

A Hoare Logic for Diverging Programs

Johannes Åman Pojhol, Magnus O. Myreen, Miki Tanaka

March 17, 2025

Abstract

This submission contains: (1) a formalisation of a small While language with support for output; (2) a standard total-correctness Hoare logic that has been proved sound and complete; and (3) a new Hoare logic for proofs about programs that diverge: this new logic has also been proved sound and complete.

Background

The theories in this submission are a port of a similar formalisation in HOL4, which, in turn, is a simplified version of a program logic for nonterminating CakeML programs [1] (which was not complete).

The HOL4 version of these theories was developed by Myreen with some lemmas proved by Åman Pojhol. The Isabelle/HOL theories were developed by Tanaka with some input from Åman Pojhol.

Contents

1	Miscellaneous lemmas	2
2	The definition of the While language	5
3	A standard total correctness Hoare logic	8
4	Lemmas about the While language	9
5	Soundness of the standard Hoare logic	26
6	A Hoare logic for diverging programs	29
7	Completeness of the standard Hoare logic	30
8	Completeness of Hoare logic for diverging programs	32
9	Soundness of Hoare logic for diverging programs	41

1 Miscellaneous lemmas

```

theory MiscLemmas imports HOL-Library.Sublist begin

inductive star :: ('a ⇒ 'a ⇒ bool) ⇒ 'a ⇒ 'a ⇒ bool for r where
  refl[simp,intro]: star r x x
| step: r x y ==> star r y z ==> star r x z

inductive star-n :: ('a ⇒ 'a ⇒ bool) ⇒ nat ⇒ 'a ⇒ 'a ⇒ bool for r where
  refl-n: star-n r 0 x x
| step-n: r x y ==> star-n r n y z ==> star-n r (Suc n) x z

declare star-n.intros[simp, intro]

lemma star-n-decompose:
  star-n r n x y ==> star-n r n' x z ==> n' < n
  ==> (∀x y z. r x y ==> r x z ==> y = z)
  ==> (∀n x y z. star-n r n x y ==> star-n r n x z ==> y = z)
  ==> star-n r (n - n') z y
  apply (induct arbitrary: n' z rule: star-n.induct, simp)
  apply clarsimp
  apply (rename-tac n z n' za)
  apply (subgoal-tac Suc n - n' = Suc (n - n'))
  prefer 2
  apply simp
  apply simp
  apply (case-tac n' < n, simp)
  apply (case-tac n')
  apply clarsimp
  apply (erule star-n.cases[of - 0]; simp)
  apply (rename-tac z' m)
  apply simp
  apply (drule-tac x=m and y=z' in meta-spec2)
  apply (subgoal-tac Suc (n - Suc m) = n - m)
  prefer 2
  apply arith
  apply simp
  apply (erule meta-mp)
  apply (erule-tac ?a1.0=Suc m in star-n.cases; simp)
  apply clarsimp
  apply (rename-tac y n z x y' n' z')
  apply (drule-tac x=x and y=y in meta-spec2)
  apply (drule-tac x=y' in meta-spec)
  apply simp
  apply (subgoal-tac n' = n)
  prefer 2
  apply arith

```

```

apply simp
apply (case-tac n; simp)
apply clarsimp
apply (erule star-n.cases; simp)
apply (erule star-n.cases; simp)
apply (rename-tac za m)
apply (drule-tac x=n - (Suc 0) and y=za in meta-spec2)
apply simp
apply (erule meta-mp)
apply (drule-tac x=m and y=y in meta-spec2)
apply (rotate-tac -1)
apply (drule-tac x=z and y=za in meta-spec2)
apply (erule star-n.cases; simp)
apply (erule star-n.cases; simp)
by fastforce

lemma star-n-add:
star-n r n x z  $\longleftrightarrow$  ( $\exists n1\ n2\ y.$  star-n r n1 x y  $\wedge$  star-n r n2 y z  $\wedge$  n = n1 + n2)
proof (rule iffI)
assume star-n r n x z
from this show  $\exists n1\ n2\ y.$  star-n r n1 x y  $\wedge$  star-n r n2 y z  $\wedge$  n = n1 + n2
by (induct rule: star-n.induct; fastforce)
next
assume H0:  $\exists n1\ n2\ y.$  star-n r n1 x y  $\wedge$  star-n r n2 y z  $\wedge$  n = n1 + n2
have H: star-n r n1 x y  $\Longrightarrow$ 
star-n r n2 y z  $\Longrightarrow$  star-n r (n1 + n2) x z for y n1 n2
by (induct rule: star-n.induct; simp)
show star-n r n x z using H0 H by blast
qed

lemma star-star-n:
star r x y  $\Longrightarrow$   $\exists n.$  star-n r n x y
by (induct rule: star.induct; fastforce)

lemma star-n-star:
star-n r n x y  $\Longrightarrow$  star r x y
by (induct rule: star-n.induct; fastforce elim!: step)

lemma star-eq-star-n:
star r x y = ( $\exists n.$  star-n r n x y)
by (meson star-n-star star-star-n)

lemma star-trans:
star r x y  $\Longrightarrow$  star r y z  $\Longrightarrow$  star r x z
by (meson star-eq-star-n star-n-add)

lemma step-rev:
star r x y  $\Longrightarrow$  r y z  $\Longrightarrow$  star r x z
by (meson star.simps star-trans)

```

```

lemma star-n-trans:
  star-n r n x y ==> star-n r n' y z ==> star-n r (n + n') x z
  using star-n-add by force

lemma step-n-rev:
  star-n r n x y ==> r y z ==> star-n r (Suc n) x z
  by (induct rule: star-n.induct; clarsimp)

lemma star-n-lastE:
  star-n r (Suc n) x z ==>
  ( $\bigwedge n' x' y' z'. n = n' \Rightarrow x = x' \Rightarrow z = z'$ 
    $\Rightarrow star-n r n' x' y' \Rightarrow r y' z' \Rightarrow P) \Rightarrow P$ 
proof (induct n arbitrary: x z)
  case 0
  then show ?case
    by cases (erule star-n.cases; force)
next
  case (Suc n)
  then show ?case
    by cases (erule star-n.cases; force) +
qed

lemma
  star-n-conjunct1: star-n ( $\lambda x y. P x y \wedge Q x y$ ) n s t ==> star-n P n s t and
  star-n-conjunct2: star-n ( $\lambda x y. P x y \wedge Q x y$ ) n s t ==> star-n Q n s t and
  star-n-commute: star-n ( $\lambda x y. P x y \wedge Q x y$ ) n s t ==> star-n ( $\lambda x y. Q x y \wedge P$ 
  x y) n s t
  by (induct rule: star-n.induct; clarsimp) +
lemma forall-swap4:
  ( $\forall x y z w. P x y z w) \longleftrightarrow (\forall z w y x. P x y z w)$  by auto

lemma prefix-drop-append:
  prefix xs ys ==> xs @ drop (length xs) ys = ys
  by (metis append-eq-conv-conj prefix-def)

lemma min-prefix:
   $\forall i. prefix(f i)(f(Suc i)) \Rightarrow (\forall i. prefix(f 0)(f i))$ 
  apply clarsimp
  using prefix-order.lift-Suc-mono-le by blast

definition wf-to-wfP where
  wf-to-wfP r  $\equiv \lambda x y. (x, y) \in r$ 

end

```

2 The definition of the While language

```
theory WhileLang imports MiscLemmas Coinductive.Coinductive-List begin
```

```
type-synonym name = char list
type-synonym val = nat
type-synonym store = name ⇒ val
type-synonym exp = store ⇒ val
```

```
datatype prog =
  Skip
  | Assign name exp
  | Print exp
  | Seq prog prog
  | If exp prog prog
  | While exp prog
```

```
type-synonym out = val list
type-synonym state = store × out
```

```
definition output-of :: state ⇒ out where
  output-of ≡ λ(-, out). out
```

```
definition subst :: name ⇒ exp ⇒ state ⇒ state where
  subst n e ≡ λ(s, out). (s(n:= e s), out)
```

```
definition print :: exp ⇒ state ⇒ state where
  print e ≡ λ(s, out). (s, out @ [e s])
```

```
definition guard :: exp ⇒ state ⇒ bool where
  guard x ≡ λ(s, -). x s ≠ 0
```

```
inductive step :: prog × state ⇒ prog × state ⇒ bool where
  step-skip: step (Skip, s) (Skip, s)
  | step-assign: step (Assign n x, s) (Skip, subst n x s)
  | step-print: step (Print x, s) (Skip, print x s)
  | step-seq1: step (Seq Skip q, s) (q, s)
  | step-seq2: step (p0, s0) (p1, s1) ⇒ p0 ≠ Skip ⇒ step (Seq p0 q, s0) (Seq p1 q, s1)
  | step-if: step (If x p q, s) ((if guard x s then p else q), s)
  | step-while: step (While x p, s) (If x (Seq p (While x p)), Skip, s)
```

```
declare step.intros[simp,intro]
```

```
inductive-cases skipE[elim!]: step (Skip, s) ct
```

```

inductive-cases assignE[elim!]: step (Assign n x, s) ct
inductive-cases printE[elim!]: step (Print x, s) ct
inductive-cases seqE[elim]: step (Seq c1 c2, s) ct
inductive-cases ifE[elim!]: step (If x c1 c2, s) ct
inductive-cases whileE[elim]: step (While x p, s) ct

lemmas step-induct = step.induct[split-format(complete)]

inductive terminates where
star step (p, s) (Skip, t)  $\implies$  terminates s p t

inductive diverges where
 $\forall t. \neg \text{terminates } s p t$ 
 $\wedge \text{out} = \text{lSup} \{ \text{llist-of out} \mid \text{out}. \exists q t. \text{star step } (p, s) (q, t, \text{out}) \}$ 
 $\implies$ 
diverges s p out

lemma step-exists':
 $\exists t. \text{step } (\text{prog}, (s, \text{out})) t$ 
proof (induct prog)
case Skip
then show ?case by fastforce
next
case (Assign x1 x2)
then show ?case
apply clarsimp
apply (rule-tac x=Skip in exI)
apply (rule-tac x=fst (subst x1 x2 (s, out)) in exI)
apply (rule-tac x=snd (subst x1 x2 (s, out)) in exI)
by fastforce
next
case (Print x)
then show ?case
apply clarsimp
apply (rule-tac x=Skip in exI)
apply (rule-tac x=fst (print x (s, out)) in exI)
apply (rule-tac x=snd (print x (s, out)) in exI)
by fastforce
next
case (Seq prog1 prog2)
then show ?case
apply (case-tac prog1 = Skip)
by fastforce+
next
case (If x1 prog1 prog2)
then show ?case by fastforce
next

```

```

case (While  $x_1$  prog)
  then show ?case by fastforce
qed

theorem step-exists:
   $\forall s. \exists t. step\ s\ t$ 
  using step-exists' by simp

theorem terminates-or-diverges:
   $(\exists t. terminates\ s\ p\ t) \vee (\exists output. diverges\ s\ p\ output)$ 
  by (clarsimp simp: diverges.simps)

lemma step-deterministic':
   $step\ (prog, st, out)\ t_1 \implies step\ (prog, st, out)\ t_2 \implies t_1 = t_2$ 
proof (induct prog arbitrary:  $t_1\ t_2$ )
  case Skip
  then show ?case by clarsimp
next
  case (Assign  $x_1\ x_2$ )
  then show ?case by clarsimp
next
  case (Print  $x$ )
  then show ?case by clarsimp
next
  case (Seq prog1 prog2)
  then show ?case by (elim seqE;clarsimp)
next
  case (If  $x_1$  prog1 prog2)
  then show ?case byclarsimp
next
  case (While  $x_1$  prog)
  then show ?case by (elim whileE;clarsimp)
qed

theorem step-deterministic:
   $step\ s\ t_1 \implies step\ s\ t_2 \implies t_1 = t_2$ 
  apply (cases s)
  using step-deterministic' byclarsimp

lemma star-step-refl:
   $star\ step\ (Skip, t_1) (Skip, t_2) \implies t_1 = t_2$ 
  by (induct (Skip,  $t_1$ ) (Skip,  $t_2$ ) arbitrary:  $t_1\ t_2$  rule: star.induct;clarsimp)

theorem terminates-deterministic:
   $terminates\ s\ p\ t_1 \implies terminates\ s\ p\ t_2 \implies t_1 = t_2$ 
proof (simp add: terminates.simps, induct (p, s) (Skip,  $t_1$ ) arbitrary:  $p\ s\ t_1$  rule: star.induct)
  case refl
  then show ?case

