

# Praktikum Spezifikation und Verifikation

Stefan Berghofer, Martin Strecker  
basierend auf Material von Tobias Nipkow

1. Überblick; Simplifikations- und Induktionsbeweise
2. Datentypdefinitionen und primitive Rekursion
3. Termersatzbeweise
4. Aussagenlogik; Rückwärtsbeweise
5. Prädikatenlogik; Mengenlehre
6. Vorwärtsbeweise; induktiv definierte Mengen
7. Taktiken

# Teil 1: Überblick

- Motivation
- Organisatorisches
- Überblick Isabelle
- Überblick Isabelle/HOL
- Primitive Rekursion; Beweise mit Induktion und Termersetzung

## Isabelle: Architektur

- *ProofGeneral*: Arbeitsumgebung, basierend auf (X)Emacs
- *Isabelle/HOL*: Ausprägung von Isabelle für HOL
- *Isabelle*: Generischer Theorembeweiser
- *Standard ML*: Implementierungsebene

## Isabelle: Theorien

**Theorien** sind Zusammenfassungen von Typen, Funktionen, Theoremen, ...

**Syntax:**

```
theory MyTh = ImpTh_1 + ... + ImpTh_n:  
  {Deklarationen, Definitionen, Theoreme, Beweise}  
end
```

- MyTh ist die gerade definierte Theorie (in Datei MyTh.thy)
- die ImpTh\_i sind importierte Theorien (Import hierarchisch)

Im folgenden meistens ausreichend:

```
theory MyTh = Main:  
  ...
```

# ProofGeneral

Erweiterung zu **XEmacs**:

- Editierkommandos wie in Emacs (Info: **C-h i**)
- Erweitert um X-Symbol (insbes. mathematische Symbole)

**Aufbau:**

- Arbeits-Buffer: `*.thy`
- Ausgabe-Buffer: `*isabelle/isar-goals*`, `-trace*`, `-response*`

**Navigation:**

- über Menu
- per Tastenkombination (Key-Bindings siehe **C-h m**)

- Motivation
- Organisatorisches
- Überblick Isabelle
- Überblick Isabelle/HOL
- Primitive Rekursion; Beweise mit Induktion und Termersetzung

# Isabelle/HOL

HOL = Higher Order Logic, bestehend aus:

- Funktionaler Programmiersprache
- Prädikatenlogik:  $\exists n. 0 < n$
- “höherer Ordnung”: erlaubt Quantifizierung über Funktionen:  
 $\forall f. f\ x = x$

## Isabelle/HOL: Formeln (1)

**Syntax:** in abnehmender Bindungsstärke:

$$\begin{array}{l}
 \text{form} ::= (\text{form}) \\
 \quad | \text{form} = \text{form} \\
 \quad | \neg \text{form} \\
 \quad | \text{form} \wedge \text{form} \quad | \quad \text{form} \vee \text{form} \quad | \quad \text{form} \longrightarrow \text{form} \\
 \quad | \forall x. \text{form} \quad | \quad \exists x. \text{form}
 \end{array}$$

**Skopus:**

“Weitestmöglich nach rechts” (bis zur nächsten schließenden Klammer)

**Bsp.:**

- $\neg A \wedge B \vee C \equiv ((\neg A) \wedge B) \vee C$
- $A = B \wedge C \equiv (A = B) \wedge C$
- $\forall x. F x \wedge G x \equiv \forall x. (F x \wedge G x)$
- $\forall x. \exists y. G x y \wedge F x \equiv \forall x. (\exists y. G x y \wedge F x)$   
 $\not\equiv \forall x. (\exists y. G x y) \wedge F x$  (zumindest syntaktisch!)

## Isabelle/HOL: Formeln (2)

### Abkürzungen:

$\forall x. \forall y. F\ x\ y$  entspricht  $\forall x\ y. F\ x\ y$ , ebenso für  $\exists$ ,  $\lambda$ , ...

### Überdeckung äußerer Bindungen durch innere Bindungen:

$\forall x\ y. (\forall x. F\ x\ y) \wedge G\ x\ y \equiv \forall x_0\ y. (\forall x_1. F\ x_1\ y) \wedge G\ x_0\ y$

### Assoziativität:

- $\wedge$  und  $\vee$  und  $\longrightarrow$  sind rechts-assoziativ.

Insbesondere:  $A \longrightarrow B \longrightarrow C \equiv A \longrightarrow (B \longrightarrow C) \not\equiv (A \longrightarrow B) \longrightarrow C$

- $=$  ist links-assoziativ

## Isabelle/HOL: Formeln (3)

### Eingabe in ProofGeneral

- über Menu X-Symbol
- (meistens) entsprechend der LaTeX-Syntax, z.B.: `\<forall>` bzw. `\<exists>`
- als `/\` bzw. `\/`

### ASCII-Schreibweise:

XSymbol	$\forall$	$\exists$	$\lambda$	$\neg$	$\wedge$	$\vee$	$\longrightarrow$	$\Rightarrow$
ASCII	<code>\&lt;forall&gt;</code>	<code>\&lt;exists&gt;</code>	<code>\&lt;lambda&gt;</code>	<code>\&lt;not&gt;</code>	<code>/\</code>	<code>\/</code>	<code>--&gt;</code>	<code>=&gt;</code>
ASCII(alt)	ALL	EX	%	~	&		-->	=>

# Isabelle/HOL: Typen

## Syntax:

$\tau$	$::=$	$(\tau)$	
		$bool$	$nat$ “Vordefinierte” Basistypen
		$'a$	$'b$ Typvariablen
		$\tau \Rightarrow \tau$	Funktionsstyp (entspr. $\rightarrow$ in ML)
		$\tau \times \tau$	Paar-Typ (altern. $*$ )
		$\tau list$	Listen-Typ
		$\dots$	benutzerdefinierte Typen

## Abkürzungen:

$$[T1, T2] \Rightarrow T3 \quad \equiv \quad T1 \Rightarrow T2 \Rightarrow T3 \quad \equiv \quad T1 \Rightarrow (T2 \Rightarrow T3)$$

## Isabelle/HOL: Terme (1)

### Syntax:

term	::=	(term)	
		c	v Konstante bzw. Variable
		term term	Funktionsanwendung
		$\lambda x. \text{term}$	Abstraktion ( $fn\ x \Rightarrow term$ in ML)
		...	

Bsp.:  $f\ (g\ x)\ y$                        $h\ (\lambda\ x.\ f\ (g\ x))$

Assoziativität:  $f\ a_1\ a_2\ a_3 \equiv ((f\ a_1)\ a_2)\ a_3$

### Syntaktischer Zucker:

- *Infix*: Vordefiniert z.B.  $+$ ,  $-$ ,  $*$ , ...  
binden schwächer als Präfix:  $(f\ x) + y \equiv f\ x + y \not\equiv f\ (x + y)$
- *Mixfix*: Ausdrücke der Art  $\Gamma \vdash x :: T$  statt  $deriv(G, x, T)$

## Isabelle/HOL: Terme (2)

$\lambda$ -Kalkül in 3 Zeilen:

- *Abstraktion:*  
 $\lambda x.t[x]$  ist anonyme Funktion mit formalem Parameter  $x$  und Rumpf  $t[x]$
- *Funktionsanwendung:*  
 $f a$  entspricht Aufruf von  $f$  mit Argument  $a$
- *Berechnung:*  
 Ersetzen des formalen durch aktuellen Parameter ( $\beta$ -Kontraktion):  
 $(\lambda x.t[x]) a \longrightarrow_{\beta} t[a]$

**Bsp.:**  $(\lambda x. x + 5) 3 \longrightarrow_{\beta} (3 + 5)$

## Isabelle/HOL: Terme (3)

Curry: ja oder nein?

- *ja*:  $f\ a_1\ a_2\ a_3$ 
  - Funktion mit *drei* Argumenten
  - Typ:  $f : [A_1, A_2, A_3] \Rightarrow T$ , d.h.  $f : A_1 \Rightarrow A_2 \Rightarrow A_3 \Rightarrow T$
  - Partielle Anwendung leicht, z.B.  $f\ a_1\ a_2 : A_3 \Rightarrow T$
  - Bevorzugen, wenn möglich
  
- *nein*:  $f\ (a_1, a_2, a_3)$ 
  - Funktion mit *einem* Argument:  $(a_1, a_2, a_3)$
  - Typ:  $f : (A_1 \times A_2 \times A_3) \Rightarrow T$
  - Partielle Anwendung umständlich, z.B.  
 $(\lambda x. f\ (a_1, a_2, x)) : A_3 \Rightarrow T$
  - Möglichst vermeiden

## Isabelle/HOL: Basistypen (1)

**bool**: Typ der Aussagen / Formeln:

*True* :: *bool*

*False* :: *bool*

*op*  $\wedge$  :: [*bool*, *bool*]  $\Rightarrow$  *bool*

*All* :: ('a  $\Rightarrow$  *bool*)  $\Rightarrow$  *bool*, wobei *All*  $(\lambda x. P x) \equiv \forall x. P x$

## Isabelle/HOL: Basistypen (2)

**nat**: Typ der natürlichen Zahlen:

$0 :: \text{nat}$

$\text{Suc} :: \text{nat} \Rightarrow \text{nat}$

$+, *, \dots :: [\text{nat}, \text{nat}] \Rightarrow \text{nat}$

! Zahlsymbole und Operatoren sind überladen:

$1 :: 'a, \quad (\text{op } +) :: 'a \Rightarrow 'a \Rightarrow 'a$

Daher mit Typen annotieren:  $1 :: \text{nat}, 1 :: \text{int}$

... soweit nicht aus Kontext klar:  $\text{Suc } 1$

- Motivation
- Organisatorisches
- Überblick Isabelle
- Überblick Isabelle/HOL
- Primitive Rekursion; Beweise mit Induktion und Termersetzung

## Funktions-Definitionen: Überblick

**Allgemein:** Alle in Isabelle/HOL definierbaren Funktionen sind terminierend.

*Denn:* Was geschähe, wenn  $f\ 3$  und  $g\ 5$  nicht terminieren:

- Gilt  $f\ 3 = g\ 5$  ??
- Was ist  $h\ (f\ 3)$  für  $h\ x \equiv x + 2$  ??  
Gilt dies auch für  $h\ x \equiv 1$  ??

**Definitionsschemata:**

- Nicht-rekursiv mit *defs*
- Primitiv-rekursiv mit *primrec*
- Wohlfundiert mit *recdef* ( $\rightsquigarrow$  später!)

## Funktionsdefinitionen: Nicht-rekursiv

bestehend aus ...

### Deklaration:

#### **consts**

```
plus2 :: "nat => nat"
```

### Definition:

#### **defs**

```
plus2_def: "plus2 n == n + 2"
```

### Kürzer:

#### **constdefs**

```
plus3 :: "nat => nat"  
"plus3 n == n + 3"
```

## Funktionsdefinitionen: Primitiv rekursiv

Erweitert Schema der prim. Rek. über *nat* aus der Rekursionstheorie

**Basis:** Induktiver Datentyp, z.B.

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

**Deklaration:**

**consts**

```
app :: "'a list => 'a list => 'a list"
```

**Definition:** Eine Gleichung pro Konstruktor:

**primrec**

```
"app []      ys = ys"
"app (x#xs)  ys = x#(app xs ys)"
```

# Beweise (1)

## Allgemeines Schema

```
lemma "..."  
  {apply (...)}*  
done
```

### *Varianten:*

- Statt *lemma* auch *theorem*
- Für spätere Referenz: Benennung des Lemmas

### Bsp.:

```
lemma app_assoc: "app (app xs ys) zs = app xs (app ys zs)"  
  apply (induct xs)  
  apply auto  
done
```

## Beweise (2)

Abkürzung bei einfachen Kommandos:

```
lemma "app (app xs ys) zs = app xs (app ys zs)"  
  by (simp add: app_assoc)
```

Abbrechen von Beweisen:

- *sorry*: Aussage wird in Lemmabasis eingefügt.  
Nur im “Quick-and-dirty”-Modus
- *oops*: Aussage nicht als Lemma verwendbar.

## Elementare Beweiskommandos (1)

### simp

- Format: *simp*
- Effekt: Reduziert akt. Beweisziel mit Gleichungen aus Simpset
- Erweiterung: (*simp add: rule1 rule2 ..*):  
Fügt zusätzlich *rule1*, *rule2* zum Simpset hinzu

### auto

- Format: *auto*
- Effekt: *simp* und prädikatenlogische Beweise und ...  
Wirkt auf *alle* Beweisziele
- Erweiterung: (*auto simp add: rule1 rule2 ..*)

## Elementare Beweiskommandos (2)

### induct

- Format: (*induct*  $v$ ), wobei  $v$  freie Variable induktiven Typs im akt. Beweisziel
- Effekt: Erzeugt je ein neues Beweisziel pro Konstruktor

## Teil 2: Datentypdefinitionen und primitive Rekursion

- Isabelle's Meta-Logik
- Datentypdefinitionen
- Primitive Rekursion: mehr Details
- Vordefinierte Typen und Terme

Im Tutorial: Bis Ende Kap. 2

## Isabelle's Meta-Logik

... erlaubt die Codierung von Objekt-Logiken, u.a. Isabelle/HOL.

