Formale Modellierung
Vorlesung 9 vom 03.06.13: Weitere Datentypen: Mengen, Multimengen, Punkte

Serge Autexier & Christoph Lüth

Universität Bremen

Sommersemester 2013

Fahrplan

- Teil I: Formale Logik
  - Einführung
  - Aussagenlogik: Syntax und Semantik, Natürliches Schließen
  - Konsistenz & Vollständigkeit der Aussagenlogik
  - Prädikatenlogik (FOL): Syntax und Semantik
  - Konsistenz & Vollständigkeit von FOL
  - FOL mit induktiven Datentypen
  - FOL mit Induktion und Rekursion
  - Die Gödel-Theoreme
  - Weitere Datentypen: Mengen, Multimengen, Punkte
- Teil II: Spezifikation und Verifikation
- Teil III: Schluß

Das Tagesmenü

- Isabelle/HOL: sicheres spezifizieren (konservative Erweiterung)
- typedecl, definition, typedef, datatype, primrec, fun, function
- typedef
  - Mengen, Multimengen, Rat
- datatype
  - Listen, Listen ohne Wiederholung, Punkte & Vektoren 3D

Mengen

- Mengen über beliebigen Typ t: Set_t
- Characterisiert über Prädikat $P : t \to o$

$$\begin{aligned} \mathsf{Collect} &: (t \to o) \to \mathsf{Set}_t \\ \in &: t \times \mathsf{Set}_t \to o \\ x \in (\mathsf{Collect}\, P) &= Px \end{aligned}$$

- Set_t nicht leer wenn Funktionenraum $t \to o$ nicht leer.

$$\begin{aligned} &\bot : t \to o \\ &\forall x_t.\neg\bot(x) \\ &\emptyset_t \in \mathsf{Set}_t \\ &\subseteq : \mathsf{Set}_t \times \mathsf{Set}_t \to o \\ &A \subseteq B \longleftrightarrow \forall x_t.x \in A \longrightarrow x \in B \\ &\mathsf{Power} : \mathsf{Set}_t \to \mathsf{Set}_{\mathsf{Set}_t} \\ &\mathsf{Power}\, A = \underbrace{\mathsf{Collect}(\lambda x_{\mathsf{Set}_t}.x \subseteq A)}_{\{x:\mathsf{Set}_t \mid x \subseteq A\}} \end{aligned}$$

In Isabelle

```
typedecl 'a set
axiomatization Collect :: "('a => bool) => 'a set" -- "comprehension"
  and member :: "'a => 'a set => bool" -- "membership"
where
  mem_Collect_eq [iff, code_unfold]: "member a (Collect P) = P a"
  and Collect_mem_eq [simp]: "Collect (%x. member x A) = A"
```


Definitionen

- Definiere nicht-rekursive Konstanten

$$\mathsf{nand} : o \to o \to o \qquad \mathsf{nand}\, xy = (\neg x) \wedge (\neg y)$$

- Definitionen sind konservative Erweiterungen
- In Isabelle/HOL

```
definition nand :: "bool => bool => bool" where
  "nand x y == (~ x) & ~ y"
```


Typdefinition

- Spezifiziere nicht-leere Teilmenge eines gegebenen Typs r
- Deklariere neuen Typ t mit Trägermenge isomorph zu Werten in spezifizierter Teilmenge
- Isomorphie wird durch inverse Funktionen $\mathsf{Abs}_t : r \to t, \mathsf{Rep}_t : t \to r$ axiomatisch Beschrieben

Typdefinitionen sind Erweiterungen

Definition 1 (Typdefinitionen)

Sei $S = ((\mathcal{T}, \mathcal{F}, \mathcal{P}), \Phi)$ eine Spezifikation, $r \in \mathcal{T}$ und $P \in \mathcal{F}orm_\Sigma$ mit genau einer freien Variable vom Typ r. Dann ist eine Erweiterung $S' = ((\mathcal{T}', \mathcal{F}', \mathcal{P}'), \Phi')$ eine Typdefinition für einen Typ $t \notin \mathcal{T}$ gdw.

