

Praktikum Spezifikation und Verifikation

Natürliches Schließen

In dieser Aufgabe geht es um den Kalkül des natürlichen Schließens, mit dessen Hilfe einige Lemmas der Aussagen-Logik bewiesen werden sollen.

Für die Beweise gelten die folgenden Spielregeln:

- Es dürfen nur diese Lemmas verwendet werden:

notI: $(A \implies \text{False}) \implies \neg A$,

notE: $[\neg A; A] \implies B$,

conjI: $[A; B] \implies A \wedge B$,

conjE: $[A \wedge B; [A; B] \implies C] \implies C$,

disjI1: $A \implies A \vee B$,

disjI2: $A \implies B \vee A$,

disjE: $[A \vee B; A \implies C; B \implies C] \implies C$,

impI: $(A \implies B) \implies A \longrightarrow B$,

impE: $[A \longrightarrow B; A; B \implies C] \implies C$,

mp: $[A \longrightarrow B; A] \implies B$

iffI: $[A \implies B; B \implies A] \implies A = B$,

iffE: $[A = B; [A \longrightarrow B; B \longrightarrow A] \implies C] \implies C$

classical: $(\neg A \implies A) \implies A$

- Es dürfen nur die Methoden *rule*, *erule* und *assumption* verwendet werden.

```
lemma I: "A --> A"
```

```
apply (rule impI)
```

```
apply assumption
```

```
done
```

```
lemma "A & B --> B & A"
```

```
apply (rule impI)
```

```
apply (erule conjE)
```

```
apply (rule conjI)
```

```
    apply assumption
  apply assumption
done
```

```
lemma "(A & B) --> (A | B)"
  apply (rule impI)
  apply (erule conjE)
  apply (erule disjI1)
done
```

```
lemma K: "A --> B --> A"
  apply (rule impI)
  apply (rule impI)
  apply assumption
done
```

```
lemma "(A | A) = (A & A)"
  apply (rule iffI)
  apply (erule disjE)
  apply (rule conjI)
  apply assumption
  apply assumption
  apply (rule conjI)
  apply assumption
  apply assumption
  apply (erule conjE)
  apply (erule disjI1)
done
```

```
lemma S: "(A --> B --> C) --> (A --> B) --> A --> C"
  apply (rule impI)
  apply (rule impI)
  apply (rule impI)
  apply (rule mp)
  apply (rule mp)
  apply assumption
  apply assumption
  apply (rule mp)
  apply assumption
  apply assumption
done
```

```
lemma "(A --> B) --> (B --> C) --> A --> C"
  apply (rule impI)
```

```
apply (rule impI)
apply (rule impI)
apply (erule mp)
apply (erule mp)
apply assumption
done
```

```
lemma " $\sim \sim A \rightarrow A$ "
apply (rule impI)
apply (rule classical)
apply (erule notE)
apply assumption
done
```

```
lemma " $(\sim A \rightarrow B) \rightarrow (\sim B \rightarrow A)$ "
apply (rule impI)
apply (rule impI)
apply (rule classical)
apply (erule notE)
apply (erule mp)
apply assumption
done
```

```
lemma " $((A \rightarrow B) \rightarrow A) \rightarrow A$ "
apply (rule impI)
apply (rule classical)
apply (erule impE)
  apply (rule impI)
  apply (erule notE)
  apply assumption
apply assumption
done
```

```
lemma " $A \mid \sim A$ "
apply (rule classical)
apply (rule notE)
  apply (rule notI)
  apply (erule notE)
  apply (rule disjI2)
  apply assumption
apply (rule notI)
apply (erule notE)
apply (rule disjI1)
apply assumption
```

done

Quotient und Rest

Geben Sie ein annotiertes imperatives Program zur Berechnung des Quotients und Rests zweier natürlichen Zahlen x und y . Beweisen Sie dann folgendes Hoare-Tripel:

```
record quo_vars =
  quo  :: nat
  rest :: nat

theorem
  "⊢ .{True}.
  `quo:= 0;
  `rest:= x;
  WHILE y ≤ `rest
  INV .{`quo * y + `rest = x}.
  DO `rest:= `rest - y;
    `quo:= `quo + 1
  OD
  .{`quo * y + `rest = x ∧ `rest < y }."
  apply hoare
  apply auto
done
```

