Karlsruher Institut für Technologie

Theorembeweiserpraktikum

Anwendungen in der Sprachtechnologie

LEHRSTUHL PROGRAMMIERPARADIGMEN

Karlsruher Institut für Technologie

Teil IX

Allgemeine Rekursion

Allgemeine Rekursion

Karlsruher Institut für Technologie

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

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**

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)"
```

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

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

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: Separatorzeichen zwischen je zwei Elemente einer Liste

```
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

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)
```

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)*

(auch bei wechselseitiger Rekursion)
Das nennt man *Regelinduktion*

Induktionsregeln

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

Funktionsname.induct

Verwendung im Induktionsbeweis:

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

(auch bei wechselseitiger Rekursion)

Das nennt man *Regelinduktion*

Beispiel: *sep.induct*

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

Induktionsregeln

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

Funktionsname.induct

Verwendung im Induktionsbeweis:

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

(auch bei wechselseitiger Rekursion)

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:

Induktionsregeln

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

Funktionsname.induct

Verwendung im Induktionsbeweis:

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

(auch bei wechselseitiger Rekursion)

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. $\bigwedge a\ v.\ map\ f\ (sep\ a\ [v]) = sep\ (f\ a)\ (map\ f\ [v])$

Ausblick

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

Aber: 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...

Ausblick

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

Aber: 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`

Teil X

Kombination von Regeln

Variables in Regeln spezifizieren mittels `of`

Manchmal nötig, um Variablen vor Regelanwendung festzulegen (z.B. wenn Isabelle passende Terme nicht inferieren kann), dann:

- eckige Klammer hinter Regelnamen
- Schlüsselwort `of`, danach einer oder mehrere Terme
- müssen natürlich zu Typ der Variable passen
- Reihenfolge wie erstes Auftreten in Regel
- _ für Variablen, die man nicht instantiieren möchte

Variablen in Regeln spezifizieren mittels `of`

Manchmal nötig, um Variablen vor Regelanwendung festzulegen (z.B. wenn Isabelle passende Terme nicht inferieren kann), dann:

- eckige Klammer hinter Regelnamen
- Schlüsselwort `of`, danach einer oder mehrere Terme
- müssen natürlich zu Typ der Variable passen
- Reihenfolge wie erstes Auftreten in Regel
- _ für Variablen, die man nicht instantiieren möchte

Beispiel:

iffE:	$[\![?P = ?Q; [\![?P \longrightarrow ?Q; ?Q \longrightarrow ?P]\!] \Longrightarrow ?R]\!] \Longrightarrow ?R$
iffE[of X]:	$[\![X = ?Q; [\![X \longrightarrow ?Q; ?Q \longrightarrow X]\!] \Longrightarrow ?R]\!] \Longrightarrow ?R$
iffE[of _ Y]:	$[\![?P = Y; [\![?P \longrightarrow Y; Y \longrightarrow ?P]\!] \Longrightarrow ?R]\!] \Longrightarrow ?R$
iffE[of X Y Z]:	$[\![X = Y; [\![X \longrightarrow Y; Y \longrightarrow X]\!] \Longrightarrow Z]\!] \Longrightarrow Z$

Variablen in Regeln spezifizieren mittels `where`

Syntax:

`Regel[where v=T]`

Wobei

- `v` die zu spezifizierende Variable in der Regel `Regel` ist
- `T` der einzusetzende Term ist

Beispiel:

`iffE:` $\llbracket ?P = ?Q;\ \llbracket ?P \longrightarrow ?Q;\ ?Q \longrightarrow ?P \rrbracket \Longrightarrow ?R \rrbracket \Longrightarrow ?R$

`iffE[where Q="X ∧ Y"]:` $\llbracket ?P = X \wedge Y;\ \llbracket ?P \longrightarrow X \wedge Y;\ X \wedge Y \longrightarrow ?P \rrbracket \Longrightarrow ?R \rrbracket \Longrightarrow ?R$

Prämissen in Regeln spezifizieren mittels `OF`

Analog zu `of`: Ganze Prämissen instantiieren

- ebenso eckige Klammer,
- Schlüsselwort `OF`, danach Regelname
- Konklusion der Regel und entspr. Prämisse müssen unifizieren
- entspr. Prämisse mit Prämissen der eingefügten Regel ersetzt
- bei mehreren `OF`s: erst linkeste Prämisse, dann die rechts davon usw.
- auch hier _ für Überspringen von Prämissen
- vor allem bei Induktionshypothesen in Isar einsetzbar

Prämissen in Regeln spezifizieren mittels *OF*

Analog zu *of*: Ganze Prämissen instantiieren

- ebenso eckige Klammer,
- Schlüsselwort *OF*, danach Regelname
- Konklusion der Regel und entspr. Prämisse müssen unifizieren
- entspr. Prämisse mit Prämissen der eingefügten Regel ersetzt
- bei mehreren *OF*s: erst linkeste Prämisse, dann die rechts davon usw.
- auch hier _ für Überspringen von Prämissen
- vor allem bei Induktionshypothesen in Isar einsetzbar

Beispiel:

conjI:	$[\![?P;\ ?Q]\!] \Longrightarrow ?P \wedge ?Q$
ccontr:	$(\neg\ ?P \Longrightarrow False) \Longrightarrow ?P$
conjI[OF ccontr]:	$[\![\neg\ ?P \Longrightarrow False;\ ?Q]\!] \Longrightarrow ?P \wedge ?Q$
conjI[OF ccontr,of X]:	$[\![\neg\ X \Longrightarrow False;\ ?Q]\!] \Longrightarrow X \wedge ?Q$

Regeln kombinieren mittels *THEN*

statt zwei Regel nacheinander auszuführen, Kombination dieser Regeln

- Konklusion der ersten passt auf eine Prämisse der zweiten Regel
- Variablen entsprechend zweiter Regel unifiziert
- erster Regelname gefolgt von eckiger Klammer
- dann Schlüsselwort *THEN* gefolgt von zweitem Regelnamen
- gut einsetzbar, falls Isabelle bei Substitution scheitert

Regeln kombinieren mittels *THEN*

statt zwei Regel nacheinander auszuführen, Kombination dieser Regeln

- Konklusion der ersten passt auf eine Prämisse der zweiten Regel
- Variablen entsprechend zweiter Regel unifiziert
- erster Regelname gefolgt von eckiger Klammer
- dann Schlüsselwort *THEN* gefolgt von zweitem Regelnamen
- gut einsetzbar, falls Isabelle bei Substitution scheitert

Beispiel:

iffI:	$\llbracket ?P \Longrightarrow ?Q;\ ?Q \Longrightarrow ?P \rrbracket \Longrightarrow ?P = ?Q$
sym:	$?s = ?t \Longrightarrow ?t = ?s$
mp:	$\llbracket ?P \longrightarrow ?Q;\ ?P \rrbracket \Longrightarrow ?Q$
iffI[THEN sym]:	$\llbracket ?s \Longrightarrow ?t;\ ?t \Longrightarrow ?s \rrbracket \Longrightarrow ?t = ?s$
iffI[THEN sym,of P Q]:	$\llbracket P \Longrightarrow Q;\ Q \Longrightarrow P \rrbracket \Longrightarrow Q = P$

iffI[THEN sym,OF mp,of P R Q]:

$$\llbracket P \Longrightarrow R \longrightarrow Q;\ P \Longrightarrow R;\ Q \Longrightarrow P \rrbracket \Longrightarrow Q = P$$