```

```

by (clarsimp intro!: star-step-refl)
next
  case (step y)
  then show ?case
    apply (rotate-tac 3)
    apply (erule star.cases)
    applyclarsimp
    apply simp
    apply (drule (1) step-deterministic)
    apply simp
    apply (drule-tac x=fst ya and y=snd ya in meta-spec2)
    by simp
qed

```

```

theorem diverges-deterministic:
  diverges s p t1  $\Rightarrow$  diverges s p t2  $\Rightarrow$  t1 = t2
  by (simp add: diverges.simps)

```

```
end
```

3 A standard total correctness Hoare logic

```
theory StdLogic imports WhileLang begin
```

```

inductive hoare :: (state  $\Rightarrow$  bool)  $\Rightarrow$  prog  $\Rightarrow$  (state  $\Rightarrow$  bool)  $\Rightarrow$  bool where
  h-skip[simp,intro!]: hoare P Skip P
  | h-assign[simp,intro!]: hoare ( $\lambda s. Q$  (subst n x s)) (Assign n x) Q
  | h-print[simp,intro!]: hoare ( $\lambda s. Q$  (print x s)) (Print x) Q
  | h-seq[intro]: hoare P p M  $\Rightarrow$  hoare M q Q  $\Rightarrow$  hoare P (Seq p q) Q
  | h-if[intro!]: hoare ( $\lambda s. P s \wedge$  guard x s) p Q
     $\Rightarrow$  hoare ( $\lambda s. P s \wedge \neg$  guard x s) q Q  $\Rightarrow$  hoare P (If x p q) Q
  | h-while : ( $\bigwedge s0.$  hoare ( $\lambda s. P s \wedge$  guard x s  $\wedge$  s = s0) p ( $\lambda s. P s \wedge R s s0$ ))
     $\Rightarrow$  wfP R  $\Rightarrow$  hoare P (While x p) ( $\lambda s. P s \wedge \neg$  guard x s)
  | h-weaken: ( $\bigwedge s. P s \Rightarrow P' s$ )  $\Rightarrow$  hoare P' p Q'  $\Rightarrow$  ( $\bigwedge s. Q' s \Rightarrow Q s$ )  $\Rightarrow$ 
    hoare P p Q

```

```

definition hoare-sem :: (state  $\Rightarrow$  bool)  $\Rightarrow$  prog  $\Rightarrow$  (state  $\Rightarrow$  bool)  $\Rightarrow$  bool where
  hoare-sem P p Q  $\equiv$ 
    ( $\forall s. P s \rightarrow (\exists t. terminates s p t \wedge Q t)$ )

```

```
end
```

4 Lemmas about the While language

```

theory WhileLangLemmas imports WhileLang Coinductive.Coinductive-List-Prefix
begin

lemma NRC-step-deterministic:
  star-n step n x y ==> star-n step n x z ==> y = z
proof (induct rule: star-n.induct)
  case (refl-n x)
  then show ?case
    apply -
    by (erule star-n.cases; simp)
next
  case (step-n x y n z)
  then show ?case
    apply (rotate-tac 3)
    apply (erule star-n.cases, simp)
    apply simp
    apply (drule (1) step-deterministic)
    by simp
qed

inductive exec where
  exec-skip: exec s Skip s
  | exec-assign: exec s (Assign n x) (subst n x s)
  | exec-print: exec s (Print x) (print x s)
  | exec-seq: exec s0 p s1 ==> exec s1 q s2 ==> exec s0 (Seq p q) s2
  | exec-if: exec s (if guard x s then p else q) s1 ==> exec s (If x p q) s1
  | exec-while1: ¬guard x s ==> exec s (While x p) s
  | exec-while2: guard x s ==> exec s p s1 ==> exec s1 (While x p) s2
    ==> exec s (While x p) s2

declare exec.intros[intro!]

lemma NRC-step[simp]:
  star-n step n (Skip,s) (Skip,t) ==> s = t
  by (induct n (Skip, s) (Skip, t) arbitrary: s t rule: star-n.induct; clar simp)

lemma terminates-Skip:
  terminates s Skip t <=> s = t
  by (fastforce simp: terminates.simps star-eq-star-n)

lemma NRC-assign[simp]:
  star-n step n (Assign n' x,s) (Skip, t) ==> t = subst n' x s
  by (induct n (Assign n' x,s) (Skip, t) arbitrary: n' x s t rule: star-n.induct)
    (fastforce dest: NRC-step)

lemma terminates-Assign:
  terminates s (Assign n x) t <=> t = subst n x s

```

```

by (fastforce simp: terminates.simps star-eq-star-n)

lemma NRC-print[simp]:
  star-n step n (Print x,s) (Skip, t)  $\implies$  t = print x s
  by (induct n (Print x,s) (Skip, t) arbitrary: x s t rule: star-n.induct)
    (fastforce dest: NRC-step)

lemma terminates-Print:
  terminates s (Print x) t  $\longleftrightarrow$  t = print x s
  by (fastforce simp: terminates.simps star-eq-star-n)

lemma terminates-If:
  terminates s (If x p q) t  $\longleftrightarrow$  terminates s (if guard x s then p else q) t
proof -
  have NRC-if1:
    star-n step n (prog.If x p q, s) (Skip, t)  $\implies$ 
       $\exists n.$  star-n step n (if guard x s then p else q, s) (Skip, t) for n
    by (induct n (If x p q,s) (Skip, t) arbitrary: x p q s t rule: star-n.induct)
      fastforce
  have NRC-if2:
    star-n step n (if guard x s then p else q, s) (Skip, t)  $\implies$ 
       $\exists n.$  star-n step n (prog.If x p q, s) (Skip, t) for n
    proof (induct n (if guard x s then p else q, s) (Skip, t)
      arbitrary: x p q s t rule: star-n.induct)
      case refl-n
      then show ?case by (fastforce split: if-split-asm)
    next
      case (step-n y n)
      then show ?case by (fastforce split: if-split-asm)
    qed
    show ?thesis
    by (fastforce dest: NRC-if1 NRC-if2 simp: star-eq-star-n terminates.simps)
  qed

lemma terminates-While:
  terminates s (While f c) t  $\longleftrightarrow$  terminates s (If f (Seq c (While f c)) Skip) t
proof -
  have NRC-while1:
    star-n step n (While f c, s) (Skip, t)  $\implies$ 
       $\exists n.$  star-n step n (prog.If f (Seq c (While f c)) Skip, s) (Skip, t) for n
    by (induct n (While f c,s) (Skip, t) arbitrary: f c s t rule: star-n.induct)
      fastforce

  have NRC-while2:
    star-n step n (prog.If f (Seq c (While f c)) Skip, s) (Skip, t)  $\implies$ 
       $\exists n.$  star-n step n (While f c, s) (Skip, t) for n
    by (induct n (If f (Seq c (While f c)) Skip, s) (Skip, t)
      arbitrary: f c s t rule: star-n.induct)
      fastforce

```

```

show ?thesis
  by (fastforce dest: NRC-while1 NRC-while2 simp: star-eq-star-n terminates.simps)
qed

definition real-step where
  real-step ≡ λ(p, s) qt. p ≠ Skip ∧ step (p,s) qt

lemma terminates:
  terminates s p t ←→ star real-step (p,s) (Skip,t)

proof –
  have P1: star-n step n (p, s) (Skip, t) ==>
    ∃ n. star-n (λ(p, s) qt. p ≠ Skip ∧ step (p, s) qt) n (p, s) (Skip, t) for n
  proof (induct n (p, s) (Skip, t) arbitrary: p s t rule: star-n.induct)
    case refl-n
    then show ?case by fastforce
  next
    case (step-n y n)
    then show ?case
      apply (drule-tac x=fst y and y=snd y in meta-spec2)
      apply simp
      by blast
    qed
    have P2: star-n (λ(p, s) qt. p ≠ Skip ∧ step (p, s) qt) n (p, s) (Skip, t) ==>
      ∃ n. star-n step n (p, s) (Skip, t) for n
    by (induct n (p, s) (Skip, t) arbitrary: p s t rule: star-n.induct) fastforce+
    show ?thesis
      by (fastforce dest: P1 P2 simp: real-step-def star-eq-star-n terminates.simps)
  qed

lemma NRC-real-step-Skip[simp]:
  star-n real-step n (Skip,s) (Skip,t) ←→ n = 0 ∧ s = t
  apply (rule iffI; simp?)
  by (induct n (Skip, s) (Skip, t) arbitrary: s t rule: star-n.induct)
    (simp add: real-step-def)+

lemma NRC-real-step-not-Skip:
  p ≠ Skip ==>
  (star-n real-step n (p,s) (Skip,t) ←→
   (∃ k m. real-step (p,s) m ∧ star-n real-step k m (Skip,t) ∧ n = k + 1))
  apply (rule iffI; rotate-tac)
  by (induct n (p, s) (Skip, t) arbitrary: p s t rule: star-n.induct, simp)
    fastforce+

lemma real-step-seqE:
  real-step (Seq p q, s) x ==>
  (x = (q, s) ==> p = Skip ==> P) ==>
  (∀ p' s'. x = (Seq p' q, s') ==> real-step (p, s) (p', s') ==> p ≠ Skip ==> P)
  ==> P
  by (clarsimp simp: real-step-def) (erule seqE; clarsimp)

```

```

lemma real-steps-Seq:
  star-n real-step n (Seq p q,s) (Skip,t)  $\longleftrightarrow$ 
    ( $\exists$  n1 n2 m.
      star-n real-step n1 (p,s) (Skip,m)  $\wedge$ 
      star-n real-step n2 (q,m) (Skip,t)  $\wedge$  n = n1 + n2 + 1)
proof
  assume H: star-n real-step n (Seq p q, s) (Skip, t)
  show  $\exists$  n1 n2 m.
    star-n real-step n1 (p, s) (Skip, m)  $\wedge$ 
    star-n real-step n2 (q, m) (Skip, t)  $\wedge$  n = n1 + n2 + 1 using H
proof (induct n (Seq p q,s) (Skip,t) arbitrary: p q s t rule: star-n.induct)
  fix y n p q s t
  assume P1: real-step (Seq p q, s) y
  and P2: star-n real-step n y (Skip, t)
  and IH:  $\bigwedge$ p q s. y = (Seq p q, s)  $\implies$ 
    ( $\exists$  n1 n2 r. star-n real-step n1 (p, s) (Skip, r)  $\wedge$ 
      star-n real-step n2 (q, r) (Skip, t)  $\wedge$ 
      n = n1 + n2 + 1)
  then show  $\exists$  n1 n2 r. star-n real-step n1 (p, s) (Skip, r)  $\wedge$ 
    star-n real-step n2 (q, r) (Skip, t)  $\wedge$  Suc n = n1 + n2 + 1
    by (fastforce elim!: real-step-seqE)
qed
next
  assume H:  $\exists$  n1 n2 m.
    star-n real-step n1 (p, s) (Skip, m)  $\wedge$ 
    star-n real-step n2 (q, m) (Skip, t)  $\wedge$  n = n1 + n2 + 1
  show star-n real-step n (Seq p q, s) (Skip, t)
    using H
    apply -
    apply (case-tac p=Skip, clarsimp)
    apply (rule step-n[rotated], simp)
    apply (clarsimp simp: real-step-def)
    apply (elim exE conjE)
proof -
  have H: star-n real-step n1 (p, s) (Skip, m)  $\implies$ 
    star-n real-step n2 (q, m) (Skip, t)  $\implies$ 
    p  $\neq$  Skip  $\implies$ 
    star-n real-step (n1 + n2 + 1) (Seq p q, s) (Skip, t) for n1 n2 m
  apply (induct n1 (p, s) (Skip, m) arbitrary: s p m rule: star-n.induct)
    apply (clarsimp simp: real-step-def)
    apply (rename-tac y n s p m)
    apply (subgoal-tac real-step (Seq p q, s) (Seq (fst y) q, snd y))
    prefer 2
    apply (clarsimp simp: real-step-def)
    apply (case-tac fst y = Skip)
    apply (subgoal-tac  $\bigwedge$ a b. fst a = b  $\implies$  ( $\exists$  c. a = (b, c)  $\wedge$  c = snd a))
    prefer 2
    applyclarsimp