... scheint durch bei Theoremen/Lemmas bzw. Beweiszuständen:

$$\bigwedge x_1 \dots x_n. \llbracket A_1; \dots ; A_n \rrbracket \Longrightarrow B$$

“Unter Prämissen  $A_1 \dots A_n$  zeige Beweisziel  $B$ ”

Aussagen der Meta-Logik werden gebildet durch:

- $\Longrightarrow$  Implikation (ASCII:  $\implies$ )
- $\bigwedge$  All-Quantor (ASCII:  $!!$ )
- $\equiv$  Gleichheit (ASCII:  $==$ )

Abkürzung:  $\llbracket P_1; \dots ; P_n \rrbracket \Longrightarrow P$

statt  $P_1 \Longrightarrow (\dots \Longrightarrow (P_n \Longrightarrow P))$

- Isabelle's Meta-Logik
- Datentypdefinitionen
- Primitive Rekursion: mehr Details
- Vordefinierte Typen und Terme

# Datentyp-Deklarationen und -Definitionen

Datentyp-Deklarationen mit *typedecl*, z.B.:

**typedecl** *varname*

“Es gibt einen Typ *varname*, aber seine interne Struktur ist unwichtig”

Datentyp-Definitionen durch eine der folgenden:

- *types*: Einführen eines Typnamens
- *datatype*: Konstruktor-generierte Typen
- *typedef*: Typen aus Mengen ( $\leadsto$  später!)

## Datentypdefinitionen mit `types` (1)

Mit `types` wird ein Typname eingeführt:

```
types nat_varname = nat  
types nat_pair = "nat × nat"  
types ('a, 'b) list_pair = "'a list × 'b list"
```

**!Syntax:** Komplexe Ausdrücke in Hochkomma!

Durch `types`

- wird nur eine Abkürzung eingeführt
- ... die von Isabelle intern expandiert wird  
(Unterschied zu `defs` auf Termebene!)
- *Konsequenz 1:* Typnamen in Fehlermeldungen etc. nicht mehr vorhanden

## Datentypdefinitionen mit `types` (2)

*Konsequenz 2: Strukturelle Typgleichheit*

*Vergleiche:*

```
types nat_triple = "nat × nat × nat"
```

```
term "(1, (2, 3)) :: nat_triple"
```

```
term "(1, ((2, 3)::nat_pair)) :: nat_triple"
```

Ausgabe in beiden Fällen: `"(1, 2, 3)" :: "nat × nat × nat"`

*Jedoch:*

```
typedecl varname2
```

```
term "(x::varname) = (y::varname)"   ergibt   "x = y"   :: "bool"
```

```
term "(x::varname) = (y::varname2)"   ergibt Fehler!
```

## Datentypdefinitionen mit datatype (1)

### Beispiel Listen:

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

### Insbesondere:

- $Nil :: 'a list$
- Wenn  $x :: 'a$  und  $xs :: 'a list$ , dann auch  $Cons x xs :: 'a list$
- Listen können nur mit  $Nil$  und  $Cons$  gebildet werden

### Schreibweise:

$[]$  statt  $Nil$ ;  $x\#xs$  statt  $Cons x xs$

$[a1,a2,a3]$  statt  $Cons a1 (Cons a2 (Cons a3 Nil))$

## Datentypdefinitionen mit datatype (2)

Allgemein:

$$\text{datatype } (\alpha_1 \dots \alpha_n) \tau = \begin{array}{l} C_1 \tau_{1,1} \dots \tau_{1,n_1} \\ | \dots \\ C_k \tau_{k,1} \dots \tau_{k,n_k} \end{array}$$

Für die Konstruktoren  $C_i$  gilt:

- *Typisierung*:  $C_i :: [\tau_{i,1} \dots \tau_{i,n_i}] \Rightarrow (\alpha_1 \dots \alpha_n) \tau$
- *Disjunktheit*: für  $i \neq j$  ist  $C_i \neq C_j$   
Also:  $\text{Nil} \neq \text{Cons } x \text{ } xs$
- *Injektivität*:  $C_i x_1 \dots x_{n_i} = C_i y_1 \dots y_{n_i} \longrightarrow x_1 = y_1 \wedge \dots \wedge x_{n_i} = y_{n_i}$   
Also:  $\text{Cons } x \text{ } xs = \text{Cons } y \text{ } ys \longrightarrow x = y \wedge xs = ys$

## Datentypdefinitionen mit datatype (3)

**Einschränkungen:** (werden von Isabelle überprüft!)

Typdefinition muß *produktiv* sein. Gegenbsp.:

```
datatype emptytp = E emptytp
```

Also: Elemente von *emptytp* haben die Form  $E (E (E \dots))$   
aber womit anfangen??

Zu konstruierender Typ darf nur *strikt positiv* vorkommen. Gegenbsp.:

```
datatype cantor = N | C "cantor  $\Rightarrow$  bool"
```

*Problem:*

Angenommen, *cantor* hat  $A$  Elemente.

*cantor  $\Rightarrow$  bool* hat dann  $2^A$  Elemente.

Dann kann  $C :: (\text{cantor} \Rightarrow \text{bool}) \Rightarrow \text{cantor}$  nicht injektiv sein.

- Isabelle's Meta-Logik
- Datentypdefinitionen
- **Primitive Rekursion: mehr Details**
- Vordefinierte Typen und Terme

## Primitiv-rekursive Definitionen (1)

Zu definieren sei Fkt.  $f :: [\tau_1 \dots \tau_i \dots \tau_m] \Rightarrow \tau$   
wobei  $\tau_i$  ein Datentyp ist.

**Schema** der primitiven Rekursion:

$$\begin{aligned} f \ x_1 \dots (C_1 \ y_{1,1} \dots y_{1,n_1}) \dots x_p &= r_1 \\ \dots & \\ f \ x_1 \dots (C_k \ y_{k,1} \dots y_{k,n_k}) \dots x_p &= r_k \end{aligned}$$

Hierbei

- sind die  $C_i$  die Konstruktoren von  $\tau_i$  und  $x_1, \dots, y_{1,1} \dots$  Variablen
- müssen in den  $r_i$  Aufrufe von  $f$  strukturell kleiner werden, d.h. nur von der Form  $f \ a_1 \dots y_{i,j} \dots a_p$  sein
- ist normalerweise  $p = m$ ,  
aber auch  $p < m$  (partielle Applikation) oder  $p > m$  (Funktionstypen)

## Primitiv-rekursive Definitionen (2)

- Die gewählte Argumentposition muß bei jedem rek. Aufruf *strukturell* kleiner werden. Nicht:

```
"foo (Suc n) = ... foo (n - 3) ..."
```

- Nicht mehrere oder verschachtelte Pattern:

```
"bar (Suc n) (Suc (Suc m)) =  
  if ... then (bar n (m+5)) else (bar (n+3) m)"
```

↪ wohlfundierte Rekursion mit `recdef`

- Höchstens eine Gleichung pro Konstruktor.  
Jedoch: Konstruktorgleichung für ein  $C_j$  darf fehlen. Entspricht:  
 $f\ x_1 \dots (C_j\ y_{j,1} \dots y_{j,n_j}) \dots x_p = \textit{arbitrary}$

## Bemerkungen zu Definitionen

... sowohl für `defs` als auch für `primrec`:

- Keine freien Variablen auf der rechten Seite:

Inkorrekt:

```
leq10_def: "leq10 n == n + d = 10"
```

Korrekt mit Quantoren:

```
leq10_def: "leq10 n == ∃ d. n + d = 10"
```

- Logisch gleichwertig:

$f\ x\ y ==\ x + y + 2$  und

$f\ x ==\ \lambda\ y. x + y + 2$

Jedoch: Kleiner operationeller Unterschied bei Simplifikation,  
z.B.  $f\ 2$

- Isabelle's Meta-Logik
- Datentypdefinitionen
- Primitive Rekursion: mehr Details
- Vordefinierte Typen und Terme

## Datentyp option

emuliert partielle Funktionen:

```
datatype 'a option = None | Some 'a
```

Interpretation:

- *None*: undefiniertes Ergebnis
- *Some e*: definiertes Ergebnis *e*

*Beispiel*: Suchen in Assoziationsliste

**consts**

```
lookup :: "[ 'k, ('k × 'v) list ] ⇒ 'v option"
```

**primrec**

```
"lookup k [] = None"
```

```
"lookup k (x#xs) =
```

```
  (if (fst x) = k then Some (snd x) else lookup k xs)"
```

## Fallunterscheidungen

Für jeden Datentyp gibt es ein `case`-Konstrukt.

z.B. für Listen:

```
"case t of  
  []      ⇒ t1  
  | x # xs ⇒ t2"
```

**Allgemein:** Ein Pattern pro Konstruktor

**Speziell:** `case` für `bool`:

`if b then t1 else t2`, wobei `b :: bool` und `t1, t2 :: 'a`

# let

führt lokale Abkürzung ein:

```
"(let x1 = t1;  
      x2 = t2;  
      x3 = t3  
      in (f x1 x2 x3))"
```

Ist logisch äquivalent zu

```
"(λ x1 x2 x3. (f x1 x2 x3)) t1 t2 t3"
```

Es gelten die üblichen Regeln (Sichtbarkeit/Überdeckung) für Variablen-Bindungen

## 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  $\sigma$  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:*

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

## 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:  $\forall x. x = x$ . In ProofGeneral: grün
- freie:  $x = x$ . In ProofGeneral: blau
- schematische:  $?x = ?x$ . In ProofGeneral: blau

Können gemischt in einem Term vorkommen:  $\forall b. f ?a y = b$

### Intuition:

- Freie Variablen  $\approx$  universelle Variablen für aktuellen Beweis  
daher unveränderlich
- Schematische Variablen  $\approx$  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  $eq_1 : l_1 = r_1 \quad \dots \quad eq_n : l_n = r_n$

Beweisziel:  $\llbracket P_1 \dots P_m \rrbracket \implies C$

### Simplifikation

- in  $P_1 \dots P_m$  und  $C$   
unter Verwendung von  $eq_1 \dots eq_n$  und der Gleichungen in  $P_1 \dots P_m$ :  
`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  $P_1 \dots P_m$ :  
`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** " $\forall h x. f x (h x) = c \implies f a b = c$ "

### Terminierung:

Nicht-Terminierung i.a. prinzipiell nicht erkennbar.

Isabelle orientiert Regeln, wo offensichtlich:

**lemma** " $a = f a \implies f (f a) = a$ "

**by** *simp*

Im allgemeinen jedoch in der Verantwortung des Benutzers:

**lemma** " $\llbracket f a = f b; f b = f a \rrbracket \implies (f a) = c$ "

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:  $f \ ?m \ ?n \equiv \ ?m + \ ?n + 3$

#### 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  $f \ m \ == \ \lambda \ n. \ m + n + 3$

## 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  $\llbracket P_1 \dots P_n \rrbracket \Longrightarrow l = r$

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

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

Bsp.:

**lemma** " $\llbracket \forall x. ((f\ x) = a) \longrightarrow (g\ x) = b; (f\ c) = a \rrbracket \Longrightarrow g\ c = b$ "

by *simp*

## Simplifikation mit bedingten Gleichungen (2)

Simplifikation wird rekursiv auf Prämissen angewandt.

