge Q \implies P$

Was passiert bei *drule*?

Die erste Prämisse der Regel wird mit der ersten passenden Prämisse des Beweisziels unifiziert.
Die übrigen Prämissen der Regel werden als neue Beweisziele hinzugefügt.

Als letztes wird ein neues Beweisziel hinzugefügt, welches die Konklusion der Regel als Prämisse enthält.

Teil IX

Automatische Taktiken

Viele automatische Taktiken für den *logical reasoner* unterscheiden sich in

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

Im Folgenden werden ein paar Taktiken vorgestellt

- Simplifikationsregeln: Gleichungen
- entsprechende Taktik: `simp` (nur erstes Ziel) bzw. `simp_all` (alle Ziele)
 - besitzt Pool an Termersetzungsregeln
 - prüft für jede solche Regel, ob Term mit linker Seite einer Gleichung unifizierbar
 - falls ja, ersetzen mit entsprechend unifizierter rechter Seite
- genauer: Termersetzung (weil Ausdruck rechts in der Gleichung nicht notwendigerweise einfacher)

- Simplifikationsregeln: Gleichungen
- entsprechende Taktik: `simp` (nur erstes Ziel) bzw. `simp_all` (alle Ziele)
 - besitzt Pool an Termersetzungsregeln
 - prüft für jede solche Regel, ob Term mit linker Seite einer Gleichung unifizierbar
 - falls ja, ersetzen mit entsprechend unifizierter rechter Seite
- genauer: Termersetzung (weil Ausdruck rechts in der Gleichung nicht notwendigerweise einfacher)

Beispiel:

aktuelles Subgoal: $C \implies P$ (*if False then A else B \longrightarrow D*)

`simp` wendet folgende Termersetzungsregel an:

HOL.if_False: (if False then ?x else ?y) = ?y

- Simplifikationsregeln: Gleichungen
- entsprechende Taktik: `simp` (nur erstes Ziel) bzw. `simp_all` (alle Ziele)
 - besitzt Pool an Termersetzungsregeln
 - prüft für jede solche Regel, ob Term mit linker Seite einer Gleichung unifizierbar
 - falls ja, ersetzen mit entsprechend unifizierter rechter Seite
- genauer: Termersetzung (weil Ausdruck rechts in der Gleichung nicht notwendigerweise einfacher)

Beispiel:

aktuelles Subgoal: $C \implies P$ (*if False then A else B \longrightarrow D*)

`simp` wendet folgende Termersetzungsregel an:

HOL.if_False: (if False then ?x else ?y) = ?y

Resultat: $C \implies P$ (*B \longrightarrow D*)

Auch bedingte Ersetzungsregeln sind möglich, also in der Form

$$[[\dots]] \implies \dots = \dots$$

Dazu: Prämissen der Regel aus aktuellen Annahmen *via Simplifikation* herleitbar

Auch bedingte Ersetzungsregeln sind möglich, also in der Form

$$[[\dots]] \implies \dots = \dots$$

Dazu: Prämissen der Regel aus aktuellen Annahmen *via Simplifikation* herleitbar

Simplifier modifizieren:

- selbstgeschriebene Simplifikationslemmas zu Taktik hinzufügen:
`apply (simp add: Regel1 Regel2 ...)`
- nur bestimmte Ersetzungsregeln verwenden:
`apply (simp only: Regel1 Regel2 ...)`
- Ersetzungsregeln aus dem Standardpool von `simp` entfernen:
`apply (simp del: Regel1 Regel2 ...)`

Auch möglich: Ersetzungsregeln in den Standardpool von `simp` einfügen
Zwei Varianten:

- Zusatz `[simp]` hinter Lemmanamen
Beispiel: **lemma** `bla [simp]: "A = True \implies A \wedge B = B"`
- mittels **declare** ... `[simp]`
Beispiel: **declare** `foo [simp] bar [simp]`
Analog: mittels **declare** `[simp del]` Ersetzungsregeln
aus Standardpool entfernen
- Analog in einem **proof**-Block: Mit **have** resp. **note**.

