

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 *typedef*, z.B.:

typedef *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 $C :: (cantor \Rightarrow bool) \Rightarrow cantor$ 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 τ_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 τ_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:

$$f\ x\ y ==\ x + y + 2\ \text{und}$$

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

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