An encoding of Interval Temporal Logic in Isabelle/HOL

Antonio Cau Ben Moszkowski David Smallwood

June 11, 2019

Abstract

These Isabelle theories introduce the semantics and syntax of Interval Temporal Logic (ITL). The ITL proof system, as introduced in [4], has been encoded and its soundness has been checked. The encoding is shallow using the Intensional Logic technique of [3]. An extensive library of ITL theorems, taken from [5], has been checked. Furthermore we provide examples of using quantification over both static (rigid) and state (flexible) variables.

Contents

theory *Interval*
 imports
  Main
begin

1 Intervals

An interval is a sequence of elements of a particular type. Intervals are similar to list in Isabelle/HOL, the difference is that intervals have instead of nil a single element at the end. So an interval of length zero is a single element whereas for Isabelle's list we have that an empty list is nil (no element present).

The usual operations on intervals are defined: length (*intlen*), *prefix*, *suffix*, *sub*, *nth*, *intfirst*, *intlast*, *intapp* and *intrev*.

In order to define the semantics of the ITL chopstar we introduce *index-sequence* which is a sequence of chop (fuse) points. This sequence is again of type interval but the elements are natural numbers. Two functions *shift* and *shiftm* are introduced that are used to add (shift) and subtract a natural number of each element in the sequence of chop (fuse) points.

1.1 Definitions

datatype *'a interval* =
  St *'a* (⌈-⌉)
 | *Cons* *'a* *'a interval* (**infixr** ⊙ *65*)
for
 map: *map*
 rel: *interval-all2*
 pred: *interval-all*

type-synonym *index* = *nat interval*

syntax
 — interval Enumeration
 -interval :: *args* => *'a interval* (⟨(-)⟩)

translations
 ⟨*x*, *xs*⟩ == *x*⊙⟨*xs*⟩
 ⟨*x*⟩ == ⌈*x*⌉

primrec (*nonexhaustive*) *intlen* :: *'a interval* ⇒ *nat* **where**

```
    intlen (St x) = 0
  | intlen (x⊙xs) = 1+ (intlen xs)

primrec (nonexhaustive) nth :: 'a interval => nat => 'a where
    nth (St x) n      = x
  | nth (Cons x xs) n = (case n of 0 ⇒ x | Suc k ⇒ nth xs k)

primrec prefix:: nat ⇒ 'a interval ⇒ 'a interval where
    prefix n (St x) = (St x)
  | prefix n (Cons x xs) = (case n of 0 ⇒ (St x) | Suc m ⇒ (Cons x  (prefix m xs)))

primrec suffix:: nat ⇒ 'a interval ⇒ 'a interval where
   suffix n (St x) = (St x)
 | suffix n (Cons x  xs) = (case n of 0 ⇒ (Cons x  xs)  | Suc m ⇒ suffix m xs)

definition sub:: nat ⇒ nat ⇒ 'a interval ⇒ 'a interval
where
 sub n k xs = (if k<n then prefix 0 (suffix n xs)
                      else prefix (k−n) (suffix n xs)
              )

primrec intfirst :: 'a interval ⇒ 'a where
    intfirst (St x)     = x
  | intfirst (Cons x -) = x

primrec intlast :: 'a interval ⇒ 'a where
    intlast (St x)      = x
  | intlast (Cons - xs) = intlast xs

primrec intapp :: 'a interval ⇒ 'a interval ⇒ 'a interval (infixr ⊖ 65) where
intapp-St: (St x) ⊖ ys = x ⊙ ys |
intapp-Cons: (x⊙xs) ⊖ ys = x ⊙ (xs ⊖ ys)

primrec intrev :: 'a interval ⇒ 'a interval where
   intrev (St x) = (St x)
|  intrev (Cons x xs) = (intrev xs) ⊖ (St x)

definition index-sequence ::  nat ⇒ index ⇒ bool where
  index-sequence x idx ≡ (nth idx 0 = x) ∧ (∀ n. n<intlen idx ⟶ nth idx n < nth idx (Suc n))

definition shift :: nat ⇒ nat ⇒ nat where
  shift k = (λ x. x+k)

definition shiftm ::  nat ⇒ nat ⇒ nat where
  shiftm k = (λ x. (if k>x then 0 else  (x−k)))
```

1.2 Lemmas

Basic lemmas are introduced for each of the above operations on intervals.

1.2.1 Interval Length

lemma *interval-intlen-gr-zero* [*simp*]:
$intlen\ xs \geq 0$
by *auto*

lemma *interval-intlen-st* :
$intlen\ (St\ x) = 0$
by *simp*

lemma *interval-intlen-cons* [*simp*]:
$(intlen\ (x \odot xs)) = (intlen\ xs) + 1$
by *simp*

lemma *interval-intlen-cons-1* :
$intlen\ l > 0 \longleftrightarrow (\exists\ x\ ls.\ l = x\ \odot ls)$
by (*induct l*) *simp-all*

lemma *interval-intlen-map*:
$intlen\ (map\ f\ xs) = intlen\ xs$
by (*induct xs*) *simp-all*

1.2.2 nth

lemma *interval-nth-zero* [*simp*]:
$nth\ (x \odot xs)\ 0\ = x$
by *simp*

lemma *interval-nth-Suc* [*simp*]:
$nth\ (x \odot xs)\ (Suc\ n) = nth\ xs\ n$
by *auto*

lemma *interval-nth-last*:
$nth\ (x \odot xs)\ (intlen\ (x \odot xs)) = nth\ xs\ (intlen\ xs)$
by *simp*

lemma *interval-nth-cons*:
assumes $0<i \wedge i<1+intlen(xs)$
shows $nth(x \odot xs)\ i = nth\ xs\ (i-1) \wedge$
$nth(x \odot xs)\ (i+1) = nth\ xs\ ((i-1)+1)$
by (*metis One-nat-def Suc-leI add.commute assms interval-nth-Suc le-add-diff-inverse2 plus-1-eq-Suc*)

lemma *interval-nth-zero-intfirst*:
$nth\ xs\ 0 = intfirst\ xs$
by (*induct xs*) *simp-all*

lemma *interval-nth-intlen-intlast*:
$nth\ xs\ (intlen\ xs) = intlast\ xs$
by (*induct xs*) *simp-all*

lemma *interval-st-intlen* :

$(xs = (St\ x)) \longleftrightarrow intlen\ xs = 0 \wedge nth\ xs\ 0 = x$
by (*induct xs*) *simp-all*

lemma *interval-eq-nth-eq* :
$(xs = ys) = (intlen\ xs = intlen\ ys \wedge (\forall\ i \leq intlen\ xs.\ nth\ xs\ i = nth\ ys\ i))$
apply (*induct xs arbitrary*: *ys*)
apply (*metis interval-st-intlen le-numeral-extra*(*3*))
apply (*case-tac ys*, *simp*)
by *fastforce*

lemma *interval-nth-map* :
$nth\ (map\ f\ xs)\ i = f\ (nth\ xs\ i)$
apply (*induct xs arbitrary*: *i*, *simp*)
apply (*case-tac i*, *simp*, *simp*)
done

1.2.3 index sequence

lemma *interval-idx-less*:
assumes *iseq*: *index-sequence x idx*
shows $(n<intlen\ idx \wedge n+k<intlen\ idx) \longrightarrow nth\ idx\ n < nth\ idx\ (Suc(n+k))$
apply (*induct k*)
using *index-sequence-def iseq* **apply** *auto*[*1*]
using *index-sequence-def iseq* **by** *auto*

lemma *interval-idx-less-last* :
assumes *index-sequence x idx*
shows $(i<intlen\ idx \wedge i+(intlen\ idx - (i+1))< intlen\ idx)$
$\longrightarrow nth\ idx\ i < nth\ idx\ (Suc(i+(intlen\ idx -(i+1))))$
using *assms interval-idx-less* **by** *blast*

lemma *interval-idx-less-last-1*:
assumes *index-sequence x idx*
shows $i<intlen\ idx \longrightarrow nth\ idx\ i < nth\ idx\ (intlen\ idx)$
using *assms interval-idx-less-last* **by** *auto*

lemma *interval-idx-greater-first*:
assumes *index-sequence x idx*
shows $(i>0 \wedge i\leq intlen\ idx) \longrightarrow x < nth\ idx\ i$
apply (*induct i*, *simp*)
using *assms*
by (*metis One-nat-def Suc-le-lessD add-Suc index-sequence-def interval-idx-less*
less-le-trans plus-1-eq-Suc)

lemma *interval-idx-cons*:
$index\text{-}sequence\ 0\ (x \odot ls) =$
$(x=0 \wedge x<nth\ ls\ 0 \wedge index\text{-}sequence\ (nth\ ls\ 0)\ ls)$
apply (*simp add*: *index-sequence-def*)
using *less-Suc-eq-0-disj* **by** *auto*

```
lemma interval-idx-shift-mono:
   mono (shift k)
by (simp add: Interval.shift-def mono-def)

lemma interval-idx-expand:
 index-sequence 0 l  ∧ (nth l (intlen l)) = (intlen xs) ∧ 0≤i ∧ i< (intlen l)
  ⟹ 0≤ (nth l i) ∧ (nth l i) ≤  (nth l (i+1)) ∧ (nth l (i+1)) ≤ (intlen xs)
apply (simp add: index-sequence-def)
apply (induct l, simp)
by (metis Suc-lessI eq-imp-le index-sequence-def interval-idx-less-last-1 less-imp-le-nat)

lemma interval-idx-shift-idx [simp]:
    ( index-sequence (x+k) (map (shift k)  idx)) = (index-sequence x idx)
by (simp add: Interval.shift-def index-sequence-def interval-intlen-map interval-nth-map)

lemma interval-idx-shiftm :
   (index-sequence k (lsk) ∧ ls = map (shiftm k) lsk) ⟹
    index-sequence 0 (ls) ∧ (intlen ls) = (intlen lsk)
by (simp add: interval-eq-nth-eq index-sequence-def shift-def shiftm-def interval-nth-map )
  (smt Suc-leI diff-less-mono index-sequence-def interval-idx-greater-first interval-intlen-map
      le-less-trans less-Suc-eq-0-disj not-less order.asym)

lemma interval-lsk-ls :
   (index-sequence k (lsk) ∧ lsk = map (shift k) ls ∧ index-sequence 0 (ls) ) =
     (index-sequence k (lsk) ∧ ls = map (shiftm k) lsk ∧ index-sequence 0 (ls) )
apply (simp add: interval-eq-nth-eq index-sequence-def shift-def shiftm-def interval-nth-map)
apply rule
apply (metis (no-types, lifting) add-diff-cancel-right' interval-intlen-map not-add-less2)
by (metis (no-types, lifting) Suc-eq-plus1 add.commute add-cancel-right-left add-diff-inverse-nat
      ex-least-nat-less interval-intlen-map le-SucE le-zero-eq not-less-zero order-refl)

lemma interval-idx-link-shiftm:
   (index-sequence k (lsk) ∧ ls = map (shiftm k) lsk ) =
    (index-sequence k (lsk) ∧ ls = map (shiftm k) lsk ∧
     index-sequence 0 (ls) ∧ (intlen ls) =(intlen lsk))
using interval-idx-shiftm by blast

lemma interval-idx-link:
   (lsk = map (shift k) ls ∧ index-sequence 0 (ls) ) =
    (lsk = map (shift k) ls ∧ index-sequence k (lsk) ∧ index-sequence 0 (ls)∧
     (intlen ls) =(intlen lsk))
by (metis Interval.shift-def add-diff-cancel-left' diff-diff-cancel diff-is-0-eq'
   interval-idx-shift-idx interval-idx-shift-mono interval-intlen-map le-numeral-extra(3) mono-def)

lemma interval-idx-bound-0 :
 assumes index-sequence 0 ls ∧ Interval.nth ls (intlen ls) = intlen (suffix k xs)
 shows   ((i≤intlen ls) ⟶ ((nth ls (i)) ≤ (intlen (suffix k xs))))
using assms
by (metis add.commute add-eq-if eq-iff interval-idx-less le-add-diff-inverse2
              le-neq-implies-less lessI less-imp-le-nat)
```

lemma *interval-idx-bound-1*:
(*index-sequence 0* (*ls*) ∧ (*nth* (*ls*) (*intlen* (*ls*))) = (*intlen* (*suffix k xs*))) ⟷
(*index-sequence 0* (*ls*) ∧ (*nth* (*ls*) (*intlen* (*ls*))) = (*intlen* (*suffix k xs*)) ∧
(∀ *i*. (*i*≤*intlen ls*) ⟶ ((*nth ls* (*i*)) ≤ (*intlen* (*suffix k xs*)))))
using *interval-idx-bound-0* **by** *blast*

1.2.4 prefix, suffix and sub

lemma *interval-prefix-state* [*simp*]:
prefix m (*St x*) = (*St x*)
by *simp*

lemma *interval-prefix-suc* [*simp*]:
prefix (*Suc m*) (*x*⊙*xs*) = *x* ⊙ (*prefix m xs*)
by *auto*

lemma *interval-prefix-zero* [*simp*]:
prefix 0 (*x*⊙*xs*) = *St x*
by *auto*

lemma *interval-prefix-zero-intfirst* [*simp*]:
prefix 0 xs = *St* (*intfirst xs*)
by (*induct xs*) *simp-all*

lemma *interval-intfirst-prefix* [*simp*]:
i≤*intlen xs* ⟹ *intfirst* (*prefix i xs*) = *intfirst xs*
by (*induct xs arbitrary*: *i*, *auto*) (*case-tac i*, *auto*)

lemma *interval-prefix-intlen* [*simp*]:
(*prefix* (*intlen xs*) *xs*) = *xs*
by (*induct xs*) *simp-all*

lemma *interval-prefix-intlen-gr-1* [*simp*]:
(*prefix* ((*intlen xs*)+*i*) *xs*) = *xs*
by (*induct xs*) *simp-all*

lemma *interval-intlen-prefix-cons* [*simp*]:
intlen(*prefix* (*Suc i*) (*x*⊙*xs*)) = *1* + *intlen*(*prefix i xs*)
using *interval-intlen-cons* **by** *auto*

lemma *interval-prefix-length* :
intlen (*prefix i xs*) = (*if i*≤ *intlen xs then i else intlen xs*)
by (*induct xs arbitrary*: *i*, *simp*) (*case-tac i*, *auto*)

