Teil 3: Termersetzungsbeweise

- Termersetzung: Grundlagen
- Einfache Gleichungen
- Bedingte Gleichungen
- Fallunterscheidungen
- Induktionsbeweise

Im Tutorial: Kap. 3.1, 3.2

Termersetzung: Grundlagen (1)

Grundlage: Gleichungen $l = r$

Langweilig: Ersetzung syntaktisch gleicher Terme:

Wenn $a + b = b$, dann $f\ (a + b) = f\ b$
(a, b konstant)

Interessanter: Ersetzung unter Variablenbelegung:

Sei $a + x = x$ für beliebiges x (a konstant, x Variable)
Dann auch $a + b = b$ (Substitution $\sigma = (x \mapsto b)$)
Also $f\ (a + b) = f\ b$

Allgemein:

- $l = r$ ist *anwendbar* in Term $t[s]$, wenn Substitution σ existiert mit $\sigma(l) = s$
- Ergebnis: Term $t[\sigma(r)]$

Termersetzung: Grundlagen (2)

Beispiel: *Gegeben:* Gleichungen

- (1): $0 + n = n$
- (2): $(Suc\ m) + n = Suc\ (m + n)$
- (3): $(Suc\ m \leq Suc\ n) = (m \leq n)$
- (4): $(0 \leq m) = True$

Ableitung:

$$
\begin{array}{rcll}
0 + Suc\ 0 & \leq & Suc\ 0 + x & (1) \text{ mit } \sigma = (n \mapsto Suc\ 0) \\
Suc\ 0 & \leq & Suc\ 0 + x & (2) \text{ mit } \sigma = (m \mapsto 0, n \mapsto x) \\
Suc\ 0 & \leq & Suc\ (0 + x) & (3) \text{ mit } \sigma = (m \mapsto 0, n \mapsto (0 + x)) \\
0 & \leq & 0 + x & (4) \text{ mit } \sigma = (m \mapsto x) \\
True & & &
\end{array}
$$

Termersetzung: Grundlagen (3)

Instantiierung erfolgt nur in der Gleichung (Matching),
nicht auch im zu ersetzenden Term (Unifikation)

Bsp.: Gleichung: $0 + n = n$

Gültiger Termersetzungsschritt:

$0 + 3 \rightsquigarrow 3$
mit Instantiierung $\sigma = (n \mapsto 3)$

Kein gültiger Termersetzungsschritt:

$(m + 3) + (2 * m) \rightsquigarrow 3 + (2 * 0)$
mit Unifikator $\sigma = (n \mapsto 3, m \mapsto 0)$

Variablen in Isabelle (1)

Drei Arten von Variablen:

- gebundene: ∀ `x. x = x`. In ProofGeneral: grün
- freie: `x = x`. In ProofGeneral: blau
- schematische: `?x = ?x`. In ProofGeneral: blau

Können gemischt in einem Term vorkommen: ∀ `b. f ?a y = b`

Intuition:

- Freie Variablen ≈ universelle Variablen für aktuellen Beweis
  daher unveränderlich
- Schematische Variablen ≈ existentielle Variablen für aktuellen Beweis
  Können während eines Beweises instantiiert werden

Variablen in Isabelle (2)

Nach Abschluß eines Beweises:

done wandelt freie in schematische Variablen um
(erlaubt Instantiierung in späteren Beweisen!)

```
lemma zero_Suc: "0 ≤ Suc n"
  apply auto
done
```

Resultat: `lemma zero_Suc: 0 ≤ Suc ?n`

- Termersetzung: Grundlagen
- Einfache Gleichungen
- Bedingte Gleichungen
- Fallunterscheidungen
- Induktionsbeweise

Simplifikation mit einfachen Gleichungen

Schema:

Gegeben: Gleichungen `eq_1 : l_1 = r_1` ... `eq_n : l_n = r_n`

Beweisziel: $[\![$ `P_1 ... P_m` $]\!] \Longrightarrow$ `C`

Simplifikation

- in `P_1 ... P_m` und `C`
  unter Verwendung von `eq_1 ... eq_n` und der Gleichungen in `P_1 ... P_m`:
  `apply (simp add: eq_1 eq_2 ... eq_n)`
- nur in `C`:
  `apply (simp (no_asm_simp) add: eq_1 eq_2 ... eq_n)`
- ohne Verwendung der Gleichungen in `P_1 ... P_m`:
  `apply (simp (no_asm_use) add: eq_1 eq_2 ... eq_n)`
- Annahmen vollständig ignorieren:
  `apply (simp (no_asm) add: eq_1 eq_2 ... eq_n)`

Simpsets (1)

Sammlung von Termersetzungsregeln, die von `simp` verwendet werden.

