Karlsruher Institut für Technologie

Institut für Programmstrukturen und Datenorganisation
Lehrstuhl Programmierparadigmen
Am Fasanengarten 5
76131 Karlsruhe
http://pp.ipd.kit.edu/

Theorembeweiserpraktikum – SS 2016

http://pp.ipd.kit.edu/lehre/SS2016/tba

Lösung 3: Datentypen und Rekursion

Abgabe: 9. Mai 2016, 12:00 Uhr
Besprechung: 10. Mai 2016

Soweit nicht anders angegeben, sind jetzt alle Beweismethoden erlaubt.

1 Rätsel: Der reiche Großvater

Zeigen Sie: *Wenn jeder arme Mann einen reichen Vater hat,*
dann gibt es einen reichen Mann mit einem reichen Großvater.

```
theorem "(∀x. ¬ rich x ⟶ rich (father x)) ⟶ (∃y. rich y ∧ rich(father(father y)))"
proof
  assume "∀x. ¬ rich x ⟶ rich (father x)"
  have "∃x. rich x"
  proof(cases "∃x. rich x")
    case True
    then show ?thesis .
  next
    case False
    then have "∀x. ¬ rich x" by simp
    then obtain x where "¬ rich x" by simp
    with ⟨∀x. ¬ rich x ⟶ rich (father x)⟩ have "rich(father x)" by simp
    then show ?thesis by rule
  qed
  then obtain x where "rich x" by (rule exE)

  show "∃y. rich y ∧ rich(father(father y))"
  proof(cases "rich(father(father x))")
    case True
    with ⟨rich x⟩
    have "rich x ∧ rich (father (father x))" by simp
    then show ?thesis by (rule exI)
  next
    case False
    then have "¬ rich (father (father x))" .
    with ⟨∀x. ¬ rich x ⟶ rich (father x)⟩ have "rich(father(father (father x)))"
      by simp
    have "rich(father x)"
    proof(rule ccontr)
      from ⟨∀x. ¬ rich x ⟶ rich (father x)⟩
      have "¬ rich (father x) ⟶ rich (father (father x))" by rule
      moreover
```

```
    assume "¬ rich (father x)"
    ultimately
    have "rich (father (father x))" by (rule impE)
    with ‹¬ rich (father (father x))› show False by contradiction
  qed
  with ‹rich(father(father (father x)))›
  have "rich (father x) ∧ rich (father (father (father x)))" by simp
  then show ?thesis by (rule exI)
 qed
qed
```

- Gibt es überhaupt einen reichen Mann?
- Überlegen Sie sich den Beweis erst auf Papier.
- Schreiben Sie dann einen Isar-Beweis, der ihrer Papier-Beweisführung entspricht.
- Sie werden Fallunterscheidungen brauchen.

2 Cantors Theorem

Sie sollen nun Cantors Theorem beweisen; dieses sagt aus, dass es keine surjektive Funktion von einer Menge auf ihre Potenzmenge geben kann. Formalisiert:

```
theorem "∃S. S ∉ range (f :: 'a ⇒ 'a set)"
proof
 let ?S = "{x. x ∉ f x}"
 show "?S ∉ range f"
 proof
  assume "?S ∈ range f"
  then obtain y where fy:"?S = f y" by (rule rangeE)
  show False
  proof(cases "y ∈ ?S")
   case True
   then have "y ∉ f y" by simp
   then have "y ∉ ?S" by(simp add:fy)
   with True show ?thesis by simp
  next
   case False
   then have "y ∈ f y" by simp
   then have "y ∈ ?S" by(simp add:fy)
   with False show ?thesis by simp
  qed
 qed
qed
```

Dabei bezeichnet `range f` die Wertemenge einer Funktion.
Hinweise:

- Der Knackpunkt des Beweises ist das Finden der richtigen Menge `S`. Versuchen Sie es erstmal alleine, erinnern Sie sich (falls bekannt) an das sogenannte *Cantor'sche Diagonalverfahren*. Ansonsten versuchen Sie ihr Glück im Internet, der Name der Übung sollte Hinweis genug sein. ;-)

- Auch hier sollten Sie sich Ihren Beweis erst auf Papier überlegen und dann möglichst analog in Isar übertragen.
- Falls Sie eine Aussage wie `b ∈ range f` haben, lässt sich daraus unmittelbar ein `x` auswählen ("obtainen"), so dass `b = f x` gilt, da die Regel `rangeE: b ∈ range f ⟹ (⋀x. b = f x ⟹ P) ⟹ P` als Eliminationsregel in allen Taktiken des automatischen Schließens existiert.