*Gefahr* der Nicht-Terminierung:

```
lemma Suc_less: "Suc n < m  $\implies$  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** " $\llbracket \neg A \wedge B; A \vee C \rrbracket \implies C$ "

**by** *simp*

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

**Vorverarbeitung:**

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

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

# 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*:

$$\text{mono } f \implies \text{lfp } f = f (\text{lfp } f)$$

### Einmaliger Termersetzungsschritt:

**lemma** " $\text{mono } h \implies \text{lfp } h = \text{xxx}$ "

**apply** (subst lfp\_unfold)

1.  $\text{mono } h \implies \text{mono } h$
2.  $\text{mono } h \implies h (\text{lfp } h) = \text{xxx}$

## 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 "[[  $\forall x y z. f (f x y) z = f x (f y z); \forall x. f x (g x) = n ]]$ 
   $\implies f a (f b (g (f a b))) = n$ "
  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:  $(A \longrightarrow (A \wedge B) = B) \wedge (\neg A \longrightarrow (A \wedge B) = \text{False})$

`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  $t.split\_asm$

$$P (\text{nat\_case } f1 \ f2 \ x) = ((x = 0 \longrightarrow P \ f1) \wedge (\forall \text{ nat. } x = \text{Suc } \text{nat} \longrightarrow P \ (f2 \ \text{nat})))$$

**Bsp.:** Vorgängerfunktion:

**constdefs**

```
pred :: "nat  $\Rightarrow$  nat"
```

```
"pred n == (case n of 0  $\Rightarrow$  0 | Suc n'  $\Rightarrow$  n')"
```

**lemma** "pred n  $\leq$  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.  $\bigwedge a \text{ list. } rev\_acc \text{ list } [] = rev \text{ list} \implies rev\_acc \text{ list } [a] = rev \text{ list } @ [a]$

## 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.  $\bigwedge a \text{ list.}$

$rev\_acc \text{ list } ys = rev \text{ list } @ ys \implies$

$rev\_acc \text{ list } (a \# ys) = rev \text{ list } @ a \# ys$

## 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

## Teil 4: Logik

- Regeln der Aussagenlogik
- Regelanwendung (vorwärts / rückwärts)
- Sichere / unsichere Regeln

Im Tutorial: Kap. 5

## HOL - die Logik

Klassische Prädikatenlogik mit folgenden Besonderheiten:

- *Höherer Ordnung (HOL):*  
Quantifizierung über Funktionen/Prädikate:  $\exists f. \forall x. f\ x = x$
- *Formeln vom Typ bool:*  
Gleichheit (=) und Äquivalenz ( $\leftrightarrow$ ) werden gleichgesetzt
- *Auswahloperator:*  
 $\varepsilon x. P\ x$  (ASCII: *SOME*  $x.P\ x$ ) bedeutet: Ein  $x$ , das  $P$  erfüllt.

## Regeln des Natürlichen Schließens

Notation:

Statt  $R: \llbracket A_1 \dots A_n \rrbracket \Longrightarrow A$  schreibe:  $\frac{A_1 \dots A_n}{A} R$

Erlaubt nachvollziehbare Darstellung von **Ableitungen**,  
z.B. von  $\llbracket A \wedge B; C \rrbracket \Longrightarrow A \wedge (C \vee D)$

$$\frac{\frac{A \wedge B}{A} \text{conjunct1} \quad \frac{C}{C \vee D} \text{disjI1}}{A \wedge (C \vee D)} \text{conjI}$$

## Regeln der Aussagenlogik

Einführungsregeln (*Intro*):

$$\frac{A \quad B}{A \wedge B} \text{ conjI}$$

$$\frac{A}{A \vee B} \text{ disjI1} \quad \frac{B}{A \vee B} \text{ disjI2}$$

$$\frac{A \implies B}{A \longrightarrow B} \text{ impI}$$

$$\frac{A \implies B \quad B \implies A}{A = B} \text{ iffI}$$

$$\frac{A \implies \text{False}}{\neg A} \text{ notI}$$

$$\frac{}{\text{True}} \text{ TrueI}$$

$$\frac{}{t = t} \text{ refl}$$

Beseitigungsregeln (*Elim*):

$$\frac{A \wedge B \quad [[A; B]] \implies C}{C} \text{ conjE}$$

$$\frac{A \vee B \quad A \implies C \quad B \implies C}{C} \text{ disjE}$$

$$\frac{A \longrightarrow B \quad A \quad B \implies C}{C} \text{ impE}$$

$$\frac{A = B}{A \implies B} \text{ iffD1} \quad \frac{A = B}{B \implies A} \text{ iffD2}$$

$$\frac{A \quad \neg A}{C} \text{ notE}$$

$$\frac{\text{False}}{C} \text{ FalseE}$$

$$\frac{s = t \quad P(s)}{P(t)} \text{ subst}$$

## Aussagenlogik: Weitere Regeln (1)

Strukturell:  $\frac{A_1 \dots A \dots A_n}{A}$  assumption

Weitere Regeln (z.T. abgeleitet):

$$\frac{A \wedge B}{A} \text{ conjunct1} \quad \frac{A \wedge B}{B} \text{ conjunct2}$$

$$\frac{A \longrightarrow B \quad A}{B} \text{ mp}$$

$$\frac{\neg A \Longrightarrow A}{A} \text{ classical} \quad \frac{\neg A \Longrightarrow \text{False}}{A} \text{ ccontr}$$

$$\frac{s = t}{t = s} \text{ sym} \quad \frac{r = s \quad s = t}{r = t} \text{ trans}$$

## Aussagenlogik: Weitere Regeln (2)

*Beachte:* Prämissen der Form  $\llbracket \dots \rrbracket \implies C$  entsprechen Teilbeweisen

*Ableitung* von conjunct1 aus conjE und assumption:

$$\frac{A \wedge B \quad \llbracket A; B \rrbracket \implies A}{A} \text{ conjE}$$

mit Teilbeweis  $\llbracket A; B \rrbracket \implies A$ :

$$\frac{A \quad B}{A} \text{ assumption}$$

- Regeln der Aussagenlogik
- **Regelanwendung (vorwärts / rückwärts)**
- Sichere / unsichere Regeln

## Regelanwendung in Isabelle: Überblick

Regelanwendung mit:

- `rule`: Vorwärtsschließen
- `frule`: Rückwärtsschließen – Prämisse bleibt erhalten
- `drule`: Rückwärtsschließen – Prämisse wird gelöscht
- `erule`: Kombination von `rule` und `drule`

## Regelanwendung in Isabelle: rule (1)

Vorwärtsschließen: apply (rule R)

Gegeben: Regel  $R$ :  $\llbracket A_1 \dots A_n \rrbracket \Longrightarrow A$

Aktuelles Beweisziel:  $\llbracket B_1 \dots B_m \rrbracket \Longrightarrow B$

Bedingung:  $A \equiv B$  (vorerst: “syntaktisch gleich”)

Neue Beweisziele:

$\llbracket B_1 \dots B_m \rrbracket \Longrightarrow A_1$

...

$\llbracket B_1 \dots B_m \rrbracket \Longrightarrow A_n$

warum korrekt?

## Regelanwendung in Isabelle: rule (2)

Regel  $R$ :  $\llbracket A_1 \dots A_n \rrbracket \Longrightarrow A$   
 mit  $A \equiv B$

Beweisziele:

$$\llbracket B_1 \dots B_m \rrbracket \Longrightarrow B \quad \rightsquigarrow \quad \begin{array}{l} \llbracket B_1 \dots B_m \rrbracket \Longrightarrow A_1 \\ \dots \\ \llbracket B_1 \dots B_m \rrbracket \Longrightarrow A_n \end{array}$$

Beweisbaum:

$$\frac{B_1 \dots B_m}{B} \quad \rightsquigarrow \quad \frac{\frac{B_1 \dots B_m}{A_1} \quad \dots \quad \frac{B_1 \dots B_m}{A_n}}{B} R$$

## Regelanwendung in Isabelle: frule / drule (1)

**Rückwärtsschließen:** apply (frule R) bzw. apply (drule R)

*Gegeben:* Regel  $R$ :  $\llbracket A_1 \dots A_n \rrbracket \Longrightarrow A$

*Aktuelles Beweisziel:*  $\llbracket B_1 \dots B_m \rrbracket \Longrightarrow B$

*Bedingung:*  $A_1 \equiv B_i$

*Neue Beweisziele:*

$\llbracket B_1 \dots B_i^? \dots B_m \rrbracket \Longrightarrow A_2$

...

$\llbracket B_1 \dots B_i^? \dots B_m \rrbracket \Longrightarrow A_n$

$\llbracket B_1 \dots B_i^? \dots B_m, A \rrbracket \Longrightarrow B$

*hierbei:*

- $B_i$  bleibt erhalten bei **frule**
- $B_i$  wird gelöscht bei **drule**

## Regelanwendung in Isabelle: frule / drule (2)

Regel  $R$ :  $\llbracket A_1 \dots A_n \rrbracket \Longrightarrow A$   
 mit  $A_1 \equiv B_i$

Beweisziele:

$$\llbracket B_1 \dots B_m \rrbracket \Longrightarrow B \quad \rightsquigarrow \quad \begin{array}{l} \llbracket B_1 \dots B_i^? \dots B_m \rrbracket \Longrightarrow A_2 \\ \dots \\ \llbracket B_1 \dots B_i^? \dots B_m \rrbracket \Longrightarrow A_n \\ \llbracket B_1 \dots B_i^? \dots B_m, A \rrbracket \Longrightarrow B \end{array}$$

Beweisbaum: (mit  $\mathcal{B} = B_1 \dots B_i^? \dots B_m$ )

$$\frac{B_1 \dots B_m}{B} \quad \rightsquigarrow \quad \frac{\frac{B_1 \dots B_i \dots B_m}{A_1} \text{ ass.} \quad \frac{\mathcal{B}}{A_2} \quad \dots \quad \frac{\mathcal{B}}{A_n}}{A} R}{B}$$

## Regelanwendung in Isabelle: erule

Kombiniertes Vorwärts- / Rückwärtsschließen: apply (erule R)

*Gegeben:* Regel  $R$ :  $\llbracket A_1 \dots A_n \rrbracket \Longrightarrow A$

*Aktuelles Beweisziel:*  $\llbracket B_1 \dots B_m \rrbracket \Longrightarrow B$

*Bedingung:*  $A_1 \equiv B_i$  und  $A \equiv B$

*Neue Beweisziele:*

$\llbracket B_1 \dots B_{i-1}, B_{i+1} \dots B_m \rrbracket \Longrightarrow A_2$

...

$\llbracket B_1 \dots B_{i-1}, B_{i+1} \dots B_m \rrbracket \Longrightarrow A_n$

*hierbei:*  $B_i$  wird gelöscht wie bei drule

## Verwendung der Regeln (1)

**Intro-Regeln** eignen sich zur Zerlegung rechts von  $\implies$ .

Anwendung mit `rule`:

**lemma** " $P \implies A \wedge B$ "

**apply** (`rule conjI`)

Ergibt:

1.  $P \implies A$

2.  $P \implies B$

**Elim-Regeln** eignen sich zur Zerlegung links von  $\implies$ .

Anwendung mit `erule`:

**lemma** " $A \wedge B \implies C$ "

**apply** (`erule conjE`)

Ergibt:

1.  $\llbracket A; B \rrbracket \implies C$

## Verwendung der Regeln (2)

**Dest-Regeln** beschreiben (evtl. schwächere) Konsequenzen.  
Anwendung mit `frule/drule`:

### constdefs

```
prime :: "nat  $\Rightarrow$  bool"
```

```
"prime p == 1 < p  $\wedge$  ( $\forall$  m. m dvd p  $\longrightarrow$  m = 1  $\vee$  m = p)"
```