Besteht aus:

- Simpsets der importierten Theorien
- Regeln, die von `datatype, primrec` etc. generiert werden
- Explizit vom Benutzer hinzugefügten Regeln

Simpsets (2)

Veränderung des Simpset

- *lokal* für diesen Befehl
  in Taktiken wie `simp` oder `auto`:
  Hinzufügen: *`apply (simp add: eq_1)`*
  Löschen: *`apply (auto simp del: eq_1)`*
  Nur die erwähnten Gleichungen: *`apply (simp only: eq_1)`*

- *global* für alle folgenden Befehle als Attribut von *`lemma`*:
  `lemma eq_1 [simp]: " ... "`

- *global* als Deklaration:
  Hinzufügen: *`declare eq_1 [simp add]`*
  Löschen: *`declare eq_1 [simp del]`*

Simplifikation: Beschränkungen

Instantiierung:
Nicht alle passenden Instantiierungen werden erkannt (HO Unif.!)

lemma `"∀ h x. f x (h x) = c ⟹ f a b = c"`

Terminierung:

Nicht-Terminierung i.a. prinzipiell nicht erkennbar.
Isabelle orientiert Regeln, wo offensichtlich:

lemma `"a = f a ⟹ f (f a) = a"`
by *simp*

Im allgemeinen jedoch in der Verantwortung des Benutzers:

lemma `"⟦ f a = f b; f b = f a ⟧ ⟹ (f a) = c"`

Problem bei *(simp (no_asm_simp))*

Regeln aus Definitionen (1)

Simplifikationsregeln aus nicht-rekursiven Definitionen

constdefs

```
f :: "[nat, nat] ⇒ nat"
"f m n == m + n + 3"
```

thm *f_def*

ergibt: `f ?m ?n ≡ ?m + ?n + 3`

Beachte:

- *f_def* nicht automatisch im Simpset
- Daher: Expansion explizit mit *(simp add: f_def)*
- Alle Argumente erforderlich: vergl. *f 2 3* und *f 2*
- Abhilfe: Definition der Form *f m == λ n. m + n + 3*

Regeln aus Definitionen (2)

Simplifikationsregeln aus Definitionen mit `primrec`

thm *rev.simps*

ergibt:

```
rev [] = []
rev (?x # ?xs) = rev ?xs @ [?x]
```

Beachte:

- Regeln automatisch im Simpset
- Ausschalten mit *(simp del: rev.simps)*

Tracing

Nachvollziehen der Simplifikationsschritte

Einschalten mit:

ML *{* set trace_simp *}*

Ausschalten mit:

ML *{* reset trace_simp *}*

Die Ausgabe erscheint im ProofGeneral-Buffer `*isabelle/isar - reponse*`

Anzeigen verfügbarer Regeln

Gesamtes Simpset: über Menu:
Isabelle/Isar – Help – Show me – simplifier rules

Spezielle Regeln zu einem Funktionssymbol:
Menu-Button "Find" oder `C-c C-f`

! Eingabe von Infix-Symbolen z.B. `"xs @ ys"`

Spezielles Theorem: z.B. `thm append_assoc`

- Termersetzung: Grundlagen
- Einfache Gleichungen
- Bedingte Gleichungen
- Fallunterscheidungen
- Induktionsbeweise

Simplifikation mit bedingten Gleichungen (1)

Bedingte Gleichungen haben die Form $[\![P_1 \ldots P_n]\!] \Longrightarrow l = r$

Die bedingte Gleichung ist *anwendbar* auf Term $t[s]$, wenn

- $l = r$ auf $t[s]$ anwendbar ist mit Substitution σ
- $\sigma(P_1) \ldots \sigma(P_n)$ beweisbar sind

Bsp.:

lemma `"⟦ ∀ x. ((f x) = a) ⟶ (g x) = b; (f c) = a ⟧ ⟹ g c = b"`
by *simp*

Simplifikation mit bedingten Gleichungen (2)

Simplifikation wird rekursiv auf Prämissen angewandt.

Gefahr der Nicht-Termininierung:

lemma *Suc_less: "Suc n < m ⟹ n < m"*
by *simp*

lemma *"Suc n < Suc m"*

apply (simp add: Suc_less) terminiert nicht

Preprocessing

Aussagen können z.T. vollständig durch Simplifier bewiesen werden:

lemma `"⟦ ¬ A ∧ B; A ∨ C ⟧ ⟹ C"`
by *simp*

(i.a. jedoch: Beweise in Aussagen-/ Prädikatenlogik, nächste Stunde!)