- $\mathcal{T}' = \mathcal{T} \cup \{t\}$
- $\mathcal{F}' = \mathcal{F} \cup \{\mathsf{Abs}_t : r \to t, \mathsf{Rep}_t : t \to r\}$
- $\mathcal{P}' = \mathcal{P} \cup \{=_t : t \times t\}$
- $\Phi' = \Phi \cup \{\ \forall x_t.\, \mathsf{Abs}_t(\mathsf{Rep}_t(x)) =_t x,$
  $\forall x_r.P(x_r) \longrightarrow \mathsf{Rep}_t(\mathsf{Abs}_t(x)) =_r x\}$
- Man kann beweisen $S \vdash \exists x_r.P(x)$ (bzw. es gilt $\exists x_r.P(x) \in \mathsf{Th}(S)$)

Beispiel: Paare

- Paare über Typen s und t

$$\text{Pair\_Rep} : s \to t \to (s \to t \to o)$$
$$\text{Pair\_Rep}(a, b) = \lambda x_s \lambda y_t . x = a \wedge y = b$$

- In Isabelle/HOL

```
definition Pair_Rep :: "'a => 'b => 'a => 'b => bool" where
  "Pair_Rep a b = (%x y. x = a & y = b)"

definition "prod = {f.  ex a b. f = Pair_Rep (a::'a) (b::'b)}"

typedef ('a, 'b) prod (infixr "*" 20) =
   "prod :: ('a => 'b => bool) set"

type_notation (xsymbols)
  "prod"  ("(_ x/ _)" [21, 20] 20)
type_notation (HTML output)
  "prod"  ("(_ x/ _)" [21, 20] 20)

definition Pair :: "'a => 'b => 'a 'b prod"  where
  "Pair a b = Abs_prod (Pair_Rep a b)"
```


Datentypen

- Datentypen sind Typdefinition mit Definition der Konstruktoren

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

- Case-Expressions für Datentypen

```
case o of None -> 1 | o of Some(n) -> x
```

- Induktive Datentypen

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

- Generierte Taktiken:
  - `case_tac`: Strukturelle Fallunterscheidung
  - `induct_tac`: Strukturelle Induktion

Rekursive Funktionen

- Drei Möglichkeiten rekursive Funktionen zu definieren:
  - `primrec`
    Strukturelle Fallunterscheidung und Rekursion
  - `fun`
    Totale, terminierende rekursive Funktionen mit Terminierungsbeweis

Erweiterung um Totale, Terminierende Funktionen is Konservativ

Definition 2 (Funktions- und Prädikatsdefinitionen)

Sei $S = ((\mathcal{T}, \mathcal{F}, \mathcal{P}), \Phi)$ eine Spezifikation, $f : \tau_1 \times \cdots \times \tau_n \to \tau_0 \notin \mathcal{F}$ ($\tau_i \in \mathcal{T}$) und $\Psi \in \mathcal{F}orm_{\Sigma \cup \{f\}}$. Dann ist eine Erweiterung $S' = ((\mathcal{T}, \mathcal{F}', \mathcal{P}), \Phi')$ eine Funktionsdefinition gdw.

- Ψ ist eine eindeutig und totale Definition für f
- f ist terminierend und alle in der Definition von f vorkommenden Funktionen und Prädikate sind terminierend
- $\Phi' = \Phi \cup \Psi$
- $\mathcal{F}' = \mathcal{F} \cup \{f : \tau_1 \times \cdots \times \tau_n \to \tau_0\}$

Analog für Prädikatsdefinitionen.

Lemma 3

Funktionsdefinitionen bzw. Prädikatsdefinitionen sind konservativ

Strukturelle Rekursive Funktionen

- case-artige definitions patterns
- Genau eine Schicht Konstruktoren

```
primrec append :: "'a list => 'a list => 'a list" (infixr "@" 65) where
append_Nil: "[] @ ys = ys" |
append_Cons: "(x#xs) @ ys = x # xs @ ys"
```

- Nicht aber

```
primrec myhalf :: "nat => nat" where
"myhalf 0 = 0" |
"myhalf (Suc 0) = 0" |
"myhalf (Suc (Suc x)) = Suc (myhalf x)"
```


Allgemeine Totale rekursive Funktionen

- `fun` zur Definition allgemeiner rekursiver Funktionen

```
fun myhalf :: "nat => nat" where
"myhalf 0 = 0" |
"myhalf (Suc 0) = 0" |
"myhalf (Suc (Suc x)) = Suc (myhalf x)"
```