```

```

apply (drule-tac x=y and y=Skip in meta-spec2)
apply (drule (1) meta-mp)
apply (elim exE conjE)
apply (rename-tac c)
apply (thin-tac c = snd y)
apply simp
apply (erule step-n)
apply (rule step-n[rotated])
apply simp
apply (clarsimp simp: real-step-def)
applyclarsimp
done
thus  $\bigwedge n1\ n2\ m.$ 
  p ≠ Skip  $\implies$ 
  star-n real-step n1 (p, s) (Skip, m)  $\implies$ 
  star-n real-step n2 (q, m) (Skip, t)  $\implies$ 
  n = n1 + n2 + 1  $\implies$ 
  star-n real-step n (Seq p q, s) (Skip, t) by blast
qed
qed

lemma terminates-Seq:
terminates s (Seq p q) t  $\longleftrightarrow$  ( $\exists m.$  terminates s p m  $\wedge$  terminates m q t)
by (fastforce simp: terminates star-eq-star-n real-steps-Seq)

lemma terminates-eq-exec:
terminates s p t  $\longleftrightarrow$  exec s p t
proof –
have P1: star-n real-step n (p, s) (Skip, t)  $\implies$  exec s p t for n
proof (induct n arbitrary: p s t rule: nat-less-induct)
  case (1 n)
  then show ?case
    apply –
    apply (case-tac p;clarsimp)
    apply (erule star-n.cases;
     clarsimp simp: real-step-def NRC-real-step-Skip[simplified real-step-def])
    apply (drule sym[where s=subst - - -],clarsimp)
    apply (erule star-n.cases;
     clarsimp simp: real-step-def NRC-real-step-Skip[simplified real-step-def])
    apply (drule sym[where s=print - -],clarsimp)
    apply (clarsimp simp: real-steps-Seq)
    apply (metis 1.hyps exec-seq less-add-Suc1 less-add-Suc2)
    apply (erule star-n.cases;
     clarsimp simp: real-step-def NRC-real-step-Skip[simplified real-step-def]
      split: if-split-asm)
    apply fastforce
    apply fastforce
    apply (erule star-n.cases;clarsimp simp: real-step-def)
    apply (erule whileE,clarsimp)

```

```

apply (erule star-n.cases;
      clarsimp split: if-split-asm simp: NRC-real-step-Skip[simplified real-step-def])
defer
apply fastforce
apply (clarsimp simp: real-steps-Seq[simplified real-step-def])
by (meson exec-while2 less-SucI less-add-Suc1 less-add-Suc2)
qed
then have P1': terminates s p t  $\implies$  exec s p t
  by (clarsimp simp: terminates-star-eq-star-n)
have P2: exec s p t  $\implies$  terminates s p t
proof (induct s p t rule: exec.induct; clarsimp)
  case (exec-skip s)
  then show ?case by (clarsimp simp: terminates-Skip)
next
  case (exec-assign s n x)
  then show ?case by (clarsimp simp: terminates-Assign)
next
  case (exec-print s x)
  then show ?case by (clarsimp simp: terminates-Print)
next
  case (exec-seq s0 p s1 q s2)
  then show ?case by (cases s1; fastforce simp: terminates-Seq)
next
  case (exec-if s x p q s1)
  then show ?case by (cases s1; fastforce simp: terminates-If)
next
  case (exec-while1 x s p)
  then show ?case by (clarsimp simp: terminates-While terminates-If terminates-Skip)
next
  case (exec-while2 x s p s1 s2)
  then show ?case
    apply (simp (no-asm) add: terminates-While terminates-If terminates-Skip)
    by (cases s1; fastforce simp: terminates-Seq)
qed
show ?thesis using P1' P2 by auto
qed

lemma terminates-While-NRC:
assumes terminates m p t
assumes p = While f c
shows  $\exists n. \text{star-}n (\lambda s t. \text{guard } f s \wedge \text{terminates } s c t) n m t \wedge \neg \text{guard } f t$ 
using assms
apply (simp add: terminates-eq-exec)
by (induct m While f c t arbitrary: f c rule: exec.induct; clarsimp; fastforce)

lemma not-diverges[simp]:
~diverges s Skip l
~diverges s (Assign n x) l

```

```

 $\sim \text{diverges } s (\text{Print } x) l$ 
by (clar simp simp: diverges.simps terminates-Skip terminates-Assign terminates-Print;
      metis surj-pair)+

lemma star-n-Skip:
  star-n step n (Skip, s) (c', t)  $\implies c' = \text{Skip} \wedge t = s$ 
  apply (induct n)
  apply (erule star-n.cases; clar simp)+
  done

lemma star-n-Seq:
  star-n step n (c, s) (c', t)  $\implies$ 
     $\exists n. \text{star-n step } n (\text{Seq } c p, s) (\text{Seq } c' p, t)$ 
  apply (case-tac c=Skip)
  apply (fastforce dest!: star-n-Skip)
  apply (induct n arbitrary: c s)
  apply (erule star-n.cases; simp; rename-tac x; case-tac x; rule-tac x=0 in exI,
  clar simp)
  apply (erule star-n.cases, simp)
  apply (rename-tac y na z)
  apply (drule-tac x=fst y and y=snd y in meta-spec2)
  apply clar simp
  apply (rename-tac ac ad bb na af bc)
  apply (case-tac ac=Skip, simp)
  apply (fastforce dest!: star-n-Skip)
  apply clar simp
  apply fastforce
  done

lemma RTC-Seq:
  star step (c,s) (c',t)  $\implies$ 
    star step (Seq c p, s) (Seq c' p, t)
  apply (clar simp simp: star-eq-star-n star-n-Seq)
  done

lemma step-output-mono:
  step s t  $\implies$  prefix (output-of (snd s)) (output-of (snd t))
  by (induct rule: step.induct; simp add: subst-def print-def split-def output-of-def)

lemma NRC-step-output-mono:
  star-n step k (c,s) (c',s')  $\implies$  prefix (output-of s) (output-of s')
  apply (induct k arbitrary: c s c' s')
  apply (erule star-n.cases; clar simp)
  apply (erule star-n.cases; clar simp)
  apply (drule step-output-mono)
  apply fastforce
  done

lemma lprefix-chain-RTC-step':

```

```

Complete-Partial-Order.chain lprefix {llist-of out | out. ( $\exists q t. step x (q,t,out))\}$ 
apply (rule chainI)
apply simp
apply (elim exE impE conjE)
by (metis lprefix-down-linear lprefix-llist-ofI prod.inject step-deterministic)

lemma star-n-step-decompose:
star-n step n x y  $\implies$  star-n step n' x z  $\implies$  n' < n
 $\implies$  star-n step (n - n') z y
by (fastforce elim!: star-n-decompose step-deterministic NRC-step-deterministic)

lemma lprefix-chain-NRC-step':
star-n step n x (q, t, out)  $\implies$ 
star-n step n' x (q', t', out')  $\implies$ 
lprefix (llist-of out) (llist-of out')  $\vee$  lprefix (llist-of out') (llist-of out)
apply (insert less-linear[of n n'])
apply (elim disjE)
defer 2
apply (drule (2) star-n-step-decompose,
metis NRC-step-output-mono internal-case-prod-conv internal-case-prod-def
lprefix-llist-of output-of-def)+
apply simp
apply (drule (1) NRC-step-deterministic)
by fastforce

lemma lprefix-chain-NRC-step:
Complete-Partial-Order.chain lprefix {llist-of out | out. ( $\exists q t. star-n step n x (q,t,out))\}$ 
by (rule chainI) (force dest: lprefix-chain-NRC-step')

lemma lprefix-chain-RTC-step:
Complete-Partial-Order.chain lprefix {llist-of out | out. ( $\exists q t. star step x (q,t,out))\}$ 
by (rule chainI) (force dest: lprefix-chain-NRC-step' star-star-n)

lemma lprefix-chain-NRC-step-ex:
Complete-Partial-Order.chain lprefix {llist-of out | out. ( $\exists q t n. star-n step n x (q,t,out))\}$ 
by (subst star-eq-star-n[symmetric]) (rule lprefix-chain-RTC-step)

definition lprefix-rel where
lprefix-rel ls ls'  $\equiv$   $\forall l \in ls. \exists l' \in ls'. lprefix l l'$ 

lemma diverges-unique:
diverges s p l  $\implies$   $\forall l'. diverges s p l' \longrightarrow l' = l$ 
using diverges-deterministic by blast

lemma terminates-unique:
terminates s p t  $\implies$   $\forall t'. terminates s p t' \longrightarrow t' = t$ 
using terminates-deterministic by blast

```

```

lemma star-n-real-step-Seq-exact:
  star-n real-step n (c, s) (Skip, t)  $\implies$  star-n step n (Seq c c', s) (Seq Skip c', t)
  by (induct (c, s) (Skip, t) arbitrary: c s t rule: star-n.induct) (auto simp: real-step-def)

lemma star-n-real-step-Seq-exact':
  star-n real-step n (c, s) (Skip, t)  $\implies$  star-n real-step n (Seq c c', s) (Seq Skip c', t)
  by (induct (c, s) (Skip, t) arbitrary: c s t rule: star-n.induct) (auto simp: real-step-def)

lemma div-Seq1-always-Seq:
   $\llbracket \text{star-n step n } (\text{Seq } c \ c', \ s) \ (q, \ s'); \ c \neq \text{Skip};$ 
   $\forall a \ b \ n. \neg \text{star-n step n } (c, \ s) \ (\text{Skip}, \ a, \ b) \rrbracket$ 
   $\implies \exists u. \ q = \text{Seq } u \ c' \wedge u \neq \text{Skip}$ 
proof (induct (Seq c c', s) (q, s') arbitrary: c c' q s s' rule: star-n.induct)
  case (step-n y n)
  then show ?case
    by cases fastforce+
qed clarsimp

lemma div-Seq1-steps:
   $\llbracket \text{star-n step n } (\text{Seq } c \ c', \ s) \ (\text{Seq } u \ c', \ s'); \ c \neq \text{Skip};$ 
   $\forall a \ b \ n. \neg \text{star-n step n } (c, \ s) \ (\text{Skip}, \ a, \ b) \rrbracket$ 
   $\implies \text{star-n step n } (c, \ s) \ (u, \ s')$ 
proof (induct (Seq c c', s) (Seq u c', s') arbitrary: c c' u s s' rule: star-n.induct)
  case (step-n y n)
  then show ?case by cases fastforce+
qed fastforce

lemma div-Seq1-lSup-eq:
   $\forall t. \neg \text{terminates } s \ c \ t$ 
   $\implies lSup \ \{ llist-of \ out \mid out. \exists q \ t. \text{star step } (c, \ s) \ (q, \ t, \ out) \} =$ 
   $lSup \ \{ llist-of \ out \mid out. \exists q \ t. \text{star step } (\text{Seq } c \ c', \ s) \ (q, \ t, \ out) \}$ 
  apply (rule lprefix-antisym)
  apply (rule chain-lSup-lprefix[OF lprefix-chain-RTC-step])
  apply (rule chain-lprefix-lSup[OF lprefix-chain-RTC-step])
  apply (clarsimp simp: terminates star-eq-star-n)
  apply (drule star-n-Seq)
  apply fastforce
  apply (rule chain-lSup-lprefix[OF lprefix-chain-RTC-step])
  apply (rule chain-lprefix-lSup[OF lprefix-chain-RTC-step])
  applyclarsimp
  apply (clarsimp simp: terminates.simps star-eq-star-n)
  apply (frule div-Seq1-always-Seq)
  apply (metis (no-types, opaque-lifting) star-n.simps surj-pair)

```

```

apply simp
apply (erule star-n.cases, fastforce)
by (metis (mono-tags, opaque-lifting) div-Seq1-steps prod.collapse refl-n step-n)

lemma star-n-real-step-step:
star-n real-step n x y ==> star-n step n x y
by (induct rule: star-n.induct; clarsimp)
(metis (no-types, lifting) prod.simps(2) real-step-def star-n.simps)

lemma div-Seq2-steps:
[| star-n real-step n' (c, s) (Skip, s'); n' < n;
  ∀ t' n. ¬ star-n real-step n (c', s') (Skip, t');
  star-n step n (Seq c c', s) (q, t, out)] ==
  ⇒ ∃ m q t'. star-n step m (c', s') (q, t', out)
apply (drule star-n-real-step-Seq-exact[where c'=c'])
apply (drule (2) star-n-step-decompose)
apply (erule-tac ?a1.0=n - n' in star-n.cases)
apply clarsimp
apply clarsimp
apply (erule step.cases; clarsimp)
apply fastforce
done

lemma div-Seq2-lSub-eq:
[| terminates s c s'; ∀ t. ¬ terminates s' c' t|] ==
  lSup {llist-of out | out. ∃ q t. star step (Seq c c', s) (q, t, out)} =
  lSup {llist-of out | out. ∃ q t. star step (c', s') (q, t, out)}
apply (rule lprefix-antisym)
apply (rule chain-lSup-lprefix[OF lprefix-chain-RTC-step])
apply clarsimp
apply (clarsimp simp: terminates star-eq-star-n)
apply (rename-tac n na)
apply (cases s; cases s'; simp)
apply (case-tac n < na)
apply (rule chain-lprefix-lSup[OF lprefix-chain-NRC-step-ex])
apply (drule (1) div-Seq2-steps; fastforce)
apply (clarsimp simp: not-less)
apply (drule star-n-real-step-Seq-exact[where c'=c'])
apply (rename-tac aa ba)
apply (subgoal-tac lprefix (llist-of ba)
  (lSup {llist-of out | out. ∃ q t n. star-n step n (c', aa, ba) (q, t,
  out)}))
prefer 2
apply (rule chain-lprefix-lSup[OF lprefix-chain-NRC-step-ex])
apply fastforce
apply (case-tac n=na; simp?)
apply (drule (1) NRC-step-deterministic)
apply clarsimp
apply (drule le-neq-trans,clarsimp, simp)