lemma *interval-prefix-length-good* [*simp*]:
assumes *i*≤ *intlen xs*
shows (*intlen* (*prefix i xs*)) = *i*
using *assms* **by** (*simp add*: *interval-prefix-length*)

```
lemma interval-prefix-length-bad [simp] :
  assumes    i > intlen xs
  shows      intlen (prefix i xs) = intlen xs
using assms by (simp add: interval-prefix-length)
```

```
lemma interval-pref-intlen-bound :
 assumes   i ≤ (intlen xs)
 shows     intlen (prefix i xs) ≤ intlen xs
using assms by (induct xs, simp) (metis interval-prefix-length)
```

```
lemma interval-suffix-length:
     intlen (suffix i xs ) = (if i≤ intlen xs then (intlen xs)−i else 0)
by (induct xs arbitrary: i, simp) (case-tac i, auto)
```

```
lemma interval-suffix-length-good [simp]:
   assumes  i≤ intlen xs
   shows    intlen (suffix i xs ) = (intlen xs)−i
using assms by (simp add: interval-suffix-length)
```

```
lemma interval-suffix-length-bad [simp]:
  assumes   i> intlen xs
  shows     intlen (suffix i xs ) = 0
using assms by (simp add: interval-suffix-length)
```

```
lemma interval-nth-prefix [simp]:
  i≤intlen xs ∧ k ≤ i ⟹ nth (prefix i xs) k = nth xs k
apply (induct xs arbitrary: i k, auto)
apply (case-tac i, auto)
apply (case-tac k, auto)
done
```

```
lemma interval-nth-suffix [simp]:
  i≤intlen xs ∧ k ≤ intlen xs −i ⟹ nth (suffix i xs) k = nth xs (i+k)
by (induct xs arbitrary: i k, auto) (case-tac i, auto)
```

```
lemma interval-suffix-prefix-help-1:
 assumes  ia+i ≤ intlen xs ∧ k ≤ ia
 shows  nth (prefix ia (suffix i xs)) k = nth (suffix i (prefix (ia +i) xs)) k
proof −
 have 1:  nth (prefix ia (suffix i xs)) k = nth (suffix i xs) k
 using interval-nth-prefix assms by (metis interval-prefix-intlen-gr-1 le-cases le-iff-add)
 have 2:  nth (suffix i xs) k = nth xs (i+k)
 using interval-nth-suffix assms by (simp add: add-le-imp-le-diff)
 have 3:  nth xs (i+k) = nth (prefix (ia+i) xs) (i+k)
 using interval-nth-prefix assms by simp
 have 4:  nth (prefix (ia+i) xs) (i+k) = nth (suffix i (prefix (ia +i) xs)) k
 using interval-nth-suffix assms by simp
 from 1 2 3 4 show ?thesis by auto
qed
```

```
lemma interval-suffix-prefix-help-2:
 assumes ia+i ≤ intlen xs
 shows   (∀ k ≤ ia . nth (prefix ia (suffix i xs)) k = nth (suffix i (prefix (ia +i) xs)) k)
using interval-suffix-prefix-help-1 using assms by fastforce

lemma interval-suffix-prefix-help-3:
 assumes ia+i ≤ intlen xs
 shows  intlen (prefix ia (suffix i xs)) = intlen (suffix i (prefix (ia +i) xs))
using assms interval-prefix-length-good interval-suffix-length-good by auto

lemma interval-suffix-prefix-swap:
 assumes ia+i ≤ intlen xs
 shows   prefix ia (suffix i xs) = suffix i (prefix (ia +i) xs)
by (simp add: interval-eq-nth-eq interval-suffix-prefix-help-2 interval-suffix-prefix-help-3 assms)

lemma interval-prefix-prefix-zero [simp]:
    prefix 0 ( prefix 0 xs ) = prefix 0 xs
by (induct xs) simp-all

lemma interval-pref-pref [simp]:
    (prefix i (prefix i xs)) = prefix i xs
by (metis interval-prefix-intlen interval-prefix-intlen-gr-1 interval-prefix-length
         less-imp-add-positive not-less)

lemma interval-pref-pref-3 [simp]:
    (prefix i (prefix (i+k) xs)) = prefix i xs
apply (induct xs arbitrary: i k, simp)
apply (case-tac i, auto)
by (simp add: Nitpick.case-nat-unfold)

lemma interval-pref-help:
 assumes i≤intlen (prefix (intlen xs − Suc 0) xs)
 shows   (prefix i (prefix (intlen xs − Suc 0) xs)) = (prefix i xs)
using assms
by (metis diff-le-self interval-pref-pref-3 interval-prefix-length
          ordered-cancel-comm-monoid-diff-class.add-diff-inverse)

lemma interval-pref-pref-help:
 assumes intlen xs >0 ∧ ia<intlen (xs)
 shows  (prefix ia (prefix (intlen xs − Suc 0) xs)) = (prefix ia  xs)
using assms
by (metis Suc-leI Suc-le-mono Suc-pred diff-le-self interval-pref-help interval-prefix-length-good)

lemma interval-pref-pref-help-1:
 assumes i>0 ∧ i≤ intlen xs
 shows  (prefix (intlen (prefix i xs) − Suc 0) (prefix i xs)) =
         (prefix (intlen (prefix i xs) − Suc 0) xs)
using assms interval-pref-pref-3 by (metis diff-le-self interval-prefix-length-good le-iff-add)

lemma interval-suffix-suc [simp]:
     suffix (Suc m) (x ⊙ xs) = suffix m xs
by auto

lemma interval-suffix-zero [simp]:
     suffix 0 xs = xs
by (induct xs) simp-all

lemma interval-suffix-intlen [simp]:
     suffix (intlen xs) xs = (St (nth xs (intlen xs)))
by (induct xs) simp-all

lemma interval-suffix-intlast [simp]:
    suffix (intlen xs) xs = St (intlast xs)
by (induct xs) simp-all

lemma interval-suffix-suffix [simp]:
     suffix i (suffix j xs) = suffix (i+j) xs
apply (induct xs arbitrary: i j, simp)
apply (case-tac i, auto)
by (simp add: Nitpick.case-nat-unfold)

lemma interval-prefix-suffix-intlen:
    intlen (prefix ia (suffix i xs)) =
     (if i ≤ intlen xs then
      (if ia≤ intlen xs −i  then ia else (intlen xs) −i )
      else 0)
by (metis interval-prefix-length interval-suffix-length le-zero-eq)

lemma interval-prefix-suffix-intlen-good [simp]:
  assumes  ia≤ intlen xs −i ∧ i ≤ intlen xs
  shows     intlen (prefix ia (suffix i xs)) = ia
using assms by (simp add: interval-prefix-suffix-intlen)

lemma interval-prefix-suffix-intlen-bad-0 [simp]:
  assumes  i> intlen xs
  shows    intlen (prefix ia (suffix i xs)) = 0
using assms by (simp add: interval-prefix-suffix-intlen)

lemma interval-prefix-suffix-intlen-bad-1 [simp] :
  assumes  i ≤ intlen xs ∧ ia > intlen xs −i
  shows    intlen (prefix ia (suffix i xs)) = (intlen xs) −i
using assms by (simp add: interval-prefix-suffix-intlen)

lemma interval-suffix-suffix-3:
 assumes i>0 ∧ ia<i ∧ i≤ intlen xs
 shows   (suffix (i−ia) (suffix ((intlen xs)−i) xs)) = (suffix (((intlen xs)−ia)) xs)
using assms by simp

lemma interval-sub-zero-prefix :
     sub 0 k xs = prefix k xs
```

```
by (simp add: Interval.sub-def)

lemma interval-sub-suffix :
  assumes  (i < j ∧ j≤ (intlen xs)−k)
  shows    (sub (i+k) (j+k) xs) = (sub i j (suffix k xs))
using assms by (simp add: Interval.sub-def)

lemma interval-sub-prefix-suffix-0:
  assumes (0 ≤ i ∧ ia+i ≤ intlen xs)
  shows     (sub i (i+ia) xs) = (prefix (ia) (suffix i xs))
using assms by (simp add: Interval.sub-def)

lemma interval-sub-prefix-suffix:
 assumes  0 ≤ i ∧ i≤j ∧ j ≤ intlen xs
 shows     (sub i j xs) = (prefix (j−i) (suffix i xs))
using assms by (simp add: Interval.sub-def)
```

1.2.5 Reverse

```
lemma interval-intlen-intapp [simp]:
  intlen (xs ⊖ ys) = (intlen xs) + (intlen ys) +1
by (induct xs arbitrary: ys) simp-all

lemma interval-intrev-intlen [simp]:
  intlen (intrev xs) = intlen xs
by (induct xs, simp, simp)

lemma interval-suffix-intapp [simp]:
 suffix (Suc (intlen xs)) (xs ⊖ ys) = ys
by (induct xs) simp-all

lemma interval-suffix-intapp2 [simp]:
 suffix (intlen xs − k) ( xs) ⊖ ys = suffix (intlen xs − k) ( xs ⊖ ys)
by (induct xs, simp)
   (metis Suc-diff-le diff-is-0-eq′ intapp-Cons interval-suffix-suc interval-suffix-zero
          intlen.simps(2) not-less-eq-eq plus-1-eq-Suc)

lemma interval-intapp-assoc [simp]:
  (xs ⊖ ys) ⊖ zs = xs ⊖ ( ys ⊖ zs )
by (induct xs) simp-all

lemma interval-intapp-nth:
  nth (xs ⊖ ys) k = (if k ≤ intlen xs
                    then (nth xs k)
                    else (nth ys (k − (intlen xs) −1)) )
apply (induct xs arbitrary: k)
apply (case-tac k, simp, simp)
apply (case-tac k, simp, simp)
done
```

```
lemma interval-rev-intapp [simp]:
  intrev (xs ⊖ ys) = (intrev ys) ⊖ (intrev xs)
by (induct xs) simp-all

lemma interval-rev-rev-ident [simp]:
  intrev (intrev xs) = xs
by (induct xs) auto

lemma interval-rev-swap :
  ((intrev xs) = ys) = (xs = intrev ys)
by auto

lemma interval-intlast-intrev:
 intlast (intrev xs) = intfirst xs
by (induct xs, auto)
   (metis Suc-eq-plus1 add.right-neutral interval.inject(1) interval-intlen-intapp
          interval-intlen-st interval-suffix-intapp interval-suffix-intlast)

lemma interval-intfirst-intrev:
 intfirst (intrev xs) = intlast xs
by (induct xs, auto)
   (metis intapp-St interval-intlast-intrev interval-rev-intapp intlast.simps(2) intrev.simps(1))

lemma interval-intrev-nth:
 k≤ intlen (intrev xs) ⟹ (nth  (intrev xs) k) = (nth  xs ((intlen xs) −k))
apply (induct xs, simp)
apply simp
apply (case-tac k)
apply (simp add: interval-intapp-nth)
by (smt Interval.nth.simps(1) Suc-diff-Suc diff-Suc-Suc diff-is-0-eq' interval-intapp-nth
        interval-intrev-intlen le-SucE less-Suc-eq-le old.nat.simps(4) old.nat.simps(5))

lemma interval-intrev-prefix:
 k≤ intlen xs ⟹ intrev( prefix k xs) = suffix ((intlen xs) − k) (intrev xs)
apply (induct xs arbitrary: k, simp)
apply simp
apply (case-tac k)
apply (metis diff-zero interval-intrev-intlen interval-suffix-intapp intrev.simps(1) old.nat.simps(4))
by (metis Suc-le-mono diff-Suc-Suc interval-intrev-intlen interval-suffix-intapp2
          intrev.simps(2) old.nat.simps(5))

lemma interval-intrev-suffix:
 k≤ intlen xs ⟹ intrev( suffix k xs) = prefix ((intlen xs) − k) (intrev xs)
by (induct xs arbitrary: k, simp, simp add: interval-intrev-prefix interval-rev-swap)

lemma interval-intrev-sub1:
 assumes  0 ≤ i ∧ i≤j ∧ j ≤ intlen xs
 shows    intrev (sub i j xs) = intrev (prefix (j−i) (suffix i xs))
using assms interval-sub-prefix-suffix by (simp add: interval-sub-prefix-suffix)
```

```
lemma interval-intrev-sub2:
 assumes  0 ≤ i ∧ i≤j ∧ j ≤ intlen xs
 shows    intrev (prefix (j−i) (suffix i xs)) = suffix ((intlen xs) − j) (intrev (suffix i xs))
using assms interval-intrev-prefix[of j−i suffix i xs]  by auto

lemma interval-intrev-sub3:
 assumes  0 ≤ i ∧ i≤j ∧ j ≤ intlen xs
 shows    suffix ((intlen xs) − j) (intrev (suffix i xs)) =
          suffix ((intlen xs) −j) (prefix ((intlen xs) − i) (intrev xs))
using assms interval-intrev-suffix[of i xs] by auto

lemma interval-intrev-sub4:
 assumes  0 ≤ i ∧ i≤j ∧ j ≤ intlen xs
 shows   suffix ((intlen xs) −j) (prefix ((intlen xs) − i) (intrev xs)) =
         sub ((intlen xs) −j) ((intlen xs) − i) (intrev xs)
using assms by (simp add: diff-le-mono2 interval-sub-prefix-suffix interval-suffix-prefix-swap)

lemma interval-intrev-sub:
 assumes  0 ≤ i ∧ i≤j ∧ j ≤ intlen xs
 shows    intrev (sub i j xs) = sub ((intlen xs) −j) ((intlen xs) − i) (intrev xs)
using assms
by (simp add: interval-intrev-sub1 interval-intrev-sub2 interval-intrev-sub3 interval-intrev-sub4)

lemma interval-intrev-idx-2:
 assumes index-sequence 0 l  ∧ (nth l (intlen l)) = (intlen xs) ∧
         0≤i ∧ i< (intlen l)
 shows  (intrev (sub (nth l i) (nth l (i+1)) xs)) =
   ( (sub ((intlen xs)− (nth l (i+1))) ((intlen xs) − (nth l i)) (intrev xs)))
using assms interval-idx-expand interval-intrev-sub[of (nth l i)  (nth l (i+1))  xs]
by blast

lemma interval-intrev-idx-3:
 assumes  index-sequence 0 l  ∧ (nth l (intlen l)) = (intlen xs) ∧
          ls = map (λ x. (intlen xs) −x) (intrev l)
 shows   (nth ls 0) = 0 ∧ (nth ls (intlen ls)) = (intlen xs) ∧ intlen ls =intlen l
using assms
by (metis diff-self-eq-0 diff-zero index-sequence-def interval-intfirst-intrev
         interval-intlast-intrev interval-intlen-map interval-intrev-intlen
         interval-nth-intlen-intlast interval-nth-map interval-nth-zero-intfirst)

lemma interval-intrev-idx-4:
  index-sequence 0 l  ∧ (nth l (intlen l)) = (intlen xs) ∧
            ls = map (λ x. (intlen xs) −x) (intrev l)
  ⟹ i≤ intlen ls −→ (nth ls i) = (intlen xs) − (nth l ((intlen l)−i))
apply (induct ls)
apply (metis diff-zero interval-intlen-st interval-intrev-idx-3 le-0-eq)
by (simp add: interval-intlen-map interval-intrev-nth interval-nth-map)

lemma interval-intrev-idx-5:
 assumes  (index-sequence 0 l  ∧ (nth l (intlen l)) = (intlen xs))
 shows   (i< intlen l ⟶
          (intlen xs) − (nth l ((intlen l)−i)) < (intlen xs) − (nth l ((intlen l)−(i+1))))
using assms
by (smt Suc-diff-Suc Suc-eq-plus1 add-gr-0 add-less-cancel-left diff-less
        index-sequence-def le-add-diff-inverse2 le-numeral-extra(3) less-diff-conv
        less-imp-le-nat not-gr-zero interval-idx-expand)

lemma interval-intrev-idx-6:
 assumes (index-sequence 0 l ∧ (nth l (intlen l)) = (intlen xs) ∧
          ls = map (λ x. (intlen xs) −x) (intrev l))
 shows   (i< intlen ls ⟶
          ((nth ls i) = (intlen xs) − (nth l ((intlen l)−i)) ∧
          (nth ls (i+1)) = (intlen xs) − (nth l ((intlen l)−(i+1))) ∧
          (nth ls i) < (nth ls (i+1))))
proof −
 have 1: (i< intlen ls ⟶ (nth ls i) = (intlen xs) − (nth l ((intlen l)−i)))
         using assms interval-intrev-idx-4 less-imp-le-nat by blast
 have 2: (i< intlen ls ⟶ (nth ls (i+1)) = (intlen xs) − (nth l ((intlen l)−(i+1))))
         using assms by (simp add: interval-intrev-idx-4)
 have 3: ( i< intlen ls ⟶
          ((nth ls i) = (intlen xs) − (nth l ((intlen l)−i)) ∧
          (nth ls (i+1)) = (intlen xs) − (nth l ((intlen l)−(i+1)))))
         using 1 2 by auto
 have 4: (i< intlen ls ⟶
            ((nth ls i) = (intlen xs) − (nth l ((intlen l)−i)) ∧
            (nth ls (i+1)) = (intlen xs) − (nth l ((intlen l)−(i+1))) ∧
            (nth ls i) < (nth ls (i+1))))
          using assms 3 index-sequence-def interval-intrev-idx-5
         by (metis interval-intlen-map interval-intrev-intlen)
 from 4 show ?thesis by blast
qed

lemma interval-intrev-idx-7:
 assumes (index-sequence 0 l ∧ (nth l (intlen l)) = (intlen xs) ∧
          ls = map (λ x. (intlen xs) −x) (intrev l))
   shows index-sequence 0 ls
using assms interval-intrev-idx-6 interval-intrev-idx-3
by (metis Suc-eq-plus1 index-sequence-def)

lemma interval-intrev-idx-8:
 assumes index-sequence 0 l ∧ (nth l (intlen l)) = (intlen xs) ∧
         ls = map (λ x. (intlen xs) −x) (intrev l) ∧ index-sequence 0 ls
 shows   i<intlen ls ⟶
         (intlen xs)− (nth l (i+1)) = nth ls ((intlen ls)−(i+1)) ∧
         (intlen xs) − (nth l i)    = (nth ls ((intlen ls) −i))
using assms interval-intrev-idx-4
by (smt Suc-eq-plus1 Suc-leI add-diff-cancel-right' assms diff-diff-cancel diff-diff-left
        diff-le-self interval-intrev-idx-3)

lemma interval-intrev-idx-9:
 assumes index-sequence 0 l  ∧ (nth l (intlen l)) = (intlen xs) ∧
         ls = map (λ x. (intlen xs) −x) (intrev l) ∧ index-sequence 0 ls
 shows   i<intlen ls ⟶
         sub ((intlen xs)− (nth l (i+1))) ((intlen xs) − (nth l i)) (intrev xs) =
         sub (nth ls ((intlen ls)−(i+1))) ((nth ls ((intlen ls) −i)) ) (intrev xs)

using interval-intrev-idx-8 using assms by fastforce

lemma interval-intrev-idx-11:
 assumes (index-sequence 0 l  ∧ (nth l (intlen l)) = (intlen xs))
 shows    i≤intlen l ⟶
        (nth l i) = (nth (map (λ x.(intlen xs)−x) (intrev (map (λ x.(intlen xs)−x) (intrev l)))) i)
using assms index-sequence-def
by (smt diff-diff-cancel diff-is-0-eq diff-less diff-zero leD le-cases not-gr-zero
       interval-intrev-idx-3 interval-intrev-idx-6 interval-intrev-idx-7)

lemma interval-intrev-idx-12:
 assumes (index-sequence 0 l  ∧ (nth l (intlen l)) = (intlen xs))
 shows    l = map (λ x. (intlen xs) −x  ) (intrev (map (λ x. (intlen xs) −x) (intrev l)))
using assms interval-intrev-idx-11
by (simp add: interval-intrev-idx-11 interval-eq-nth-eq interval-intlen-map)


end
```

2 Semantics

```
theory Semantics
imports Interval HOL−TLA.Intensional
begin
```

This theory mechanises a *shallow* embedding of ITL using the *Interval* and *Intensional* theories. A shallow embedding represents ITL using Isabelle/HOL predicates, while a *deep* embedding [1] would represent ITL formulas as mutually inductive datatypes. See, e.g., [6] for a discussion about deep vs. shallow embeddings in Isabelle/HOL. The choice of a shallow over a deep embedding is motivated [3, 2] by the following factors: a shallow embedding is usually less involved, and existing Isabelle theories and tools can be applied more directly to enhance automation; due to the lifting in the *Intensional* theory, a shallow embedding can reuse standard logical operators, whilst a deep embedding requires a different set of operators for formulas. Finally, since our target is system verification rather than proving meta-properties of the logic, which requires a deep embedding, a shallow embedding is more fit for purpose.

