Karlsruher Institut für Technologie

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

Theorembeweiserpraktikum – SS 2018

http://pp.ipd.kit.edu/lehre/SS2018/tba

Lösung 4: Allgemeine und wechselseitige Rekursion

Abgabe: 21. Mai 2018, *23:59 Uhr*
Besprechung: 22. Mai 2018

1 Listen zusammenfügen

In dieser Aufgabe soll eine Funktion *interleave* definiert werden, welche zwei Listen durch Verschachtelung zusammenfügt. Beispiel:
interleave [a1, a2, a3] [b1, b2, b3] = [a1, b1, a2, b2, a3, b3] und
interleave [a1] [b1, b2, b3] = [a1, b1, b2, b3].

```
fun interleave :: "'a list ⇒ 'a list ⇒ 'a list"
where
  "interleave xs []         = xs"
| "interleave [] ys         = ys"
| "interleave (x#xs) (y#ys) = x#y#(interleave xs ys)"
```

Unter welchen Voraussetzungen gilt die folgende Aussage? Beweisen Sie das entsprechende Theorem.
interleave (xs₁ @ xs₂) (ys₁ @ ys₂) = interleave xs₁ ys₁ @ interleave xs₂ ys₂
(Hinweis: Subskripte können mit 'Strg+E <Pfeiltaste unten>' eingegeben werden)

```
theorem interleave_append: "length xs₁ = length ys₁
  ⟹ interleave (xs₁@xs₂) (ys₁@ys₂) = (interleave xs₁ ys₁)@(interleave xs₂ ys₂)"
proof (induction xs₁ ys₁ arbitrary:xs₂ ys₂ rule:interleave.induct)
qed auto — Mit der Induktionsregel von interleave

theorem interleave_append2: "length xs₁ = length ys₁
  ⟹ interleave (xs₁@xs₂) (ys₁@ys₂) = (interleave xs₁ ys₁)@(interleave xs₂ ys₂)"
proof (induction xs₁ arbitrary:ys₁ xs₂ ys₂)
  — Mit Induktion über den Datentyp, etwas aufwändiger
  case (Cons x xs₁)
  then show ?case
    by (cases ys₁) auto
qed auto
```

2 Merge Sort

Wir arbeiten im Folgenden nur auf Listen über natürlichen Zahlen.
Definieren Sie ein Prädikat *sorted*, welches prüft, ob in einer Liste jedes Element kleiner oder gleich den folgenden ist; *le n xs* ist *True* g.d.w. *n* kleiner oder gleich allen Elementen in *xs*.

```
fun le :: "nat ⇒ nat list ⇒ bool"
```

```
  where "le n [] = True"
  | "le n (x#xs) = (n ≤ x ∧ le n xs)"
fun sorted :: "nat list ⇒ bool"
  where "sorted [] = True"
  | "sorted (x#xs) = (le x xs ∧ sorted xs)"
```

Implementieren Sie nun *Merge Sort*: Eine Liste wird durch Aufteilung in zwei Listen sortiert, welche einzeln sortiert und wieder zusammengefügt werden.
Definieren Sie mittels **fun** zwei Funktionen

```
fun merge :: "nat list ⇒ nat list ⇒ nat list"
where
  "merge xs []         = xs"
| "merge [] ys         = ys"
| "merge (x#xs) (y#ys) =
    (if (x ≤ y) then x#(merge xs (y#ys)) else y#(merge (x#xs) ys))"
fun msort :: "nat list ⇒ nat list"
where
  "msort []  = []"
| "msort [x] = [x]"
| "msort xs  = (let n = (length xs) div 2 in
                (merge (msort (take n xs)) (msort (drop n xs)) ))"
```

und zeigen Sie

```
lemma le_le:
  "⟦ le x xs;  x' ≤ x ⟧ ⟹ le x' xs"
by (induction xs) auto
lemma le_merge:
  "⟦ le x xs; le x ys ⟧ ⟹ le x (merge xs ys)"
by (induction xs ys rule:merge.induct) auto
lemma sorted_merge_sorted:
  "⟦ sorted xs; sorted ys ⟧ ⟹ sorted(merge xs ys)"
by (induction xs ys rule: merge.induct)(auto intro: le_merge simp add: le_le)

theorem "sorted (msort xs)"
by (induction xs rule:msort.induct)(auto intro: sorted_merge_sorted)
```

Sie werden dafür Hilfslemmas über `le` und `sorted` beweisen müssen.
Hinweise:

- Um eine Liste in zwei fast gleichlange Hälften zu zerteilen, können Sie die Funktionen `n div 2`, `take` und `drop` verwenden, wobei `take n xs` die ersten `n` Elemente von `xs` zurückgibt, `drop n xs` den Rest.
- Versuchen Sie erstmal, das Lemma alleine zu lösen und selbst herauszufinden, welche Hilfslemmas Sie dafür brauchen. Falls Sie so nicht weiterkommen, hier ein paar Überlegungen:
  - Was muss gelten, damit `merge sorted` ist?
  - Was muss gelten, wenn der zweite Parameter von `le merge` ist?
  - Wie verhält sich `le`, wenn der erste Parameter kleiner wird?
- Und nun noch eine Meta-Frage: Haben Sie wirklich bewiesen, dass `msort` eine korrekte Sortierfunktion ist?