```

```

apply (frule-tac n=n and n'=na in star-n-step-decompose; simp?)
apply (drule-tac k=n-na in NRC-step-output-mono)
apply (clarsimp simp: output-of-def)
apply (meson lprefix-llist-of lprefix-trans)
apply (rule chain-lSup-lprefix[OF lprefix-chain-RTC-step])
apply (rule chain-lprefix-lSup[OF lprefix-chain-RTC-step])
apply (clarsimp simp: terminates star-eq-star-n)
apply (drule star-n-real-step-Seq-exact[where c'=c'])
apply (subgoal-tac step (Seq Skip c', s') (c', s'))
apply (drule (1) step-n-rev)
apply (clarsimp dest!: star-n-star simp: star-eq-star-n[symmetric])
apply (drule (1) star-trans)
apply fastforce
apply clarsimp
done

lemma diverges-Seq:
  diverges s (Seq c'') l  $\longleftrightarrow$  diverges s c l  $\vee$  ( $\exists t$ . terminates s c t  $\wedge$  diverges t c' l)
apply (rule iffI)
apply (case-tac  $\exists t$ . terminates s c t; simp)
prefer 2
apply (erule diverges.cases)
apply (rule diverges.intros)
apply (clarsimp simp: terminates.simps star-eq-star-n)
apply (metis (no-types, opaque-lifting) div-Seq1-steps div-Seq1-always-Seq refl-n
star-n-Seq)
apply (rule disjI2, clarsimp)
apply (rename-tac a b)
apply (rule-tac x=a in exI, rule-tac x=b in exI,clarsimp)
apply (clarsimp simp: diverges.simps terminates-Seq)
apply (simp add: terminates-Seq forall-swap4)
apply (drule-tac x=a and y=b in spec2)
apply clarsimp
apply (drule div-Seq2-lSub-eq; fastforce)
apply (erule disjE)
apply (rule diverges.intros,clarsimp)
apply (rule conjI;clarsimp)
apply (erule diverges.cases,clarsimp)
apply (simp add: terminates-Seq)
apply (fastforce simp: div-Seq1-lSup-eq diverges.simps)
applyclarsimp
apply (erule diverges.cases,clarsimp)
apply (rule diverges.intros,clarsimp)
apply (rule conjI;clarsimp)
using terminates-Seq terminates-deterministic apply blast
apply (drule div-Seq2-lSub-eq)
by simp+

```

lemma div-true-If-lSup-eq:

```

 $\llbracket \forall t. \neg \text{terminates } s p t; \text{guard } f s \rrbracket$ 
 $\implies lSup \{ llist-of out | out. \exists qa t. \text{star step} (\text{prog.If } f p q, s) (qa, t, out) \} =$ 
 $lSup \{ llist-of out | out. \exists q t. \text{star step} (p, s) (q, t, out) \}$ 
apply (rule lprefix-antisym)
apply (rule chain-lSup-lprefix[OF lprefix-chain-RTC-step])
apply (rule chain-lprefix-lSup[OF lprefix-chain-RTC-step])
apply clarsimp
apply (erule star.cases; fastforce)
apply (rule chain-lSup-lprefix[OF lprefix-chain-RTC-step])
apply (rule chain-lprefix-lSup[OF lprefix-chain-RTC-step])
apply clarsimp
apply (subgoal-tac step (If f p q, s) (if guard f s then p else q, s))
apply simp
apply (drule (1) step[where r=step])
apply fastforce
apply (rule step-if)
done

lemma div-false-If-lSup-Eq:
 $\llbracket \forall t. \neg \text{terminates } s q t; \neg \text{guard } f s \rrbracket$ 
 $\implies lSup$ 
 $\{ llist-of out | out. \exists qa t. \text{star step} (\text{prog.If } f p q, s) (qa, t, out) \} =$ 
 $lSup \{ llist-of out | out. \exists q' t. \text{star step} (q, s) (q', t, out) \}$ 
apply (rule lprefix-antisym)
apply (rule chain-lSup-lprefix[OF lprefix-chain-RTC-step])
apply (rule chain-lprefix-lSup[OF lprefix-chain-RTC-step])
apply clarsimp
apply (erule star.cases; fastforce)
apply (rule chain-lSup-lprefix[OF lprefix-chain-RTC-step])
apply (rule chain-lprefix-lSup[OF lprefix-chain-RTC-step])
apply clarsimp
apply (subgoal-tac step (If f p q, s) (if guard f s then p else q, s))
apply simp
apply (drule (1) step[where r=step])
apply fastforce
apply (rule step-if)
done

lemma diverges-If:


diverges s (If f p q) l = diverges s (if guard f s then p else q) l

by (fastforce simp: diverges.simps terminates-If


div-true-If-lSup-eq div-false-If-lSup-Eq)

lemma While-body-add3real-step:
 $\llbracket \text{star-}n \text{ real-step } n (c, s) (\text{Skip}, s'); \text{guard } g s \rrbracket$ 
 $\implies \text{star-}n \text{ real-step } (n + 3) (\text{While } g c, s) (\text{While } g c, s')$ 
apply (drule star-n-real-step-Seq-exact[where c'=While g c])
apply (subgoal-tac real-step (Seq Skip (While g c), s') (While g c, s'))
apply (drule (1) step-n-rev)

```

```

apply (thin-tac real-step - -)
apply (subgoal-tac real-step (If g (Seq c (While g c)) Skip, s)
         (if guard g s then Seq c (While g c) else Skip, s))
apply (simp only: if-True)
apply (drule (1) step-n[rotated])
apply (thin-tac real-step - -)
apply (subgoal-tac real-step (While g c, s) (If g (Seq c (While g c)) Skip, s))
apply (drule (1) step-n[rotated])
apply (thin-tac real-step - -)
apply (simp add: numeral-3-eq-3)
apply (fastforce simp: real-step-def)+
apply (simp (no-asm) add: real-step-def)
apply (fastforce simp: real-step-def)
done

```

lemmas *While-body-add3step = While-body-add3real-step[THEN star-n-real-step-step]*

```

lemma div-body-While-lSup-eq:
 $\llbracket \text{guard } g \text{ s; } \forall t. \neg \text{terminates } s \text{ c } t \rrbracket$ 
 $\implies lSup \{ llist-of \text{out} \mid \text{out}. \exists q \text{ t. star step (c, s) (q, t, out)} \} =$ 
 $lSup \{ llist-of \text{out} \mid \text{out}. \exists q \text{ t. star step (While g c, s) (q, t, out)} \}$ 
apply (rule lprefix-antisym)
apply (rule chain-lSup-lprefix[OF lprefix-chain-RTC-step])
apply clarsimp
apply (rule chain-lprefix-lSup[OF lprefix-chain-RTC-step])
apply (clarsimp simp: star-eq-star-n)
apply (subgoal-tac step (If g (Seq c (While g c)) Skip, s)
         (if guard g s then (Seq c (While g c)) else Skip, s))
prefer 2
apply (rule step-if)
apply simp
apply (subgoal-tac step (While g c, s) (If g (Seq c (While g c)) Skip, s))
prefer 2
apply (rule step-while)
apply (drule star-n-Seq[where p=While g c])
apply (elim exE)
apply fastforce
apply (frule div-Seq1-lSup-eq[where c'=While g c])
apply (frule div-true-If-lSup-eq[where p=Seq c (While g c) and q=Skip, rotated,
THEN sym])
apply (clarsimp simp: terminates-Seq)
apply simp
apply (rule chain-lSup-lprefix[OF lprefix-chain-RTC-step])
apply (rule chain-lprefix-lSup[OF lprefix-chain-RTC-step])
apply (clarsimp simp: star-eq-star-n)
apply (erule star-n.cases)
apply fastforce
apply clarsimp
apply (erule step.cases; simp)

```

```

apply clarsimp
apply fastforce
done

lemma inf-loop-While-steps:
  [ star-n real-step n' (c, s) (Skip, s'); n' + 3 < n; guard g s;
    star-n step n (While g c, s) (q, t, out) ]
  ==> ∃ q' t' n. star-n step n (While g c, s') (q', t', out)
by(blast dest: While-body-add3step star-n-step-decompose)

lemma inf-loop-While-lSup-eq:
  [guard g s; terminates s c (a, b); ∀ aa ba. ¬ terminates (a, b) (While g c) (aa, ba)]
  ==> lSup {llist-of out | out. ∃ q t. star step (While g c, a, b) (q, t, out)} =
      lSup {llist-of out | out. ∃ q t. star step (While g c, s) (q, t, out)}
apply (rule lprefix-antisym)
apply (rule chain-lSup-lprefix[OF lprefix-chain-RTC-step])
apply (rule chain-lprefix-lSup[OF lprefix-chain-RTC-step])
applyclarsimp
apply (clarsimp simp: terminates star-eq-star-n)
apply (frule (1) While-body-add3step)
apply (drule (1) star-n-trans, fastforce)
apply (rule chain-lSup-lprefix[OF lprefix-chain-RTC-step])
applyclarsimp
apply (clarsimp simp: terminates star-eq-star-n)
apply (rename-tac n na)
apply (case-tac n + 3 < na)
apply (rule chain-lprefix-lSup[OF lprefix-chain-NRC-step-ex])
apply (fastforce dest!: inf-loop-While-steps)
apply (clarsimp simp: not-less)
apply (drule (1) While-body-add3step)
apply (rule lprefix-trans[of - llist-of b])
applyclarsimp
apply (case-tac na = n + 3; simp?)
apply (drule (1) NRC-step-deterministic)
applyclarsimp
apply (drule le-neq-trans,clarsimp, simp)
apply (frule-tac n=n+3 and n'=na in star-n-step-decompose; simp?)
apply (drule-tac k=n+3-na in NRC-step-output-mono)
apply (clarsimp simp: output-of-def)
apply (rule chain-lprefix-lSup[OF lprefix-chain-NRC-step-ex])
apply fastforce
done

lemma diverges-While:
  diverges s (While g c) l ↔ diverges s (If g (Seq c (While g c)) Skip) l
apply (rule iffI;clarsimp simp: diverges-If diverges-Seq)
defer
apply (case-tac guard g s; simp)

```

```

apply (clarsimp simp: diverges-Seq)
apply (erule disjE;clarsimp?)
  apply (clarsimp simp: diverges.simps terminates-While terminates-If terminates-Seq)
    apply (fastforce simp: div-body-While-lSup-eq)
    apply (simp (no-asm) add: diverges.simps terminates-While terminates-If)
    applyclarsimp
    apply (simp only: terminates-Seq del: )
    apply (rule conjI)
    apply (clarsimp simp: terminates-deterministic,clarsimp simp: diverges.simps)
    apply (clarsimp simp: diverges.simps)
    apply (fastforce simp: inf-loop-While-lSup-eq)
    apply (case-tac guard g s; simp)
    apply (subgoal-tac diverges s (While g c) l ==> guard g s
           ==> (∀ a b. ¬ terminates s (Seq c (While g c)) (a, b)))
    prefer 2
    apply (clarsimp simp: diverges.simps terminates-While terminates-If)
    apply (clarsimp simp: terminates-Seq diverges.simps)
    apply (rule context-conjI)
    prefer 2
    apply (fastforce simp: div-body-While-lSup-eq)
    applyclarsimp
    apply (rename-tac a b)
    apply (simp add: forall-swap4)
    apply (drule-tac x=a and y=b in spec2)
    applyclarsimp
    apply (rotate-tac 2)
    apply (drule-tac x=a and y=b in spec2)
    applyclarsimp
    apply (erule notE)
    apply (fastforce simp: inf-loop-While-lSup-eq)
    apply (clarsimp simp: diverges.simps terminates-While terminates-If)
    apply (clarsimp simp: terminates.simps)
  by (metis star.refl surj-pair)

```

lemma NRC-terminates:

```

assumes star-n (λx y. terminates x c y) i s t
shows ∀ t1. star-n (λx y. terminates x c y) i s t1 ↔ (t = t1)

```

proof -

```

have ⋀ a b. star-n (λx. terminates x c) i s (a, b) ==> t = (a, b) using assms