2.1 Types of Formulas

To mechanise the ITL semantics, the following type abbreviations are used:

```
type-synonym ('a,'b) formfun = 'a interval ⇒ 'b
type-synonym 'a formula      = ('a,bool) formfun
type-synonym ('a,'b) stfun   = 'a ⇒ 'b
type-synonym 'a stpred       = ('a,bool) stfun
```

instance
fun :: (*type*,*type*) *world* **..**

instance
prod :: (*type*,*type*) *world* **..**

instance
interval :: (*type*) *world* **..**

Pair, function, and interval are instantiated to be of type class world. This allows use of the lifted Intensional logic for formulas, and standard logical connectives can therefore be used.

2.2 Semantics of ITL

The semantics of ITL is defined.

definition *skip-d* :: (*'a* ::*world*) *formula*
where *skip-d* ≡ λ*s*. *intlen s* =*1*

definition *chop-d* :: (*'a* ::*world*) *formula* ⇒ (*'a* ::*world*) *formula* ⇒ (*'a* ::*world*) *formula*
where *chop-d F1 F2* ≡ λ *s*. ∃ *n*. *0*≤ *n* ∧ *n*≤*intlen s* ∧ ((*prefix n s*) ⊨ *F1*) ∧ ((*suffix n s*) ⊨ *F2*)

definition *chopstar-d* :: (*'a*::*world*) *formula* ⇒ *'a formula*
where *chopstar-d F* ≡ λ*s*. (∃ (*l*::*index*). *index-sequence 0 l* ∧ (*nth l* (*intlen l*)) = (*intlen s*) ∧
(∀ *i*. (*0*≤*i* ∧ *i*< (*intlen l*)) ⟶
((*sub* (*nth l i*) (*nth l* (*i*+*1*)) *s*) ⊨ *F*)
)
)

definition *reverse-d* :: (*'a*::*world*, *'b*) *formfun* ⇒ (*'a*, *'b*) *formfun*
where *reverse-d F* ≡ λ *s*. *intrev s* ⊨ *F*

definition *current-val-d* :: (*'a*::*world*,*'b*) *stfun* ⇒ (*'a*,*'b*) *formfun*
where *current-val-d f* ≡ λ *s*. (*nth s 0*) ⊨ *f*

definition *next-val-d* :: (*'a*::*world*,*'b*) *stfun* ⇒ (*'a*,*'b*) *formfun*
where *next-val-d f* ≡ λ *s*. *if intlen s* >*0 then* ((*nth s 1*) ⊨ *f*) *else* (ϵ (*x*::*'b*). *x*=*x*)

definition *fin-val-d* :: (*'a*::*world*,*'b*) *stfun* ⇒ (*'a*,*'b*) *formfun*
where *fin-val-d f* ≡ λ *s*. (*nth s* (*intlen s*)) ⊨ *f*

definition *penult-val-d* :: (*'a*::*world*,*'b*) *stfun* ⇒ (*'a*,*'b*) *formfun*
where *penult-val-d f* ≡ λ *s*. *if intlen s* >*0 then* (*nth s* ((*intlen s*)−*1*) ⊨ *f*) *else* (ϵ (*x*::*'b*). *x*=*x*)

2.2.1 Concrete Syntax

This is the concrete syntax for the (abstract) operators above.

syntax
-skip-d :: *lift* ((*skip*))

-chop-d :: [*lift*,*lift*] ⇒ *lift* ((-;-) [*84*,*84*] *83*)
-chopstar-d :: *lift* ⇒ *lift* ((-*) [*85*] *85*)
-reverse-d :: *lift* ⇒ *lift* ((-^{r}) [*85*] *85*)
-current-val-d :: *lift* ⇒ *lift* (($-) [*100*] *99*)
-next-val-d :: *lift* ⇒ *lift* ((-$) [*100*] *99*)
-fin-val-d :: *lift* ⇒ *lift* ((!-) [*100*] *99*)
-penult-val-d :: *lift* ⇒ *lift* ((-!) [*100*] *99*)
TEMP :: *lift* ⇒ *'b* ((*TEMP* -))

syntax (*ASCII*)
-skip-d :: *lift* ((*skip*))
-chop-d :: [*lift*,*lift*] ⇒ *lift* ((-;-) [*84*,*84*] *83*)
-chopstar-d :: *lift* ⇒ *lift* ((*chopstar* -) [*85*] *85*)
-reverse-d :: *lift* ⇒ *lift* ((*reverse* -) [*85*] *85*)
-current-val-d :: *lift* ⇒ *lift* (($-) [*100*] *99*)
-next-val-d :: *lift* ⇒ *lift* ((-$) [*100*] *99*)
-fin-val-d :: *lift* ⇒ *lift* ((!-) [*100*] *99*)
-penult-val-d :: *lift* ⇒ *lift* ((-!) [*100*] *99*)
translations
-skip-d ⇌ *CONST skip-d*
-chop-d ⇌ *CONST chop-d*
-chopstar-d ⇌ *CONST chopstar-d*
-reverse-d ⇌ *CONST reverse-d*
-current-val-d ⇌ *CONST current-val-d*
-next-val-d ⇌ *CONST next-val-d*
-fin-val-d ⇌ *CONST fin-val-d*
-penult-val-d ⇌ *CONST penult-val-d*
TEMP F ⇀ (*F*:: (- *interval*) ⇒ -)

2.3 Abbreviations

Some standard temporal abbreviations, with their concrete syntax.

definition *sometimes-d* :: (*'a*::*world*) *formula* ⇒ *'a formula*
where *sometimes-d F* ≡ *LIFT*(#*True*;*F*)

definition *di-d* :: (*'a*::*world*) *formula* ⇒ *'a formula*
where *di-d F* ≡ *LIFT*(*F*;#*True*)

definition *da-d* :: (*'a*::*world*) *formula* ⇒ *'a formula*
where *da-d F* ≡ *LIFT*(#*True*;(*F*;#*True*))

definition *next-d* :: (*'a*::*world*) *formula* ⇒ *'a formula*
where *next-d F* ≡ *LIFT*(*skip*;*F*)

definition *prev-d* :: (*'a*::*world*) *formula* ⇒ *'a formula*
where *prev-d F* ≡ *LIFT*(*F*;*skip*)

2.3.1 Concrete Syntax

syntax

-sometimes-d :: *lift* ⇒ *lift* ((◇-) [*88*] *87*)
-di-d :: *lift* ⇒ *lift* ((*di* -) [*88*] *87*)
-da-d :: *lift* ⇒ *lift* ((*da* -) [*88*] *87*)
-next-d :: *lift* ⇒ *lift* ((○ -) [*88*] *87*)
-prev-d :: *lift* ⇒ *lift* ((*prev* -) [*88*] *87*)

syntax (*ASCII*)
-sometimes-d :: *lift* ⇒ *lift* ((<>-) [*88*] *87*)
-di-d :: *lift* ⇒ *lift* ((*di* -) [*88*] *87*)
-da-d :: *lift* ⇒ *lift* ((*da* -) [*88*] *87*)
-next-d :: *lift* ⇒ *lift* ((*next* -) [*88*] *87*)
-prev-d :: *lift* ⇒ *lift* ((*prev* -) [*88*] *87*)

translations
-sometimes-d ⇌ *CONST sometimes-d*
-di-d ⇌ *CONST di-d*
-da-d ⇌ *CONST da-d*
-next-d ⇌ *CONST next-d*
-prev-d ⇌ *CONST prev-d*

definition *always-d* :: (*'a*::*world*) *formula* ⇒ *'a formula*
where *always-d F* ≡ *LIFT*(¬(◇(¬*F*)))

definition *bi-d* :: (*'a*::*world*) *formula* ⇒ *'a formula*
where *bi-d F* ≡ *LIFT*(¬(*di*(¬*F*)))

definition *ba-d* :: (*'a*::*world*) *formula* ⇒ *'a formula*
where *ba-d F* ≡ *LIFT*(¬(*da*(¬*F*)))

definition *wnext-d* :: (*'a*::*world*) *formula* ⇒ *'a formula*
where *wnext-d F* ≡ *LIFT*(¬(○(¬*F*)))

definition *wprev-d* :: (*'a*::*world*) *formula* ⇒ *'a formula*
where *wprev-d F* ≡ *LIFT*(¬(*prev*(¬*F*)))

definition *more-d* :: (*'a*::*world*) *formula*
where *more-d* ≡ *LIFT*(○(#*True*))

syntax
-always-d :: *lift* ⇒ *lift* ((□-) [*88*] *87*)
-bi-d :: *lift* ⇒ *lift* ((*bi* -) [*88*] *87*)
-ba-d :: *lift* ⇒ *lift* ((*ba* -) [*88*] *87*)
-wnext-d :: *lift* ⇒ *lift* ((*wnext* -) [*88*] *87*)
-wprev-d :: *lift* ⇒ *lift* ((*wprev* -) [*88*] *87*)
-more-d :: *lift* ((*more*))

syntax (*ASCII*)
-always-d :: *lift* ⇒ *lift* (([]-) [*88*] *87*)
-bi-d :: *lift* ⇒ *lift* ((*bi* -) [*88*] *87*)
-ba-d :: *lift* ⇒ *lift* ((*ba* -) [*88*] *87*)
-wnext-d :: *lift* ⇒ *lift* ((*wnext* -) [*88*] *87*)
-wprev-d :: *lift* ⇒ *lift* ((*wprev* -) [*88*] *87*)
-more-d :: *lift* ((*more*))

translations
-always-d ⇌ *CONST always-d*
-bi-d ⇌ *CONST bi-d*
-ba-d ⇌ *CONST ba-d*
-wnext-d ⇌ *CONST wnext-d*
-wprev-d ⇌ *CONST wprev-d*
-more-d ⇌ *CONST more-d*

definition *empty-d* :: (*'a*::*world*) *formula*
where *empty-d* ≡ *LIFT*(¬(*more*))

definition *dm-d* :: (*'a*::*world*) *formula* ⇒ *'a formula*
where *dm-d F* ≡ *LIFT*(#*True*;(*more* ∧ *F*))

syntax
-empty-d :: *lift* ((*empty*))
-dm-d :: *lift* ⇒ *lift* ((*dm* -) [*88*] *87*)

syntax (*ASCII*)
-empty-d :: *lift* ((*empty*))
-dm-d :: *lift* ⇒ *lift* ((*dm* -) [*88*] *87*)

translations
-empty-d ⇌ *CONST empty-d*
-dm-d ⇌ *CONST dm-d*

definition *bm-d* :: (*'a*::*world*) *formula* ⇒ *'a formula*
where *bm-d F* ≡ *LIFT*(¬(*dm*(¬*F*)))

definition *init-d* :: (*'a*::*world*) *formula* ⇒ *'a formula*
where *init-d F* ≡ *LIFT*((*empty* ∧ *F*);#*True*)

definition *fin-d* :: (*'a*::*world*) *formula* ⇒ *'a formula*
where *fin-d F* ≡ *LIFT*(□(*empty* ⟶ *F*))

definition *halt-d* :: (*'a*::*world*) *formula* ⇒ *'a formula*
where *halt-d F* ≡ *LIFT*(□(*empty* = *F*))

definition *initonly-d* :: (*'a*::*world*) *formula* ⇒ *'a formula*

where *initonly-d F* ≡ *LIFT*(*bi*(*empty* = *F*))

definition *keep-d* :: ('*a*::*world*) *formula* ⇒ '*a formula*
where *keep-d F* ≡ *LIFT*(*ba*(*skip* ⟶ *F*))

definition *yields-d* :: ('*a*::*world*) *formula* ⇒ '*a formula* ⇒ '*a formula*
where *yields-d F1 F2* ≡ *LIFT*(¬(*F1*;(¬*F2*)))

definition *ifthenelse-d* :: ('*a*::*world*) *formula* ⇒ '*a formula* ⇒ '*a formula* ⇒ '*a formula*
where *ifthenelse-d F G H* ≡ *LIFT*((*F* ∧ *G*) ∨ (¬*F* ∧ *H*))

primrec *power-chop-d* :: ('*a*::*world*) *formula* ⇒ *nat* ⇒ '*a formula*
where *power-0* : (*power-chop-d F 0*) = *LIFT*(*empty*)
| *power-Suc*: (*power-chop-d F* (*Suc n*)) = *LIFT*((*F* ∧ *more*);(*power-chop-d F n*))

primrec *len-d* :: *nat* ⇒ ('*a*::*world*) *formula*
where *len-0* : (*len-d 0*) = *LIFT*(*empty*)
| *len-Suc*: (*len-d* (*Suc n*)) = *LIFT*(*skip*;(*len-d n*))

primrec *power-d* :: ('*a*::*world*) *formula* ⇒ *nat* ⇒ '*a formula*
where *pow-0* : (*power-d F 0*) = *LIFT*(*empty*)
| *pow-Suc*: (*power-d F* (*Suc n*)) = *LIFT*((*F*);(*power-d F n*))

syntax
-bm-d :: *lift* ⇒ *lift* ((*bm* -) [*88*] *87*)
-init-d :: *lift* ⇒ *lift* ((*init* -) [*88*] *87*)
-fin-d :: *lift* ⇒ *lift* ((*fin* -) [*88*] *87*)
-halt-d :: *lift* ⇒ *lift* ((*halt* -) [*88*] *87*)
-initonly-d :: *lift* ⇒ *lift* ((*initonly* -) [*88*] *87*)
-keep-d :: *lift* ⇒ *lift* ((*keep* -) [*88*] *87*)
-yields-d :: [*lift*,*lift*] ⇒ *lift* ((- *yields* -) [*88*,*88*] *87*)
-ifthenelse-d :: [*lift*,*lift*,*lift*] ⇒ *lift* ((*if*$_i$ - *then* - *else* -) [*88*,*88*,*88*] *87*)
-len-d :: *nat* ⇒ *lift* ((*len* -) [*88*] *87*)
-power-chop-d :: [*lift*,*nat*] ⇒ *lift* ((*powerchop* - -) [*88*,*88*] *87*)
-power-d :: [*lift*,*nat*] ⇒ *lift* ((*power* - -) [*88*,*88*] *87*)