Auch möglich: Ersetzungsregeln in den Standardpool von `simp` einfügen
Zwei Varianten:

- Zusatz `[simp]` hinter Lemmanamen
Beispiel: **lemma** `bla [simp]: "A = True \implies A \wedge B = B"`
- mittels **declare** ... `[simp]`
Beispiel: **declare** `foo [simp] bar [simp]`
Analog: mittels **declare** `[simp del]` Ersetzungsregeln
aus Standardpool entfernen
- Analog in einem **proof**-Block: Mit **have** resp. **note**.

Vorsicht!

- Nur Regeln zu Standardpool hinzufügen, deren rechte Seite einfacher als linke Seite!
- Sicherstellen, dass `simp` durch neue Regeln nicht in Endlosschleifen hängenbleibt!

- 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

- 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*

fastforce ist eine in der Praxis oft gut brauchbare Taktik für das 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(fastforce 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(fastforce dest:foo)`
- alles beliebig kombinierbar, z.B.
`apply(auto dest:foo intro:bar simp:zip zap)`

- Resolutionskalkül
- nur auf oberstes Zwischenziel angewendet
- versucht Ziel komplett zu lösen, ansonsten Abbruch
- es müssen alle Regeln angegeben werden, die verwendet werden sollen

metis kann sehr gut mit Gleichungen umgehen, wenn diese nicht nur von links nach rechts angewendet werden sollen.

Teil X

Datentypen und primitive Rekursion

Algebraische Datentypen werden über eine Menge von *Konstruktoren* beschrieben, die wiederum weitere Typen als *Parameter* haben. Jeder Wert des Typs ist durch genau einen Konstruktor und den Werten dessen Parameters beschrieben.

Ein Parameter kann auch der definierte Datentyp selbst sein. In dem Fall spricht man von einem rekursiven Datentyp.

Algebraische Datentypen werden über eine Menge von *Konstruktoren* beschrieben, die wiederum weitere Typen als *Parameter* haben. Jeder Wert des Typs ist durch genau einen Konstruktor und den Werten dessen Parameters beschrieben.

Ein Parameter kann auch der definierte Datentyp selbst sein. In dem Fall spricht man von einem rekursiven Datentyp.

Beispiele:

- Eine natürliche Zahl ist entweder Null, oder der Nachfolger einer natürlichen Zahl
- Eine Liste ist entweder leer oder eine weitere List mit zusätzlichem Kopfelement.

Algebraische Datentypen werden über eine Menge von *Konstruktoren* beschrieben, die wiederum weitere Typen als *Parameter* haben. Jeder Wert des Typs ist durch genau einen Konstruktor und den Werten dessen Parameters beschrieben.

Ein Parameter kann auch der definierte Datentyp selbst sein. In dem Fall spricht man von einem rekursiven Datentyp.

Beispiele:

- Eine natürliche Zahl ist entweder Null, oder der Nachfolger einer natürlichen Zahl
- Eine Liste ist entweder leer oder eine weitere List mit zusätzlichem Kopfelement.

Formalisierung in Isabelle/HOL am Bsp. natürliche Zahlen:

datatype *nat* = 0 | Suc *nat*

Also Konstruktoren von nat: 0 und Suc

Parametertypen

Soll der Typ eines Konstruktorparameters nicht festgelegt sein, verwendet man den Parametertyp 'a.

Soll der Typ eines Konstruktorparameters nicht festgelegt sein, verwendet man den Parametertyp 'a.

Beispiel: Listen mit Typparameter

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

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

Soll der Typ eines Konstruktorparameters nicht festgelegt sein, verwendet man den Parametertyp 'a.

Beispiel: Listen mit Typparameter

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

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

Damit kann man z.B. folgende Typen bilden:

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

Definition von Funktionen über rekursive Datentypen: **fun** ein Parameter der Funktion kann dabei in seine Konstruktoren aufgeteilt werden.

Auf der rechten Seite der Gleichungen sollte die definierte Funktion höchstens auf die Parameter des Konstruktors angewandt werden.

Beispiel:

```
fun length :: "'a list ⇒ nat"
  where "length [] = 0"
        | "length (x#xs) = Suc (length xs)"
```

```
fun tl :: "'a list ⇒ 'a list"
  where "tl [] = []"
        | "tl (x#xs) = xs"
```

Es müssen nicht alle Konstruktoren spezifiziert werden:

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

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

Bei nicht enthaltenen Konstruktoren wie z.B. `hd []` nimmt die Funktion den Wert `undefined` an, ein fester Wert in jedem Typ, über den nichts bekannt ist.

