Rechnerübung zu Theorembeweiser und ihre Anwendungen

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

Lehrstuhl Programmierparadigmen
IPD Snelting
Universität Karlsruhe (TH)

Teil VII

Allgemeine Rekursion

Allgemeine Rekursion

oftmals ist primitive Rekursion mit einer Regel pro Konstruktor zu einschränkend
manche rekursiven Definitionen haben z.B. zwei Basisfälle oder brauchen Rekursion in mehr als einem Parameter

Beispiel: Fibonacci-Zahlen
mit *primrec* (auf einfache Weise) nicht möglich!

Allgemeine Rekursion

oftmals ist primitive Rekursion mit einer Regel pro Konstruktor zu einschränkend
manche rekursiven Definitionen haben z.B. zwei Basisfälle oder brauchen Rekursion in mehr als einem Parameter

Beispiel: Fibonacci-Zahlen
mit *primrec* (auf einfache Weise) nicht möglich!

fun

Lösung: **fun**!
ermöglicht "mächtigere" Rekursion, so z.B. mehrere Basisfälle oder Rekursion in mehreren Parametern
Syntax: analog zu *primrec*

Beispiel "mehrere Basisfälle": Fibonacci-Zahlen

```
fun fib :: "nat ⇒ nat"
  where "fib 0 = 1"
  | "fib (Suc 0) = 1"
  | "fib (Suc (Suc n)) = fib n + fib (Suc n)"
```

Beispiel "Rekursion in mehreren Parametern": Zippen von Listen

```
fun zip :: "'a list ⇒ 'b list ⇒ ('a × 'b) list"
  where "zip [] [] = []"
  | "zip (a#as) (b#bs) = (a,b)#zip as bs"
```

fun

Lösung: **fun**!
ermöglicht "mächtigere" Rekursion, so z.B. mehrere Basisfälle oder Rekursion in mehreren Parametern
Syntax: analog zu *primrec*

Beispiel "mehrere Basisfälle": Fibonacci-Zahlen

```
fun fib :: "nat ⇒ nat"
  where "fib 0 = 1"
  | "fib (Suc 0) = 1"
  | "fib (Suc (Suc n)) = fib n + fib (Suc n)"
```

Beispiel "Rekursion in mehreren Parametern": Zippen von Listen

```
fun zip :: "'a list ⇒ 'b list ⇒ ('a × 'b) list"
  where "zip [] [] = []"
  | "zip (a#as) (b#bs) = (a,b)#zip as bs"
```

fun

Lösung: **fun**!
ermöglicht "mächtigere" Rekursion, so z.B. mehrere Basisfälle oder Rekursion in mehreren Parametern
Syntax: analog zu *primrec*

Beispiel "mehrere Basisfälle": Fibonacci-Zahlen

```
fun fib :: "nat ⇒ nat"
  where "fib 0 = 1"
  | "fib (Suc 0) = 1"
  | "fib (Suc (Suc n)) = fib n + fib (Suc n)"
```

Beispiel "Rekursion in mehreren Parametern": Zippen von Listen

```
fun zip :: "'a list ⇒ 'b list ⇒ ('a × 'b) list"
  where "zip [] [] = []"
  | "zip (a#as) (b#bs) = (a,b)#zip as bs"
```

fun

fun definiert Funktionen durch Pattern Matching
"lineare Patterns": unterschiedliche Variablen auf den linken Seiten

möglich: Überlappen von Patterns
dann Reihenfolge bestimmend
es wird immer die erste passende Regel angewandt
damit möglich: default-Regel, die alle restlichen Fälle beinhaltet

Beispiel: zwischen je zwei Elemente einer Liste Seperatorzeichen einfügen

```
fun sep :: "'a => 'a list ⇒ 'a list"
where "sep a (x#y#zs) = x#a#sep a (y#zs)"
  |"sep a xs          = xs"
```

fun

fun definiert Funktionen durch Pattern Matching
"lineare Patterns": unterschiedliche Variablen auf den linken Seiten

möglich: Überlappen von Patterns
dann Reihenfolge bestimmend
es wird immer die erste passende Regel angewandt
damit möglich: default-Regel, die alle restlichen Fälle beinhaltet