syntax (*ASCII*)
-bm-d :: *lift* ⇒ *lift* ((*bm* -) [*88*] *87*)
-init-d :: *lift* ⇒ *lift* ((*init* -) [*88*] *87*)
-fin-d :: *lift* ⇒ *lift* ((*fin* -) [*88*] *87*)
-halt-d :: *lift* ⇒ *lift* ((*halt* -) [*88*] *87*)
-initonly-d :: *lift* ⇒ *lift* ((*initonly* -) [*88*] *87*)
-keep-d :: *lift* ⇒ *lift* ((*keep* -) [*88*] *87*)
-yields-d :: [*lift*,*lift*] ⇒ *lift* ((- *yields* -) [*88*,*88*] *87*)
-ifthenelse-d :: [*lift*,*lift*,*lift*] ⇒ *lift* ((*if*$_i$ - *then* - *else* -) [*88*,*88*,*88*] *87*)
-len-d :: *nat* ⇒ *lift* ((*len* -) [*88*] *87*)
-power-chop-d :: [*lift*,*nat*] ⇒ *lift* ((*powerchop* - -) [*88*,*88*] *87*)
-power-d :: [*lift*,*nat*] ⇒ *lift* ((*power* - -) [*88*,*88*] *87*)

translations
-bm-d ⇌ *CONST bm-d*
-init-d ⇌ *CONST init-d*
-fin-d ⇌ *CONST fin-d*
-halt-d ⇌ *CONST halt-d*
-initonly-d ⇌ *CONST initonly-d*
-keep-d ⇌ *CONST keep-d*
-yields-d ⇌ *CONST yields-d*
-ifthenelse-d ⇌ *CONST ifthenelse-d*
-len-d ⇌ *CONST len-d*
-power-chop-d ⇌ *CONST power-chop-d*
-power-d ⇌ *CONST power-d*

definition *ifthen-d* :: (*'a*::*world*) *formula* ⇒ *'a formula* ⇒ *'a formula*
where *ifthen-d F G* ≡ *LIFT*(if_i *F then G else* #*True*)

definition *while-d* :: (*'a*::*world*) *formula* ⇒ *'a formula* ⇒ *'a formula*
where *while-d F G* ≡ *LIFT*((*F* ∧ *G*)* ∧ (*fin* ((¬*F*))))

syntax
-ifthen-d :: [*lift*,*lift*] ⇒ *lift* ((if_i *- then -*) [*88*,*88*] *87*)
-while-d :: [*lift*,*lift*] ⇒ *lift* ((*while - do -*) [*88*,*88*] *87*)

syntax (*ASCII*)
-ifthen-d :: [*lift*,*lift*] ⇒ *lift* ((if_i *- then -*) [*88*,*88*] *87*)
-while-d :: [*lift*,*lift*] ⇒ *lift* ((*while - do -*) [*88*,*88*] *87*)

translations
-ifthen-d ⇌ *CONST ifthen-d*
-while-d ⇌ *CONST while-d*

definition *repeat-d* :: (*'a*::*world*) *formula* ⇒ *'a formula* ⇒ *'a formula*
where *repeat-d F G* ≡ *LIFT*(*F*;*while* (¬ *G*) *do F*)

syntax
-repeat-d :: [*lift*,*lift*] ⇒ *lift* ((*repeat - until -*) [*88*,*88*] *87*)

syntax (*ASCII*)
-repeat-d :: [*lift*,*lift*] ⇒ *lift* ((*repeat - until -*) [*88*,*88*] *87*)

translations
-repeat-d ⇌ *CONST repeat-d*

definition *next-assign-d* :: (*'a*::*world*,*'b*) *stfun* ⇒ (*'a*,*'b*) *formfun* ⇒ *'a formula*
where *next-assign-d v e* ≡ *LIFT*(*v*$ = *e*)

definition *prev-assign-d* :: (*'a*::*world*,*'b*) *stfun* ⇒ (*'a*,*'b*) *formfun* ⇒ *'a formula*

where *prev-assign-d v e* ≡ *LIFT*(*v*! = *e*)

definition *always-eq-d* :: (*'a*::*world*,*'b*) *stfun* ⇒ (*'a*,*'b*) *formfun* ⇒ *'a formula*
where *always-eq-d v e* ≡ λ *s*. *s* ⊨ □($*v* = *e*)

definition *temporal-assign-d* :: (*'a*::*world*,*'b*) *stfun* ⇒ (*'a*,*'b*) *formfun* ⇒ *'a formula*
where *temporal-assign-d v e* ≡ λ *s*. *s* ⊨ !*v* = *e*

definition *gets-d* :: (*'a*::*world*,*'b*) *stfun* ⇒ (*'a*,*'b*) *formfun* ⇒ *'a formula*
where *gets-d v e* ≡ λ *s*. *s* ⊨ *keep*(*temporal-assign-d v e*)

definition *stable-d* :: (*'a*::*world*,*'b*) *stfun* ⇒ *'a formula*
where *stable-d v* ≡ λ *s*. *s* ⊨ *gets-d v* (*current-val-d v*)

definition *padded-d* :: (*'a*::*world*,*'b*) *stfun* ⇒ *'a formula*
where *padded-d v* ≡ λ *s*. *s* ⊨ (*stable-d v*);*skip* ∨ *empty*

definition *padded-temp-assign-d* :: (*'a*::*world*,*'b*) *stfun* ⇒ (*'a*,*'b*) *formfun* ⇒ *'a formula*
where *padded-temp-assign-d v e* ≡ λ *s*. *s* ⊨ (*temporal-assign-d v e*) ∧ (*padded-d v*)

syntax
-next-assign-d :: [*lift*,*lift*] ⇒ *lift* ((- := -) [*50*,*51*] *50*)
-prev-assign-d :: [*lift*,*lift*] ⇒ *lift* ((- =: -) [*50*,*51*] *50*)
-always-eq-d :: [*lift*,*lift*] ⇒ *lift* ((- ≈ -) [*50*,*51*] *50*)
-temporal-assign-d :: [*lift*,*lift*] ⇒ *lift* ((- ← -) [*50*,*51*] *50*)
-gets-d :: [*lift*,*lift*] ⇒ *lift* ((- *gets* -) [*50*,*51*] *50*)
-stable-d :: *lift* ⇒ *lift* ((*stable* -) [*51*] *50*)
-padded-d :: *lift* ⇒ *lift* ((*padded* -) [*51*] *50*)
-padded-temp-assign-d :: [*lift*,*lift*] ⇒ *lift* ((- <∼ -) [*50*,*51*] *50*)

syntax (*ASCII*)
-next-assign-d :: [*lift*,*lift*] ⇒ *lift* ((- := -) [*50*,*51*] *50*)
-prev-assign-d :: [*lift*,*lift*] ⇒ *lift* ((- =: -) [*50*,*51*] *50*)
-always-eq-d :: [*lift*,*lift*] ⇒ *lift* ((- *alweqv* -) [*50*,*51*] *50*)
-temporal-assign-d :: [*lift*,*lift*] ⇒ *lift* ((- <−− -) [*50*,*51*] *50*)
-gets-d :: [*lift*,*lift*] ⇒ *lift* ((- *gets* -) [*50*,*51*] *50*)
-stable-d :: *lift* ⇒ *lift* ((*stable* -) [*51*] *50*)
-padded-d :: *lift* ⇒ *lift* ((*padded* -) [*51*] *50*)
-padded-temp-assign-d :: [*lift*,*lift*] ⇒ *lift* ((- <~ -) [*50*,*51*] *50*)

translations
-next-assign-d ⇌ *CONST next-assign-d*
-prev-assign-d ⇌ *CONST prev-assign-d*
-always-eq-d ⇌ *CONST always-eq-d*
-temporal-assign-d ⇌ *CONST temporal-assign-d*
-gets-d ⇌ *CONST gets-d*
-stable-d ⇌ *CONST stable-d*
-padded-d ⇌ *CONST padded-d*
-padded-temp-assign-d ⇌ *CONST padded-temp-assign-d*

2.4 Properties of Operators

The following lemmas show that these operators have the expected semantics.

```
lemma skip-defs :
 (w ⊨ skip) = ( intlen w =1)
by (simp add: skip-d-def)
```

```
lemma chop-defs :
 (w ⊨ F1 ; F2) = (∃ n . 0≤n ∧ n≤intlen w ∧ ((prefix n w)⊨ F1) ∧ ( (suffix n w) ⊨ F2)  )
by (simp add: chop-d-def)
```

```
lemma sometimes-defs :
  (w ⊨ ◇ F) = (∃ n. 0≤n ∧ n≤intlen w ∧ ((suffix n w) ⊨ F))
by (simp add: Semantics.sometimes-d-def chop-defs)
```

```
lemma always-defs :
  (w ⊨ □ F) =(∀ n. 0≤n ∧ n≤intlen w ⟶ ((suffix n w) ⊨ F))
by (simp add: always-d-def sometimes-defs)
```

```
lemma di-defs :
  (w ⊨ di F) = (∃ n. 0≤n ∧ n≤intlen w ∧ ((prefix n w) ⊨ F))
by (simp add: Semantics.di-d-def chop-defs)
```

```
lemma bi-defs :
  (w ⊨ bi F) = (∀ n. 0≤n ∧ n≤intlen w ⟶ ((prefix n w) ⊨ F))
by (simp add: Semantics.bi-d-def di-defs)
```

```
lemma da-defs :
  (w ⊨ da F) = (∃ n na. 0≤n ∧  na+n ≤ intlen w ∧ ((sub n (na+n) w) ⊨ F))
apply (simp add: Semantics.da-d-def chop-defs)
using interval-prefix-length-good interval-suffix-length-good
by (smt add.commute add-diff-cancel-left' add-leD2 interval-sub-prefix-suffix-0 le-iff-add
        nat-add-left-cancel-le zero-le)
```

```
lemma ba-defs :
 (w ⊨ ba F) = (∀ n na. 0≤n ∧  na+n ≤ intlen w ⟶ ((sub n (na+n) w) ⊨ F))
by (simp add: ba-d-def da-defs)
```

```
lemma next-defs :
  (w ⊨ ○ F) = (intlen w > 0 ∧ ((suffix 1 w)⊨ F) )
apply (simp add: next-d-def chop-defs skip-defs)
using Suc-le-eq by force
```

```
lemma wnext-defs :
 (w ⊨ wnext F) = (intlen w = 0 ∨ ((suffix 1 w)⊨ F) )
by (simp add: wnext-d-def next-defs)
```

```
lemma prev-defs :
  (w ⊨ prev F) = (intlen w > 0 ∧ ((prefix ((intlen w)−1) w)⊨ F) )
by (simp add: prev-d-def chop-defs skip-defs)
```

```
 (metis One-nat-def Suc-leI diff-diff-cancel diff-is-0-eq′ diff-le-self
        interval-suffix-length-good neq0-conv zero-neq-one)

lemma wprev-defs :
  (w ⊨ wprev F) = (intlen w = 0 ∨ ((prefix ((intlen w)−1) w)⊨ F) )
by (metis (mono-tags, lifting) less-le prev-defs unl-lift wprev-d-def zero-le)

lemma more-defs :
  (w ⊨ more) = (intlen w>0)
by (simp add: more-d-def next-defs)

lemma empty-defs :
  (w ⊨ empty) = (intlen w = 0)
by (simp add: empty-d-def more-defs)

lemma init-defs :
  (w ⊨ init F) = ( (Interval.prefix 0 w) ⊨ F )
by (simp add: init-d-def empty-defs chop-defs) auto

lemma initalt-defs :
  (w ⊨ bi( empty ⟶ F)) = ( (Interval.prefix 0 w) ⊨ F )
by (simp add: bi-defs empty-defs)

lemma fin-defs :
  (w ⊨ fin F) = ( (Interval.suffix (intlen w) w) ⊨ F)
by (simp add: fin-d-def empty-defs always-defs)

lemma finalt-defs :
  (w ⊨ #True;(F ∧ empty)) = ( (Interval.suffix (intlen w) w) ⊨ F)
by (simp add: chop-defs empty-defs) fastforce

lemma halt-defs :
  (w ⊨ halt(F)) = (∀ n≤intlen w. (intlen w = n) = F (suffix n w))
by (simp add: halt-d-def empty-defs always-defs)

lemma initonly-defs :
  (w ⊨ initonly(F)) = (∀ n≤intlen w. (n = 0) = F (prefix n w))
by (simp add: initonly-d-def bi-defs empty-defs)

lemma ifthenelse-defs:
  (w ⊨ ifᵢ F then G else H) =
   ( ((w ⊨ F) ∧ (w ⊨ G)) ∨ (( ¬(w ⊨ F) ∧ (w ⊨ H))) )
by (simp add: ifthenelse-d-def)

lemma len-defs :
  (w ⊨ len n) = (intlen w =n)
by (induct n arbitrary: w, simp add: len-d-def empty-defs,
    simp add: len-d-def chop-defs skip-defs) fastforce

lemma currentval-defs :
  (s ⊨ $v) = (v (nth s 0))
by (simp add: current-val-d-def)

lemma nextval-defs :
 (s ⊨ v$) = (if intlen s >0 then (v (nth s 1)) else (ϵ x. x=x))
by (simp add: next-val-d-def)

lemma finval-defs :
  (s ⊨ !v) = (v (nth s (intlen s)))
by (simp add: fin-val-d-def)

lemma penultval-defs :
  (s ⊨ v!) = (if intlen s >0 then (v (nth s ((intlen s)−1))) else (ϵ x. x=x))
by (simp add: penult-val-d-def)

lemma next-assign-defs :
  intlen s >0 ⟹ (s ⊨ v := e) = v (Interval.nth s 1) = e s
by (auto simp: next-assign-d-def next-val-d-def)

lemma prev-assign-defs :
  intlen s >0 ⟹ (s ⊨ v =: e) = v (Interval.nth s ((intlen s)−1)) = e s
by (auto simp: prev-assign-d-def penult-val-d-def)

lemma always-eqv-defs :
  (s ⊨ v ≈ e) = (∀ i≤ intlen s. v (Interval.nth s i) = e (suffix i s))
by (simp add: always-eq-d-def always-defs current-val-d-def)

lemma temporal-assign-defs :
  (s ⊨ v ← e) = (v (Interval.nth s (intlen s)) = e s)
by (simp add: temporal-assign-d-def fin-val-d-def)

lemma gets-defs :
  (s ⊨ v gets e) = (∀ i < intlen s. v (Interval.nth s (Suc i)) = e (sub i (i+1) s) )
apply (simp add: gets-d-def keep-d-def ba-defs skip-defs sub-def temporal-assign-defs)
using Suc-le-eq by blast

lemma stable-defs-help:
 (∀ i<intlen s. v (Interval.nth s (Suc i)) = v (Interval.nth s i)) =
  (∀ i≤intlen s. v (Interval.nth s i) = v (Interval.nth s 0))
proof
  (induct s)
  case (St x)
  then show ?case by simp
  next
  case (Cons x1a s)
  then show ?case
  by (smt Suc-lessI interval-nth-Suc intlen.simps(2) le-SucE le-neq-implies-less le-simps(1)
          less-Suc-eq plus-1-eq-Suc zero-less-Suc)
qed
```

```
lemma stable-defs:
 (s ⊨ stable v) = (∀ i≤intlen s. (v (nth s i)) = (v (nth s 0)))
by (simp add: stable-d-def gets-defs current-val-d-def sub-def stable-defs-help)

lemma padded-defs :
  (s ⊨ padded v) = ((∀ i< intlen s. (v (nth s i)) = (v (nth s 0))) ∨ intlen s =0)
apply (simp add: padded-d-def stable-defs chop-d-def skip-defs empty-defs interval-suffix-length)
by (smt Suc-leI Suc-pred diff-diff-cancel interval-intlen-gr-zero le-neq-implies-less le-simps(1)
        less-Suc-eq)

lemma padded-temporal-assign-defs :
  (s ⊨ v <∼ e) =
  ((s ⊨ padded v) ∧
    (v (Interval.nth s (intlen s)) = e s ))
by (simp add: padded-temp-assign-d-def padded-defs temporal-assign-defs, auto)

lemma linalw:
  a ≤ b ∧ b≤intlen w ∧ ((suffix a w  )⊨ □ A) ⟶ ((suffix b w ) ⊨ □ A)
apply (simp add: always-defs)
by (smt add.assoc add.commute interval-suffix-length-good le-add-diff-inverse le-trans
        ordered-cancel-comm-monoid-diff-class.le-diff-conv2)
```