```

proof(induct i arbitrary: s t)

case 0

show ?case using 0

by cases (erule star-n.cases; simp)

next

case (Suc i)

show ?case using Suc.prems

apply cases

apply(erule star-n.cases; simp)

```

    by(blast dest: terminates-deterministic Suc.hyps)
qed
thus ?thesis using assms by fastforce
qed

lemma step-output-append:
step (c, a, b) (c', a', b') ==> ∃ new. b' = b @ new
  by (induct c arbitrary: c' a a' b; fastforce simp: subst-def print-def dest!: seqE
whileE)

lemma step-output-extend':
step (c, a, b) (c', a', b @ new) ==> ∀ xs. step (c, a, xs) (c', a', xs @ new)
apply (induct c arbitrary: c' a a' b)
  apply clar simp
  apply clar simp
  apply (rename-tac x1 x2 a a' b xs)
  apply (cases new; simp)
  apply (subgoal-tac (a', xs) = subst x1 x2 (a, xs))
    apply clar simp
    apply (clar simp simp: subst-def)
    apply (clar simp simp: subst-def)
  apply clar simp
  apply (rename-tac x a a' b xs)
  apply (subgoal-tac (a', xs @ new) = print x (a, xs))
    apply clar simp
    apply (clar simp simp: print-def)
  apply (erule seqE)
  apply fastforce
  apply fastforce
  apply (erule ifE)
  apply (erule Pair-inject)
  apply (erule Pair-inject)
  apply (simp only: append-self-conv append.right-neutral)
  apply (rule allI)
  apply (rename-tac x1 c1 c2 c' a a' b xs)
  apply (subgoal-tac guard x1 (a, b) = guard x1 (a, xs))
  prefer 2
  apply (simp add: guard-def)
  apply force
  apply (erule whileE)
  apply (erule Pair-inject)
  apply (erule Pair-inject)
  apply (simp only: append-self-conv append.right-neutral)
  apply (rule allI)
  apply (rename-tac x1 c c' a a' b xs)
  apply (subgoal-tac guard x1 (a, b) = guard x1 (a, xs))
  prefer 2
  apply (simp add: guard-def)
  apply force

```

done

```
lemma step-output-extend:
  step (c, a, b) (c', a', b')  $\implies \exists new. b' = b @ new \wedge (\forall xs. step (c, a, xs) (c', a', xs @ new))$ 
  by (metis step-output-append step-output-extend')

lemma star-n-real-step-output-extend:
  star-n real-step n (c, s) (Skip, t)  $\implies$ 
     $\exists new. snd t = snd s @ new \wedge (\forall xs. \exists n. star-n real-step n (c, fst s, xs) (Skip, fst t, xs @ new))$ 
  apply (induct n arbitrary: c s t)
  apply (erule star-n.cases; fastforce)
  apply clarsimp
  apply (erule star-n.cases; simp)
  apply clarsimp
  apply (rename-tac c a b c' a0 b0 n a' b')
  apply (drule-tac x=c' in meta-spec)
  apply (drule-tac x=a0 and y=b0 in meta-spec2)
  apply (drule-tac x=a' and y=b' in meta-spec2)
  apply clarsimp
  apply (subgoal-tac c ≠ Skip ∧ step (c, a, b) (c', a0, b0))
  prefer 2
  apply (clarsimp simp: real-step-def)
  apply clarsimp
  apply (drule step-output-extend)
  apply clarsimp
  apply (rotate-tac -1)
  apply (rename-tac new newa xs)
  apply (drule-tac x=xs in spec)
  apply (drule-tac x=xs@newa in spec)
  apply clarsimp
  apply (thin-tac real-step - -)
  apply (subgoal-tac real-step (c, a, xs) (c', a0, xs @ newa))
  prefer 2
  apply (clarsimp simp: real-step-def)
  apply (rename-tac na)
  apply (rule-tac x=Suc na in exI)
  apply fastforce
  done

lemma terminates-history:
  terminates s c t  $\implies$ 
     $\exists new. snd t = snd s @ new \wedge (\forall xs. terminates (fst s, xs) c (fst t, xs @ new))$ 
  apply (clarsimp simp: terminates star-eq-star-n)
  using star-n-real-step-output-extend by auto
```

```

lemma terminates-ignores-history:
  terminates (s, out1) c (t, out2)  $\implies$ 
  terminates (s, []) c (t, drop (length out1) out2)
  by (metis append.simps(1) append-eq-conv-conj prod.sel(1) prod.sel(2) terminates-history)

end

```

5 Soundness of the standard Hoare logic

```

theory StdLogicSoundness imports StdLogic WhileLangLemmas begin

theorem Hoare-soundness:
  hoare P c Q  $\implies$  hoare-sem P c Q
proof (induct P c Q rule: hoare.induct)
  case (h-while P x p R)
  then show ?case
    apply (clarsimp simp only: hoare-sem-def)
    apply (rename-tac a b)
    apply (erule-tac P=P (a, b) in rev-mp)
    apply (erule wfp-induct)
    apply clarsimp
    apply (simp (no-asm) add: terminates-While)
    apply (simp (no-asm) add: terminates-If terminates-Seq terminates-Skip)
    by blast
  next
    case (h-weaken P P' p Q' Q)
    then show ?case
      apply (clarsimp simp: hoare-sem-def)
      apply (rename-tac a b)
      apply (drule-tac x=a and y=b in meta-spec2)
      by fastforce
  qed (fastforce simp: hoare-sem-def terminates-Skip terminates-Assign
        terminates-Print terminates-Seq terminates-If)+

end

```

```

theory CoinductiveLemmas imports Coinductive.Coinductive-List begin

```

```

lemma lSup-lappend:
   $\llbracket \text{Complete-Partial-Order}.chain\ lprefix\ A; A \neq \{\} \rrbracket$ 
   $\implies lSup\ (lappend\ xs\ `A) = lappend\ xs\ (lSup\ A)$ 
  apply (induct xs;clarsimp)
  defer
  apply (subst image-image[symmetric, where f= $\lambda x. LCons - x$ ])
  apply (clarsimp simp: lSup-LCons)
  apply (clarsimp simp: ccpo.admissible-def)
  apply (rename-tac B)
  apply (case-tac lfinite (lSup B))
  defer

```

```

apply (rule lprefix-antisym)
apply (rule chain-lSup-lprefix)
apply (rule chainI)
apply (fastforce simp: lappend-inf)
apply clarsimp
apply (rule chain-lprefix-lSup)
apply (rule chainI)
apply (fastforce simp: lappend-inf)
apply (fastforce simp: lappend-inf)
apply (erule lfinite-rev-induct)
apply clarsimp
by (metis mcont-contD mcont-lappend2)

lemma lSup-lmap:
  [Complete-Partial-Order.chain lprefix A; A ≠ {}]
  ==> lSup ((lmap f) ` A) = lmap f (lSup A)
by (metis mcont-contD mcont-lmap)

lemma lSup-lconcat:
  [Complete-Partial-Order.chain lprefix A; A ≠ {}]
  ==> lSup (lconcat ` A) = lconcat (lSup A)
by (metis mcont-contD mcont-lconcat)

lemma cpo-llist-of-up:
  Complete-Partial-Order.chain lprefix {x. ∃ i. x = llist-of [0..i]}
apply (rule chainI)
apply clarsimp
apply (rename-tac i i')
apply (case-tac i ≤ i')
apply (simp (no-asm) add: prefix-def)
apply (rule-tac x=[i..i] in exI)
using le-Suc-ex upt-add-eq-append apply blast
apply (simp add: not-le)
apply (erule noteE)
apply (drule less-imp-le)
apply (simp (no-asm) add: prefix-def)
apply (rule-tac x=[i'..i] in exI)
using le-Suc-ex upt-add-eq-append by blast

lemma iterates-Suc-is-lSup-up:
  iterates Suc 0 = lSup {x. ∃ i. x = llist-of [0..i]}
proof (coinduct rule : llist.coinduct
  [where R=λx y. ltl x = lmap Suc x ∧ ltl y = lmap Suc y ∧ lnull x = lnull y
   ∧ lhd x = lhd y])
case Eq-llist
then show ?case
apply (rule conjI)
apply (simp add: lmap-iterates)
apply (subst lSup-lmap[symmetric, OF cpo-llist-of-up])

```

```

apply fastforce
apply clarsimp
apply (rule conjI)
apply (rule arg-cong[where f=lSup])
apply (rule equalityI; clarsimp simp: subset-iff image-def)
  apply (metis Suc-pred lmap-llist-of map-Suc-up)
apply (rename-tac i)
apply (rule-tac x=llist-of [0..<Suc i] in bexI)
  apply (metis ltl-llist-of map-Suc-up tl-up)
  apply fastforce
apply (rule conjI)
apply fastforce
apply (simp add: lSup.code)
apply (rule conjI; clarsimp)
  apply (metis lnull-llist-of neq0-conv not-less upt-eq-Nil-conv)
  apply (rename-tac i)
apply (rule the-equality; clarsimp simp: image-def)
apply (rule-tac x=llist-of [0..<i] in bexI, clarsimp)
  apply fastforce
done
qed simp

abbreviation (input) flat :: "'a list llist ⇒ 'a llist where
  flat xs ≡ lconcat (lmap llist-of xs)

lemma flat-inf-llist-lSup:
  flat (inf-llist f) = lSup {x. ∃ i. x = llist-of (concat (map f [0..<i]))}
  apply (clarsimp simp: Coinductive-List.inf-llist-def)
  apply (subst iterates-Suc-is-lSup-up)
  apply clarsimp
  apply (subst lSup-lmap[symmetric, OF cpo-llist-of-up])
  apply fastforce
  apply (clarsimp simp: image-def)
  apply (subst lSup-lmap[symmetric])
    apply (rule chainI)
    apply clarsimp
    apply (metis (mono-tags, lifting) chain-def cpo-llist-of-up lprefix-llist-of map-mono-prefix
          mem-Collect-eq)
  apply fastforce
  apply (clarsimp simp add: lconcat-llist-of[symmetric])
  apply (subst lmap-llist-of[symmetric])
  apply (subst llist.map-comp[symmetric])
  apply (subst lSup-lconcat[symmetric])
    apply (rule chainI)
    apply clarsimp
    using cpo-llist-of-up
    apply (metis (no-types, opaque-lifting) le-Suc-ex less-imp-add-positive map-mono-prefix
          not-less prefix-def upt-add-eq-append zero-le)
  apply fastforce

```

```

apply (rule arg-cong[of _ - lSup])
apply (fastforce simp: image-def)
done

lemma upper-subset-lSup-eq:
  [Complete-Partial-Order.chain lprefix B; A ⊆ B;
   ∀ x∈B. ∃ y∈A. lprefix x y] ⇒ lSup B = lSup A
apply (rule lprefix-antisym)
apply (rule chain-lSup-lprefix, simp)
apply (rename-tac xs)
apply (drule-tac x=xs in bspec, simp)
apply (meson chain-subset less-imp-le llist.lub-upper lprefix-trans)
apply (rule chain-lSup-lprefix)
using chain-subset apply blast
using llist.lub-upper by blast

lemma lmap-iterates-id:
  lmap (λz. x) (iterates Suc 0) = iterates id x
apply coinduction
apply (simp add: lmap-iterates[symmetric] llist.map-comp)
done

end

```

6 A Hoare logic for diverging programs

```

theory DivLogic
  imports WhileLang StdLogic CoinductiveLemmas
begin

definition ignores-output :: (val ⇒ state ⇒ bool) ⇒ bool where
  ignores-output H ≡ ∀ i s out1 out2. H i (s,out1) = H i (s,out2)

inductive pohjola where
  p-seq-d: pohjola P p (D::val llist ⇒ bool) ⇒ pohjola P (Seq p q) D
  | p-seq-h: hoare P p M ⇒ pohjola M q D ⇒ pohjola P (Seq p q) D
  | p-if: pohjola (λs. P s ∧ guard x s) p D ⇒
    pohjola (λs. P s ∧ ~guard x s) q D ⇒ pohjola P (If x p q) D
  |
  p-while1: (∀s. P s ⇒
    (exists H ev.
      guard x s ∧ H 0 s ∧ ignores-output H ∧
      D (flat (LCons (output-of s) (inf-list ev))) ∧
      (∀i. hoare (λs. H i s ∧ output-of s = []) p
        (λs. H (i+1) s ∧ output-of s = ev i ∧ guard x s))) ⇒
    pohjola P (While x p) D
  |
  p-while2: (∀s. P s → guard x s) ⇒ wfP R ⇒
    (∀s0. hoare (λs. P s ∧ b s ∧ s = s0) p (λs. P s ∧ R s s0)) ⇒

