Karlsruher Institut für Technologie

**Institut für Programmstrukturen und Datenorganisation**
**Lehrstuhl Programmierparadigmen**
Adenauerring 20a
76131 Karlsruhe
http://pp.info.uni-karlsruhe.de/

# Theorembeweiserpraktikum – SS 2011

http://pp.info.uni-karlsruhe.de/lehre/SS2011/tba

**Blatt 3: Rekursion**

Abgabe: 2. Mai 2011
Besprechung: 3. Mai 2011

## 1 Rekursive Datenstrukturen

In dieser Übung soll eine rekursive Datenstruktur für Binärbäume erstellt werden. Außerdem sollen Funktionen über Binärbäume definiert und Aussagen darüber gezeigt werden. Dazu dürfen Sie auch automatische Taktiken, z.B. `auto` verwenden. Und 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 = ...*

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

**consts**
  *preOrder :: "'a tree ⇒ 'a list"*
  *postOrder :: "'a tree ⇒ 'a list"*
  *inOrder :: "'a tree ⇒ 'a list"*

Als nächstes definieren Sie eine Funktion *mirror*, welche das Spiegelbild eines *'a tree* zurückgibt.

**consts** *mirror :: "'a tree ⇒ 'a tree"*

Seien *xOrder* und *yOrder* Verfahren zum Durchlaufen von Bäumen, beliebig ausgewählt aus *preOrder*, *postOrder* und *inOrder*. Formulieren und zeigen Sie alle gültigen Eigenschaften dieser Art:

**lemma** *"xOrder (mirror xt) = rev (yOrder xt)"*

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

**consts**
  *root :: "'a tree ⇒ 'a"*
  *leftmost :: "'a tree ⇒ 'a"*
  *rightmost :: "'a tree ⇒ 'a"*

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

```
theorem "hd (preOrder xt) = last (postOrder xt)"
theorem "hd (preOrder xt) = root xt"
theorem "hd (inOrder xt) = root xt"
theorem "last (postOrder xt) = root xt"
theorem "hd (inOrder xt) = leftmost xt"
theorem "last (inOrder xt) = rightmost xt"
```

Und hier noch ein etwas komplizierteres Theorem. Ein Tipp: Fallunterscheidung auf $\bigwedge$-quantifizierte Variablen mittels *case_tac* statt *case*.

```
lemma "(mirror xt = mirror xt') = (xt = xt')"
```

## 2 Wechselseitige Rekursion

In bestimmten Fällen muss man Datentypen definieren, die voneinander abhängig sind, d.h. der eine wird im anderen verwendet und anders herum. Um dann Aussagen über diese Datentypen machen zu können, braucht man wechselseitige Rekursion.
Wir wollen jetzt einen Datentyp definieren für arithmetische und boole'sche Aussagen. Der Typ der vorkommenden Variablen(namen) soll nicht spezifiziert werden, deshalb verwenden wir für sie den Typparameter *'a*. Da in arithmetischen Ausdrücken boole'sche verwendet werden können (Bsp. "if $m < n$ then $n - m$ else $m - n$") bzw. anders herum (Bsp. "$m - n < m + n$"), müssen wir sie wechselseitig rekursiv definieren:

```
datatype
  'a aexp =    — arithmetische Ausdr"ucke
    IF "'a bexp" "'a aexp" "'a aexp" — funktionales if-then-else, entspricht ?: z.B. in Java
  | Sum "'a aexp" "'a aexp"  — Addition
  | Diff "'a aexp" "'a aexp" — Subtraktion
  | Var 'a                   — Variablen (speichern natürliche Zahlen)
  | Const nat                — Konstanten (natürliche Zahlen)

  and — wechselseitige Rekursion
  'a bexp =    — boole'sche Ausdrücke
    Less "'a aexp" "'a aexp"
  | And "'a bexp" "'a bexp"
  | Neg "'a bexp"
```