2.5 Soundness Axioms

2.5.1 ChopAssoc

```
lemma ChopAssocSemHelp:
 (∃ i ia . i≤intlen σ ∧ ia ≤ intlen σ −i ∧ (prefix i σ ⊨ f) ∧
    (prefix ia (suffix i σ) ⊨ g) ∧ (suffix (ia + i) σ ⊨ h)) =
  (∃ j ja . j ≤intlen σ ∧ ja ≤ j ∧ (prefix ja (prefix j σ) ⊨ f) ∧
    (suffix ja (prefix j σ) ⊨ g) ∧ (suffix j σ ⊨ h))
by (smt Nat.le-diff-conv2 add-diff-cancel-left′ interval-pref-pref-3 interval-suffix-prefix-swap
        le-add1 le-add-diff-inverse2 le-trans)

lemma ChopAssocSemHelp2:
  (σ ⊨  f ; (g ; h)) = (σ ⊨ (f;g);h)
proof −
 have (σ ⊨  f ; (g ; h)) =
        ((∃ i≤intlen σ. (prefix i σ ⊨ f) ∧ (∃ ia≤intlen (suffix i σ).
          (prefix ia (suffix i σ) ⊨ g) ∧ (suffix (ia + i) σ ⊨ h))))
 by (simp add: chop-defs)
 also have ... =
             (∃ i ia . i≤intlen σ ∧ ia ≤ intlen σ −i ∧ (prefix i σ ⊨ f) ∧
         (prefix ia (suffix i σ) ⊨ g) ∧ (suffix (ia + i) σ ⊨ h))
 by fastforce
 also have ... =
           (∃ j ja . j ≤intlen σ ∧ ja ≤ j ∧ (prefix ja (prefix j σ) ⊨ f) ∧
           (suffix ja (prefix j σ) ⊨ g) ∧ (suffix j σ ⊨ h))
 using ChopAssocSemHelp[of σ f g h]  by blast
 also have ... =
           (∃ i≤intlen σ. (∃ ia≤intlen (prefix i σ).  (prefix ia (prefix i σ) ⊨ f) ∧
```

(*suffix ia* (*prefix i σ*) ⊨ *g*)) ∧ (*suffix i σ* ⊨ *h*))
by *fastforce*
also have ... =
(σ ⊨ (*f*;*g*);*h*) **by** (*simp add*: *chop-defs*)
finally show (σ ⊨ *f* ; (*g* ; *h*)) = (σ ⊨ (*f*;*g*);*h*) **.**
qed

lemma *ChopAssocSem*:
(σ ⊨ *f* ; (*g* ; *h*) = (*f*;*g*);*h*)
using *ChopAssocSemHelp2* **using** *unl-lift2* **by** *blast*

2.5.2 OrChopImp

lemma *OrChopImpSem*:
(σ ⊨ (*f* ∨ *g*);*h* ⟶ *f*;*h* ∨ *g*;*h*)
by (*simp add*: *chop-defs*) *blast*

2.5.3 ChopOrImp

lemma *ChopOrImpSem*:
(σ ⊨ *f*;(*g* ∨ *h*) ⟶ *f*;*g* ∨ *f*;*h*)
by (*simp add*: *chop-defs*) *blast*

2.5.4 EmptyChop

lemma *EmptyChopSem*:
(σ ⊨ *empty* ; *f* = *f*)
by (*simp add*: *empty-defs chop-defs*) *auto*

2.5.5 ChopEmpty

lemma *ChopEmptySem*:
(σ ⊨ *f*;*empty* = *f*)
by (*simp add*: *empty-defs chop-defs*) *auto*

2.5.6 StateImpBi

lemma *StateImpBiSem*:
(σ ⊨ *init f* ⟶ *bi* (*init f*))
by (*simp add*: *init-defs bi-defs*)

2.5.7 NextImpNotNextNot

lemma *NextImpNotNextNotSem*:
(σ ⊨ ○ *f* ⟶ ¬ (○ (¬ *f*)))
by (*simp add*: *next-defs*)

2.5.8 BiBoxChopImpChop

lemma *BiBoxChopImpChopSem*:
(σ ⊨ *bi* (*f* ⟶ *f1*) ∧ □(*g* ⟶ *g1*) ⟶ *f*;*g* ⟶ *f1*;*g1*)
by (*simp add*: *bi-defs always-defs chop-defs*) *fastforce*

2.5.9 BoxInduct

```
lemma box-induct-help-1 :
  (σ ⊨ f) ∧ (∀ i. Suc 0 ≤ intlen σ − i ⟶
        i ≤ intlen σ ⟶ (suffix i σ ⊨ f) ⟶ (suffix (Suc i) σ ⊨ f))
     ⟹ (∀ j. j ≤ intlen σ ⟶ (suffix j σ ⊨ f))
proof
    fix j
    show (σ ⊨ f) ∧ (∀ i. Suc 0 ≤ intlen σ − i ⟶
         i ≤ intlen σ ⟶ (suffix i σ ⊨ f) ⟶ (suffix (Suc i) σ ⊨ f))
          ⟹ j ≤ intlen σ ⟶ (suffix j σ ⊨ f)
     proof
       (induct j arbitrary: σ)
       case 0
       then show ?case by simp
       next
       case (Suc j)
       then show ?case
       by (metis Nat.le-diff-conv2 One-nat-def Suc-eq-plus1-left Suc-leD)
       qed
qed

lemma BoxInductSem:
   (σ ⊨ □ (f ⟶ wnext f) ∧ f ⟶ □ f)
apply (simp add: always-defs wnext-defs)
using box-induct-help-1 by (metis One-nat-def diff-self-eq-0 not-one-le-zero)
```