@ 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 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 Definition?

```
fun rev :: "'a list  $\Rightarrow$  'a list"  
  where "rev [] = []"  
  | "rev (x#xs) = rev xs @ [x]"
```

„Wenn eine Aussage für jeden möglichen Konstruktor gilt, dann auch für alle Werte des Typs.“

Regel für Listen:

$$\text{list.exhaust: } (xs = [] \implies P) \implies (\bigwedge a \text{ as. } xs = a \# \text{ as} \implies P) \implies P$$

„Wenn eine Aussage für jeden möglichen Konstruktor gilt, dann auch für alle Werte des Typs.“

Regel für Listen:

$$\text{list.exhaust: } (xs = [] \implies P) \implies (\bigwedge a \text{ as. } xs = a \# \text{ as} \implies P) \implies P$$

Wird meist mit der Beweistaktik `cases` verwendet:

lemma `hd_Cons_tl`: "`xs ≠ []` \longrightarrow `hd xs # tl xs = xs`"

proof (`rule impI`)

assume "`xs ≠ []`" **show** "`hd xs # tl xs = xs`"

proof (`cases xs`)

assume "`xs = []`"

with $\langle xs \neq [] \rangle$ **have** `False..`

then show `?thesis..`

next

fix `y ys`

assume "`xs = y # ys`"

then show `?thesis` **by** `simp`

qed

qed

Warum **proof** (*cases* ..) und nicht einfach **proof** (*rule List.exhaust*)?
Wegen benannter Fälle!

lemma *hd_Cons_tl*: "*xs* \neq [] \longrightarrow *hd xs # tl xs = xs*"

proof (*rule impI*)

assume "*xs* \neq []"

show "*hd xs # tl xs = xs*"

proof (*cases xs*)

case *Nil*

with \langle *xs* \neq [] \rangle **have** *False..*

thus *?thesis..*

next

case (*Cons y ys*)

then show *?thesis* **by** *simp*

qed

qed

(Bei *cases* auf Aussagen heißen die Fälle *True* und *False*.)

Warum **proof** (*cases* ..) und nicht einfach **proof** (*rule List.exhaust*)?
Wegen benannter Fälle!

```
lemma hd_Cons_tl: "xs ≠ [] → hd xs # tl xs = xs"
```

```
proof (rule impI)
```

```
  assume "xs ≠ []"
```

```
  show "hd xs # tl xs = xs"
```

```
  proof (cases xs)
```

```
    case Nil
```

```
    with ⟨xs ≠ []⟩ have False..
```

```
    thus ?thesis..
```

```
  next
```

```
    case (Cons y ys)
```

```
    then show ?thesis by simp
```

```
  qed
```

```
qed
```

(Bei *cases* auf Aussagen heißen die Fälle *True* und *False*.)

Hier geht natürlich auch einfach

```
lemma hd_Cons_tl: "xs ≠ [] ⇒ hd xs # tl xs = xs"
```

```
by (cases xs)auto
```

Fallunterscheidung genügt bei rekursiven Datentypen nicht immer, weil man die Aussage schon für die rekursiven Parameter braucht. Dann hilft **strukturelle Induktion**

Fallunterscheidung genügt bei rekursiven Datentypen nicht immer, weil man die Aussage schon für die rekursiven Parameter braucht. Dann hilft **strukturelle Induktion**

z.B. für Listen sieht die Regel wie folgt aus:

$$\text{list.induct: } \llbracket P [] ; \bigwedge x xs. P xs \implies P (x \# xs) \rrbracket \implies P xs$$

Fallunterscheidung genügt bei rekursiven Datentypen nicht immer, weil man die Aussage schon für die rekursiven Parameter braucht. Dann hilft **strukturelle Induktion**

z.B. für Listen sieht die Regel wie folgt aus:

$$\text{list.induct: } \llbracket P []; \bigwedge x \text{ xs. } P \text{ xs} \implies P (x \# \text{xs}) \rrbracket \implies P \text{ xs}$$

Komfortabler als *rule* ist die Methode *induction*:

- Benannte Fälle wie bei *cases*
- *Fall.IH* ist die Induktionshypothese, also die Aussage für die rekursiven Parameter
- ggf. *Fall.premis* sind die Annahmen, wie sie im Lemma stehen.
- ggf. *Fall.hyps* sind zusätzliche Aussagen, die durch die Induktionsregel eingefügt werden.