```
even :: "nat  $\Rightarrow$  bool"
```

```
"even n ==  $\exists$  k. n = 2 * k"
```

```
odd :: "nat  $\Rightarrow$  bool"
```

```
"odd n ==  $\neg$  even n"
```

```
lemma even_prime_2: "[[ prime p; even p ]]  $\Longrightarrow$  p = 2"
```

```
lemma "[[ prime p; 2 < p ]]  $\Longrightarrow$  odd p"
```

```
apply (simp add: odd_def)
```

```
apply (rule notI)
```

```
apply (drule even_prime_2)
```

```
  apply assumption
```

```
apply simp
```

```
done
```

## Regelanwendung und Unifikation

**Anwendbarkeit:** Statt syntaktischer Gleichheit genügt *Unifizierbarkeit*

Speziell für (rule R):

*Gegeben:* Regel  $R: \llbracket A_1 \dots A_n \rrbracket \Longrightarrow A$

*Aktuelles Beweisziel:*  $\llbracket B_1 \dots B_m \rrbracket \Longrightarrow B$

*Bedingung:*  $A$  und  $B$  unifizierbar mit Unifikator  $\sigma$ , d.h.  $\sigma(A) \equiv \sigma(B)$

*Neue Beweisziele:*

$\llbracket \sigma(B_1) \dots \sigma(B_m) \rrbracket \Longrightarrow \sigma(A_1)$

...

$\llbracket \sigma(B_1) \dots \sigma(B_m) \rrbracket \Longrightarrow \sigma(A_n)$

Analog für (frule R), (drule R), (erule R)

- Regeln der Aussagenlogik
- Regelanwendung (vorwärts / rückwärts)
- Sichere / unsichere Regeln

## Sichere / unsichere Regeln (1)

**Sichere Regel:** Ohne Informationsverlust: Beweisbarkeit bleibt erhalten

*Bsp.:*

**lemma** " $A \wedge B \implies P$ "

**apply** (*erule conjE*)

Ergibt:

$$1. \llbracket A; B \rrbracket \implies P$$

*Daher:* wende sichere Regeln zuerst an

*Sichere aussagenlog. Regeln:*

conjI, impI, notI, iffI, TrueI, refl, classical, conjE, disjE, FalsE

## Sichere / unsichere Regeln (2)

**Unsichere Regel:** Mit Informationsverlust: Beweisbarkeit kann verloren gehen

*Bsp.:*

**lemma** " $\llbracket A; B; C \rrbracket \implies (A \wedge B) \vee (C \wedge D)$ "

**apply** (*rule disjI2*)

Ergibt:

$$1. \llbracket A; B; C \rrbracket \implies C \wedge D$$

*Daher:* wende unsichere Regeln möglichst spät an

*Unsichere aussagenlog. Regeln:*

disjI1, disjI2, impE, iffD1, iffD2, notE, FalseE, subst

## Sichere / unsichere Regeln (3)

Auch bei sicheren Regeln:

Aufspaltung des Beweisbaums vermeiden (iffE, disjE, conjI)

*Vergleiche:*

Anwendung von conjE vor disjE, conjI

Anwendung von disjE, conjI vor conjE

bei

**lemma** " $\llbracket A \wedge B; C \vee D \rrbracket \implies A \wedge (D \vee C)$ "

## Teil 5: Prädikatenlogik

- Regeln der Prädikatenlogik
- Regelveränderung
- Regelkombination (Tacticals) und Beweissuche
- Mengen in HOL

## Regelformat mit Parametern (1)

**Format** von Regeln / Beweiszielen:  $\bigwedge x_1 \dots x_n . \llbracket P_1 \dots P_m \rrbracket \implies P$

Die *Parameter*  $x_1 \dots x_n$  entsprechen lokalen Konstanten.

Umbenennung mit `apply (rename_tac y1 ... yk)`  
ergibt  $\bigwedge x_1 \dots y_1 \dots y_k . \llbracket P_1' \dots P_m' \rrbracket \implies P'$

## Regelformat mit Parametern (2)

Skopus von Variablen / Parametern:

*Variablen:* Quantoren der Objektlogik maximal bis zum nächsten  $\implies$

*Parameter:* Auch über  $\implies$  hinweg.

Also:

$$\wedge x y z . \llbracket \forall y . P y \longrightarrow Q z y ; Q x y \rrbracket \implies \exists z . Q z y$$

entspricht

$$\wedge x_1 y_1 z_1 . \llbracket (\forall y_2 . P y_2 \longrightarrow Q z_1 y_2) ; Q x_1 y_1 \rrbracket \implies \exists z_2 . Q z_2 y_1$$

## Regeln der Prädikatenlogik

Einführungsregeln (*Intro*):

$$\frac{\bigwedge x. P x}{\forall x. P x} \text{allI}$$

$$\frac{P ?x}{\exists x. P x} \text{exI}$$

Beseitigungsregeln (*Elim*):

$$\frac{\forall x. P x}{P ?t} \text{spec}$$

$$\frac{\forall x. P x \quad P ?x \implies R}{R} \text{allE}$$

$$\frac{\exists x. P x \quad \bigwedge x. P x \implies R}{R} \text{exE}$$

*Beachte:*

- allI und exE führen *neue* Parameter ein, dabei evtl. Umbenennung
- Codiert *Variablenbedingung* des Natürlichen Schließens

- Andernfalls: Falsche Schlüsse der Art:

$$\bigwedge x. P x \implies P x \rightsquigarrow \bigwedge x. P x \implies \forall x. P x \rightsquigarrow \exists x. P x \implies \forall x. P x$$

## Exkurs: Termgleichheit und Unifikation (1)

**Gleichheit** wird interpretiert:

- modulo  $\alpha$ : Umbenennung gebundener Variablen

**lemma** " $(\lambda x. x + 5) = (\lambda y. y + 5)$ "

**by** *(rule refl)*

- modulo  $\beta$ : Berechnung im  $\lambda$ -Kalkül

**lemma** " $(\lambda x. x + 5) = (\lambda z x. x + z) 5$ "

**by** *(rule refl)*

- modulo  $\eta$ : Schwache Form der Extensionalität

**lemma** " $(\lambda x. f x) = f$ "

**by** *(rule refl)*

## Exkurs: Termgleichheit und Unifikation (2)

**Unifikation:** Bestimmen von Lösungen für schematische Variablen, so daß Terme gleich werden.

... jedoch nicht beliebige Lösungen. Verhindern:

$$\frac{\frac{\frac{}{\wedge y. ?x = y} \text{ refl}}{\forall y. ?x = y} \text{ allI} \quad P \ ?x}{(\forall y. ?x = y) \wedge P \ ?x} \text{ conjI} \quad \rightsquigarrow \quad \frac{\frac{\frac{}{\wedge y. y = y} \text{ refl}}{\forall y. y = y} \text{ allI} \quad P \ y}{(\forall y. y = y) \wedge P \ y} \text{ conjI}}{\exists x. (\forall y. x = y) \wedge P \ x} \text{ exI}$$

**Fazit:** Schematische Variablen nur mit Konstanten/ freien Variablen lösen, nicht mit Parametern!

## Exkurs: Termgleichheit und Unifikation (3)

Schematische Variablen können legal von Parametern abhängen:  $?x p_1 \dots p_n$

**Intuitiv:**

“Lösung von  $?x p_1 \dots p_n = t$  ist möglich,  
wenn  $t$  nur Parameter aus  $\{p_1 \dots p_n\}$  enthält”

**Formal:**

$?x$  ist Funktion, angewandt auf  $p_1 \dots p_n$

Lösung ist Funktion  $?x := \lambda p_1 \dots p_n. t$

$\rightsquigarrow$  **Unifikation höherer Ordnung**, jedoch gutmütiges Fragment

## Anpassen von Parameterkontexten

Einige Schritte im Beweis von ...

**lemma** " $\forall x. \exists y. x = y$ "

**apply** (*rule allI*)

**apply** (*rule exI*)

1.  $\bigwedge x. x = ?y1\ x$

Warum gelingt Anwendung von *refl*:  $?t = ?t$ ?

**Lifting** der Regel auf  $\bigwedge x. ?t\ x = ?t\ x$

dann Resolution

*Was passiert bei*  $\exists x. \forall y. x = y$  ??

# Lifting bei Resolution (1)

$$\begin{array}{c}
 \psi_1[\vec{v}] \quad \dots \quad \psi_m[\vec{v}] \\
 \hline
 \begin{array}{c}
 \psi[\vec{v}] \\
 \stackrel{=?}{=} \\
 \bigwedge \vec{x}_i. \vec{\Gamma}_i \implies \phi_i
 \end{array} \\
 \phi_1 \quad \dots \quad \dots \quad \phi_n \\
 \hline
 \phi
 \end{array}$$

## Lifting bei Resolution (2)

$$\frac{\bigwedge \vec{x}_i. \vec{\Gamma}_i \Longrightarrow \psi_1[\vec{v} \ \vec{x}_i] \quad \dots \quad \bigwedge \vec{x}_i. \vec{\Gamma}_i \Longrightarrow \psi_m[\vec{v} \ \vec{x}_i]}{\quad}$$

$$\bigwedge \vec{x}_i. \vec{\Gamma}_i \Longrightarrow \psi[\vec{v} \ \vec{x}_i]$$

=?

 $\phi_1 \quad \dots$ 

$$\bigwedge \vec{x}_i. \vec{\Gamma}_i \Longrightarrow \phi_i$$

 $\dots \quad \phi_n$ 


---

 $\phi$

## Lifting bei Resolution (3)

$$\frac{\phi_1 \quad \dots \quad \bigwedge \vec{x}_i. \vec{\Gamma}_i \implies \psi_1[\vec{v} \quad \vec{x}_i] \quad \dots \quad \bigwedge \vec{x}_i. \vec{\Gamma}_i \implies \psi_m[\vec{v} \quad \vec{x}_i] \quad \dots \quad \phi_n}{\phi}$$

## Sichere / unsichere Regeln (1)

**Sichere Regeln** der Prädikatenlogik:  $\text{exE}$ ,  $\text{allI}$

**Unsicher:**  $\text{spec}$  /  $\text{exI}$  wegen falscher Parameterabhängigkeiten:

**lemma** " $(\exists x. \forall y. R x y) \implies (\forall y. \exists x. R x y)$ "

Welche Folge von Regelanwendungen führt zum Erfolg?

## Sichere / unsichere Regeln (2)

**Unsicher:** `drule spec` (synonym: `erule allE`)

... wenn Prämisse noch einmal benötigt wird

*Lösung:* `frule spec`

*Bsp.:*

```
lemma "[[ P a;  $\forall x. P x \longrightarrow P (f x)$  ]]  $\implies P (f (f a))$ "
apply (frule_tac x=a in spec)
apply (drule_tac x="f a" in spec)
apply (drule mp) apply assumption
apply (drule mp) apply assumption
apply assumption
done
```

- Regeln der Prädikatenlogik
- **Regelveränderung**
- Regelkombination (Tacticals) und Beweissuche
- Mengen in HOL

## Regelanwendung mit Instanziierung

**Explizite Instanziierung** oft günstiger als Instant. durch Unifikation:

```
lemma "∃ x::nat. 10 < x ∧ x < 20 ∧ x mod 7 = 0"
apply (rule_tac x=14 in exI)
by simp
```

**Allgemein:** Wenn Regel  $R$  die schematischen Variablen  $?x_1 \dots ?x_n$  hat:

```
apply (rule_tac x_1="t_1" and ... and x_n="t_n" in R)
```

**Analog:** *drule\_tac*, *frule\_tac*, *erule\_tac*

! Die  $x_i$  sind Namen von schematischen Variablen,  
*nicht* Namen von Variablen im Beweis:

```
lemma "∀ z. P z ⇒ P a"
apply (drule_tac x=a in spec)
apply assumption
done
```

## Regeln on-the-fly: Instantiierung

Gegeben:

`zdiv_mono1: [[?a ≤ ?a'; 0 < ?b]] ⇒ ?a div ?b ≤ ?a' div ?b`

Instantiierung positional:

`thm zdiv_mono1 [of _ _ 2]`

`[[?a ≤ ?a'; 0 < 2]] ⇒ ?a div 2 ≤ ?a' div 2`

Instantiierung einzelner Variablen:

`thm zdiv_mono1 [where b=2]`

`[[?a ≤ ?a'; 0 < 2]] ⇒ ?a div 2 ≤ ?a' div 2`

Zusätzlich Simplifikation:

`thm zdiv_mono1 [where b=2, simplified]`

`?a ≤ ?a' ⇒ ?a div 2 ≤ ?a' div 2`

## Regeln on-the-fly: Resolution (1)

### THEN

*Gegeben:* Regeln

$$RA_1 : \llbracket A_1 \dots A_n \rrbracket \implies B_1$$

$$RB : \llbracket B_1 \dots B_m \rrbracket \implies B$$

$RA_1$  [THEN  $RB$ ] ergibt

$$\llbracket A_1 \dots A_n, B_2 \dots B_m \rrbracket \implies B$$

### OF

... ebenso:  $RB$  [OF  $RA_1$ ]

Iteriert:  $RB$  [OF  $RA_1$   $RA_2$  ...]

## Regeln on-the-fly: Resolution (2)

Beispiel: Beweise

**lemma** `"inj f  $\implies$  map (inv f) (map f xs) = xs"`

Leider falsche Richtung:

`map_compose: map (?f  $\circ$  ?g) ?xs = map ?f (map ?g ?xs)`

Besser:

`map_compose [THEN sym]: map ?f1 (map ?g1 ?xs1) = map (?f1  $\circ$  ?g1) ?xs1`

Also:

**lemma** `"inj f  $\implies$  map (inv f) (map f xs) = xs"`

**apply** `(simp only: map_compose [THEN sym])`

**apply** `(simp add: inj_iff id_def del: map_compose)`

**done**

- Regeln der Prädikatenlogik
- Regelveränderung
- Regelkombination (Tacticals) und Beweissuche
- Mengen in HOL

## Tacticals

Kombinieren einzelne Taktiken:

- Hintereinanderausführung: T1, T2
- Alternative: T1 | T2
- Mehrfachausführung: T+
- Keine oder eine Ausführung: T?

*Bsp.:*