```

```

pohjola ( $\lambda s. P s \wedge \sim b s$ ) p D  $\implies$  pohjola P (While x p) D
| p-const: pohjola ( $\lambda s. \text{False}$ ) p D
| p-case: pohjola ( $\lambda s. P s \wedge b s$ ) p D  $\implies$  pohjola ( $\lambda s. P s \wedge \sim b s$ ) p D  $\implies$  pohjola P p D
| p-weaken: ( $\forall s. P s \longrightarrow P' s$ )  $\implies$  pohjola P' p D'  $\implies$  ( $\forall s. D' s \longrightarrow D s$ )  $\implies$  pohjola P p D
print-theorems

```

```

definition pohjola-sem where
  pohjola-sem P p D  $\equiv$ 
     $\forall s. P s \longrightarrow (\exists l. \text{diverges } s p l \wedge D l)$ 
end

```

7 Completeness of the standard Hoare logic

```
theory StdLogicCompleteness imports StdLogic WhileLangLemmas begin
```

lemma Hoare-strengthen:

```
( $\bigwedge s. P s \implies P' s$ )  $\implies$  hoare P p Q  $\implies$  hoare P p Q using h-weaken by blast
```

lemma Hoare-strengthen-post:

```
( $\bigwedge s. Q' s \implies Q s$ )  $\implies$  hoare P p Q'  $\implies$  hoare P p Q using h-weaken by blast
```

theorem Hoare-While:

```
assumes h1: ( $\bigwedge s. P s \implies R s$ )
assumes h2: ( $\bigwedge s. R s \wedge \neg \text{guard } x s \implies Q s$ )
assumes h3:  $\bigwedge s0. \text{hoare } (\lambda s. R s \wedge \text{guard } x s \wedge s = s0) p (\lambda s. R s \wedge m s < ((m s0)::nat))$ 
```

```
shows hoare P (While x p) Q
```

```
apply (rule-tac P'=R and Q'= $\lambda s. R s \wedge \neg \text{guard } x s$  in h-weaken; (simp add: h1 h2) ?)
apply (rule h-while[OF h3])
apply (clar simp simp: wfp-def)
using wf-measure[where f=m, simplified measure-def inv-image-def]
by auto
```

lemma NRC-lemma:

```
star-n ( $\lambda s t. \text{guard } f s \wedge \text{terminates } s c t$ ) k0 m t0  $\implies$ 
```

```
star-n ( $\lambda s t. \text{guard } f s \wedge \text{terminates } s c t$ ) k1 m t1  $\implies$ 
```

```
 $\neg \text{guard } f t0 \wedge \neg \text{guard } f t1 \implies$ 
```

```
t0 = t1  $\wedge$  k0 = k1
```

```
apply (induct k0 arbitrary: k1 m; clar simp)
```

```
apply (case-tac k1; clar simp)
```

```
apply (erule star-n.cases)+
```

```
apply clar simp+
```

```
apply (erule star-n.cases)+
```

```

apply clarsimp+
apply (case-tac k1;clarsimp)
apply (erule star-n.cases)+
  applyclarsimp+
  apply (erule star-n.cases)+
    applyclarsimp+
    apply (rename-tac k1)
  apply (drule-tac x=k1 in meta-spec)
  apply (erule star-n.cases)
  applyclarsimp+
  apply (erule star-n.cases)
  applyclarsimp+
  apply (drule (1) terminates-deterministic)
  applyclarsimp
done

lemma Hoare-terminates:
hoare ( $\lambda s. \exists t. \text{terminates } s c t \wedge Q t$ ) c Q
proof (induct c arbitrary: Q)
  case (Seq c1 c2)
  then show ?case
    apply (clarsimp simp: terminates-Seq)
    apply (rule-tac M= $\lambda s. \exists s'. \text{terminates } s c2 s' \wedge Q s'$  in h-seq)
    apply (drule-tac x= $\lambda s. \exists s'. \text{terminates } s c2 s' \wedge Q s'$  in meta-spec)
    applyclarsimp
    apply (rule Hoare-strengthen[rotated], simp, fastforce)
    by fastforce
  next
  case (If x1 c1 c2)
  then show ?case
    apply (clarsimp simp: terminates-If)
    apply (rule h-if)
    by (rule Hoare-strengthen[rotated], fastforce+)+
  next
  case (While x1 c)
  then show ?case
    apply (rule-tac m= $\lambda s. \text{THE } n. \exists t. (\text{star-n } (\lambda s t. \text{guard } x1 s \wedge \text{terminates } s c t) n s t)$ 
       $\wedge \neg \text{guard } x1 t$ 
      in Hoare-While)
    apply assumption
    apply (clarsimp simp: terminates-While terminates-If terminates-Skip)
    apply (rename-tac s0)
    apply (rule-tac Q'= $\lambda s. \text{terminates } s0 c s \wedge (\exists a b. \text{terminates } s (\text{While } x1 c) (a, b))$ 
       $\wedge \text{guard } x1 s0 \wedge Q (a, b)$ 
      in Hoare-strengthen-post[rotated]))
    apply (drule-tac x= $\lambda s. \text{terminates } s0 c s \wedge (\exists a b. \text{terminates } s (\text{While } x1 c) (a, b))$ 
       $\wedge \text{guard } x1 s0 \wedge Q (a, b)$ 
      in Hoare-strengthen-post[rotated]))
  
```

```

 $\wedge \text{guard } x1 s0 \wedge Q(a, b)$ 
in meta-spec)
apply clarsimp
apply (rule Hoare-strengthen[rotated])
apply assumption
apply clarsimp
apply (subst (asm) (3) terminates-While)
apply (clarsimp simp: terminates-If terminates-Seq)
apply fastforce
apply (rule conjI)
apply fastforce
applyclarsimp
apply (drule-tac p=While x1 c and f=x1 and c=c in terminates-While-NRC)
applyclarsimp
apply (erule exE)
apply (subst the-equality)
apply (rename-tac ab bb n)
apply (rule-tac x=ab in exI, rule-tac x=bb in exI)
apply (fastforce dest: NRC-lemma)+
applyclarsimp
apply (drule step-n[rotated], simp)
apply (subst the-equality)
apply (fastforce dest: NRC-lemma)+
done
qed (clarsimp simp: terminates-Skip terminates-Assign terminates-Print)+
```

theorem Hoare-completeness:

```

hoare-sem P c Q  $\implies$  hoare P c Q
apply (unfold hoare-sem-def)
using h-weaken[OF - Hoare-terminates] by blast
```

```

lemma hoare-pre-False:
hoare ( $\lambda\_. \text{False}$ ) prog Q
apply (rule Hoare-completeness)
apply (simp add: hoare-sem-def)
done
```

end

8 Completeness of Hoare logic for diverging programs

```

theory DivLogicCompleteness
imports DivLogic StdLogicCompleteness StdLogicSoundness
begin
```

```

declare pohjola.intros[intro, simp]
declare pohjola.intros(7)[simp del]
declare pohjola.intros(7)[rule del]
declare pohjola.intros(6)[rule del]

theorem pohjola-strengthen:
   $\llbracket pohjola P' p D; \forall s. P s \rightarrow P' s \rrbracket \implies pohjola P p D$ 
  by blast

inductive div-at-iteration where
  guard f s  $\implies$  diverges s c l  $\implies$  D l  $\implies$  div-at-iteration 0 s f c D
  | guard f s  $\implies$  terminates s c t  $\implies$  div-at-iteration n t f c D  $\implies$ 
    div-at-iteration (Suc n) s f c D
print-theorems

inductive-cases
  div-at-0 [elim!]: div-at-iteration 0 s f c D and
  div-at-S [elim!]: div-at-iteration (Suc n) s f c D
print-theorems

theorem div-at-iteration-11:
  div-at-iteration i s f c D  $\implies$ 
    div-at-iteration j s f c D  $\implies$  i = j
  apply (induct arbitrary: j rule: div-at-iteration.induct)
  apply (erule div-at-iteration.cases; simp)
  apply (erule diverges.cases; clarsimp)
  apply (rotate-tac -1)
  apply (erule div-at-iteration.cases)
  apply (erule diverges.cases, blast)
  apply safe
  apply (drule (1) terminates-deterministic)
  apply clarsimp
  done

lemma star-n-While-flatten:
  star-n (λs t. star step (While x p, s) (While x p, t)
     $\wedge$  terminates s p t  $\wedge$  guard x s) i s t
   $\implies \exists n'. star-n step n' (While x p, s) (While x p, t) \wedge n' \geq i$ 
  apply (induct arbitrary: s t rule: nat-less-induct)
  apply (rename-tac n s t)
  apply (clarsimp simp: star-eq-star-n)
  apply (case-tac n; simp)
  applyclarsimp
  apply (erule star-n.cases;clarsimp)
  apply fastforce
  applyclarsimp
  apply (erule star-n.cases;clarsimp)
  apply (rename-tac ac bc n ad bd na)
  apply (drule-tac x=n in spec)

```

```

apply clarsimp
apply (drule-tac x=ac and y=bc in spec2)
apply (drule-tac x=ad and y=bd in spec2)
apply simp
apply clarsimp
apply (clarsimp simp: terminates star-eq-star-n)
apply (rename-tac n' nb)
apply (drule (1) While-body-add3step)
apply (thin-tac star-n - na - -)
apply (frule (1) star-n-trans)
apply (rule-tac x=nb + 3 + n' in exI)
apply fastforce
done

lemma diverges-init-state:
  [termination s c t; diverges t (While g c) l; guard g s]  $\implies$  diverges s (While g c) l
  apply (simp (no-asm) add: diverges-While diverges-If)
  apply clarsimp
  apply (simp (no-asm) add: diverges-Seq)
  apply (rule disjI2)
  apply (rule-tac x=fst t in exI, rule-tac x=snd t in exI)
  by fastforce

lemma diverges-init-state-n:
  [star-n ( $\lambda s t.$  terminates s c t  $\wedge$  guard g s) n s t; diverges t (While g c) l; guard g t]
     $\implies$  diverges s (While g c) l
  apply (induct n arbitrary: s t)
  apply (erule star-n.cases,clarsimp)
  apply (clarsimp simp: terminates)
  apply (erule star-n.cases;clarsimp)
  apply (rename-tac a b a' b' n a'' b'')
  apply (drule-tac x=a' and y=b' in meta-spec2)
  apply (drule-tac x=a'' and y=b'' in meta-spec2)
  applyclarsimp
  apply (frule diverges-init-state; simp?)
  done

lemma div-at-i-unwind:
  div-at-iteration i s g c D
   $\longleftrightarrow$  ( $\exists t.$  star-n ( $\lambda s t.$  terminates s c t  $\wedge$  guard g s) i s t  $\wedge$  guard g t
     $\wedge$  ( $\exists l.$  diverges t c l  $\wedge$  D l))
  apply (rule iffI)
  apply (induct i arbitrary: s)
  apply fastforce
  applyclarsimp
  apply (rename-tac a' b')
  apply (drule-tac x=a' and y=b' in meta-spec2)
  apply fastforce

```

```

apply clarsimp
apply (induct i arbitrary: s)
apply (erule star-n.cases; simp)
apply (rule div-at-iteration.intros(1); simp)
apply (erule star-n.cases; simp)
apply clarsimp
apply (rename-tac l a b a' b' n a'' b'')
apply (rule div-at-iteration.intros(2); simp?)
apply (drule-tac x=a'' and y=b'' in meta-spec2)
apply (drule-tac x=l in meta-spec)
apply (drule-tac x=a' and y=b' in meta-spec2)
apply clarsimp
done

lemma diverging-body-diverges:
 $\llbracket \text{diverges } s \text{ c } l; \text{guard } g \text{ s} \rrbracket \implies \text{diverges } s (\text{While } g \text{ c}) \text{ l}$ 
apply (simp add: diverges-While diverges-If)
apply (simp add: diverges-Seq)
done

lemma non-diverging-inf-loop:
 $\llbracket \forall i. \neg \text{div-at-iteration } i \text{ s } g \text{ c } D; \text{diverges } s (\text{While } g \text{ c}) \text{ l}; D \text{ l} \rrbracket$ 
 $\implies \forall i. \exists t. \text{star-n} (\lambda s t. (\exists k. \text{star-n step } k (\text{While } g \text{ c}, s) (\text{While } g \text{ c}, t))$ 
 $\quad \wedge \text{terminates } s \text{ c } t \wedge \text{guard } g \text{ s}) \text{ i } s \text{ t}$ 
apply clarsimp
apply (simp add: diverges-While diverges-If)
apply (case-tac guard g s; simp)
apply (simp add: diverges-Seq)
apply (erule disjE)
apply (fastforce intro!: div-at-iteration.intros)
apply clarsimp
apply (rule nat.induct)
apply (cases s; blast)
apply clarsimp
apply (rename-tac a b n a' b')
apply (subgoal-tac guard g (a', b'))
prefer 2
apply (rule ccontr)
apply (frule star-n-While-flatten[simplified star-eq-star-n])
apply clarsimp
apply (subgoal-tac star-n step (Suc (Suc 0)) (While g c, a', b'))
 $\quad (\text{if guard } g \text{ (a', b')} \text{ then Seq } c (\text{While } g \text{ c}) \text{ else Skip, a', b'})$ 
prefer 2
apply fastforce
apply (drule (1) star-n-trans)
apply (drule (2) diverges-init-state)
apply (clarsimp simp: diverges.simps terminates.simps star-eq-star-n)
apply (insert terminates-or-diverges)
apply (drule-tac x=(a', b') and y=c in meta-spec2)

