Rechnerübung zu Theorembeweiser und ihre Anwendungen

Prof. Dr.-Ing. Gregor Snelting
Dipl.-Inf. Univ. Daniel Wasserrab

Lehrstuhl Programmierparadigmen
IPD Snelting
Universität Karlsruhe (TH)

Teil IV

Rekursive Datentypen und primitive Rekursion

Deklaration von Funktionen

Festlegen von Namen und Signatur einer Funktion: Deklaration
dazu in Isabelle das Schlüsselwort **consts:**

consts *length :: "'a list ⇒ nat"*

Vorsicht! Gibt der Funktion noch keinerlei Semantik!
Wird in den Übungen verwendet, um Funktionen einzuführen, die sie selbst noch definieren (also mit Semantik versehen) sollen

Rekursive Datentypen

Viele Datentypen mit Selbstbezug, z.B.

- natürliche Zahl (ungleich 0) ist Nachfolger einer natürlichen Zahl
- nichtleere Liste ist Liste mit zusätzlichem Kopfelement
- nichtleere Menge ist Menge mit einem zusätzlichen Element

Formalisierung in Isabelle/HOL am Bsp. natürliche Zahlen:

datatype *nat = 0 | Suc nat*

Also Konstruktoren von nat: 0 und Suc (Präfix)

Rekursive Datentypen

Viele Datentypen mit Selbstbezug, z.B.

- natürliche Zahl (ungleich 0) ist Nachfolger einer natürlichen Zahl
- nichtleere Liste ist Liste mit zusätzlichem Kopfelement
- nichtleere Menge ist Menge mit einem zusätzlichen Element

Formalisierung in Isabelle/HOL am Bsp. natürliche Zahlen:

datatype *nat = 0 | Suc nat*

Also Konstruktoren von nat: 0 und Suc (Präfix)

Parametertypen

verschiedene Typen in Containerdatentypen: Parametertyp `'a`
kann bei Verwendung entprechen initialisiert werden (muss aber nicht)

Beispiel: Listen mit Typparameter

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

Konstruktoren von list: [] und # (Infix) (x#[] = [x])

Funktionsdeklaration kann jetzt z.B. so aussehen:

```
consts foo :: "nat list ⇒ bool"
consts bar :: "nat ⇒ bool list ⇒ nat"
consts zip :: "'a list ⇒ 'a"
```

Parametertypen

verschiedene Typen in Containerdatentypen: Parametertyp `'a`
kann bei Verwendung entprechen initialisiert werden (muss aber nicht)

Beispiel: Listen mit Typparameter

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

Konstruktoren von list: [] und # (Infix) (x#[] = [x])

Funktionsdeklaration kann jetzt z.B. so aussehen:

```
consts foo :: "nat list ⇒ bool"
consts bar :: "nat ⇒ bool list ⇒ nat"
consts zip :: "'a list ⇒ 'a"
```

Parametertypen

verschiedene Typen in Containerdatentypen: Parametertyp *'a*
kann bei Verwendung entprechen initialisiert werden (muss aber nicht)

Beispiel: Listen mit Typparameter

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

Konstruktoren von list: [] und # (Infix) (x#[] = [x])

Funktionsdeklaration kann jetzt z.B. so aussehen:

```
consts foo :: "nat list ⇒ bool"
consts bar :: "nat ⇒ bool list ⇒ nat"
consts zip :: "'a list ⇒ 'a"
```

primrec

Definition von Funktionen über rekursive Datentypen: **primrec**
kombiniert Deklaration und Definition (frühere **consts** löschen!)
ein Parameter der Funktion muss in seine Konstruktoren aufgeteilt werden
Beispiel:

primrec *length :: "'a list ⇒ nat"*
  where *"length [] = 0"*
  | "length (x#xs) = Suc(length xs)"

primrec *tl :: "'a list ⇒ 'a list"*
  where *"tl [] = []"*
  |"tl (x#xs) = xs"

primrec

Es müssen nicht alle Konstruktoren spezifiziert werden:

```
primrec hd :: "'a list ⇒ 'a"
  where "hd(x#xs) = x"

primrec last :: "'a list ⇒ 'a"
  where "last(x#xs) = (if xs=[] then x else last xs)"
```

@ und rev

weiterer Infixoperator: `@` hängt Listen zusammen
Beispiel: `[0,4]@[2] = [0,4,2]`

`rev` dreht Listen um, also `rev [0,4,2] = [2,4,0]`
Wie lautet die entsprechende Deklaration/Definition?

@ und rev

weiterer Infixoperator: `@` hängt Listen zusammen
Beispiel: `[0,4]@[2] = [0,4,2]`

`rev` dreht Listen um, also `rev [0,4,2] = [2,4,0]`
Wie lautet die entsprechende Deklaration/Definition?