Vorverarbeitung:

$$
\begin{array}{rcll}
A & \rightsquigarrow & A = True & \\
\neg A & \rightsquigarrow & A = False & \\
A \longrightarrow B & \rightsquigarrow & A \Rightarrow B & \\
A \wedge B & \rightsquigarrow & A, B & \text{zwei Regeln} \\
\forall x.A[x] & \rightsquigarrow & A[?x] &
\end{array}
$$

Dann Verwendung von Regeln wie $P \wedge True = P$ und $False \vee Q = Q$

Geordnete Termersetzung

Einmalige Termersetzung: `subst`

`simp` kann

- *fehlschlagen* bei geordneter Termersetzung in falscher Richtung: z.B. Anwendung von `m * n = n * m` auf Ziel `(a::nat) * b = c`
- *nicht terminieren*, z.B. Anwendung von `lfp_unfold`:

  `mono f ⟹ lfp f = f (lfp f)`

Einmaliger Termersetzungsschritt:

lemma `"mono h ⟹ lfp h = xxx"`
apply `(subst lfp_unfold)`

```
1. mono h ⟹ mono h
2. mono h ⟹ h (lfp h) = xxx
```

Manipulation des Beweiszustands

Unter-Beweisziele verzögern / vorziehen:

- `defer` macht aktuelles Beweisziel zum letzten Beweisziel
- `prefer` n macht Beweisziel n zum aktuellen

Anzeigen von Typen und Sorten:

- *Problem:* Manche Aussagen nur für bestimmte Typen gültig
- *Bsp.:* Vergleiche `(a + b) + c = a + (b + c)`
  und `((a::nat) + b) + c = a + (b + c)`
- Anzeige (de-)aktivieren:
  im Menu `Isabelle/Isar - Settings - Show Types` bzw. `Show Sorts`

Einfügen lokaler Lemmas

Manche Aussagen sind nur über Umwege zu beweisen
(vergl. "kritische Paare" bei Reduktionssystemen)

`subgoal_tac` (Schnittregel):

- führt lokales Lemma `L` als Prämisse ein
- erzeugt `L` als neues Beweisziel

```
lemma "⟦ ∀ x y z. f (f x y) z = f x (f y z); ∀ x. f x (g x) = n ⟧
  ⟹ f a (f b (g (f a b))) = n"
  apply (subgoal_tac "n = f (f a  b) (g (f a b))")
  apply simp        — original goal
  apply simp        — new subgoal
done
```

- Termersetzung: Grundlagen
- Einfache Gleichungen
- Bedingte Gleichungen
- Fallunterscheidungen
- Induktionsbeweise

Fallunterscheidungen (1)

mit `split`

Für *if*: Unterscheidung nach der Bedingung

Bsp.:

lemma *"(A ∧ B) = (if A then B else False)"*
apply *(split split_if)*

ergibt: *(A ⟶ (A ∧ B) = B) ∧ (¬ A ⟶ (A ∧ B) = False)*

split_if wird von Simplifier standardmäßig angewandt.

Split in Prämissen: mit *split_if_asm*

Fallunterscheidungen (2)

Allgemein: Jeder Datentyp `t` hat Split-Regeln `t.split` und `t.split_asm`

$$P\ (nat_case\ f1\ f2\ x) = ((x = 0 \longrightarrow P\ f1) \wedge (\forall nat.\ x = Suc\ nat \longrightarrow P\ (f2\ nat)))$$

Bsp.: Vorgängerfunktion:

```
constdefs
  pred :: "nat ⇒ nat"
  "pred n == (case n of 0 ⇒ 0 | Suc n' ⇒ n')"

lemma "pred n ≤ n"
by (simp add: pred_def split add: nat.split)
```

- Termersetzung: Grundlagen
- Einfache Gleichungen
- Bedingte Gleichungen
- Fallunterscheidungen
- Induktionsbeweise

Induktionsbeweise (1)

Gegeben: `rev` mit Accumulator:

```
consts
  rev_acc :: "['a list, 'a list] ⇒ 'a list"
primrec
  "rev_acc [] ys = ys"
  "rev_acc (x#xs) ys = (rev_acc xs (x#ys))"
```

Zeige:

```
lemma "rev_acc xs [] = rev xs"
apply (induct xs)
apply simp      — base case
apply simp      — step case
```

Ergibt:

```
1. ⋀a list. rev_acc list [] = rev list ⟹ rev_acc list [a] = rev list @ [a]
```

Induktionsbeweise (2)

Problem: Prämisse nicht anwendbar.

Zweiter Versuch: Verallgemeinerung

lemma *"rev_acc xs ys = (rev xs) @ ys"*
apply *(induct xs)*
apply *simp* — base case
apply *simp* — step case

Ergibt:

```
1. ⋀a list.
      rev_acc list ys = rev list @ ys ⟹
      rev_acc list (a # ys) = rev list @ a # ys
```

Induktionsbeweise (3)

Problem: Festes `ys`. Daher: All-Quantifizierung:

```
lemma rev_acc_rev: "∀ ys. rev_acc xs ys = (rev xs) @ ys"
apply (induct xs)
apply simp      — base case
apply (intro strip)
apply (simp (no_asm_use))

apply simp      — step case
done
```

Damit ursprünglicher Beweis:

```
lemma "rev_acc xs [] = rev xs"
by (simp add: rev_acc_rev)
```

Erzeugen des Regelformats