Beispiel: zwischen je zwei Elemente einer Liste Seperatorzeichen einfügen

```
fun sep :: "'a => 'a list ⇒ 'a list"
where "sep a (x#y#zs) = x#a#sep a (y#zs)"
  |"sep a xs          = xs"
```

fun

fun definiert Funktionen durch Pattern Matching
"lineare Patterns": unterschiedliche Variablen auf den linken Seiten

möglich: Überlappen von Patterns
dann Reihenfolge bestimmend
es wird immer die erste passende Regel angewandt
damit möglich: default-Regel, die alle restlichen Fälle beinhaltet

Beispiel: zwischen je zwei Elemente einer Liste Seperatorzeichen einfügen

```
fun sep :: "'a => 'a list ⇒ 'a list"
where "sep a (x#y#zs) = x#a#sep a (y#zs)"
  |"sep a xs          = xs"
```

Simplifikationsregeln

In **fun** definierte Regeln landen im Simplifier, können auch direkt mit *Funktionsname.simps* angesprochen werden

Beispiel: *fib.simps*:

```
fib 0 = 1
fib (Suc 0) = 1
fib (Suc (Suc ?n)) = fib ?n + fib (Suc ?n)
```

Beispiel: *sep.simps*:

```
sep ?a (?x # ?y # ?zs) = ?x # ?a # sep ?a (?y # ?zs)
sep ?a [] = []
sep ?a [?v] = [?v]
```

Beachte: Defaultregel (sep a xs = xs) generiert **zwei** Regeln, damit Pattern Matching vollständig

Simplifikationsregeln

In **fun** definierte Regeln landen im Simplifier, können auch direkt mit *Funktionsname.simps* angesprochen werden

Beispiel: *fib.simps*:

```
fib 0 = 1
fib (Suc 0) = 1
fib (Suc (Suc ?n)) = fib ?n + fib (Suc ?n)
```

Beispiel: *sep.simps*:

```
sep ?a (?x # ?y # ?zs) = ?x # ?a # sep ?a (?y # ?zs)
sep ?a [] = []
sep ?a [?v] = [?v]
```

Beachte: Defaultregel (sep a xs = xs) generiert **zwei** Regeln, damit Pattern Matching vollständig

Simplifikationsregeln

In **fun** definierte Regeln landen im Simplifier, können auch direkt mit *Funktionsname.simps* angesprochen werden

Beispiel: *fib.simps*:

```
fib 0 = 1
fib (Suc 0) = 1
fib (Suc (Suc ?n)) = fib ?n + fib (Suc ?n)
```

Beispiel: *sep.simps*:

```
sep ?a (?x # ?y # ?zs) = ?x # ?a # sep ?a (?y # ?zs)
sep ?a [] = []
sep ?a [?v] = [?v]
```

Beachte: Defaultregel (sep a xs = xs) generiert **zwei** Regeln, damit Pattern Matching vollständig

Induktionsregeln

analog definiert **fun** auch für jede Funktion eine Induktionsregel

Funktionsname.induct

Verwendung im Induktionsbeweis:

apply(*induct Funktionsparameter rule:Funktionsname.induct*)

Das nennt man *Regelinduktion*

Beispiel: *sep.induct*

```
⟦⋀a x y zs. ?P a (y # zs) ⟹ ?P a (x # y # zs); ⋀a. ?P a []; ⋀a v. ?P a [v]⟧ ⟹ ?P ?a0.0 ?a1.0
```

lemma *"map f (sep x ys) = sep (f x) (map f ys)"*

apply(*induct x ys rule:sep.induct*) generiert folgende 3 subgoals:

```
1. ⋀a x y zs. map f (sep a (y # zs)) = sep (f a) (map f (y # zs)) ⟹
     map f (sep a (x # y # zs)) = sep (f a) (map f (x # y # zs))
2. ⋀a. map f (sep a []) = sep (f a) (map f [])
3. ⋀a v. map f (sep a [v]) = sep (f a) (map f [v])
```