Beispiel für strukturelle Induktion

lemma "length (xs @ ys) = length xs + length ys"

proof (induction xs)

case Nil

have "length ([] @ ys) = length ys" **by** simp

also have "... = 0 + length ys" **by** simp

also have "... = length [] + length ys" **by** simp

finally show ?case.

next

case (Cons x xs)

have "length ((x # xs) @ ys) = length (x # (xs @ ys))" **by** simp

also have "... = Suc (length (xs @ ys))" **by** simp

also from Cons.IH **have** "... = Suc (length xs + length ys)"..

also have "... = Suc (length xs) + length ys" **by** simp

also have "... = length (x # xs) + length ys" **by** simp

finally show ?case.

qed

Beispiel für strukturelle Induktion

lemma "length (xs @ ys) = length xs + length ys"

proof (induction xs)

case Nil

have "length ([] @ ys) = length ys" **by** simp

also have "... = 0 + length ys" **by** simp

also have "... = length [] + length ys" **by** simp

finally show ?case.

next

case (Cons x xs)

have "length ((x # xs) @ ys) = length (x # (xs @ ys))" **by** simp

also have "... = Suc (length (xs @ ys))" **by** simp

also from Cons.IH **have** "... = Suc (length xs + length ys)"..

also have "... = Suc (length xs) + length ys" **by** simp

also have "... = length (x # xs) + length ys" **by** simp

finally show ?case.

qed

oder natürlich einfach

by (induction xs) auto

Problem: zu spezielle Induktionshypothesen

lemma " $rev\ xs = rev\ ys \implies xs = ys$ "

proof (*induction xs*)

- Fall `[]` mit *simp* lösbar:
case Nil then show ?case by simp

Problem: zu spezielle Induktionshypothesen

lemma " $rev\ xs = rev\ ys \implies xs = ys$ "

proof (*induction xs*)

- Fall $[]$ mit *simp* lösbar:

case Nil then show ?case by simp

- bei Induktionsschritt bekommen wir:

Cons.IH: $rev\ xs = rev\ ys \implies xs = ys$

*Cons.prem*s: $rev\ (x \# xs) = rev\ ys$

aber $rev\ ys$ kann nicht gleichzeitig $rev\ xs$ **und** $rev\ (a \# xs)$ sein!

\Rightarrow Induktionshypothese nicht verwendbar

\Rightarrow So kommen wir nicht weiter.

Lösung:

y_s muss in der Induktionshypothese eine freie Variable sein!

Lösung:

ys muss in der Induktionshypothese eine freie Variable sein!

Umsetzung:

ys nach Schlüsselwort *arbitrary* in der Induktionsanweisung. Damit wird die Induktionshypothese für ys meta-allquantifiziert:

lemma "*rev xs = rev ys \implies xs = ys*"

proof (*induction xs arbitrary: ys*)

Lösung:

ys muss in der Induktionshypothese eine freie Variable sein!

Umsetzung:

ys nach Schlüsselwort *arbitrary* in der Induktionsanweisung. Damit wird die Induktionshypothese für ys meta-allquantifiziert:

lemma " $rev\ xs = rev\ ys \implies xs = ys$ "

proof (*induction xs arbitrary: ys*)

Nun liefert uns

case (*Cons x xs ys*)

diese Aussagen:

Cons.IH: $rev\ xs = rev\ ?ys \implies xs = ?ys$

Cons.prem: $rev\ (x \# xs) = rev\ ys$

wobei die Variable mit dem Fragezeichen frei ist, also mit jeder beliebigen Liste verwendet werden kann.

Heuristiken für (bisher scheiternde) Induktionen:

- alle freien Variablen (außer Induktionsvariable) mit *arbitrary*
- Induktion immer über das Argument, über das die Funktion rekursiv definiert ist
- Generalisiere zu zeigendes Ziel: Ersetze Konstanten durch Variablen