@ und rev

weiterer Infixoperator: `@` hängt Listen zusammen
Beispiel: *[0,4]@[2] = [0,4,2]*

rev dreht Listen um, also *rev [0,4,2] = [2,4,0]*
Wie lautet die entsprechende Deklaration/Definition?

primrec *rev :: "'a list ⇒ 'a list"*
  where *"rev [] = []"*
  | "rev(x#xs) = rev(xs) @ [x]"

Strukturelle Induktion

Beweise über rekursive Datentypen mittels strukturreller Induktion
d.h. Induktion über Konstruktoren des Datentyps

In Isabelle/HOL:
lemma *hd_Cons_tl: "xs* $\neq$ *[]* $\Longrightarrow$ *hd xs # tl xs = xs"*
apply *(induct xs)*
apply *auto*
done
wendet strukturelle Induktion mit Datentypkonstruktoren von *xs* an
automatische Taktik beendet Beweis

Probleme mit Induktion

Problem: zu spezielle Induktionshypothesen

lemma *"(rev xs = rev ys) = (xs = ys)"*
Induktion auf *xs* ermöglicht Lösen des *[]*-Falles
bei Induktionsschritt bleibt:

$\bigwedge$a *xs*.
    (rev xs = rev ys) = (xs = ys) $\Longrightarrow$
    (rev (a # xs) = rev ys) = (a # xs = ys)

Probleme mit Induktion

Problem: zu spezielle Induktionshypothesen

lemma *"(rev xs = rev ys) = (xs = ys)"*
Induktion auf *xs* ermöglicht Lösen des *[]*-Falles
bei Induktionsschritt bleibt:

$\bigwedge$a *xs*.
  (rev xs = rev ys) = (xs = ys) $\Longrightarrow$
  (rev (a # xs) = rev ys) = (a # xs = ys)

nicht lösbar!
ys kann nicht gleich *xs* und *a # xs* sein!

Probleme mit Induktion

Idee: *ys* muss im Induktionsschritt freie Variable sein!
Lösung: *ys* nach *arbitrary* Schlüsselwort in Induktionsanweisung damit Induktionsschritt für *ys* meta-allquantifiziert:
apply *(induct xs arbitrary:ys)*

Resultiert in Induktionsschritt:

$$\bigwedge a\ xs\ ys.\ (\bigwedge ys.\ (rev\ xs = rev\ ys) = (xs = ys)) \Longrightarrow (rev\ (a\ \#\ xs) = rev\ ys) = (a\ \#\ xs = ys)$$

Heuristiken für (bisher scheiternde) Induktionen:

- alle freien Variablen (außer Induktionsvariable) mit *arbitrary*
- Induktion immer über das Argument, über das die Funktion rekursiv definiert ist
- Ziele durch Ersetzen von Konstanten durch Variablen generalisieren

Probleme mit Induktion

Idee: *ys* muss im Induktionsschritt freie Variable sein!
Lösung: *ys* nach *arbitrary* Schlüsselwort in Induktionsanweisung damit Induktionsschritt für *ys* meta-allquantifiziert:
apply *(induct xs arbitrary:ys)*

Resultiert in Induktionsschritt:

$$\bigwedge a\ xs\ ys.\ (\bigwedge ys.\ (rev\ xs = rev\ ys) = (xs = ys)) \Longrightarrow (rev\ (a\ \#\ xs) = rev\ ys) = (a\ \#\ xs = ys)$$

Heuristiken für (bisher scheiternde) Induktionen:

- alle freien Variablen (außer Induktionsvariable) mit *arbitrary*
- Induktion immer über das Argument, über das die Funktion rekursiv definiert ist
- Ziele durch Ersetzen von Konstanten durch Variablen generalisieren

Probleme mit Induktion

Idee: *ys* muss im Induktionsschritt freie Variable sein!
Lösung: *ys* nach *arbitrary* Schlüsselwort in Induktionsanweisung damit Induktionsschritt für *ys* meta-allquantifiziert:
apply *(induct xs arbitrary:ys)*

Resultiert in Induktionsschritt:

$$\bigwedge a\ xs\ ys.\ (\bigwedge ys.\ (rev\ xs = rev\ ys) = (xs = ys)) \Longrightarrow (rev\ (a\ \#\ xs) = rev\ ys) = (a\ \#\ xs = ys)$$

Heuristiken für (bisher scheiternde) Induktionen:

- alle freien Variablen (außer Induktionsvariable) mit *arbitrary*
- Induktion immer über das Argument, über das die Funktion rekursiv definiert ist
- Ziele durch Ersetzen von Konstanten durch Variablen generalisieren