

## Praktikum Spezifikation und Verifikation

theory 16 = Main:

datatype slot = A | B | C

types config = "slot  $\Rightarrow$  nat list"

types move = "slot  $\times$  slot"

consts

```
other :: "slot  $\Rightarrow$  slot  $\Rightarrow$  slot"  
move  :: "nat  $\Rightarrow$  slot  $\Rightarrow$  slot  $\Rightarrow$  move list"  
step  :: "config  $\Rightarrow$  move  $\Rightarrow$  config option"  
exec  :: "config  $\Rightarrow$  move list  $\Rightarrow$  config option"  
hanoi :: "config  $\Rightarrow$  bool"  
tower :: "nat  $\Rightarrow$  nat list"  
ordered :: "nat list  $\Rightarrow$  bool"  
lt     :: "nat  $\Rightarrow$  nat list  $\Rightarrow$  bool"
```

primrec

```
"other A x = (if x = B then C else B)"  
"other B x = (if x = A then C else A)"  
"other C x = (if x = A then B else A)"
```

primrec

```
"move 0 src dst = []"  
"move (Suc n) src dst =  
  move n src (other src dst) @ [(src,dst)] @ move n (other src dst) dst"
```

primrec

```
"exec c [] = Some c"  
"exec c (x#xs) = (let cfg' = step c x in if cfg' = None then None else exec  
(the cfg') xs)"
```

primrec

```
"ordered [] = True"  
"ordered (x#xs) = (lt x xs  $\wedge$  ordered xs)"
```

primrec

```

"lt n [] = True"
"lt n (x#xs) = (n < x ∧ lt n xs)"

primrec
  "tower 0 = []"
  "tower (Suc n) = tower n @ [Suc n]"

defs
  step_def:
  "step c x ≡ let (src,dst) = x in
    if c src = [] then None
    else let src' = tl (c src);
          m = hd (c src);
          dst' = m # (c dst);
          c' = (c (src:= src')) (dst:= dst')
    in if hanoi c' then Some c' else None"

  hanoi_def:
  "hanoi cfg ≡ ∀s. ordered (cfg s)"

lemma [simp]:
  "other x y ≠ x ∧ other x y ≠ y"
  by (cases x, auto)

lemma "move 1 A C = [(A,C)]"
  by simp

lemma "move 2 A C = [(A, B), (A, C), (B, C)]"
  by (simp add: nat_number)

lemma "move 3 A C = [(A, C), (A, B), (C, B), (A, C), (B, A), (B, C), (A, C)]"
  by (simp add: nat_number)

lemma [simp]:
  "∀cfg. exec cfg (a@b) = (let cfg' = exec cfg a in if cfg' = None then None
  else exec (the cfg') b)"
  by (induct a, auto simp add: Let_def)

lemma neq_Nil_snoC:
  "∀n. length xs = Suc n ⟶ (∃x' xs'. xs = xs' @ [x'])"
  apply (induct xs)
  apply simp
  apply clarsimp
  apply (case_tac list)
  apply simp
  apply clarsimp
  done

```

```

lemma otherF [simp]: "x = other x y  $\implies$  False"
  apply (cases x, auto split: split_if_asm)
  done

lemma [simp]: "x  $\neq$  y  $\implies$  other x (other x y) = y"
  apply (cases x)
  apply (cases y, auto)+
  done

lemma [simp]: "x  $\neq$  y  $\implies$  other (other x y) y = x"
  apply (cases x)
  apply (cases y, auto)+
  done

consts
  gt :: "nat  $\Rightarrow$  nat list  $\Rightarrow$  bool"
primrec
  "gt n [] = True"
  "gt n (x#xs) = (x < n  $\wedge$  gt n xs)"

lemma [simp]:
  "lt n (a@b) = (lt n a  $\wedge$  lt n b)"
  apply (induct a)
  apply auto
  done

lemma [simp]:
  "gt n (a@b) = (gt n a  $\wedge$  gt n b)"
  apply (induct a)
  apply auto
  done

lemma lt_mono [rule_format, simp]:
  "a < b  $\longrightarrow$  lt b xs  $\longrightarrow$  lt a xs"
  apply (induct xs)
  apply auto
  done

lemma [simp]:
  "ordered (a@n#b) = (ordered a  $\wedge$  lt n b  $\wedge$  gt n a  $\wedge$  ordered b)"
  apply (induct a)
  apply simp
  apply auto
  done

lemma gt_def:

```

```
"gt n xs = ( $\forall x \in \text{set } xs. x < n$ )"
by (induct xs, auto)
```

lemma [simp]:

```
"xs  $\neq$  []  $\longrightarrow$  last xs  $\in$  set xs"
apply (induct xs, auto)
done
```

lemma [simp]:

```
"[[cfg src = ts' @ t' # xs; hanoi cfg; ts'  $\neq$  []]]  $\implies$  last ts' < t'"
apply (unfold hanoi_def)
apply (erule_tac x = src in allE)
apply (clarsimp simp add: gt_def)
done
```

lemma neq\_other:

```
"[ s  $\neq$  src; s  $\neq$  dst; src $\neq$ dst ]  $\implies$  s = other src dst"
apply (cases src, auto)
apply (cases s, auto)
apply (cases s, auto)
apply (cases dst, auto)
apply (cases s, auto)
apply (cases s, auto)
apply (cases s, auto)
apply (cases dst, auto)
apply (cases s, auto)
apply (cases s, auto)
apply (cases dst, auto)
done
```