Induktionsregeln

analog definiert **fun** auch für jede Funktion eine Induktionsregel

Funktionsname.induct

Verwendung im Induktionsbeweis:

apply(*induct Funktionsparameter rule:Funktionsname.induct*)

Das nennt man *Regelinduktion*

Beispiel: *sep.induct*

$$\llbracket \bigwedge a\ x\ y\ zs.\ ?P\ a\ (y\ \#\ zs) \Longrightarrow ?P\ a\ (x\ \#\ y\ \#\ zs);\ \bigwedge a.\ ?P\ a\ [];\ \bigwedge a\ v.\ ?P\ a\ [v] \rrbracket \Longrightarrow ?P\ ?a0.0\ ?a1.0$$

lemma *"map f (sep x ys) = sep (f x) (map f ys)"*

apply(*induct x ys rule:sep.induct*) generiert folgende 3 subgoals:

1. $\bigwedge a\ x\ y\ zs.\ map\ f\ (sep\ a\ (y\ \#\ zs)) = sep\ (f\ a)\ (map\ f\ (y\ \#\ zs)) \Longrightarrow$
   $map\ f\ (sep\ a\ (x\ \#\ y\ \#\ zs)) = sep\ (f\ a)\ (map\ f\ (x\ \#\ y\ \#\ zs))$
2. $\bigwedge a.\ map\ f\ (sep\ a\ []) = sep\ (f\ a)\ (map\ f\ [])$
3. $\bigwedge a\ v.\ map\ f\ (sep\ a\ [v]) = sep\ (f\ a)\ (map\ f\ [v])$

Induktionsregeln

analog definiert **fun** auch für jede Funktion eine Induktionsregel

Funktionsname.induct

Verwendung im Induktionsbeweis:

apply *(induct Funktionsparameter rule:Funktionsname.induct)*

Das nennt man *Regelinduktion*

Beispiel: *sep.induct*

$$[\![\bigwedge a\ x\ y\ zs.\ ?P\ a\ (y\ \#\ zs) \Longrightarrow ?P\ a\ (x\ \#\ y\ \#\ zs);\ \bigwedge a.\ ?P\ a\ [];\ \bigwedge a\ v.\ ?P\ a\ [v]]\!] \Longrightarrow ?P\ ?a0.0\ ?a1.0$$

lemma *"map f (sep x ys) = sep (f x) (map f ys)"*

apply *(induct x ys rule:sep.induct)* generiert folgende 3 subgoals:

1. $\bigwedge a\ x\ y\ zs.\ map\ f\ (sep\ a\ (y\ \#\ zs)) = sep\ (f\ a)\ (map\ f\ (y\ \#\ zs)) \Longrightarrow$
   $map\ f\ (sep\ a\ (x\ \#\ y\ \#\ zs)) = sep\ (f\ a)\ (map\ f\ (x\ \#\ y\ \#\ zs))$
2. $\bigwedge a.\ map\ f\ (sep\ a\ []) = sep\ (f\ a)\ (map\ f\ [])$
3. $\bigwedge a\ v.\ map\ f\ (sep\ a\ [v]) = sep\ (f\ a)\ (map\ f\ [v])$

Ausblick

fun hat **primrec** weitestgehend ersetzt, nur in Randfällen noch Arbeit mit **primrec**

auch mit **fun** kann es Probleme geben, z.B. bei wechselseitiger Rekursion oder falls **fun** Termination nicht selbst sicherstellen kann
Lösung: **function**! Braucht jedoch selbstgeschriebenen Vollständigkeits- und Terminationsbeweis...

mehr dazu: **function**-Tutorial
`http://isabelle.in.tum.de/dist/Isabelle/doc/functions.pdf`

Ausblick

fun hat **primrec** weitestgehend ersetzt, nur in Randfällen noch Arbeit mit **primrec**

auch mit **fun** kann es Probleme geben, z.B. bei wechselseitiger Rekursion oder falls **fun** Termination nicht selbst sicherstellen kann
Lösung: **function**! Braucht jedoch selbstgeschriebenen Vollständigkeits- und Terminationsbeweis...

mehr dazu: **function**-Tutorial
`http://isabelle.in.tum.de/dist/Isabelle/doc/functions.pdf`