```
lemma "[[ P a;  $\forall x. P x \longrightarrow P (f x)$  ]]  $\implies P (f (f a))$ "
apply (frule_tac x=a in spec)
apply (drule_tac x="f a" in spec)
apply (drule mp, assumption+)+
done
```

## Bereinigung von Formeln

- `apply (intro ...)`: Wiederholte Anwendung von Intro-Regel ... auf Beweisziel und entstehende Unterziele
- *z.B.*: `apply (intro allI)`: Beseitigt alle All-Quantoren in Conclusion
- `apply (elim ...)`: Wiederholte Anwendung von Elim-Regel ... auf Beweisziel und entstehende Unterziele
- *z.B.*: `apply (elim conjE)`: Beseitigt alle Konjunktionen in Prämissen
- `apply clarify`: Wiederholte Anwendung sicherer Regeln ohne Aufspaltung des Beweisziels
- `apply (clarsimp simp add: ..)` Verschränkung von `clarify` und `simp`

# Automatisierte Beweissuche (1)

## Prädikatenlogische Beweissuche:

- `apply (fast intro: ... dest: ...)`:  
Anwendung von Taktiken; HO-Unifikationsalgorithmus
- `apply (blast intro: ... dest: ...)`:  
Effizienter Tableau-Beweiser; nur einfache Unifikation

## *Beide:*

- Anwendung nur auf aktuelles Beweisziel
- Scheitern, wenn Ziel nicht beweisbar

## Automatisierte Beweissuche (2)

**Presburger Arithmetik:** Lineares Fragment der Arithmetik

*Entscheidungsprozedur:* `arith`

**lemma** " $\exists x::\text{nat}. (x + 3 = 10 \wedge (\forall y. x < y \longrightarrow 12 < 2 * y))$ "

**by** `arith`

*Bei Formeln außerhalb des Fragments:* Abstraktion

**lemma** " $(x::\text{nat}) * y \leq 2 * x * y$ "

**by** `arith`

Evtl. Scheitern:  $(x::\text{nat}) * y \leq 2 * y * x$

## Automatisierte Beweissuche (3)

**Kombination** von `simp`, prädikatenlogischen Beweisen, Arithmetik:

```
apply (auto simp add: ... del: ...  
      intro: ... dest: ...)
```

- Anwendung auf alle Beweisziele
  - Nicht beweisbare Ziele werden vereinfacht
- ! *Ratsam*: Nur als letzten Schritt in Beweisskript anwenden

- Regeln der Prädikatenlogik
- Regelveränderung
- Regelkombination (Tacticals) und Beweissuche
- Mengen in HOL

## Mengen in HOL (1)

Polymorpher Typ `'a set`

Wichtigste Operation: **Mengen-Komprehension**

`Collect`  $::$  `"('a  $\Rightarrow$  bool)  $\Rightarrow$  'a set"`

Schreibweise: `"{x. P}"`  $==$  `"Collect (%x. P)"`

*Bsp.:*

**term** `"{x::nat. 2 < x}"`

**term** `"{xs. length xs = 8}"`

**Axiomatisierung:**

`mem_Collect_eq`: `(?a  $\in$  {x. ?P x}) = ?P ?a`

`Collect_mem_eq`: `{x. x  $\in$  ?A} = ?A`

## Mengen in HOL (2)

Einige **Mengenfunktionen** und deren Definition:

$$\{\} \equiv \{x. \text{False}\}$$

$$\text{UNIV} \equiv \{x. \text{True}\}$$

$$?A \cup ?B \equiv \{x. x \in ?A \vee x \in ?B\}$$

$$?A \cap ?B \equiv \{x. x \in ?A \wedge x \in ?B\}$$

$$\text{insert } ?a ?B \equiv \{x. x = ?a\} \cup ?B$$

Damit gelten u.a. folgende Lemmas: (**Beziehung zu  $\wedge$ ,  $\vee$ ,  $\longrightarrow$ ?**)

$$?A \cap ?B \subseteq ?A$$

$$?A \cap ?B \subseteq ?B$$

$$\llbracket ?C \subseteq ?A; ?C \subseteq ?B \rrbracket \implies ?C \subseteq ?A \cap ?B$$

$$?A \subseteq ?A \cup ?B$$

$$?B \subseteq ?A \cup ?B$$

$$\llbracket ?A \subseteq ?C; ?B \subseteq ?C \rrbracket \implies ?A \cup ?B \subseteq ?C$$

## Mengen in HOL (3)

### Quantifizierung über Mengen

- All-Quantor:  $\forall x \in A. P x$   
entspricht  $\forall x. x \in A \longrightarrow P x$
- Existenz-Quantor:  $\exists x \in A. P x$   
entspricht  $\exists x. x \in A \wedge P x$

## Mengen in HOL (4)

### Regeln:

Einführung:  $(\bigwedge x. x \in ?A \implies ?P\ x) \implies \forall x \in ?A. ?P\ x$   
 $\llbracket ?P\ ?x; ?x \in ?A \rrbracket \implies \exists x \in ?A. ?P\ x$

### Elimination:

$\llbracket \forall x \in ?A. ?P\ x; ?x \in ?A \rrbracket \implies ?P\ ?x$

$\llbracket \forall x \in ?A. ?P\ x; ?P\ ?x \implies ?Q; ?x \notin ?A \implies ?Q \rrbracket \implies ?Q$

$\llbracket \exists x \in ?A. ?P\ x; \bigwedge x. \llbracket x \in ?A; ?P\ x \rrbracket \implies ?Q \rrbracket \implies ?Q$

## Teil 6: Induktiv definierte Mengen

- Induktive Mengen: Definition
- Regelinduktion und Regelinversion
- Gegenseitig induktive Mengen
- Anwendung: Programmiersprachensemantik

Im Tutorial: Kap. 7

# Induktive Mengen

**Idee:** Beschreibe eine Menge dadurch, wie sie erzeugt wird.

**Bsp.:** Gerade Zahlen *even*:

1.  $0$  ist eine gerade Zahl.
2. Wenn  $n$  eine gerade Zahl ist, so auch  $n+2$ .
3. Nur die mit 1. und 2. erzeugten Zahlen sind gerade

Also:

$0 \in \text{even} \rightsquigarrow 2 \in \text{even} \rightsquigarrow 4 \in \text{even} \dots$

## Induktive Mengen: Definition

Einführungsregeln drücken Erzeugungsprinzip aus:

**consts** *even* :: "nat set"

**inductive** *even*

**intros**

*ZeroI*: "0 ∈ *even*"

*Add2I*: "*n* ∈ *even* ⇒ *Suc*(*Suc* *n*) ∈ *even*"

Induktionsregel besagt: *even* ist die kleinste so erzeugte Menge:

*even.induct* :

$\llbracket ?xa \in \text{even}; ?P\ 0; \bigwedge n. \llbracket n \in \text{even}; ?P\ n \rrbracket \implies ?P\ (\text{Suc}\ (\text{Suc}\ n)) \rrbracket \implies ?P\ ?xa$

## Enthaltensein in induktiver Menge

Nachweis durch wiederholte Rückwärts-Anwendung der Einführungsregeln:

```
lemma "Suc (Suc (Suc (Suc 0))) ∈ even"  
apply (rule Add2I)  
apply (rule Add2I)  
apply (rule ZeroI)  
done
```

*Nicht-Enthaltensein:* Erfordert Induktion

```
lemma "Suc (Suc (Suc 0)) ∈ even"  
apply (rule Add2I)  
oops
```

## Vergleich mit Prolog

Einführungsregeln durch **Fakten**

```
even(0).  
even(suc (suc (N))) :- even(N).
```

Abschlußeigenschaft durch **negation by failure**

```
? even(suc (suc (suc (suc (0))))).  
> yes.
```

```
? even(suc (suc (suc (0)))).  
> no.
```

## Einschränkungen (1)

**Erforderlich:** Strikt positives Vorkommen der ind. Menge in Prämissen.

*Nicht gültig:* Regel der Art:

$$n \notin \text{evenneg} \implies (\text{Suc } n) \in \text{evenneg}$$

**Grund:** Was ist die minimale Menge, die erzeugt wird durch:

$$a \notin S \implies b \in S$$

$$b \notin S \implies a \in S$$

$\{a\}$  oder  $\{b\}$  oder  $\{a, b\}$  ??

## Einschränkungen (2)

Wenig sinnvoll (aber legal): Regeln ohne “Basisfall”:

```
consts evenempty :: "nat set"
```

```
inductive evenempty
```

```
  intros
```

```
  Add2Ie: "n ∈ evenempty ⇒ Suc(Suc n) ∈ evenempty"
```

Zeige:  $evenempty = \{\}$

Wie verhält sich Prolog hier?

- Induktive Mengen: Definition
- Regelinduktion und Regelinversion
- Gegenseitig induktive Mengen
- Anwendung: Programmiersprachensemantik

## Regelinduktion (1)

### Induktionsprinzip:

$$\llbracket ?xa \in \text{even}; ?P 0; \bigwedge n. \llbracket n \in \text{even}; ?P n \rrbracket \implies ?P (\text{Suc} (\text{Suc} n)) \rrbracket \implies ?P ?xa$$

Um nachzuweisen, daß  $P$  für jedes  $a$  von  $\text{even}$  gilt:

- Weise  $P$  für den Basisfall nach
- Weise nach, daß  $P$  bei jeder Regelanwendung erhalten bleibt

### Bsp.:

**lemma** " $n \in \text{even} \implies \exists k. n = 2 * k$ "

**apply** (*erule even.induct*)

**apply** *simp*

**apply** (*erule exE*)

**apply** (*rule\_tac x="Suc k" in exI*)

**apply** *simp*

**done**

## Regelinduktion (2)

Regelinduktion anwendbar für Beweisziele der Form

$$\llbracket \dots n \in \text{even} \dots \rrbracket \implies P n$$

Eventuell Beweisziel geeignet umformen.

**Bsp.:** Zeige  $\text{Suc } (2 * k) \notin \text{even}$

**lemma** *even\_not\_Suc2k* [*rule\_format*]: " $n \in \text{even} \implies \forall k. n \neq \text{Suc } (2 * k)$ "

**apply** (*erule even.induct*)

**apply** *simp*

**apply** *arith*

**done**

**lemma** " $\text{Suc } (2 * k) \notin \text{even}$ "

**by** (*blast dest: even\_not\_Suc2k*)

## Regelinduktion (3)

Nicht alle Beweise über induktive Mengen brauchen Regelinduktion:

```
lemma "2 * k ∈ even"  
apply (induct k)  
apply simp  
apply (rule ZeroI)  
apply simp  
apply (rule Add2I)  
apply assumption  
done
```

Nützlich: Erzeugungsregeln als Intros deklarieren:

```
declare even.intros [intro]  
  
lemma "2 * k ∈ even"  
by (induct k) auto
```

## Regelinversion (1)

Teilweise hinreichend: **Fallunterscheidung** über Regel:

```
lemma "Suc 0 ∉ even"
  apply (rule notI)
  apply (erule even.cases)
  apply simp+
done
```

Hierbei ist:

*even.cases* :

$$\llbracket ?a \in \text{even}; ?a = 0 \implies ?P; \bigwedge n. \llbracket ?a = \text{Suc} (\text{Suc } n); n \in \text{even} \rrbracket \implies ?P \rrbracket \implies ?P$$

**!** *S.induct* verhält sich zu *S.cases* wie  
*Induktion zu Fallunterscheidung bei Datentypen*

## Regelinversion (2)

Elimination irrelevanter Teilfälle automatisch mit `ind_cases`.

Vergleiche:

```
lemma "Suc (Suc n) ∈ even ⇒ n ∈ even"  
apply (erule even.cases)  
apply simp  
apply simp  
done
```

und

```
lemma "Suc (Suc n) ∈ even ⇒ n ∈ even"  
apply (ind_cases "Suc (Suc n) ∈ even")  
apply assumption  
done
```

- Induktive Mengen: Definition
- Regelinduktion und Regelinversion
- **Gegenseitig induktive Mengen**
- Anwendung: Programmiersprachensemantik

## Gegenseitig induktive Mengen: Definition

### Definition

#### consts

```
even_eo :: "nat set"
odd_eo  :: "nat set"
```

#### inductive "even\_eo" "odd\_eo"

#### intros

```
even0: "0 ∈ even_eo"
evenS: "n ∈ odd_eo ⇒ (Suc n) ∈ even_eo"
oddS:  "n ∈ even_eo ⇒ (Suc n) ∈ odd_eo"
```

### Generiertes Induktionsprädikat `even_eo_odd_eo.induct` :

$$\begin{aligned} & \llbracket ?P1.0\ 0; \\ & \bigwedge n. \llbracket n \in \text{odd\_eo}; ?P2.0\ n \rrbracket \implies ?P1.0\ (\text{Suc}\ n); \\ & \bigwedge n. \llbracket n \in \text{even\_eo}; ?P1.0\ n \rrbracket \implies ?P2.0\ (\text{Suc}\ n) \rrbracket \\ & \implies (?xb \in \text{even\_eo} \longrightarrow ?P1.0\ ?xb) \wedge (?xa \in \text{odd\_eo} \longrightarrow ?P2.0\ ?xa) \end{aligned}$$

## Gegenseitig induktive Mengen: Beweise

### Beachte:

- Eine Teilaussage für jeden Datentyp erforderlich
- Anwendung des Induktionsprädikats mit `rule`