```

```

apply (erule disjE)
apply clarsimp
apply (rename-tac aa ba)
apply (rule-tac x=aa in exI)
apply (rule-tac x=ba in exI)
apply (rule step-n-rev, simp)
apply clarsimp
apply (clarsimp simp: terminates star-eq-star-n)
apply (rename-tac nb)
apply (drule-tac n=nb in While-body-add3step, simp, fastforce)

apply clarsimp
apply (frule (1) diverging-body-diverges)
apply (frule star-n-conjunct2)
apply (drule (2) diverges-init-state-n)
apply (drule (2) diverges-init-state)
apply (drule (1) diverges-deterministic, simp)
apply (simp add: div-at-i-unwind)
apply (drule star-n-conjunct2)
by blast

lemma While-lemma:

$$\begin{aligned} & \llbracket \forall i. \neg \text{div-at-iteration } i s g c D; \text{diverges } s (\text{While } g c) l; D l \rrbracket \\ & \implies \exists ts. ts 0 = s \wedge (\forall i. \text{guard } g (ts i) \wedge \text{terminates } (ts i) c (ts (\text{Suc } i)) \\ & \quad \wedge (\exists k. \text{star-n step } (i+k) (\text{While } g c, ts 0) (\text{While } g c, ts i))) \end{aligned}$$

apply (rule-tac x=λi. SOME t. star-n (λs t. terminates s c t) i s t in exI)
apply (rule context-conjI)
apply (rule some-equality; clarsimp?)
apply (erule star-n.cases; simp)
apply clarsimp
apply (rename-tac i)
apply (drule (2) non-diverging-inf-loop)
apply (rule someI2-ex)
apply (drule-tac x=i in spec, clarsimp)
apply (drule star-n-conjunct2[THEN star-n-conjunct1])
apply fastforce
apply (frule-tac x=i in spec)
apply (elim exE conjE)
apply (frule star-n-conjunct2[THEN star-n-conjunct1])
apply (rename-tac t)
apply (drule NRC-terminates, drule-tac x=t in spec, simp)
apply (drule-tac x=Suc i in spec)
apply (elim exE conjE)
apply (rename-tac a b)
apply (frule-tac t=(a, b) in star-n-conjunct2[THEN star-n-conjunct1])
apply (erule star-n-lastE, clarsimp)
apply (rename-tac ad bd ae be k)
apply (frule-tac t=(ad, bd) in star-n-conjunct2[THEN star-n-conjunct1])

```

```

apply (drule NRC-terminates, drule-tac x=(ad, bd) in spec, clarsimp)
apply (rename-tac b n a' b' a a'' b'' k)
apply (rule conjI)
apply (rule someI2-ex, fastforce)
apply (fastforce simp: NRC-terminates)
apply (simp add: star-eq-star-n[symmetric])
apply (drule star-n-While-flatten, clarsimp)
apply (rule-tac x=n' - n in exI, clarsimp)
done

lemma H-for-Nil-output:
H i (a, b) ==> ignores-output H ==> H i (a, [])
using ignores-output-def by blast

lemma output-of-simp[simp]:
output-of (a, b) = b by (fastforce simp: output-of-def)

lemma star-n-step-output-extend:
star-n step n (c, s) (c', t) ==>
  ∃ new. snd t = snd s @ new ∧
    (∀ xs. star-n step n (c, fst s, xs)
      (c', fst t, xs @ new))
apply (induct n arbitrary: c s t c')
apply (erule star-n.cases; fastforce)
apply clarsimp
apply (erule star-n.cases; simp)
apply clarsimp
apply (rename-tac c a b c' a' b' n c'' a'' b'')
apply (drule-tac x=c' in meta-spec)
apply (drule-tac x=a' and y=b' in meta-spec2)
apply (drule-tac x=a'' and y=b'' in meta-spec2)
apply (drule-tac x=c'' in meta-spec)
apply clarsimp
apply (drule step-output-extend)
apply clarsimp
apply (rotate-tac -1)
apply (rename-tac newa xs)
apply (drule-tac x=xs in spec)
apply (drule-tac x=xs@newa in spec)
apply clarsimp
done

lemma lappend-initial-output:
{llist-of out | out.
  ∃ q t. star step (While x p, a, b) (q, t, out)} =
lappend (llist-of b) ` {llist-of out | out.
  ∃ q t. star step (While x p, a, []) (q, t, out)}

```

```

apply (rule equalityI; clarsimp simp: star-eq-star-n)
apply (drule star-n-step-output-extend)
applyclarsimp
apply (drule-tac x=Nil in spec)
applyclarsimp
apply (metis (mono-tags, lifting) lappend-llist-of-llist-of mem-Collect-eq rev-image-eqI)
apply (drule star-n-step-output-extend)
applyclarsimp
apply (drule-tac x=b in spec)
apply (rename-tac out q t n)
apply (rule-tac x=b@out in exI)
using lappend-llist-of-llist-of by blast

```

lemma ts-accum:

```

forall i. prefix (output-of (ts i)) (output-of (ts (Suc i))) ==>
concat (map ( $\lambda$ i. drop (length (output-of (ts i))) (output-of (ts (Suc i)))) [0.. $<$ i])
= drop (length (output-of (ts 0))) (output-of (ts i))
apply (induct i)
applyclarsimp
applyclarsimp
apply (rule injD[where f= $\lambda$ x. output-of (ts 0) @ x])
apply (metis append-eq-append-conv injI)
apply (frule min-prefix)
apply (rotate-tac -1)
apply (rename-tac i)
apply (frule-tac x=i in spec)
apply (drule prefix-drop-append)
apply (drule-tac x=Suc i in spec)
apply (drule prefix-drop-append)
apply (simp only: append-assoc[symmetric])
apply (frule-tac x=i in spec)
apply (drule prefix-drop-append)
applyclarsimp
done

```

theorem Pohjola-diverges:

```

pohjola ( $\lambda$ s.  $\exists$ l. diverges s c l  $\wedge$  D l) c D
proof (induct c)
case (Seq c1 c2)
then show ?case
apply (clarsimp simp: diverges-Seq)
apply (rule-tac b= $\lambda$ s.  $\exists$ l. diverges s c1 l  $\wedge$  D l in p-case)
apply (rule pohjola-strengthen[where P= $\lambda$ s. P s  $\wedge$  Q s and P'=Q for P Q];
clarsimp)
apply (rule-tac P'= $\lambda$ s.  $\exists$ l. ( $\exists$ t. terminates s c1 t  $\wedge$  diverges t c2 l)  $\wedge$  D l
in pohjola-strengthen[rotated])
apply fastforce

```

```

apply (rule-tac  $M=\lambda s. \exists l. \text{diverges } s c2 l \wedge D l \text{ in } p\text{-seq-}h; \text{clarsimp}$ )
by (rule-tac  $Q'=\lambda s. \exists l. \text{diverges } s c2 l \wedge D l \text{ in } h\text{-weaken[OF - Hoare-terminates];}$ 
fastforce)
next
case (If  $x1 c1 c2$ )
then show ?case
apply (clarsimp simp: diverges-If)
apply (rule p-if)
apply (erule pohjola-strengthen, clarsimp)
by (erule pohjola-strengthen, clarsimp)
next
case (While  $x1 c$ )
then show ?case
apply (rule-tac  $b=\lambda s. \exists i. \text{div-at-iteration } i s x1 c D \text{ in } p\text{-case}$ )
apply (rule-tac  $P'=\lambda s. \exists i. \text{div-at-iteration } i s x1 c D \text{ in } \text{pohjola-strengthen[rotated]}$ )
apply clarsimp
apply (rule-tac  $R=\text{wf-to-wfP}$  (measure ( $\lambda s. \text{SOME } i. \text{div-at-iteration } i s x1 c$ 
 $D$ )))
and  $b=\lambda s. \neg \text{div-at-iteration } 0 s x1 c D$ 
in p-while2)
apply clarsimp
apply (erule div-at-iteration.cases; clarsimp)
apply (simp only: wf-to-wfP-def wfp-def)
using wf-measure
apply (metis (no-types, lifting) case-prodE mem-Collect-eq subsetI wf-subset)
apply clarsimp
apply (rule Hoare-completeness)
apply (clarsimp simp: hoare-sem-def wf-to-wfP-def)
apply (case-tac  $i$ ; clarsimp)
apply (rename-tac  $a b n a' b'$ )
apply (rule-tac  $x=a' \text{ in } exI$ , rule-tac  $x=b' \text{ in } exI$ , clarsimp)
apply (frule (2) div-at-iteration.intros(2))
apply (rule conjI, fastforce)
apply (rule someI2-ex, fastforce)
apply (drule (1) div-at-iteration-11)
apply (rule someI2, simp)
apply (drule (1) div-at-iteration-11, clarsimp)
apply clarsimp
apply (rule pohjola-strengthen, simp)
apply fastforce
apply (rule p-while1, clarsimp)
apply (drule (2) While-lemma)
apply clarsimp
apply (rule context-conjI)
apply (drule-tac  $x=0 \text{ in } \text{spec}$ , clarsimp)
apply (rename-tac  $ts$ )
apply (rule-tac  $x=\lambda i s. \text{fst } s = \text{fst } (ts i) \text{ in } exI$ )
apply clarsimp
apply (rule context-conjI)

```

```

apply (clarsimp simp: ignores-output-def)
apply (rule-tac x=λi. drop (length (output-of (ts i))) (output-of (ts (Suc i)))
in exI)
apply (rule conjI;clarsimp?)
defer
apply (rule Hoare-completeness)
apply (clarsimp simp: hoare-sem-def)
apply (rename-tac i)
apply (frule-tac x=i in spec)
apply (drule-tac x=Suc i in spec,clarsimp)
apply (intro conjI; (clarsimp simp: output-of-def guard-def; fail) ?)
apply (drule terminates-history,clarsimp)
apply (drule-tac x=Nil in spec)
apply (clarsimp simp: split-def output-of-def)
apply (thin-tac pohjola ---)
apply (clarsimp simp: diverges.simps)
apply (erule back-subst[of D])
apply (simp only: lappend-initial-output[THEN arg-cong[where f=lSup]])
apply (subst lSup-lappend)
apply (rule chainI)
applyclarsimp
apply (meson lprefix-chain-NRC-step' lprefix-list-of star-eq-star-n)
apply blast
apply (rename-tac a b ts)
apply (rule-tac f=lappend (list-of b) in arg-cong)
apply (subst lmap-inf-list[symmetric])
apply (simp only: flat-inf-list-lSup)
apply (subst ts-accum)
applyclarsimp
apply (rename-tac i)
apply (drule-tac x=i in spec)
apply safe
apply (meson NRC-step-output-mono star-star-n terminates.cases)
apply (rename-tac a b ts)
apply (rule upper-subset-lSup-eq[OF lprefix-chain-RTC-step])
applyclarsimp
apply (rename-tac i)
apply (drule-tac x=i in spec)
applyclarsimp
apply (drule-tac star-n-step-output-extend,clarsimp simp: star-eq-star-n)
apply (drule-tac x=Nil in spec)
applyclarsimp
apply (metis (mono-tags, opaque-lifting) append-eq-conv-conj output-of-simp
prod.exhaust-sel)
apply (clarsimp simp: star-eq-star-n)
apply (rename-tac n)
apply (subgoal-tac ∃ i k. star-n step (i + k) (While x1 c, a, b) (While x1 c, ts
i))
    ∧ n < i + k)

```

```

apply clar simp
apply (rename-tac i k)
apply (drule-tac x=i in spec)
apply clar simp
apply (drule-tac n=i + k in star-n-step-output-extend)
apply clar simp
apply (rename-tac new)
apply (drule-tac x=Nil in spec, clar simp)
apply (rule-tac x=llist-of new in exI)
apply clar simp
apply (rule conjI)
  apply (metis append-eq-conv-conj output-of-def snd-def)
apply (metis (no-types, lifting) NRC-step-output-mono output-of-simp star-n-step-decompose)
  by (meson not-less-less-Suc-eq trans-less-add1)
qed clar simp+