3 Rekursive Datenstrukturen

In dieser Übung soll eine rekursive Datenstruktur für Binärbäume erstellt werden. Außerdem sollen Funktionen über Binärbäume definiert und Aussagen darüber gezeigt werden Denken Sie daran: *Recursion is proved by induction!*.
Zuerst definieren Sie den Datentypen für (nichtleere) Binärbäume. Sowohl Blätter (ohne Nachfolger) als auch innere Knoten (mit genau 2 Nachfolgern) speichern Information. Der Typ der Information soll beliebig sein, also arbeiten sie mit Typparameter `'a`.

```
datatype 'a tree = Leaf "'a" | Node "'a" "'a tree" "'a tree"
```

Definieren Sie jetzt die Funktionen `preOrder`, `postOrder` und `inOrder`, welche einen `'a tree` in der entsprechenden Ordnung durchlaufen:

```
fun preOrder :: "'a tree ⇒ 'a list"
  where "preOrder (Leaf l) = [l]"
  | "preOrder (Node n l r) = n#(preOrder l)@(preOrder r)"
fun postOrder :: "'a tree ⇒ 'a list"
  where "postOrder (Leaf l) = [l]"
  | "postOrder (Node n l r) = (postOrder l)@(postOrder r)@[n]"
fun inOrder :: "'a tree ⇒ 'a list"
  where "inOrder (Leaf l) = [l]"
  | "inOrder (Node n l r) = (inOrder l)@n#(inOrder r)"
```

Definieren Sie nun eine Funktion `mirror`, welche das Spiegelbild eines `'a tree` zurückgibt.

```
fun mirror :: "'a tree ⇒ 'a tree"
  where "mirror (Leaf l) = Leaf l"
  | "mirror (Node n l r) = Node n (mirror r) (mirror l)"
```

Seien `xOrder` und `yOrder`, beliebig ausgewählt aus `preOrder`, `postOrder` und `inOrder`. Formulieren und zeigen Sie alle gültigen Eigenschaften der Art:

```
lemma preOrder_mirror_rev_postOrder:
  "preOrder(mirror xt) = rev(postOrder xt)"
by(induction xt) auto

lemma postOrder_mirror_rev_preOrder:
  "postOrder(mirror xt) = rev(preOrder xt)"
by(induction xt) auto

lemma inOrder_mirror_rev_inOrder:
  "inOrder(mirror xt) = rev(inOrder xt)"
by(induction xt) auto
```

Definieren Sie die Funktionen `root`, `leftmost` und `rightmost`, welche die Wurzel, das äußerst links bzw. das äußerst rechts gelegene Element zurückgeben.

```
fun root :: "'a tree ⇒ 'a"
  where "root (Leaf l) = l"
  | "root (Node n l r) = n"

fun leftmost :: "'a tree ⇒ 'a"
  where "leftmost (Leaf l) = l"
  | "leftmost (Node n l r) = leftmost l"

fun rightmost :: "'a tree ⇒ 'a"
  where "rightmost (Leaf l) = l"
  | "rightmost (Node n l r) = rightmost r"
```

Beweisen Sie folgende Theoreme oder zeigen Sie ein Gegenbeispiel (dazu kann man u.a. **quickcheck** oder **nitpick** verwenden). Es kann nötig sein, erst bestimmte Hilfslemmas zu beweisen.

```
lemma [simp]: "inOrder xt ≠ []"
by(induction xt) auto
theorem "hd (preOrder xt) = last (postOrder xt)" by (cases xt) auto
theorem "hd (preOrder xt) = root xt" by (cases xt) auto
theorem "hd (inOrder xt) = root xt" quickcheck oops
theorem "last (postOrder xt) = root xt" by (cases xt) auto
theorem "hd (inOrder xt) = leftmost xt" by (induction xt) auto
theorem "last (inOrder xt) = rightmost xt" by (induction xt) auto
```

Und hier noch ein etwas komplizierteres Theorem.

```
lemma "(mirror xt = mirror xt') = (xt = xt')"
proof(induction xt arbitrary: xt')
  case (Leaf a xt')
  show ?case by (cases xt')auto
next
  case (Node a xt1 xt2 xt')
  then show ?case by (cases xt')auto
qed
```