```
lemma "(n ∈ even_eo → n ≠ 0 → (n - 1) ∈ odd_eo) ∧  
        (n ∈ odd_eo → (n - 1) ∈ even_eo)"  
apply (rule even_eo_odd_eo.induct)  
apply simp+  
done
```

## Organisatorisches: Zeitplanung

**10.12.2003**

*Vorlesung:* Rekursive Definitionen; Imperative Programme

*Aufgabe:* Aufspannende Bäume in Graphen

**17.12.2003**

*Vorlesung:* Isar

*Aufgabe:* Imperative Listensortierfunktion

**7.1.2004**

*Vorlesung:* Formalisierung von Programmiersprachensemantik

*Aufgabe:* Lebendigkeitsanalyse für Variablen / Programmoptimierung (1)

**14.1.2004**

*Vorlesung:* –

*Aufgabe:* Lebendigkeitsanalyse für Variablen / Programmoptimierung (2)

**21.1.2004** Abschlußpräsentation

## Teil 7: Rekursion; Imperative Programme

- Terminierend rekursive Funktionen [Im Tutorial: Kap. 3.5 / 9.2](#)
  - Definition mit *recdef*, Nachweis der Terminierung
  - Beweise über rekursive Funktionen
- Imperative Programme

## Definition mit recdef

### Wesentliche Elemente:

- Funktions-Deklaration
- Funktions-Definition in ML-Stil
- Angabe einer Terminationsordnung

```
consts fib :: "nat  $\Rightarrow$  nat"  
recdef fib "measure( $\lambda$  n. n)"  
  "fib 0 = 0"  
  "fib (Suc 0) = 1"  
  "fib (Suc(Suc x)) = fib x + fib (Suc x)"
```

## Terminationsordnung (1)

**Intuition:** Argumente müssen bei jedem rek. Aufruf “kleiner” werden

**Problem:** Was heißt “kleiner”?

*Bsp fib:* Klar:

- $x < (\text{Suc}(\text{Suc } x))$
- $\text{Suc } x < (\text{Suc}(\text{Suc } x))$

Und in folgender Funktion?

$\text{ten } x = \text{if } 10 \leq x \text{ then } 10 \text{ else } \text{ten } (\text{Suc } x)$

## Terminationsordnung (2)

```

consts ten :: "nat  $\Rightarrow$  nat"
recdef ten "measure( $\lambda$  n. 10 - n)"
  "ten x = (if (10  $\leq$  x) then 10 else ten (Suc x))"

```

Begründung:  $10 - (\text{Suc } x) < 10 - x$ , wenn  $\neg (10 \leq x)$

**Allgemein:** Argumente müssen bzgl. *wohlfundierter* Ordnung kleiner werden.

**Wohlfundiertheit** einer Relation  $R$ :

$$wf\ R \equiv \forall P. (\forall x. (\forall y. (y, x) \in R \longrightarrow P\ y) \longrightarrow P\ x) \longrightarrow (\forall x. P\ x)$$

“Eine Eigenschaft  $P$  gilt generell, wenn sie sich jeweils von kleineren auf größere Elemente vererbt”.

*Informell:* Eine solche Eigenschaft ist: “terminiert” / “ist definiert auf”

## Funktionsdefinition: Beispiele (1)

Frage: Warum terminiert:

$$\text{gcd } (0, v) = v$$

$$\text{gcd } (u, 0) = u$$

$$\text{gcd } (u, v) = (\text{if } v \leq u \text{ then } \text{gcd } (u - v, v) \text{ else } \text{gcd } (v - u, u))$$

## Funktionsdefinition: Beispiele (2)

Antwort:

```
consts gcd :: "(nat × nat) ⇒ nat"
recdef gcd "measure (λ (u,v). u + v)"
  "gcd (0, v) = v"
  "gcd (u, 0) = u"
  "gcd (u, v) = (if v ≤ u then gcd (u - v, v) else gcd (v - u, u))"

thm gcd.simps
```

!Definitionen mit *recdef* haben nur *ein* Argument (...evtl. ein *n*-Tupel)

!Bei mehrdeutigen Patterns gewinnt das erste passende

## Funktionsdefinition: Beispiele (3)

Frage: Warum terminiert:

`gcd_lex (0, v) = v`

`gcd_lex (u, v) = (if u ≤ v then gcd_lex (u, v-u) else gcd_lex (v, u))`

## Funktionsdefinition: Beispiele (4)

**Antwort:** Lexikographische Ordnung!

```
consts gcd_lex :: "(nat × nat) ⇒ nat"
recdef gcd_lex "measure (λ u. u) <*lex*> measure (λ v. v)"
  "gcd_lex (0, v) = v"
  "gcd_lex (u, v) = (if u ≤ v then gcd_lex (u, v - u) else gcd_lex (v, u))"
```

Dabei:  $(a, b) <_{lex} (a', b')$  gdw.

$a < a'$  oder

$a = a'$  und  $b < b'$

## Funktionsdefinition: Hinweise zur Terminierung

Quicksort:

```
consts qs :: "nat list  $\Rightarrow$  nat list"  
recdef qs "measure length"  
  "qs [] = []"  
  "qs(x#xs) =  
    qs (filter ( $\lambda$  y.  $y \leq x$ ) xs) @ [x] @ qs (filter ( $\lambda$  y.  $x < y$ ) xs)"  
(hints recdef_simp: less_Suc_eq_le)
```

!Anzeige nicht automatisch beweisbarer Aussagen bei Terminierungsnachweis

- Definition mit *recdef*, Nachweis der Terminierung
- Beweise über rekursive Funktionen

## Wohlfundierte Induktion

Definition von Wohlfundiertheit verkörpert *Induktionsprinzip*:

*wf\_induct*:

$$\llbracket wf\ ?r; \bigwedge x. \forall y. (y, x) \in ?r \longrightarrow ?P\ y \implies ?P\ x \rrbracket \implies ?P\ ?a$$

Für jede rek. Funktion  $f$  wird Induktionsformel  $f.induct$  generiert

*Bsp. ten.induct*:

$$(\bigwedge x. \neg 10 \leq x \longrightarrow ?P\ (Suc\ x) \implies ?P\ x) \implies ?P\ ?x$$

## Wohlfundierte Induktion: Beispiele (1)

```
ten.induct:
```

```
 $(\bigwedge x. \neg 10 \leq x \longrightarrow ?P (\text{Suc } x) \implies ?P x) \implies ?P ?x$ 
```

```
lemma "ten x = 10"
```

```
apply (induct x rule: ten.induct)
```

```
thm ten.simps
```

```
thm gcd.simps
```

```
apply simp
```

```
done
```

## Wohlfundierte Induktion: Beispiele (2)

`gcd.induct:`

```

[[ ?P 0 0;
  ∧ ab. ?P 0 (Suc ab);
  ∧ y. ?P (Suc y) 0;
  ∧ z ae. [[¬ Suc ae ≤ Suc z → ?P (Suc ae - Suc z) (Suc z);
            Suc ae ≤ Suc z → ?P (Suc z - Suc ae) (Suc ae)]]
           ⇒ ?P (Suc z) (Suc ae)]]
⇒ ?P ?u ?v

```

**lemma** `gcd_divides`: " $g = \text{gcd}(a,b) \longrightarrow g \text{ dvd } a \wedge g \text{ dvd } b$ "

**apply** (`induct_tac a b rule: gcd.induct`)

— vier Unterziele

- Terminierend rekursive Funktionen
- Imperative Programme

# Imperative Programme in Isabelle (1)

Codierung einfacher while-Programme:

$$\begin{array}{l}
 c = \text{SKIP} \\
 | \quad x := a \\
 | \quad c_0; c_1 \\
 | \quad \text{IF } b \text{ THEN } c_0 \text{ ELSE } c_1 \text{ FI} \\
 | \quad \text{WHILE } b \text{ INV } \{I\} \text{ DO } c \text{ OD} \\
 p = \text{VARS } v_1 \dots v_n \{P\} c \{Q\}
 \end{array}$$

## Imperative Programme in Isabelle (2)

Beispiel: Multiplikation

**lemma**

```
"VARs (m::nat) n a b
  {m = 0 ∧ n = 0}
  WHILE m ≠ a
  INV {n = m * b}
  DO n := n + b; m := m + 1 OD
  {n = a * b}"
```

**apply** *vcg*

**apply** *auto*

**done**

## Teil 8: Lesbare Beweise in Isar

- Basissprache
- Aussagenlogische Beweise
- Prädikatenlogische Beweise
- Induktions-Beweise
- Gleichungsketten: Calculations

## Motivation (1)

### Taktik-Script:

```
theorem ex1 : "( $\exists x. \forall y. P x y$ )  $\longrightarrow$  ( $\forall y. \exists x. P x y$ )"  
apply (rule impI)  
apply (erule exE)  
apply (rule allI)  
apply (rule exI)  
apply (drule spec)  
apply assumption  
done
```

- Maschinenlesbar, aber schwer verständlich
- Kompakt
- Meistens resistent gegen kleine Modifikationen

## Motivation (2)

Script in Isabelle/Isar:

```

theorem ex2 : "( $\exists$  x.  $\forall$  y. P x y)  $\longrightarrow$  ( $\forall$  y.  $\exists$  x. P x y)"
proof (rule impI)
  assume " $\exists$  x .  $\forall$  y. P x y" then show " $\forall$  y.  $\exists$  x . P x y"
  proof (rule exE)
    fix x assume h: " $\forall$  y. P x y" show " $\forall$  y.  $\exists$  x. P x y"
    proof (rule allI)
      fix y from h have "P x y" by (rule spec)
      then show " $\exists$  x . P x y" by (rule exI)
    qed
  qed
qed

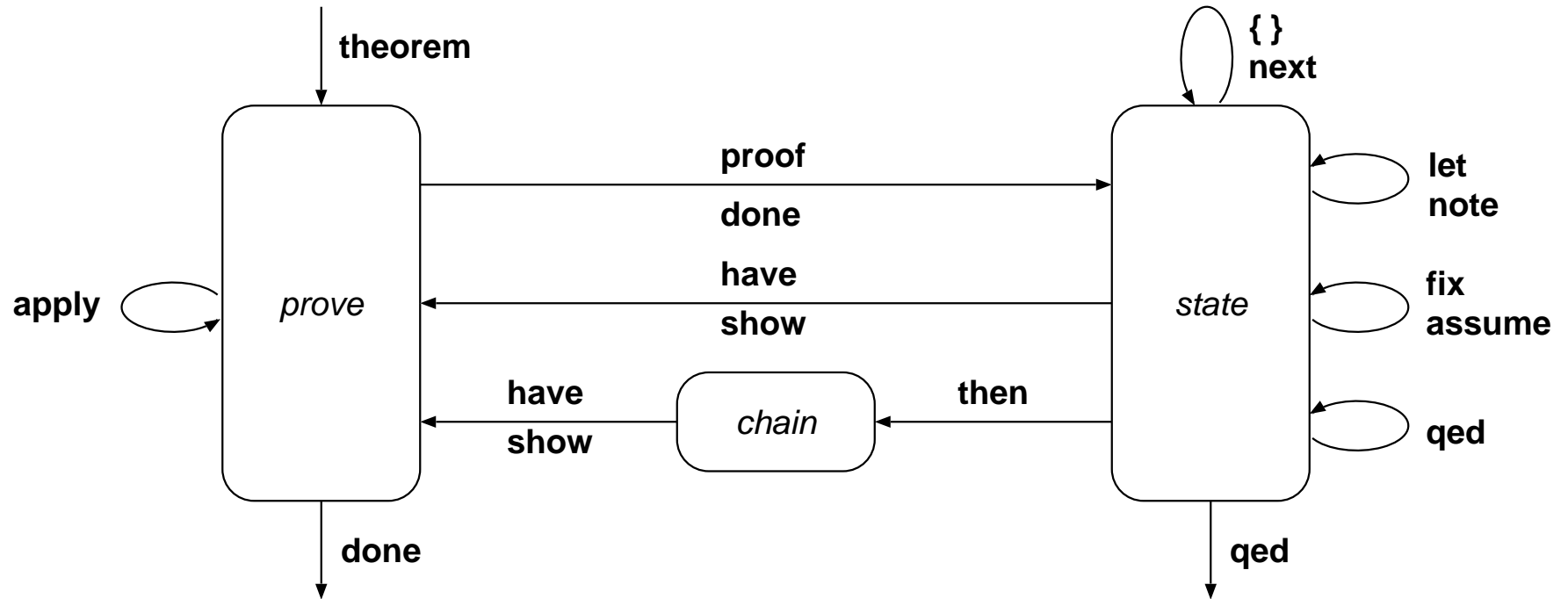
```

- Maschinenlesbar, zudem verständlich
- Verbos (jedoch: Abkürzungen möglich)
- u.U. leichter zu warten bei größeren Modifikationen

## Isabelle/Isar Basissprache

$goal$  ::= **theorem**  $proposition$   $proof$   
 $proof$  ::= **proof**  $method$ ?  $statement$ \* **qed**  $method$ ?  
 | **apply**  $method$   $proof$  | **done**  
 $statement$  ::= **fix**  $variable$ <sup>+</sup> | **assume**  $proposition$ <sup>+</sup>  
 | **then**? (**show** | **have**)  $proposition$   $proof$   
 | **note**  $name = name$ <sup>+</sup>  
 | **next** | {  $statement$  }  
 $proposition$  ::= ( $label$  :)?  $string$

# Isabelle/Isar Zustandsautomat



- Basissprache
- Aussagenlogische Beweise
- Prädikatenlogische Beweise
- Induktions-Beweise
- Gleichungsketten: Calculations

## Isar: Aussagenlogische Beweise (1)

### Einführungsregeln