Sortieren

Implementieren Sie die Funktion *insort* des Sortieralgorithmus aus Aufgabenblatt 2 imperativ. Eingabe ist eine Liste ls , Ausgabe eine neue Liste ss , in die das Element n geordnet eingefügt wurde. Zeigen Sie folgendes Hoare-Tripel:

```
consts
  gr :: "nat ⇒ nat list ⇒ bool"
primrec
  "gr j [] = True"
  "gr j (x#xs) = (x < j ∧ gr j xs)"

lemma [simp]:
  "gr j (a@b) = (gr j a ∧ gr j b)"
  by (induct_tac a, auto)

lemma [simp]:
  "gr j xs → insort j xs = xs @ [j]"
```

```

apply (induct_tac xs)
  apply simp
  apply simp
done

lemma [simp]:
  "gr j xs  $\longrightarrow$   $\neg$  hd ys < j  $\longrightarrow$  insert j (xs @ ys) = xs @ j # ys"
  apply (induct_tac xs)
    apply simp
    apply (case_tac ys)
      apply simp
      apply simp
    apply simp
  done

record insert_vars =
  ls :: "nat list"
  ss :: "nat list"

theorem
  " $\vdash$   $\{list = \text{'ls}\}$ .
   $\text{'ss} := []$ ;
  WHILE  $\text{'ls} \neq [] \wedge \text{hd } \text{'ls} < n$ 
  INV  $\{list = \text{'ss} @ \text{'ls} \wedge \text{gr } n \text{'ss}\}$ . DO
     $\text{'ss} := \text{'ss} @ [\text{hd } \text{'ls}]$ ;
     $\text{'ls} := \text{tl } \text{'ls}$ 
  OD;
   $\text{'ss} := \text{'ss} @ [n]$ ;
  WHILE  $\text{'ls} \neq []$ 
  INV  $\{\text{'ss} @ \text{'ls} = \text{insert } n \text{ list}\}$ . DO
     $\text{'ss} := \text{'ss} @ [\text{hd } \text{'ls}]$ ;
     $\text{'ls} := \text{tl } \text{'ls}$ 
  OD
   $\{ \text{'ss} = \text{insert } n \text{ list} \}$ ."
  apply hoare
    apply simp
    apply simp
    apply clarsimp
    apply (erule disjE)
    apply simp
    apply simp
  apply clarsimp
done

```

▷ **Hinweise:**

- Die Eingabeliste `xs` darf verändert werden.
- Um eine verkettete Liste zu simulieren, dürfen Sie im Programm selbst nur die Funktionen `hd`, `tl`, und `@ [x]` (append mit einer einelementigen Liste) benutzen. In den Invarianten steht ihnen die volle Mächtigkeit von HOL mit allen Funktionen und Ausdrücken zur Verfügung.
- Sie werden für die Invariante evtl. eine neue Funktion definieren müssen.

Freiwillige Zusatzaufgabe:

Implementieren Sie den gesamten Sortieralgorithmus (die Funktion `sort` von Aufgabenblatt 2) imperativ, und beweisen Sie, dass er das gleiche Ergebnis liefert wie `sort`. Sie können alternativ auch zeigen, dass für das Ergebnis `sorted` gilt, und `count` nicht verändert wird.

lemma [simp]:

```
"length (insort x xs) = Suc (length xs)"  
by (induct_tac xs, auto)
```

lemma drop_take [simp]:

```
"∀n. drop n xs = [] → take n xs = xs"  
apply (induct_tac xs)  
apply auto  
apply (case_tac n)  
apply simp  
apply simp  
done
```

lemma drop_take2 [simp]:

```
"∀n. (∃ys. drop n xs = y # ys) → take (Suc n) xs = take n xs @ [y]"  
apply (induct_tac xs)  
apply auto  
apply (case_tac n)  
apply auto  
done
```

lemma drop_suc [simp]:

```
"∀n. (∃y. drop n xs = y#ys) → drop (Suc n) xs = ys"  
apply (induct_tac xs)  
apply auto  
apply (case_tac n)  
apply auto  
done
```

lemma [simp]:

```

"insert x (insert y xs) = insert y (insert x xs)"
apply (induct_tac xs)
apply simp
apply simp
done

```

```

lemma [simp]:
  "sort (xs @ [y]) = insert y (sort xs)"
  apply (induct_tac xs)
  apply simp
  apply simp
  done

```

```

record sort_vars = insert_vars +
  xs :: "nat list"
  ts :: "nat list"

```

theorem

```

"⊢ .{`xs = input}.
`ss := [];
WHILE `xs ≠ []
INV .{`ss = sort (take (length `ss) input) ∧ `xs = drop (length `ss) input}.
DO
  `ls := `ss;
  `ts := `ls;
  `ss := [];
  WHILE `ls ≠ [] ∧ hd `ls < hd `xs
  INV .{`ts = `ss @ `ls ∧ gr (hd `xs) `ss ∧
      sort (take (length `ts) input) = `ts ∧
      `xs = drop (length `ts) input ∧ `xs ≠ []}.
  DO
    `ss := `ss @ [hd `ls];
    `ls := tl `ls
  OD;
  `ss := `ss @ [hd `xs];
  WHILE `ls ≠ []
  INV .{`ss @ `ls = insert (hd `xs) `ts ∧
      sort (take (length `ts) input) = `ts ∧
      `xs = drop (length `ts) input ∧ `xs ≠ []}.
  DO
    `ss := `ss @ [hd `ls];
    `ls := tl `ls
  OD;
  `xs := tl `xs
OD
.{`ss = sort input}."
apply hoare

```

```
    apply simp
    apply clarsimp
    apply clarsimp
    apply clarsimp
    apply (erule disjE)
    apply simp
    apply simp
    apply clarsimp
    apply clarsimp
    apply (clarsimp simp add: neq_Nil_conv)
    apply clarsimp
    apply (rotate_tac -2)
    apply simp
done

end
```