Wir brauchen auch noch eine Umgebung, die die Werte der Variablen liefert, also eine Funktion von Typ der Variablen (*'a*) nach *nat*:

```
type_synonym 'a env = "'a ⇒ nat"
```

Definieren Sie jetzt Auswertungsfunktionen *evala* bzw. *evalb*, welche unter Verwendung einer Umgebung *'a env* das Resultat der Operation liefert. Da die Datentypen wechselseitig rekursiv sind, sind es auch die Funktionen, die auf ihnen operieren. Deshalb müssen *evala* und *evalb* innerhalb eines *primrec* definiert werden:

```
primrec  evala :: "'a aexp ⇒ 'a env ⇒ nat"
  and evalb :: "'a bexp ⇒ 'a env ⇒ bool"
  — erweitern Sie diese Definition
where
  "evala (Sum a1 a2) env  = evala a1 env + evala a2 env"

  | "evalb (Less a1 a2) env = (evala a1 env < evala a2 env)"
```

Analog definieren Sie jetzt zwei Funktionen, welche Variablensubstitution durchführen:

```
consts
  substa :: "('a ⇒ 'b aexp) ⇒ 'a aexp ⇒ 'b aexp"
  substb :: "('a ⇒ 'b aexp) ⇒ 'a bexp ⇒ 'b bexp"
```

Der erste Parameter ist die Substitution, eine Funktion, welche Variablen auf Ausdrücke abbildet. Sie wird auf alle Variablen des Ausdrucks angewandt, weshalb der Resultattyp ein Ausdruck mit Variablen vom Typ `'b` ist:

Beweisen Sie nun, dass die Substitutionsfunktion `Var` einen Ausdruck in sich selbst überführt. Wenn man versucht, diese Aussage einzeln für arithmetische bzw. boole'sche Ausdrücke zu zeigen, wird man feststellen, dass man jeweils die Aussage für die entsprechend anderen Ausdrücke im Induktionsschritt benötigt. Also müssen beide Theoreme gleichzeitig gezeigt werden.

```
lemma "substa Var (a::'a aexp) = a"
      "substb Var (b::'a bexp) = b"
```

Wir beweisen jetzt ein fundamentales Theorem über die Interaktion zwischen Auswertung und Substitution: wenn man eine Substitution `s` auf einen Ausdruck `a` anwendet und dann mittels einer Umgebung `env` auswertet, erhält man das gleiche Resultat wie wenn man `a` auswertet mittels einer Umgebung, welche jede Variable `x` auf den Wert `s(x)` unter `env` abbildet.

```
lemma "evala (substa s a) env = evala a (λx. evala (s x) env)"
      "evalb (substb s b) env = evalb b (λx. evala (s x) env)"
```

Abschließend sollen Sie eine Normalisierungsfunktion `norma` definieren. Diese soll `'a aexp`s so umbauen, dass in der Bedingung eines `IF` nur `Less` stehen darf; falls dort `And` oder `Neg` steht, muss der `IF`-Ausdruck umgebaut werden. Dafür brauchen sie eine weitere, zu `norma` wechselseitig rekursive Funktion; wie sieht diese aus? Beweisen Sie dann zwei Aussagen darüber:

1. `norma` verändert nicht den Wert, den `evala` liefert
2. `norma` ist wirklich normal, d.h. keine `And`s oder `Neg`s tauchen in den `IF`-Bedingungen auf (dafür brauchen Sie wiederum zwei wechselseitig rekursive Funktionen; welche?).

Beide Lemmas brauchen auch eine Aussage für die Funktion, welche die `IF`s umbaut.

```
consts norma :: "'a aexp ⇒ 'a aexp"
```

Hinweis: Lesen Sie sich Kap. 2.2 (Evaluation) im Isabelle-Tutorial durch. Die dort vorgestellten Befehle ermöglichen eine Auswertung einer Funktion, also das Testen, ob die Funktionen korrekt definiert sind.