```
lemma "A  $\longrightarrow$  A  $\vee$  B"  
proof (rule impI)  
  assume a: A  
  note a then  
  show "A  $\vee$  B"  
  proof (rule disjI1)  
  qed  
qed
```

- Aufsammeln von Annahmen mit **assume**
- Notieren von Annahmen zur späteren Verwendung mit **note**

## Isar: Aussagenlogische Beweise (2)

### Abkürzungen:

- Standard-Introduktionsregel auslassen (bei **proof**)
- **from**  $a$  statt **note**  $a$  **then**
- Verwenden der Standard-Intro-Regel: ..

**lemma** " $A \longrightarrow A \vee B$ "

**proof**

**assume**  $a: A$

**from**  $a$

**show** " $A \vee B$ " ..

**qed**

## Isar: Aussagenlogische Beweise (3)

### Eliminationsregeln / Explizites Aufsammeln von Hypothesen

**lemma**  $"A \wedge B \longrightarrow B \wedge A"$

**proof**

**assume**  $h: "A \wedge B"$

**from**  $h$  **have**  $a: A$  **by**  $(rule\ conjE)$

**from**  $h$  **have**  $b: B$  **by**  $(rule\ conjE)$

**from**  $b$   $a$

**show**  $"B \wedge A"$  **by**  $(rule\ conjI)$

**qed**

**!**Reihenfolge der Fakten bei **from**:

Sequenzielle Instantiierung von  $conjI$ :  $\llbracket ?P; ?Q \rrbracket \Longrightarrow ?P \wedge ?Q$

## Isar: Aussagenlogische Beweise (4)

### Implizites Aufsammeln von Hypothesen:

- mit **moreover** und **ultimately**  
(keine explizite Benennung notwendig)
- Abkürzung: Verwenden der Standard-Elim-Regel: ..

**lemma**  $"A \wedge B \longrightarrow B \wedge A"$

**proof**

**assume**  $h: "A \wedge B"$

**from**  $h$  **have**  $B$  ..

**moreover**

**from**  $h$  **have**  $A$  ..

**ultimately**

**show**  $"B \wedge A"$  ..

**qed**

## Isar: Abkürzung von Teilformeln (1)

Aktuelle Annahme: *this*

Aktuelles Beweisziel: *?thesis*

**lemma** " $A \wedge B \longrightarrow A \vee B$ "

**proof**

**assume** *h*: " $A \wedge B$ "

**from** *this* **show** " $A \vee B$ "

**proof**

**assume** *A* **show** *?thesis* ..

**qed**

**qed**

## Isar: Abkürzung von Teilformeln (2)

durch Pattern-Matching:

**lemma**  $"(A \wedge B) \wedge (C \wedge D) \longrightarrow (A \wedge C)"$  (is  $"?PA \wedge ?PC \longrightarrow ?P"$ )

**proof**

**assume**  $"?PA \wedge ?PC"$

**from** *this* **show**  $?P$

**proof**

**assume**  $pa: ?PA$  **and**  $pc: ?PC$

**from**  $pa$  **have**  $A$  ..

**moreover**

**from**  $pc$  **have**  $C$  ..

**ultimately have**  $?P$  ..

**thus** *?thesis* .

**qed**

**qed**

## Abgeleitete Schlüsselwörter

<b>by</b> $m_1$ $m_2$	≡	<b>proof</b> $m_1$ <b>qed</b> $m_2$
..	≡	<b>by</b> <i>rule</i>
.	≡	<b>by</b> <i>this</i>
<b>hence</b>	≡	<b>then have</b>
<b>thus</b>	≡	<b>then show</b>
<b>from</b> $\vec{a}$	≡	<b>note</b> <i>this</i> = $\vec{a}$ <b>then</b>
<b>with</b> $\vec{a}$	≡	<b>from</b> $\vec{a}$ <i>this</i>

- Basissprache
- Aussagenlogische Beweise
- **Prädikatenlogische Beweise**
- Induktions-Beweise
- Gleichungsketten: Calculations

## Isar: Prädikatenlogische Beweise (1)

Fixieren universell quantifizierter Variablen:

```
lemma " $\forall x. (P x \longrightarrow (\exists y. P y))$ "
```

```
proof
```

```
  fix a
```

```
  show " $(P a \longrightarrow (\exists y. P y))$ "
```

```
  proof
```

```
    assume "P a"
```

```
    then show " $\exists y. P y$ " .. — (rule exI)
```

```
  qed
```

```
qed
```

**!fix** erlaubt Umbenennung von Variablen

## Isar: Prädikatenlogische Beweise (2)

Existentiell quantifizierte Annahmen:

Dekomposition mit **obtain** *x* **where**

**lemma assumes** *h*: " $\exists x. P x \wedge Q x$ " **shows** " $\exists x. P x$ "

**proof** -

**from** *h* **obtain** *x* **where** " $P x \wedge Q x$ " ..

**from** *this* **have** " $P x$ " ..

**then show** *?thesis* ..

**qed**

!Nicht-Anwendung eines initialen Beweisschritts mit **proof** -

!Kombination **assumes** / **shows** anstelle von Implikation  $\implies$

- Basissprache
- Aussagenlogische Beweise
- Prädikatenlogische Beweise
- **Induktions-Beweise**
- Gleichungsketten: Calculations

## Isar: Fallunterscheidungen

.. über den Datentyp *bool*

```
lemma "P  $\vee$   $\neg$  P"
```

```
proof (cases P)
```

```
  case True thus ?thesis ..
```

```
next
```

```
  case False thus ?thesis ..
```

```
qed
```

oder über andere induktive Datentypen:

```
lemma "length (tl xs) = length xs - 1"
```

```
proof (cases xs)
```

```
  case Nil thus ?thesis by simp
```

```
next
```

```
  case (Cons x xs')
```

```
  thus ?thesis by simp
```

```
qed
```

## Isar: Induktion

```
lemma "sum (xs @ ys) = sum xs + sum ys"  
proof (induct xs)  
  case Nil show ?case by simp  
next  
  case (Cons x xs') show ?case  
    by simp  
qed
```

**case** bewirkt:

- implizites **fix** der Variablen des Falls
- implizites **assume** der Hypothesen

Benennung der Teilfälle mit *?case*.

## Isar: Regel-Induktion

```

lemma "n ∈ even ⇒ ∃ k. n = 2 * k"
proof (induct rule: even.induct)
  case ZeroI show ?case by simp
next
  case (Add2I n)
  have n_ev: "n ∈ even" .
  have nke: "∃ k. n = 2 * k" .
  from nke obtain k where nk: "n = 2 * k" ..
  show ?case
  proof
    show "Suc (Suc n) = 2 * (Suc k)" by simp
  qed
qed

```

- Basissprache
- Aussagenlogische Beweise
- Prädikatenlogische Beweise
- Induktions-Beweise
- Gleichungsketten: Calculations

## Isar: Calculational Reasoning

**lemma** *dvd\_minus*:

**assumes** *du*: "(*d*::nat) dvd *u*"

**assumes** *dv*: "*d* dvd *v*"

**shows** "*d* dvd *u* - *v*"

**proof** -

**from** *du* **obtain** *ku* **where** "*u* = *d* \* *ku*" **by** - (*simp* *add*: *dvd\_def*, *rule* *exE*)

**moreover**

**from** *dv* **obtain** *kv* **where** "*v* = *d* \* *kv*" **by** - (*simp* *add*: *dvd\_def*, *rule* *exE*)

**ultimately have** "*u* - *v* = *d* \* *ku* - *d* \* *kv*" **by** *simp*

**also have** "... = *d* \* (*ku* - *kv*)" **by** (*simp* *add*: *diff\_mult\_distrib2*)

**finally show** *?thesis* **by** *simp*

**qed**

## Teil 9: Modellierung von Semantik

- Modellierung von Programmiersprachen-Semantik
- Projekt: Datenflussanalyse

# Modellierung von Semantik (1)

## Operationelle Semantik:

Beschreibung der Programmausführung durch abstrakte Maschine

## Ingredientien:

- Zustandsbegriff
- Programmiersprache (Ausdrücke, Anweisungen, ...)
- Ausführungsrelation

## Modellierung von Semantik (2)

Im folgenden:

- Einfache imperative Sprache (“while-Sprache”)
- Keine komplexe Speicherstruktur (Pointer etc.)
- Kein komplexer Kontrollfluß (Prozeduren, Exceptions etc.)

Es geht auch besser, z.B. Modellierung der Semantik von Java

↪ **Semantik-Vorlesung nächstes Semester**

# Zustände

Speicherzellen: uninterpretiert

**typedcl** *loc*

Werte: hier nur nat. Zahlen

Zustand:

**types**

*state = "loc  $\Rightarrow$  nat"*

# Programmiersprache: Ausdrücke

**datatype** *binop* = *BPlus* | *BMinus* | *BEq*

**datatype** *expr*

= *N nat* — constant

| *Var loc* — variable

| *Op binop expr expr*

## Programmiersprache: Anweisungen

**datatype** *com*

= *SKIP*

| *Assign loc expr* ("\_ ::= \_ ")

| *Semi com com* ("\_; \_")

| *Cond expr com com* ("IF \_ THEN \_ ELSE \_")

| *While expr com* ("WHILE \_ DO \_")

### Beispiel:

Abstrakte Syntax:

```
"WHILE (Var y) DO
  (x ::= (Op BPlus (Var y) (N 2)));
  y ::= (Op BPlus (Var y) (N 1)))"
```

... steht für:

```
WHILE y DO {
  x = y + 2;
  y = y + 1;
}
```

## Evaluierungs- und Ausführungsrelationen

Evaluierung von Ausdrücken:

**consts**

$eval_e :: "[expr, state] \Rightarrow nat"$

... zu definieren

Ausführungsrelation für Anweisungen: (warum keine Funktion?)

**consts**  $eval_c :: "(com \times state \times state) set"$

Geschrieben als:

**translations**  $"\langle c, s \rangle \longrightarrow_c s'" == "(c, s, s') \in eval_c"$

## Ausführungsrelation: einige Regeln

### “Big-Step”-Semantik

#### inductive evalc intros

*Assign*: " $\langle x := a, s \rangle \longrightarrow_c s(x := (\text{evale } a \ s))$ "

*Semi*: " $\llbracket \langle c_0, s \rangle \longrightarrow_c s''; \langle c_1, s'' \rangle \longrightarrow_c s' \rrbracket$   
 $\implies \langle c_0; c_1, s \rangle \longrightarrow_c s'$ "

*WhileFalse*: " $\text{evale } b \ s = 0$   
 $\implies \langle \text{WHILE } b \ \text{DO } c, s \rangle \longrightarrow_c s$ "

*WhileTrue*: " $\llbracket \text{evale } b \ s \neq 0; \langle c, s \rangle \longrightarrow_c s''; \langle \text{WHILE } b \ \text{DO } c, s'' \rangle \longrightarrow_c s' \rrbracket$   
 $\implies \langle \text{WHILE } b \ \text{DO } c, s \rangle \longrightarrow_c s'$ "

## Ausführungsrelation: Beispiel

Anfangszustand: Alle Variablen auf 0

**lemma** " $m \neq n \implies p \neq m \implies p \neq n \implies$   
 $\langle (m ::= (N\ 3));$   
 $(n ::= (N\ 2));$   
 $(WHILE\ (Var\ n)\ DO$   
 $(p ::= (Op\ BPlus\ (Var\ p)\ (Var\ m)));$   
 $n ::= (Op\ BMinus\ (Var\ n)\ (N\ 1))))))$ ,  
 $\lambda\ v.\ 0 \rangle \longrightarrow_c\ ?s'$ "

- Modellierung von Programmiersprachen-Semantik
- **Projekt: Datenflussanalyse**

## Lebendige Variablen: Begriff

Variable  $v$  ist **lebendig** an einem Programmpunkt  $P$ , wenn

- in jedem weiteren möglichen Programmverlauf auf die Variable lesend zugegriffen wird
- ohne daß  $v$  vorher überschrieben worden ist.

Bsp.:

$x = 3;$  (1)

$y = 5;$  (2)

$y = x + 3;$  (3)

$x = x + y;$  (4)

$x$  lebendig (mindestens) nach (1), (2), (3)

$y$  lebendig (mindestens) nach (3)

$y$  nicht lebendig nach (2)

## Lebendigkeitsanalyse: Verwendung

### Compileroptimierung:

Zuweisungen an nicht-lebendige Variablen können gestrichen werden (“dead code elimination”)

$x = 3;$  (1)

$y = 5;$  (2)

$y = x + 3;$  (3)

$x = x + y;$  (4)

$x = 3;$  (1)

$y = x + 3;$  (3)

$x = x + y;$  (4)

Korrektheitsnachweis?

## Lebendige Variablen: Verfeinerung

Frage: Ist  $x$  lebendig nach (4)?

· · ·  
 $x = x + y; \quad (4)$   
· · ·

Antwort:

- *Ja*, wenn  $x$  noch verwendet wird,  
z.B. in `return(x + y)`
- *Nein*, wenn  $x$  nicht mehr verwendet wird,  
z.B. in `return(y)`

↪ Betrachte Lebendigkeit bzgl. Menge **relevanter Variablen**

## Äquivalenz bzgl. relevanter Variablen (1)

Äquivalenz zweier Funktionen auf Menge  $A$ :