2.5.10 ChopStarEqv

```
lemma chopstar-help-1:
  ( ∃ l. l= ⟨0⟩ ∧ index-sequence 0 l ∧
         Interval.nth l (intlen l) = (intlen σ) ∧
         (∀ i. (0≤i ∧ i< (intlen l)) ⟶
                              ((sub (nth l i) (nth l (i+1)) σ) ⊨ f)
                          )) ⟷ (intlen σ =0)
by (simp add: index-sequence-def)

lemma chopstar-help-2:
 (∀ i. (0<i ∧ i< 1+(intlen ls)) ⟶
                              ((sub (nth ls (i−1)) (nth ls ((i−1)+1)) σ) ⊨ f)
                          ) =
  (∀ i. (0≤i ∧ i< (intlen ls)) ⟶
                              ((sub (nth ls (i)) (nth ls ((i)+1)) σ) ⊨ f)
                 )
by (metis Suc-eq-plus1 Suc-pred add-diff-cancel-right' add-less-cancel-left
          add-nonneg-pos le-add2 le-add-same-cancel2 plus-1-eq-Suc zero-less-one)

lemma chop-power-chain:
  (∃ (l::index). (intlen l) = (Suc n) ∧ index-sequence 0 l ∧ (nth l (intlen l)) = (intlen σ) ∧
                          (∀ i. (0≤i ∧ i< (intlen l)) ⟶
                              ((sub (nth l i) (nth l (i+1)) σ) ⊨ f)
```

```
                    )
              ) =
   (∃ k. 0 ≤k ∧ k ≤ intlen σ ∧ k > 0 ∧
             (sub 0 k σ ⊨ f ) ∧
             (∃ ls. (intlen ls) = n ∧index-sequence 0 (ls) ∧
                   (nth (ls) (intlen (ls))) = (intlen (suffix k σ))
              ∧ (∀ i. (0≤i ∧ i< (intlen ls)) ⟶
                         ((sub (nth ls (i)) (nth ls ((i)+1)) (suffix k σ)) ⊨ f )
                ))
        )
proof −
 have (∃ (l::index). (intlen l) = (Suc n) ∧ index-sequence 0 l ∧
                 (nth l (intlen l)) = (intlen σ) ∧
                   (∀ i. (0≤i ∧ i< (intlen l)) ⟶
                      ((sub (nth l i) (nth l (i+1)) σ) ⊨ f )
                    )
              )
      =
      (∃ x ls l. (intlen l) = (Suc n) ∧ l=x⊙ls ∧ index-sequence 0 l ∧
               (nth l (intlen l)) = (intlen σ) ∧
                   (∀ i. (0≤i ∧ i< (intlen l)) ⟶
                      ((sub (nth l i) (nth l (i+1)) σ) ⊨ f )
                    )
             )
 using interval-intlen-cons-1 by (metis zero-less-Suc)
 also have ... =
           (∃ x ls l. (intlen l) = (Suc n) ∧ l=x⊙ls ∧ index-sequence 0 (x⊙ls) ∧
                  (nth (x⊙ls) (intlen (x⊙ls))) = (intlen σ) ∧
                   (∀ i. (0≤i ∧ i< (intlen (x⊙ls))) ⟶
                      ((sub (nth (x⊙ls) i) (nth (x⊙ls) (i+1)) σ) ⊨ f )
                    )
                 )
 by auto
 also have ... =
           (∃ x ls . (intlen ls) = n ∧ index-sequence 0 (x⊙ls) ∧
                 (nth (x⊙ls) (intlen (x⊙ls))) = (intlen σ) ∧
                   (∀ i. (0≤i ∧ i< (intlen (x⊙ls))) ⟶
                      ((sub (nth (x⊙ls) i) (nth (x⊙ls) (i+1)) σ) ⊨ f )
                    )
              )
 by auto
 also have ... =
           (∃ x ls . (intlen ls) = n ∧ x =0 ∧ index-sequence 0 (x ⊙ ls) ∧
                 (nth (ls) (intlen (ls))) = (intlen σ) ∧
                   ((∀ i. (0≤i ∧ i< (intlen (x⊙ls))) ⟶
                      ((sub (nth (x⊙ls) i) (nth (x⊙ls) (i+1)) σ) ⊨ f ))
                    )
               )
 by (simp add: index-sequence-def )
 also have ... =
          (∃ x ls . (intlen ls) = n ∧ x =0 ∧ index-sequence (nth ls 0) (ls) ∧
                     (nth (ls) (intlen (ls))) = (intlen σ) ∧
                    (x < (nth ls 0) ∧
                    ((∀ i. (0≤i ∧ i< (intlen (x⊙ls))) ⟶
                           ((sub (nth (x⊙ls) i) (nth (x⊙ls) (i+1)) σ) ⊨ f))
                        )
                    )
                )
using interval-idx-cons by auto
also have ... =
          (∃ x ls . (intlen ls) = n ∧ x =0 ∧ index-sequence (nth ls 0) (ls) ∧
                     (nth (ls) (intlen (ls))) = (intlen σ) ∧
                    (x < (nth ls 0) ∧
                     ((sub (nth (x⊙ls) 0) (nth (x⊙ls) (1)) σ) ⊨ f)
                    ∧
                    ((∀ i. (0<i ∧ i< 1+ (intlen (ls))) ⟶
                           ((sub (nth (x⊙ls) i) (nth (x⊙ls) (i+1)) σ) ⊨ f))
                        )
                    )
                )
by (metis (no-types, lifting) One-nat-def add.right-neutral add-Suc add-Suc-right
          add-cancel-right-left interval-intlen-cons not-gr-zero zero-le zero-less-Suc)
also have ... =
          (∃ x ls . (intlen ls) = n ∧ x =0 ∧ index-sequence (nth ls 0) (ls) ∧
                   (nth (ls) (intlen (ls))) = (intlen σ) ∧
                    (x < (nth ls 0) ∧ (nth (x⊙ls) 0) = x ∧ (nth (x⊙ls) (1)) = (nth ls 0) ∧
                     ((sub (nth (x⊙ls) 0) (nth (x⊙ls) (1)) σ) ⊨ f)
                    ∧
                    ((∀ i. (0<i ∧ i< 1+ (intlen (ls))) ⟶
                           ((sub (nth (x⊙ls) i) (nth (x⊙ls) (i+1)) σ) ⊨ f))
                        )
                    )
                )
by auto
also have ... =
          (∃ x ls . (intlen ls) = n ∧ x =0 ∧ index-sequence (nth ls 0) (ls) ∧
                       (nth (ls) (intlen (ls))) = (intlen σ) ∧
                    (x < (nth ls 0) ∧ (nth (x⊙ls) 0) = x ∧ (nth (x⊙ls) (1)) = (nth ls 0) ∧
                     ((sub x (nth ls 0) σ) ⊨ f)
                    ∧
                    ((∀ i. (0<i ∧ i< 1+ (intlen (ls))) ⟶
                           ((sub (nth (x⊙ls) i) (nth (x⊙ls) (i+1)) σ) ⊨ f))
                        )
                    )
                )
by auto
also have ... =
          (∃ x ls . (intlen ls) = n ∧ x =0 ∧ index-sequence (nth ls 0) (ls) ∧
                   (nth (ls) (intlen (ls))) = (intlen σ) ∧
                    (x < (nth ls 0) ∧
                     ((sub x (nth ls 0) σ) ⊨ f)
                    ∧
                    ((∀ i. (0<i ∧ i< 1+ (intlen (ls))) ⟶
                           ((sub (nth (x⊙ls) i) (nth (x⊙ls) (i+1)) σ) ⊨ f))
                        )
                    )
                )
by auto
also have ... =
          (∃ x ls . (intlen ls) = n ∧ x =0 ∧ index-sequence (nth ls 0) (ls) ∧
                   (nth (ls) (intlen (ls))) = (intlen σ) ∧
              ( x < (nth ls 0) ∧
              ((sub x (nth ls 0) σ) ⊨ f)
              ∧
               (∀ i. (0<i ∧ i< 1+(intlen ls)) ⟶
                           ((sub (nth ls (i−1)) (nth ls ((i−1)+1)) σ) ⊨ f)
                        )))
using interval-nth-cons by metis
also have ... =
          (∃ x ls . (intlen ls) = n ∧ x =0 ∧ index-sequence (nth ls 0) (ls) ∧
                   (nth (ls) (intlen (ls))) = (intlen σ) ∧
              (x < (nth ls 0) ∧
              ((sub x (nth ls 0) σ) ⊨ f))
              ∧ (∀ i. (0≤i ∧ i< (intlen ls)) ⟶
                           ((sub (nth ls (i)) (nth ls ((i)+1)) σ) ⊨ f)
                    )
               )
using chopstar-help-2 by (metis (mono-tags))
also have ... =
          (∃  ls . (intlen ls) = n ∧ index-sequence (nth ls 0) (ls) ∧
                     (nth (ls) (intlen (ls))) = (intlen σ) ∧
              (0 < (nth ls 0) ∧
              ((sub 0 (nth ls 0) σ) ⊨ f))
              ∧ (∀ i. (0≤i ∧ i< (intlen ls)) ⟶
                           ((sub (nth ls (i)) (nth ls ((i)+1)) σ) ⊨ f)
                    )
               )
by simp
also have ... =
          (∃  lsk .  (intlen lsk) = n ∧ (nth lsk 0) ≤ intlen σ ∧ (nth lsk 0) > 0 ∧
                ((sub 0 (nth lsk 0) σ) ⊨ f) ∧
                index-sequence (nth lsk 0) (lsk) ∧
                (nth (lsk) (intlen (lsk))) = (intlen σ) ∧
                (∀ i. (0≤i ∧ i< (intlen lsk)) ⟶
                           ((sub (nth lsk (i)) (nth lsk ((i)+1)) σ) ⊨ f)
                    )
               )
by (metis Suc-eq-plus1 Suc-pred add.left-neutral eq-iff interval-idx-less-last
          interval-intlen-gr-zero le-neq-implies-less lessI less-imp-le-nat)
also have ... =
          (∃ k lsk. (intlen lsk) = n ∧ (nth lsk 0) ≤ intlen σ ∧
                    (nth lsk 0) > 0 ∧ k=(nth lsk 0) ∧
                  (sub 0 (nth lsk 0) σ ⊨ f) ∧
                  index-sequence (nth lsk 0) (lsk) ∧
                  (nth (lsk) (intlen (lsk))) = (intlen (σ)) ∧
                  (∀ i. (0≤i ∧ i< (intlen lsk)) ⟶
                             ((sub ((nth lsk (i))) ((nth lsk ((i)+1))) (σ)) ⊨ f)
                  )
        )
by auto
also have ... =
          (∃ k lsk. (intlen lsk) = n ∧ 0 ≤k ∧ k ≤ intlen σ ∧ k > 0 ∧ k=(nth lsk 0) ∧
             (sub 0 k σ ⊨ f) ∧
             (index-sequence k (lsk) ∧
                       (nth (lsk) (intlen (lsk))) = ((intlen (suffix k σ))+k) ∧
                 (∀ i. (0≤i ∧ i< (intlen lsk)) ⟶
                            ((sub ((nth lsk (i))) ((nth lsk ((i)+1))) (σ)) ⊨ f)
                 ))
        )
apply (simp add: interval-prefix-suffix-intlen interval-suffix-length interval-prefix-length)
by auto
also have ... =
          (∃ k lsk. (intlen lsk) = n ∧  0 ≤k ∧ k ≤ intlen σ ∧ k > 0 ∧
             (sub 0 k σ ⊨ f) ∧
             (index-sequence k (lsk) ∧
                       (nth (lsk) (intlen (lsk))) = ((intlen (suffix k σ))+k) ∧
                 (∀ i. (0≤i ∧ i< (intlen lsk)) ⟶
                            ((sub ((nth lsk (i))) ((nth lsk ((i)+1))) (σ)) ⊨ f)
                 ))
        )
using index-sequence-def by auto
also have ... =
          (∃ k.   0 ≤k ∧ k ≤ intlen σ ∧ k > 0 ∧
             (sub 0 k σ ⊨ f) ∧
             (∃ ls lsk. (intlen lsk) = n ∧ index-sequence k (lsk) ∧
                        ls = map (shiftm k) lsk ∧
                       (nth (lsk) (intlen (lsk))) = ((intlen (suffix k σ))+k) ∧
                 (∀ i. (0≤i ∧ i< (intlen lsk)) ⟶
                            ((sub ((nth lsk (i))) ((nth lsk ((i)+1))) (σ)) ⊨ f)
                 ))
        )
by blast
also have ... =
          (∃ k.   0 ≤k ∧ k ≤ intlen σ ∧ k > 0 ∧
             (sub 0 k σ ⊨ f) ∧
             (∃ ls lsk. (intlen lsk) = n ∧ index-sequence k (lsk) ∧
                        ls = map (shiftm k) lsk ∧
                        index-sequence 0 (ls) ∧ (intlen ls) = n ∧
                       (nth (lsk) (intlen (lsk))) = ((intlen (suffix k σ))+k) ∧
                 (∀ i. (0≤i ∧ i< (intlen lsk)) ⟶
                    ((sub ((nth lsk (i))) ((nth lsk ((i)+1))) (σ)) ⊨ f)
               ))
        )
using interval-idx-link-shiftm by blast
also have ... =
          (∃ k.   0 ≤k ∧ k ≤ intlen σ ∧ k > 0 ∧
            (sub 0 k σ ⊨ f) ∧
            (∃ ls lsk. (intlen lsk) = n  ∧ index-sequence k (lsk) ∧
                     lsk = map (shift k) ls ∧
                     index-sequence 0 (ls) ∧ (intlen ls) = n  ∧
                     (nth (lsk) (intlen (lsk))) = ((intlen (suffix k σ))+k) ∧
               (∀ i. (0≤i ∧ i< (intlen lsk)) ⟶
                    ((sub ((nth lsk (i))) ((nth lsk ((i)+1))) (σ)) ⊨ f)
               ))
        )
using interval-lsk-ls by blast
also have ... =
          (∃ k ls lsk .   0 ≤k ∧ k ≤ intlen σ ∧ k > 0 ∧
            (sub 0 k σ ⊨ f) ∧
            ( (intlen lsk) = n  ∧ lsk = map (shift k) ls ∧
                      index-sequence 0 (ls) ∧
                      index-sequence k (lsk) ∧
                     (nth (ls) (intlen (ls))) = (intlen (suffix k σ)) ∧
               (∀ i. (0≤i ∧ i< (intlen ls)) ⟶
                    ((sub ((nth ls (i))+k) ((nth ls ((i)+1))+k) (σ)) ⊨ f)
               ))
        )
apply (simp add: Interval.shift-def interval-intlen-map interval-nth-map) by blast
also have ... =
          (∃ k ls lsk.   0 ≤k ∧ k ≤ intlen σ ∧ k > 0 ∧
            (sub 0 k σ ⊨ f) ∧
            ( (intlen lsk) = n  ∧ lsk = map (shift k) ls ∧
                     (intlen ls) = n  ∧ index-sequence 0 (ls) ∧
                     (nth (ls) (intlen (ls))) = (intlen (suffix k σ)) ∧
               (∀ i. (0≤i ∧ i< (intlen ls)) ⟶
                    ((sub ((nth ls (i))+k) ((nth ls ((i)+1))+k) (σ)) ⊨ f)
               ))
        )
using interval-idx-link by blast
also have ... =
          (∃ k. 0 ≤k ∧ k ≤ intlen σ ∧ k > 0 ∧
            (sub 0 k σ ⊨ f) ∧
            (∃ ls. (intlen ls) = n  ∧ index-sequence 0 (ls) ∧
                 (nth (ls) (intlen (ls))) = (intlen (suffix k σ)) ∧
               (∀ i. (0≤i ∧ i< (intlen ls)) ⟶
                    ((sub ((nth ls (i))+k) ((nth ls ((i)+1))+k) (σ)) ⊨ f)
               ))
        )
by (simp add: interval-intlen-map)
also have ... =
         (∃ k . 0 ≤k ∧ k ≤ intlen σ ∧ k > 0 ∧
            (sub 0 k σ |= f ) ∧
            (∃ ls. (intlen ls) = n  ∧ index-sequence 0 (ls) ∧
                  (nth (ls) (intlen (ls))) = (intlen (suffix k σ)) ∧
                (∀ i≤intlen ls. Interval.nth ls i ≤ intlen (suffix k σ)) ∧
                (∀ i. (0≤i ∧ i< (intlen ls)) −→
                         ((sub ((nth ls (i))+k) ((nth ls ((i)+1))+k) (σ)) |= f )
                )
            )
        )
using interval-idx-bound-1 by blast
also have  ... =
         (∃ k. 0 ≤k ∧ k ≤ intlen σ ∧ k > 0 ∧
            (sub 0 k σ |= f ) ∧
            (∃ ls. (intlen ls) = n  ∧ index-sequence 0 (ls) ∧
                  (nth (ls) (intlen (ls))) = (intlen (suffix k σ)) ∧
               (∀ i≤intlen ls. Interval.nth ls i ≤ intlen (suffix k σ))
              ∧ (∀ i. (0≤i ∧ i< (intlen ls)) −→
                         ((sub (nth ls (i)) (nth ls ((i)+1)) (suffix k σ)) |= f )
                )
            )
        )
by (smt add.commute index-sequence-def interval-idx-expand interval-sub-suffix
        interval-suffix-length-good plus-1-eq-Suc)
also have  ... =
         (∃ k. 0 ≤k ∧ k ≤ intlen σ ∧ k > 0 ∧
            (sub 0 k σ |= f ) ∧
            (∃ ls. (intlen ls) = n  ∧ index-sequence 0 (ls) ∧
                  (nth (ls) (intlen (ls))) = (intlen (suffix k σ))
              ∧ (∀ i. (0≤i ∧ i< (intlen ls)) −→
                         ((sub (nth ls (i)) (nth ls ((i)+1)) (suffix k σ)) |= f )
                ))
        )
using interval-idx-bound-1 by blast
finally show (∃ (l::index). (intlen l) = (Suc n) ∧ index-sequence 0 l ∧
               (nth l (intlen l)) = (intlen σ) ∧
               (∀ i. (0≤i ∧ i< (intlen l)) −→
                     ((sub (nth l i) (nth l (i+1)) σ) |= f )
               )
             ) =
             (∃ k. 0 ≤k ∧ k ≤ intlen σ ∧ k > 0 ∧ (sub 0 k σ |= f ) ∧
               (∃ ls. (intlen ls) = n ∧index-sequence 0 (ls) ∧
                  (nth (ls) (intlen (ls))) = (intlen (suffix k σ)) ∧
                  (∀ i. (0≤i ∧ i< (intlen ls)) −→
                     ((sub (nth ls (i)) (nth ls ((i)+1)) (suffix k σ)) |= f )
                  )
               )
             )
             .
qed
```

```
lemma chop-power-eqv-sem:
  (∃ n. (σ ⊨ (power-chop-d f n))) =
   ((σ ⊨ empty) ∨ (∃ n. (σ ⊨ (f ∧ more); (power-chop-d f n))))
by (metis not0-implies-Suc power-chop-d.power-0 power-chop-d.power-Suc)

lemma chopstar-eqv-power-chop-help:
   ( σ ⊨ power-chop-d f n) =
    (∃ (l::index). intlen(l) = n ∧  index-sequence 0 l  ∧
      (nth l (intlen l)) = (intlen ( σ)) ∧
      (∀ i. (0≤i ∧ i< (intlen l)) ⟶
            ((sub (nth l i) (nth l (i+1)) (σ)) ⊨ f)
      )
    )
proof
 (induct n arbitrary: σ)
 case 0
 then show ?case using index-sequence-def chopstar-help-1 empty-defs
 by (metis interval-intlen-st power-chop-d.power-0)
 next
 case (Suc n)
 then show ?case
  proof −
   have 1: (σ ⊨ power-chop-d f (Suc n)) = (σ ⊨ ((f ∧ more);(power-chop-d f n)))
   by simp
   have 2: (σ ⊨ ((f ∧ more);(power-chop-d f n))) =
            (∃ k. 0 ≤k ∧ k ≤ intlen (σ) ∧ k > 0 ∧
              (prefix k (σ) ⊨ f) ∧
              (suffix k (σ) ⊨ power-chop-d f n)
            )

  by (simp add: more-defs chop-defs) auto
  have 3: (∃ k. 0 ≤k ∧ k ≤ intlen (σ) ∧ k > 0 ∧
              (prefix k (σ) ⊨ f) ∧
              (suffix k (σ) ⊨ power-chop-d f n)
          ) =
          (∃ k. 0 ≤k ∧ k ≤ intlen (σ) ∧ k > 0 ∧
              (sub 0 k (σ) ⊨ f) ∧
              (suffix k (σ) ⊨ power-chop-d f n)
          )
  by (simp add: interval-sub-zero-prefix)
  have 4: (∃ k. 0 ≤k ∧ k ≤ intlen (σ) ∧ k > 0 ∧
              (sub 0 k (σ) ⊨ f) ∧
              (suffix k (σ) ⊨ power-chop-d f n)
          ) =
          (∃ k. 0 ≤k ∧ k ≤ intlen (σ) ∧ k > 0 ∧
              (sub 0 k (σ) ⊨ f) ∧
              (∃ (l::index). intlen(l) = n ∧  index-sequence 0 l  ∧
                 (nth l (intlen l)) = (intlen (suffix k σ)) ∧
                 (∀ i. (0≤i ∧ i< (intlen l)) ⟶
                ((sub (nth l i) (nth l (i+1)) (suffix k σ)) ⊨ f)
             )
           )
         )
    using Suc.hyps by auto
    have 5:
        (∃ (l::index). (intlen l) = (Suc n) ∧ index-sequence 0 l ∧
          (nth l (intlen l)) = (intlen σ) ∧
          (∀ i. (0≤i ∧ i< (intlen l)) ⟶
               ((sub (nth l i) (nth l (i+1)) σ) ⊨ f)
          )
        ) =
        (∃ k. 0 ≤k ∧ k ≤ intlen σ ∧ k > 0 ∧
           (sub 0 k σ ⊨ f) ∧
           (∃ ls. (intlen ls) = n ∧index-sequence 0 (ls) ∧
              (nth (ls) (intlen (ls))) = (intlen (suffix k σ)) ∧
              (∀ i. (0≤i ∧ i< (intlen ls)) ⟶
                   ((sub (nth ls (i)) (nth ls ((i)+1)) (suffix k σ)) ⊨ f)
              )
           )
        )
     using chop-power-chain by simp
     from 1 2 3 4 5 show ?thesis by auto
   qed
qed

lemma chopstar-eqv-power-chop:
  (σ ⊨ f⋆) = (∃ k. (σ ⊨ power-chop-d f k))
by (simp add: chopstar-d-def chopstar-eqv-power-chop-help)

lemma ChopstarEqvSem:
   (σ ⊨ (f⋆ =  (empty ∨  (f ∧ more); (f⋆))) )
using chopstar-eqv-power-chop
by (smt chop-d-def chop-power-eqv-sem unl-lift2)
```

2.6 Quantification over State (Flexible) Variables

The hidden state approach, as used in the embedding of TLA in Isabelle/HOL TLA embedding [3, 2], is used. Here [3, 2], a state space is defined by its projections, and everything else is unknown. Thus, a variable is a projection of the state space, and has the same type as a state function. Moreover, strong typing is achieved, since the projection function may have any result type. To achieve this, the state space is represented by an undefined type, which is an instance of the *world* class to enable use with the *Intensional* theory.

```
typedecl state

instance state :: world ..

type-synonym 'a statefun = (state,'a) stfun
type-synonym statepred  = bool statefun
```

type-synonym *'a tempfun* = (*state*,*'a*) *formfun*
type-synonym *temporal* = *state formula*

Similar to [3, 2] we define a state to be an anonymous type whose only purpose is to provide Skolem constants. Similarly, we do not define a type of state variables separate from that of arbitrary state functions, again in order to simplify the definition of flexible quantification later on. Nevertheless, we need to distinguish state variables. Note we deviate from [3, 2] in that we do not use axioms but use definitions and lemmas.

2.7 Temporal Quantifiers

definition *exist-state-d* :: (*'a statefun* ⇒ *temporal*) ⇒ *temporal* (**binder** *Eex* *10*)
where *exist-state-d* *F* ≡ (λ*s*. (∃ *x*. *s* ⊨ *F x*))

syntax
-Eex :: [*idts*, *lift*] ⇒ *lift* ((*3*∃∃ -./ -) [*0*,*10*] *10*)

translations
-Eex v A == *Eex v*. *A*

definition *forall-state-d* :: (*'a statefun* ⇒ *temporal*) ⇒ *temporal* (**binder** *Aall* *10*)
where *forall-state-d F* ≡ *LIFT*(¬(∃∃ *x*. ¬(*F x*)))

syntax
-Aall :: [*idts*, *lift*] ⇒ *lift* ((*3*∀∀ -./ -) [*0*,*10*] *10*)

translations
-Aall v A == *Aall v*. *A*

2.8 Unlifting attributes and methods

The following is again from [3, 2] but adapted for our need.

lemma *int-eq-true*: ⊢ *P* ⟹ ⊢ *P* = #*True*
by *auto*

lemma *int-eq*: ⊢ *X* = *Y* ⟹ *X* = *Y*
by (*auto simp*: *inteq-reflection*)

lemma *int-iffI*:
assumes ⊢ *F* ⟶ *G* **and** ⊢ *G* ⟶ *F*
shows ⊢ *F* = *G*
using *assms* **by** *force*

lemma *int-iffD1*: **assumes** *h*: ⊢ *F* = *G* **shows** ⊢ *F* ⟶ *G*
using *h* **by** *auto*

lemma *int-iffD2*: **assumes** *h*: ⊢ *F* = *G* **shows** ⊢ *G* ⟶ *F*
using *h* **by** *auto*

```
lemma lift-imp-trans:
  assumes ⊢ A ⟶ B and ⊢ B ⟶ C
  shows ⊢ A ⟶ C
  using assms by force

lemma lift-imp-neg: assumes ⊢ A ⟶ B shows ⊢ ¬B ⟶ ¬A
  using assms by auto

lemma lift-and-com: ⊢ (A ∧ B) = (B ∧ A)
  by auto
```

Attribute which unlifts an intensional formula

```
ML ‹
fun unl-rewr ctxt thm =
    let
      val unl = (thm RS @{thm intD})
                                  handle THM - => thm
      val rewr = rewrite-rule ctxt @{thms intensional-rews}
     in
      unl |> rewr
     end;
›
attribute-setup unlifted = ‹
  Scan.succeed (Thm.rule-attribute [] (unl-rewr o Context.proof-of))
› unlift intensional formulas

attribute-setup unlift-rule = ‹
  Scan.succeed
    (Thm.rule-attribute []
      (Context.proof-of #> (fn ctxt => Object-Logic.rulify ctxt o unl-rewr ctxt)))
› unlift and rulify intensional formulas
```

Attribute which turns an intensional formula into a rewrite rule. Formulas F that are not equalities are turned into $F \equiv \#True$.

```
ML ‹
fun int-rewr thm =
   (thm RS @{thm inteq-reflection})
     handle THM - => ((thm RS @{thm int-eq-true}) RS @{thm inteq-reflection});
›

attribute-setup simp-unl = ‹
    Attrib.add-del
      (Thm.declaration-attribute
        (fn th => Simplifier.map-ss (Simplifier.add-simp (int-rewr th))))
      (K (NONE, NONE)) — note only adding – removing is ignored
› add thm unlifted from rewrites from intensional formulas

attribute-setup int-rewrite = ‹ Scan.succeed (Thm.rule-attribute [] (fn - => int-rewr)) ›
  produce rewrites from intensional formulas
```

end

theory *ITL*
imports
Semantics
begin

3 Axioms and Rules

The ITL axiom and proof rules are introduced (taken from [4]) together with the validity operation. The soundness of the rules and axioms are checked using the lemmas of Semantics.thy.

