

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 , λ , ...

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 | λ | \neg | \wedge | \vee | \longrightarrow | \Rightarrow |
| ASCII | <code>\<forall></code> | <code>\<exists></code> | <code>\<lambda></code> | <code>\<not></code> | <code>/\</code> | <code>\/</code> | <code>--></code> | <code>=></code> |
| ASCII(alt) | ALL | EX | % | ~ | & | | --> | => |

Isabelle/HOL: Typen

Syntax:

| | | | |
|--------|-------|-------------------------|---|
| τ | $::=$ | (τ) | |
| | | $bool$ | 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)

λ -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 (β -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