3 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 Ausdrücke
    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 vom 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 **fun** definiert werden:

```
fun evala :: "'a aexp ⇒ 'a env ⇒ nat" and evalb :: "'a bexp ⇒ 'a env ⇒ bool"
  — erweitern Sie diese Definition
  where "evala (IF b a1 a2) env =
     (if evalb b env then evala a1 env else evala a2 env)"
  | "evala (Sum a1 a2) env      = evala a1 env + evala a2 env"
  | "evala (Diff a1 a2) env     = evala a1 env - evala a2 env"
  | "evala (Var v) env          = env v"
  | "evala (Const c) env        = c"

  | "evalb (Less a1 a2) env     = (evala a1 env < evala a2 env)"
  | "evalb (And b1 b2) env      = (evalb b1 env ∧ evalb b2 env)"
  | "evalb (Neg b) env          = (¬ evalb b env)"
```

Analog definieren Sie jetzt zwei Funktionen, welche Variablensubstitution durchführen:

```
fun substa :: "('a ⇒ 'b aexp) ⇒ 'a aexp ⇒ 'b aexp"
  and substb :: "('a ⇒ 'b aexp) ⇒ 'a bexp ⇒ 'b bexp"
  where "substa s (IF b a1 a2) = IF (substb s b) (substa s a1) (substa s a2)"
  | "substa s (Sum a1 a2)      = Sum (substa s a1) (substa s a2)"
  | "substa s (Diff a1 a2)     = Diff (substa s a1) (substa s a2)"
```

```
  | "substa s (Var v)          = s v"
  | "substa s (Const c)        = Const c"

  | "substb s (Less a1 a2)     = Less (substa s a1) (substa s a2)"
  | "substb s (And b1 b2)      = And (substb s b1) (substb s b2)"
  | "substb s (Neg b)          = Neg (substb s b)"
```

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"
  and "substb Var (b::'a bexp) = b"
by(induction a and b) simp_all
```

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)"
  and "evalb (substb s b) env = evalb b (λx. evala (s x) env)"
by(induction a and b) simp_all
```

Abschließend sollen Sie eine Normalisierungsfunktion `norma` mit Typ `'a aexp ⇒ 'a aexp` 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:

- `norma` verändert nicht den Wert, den `evala` liefert
- `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.

```
fun norma :: "'a aexp ⇒ 'a aexp"
  and normif :: "'a bexp ⇒ 'a aexp ⇒ 'a aexp ⇒ 'a aexp"
  where "norma (IF b a1 a2)  = normif b (norma a1) (norma a2)"
  | "norma (Sum a1 a2)       = Sum (norma a1) (norma a2)"
  | "norma (Diff a1 a2)      = Diff (norma a1) (norma a2)"
  | "norma (Var v)           = Var v"
  | "norma (Const c)         = Const c"

  | "normif (Less a1 a2) t e = IF(Less (norma a1) (norma a2)) t e"
  | "normif (And b1 b2) t e  = normif b1 (normif b2 t e) e"
  | "normif (Neg b) t e      = normif b e t"

lemma "evala (norma a) env = evala a env"
  and "evala (normif b t e) env = evala (IF b t e) env"
```

```
by (induction a and b arbitrary: and t e) auto

fun normala :: "'a aexp ⇒ bool" and normalb :: "'a bexp ⇒ bool"
  where "normala (IF b a1 a2) = (normalb b ∧ normala a1 ∧ normala a2)"
  | "normala (Sum a1 a2)      = (normala a1 ∧ normala a2)"
  | "normala (Diff a1 a2)     = (normala a1 ∧ normala a2)"
  | "normala (Var v)          = True"
  | "normala (Const c)        = True"

  | "normalb (Less a1 a2)     = (normala a1 ∧ normala a2)"
  | "normalb (And b1 b2)      = False"
  | "normalb (Neg b)          = False"


lemma "normala (norma (a::'a aexp))"
  and "normala (normif (b::'a bexp) t e) = (normala t ∧ normala e)"
by (induction a and b arbitrary: and t e) auto
```