```
equiv_on :: "[ 'a set, 'a  $\Rightarrow$  'b, 'a  $\Rightarrow$  'b ]  $\Rightarrow$  bool"
```

```
"equiv_on A f1 f2 ==  $\forall x \in A. f1 x = f2 x$ "
```

[Funktionen hier: Programmzustände]

## Äquivalenz bzgl. relevanter Variablen (2)

<i>Original</i>		<i>Optimierung 1</i>		<i>Optimierung 2</i>	
$x = 3;$	(1)	$x = 3;$	(1)	$x = 3;$	(1)
$y = 5;$	(2)				
$y = x + 3;$	(3)	$y = x + 3;$	(3)	$y = x + 3;$	(3)
$x = x + y;$	(4)	$x = x + y;$	(4)		
<i>s_orig</i>		<i>s_opt1</i>		<i>s_opt2</i>	

Es gilt:

*equiv\_on {x,y} s\_orig s\_opt1*

*equiv\_on {y} s\_orig s\_opt2*

Es gilt nicht:

*equiv\_on {x} s\_orig s\_opt2*

## Lebendige Variablen: Komplikationen (1)

Ist  $y$  lebendig nach (1),  
da  $y$  in (2) nicht gebraucht wird?  
Ist folgende Optimierung korrekt?

<i>Original</i>		<i>Optimierung</i>	
$y = 5;$	(1)		
IF ( $x > 0$ )		IF ( $x > 0$ )	
THEN $x = x + 1$	(2)	THEN $x = x + 1$	(2)
ELSE $x = x + y$	(3)	ELSE $x = x + y$	(3)
return( $x$ )		return( $x$ )	

## Lebendige Variablen: Komplikationen (2)

Ist  $x$  lebendig nach (3),  
da  $x$  nach Ende der Schleife nicht mehr gebraucht wird?  
Ist folgende Optimierung korrekt?

<i>Original</i>		<i>Original</i>	
$x = 2;$	(1)	$x = 2;$	(1)
$y = 5;$	(2)	$y = 5;$	(2)
WHILE $x > 0$ DO		WHILE $x > 0$ DO	
$x = x - 1;$	(3)	SKIP;	(3)
return( $y$ )		return( $y$ )	

# Datenflußanalyse (1)

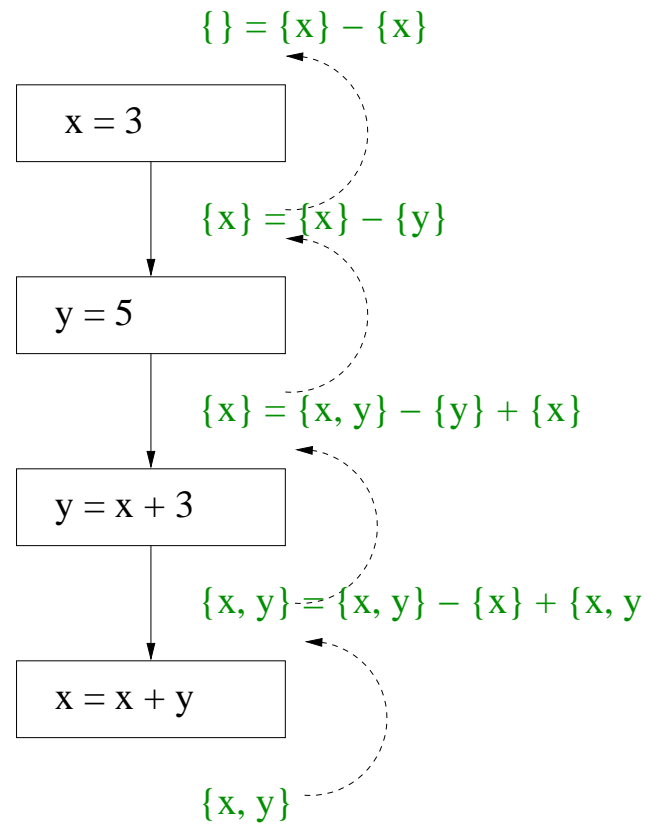
Beachte alle Pfade im Kontrollflußgraphen!

**Vorgehen:** Traversiere Kontrollflußgraphen von hinten nach vorne

- Bekannt: Lebendige Variablen  $A$  *nach* Anweisung
- Berechne: Lebendige Variablen  $B$  *vor* Anweisung

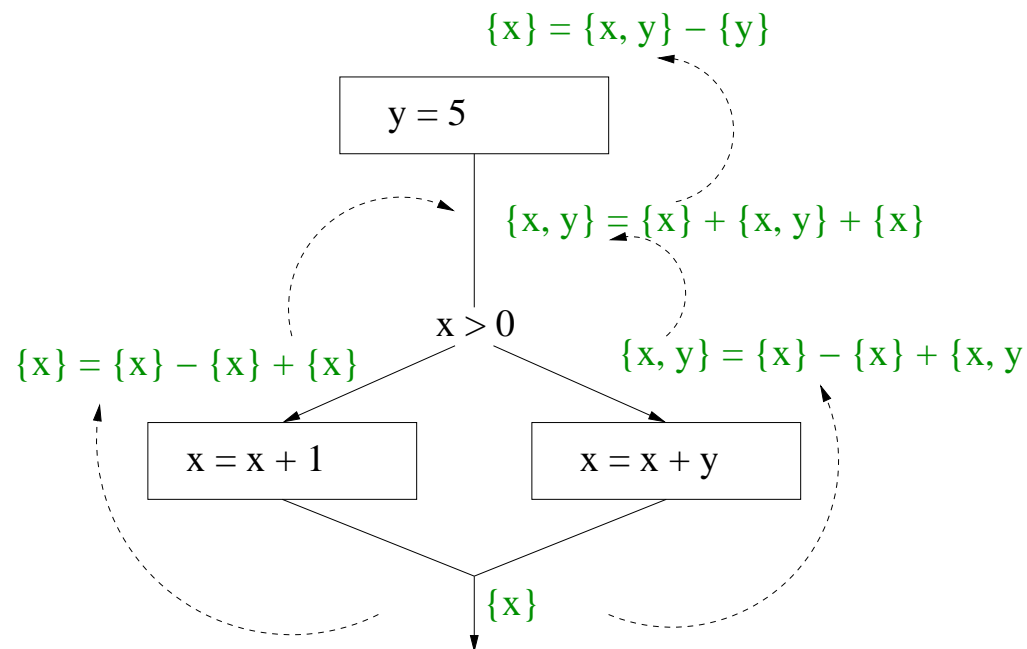
## Datenflußanalyse (2)

### Kontrollfluß ohne Verzweigung



## Datenflußanalyse (3)

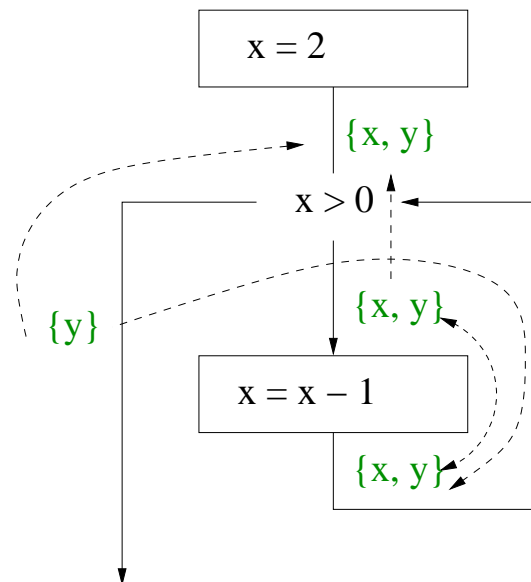
### Kontrollfluß mit Verzweigung (Conditional)



**Prinzip:** Schätze Menge lebendiger Variablen *konservativ* ab  
 “Lieber zu viele Variablen als lebendig bezeichnen als zu wenige”

## Datenflußanalyse (4)

### Schleife



*Frage:* Warum muss  $x$  in der Schleife lebendig sein?

## Analyse von Schleifen (1)

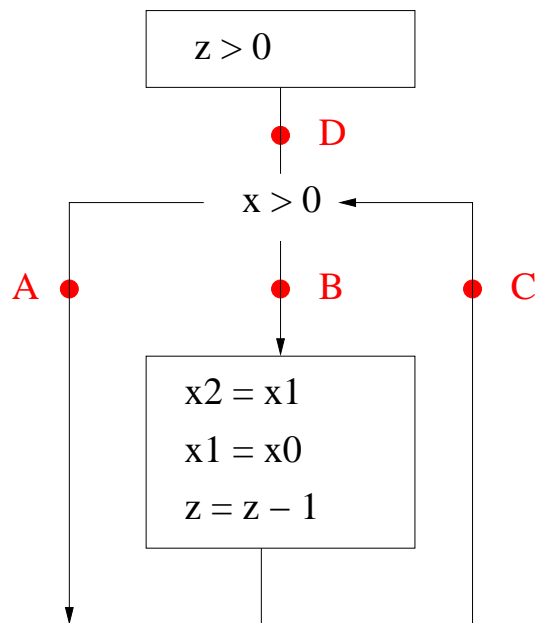
Warum genügt einmalige Traversierung nicht?

*Bsp:* Kann Zuweisung (2) gelöscht werden?

```
WHILE z > 0 DO  
  x2 = x1;      (1)  
  x1 = x0;      (2)  
  z = z - 1;    (3)  
return(x2)
```

## Analyse von Schleifen (2)

Datenfluß-Gleichungen:



Bestimme kleinste Lösung von:

$$\{x_2\} \subseteq A$$

$$B \subseteq C - \{x_2\} \cup \{x_0, x_1\}$$

$$C \subseteq A \cup B \cup \{z\}$$

$$D \subseteq A \cup B \cup \{z\}$$

Damit insbesondere:

$$\{x_0, x_1\} \subseteq C$$

## Analyse von Schleifen (3)

Vereinfachung:

Gleichungen:

$$B = A \cup FV_c \cup FV_e$$

$$D = A \cup B \cup FV_e$$

wobei

$FV_c$  die freien Variablen des Schleifenkörpers

$FV_e$  die freien Variablen der Schleifenbedingung

*Damit:* Analyse etwas ungenauer, aber immer noch korrekt

Zu diesem Thema interessantes SEP zu vergeben!

# Datenflußanalyse und Optimierung

**Aufgabe:** Definition einer Funktion

$optimLV :: "[com, loc\ set] \Rightarrow com \times (loc\ set)"$

Sei

- $c$  ein zu optimierendes Programm
- $A$  die Menge der lebendigen Variablen *nach* Ausführung von  $c$

Dann gilt für  $optimLV\ c\ A = (c', B)$

- $c'$  ist das optimierte Programm
- $B$  die Menge der lebendigen Variablen *vor* Ausführung von  $c$

## Korrektheitsaussage

Das nicht-optimierte Programm  $c$   
 und das optimierte Programm  $c'$   
 sind äquivalent bezüglich der Menge lebendiger Variablen  $A$ .

**lemma** *optim\_correct*: "  
 $\llbracket \langle c, s \rangle \longrightarrow_c s'; \text{optimLV } c \ A = (c', B); \langle c', s \rangle \longrightarrow_c so' \rrbracket \implies$   
*equiv\_on*  $A \ s' \ so'$ "

## Beweis

- Induktion über Ausführungsrelation des nicht-optimierten Programms
- “Nachziehen” des optimierten Programms

Teilfall Komposition mit  $c$  als  $c1; c2$