```

theorem Pohjola-completeness:

```

pohjola-sem P c D ==> pohjola P c D
apply (clar simp simp: pohjola-sem-def)
using pohjola-strengthen[OF Pohjola-diverges] by simp

```

end

9 Soundness of Hoare logic for diverging programs

theory DivLogicSoundness **imports** StdLogicSoundness DivLogicCompleteness **begin**

```

lemma p-loop-deterministic:
star-n (λs t. guard x s ∧ terminates s p t) n s t ==>
star-n (λs t. guard x s ∧ terminates s p t) n s t' ==> t = t'
proof(induct rule: star-n.induct)
  case (refl-n x) thus ?case by cases simp
next
  case (step-n x y n z)
  show ?case using step-n.prem step-n.hyps
    by cases (force dest: terminates-deterministic)
qed

lemma loop-accum:
  ∀ i. hoare (λs. H i s ∧ output-of s = []) p
    (λs. H (Suc i) s ∧ output-of s = ev i ∧ guard x s);
    guard x (a, b); H 0 (a, b); ignores-output H]
  ==> ∀ i. ∃ s. star-n (λs t. guard x s ∧ terminates s p t) i (a, b) s ∧
    guard x s ∧ H i s
apply (rule allI)
apply (rule nat.induct)
apply clar simp
apply (rule-tac x=a in exI)

```

```

apply (rule-tac  $x=b$  in  $exI$ )
apply clarsimp
apply clarsimp
apply (rename-tac  $n a' b'$ )
apply (drule-tac  $x=n$  in spec)
apply (drule Hoare-soundness)
apply (clarsimp simp: hoare-sem-def)
apply (drule-tac  $x=a'$  in spec)
apply (frule-tac  $i=n$  in  $H$ -for-Nil-output[rotated], simp)
apply clarsimp
apply (drule-tac  $s=(a', \[])$  in terminates-history)
apply clarsimp
apply (drule-tac  $x=b'$  in spec)
apply (drule step-n-rev)
apply fastforce
apply (fastforce simp: guard-def ignores-output-def)
done

lemma output-accum:
   $\llbracket \text{star-}n (\lambda s t. \text{guard } x s \wedge \text{terminates } s p t) i (a, b) s; \right.$ 
   $\text{guard } x (a, b); H 0 (a, b); \text{ignores-output } H;$ 
   $\forall i. \text{hoare } (\lambda s. H i s \wedge \text{output-of } s = \[]) p$ 
   $\quad (\lambda s. H (\text{Suc } i) s \wedge \text{output-of } s = ev i \wedge \text{guard } x s) \rrbracket$ 
   $\implies \text{output-of } s = b @ (\text{concat } (\text{map } ev [0..<i]))$ 
proof(induct i arbitrary: s)
  case 0 thus ?case by casesclarsimp
next
  case (Suc n) thus ?case
    applyclarsimp
    apply (erule star-n-lastE)
    applyclarsimp
    apply (rename-tac  $a' b' a'' b''$ )
    apply (frule (3) loop-accum)
    apply (rotate-tac -1)
    apply (drule-tac  $x=n$  in spec)
    applyclarsimp
    apply (drule (1) p-loop-deterministic,clarsimp)
    apply (rename-tac  $a'$ )
    apply (drule-tac  $x=n$  in spec)
    apply (drule Hoare-soundness)
    apply (clarsimp simp: hoare-sem-def)
    apply (drule-tac  $x=a'$  in spec)
    apply (subgoal-tac  $H n (a', \[])$ )
    prefer 2
    apply (fastforce simp: ignores-output-def)
    applyclarsimp
    apply (drule-tac  $x=a'$  and  $y=b'$  in meta-spec2)
    applyclarsimp
    apply (drule-tac  $s=(a', \[])$  in terminates-history)

```

```

apply clarsimp
apply (drule-tac x=b @ concat (map ev [0..<n]) in spec)
apply clarsimp
apply (drule (1) terminates-deterministic,clarsimp)
done
qed

lemma helper-lemma:
 $\llbracket \text{guard } x (a, b); H 0 (a, b); \text{ignores-output } H; \forall i. \text{hoare } (\lambda s. H i s \wedge \text{output-of } s = []) p \\ (\lambda s. H (\text{Suc } i) s \wedge \text{output-of } s = ev i \wedge \text{guard } x s) \rrbracket \implies \forall i. \exists s. \text{star-}n (\lambda s t. \text{guard } x s \wedge \text{terminates } s p t) i (a, b) s \wedge \\ \text{guard } x s \wedge H i s \wedge \text{output-of } s = b @ (\text{concat } (\text{map } ev [0..< i]))$ 
applyclarsimp
apply (frule (3) loop-accum)
apply (rotate-tac -1)
apply (rename-tac i)
apply (drule-tac x=i in spec)
applyclarsimp
apply (frule (4) output-accum)
apply (rename-tac a' b')
apply (rule-tac x=a' in exI)
apply (clarsimp simp: output-of-def)
done

lemma add-While-loops:
 $\llbracket \text{star-}n (\lambda s t. \text{guard } x s \wedge \text{terminates } s p t) i (a, b) s \rrbracket \implies \text{star-}n (\lambda s t. \text{star step } (\text{While } x p, s) (\text{While } x p, t) \wedge \text{terminates } s p t \wedge \\ \text{guard } x s) i (a, b) s$ 
proof(induct i arbitrary: a b s)
case 0
then show ?case by cases simp
next
case (Suc n a b s)
thus ?case
by(force dest: While-body-add3real-step star-n-real-step-step
      simp: terminates star-eq-star-n
      elim: step-n-rev star-n-lastE)
qed

lemma loop-upper-bound:
 $\llbracket \forall i. \exists s. \text{star-}n (\lambda s t. \text{guard } x s \wedge \text{terminates } s p t) i (a, b) s \wedge \text{guard } x s \wedge H i s; \\ \text{star-}n \text{ step } n (\text{While } x p, a, []) (q, t, out) \rrbracket \implies \\ \exists i' n'. \text{star-}n (\lambda s t. \text{star step } (\text{While } x p, s) (\text{While } x p, t) \\ \wedge \text{terminates } s p t \wedge \text{guard } x s) i (a, b) t' \wedge \\ \text{star-}n \text{ step } n' (\text{While } x p, a, b) (\text{While } x p, t') \wedge n \leq n'$ 
apply (insert add-While-loops[of x p])
apply (subgoal-tac  $\exists i > n. \exists s. \text{star-}n$ 

```

```


$$(\lambda s t. \text{star step} (\text{While } x p, s) (\text{While } x p, t) \wedge
\text{terminates } s p t \wedge \text{guard } x s)
i (a, b) s)$$


prefer 2
apply (meson lessI)
apply safe
apply (rename-tac i a' b')
apply (rule-tac x=i in exI)
apply (drule-tac x=i in spec)+
apply (rule-tac x=(a', b') in exI)
apply clarsimp
apply (drule star-n-While-flatten)
apply fastforce
done

theorem Pohjola-soundness:
\implies pohjola-sem P c Q
unfolding pohjola-sem-def
proof (induct rule: pohjola.induct)
case (p-seq-d P p D q)
then show ?case
using diverges-Seq by blast
next
case (p-seq-h P p M q D)
then show ?case
apply clarsimp
apply (frule Hoare-soundness)
apply (clarsimp simp: hoare-sem-def)
apply (rotate-tac -1)
apply (rename-tac a b)
apply (drule-tac x=a and y=b in spec2)
byclarsimp (meson diverges-Seq)
next
case (p-if P x p D q)
then show ?case
apply clarsimp
apply (rename-tac a b)
apply (drule-tac x=a and y=b in spec2)+
using diverges-If by fastforce
next
case (p-while1 P x D p)
then show ?case
apply clarsimp
apply (rename-tac a b)
apply (drule-tac x=a and y=b in meta-spec2)
apply clarsimp
apply (frule (3) loop-accum)
apply (rename-tac ev)
apply (rule-tac x=flat (LCons (output-of (a, b)) (inf-list ev)) in exI)

```

```

apply (simp (no-asm-simp) add: diverges.simps)
apply (rule context-conjI)
apply clarsimp
apply (drule terminates-While-NRC, simp)
apply clarsimp
apply (rename-tac n)
apply (drule-tac x=n in spec)+
apply clarsimp
apply (drule (1) p-loop-deterministic,clarsimp)
apply (subst lappend-initial-output)
apply (subst lSup-lappend)
apply (rule chainI)
applyclarsimp
apply (meson lprefix-chain-NRC-step' lprefix-llist-of star-eq-star-n)
apply blast
apply (rule-tac f=lappend (llist-of b) in arg-cong)
apply (subst lmap-inf-llist[symmetric])
apply (subst flat-inf-llist-lSup)
apply (rule sym)
apply (rule upper-subset-lSup-eq[OF lprefix-chain-RTC-step])
apply safe
apply (simp (no-asm-simp))
apply (rotate-tac -2)
apply (rename-tac i)
apply (drule-tac x=i in spec)
apply safe
apply (frule (4) output-accum)
applyclarsimp
apply (drule add-While-loops[THEN star-n-While-flatten])
applyclarsimp
apply (drule star-n-step-output-extend,clarsimp)
apply (drule-tac x=Nil in spec,clarsimp simp: star-eq-star-n)
apply fastforce
apply (simp only: star-eq-star-n, erule exE)
apply (rename-tac n)
apply (drule (1) loop-upper-bound, elim exE conjE)
apply (frule star-n-conjunct2[THEN star-n-commute])
apply (frule (4) output-accum)
applyclarsimp
apply (drule-tac n=n in star-n-step-output-extend)
applyclarsimp
apply (drule-tac x=b in spec)
apply (simp add: le-eq-less-or-eq)
apply (erule disjE)
apply (drule (2) star-n-step-decompose)
using NRC-step-output-mono apply fastforce
apply simp
apply (drule (1) NRC-step-deterministic, simp)
apply fastforce

```

```

done
next
  case (p-while2 P x R b p D)
  then show ?case
    apply –
    apply rotate-tac
    apply (intro allI)
    apply (erule wfp-induct)
    apply (intro impI allI)
    apply (rename-tac xa)
    apply (case-tac b xa)
    apply (drule-tac x=xa in spec)
    apply (drule Hoare-soundness)+
    apply (clar simp simp: hoare-sem-def)
    apply (rotate-tac -6)
    apply (rename-tac aa bb)
    apply (drule-tac x=aa and y=bb in spec2)
    apply simp
    apply (elim exE)
    apply (simp (no-asm) add: diverges-While diverges-If)
    apply (simp add: diverges-Seq)
    apply (rename-tac l)
    apply (rule-tac x=l in exI)
    apply (rule conjI; simp?)
    apply fastforce
    apply (simp (no-asm) add: diverges-While diverges-If)
    apply (simp add: diverges-Seq)
    by fastforce
qed fastforce+

end

```

10 Examples

theory *Examples imports DivLogicSoundness*

begin

```

definition pure-loop :: prog where
  pure-loop = While (λ-. 1) Skip

lemma pure-loop-correct:
  pohjola (λs. output-of s = []) pure-loop (λl. l = LNil)
  unfolding pure-loop-def
  apply (rule p-while1)
  apply (rule-tac x=λi s. True in exI)
  apply (simp add: ignores-output-def guard-def)
  apply (rule-tac x=λ-. [] in exI, simp)
done

```

```

definition zero :: prog where
  zero = While (λ-. 1) (Print (λ-. 0))

definition zero-llist :: nat llist where
  zero-llist = iterates id 0

lemma zero-correct:
  pohjola (λs. output-of s = []) zero (λl. l = zero-llist)
  unfolding zero-def
  apply (rule p-while1)
  apply (rule-tac x=λi s. True in exI)
  apply (simp add: ignores-output-def guard-def)
  apply (rule-tac x=λ-. [0] in exI)
  apply (rule conjI; clar simp)
  apply (simp add: Coinductive-List.inf-llist-def zero-llist-def)
  apply (simp add: lmap-iterates-id)
  apply (rule h-weaken)
  prefer 2
  apply (rule h-print)
  by (auto simp: print-def)

definition ex2 where
  ex2 = While (λ-. 1) zero

lemma ex2-correct:
  pohjola (λs. output-of s = []) ex2 (λl. l = zero-llist)
  unfolding ex2-def
  apply (rule p-while2[where R=λx y. (x, y) ∈ (measure (λx. 0)) and b=λ-.
  False])
  apply (simp-all add: guard-def)
  apply (rule hoare-pre-False)
  apply (rule zero-correct)
  done

end

```

References

- [1] J. A. Pohjola, H. Rostedt, and M. O. Myreen. Characteristic formulae for liveness properties of non-terminating CakeML programs. In *Interactive Theorem Proving (ITP)*. LIPIcs, 2019.