3.1 Rules

lemma *MP* :
assumes ⊢ *f* ⟶ *g*
⊢ *f*
shows ⊢ *g*
using *assms*(*1*) *assms*(*2*) **by** *fastforce*

lemma *BoxGen* :
assumes ⊢ *f*
shows ⊢ □ *f*
using *assms* **by** (*auto simp*: *always-defs*)

lemma *BiGen*:
assumes ⊢ *f*
shows ⊢ *bi f*
using *assms* **by** (*auto simp*: *bi-defs*)

3.2 Axioms

lemma *ChopAssoc* :
⊢ *f* ; (*g* ; *h*) = (*f*;*g*);*h*
using *ChopAssocSem Valid-def* **by** *blast*

lemma *OrChopImp* :
⊢ (*f* ∨ *g*);*h* ⟶ *f*;*h* ∨ *g*;*h*
using *OrChopImpSem Valid-def* **by** *blast*

lemma *ChopOrImp* :
⊢ *f*;(*g* ∨ *h*) ⟶ *f*;*g* ∨ *f*;*h*
using *ChopOrImpSem Valid-def* **by** *blast*

lemma *EmptyChop* :
⊢ *empty* ; *f* = *f*
using *EmptyChopSem Valid-def* **by** *blast*

lemma *ChopEmpty* :
$\vdash f;empty = f$
using *ChopEmptySem Valid-def* **by** *blast*

lemma *StateImpBi* :
$\vdash init\ f \longrightarrow\ bi\ (init\ f)$
using *StateImpBiSem Valid-def* **by** *blast*

lemma *NextImpNotNextNot* :
$\vdash \bigcirc f \longrightarrow \neg (\bigcirc (\neg f))$
using *NextImpNotNextNotSem Valid-def* **by** *blast*

lemma *BiBoxChopImpChop* :
$\vdash bi\ (f \longrightarrow f1) \wedge \Box(g \longrightarrow g1) \longrightarrow f;g \longrightarrow f1;g1$
using *BiBoxChopImpChopSem Valid-def* **by** *blast*

lemma *BoxInduct* :
$\vdash \Box\ (f \longrightarrow wnext\ f) \wedge f \longrightarrow \Box\ f$
using *BoxInductSem Valid-def* **by** *blast*

lemma *ChopstarEqv* :
$\vdash f^{\star} = (empty \vee (f \wedge more);\ f^{\star})$
using *ChopstarEqvSem Valid-def* **by** *blast*

3.3 Quantification

lemma *EExI* :
$\vdash F\ y \longrightarrow (\exists\exists\ x\ .\ F\ x)$
by (*simp add*: *exist-state-d-def Valid-def*, *auto*)

lemma *EExE*:
$[\![\bigwedge x.\ \vdash F\ x \longrightarrow G]\!] \Longrightarrow \vdash (\exists\exists\ x.\ F\ x) \longrightarrow G$
by (*metis* (*mono-tags*, *lifting*) *Valid-def exist-state-d-def unl-lift2*)

lemma *EExVal*:
$(w \models (\exists\exists\ x.\ F\ x)) =$
$(\exists\ x\ (val :: 'a\ interval).\ (\ (\ val = (map\ x\ w) \wedge\ (w \models F\ x))))$
by (*simp add*: *exist-state-d-def*)

lemma *AAxDef*:
$\vdash (\forall\forall\ x.\ F\ x) = (\neg(\exists\exists\ x.\ \neg(F\ x)))$
by (*simp add*: *Valid-def forall-state-d-def exist-state-d-def*)

lemma *EExRev* :
$\vdash (\exists\exists\ x.\ F\ x)^{r} = (\exists\exists\ x.\ (F\ x)^{r})$
by (*simp add*: *Valid-def exist-state-d-def reverse-d-def*)

lemma *ExEqvRule*:
assumes $\bigwedge\ x.\ \vdash (f\ x) = (g\ x)$

shows ⊢ (∃ *x*. *f x*) = (∃ *x*. *g x*)
using *assms* **by** *fastforce*

3.4 Lemmas about *current-val*

lemma *current-const*: ⊢ $(#*c*) = #*c*
by (*auto simp*: *current-val-d-def*)

lemma *current-fun1*: ⊢ $(*f*<*x*>) = *f* <$*x*>
by (*auto simp*: *current-val-d-def*)

lemma *current-fun2*: ⊢ $(*f*<*x*,*y*>) = *f* <$*x*,$*y*>
by (*auto simp*: *current-val-d-def*)

lemma *current-fun3*: ⊢ $(*f*<*x*,*y*,*z*>) = *f* <$*x*,$*y*,$*z*>
by (*auto simp*: *current-val-d-def*)

lemma *current-forall*: ⊢ $(∀ *x*. *P x*) = (∀ *x*. $(*P x*))
by (*auto simp*: *current-val-d-def*)

lemma *current-exists*: ⊢ $(∃ *x*. *P x*) = (∃ *x*. $(*P x*))
by (*auto simp*: *current-val-d-def*)

lemma *current-exists1*: ⊢ $(∃! *x*. *P x*) = (∃! *x*. $(*P x*))
by (*auto simp*: *current-val-d-def*)

lemmas *all-current* = *current-const current-fun1 current-fun2 current-fun3*
current-forall current-exists current-exists1

lemmas *all-current-unl* = *all-current*[*THEN intD*]
lemmas *all-current-eq* = *all-current*[*THEN inteq-reflection*]

3.5 Lemmas about *next-val*

lemma *next-const*: ⊢ *more* ⟶ (#*c*)$ = #*c*
by (*auto simp*: *next-val-d-def more-defs*)

lemma *next-fun1*: ⊢ *more* ⟶ *f*<*x*>$ = *f*<*x*$>
by (*auto simp*: *next-val-d-def more-defs*)

lemma *next-fun2*: ⊢ *more* ⟶ *f*<*x*,*y*>$ = *f* <*x*$,*y*$>
by (*auto simp*: *next-val-d-def more-defs*)

lemma *next-fun3*: ⊢ *more* ⟶ *f*<*x*,*y*,*z*>$ = *f* <*x*$,*y*$,*z*$>
by (*auto simp*: *next-val-d-def more-defs*)

lemma *next-forall*: ⊢ *more* ⟶ (∀ *x*. *P x*)$ = (∀ *x*. (*P x*)$)
by (*auto simp*: *next-val-d-def*)

lemma *next-exists*: ⊢ *more* ⟶ (∃ *x*. *P x*)$ = (∃ *x*. (*P x*)$)

by (*auto simp*: *next-val-d-def*)

lemma *next-exists1*: ⊢ *more* ⟶ (∃! *x*. *P x*)$ = (∃! *x*. (*P x*)$)
by (*auto simp*: *next-val-d-def more-defs*)

lemmas *all-next* = *next-const next-fun1 next-fun2 next-fun3*
next-forall next-exists next-exists1

lemmas *all-next-unl* = *all-next*[*THEN intD*]

3.6 Lemmas about *fin-val*

lemma *fin-const*: ⊢ !(#*c*) = #*c*
by (*auto simp*: *fin-val-d-def*)

lemma *fin-fun1*: ⊢ !(*f*<*x*>) = *f* <!*x*>
by (*auto simp*: *fin-val-d-def*)

lemma *fin-fun2*: ⊢ !(*f*<*x*,*y*>) = *f* <!*x*, !*y*>
by (*auto simp*: *fin-val-d-def*)

lemma *fin-fun3*: ⊢ !(*f*<*x*,*y*,*z*>) = *f* <!*x*,!*y*,!*z*>
by (*auto simp*: *fin-val-d-def*)

lemma *fin-forall*: ⊢ !(∀ *x*. *P x*) = (∀ *x*. !(*P x*))
by (*auto simp*: *fin-val-d-def*)

lemma *fin-exists*: ⊢ !(∃ *x*. *P x*) = (∃ *x*. !(*P x*))
by (*auto simp*: *fin-val-d-def*)

lemma *fin-exists1*: ⊢ !(∃! *x*. *P x*) = (∃! *x*. !(*P x*))
by (*auto simp*: *fin-val-d-def*)

lemmas *all-fin* = *fin-const fin-fun1 fin-fun2 fin-fun3*
fin-forall fin-exists fin-exists1

lemmas *all-fin-unl* = *all-fin*[*THEN intD*]
lemmas *all-fin-eq* = *all-fin*[*THEN inteq-reflection*]

3.7 Lemmas about *penult-val*

lemma *penult-const*: ⊢ *more* ⟶ (#*c*)! = #*c*
by (*auto simp*: *penult-val-d-def more-defs*)

lemma *penult-fun1*: ⊢ *more* ⟶ *f*<*x*>! = *f*<*x*!>
by (*auto simp*: *penult-val-d-def more-defs*)

lemma *penult-fun2*: ⊢ *more* ⟶ *f*<*x*,*y*>! = *f* <*x*!,*y*!>
by (*auto simp*: *penult-val-d-def more-defs*)

lemma *penult-fun3*: $\vdash more \longrightarrow f{<}x,y,z{>}! = f\ {<}x!,y!,z!{>}$
by (*auto simp*: *penult-val-d-def more-defs*)

lemma *penult-forall*: $\vdash more \longrightarrow (\forall\ x.\ P\ x)! = (\forall\ x.\ (P\ x)!)$
by (*auto simp*: *penult-val-d-def*)

lemma *penult-exists*: $\vdash more \longrightarrow (\exists\ x.\ P\ x)! = (\exists\ x.\ (P\ x)!)$
by (*auto simp*: *penult-val-d-def*)

lemma *penult-exists1*: $\vdash more \longrightarrow (\exists!\ x.\ P\ x)! = (\exists!\ x.\ (P\ x)!)$
by (*auto simp*: *penult-val-d-def more-defs*)

lemmas *all-penult* = *penult-const penult-fun1 penult-fun2 penult-fun3*
penult-forall penult-exists penult-exists1

lemmas *all-penult-unl* = *all-penult*[*THEN intD*]

3.8 Basic temporal variables properties

lemma *empty-imp-fin-eqv-curr*:
$\vdash empty \longrightarrow\ !v = \$v$
by (*simp add*: *Valid-def current-val-d-def empty-defs finval-defs*)

lemma *skip-imp-fin-eqv-next*:
$\vdash skip \longrightarrow\ !v = v\$$
by (*simp add*: *Valid-def skip-defs next-val-d-def finval-defs*)

lemma *skip-imp-penult-eqv-curr*:
$\vdash skip \longrightarrow v! = \$v$
by (*simp add*: *Valid-def skip-defs penultval-defs current-val-d-def*)

3.9 Time reversal properties

lemma *rev-const* :
$\vdash (\#c)^r = \#c$
by (*auto simp*: *reverse-d-def*)

lemma *rev-fun1* :
$\vdash (f{<}x{>})^r = f{<}x^r{>}$
by (*auto simp*: *reverse-d-def*)

lemma *rev-fun2*:
$\vdash (f{<}x,y{>})^r = f{<}x^r,y^r{>}$
by (*auto simp*: *reverse-d-def*)

lemma *rev-fun3*:
$\vdash (f{<}x,y,z{>})^r = f{<}x^r,y^r,z^r{>}$
by (*auto simp*: *reverse-d-def*)

lemma *rev-forall*:

$\vdash (\forall\ x.\ P\ x)^r = (\forall\ x.\ (P\ x)^r)$
by (*auto simp*: *reverse-d-def*)

lemma *rev-exists*:
$\vdash (\exists\ x.\ P\ x)^r = (\exists\ x.\ (P\ x)^r)$
by (*auto simp*: *reverse-d-def*)

lemma *rev-exists1*:
$\vdash (\exists!\ x.\ P\ x)^r = (\exists!\ x.\ (P\ x)^r)$
by (*auto simp*: *reverse-d-def*)

lemma *rev-current*:
$\vdash (\$v)^r = (!v)$
by (*auto simp*: *interval-intrev-nth current-val-d-def fin-val-d-def reverse-d-def*)

lemma *rev-next*:
$\vdash\ (v\$)^r = (v!)$
by (*auto simp*: *interval-intrev-nth next-val-d-def penult-val-d-def reverse-d-def*)

lemma *rev-penult*:
$\vdash\ (v!)^r = (v\$)$
by (*auto simp*: *interval-intrev-nth next-val-d-def penult-val-d-def reverse-d-def*)

lemma *rev-fin*:
$\vdash (!v)^r = (\$v)$
by (*auto simp*: *interval-intrev-nth fin-val-d-def current-val-d-def reverse-d-def*)

lemma *EqvReverseReverse*:
$\vdash (f^r)^r = f$
by (*simp add*: *Valid-def reverse-d-def*)

lemma *ReverseEqv*:
$(\vdash f) \longleftrightarrow (\vdash f^r)$
by (*metis Valid-def interval-rev-swap reverse-d-def*)

lemma *RevSkip*:
$\vdash skip^r = skip$
by (*simp add*: *Valid-def reverse-d-def skip-defs*)

lemma *RevChop*:
$\vdash (f;g)^r = (g^r;f^r)$
apply (*simp add*: *Valid-def reverse-d-def chop-d-def*)
using *interval-intrev-prefix interval-intrev-suffix*
by (*metis diff-diff-cancel diff-le-self*)

lemma *RMoreEqvMore*:
$\vdash more^r = more$
apply (*simp add*: *Valid-def more-d-def next-d-def chop-d-def skip-d-def reverse-d-def*)
by (*simp add*: *interval-prefix-length*)

lemma *REmptyEqvEmpty*:
$\vdash$ *empty*^{r} = *empty*
by (*metis RMoreEqvMore empty-d-def int-eq rev-fun1*)

lemma *PowerChopCommute*:
$\vdash$ ((*f* $\wedge$ *more*);(*powerchop f n*)) = (*powerchop f n*);(*f* $\wedge$ *more*)
proof
(*induct n*)
case *0*
then show *?case* **using** *EmptyChopSem ChopEmptySem power-0 Valid-def* **by** (*metis inteq-reflection*)
next
case (*Suc n*)
then show *?case*
by (*metis ChopAssocSem intI inteq-reflection power-chop-d.power-Suc*)
qed

lemma *REqvRule*:
assumes $\vdash$ *f* =*g*
shows $\vdash$ (f^r) = (g^r)
using *assms*
using *inteq-reflection* **by** *force*

lemma *RevPowerChop*:
$\vdash$ (*powerchop f n*)^r = (*powerchop* (f^r) *n*)
proof
(*induct n*)
case *0*
then show *?case* **using** *REmptyEqvEmpty* **by** *auto*
next
case (*Suc n*)
then show *?case*
by (*metis PowerChopCommute RevChop RMoreEqvMore int-eq power-chop-d.power-Suc rev-fun2*)
qed

lemma *RevChopstar*:
$\vdash$ $(f^\star)^r = (f^r)^\star$
proof –
have *1*: $\vdash$ ($f^\star$) = ($\exists$ *n*. *powerchop f n*)
  by (*simp add*: *chopstar-eqv-power-chop Valid-def*)
have *2*: $\vdash$ $(f^\star)^r$ = ($\exists$ *n*. *powerchop f n*)^r
  using *REqvRule 1* **by** *blast*
have *3*: $\vdash$ ($\exists$ *n*. *powerchop f n*)^r = ($\exists$ *n*. (*powerchop f n*)^r)
  by (*simp add*: *rev-exists*)
have *4*: $\vdash$ ($\exists$ *n*. (*powerchop f n*)^r) = ($\exists$ *n*. (*powerchop* (f^r) *n*))
  by (*simp add*: *RevPowerChop ExEqvRule*)
have *5*: $\vdash$ ($\exists$ *n*. (*powerchop* (f^r) *n*)) = $(f^r)^\star$
  by (*simp add*: *chopstar-eqv-power-chop Valid-def*)
from *2 3 4 5* **show** *?thesis* **by** *fastforce*
qed

lemmas *all-rev* = *rev-const rev-fun1 rev-fun2 rev-fun3 rev-forall rev-exists*
rev-exists1 rev-current rev-next rev-penult rev-fin RevSkip RevChop RevChopstar

lemmas *all-rev-unl* = *all-rev*[*THEN intD*]
lemmas *all-rev-eq* = *all-rev*[*THEN inteq-reflection*]

end

theory *Theorems*
imports
ITL
begin

4 ITL theorems

We give the proofs of a list of ITL theorems. These proofs and theorems were from [5].