lemma ordered\_appendI [rule\_format]:

```
"ordered a  $\longrightarrow$  lt t b  $\longrightarrow$  gt t a  $\longrightarrow$  ordered b  $\longrightarrow$  ordered (a@b)"
by (induct a, auto)
```

lemma [simp]:

```
" $\forall$ cfg. exec cfg xs = Some cfg'  $\longrightarrow$  hanoi cfg  $\longrightarrow$  hanoi cfg'"
apply (induct xs)
apply simp
apply (auto simp add: step_def Let_def split: split_if_asm)
done
```

lemma hanoi\_lemma:

```
" $\forall$ cfg src dst t xs ys zs.
  cfg src = t @ xs  $\longrightarrow$  cfg dst = ys  $\longrightarrow$  cfg (other src dst) = zs  $\longrightarrow$ 
  length t = n  $\longrightarrow$ 
  hanoi cfg  $\longrightarrow$ 
  lt (last t) ys  $\longrightarrow$  lt (last t) zs  $\longrightarrow$ 
```

```

src ≠ dst →
(∃cfg'. exec cfg (move n src dst) = Some cfg' ∧ cfg' src = xs ∧ cfg' dst = t
@ ys ∧ cfg' (other src dst) = zs)"
apply (induct n)
  apply simp
apply clarsimp
apply (case_tac "n=0")
  apply (simp add: Let_def)
  apply (case_tac t)
    apply simp
    apply simp
  apply (rule conjI)
    apply (clarsimp simp add: step_def Let_def hanoi_def)
    apply (erule_tac x = src in allE)
    apply simp
  apply (clarsimp simp add: step_def Let_def)
apply clarsimp
apply (subgoal_tac "∃t' ts'. t = ts' @ [t']")
  prefer 2
  apply (simp add: neq_Nil_snoC)
  apply clarsimp
  apply (frule spec, erule allE, erule_tac x = "other src dst" in allE, erule
allE, erule allE, erule impE, assumption)
  apply (erule impE, rule refl)
  apply (erule impE, assumption)
  apply simp
  apply (subgoal_tac "last ts' < t'")
    apply (erule impE)
      apply (erule lt_mono, assumption)
      apply (erule impE)
        apply (erule lt_mono, assumption)
        apply (erule impE)
          apply rule
          apply (erule otherF)
    prefer 2
    apply simp
  apply clarsimp
  apply (clarsimp simp add: Let_def)
  apply (rule conjI)
  apply (clarsimp simp add: step_def Let_def hanoi_def)
  apply (rule conjI)
    apply (erule_tac x=src in allE)
    apply clarsimp
  apply clarsimp
  apply (drule neq_other, assumption, assumption)
  apply simp
  apply (frule_tac x="other src dst" in spec)

```

```

apply (drule_tac x="src" in spec)
apply clarsimp
apply (rule ordered_appendI, assumption+)
apply (clarsimp simp add: step_def Let_def)
apply (erule_tac x="cfg'(src := xs, dst := t' # cfg dst)" in allE)
apply (erule_tac x="other src dst" in allE)
apply (erule_tac x="dst" in allE)
apply (erule allE)+
apply (erule impE)
  apply simp
apply (erule impE, rule refl)
apply (erule impE)
  apply simp
apply (erule impE)
  apply simp
  apply (rule lt_mono)
  apply (subgoal_tac "last ts' < t'")
    prefer 2
    apply simp
  apply assumption+
apply (erule impE)
  apply (subgoal_tac "last ts' < t'")
    prefer 2
    apply simp
  apply (unfold hanoi_def)
  apply (erule_tac x = src in allE)
  apply (erule lt_mono)
  apply simp
apply clarsimp
done

```

```

lemma [simp]:
  "length (tower n) = n"
  by (induct n, auto)

```

```

lemma
  "lt 0 (tower n)"
  by (induct n, auto)

```

```

lemma gt_mono [rule_format, simp]:
  "x < y  $\longrightarrow$  gt x xs  $\longrightarrow$  gt y xs"
  apply (induct xs)
  apply auto
  done

```

```

lemma [simp]:
  "gt (Suc n) (tower n)"

```

```

apply (induct n)
apply auto
apply (rule gt_mono)
defer
apply assumption
apply simp
done

```

```

lemma [simp]:
  "ordered (tower n)"
  apply (induct n)
  apply auto
  done

```

```

lemma hanoi_start:
  "[[ cfg A = tower n; cfg B = []; cfg C = [] ] ] ==>
  hanoi cfg"
  apply (unfold hanoi_def)
  apply (rule allI)
  apply (case_tac s)
  apply auto
  done

```

```

theorem hanoi:
  "[[cfg A = tower n;
  cfg B = [];
  cfg C = []]] ==>
  ∃ cfg'. exec cfg (move n A C) = Some cfg' ∧
  cfg' A = [] ∧
  cfg' B = [] ∧
  cfg' C = tower n"
  apply (frule hanoi_start, assumption+)
  apply (insert hanoi_lemma [of n])
  apply (erule_tac x=cfg in allE)
  apply (erule_tac x=A in allE)
  apply (erule_tac x=C in allE)
  apply (erule_tac x="tower n" in allE)
  apply (erule allE)+
  apply (erule impE)
  apply simp
  apply (erule impE, assumption)+
  apply (erule impE, simp)
  apply clarsimp
  done

```

```
lemma " $\forall x y. \text{length } (\text{move } n \ x \ y) = 2^n - 1$ "  
  apply (induct n)  
    apply simp  
    apply auto  
    apply arith  
  done  
  
end
```