- Automatischer Terminierungsbeweiser
- Definition wird verweigert, falls Terminierungsbeweis mislingt

Terminierung

- Beispiele:
  - half(x) eine Hypothese pro Rekursionsgleichung
  - fib(x): mehrere Hypothesen pro Rekursionsgleichung
  - gcd(x, y): lexicographische Ordnung
- Beweise alle Hypothesen im Kalkül. Terminierung gilt relativ zur Terminierung der anderen involvierten Funktionen und Prädikate.
- Analog für Prädikate auf $\mathbb{N}$ mit bedingten Äquivalenzen
- Allgemeine Typen: für frei erzeugte Datentypen kann Abbildung in natürliche Zahlen definiert werden, die die Anzahl der Konstruktoren zählt. Damit lässt sich das Terminierungsverfahren auf all frei erzeugten Datentypen erweitern

Quotienten Typen

- Gegeben natürliche Zahlen $\mathbb{N}$
- Rationale Zahl ist Paar nat. Zahlen (n, m) so dass $m \neq 0$

```
definition israt :: "(nat * nat) => bool" where
"israt = (%x . -((snd x) = 0)"
```

- typedef führt nicht zu gewünschtem Ergebnis

```
typedef Rat = "{x . israt(x)}"
```

- Braucht Gleichheit der Paare zusätzlich

```
definition ratrel :: "(nat * nat) =>  (nat * nat) => bool" where
"ratrel =  (%x y .-(snd x = 0) & -(snd y = 0) &
                  snd x * fst y = fst x * snd y)"
```

- Quotienten Typen:

```
quotient_type rat = "nat * nat" / partial: "ratrel"
```

Integers

```
definition intrel :: "(nat * nat) => (nat * nat) => bool" where
  "intrel = (%(x, y) (u, v). x + v = u + y)"

quotient_type int = "nat * nat" / "intrel"
```


Multimengen

```
typedef 'a mset = "{f::'a => nat . True}"
 apply (auto)
done

definition emptyMSet :: "'a mset" where
"emptyMSet = Abs_mset(%x.0)"

definition unionMSet :: "'a mset => 'a mset => 'a mset" where
"unionMSet =
 (%m1 m2. Abs_mset (%n.((Rep_mset m1 n) + (Rep_mset m2 n))))"

definition interMSet :: "'a mset => 'a mset => 'a mset" where
"interMSet =
 (%m1 m2. Abs_mset (%n.(min (Rep_mset m1 n) (Rep_mset m2 n))))"
```


Punkte und Mehr

- Punkte im 3D

```
typedef point3D = "{p::(int * int * int).True}"
 apply (auto)
done

definition origin :: "point3D" where
"origin = (Abs_point3D (0,0,0))"
```

- Vektoren 3D

```
datatype Vektor3D = Vektor int * int * int

definition skalarProdukt :: "Vektor3D => Vektor3D => Vektor3D" where
"skalarProdukt = (% v1 v2 .
  case (v1,v2) of ((Vektor x y z), (Vektor u v w))
    => .."

definition point2Vektor :: "point3D => point3D => Vektor3D" where
...
```


Listen ohne doppelte Vorkommen

- Listen ohne doppelte Vorkommen

```
primrec contains :: "'a list => 'a => bool" where
"contains Nil x = False" |
"contains (Cons x xs) y = ((x = y) or contains xs x)"

fun count :: "'a => 'a list => nat" where
"count x Nil = 0" |
"count x (Cons y xs) = (if (x=y) then (Suc (count x xs))
                        else (count x xs))"


definition noDuplicates :: "'a list => bool" where
"noDuplicates = (%l::'a list.\<forall>x::'a.(contains l x)
                                --> (count x l) = (Suc 0))"

typedef 'a setlist = "{l::'a list . noDuplicates l}"
```


Zusammenfassung

- Isabelle/HOL
- Primitiven vordefiniert zur sicheren Einführung neuer Typen und Funktionen
  - `typedef, datatype, quotient_type`
  - `primrec, fun`
- Typen
  - Paare
  - Option, Listen
  - Rat, Int
  - Multimengen
  - Punkte und Vektoren im 3D
  - Listen ohne doppelte Vorkommen