4.1 Propositional reasoning

This is a list of propositional logic theorems used in the proofs of the ITL theorems.

lemma *IfThenElseImp*:
⊢ (*if*$_i$ *g then f else f1*) ⟶ ((*g* ⟶ *f*) ∧ (¬*g* ⟶ *f1*))
by (*simp add*: *ifthenelse-defs Valid-def*)

lemma *Prop01*:
assumes ⊢ *f* ⟶ ¬ *g* ∨ *h*
shows ⊢ *g* ∧ *f* ⟶ *h*
using *assms* **by** *auto*

lemma *Prop02*:
assumes ⊢ *f* ⟶ *g*
⊢ *f1* ⟶ *g*
shows ⊢ *f* ∨ *f1* ⟶ *g*
using *assms*(*1*) *assms*(*2*) **by** *fastforce*

lemma *Prop03*:
assumes ⊢ *f* = (*g* ∨ *h*)
shows ⊢ *h* ⟶ *f*
using *assms* **by** *auto*

lemma *Prop04*:
assumes ⊢ *f* = *h*
⊢ *f* = *h1*
shows ⊢ *h1* = *h*
using *assms*(*1*) *assms*(*2*) **using** *int-eq* **by** *auto*

lemma *Prop05*:
assumes $\vdash f \longrightarrow g$
shows $\vdash f \longrightarrow h \lor g$
using *assms* **by** *auto*

lemma *Prop06*:
assumes $\vdash f = (g \lor h)$
$\vdash h = h1$
shows $\vdash f = (g \lor h1)$
using *assms*(1) *assms*(2) **by** *fastforce*

lemma *Prop07*:
assumes $\vdash f \longrightarrow g \lor h$
shows $\vdash f \land \neg g \longrightarrow h$
using *assms* **by** *auto*

lemma *Prop08*:
assumes $\vdash f \longrightarrow g \lor h$
$\vdash h \longrightarrow h1$
shows $\vdash f \longrightarrow g \lor h1$
using *assms*(1) *assms*(2) **by** *fastforce*

lemma *Prop09*:
assumes $\vdash f \land g \longrightarrow h$
shows $\vdash f \longrightarrow (g \longrightarrow h)$
using *assms* **by** *auto*

lemma *Prop10*:
assumes $\vdash f \longrightarrow g$
shows $\vdash f = (f \land g)$
using *assms* **by** *auto*

lemma *Prop11*:
$(\vdash f = f1) = ((\vdash f \longrightarrow f1) \land (\vdash f1 \longrightarrow f))$
by (*auto simp*: *Valid-def*)

lemma *Prop12*:
$(\vdash f \longrightarrow (f1 \land f2)) = ((\vdash f \longrightarrow f1) \land (\vdash f \longrightarrow f2))$
by (*auto simp*: *Valid-def*)

4.2 State formulas

The *init* operator denotes state formulas, i.e., ITL formula that only constrain the first state of an interval.

lemma *Initprop* :
$\vdash ((init\ f) \land (init\ g)) = init(f \land g)$
$\vdash (\neg (init\ f)) = init\ (\neg f)$
$\vdash ((init\ f) \lor (init\ g)) = init\ (f \lor g)$
$\vdash init\ \#True$
by (*auto simp*: *init-defs*)

lemma *Finprop* :
$\vdash ((\#True;(f \wedge empty)) \wedge (\#True;(g \wedge empty))) = (\#True;((f \wedge g) \wedge empty))$
$\vdash ((\#True;(f \wedge empty)) \vee (\#True;(g \wedge empty))) = (\#True;((f \vee g) \wedge empty))$
$\vdash (\#True;((\#True) \wedge empty))$
$\vdash (\neg (\#True;(f \wedge empty))) = (\#True;(\neg f \wedge empty))$
by (*auto simp*: *finalt-defs*) (*simp add*: *chop-defs empty-defs interval-suffix-length*, *fastforce*)

4.3 Basic Theorems

lemma *BiChopImpChop* :
$\vdash bi\ (f \longrightarrow f1) \longrightarrow f;g \longrightarrow f1;g$
proof −
have *1*: $\vdash g \longrightarrow g$ **by** *auto*
hence *2*: $\vdash \Box\ (\ g \longrightarrow g)$ **by** (*rule BoxGen*)
have *3*: $\vdash bi\ (\ f \longrightarrow f1) \wedge \Box(g \longrightarrow g) \longrightarrow f;g \longrightarrow f1;g$ **by** (*rule BiBoxChopImpChop*)
from *2 3* **show** *?thesis* **by** *fastforce*
qed

lemma *AndChopA*:
$\vdash (f \wedge f1);g \longrightarrow f;g$
proof −
have *1*: $\vdash f \wedge f1 \longrightarrow f$ **by** *auto*
hence *2*: $\vdash bi\ (f \wedge f1 \longrightarrow f)$ **by** (*rule BiGen*)
have *3*: $\vdash bi\ (f \wedge f1 \longrightarrow f) \longrightarrow (f \wedge f1);g \longrightarrow f;g$ **by** (*rule BiChopImpChop*)
from *2 3* **show** *?thesis* **using** *MP* **by** *blast*
qed

lemma *AndChopB*:
$\vdash (f \wedge f1);g \longrightarrow f1;g$
proof −
have *1*: $\vdash f \wedge f1 \longrightarrow f1$ **by** *auto*
hence *2*: $\vdash bi\ (f \wedge f1 \longrightarrow f1)$ **by** (*rule BiGen*)
have *3*: $\vdash bi\ (f \wedge f1 \longrightarrow f1) \longrightarrow (f \wedge f1);g \longrightarrow f1;g$ **by** (*rule BiChopImpChop*)
from *2 3* **show** *?thesis* **using** *MP* **by** *blast*
qed

lemma *NextChop*:
$\vdash (\bigcirc f);g = \bigcirc(f;g)$
proof −
have *1*: $\vdash skip;(f;g) = (skip;f);g$ **by** (*rule ChopAssoc*)
show *?thesis* **by** (*metis 1 int-eq next-d-def*)
qed

lemma *BoxChopImpChop* :
$\vdash \Box\ (g \longrightarrow g1) \longrightarrow f;g \longrightarrow f;g1$
proof −
have *1*: $\vdash g \longrightarrow g$ **by** *auto*
hence *2*: $\vdash bi\ (\ g \longrightarrow g)$ **by** (*rule BiGen*)
have *3*: $\vdash bi\ (\ f \longrightarrow f) \wedge \Box(g \longrightarrow g1) \longrightarrow f;g \longrightarrow f;g1$ **by** (*rule BiBoxChopImpChop*)

```
 from 2 3 show ?thesis by fastforce
qed

lemma LeftChopImpChop:
 assumes ⊢ f ⟶ f1
 shows  ⊢ f;g ⟶ f1;g
proof −
 have 1: ⊢ f ⟶ f1 using assms by auto
 hence 2: ⊢ bi (f ⟶ f1) by (rule BiGen)
 have 3: ⊢ bi (f ⟶ f1) ⟶ f;g ⟶ f1;g by (rule BiChopImpChop)
 from 2 3 show ?thesis using MP by blast
qed

lemma RightChopImpChop:
 assumes ⊢ g ⟶ g1
 shows  ⊢ f;g ⟶ f;g1
proof −
 have 1: ⊢ g ⟶ g1 using assms by auto
 hence 2: ⊢ □ (g ⟶ g1) by (rule BoxGen)
 have 3: ⊢ □ (g ⟶ g1) ⟶ f;g ⟶ f;g1 by (rule BoxChopImpChop)
 from 2 3 show ?thesis using MP by blast
qed

lemma RightChopEqvChop:
 assumes ⊢ g = g1
 shows  ⊢ (f;g) = (f;g1)
proof −
 have 1: ⊢ g = g1 using assms by auto
 have 2: (⊢ g ⟶ g1) ⟹ (⊢ f;g ⟶ f;g1) by (rule RightChopImpChop)
 have 3: (⊢ g1 ⟶ g) ⟹ (⊢ f;g1 ⟶ f;g) by (rule RightChopImpChop)
 from 1 2 3 show ?thesis by fastforce
qed

lemma ChopOrEqv:
 ⊢ f;(g ∨ g1) = (f;g ∨ f;g1)
proof −
 have 1: ⊢ g ⟶ g ∨ g1 by auto
 hence 2: ⊢ f;g ⟶ f;(g ∨ g1) by (rule RightChopImpChop)
 have 3: ⊢ g1 ⟶ g ∨ g1 by auto
 hence 4: ⊢ f;g1 ⟶ f;(g ∨ g1) by (rule RightChopImpChop)
 from 2 4 show ?thesis by (meson ChopOrImp Prop02 Prop11)
qed

lemma OrChopEqv:
 ⊢ (f ∨ f1);g = (f;g ∨ f1;g)
proof −
 have 1: ⊢ f ⟶ f ∨ f1 by auto
 hence 2: ⊢ f;g ⟶ (f∨ f1);g by (rule LeftChopImpChop)
 have 3: ⊢ f1 ⟶ f ∨ f1 by auto
 hence 4: ⊢ f1;g ⟶ (f∨ f1);g by (rule LeftChopImpChop)
 from 2 4 show ?thesis
 by (meson OrChopImp int-iffI Prop02)
qed

lemma OrChopImpRule:
 assumes ⊢ f ⟶ f1 ∨ f2
 shows   ⊢ f;g ⟶ (f1;g) ∨ (f2;g)
proof −
 have 1: ⊢ f ⟶ f1 ∨ f2 using assms by auto
 hence 2: ⊢ f;g ⟶ (f1 ∨ f2);g by (rule LeftChopImpChop)
 have 3: ⊢ (f1 ∨ f2); g = (f1;g ∨ f2;g) by (rule OrChopEqv)
 from 2 3 show ?thesis by fastforce
qed

lemma LeftChopEqvChop:
 assumes ⊢ f = f1
 shows   ⊢ f;g = (f1;g)
proof −
 have 1: ⊢ f = f1 using assms by auto
 hence 2: ⊢ f ⟶ f1 by auto
 hence 3: ⊢ f;g ⟶ f1;g by (rule LeftChopImpChop)
 have ⊢ f1 ⟶ f using 1 by auto
 hence 4: ⊢ f1;g ⟶ f;g by (rule LeftChopImpChop)
 from 3 4 show ?thesis by (simp add: int-iffI)
qed

lemma OrChopEqvRule:
 assumes ⊢ f = (f1 ∨ f2)
 shows   ⊢ f;g = ((f1;g) ∨ (f2;g))
proof −
 have 1: ⊢ f = (f1 ∨ f2) using assms by auto
 hence 2: ⊢ f;g = ((f1 ∨ f2);g) by (rule LeftChopEqvChop)
 have 3: ⊢ (f1 ∨ f2);g = (f1;g ∨ f2;g) by (rule OrChopEqv)
 from 2 3 show ?thesis by fastforce
qed

lemma NextImpNext:
 assumes ⊢ f ⟶ g
 shows   ⊢ ○ f ⟶ ○ g
proof −
 have 1: ⊢ f ⟶ g using assms by auto
 hence 2: ⊢ □ (f ⟶ g) by (rule BoxGen)
 have 3: ⊢ □ (f ⟶ g) ⟶ (skip;f) ⟶ (skip;g) by (rule BoxChopImpChop)
 have 4: ⊢(skip;f) ⟶ (skip;g) by (metis 2 3 MP)
 from 4 show ?thesis by (metis next-d-def)
qed

lemma ChopOrImpRule:
 assumes ⊢ g ⟶ g1 ∨ g2
 shows   ⊢ f;g ⟶ (f;g1) ∨ (f;g2)
```

```
proof −
 have 1: ⊢ g ⟶ g1 ∨ g2 using assms by auto
 hence 2: ⊢ f;g ⟶ f;(g1 ∨ g2) by (rule RightChopImpChop)
 have 3: ⊢ f;(g1 ∨ g2) = (f;g1 ∨ f;g2) by (rule ChopOrEqv)
 from 2 3 show ?thesis by fastforce
qed
```

```
lemma NextImpDist:
⊢ ○ (f ⟶ g) ⟶ ○ f ⟶ ○ g
proof −
 have 1: ⊢ (¬ (f ⟶ g)) = (f ∧ ¬ g) by auto
 hence 2: ⊢ skip;(¬ (f ⟶ g)) = skip;(f ∧ ¬ g) by (rule RightChopEqvChop)
 have 3: ⊢ f ⟶ g ∨ (f ∧ ¬ g) by auto
 hence 4: ⊢ skip;f ⟶ (skip;g) ∨ (skip;(f ∧ ¬ g)) by (rule ChopOrImpRule)
 hence 5: ⊢ ¬ (skip;(f ∧ ¬ g)) ⟶ (skip;f) ⟶ (skip;g) by auto
 have 6: ⊢ ¬ (skip;(¬(f ⟶ g))) ⟶ (skip;f) ⟶ (skip;g) using 2 5 by fastforce
 hence 7: ⊢ ¬ (○(¬(f ⟶ g))) ⟶ (○ f) ⟶ (○ g) by (simp add: next-d-def)
 have 8: ⊢ ○(f ⟶ g) ⟶ ¬ (○(¬(f ⟶ g))) by (rule NextImpNotNextNot)
 from 7 8 show ?thesis using lift-imp-trans by blast
qed
```

```
lemma ChopImpDiamond:
⊢ f;g ⟶ ◇ g
proof −
 have 1: ⊢ f ⟶ #True by auto
 hence 2: ⊢ f;g ⟶ #True;g by (rule LeftChopImpChop)
 from 2 show ?thesis by (simp add: sometimes-d-def)
qed
```

```
lemma NowImpDiamond:
⊢ f ⟶ ◇ f
proof −
 have 1: ⊢ empty;f = f by (rule EmptyChop)
 have 2: ⊢ empty ⟶ #True by auto
 hence 3: ⊢ empty;f ⟶ #True;f by (rule LeftChopImpChop)
 have 4: ⊢ f ⟶ #True;f using 1 3 by fastforce
 from 4 show ?thesis by (simp add: sometimes-d-def)
qed
```

```
lemma BoxElim:
⊢ □ f ⟶ f
proof −
  have 1: ⊢ ¬ f ⟶ ◇ (¬ f) by (rule NowImpDiamond)
  hence 2: ⊢ ¬ (◇ (¬ f)) ⟶ f by auto
  from 2 show ?thesis by (metis always-d-def)
qed
```

```
lemma NextDiamondImpDiamond:
 ⊢ ○ (◇ f) ⟶ ◇ f
```

```
proof –
 have  1: ⊢ skip;(#True;f) = ((skip;#True);f) by (rule ChopAssoc)
 hence 2: ⊢ (skip;#True);f = skip;(#True;f) by auto
 hence 3: ⊢ (skip;#True);f = ○(◇f) by (simp add: next-d-def sometimes-d-def)
 have  4: ⊢ (skip;#True);f ⟶ ◇ f by (rule ChopImpDiamond)
 from 3 4 show ?thesis by fastforce
qed

lemma BoxImpNowAndWeakNext:
 ⊢ □ f ⟶ (f ∧ wnext ( □ f) )
proof –
  have  1: ⊢ ¬ f ⟶ ◇ (¬ f)  by (rule NowImpDiamond)
  hence 2: ⊢ ¬ (◇ (¬ f)) ⟶ f by auto
  hence 3: ⊢ □ f ⟶ f by (metis always-d-def)
  have  4: ⊢ ○ ( ◇ (¬ f)) ⟶ ◇ ( ¬ f )  by (rule NextDiamondImpDiamond)
  have  5: ⊢ ¬ ¬ (◇ (¬ f)) ⟶ ◇( ¬ f )  by auto
  hence 6: ⊢ ○ ( ¬ ¬ (◇ (¬ f)) )  ⟶ ○ (◇( ¬ f ))  by (rule NextImpNext)
  have  7: ⊢ ○ ( ¬ ¬ (◇ (¬ f)) )  ⟶ ◇ ( ¬ f ) using 4 6 by auto
  hence 8: ⊢ ○ ( ¬( □ f)) ⟶ ◇ ( ¬ f )  by (simp add: always-d-def)
  hence 9: ⊢ ¬ (◇ ( ¬ f ))  ⟶ ¬ ( ○ ( ¬( □ f))) by auto
  hence 10:⊢ □f ⟶ wnext ( □ f ) by (simp add: always-d-def wnext-d-def)
  from 3 10 show ?thesis by fastforce
qed

lemma BoxImpBoxRule:
 assumes ⊢ f ⟶ g
 shows  ⊢ □ f⟶ □ g
proof –
 have  1: ⊢ f ⟶ g using assms by auto
 hence 2: ⊢ ¬ g ⟶  ¬ f by auto
 hence 3: ⊢ □(¬ g ⟶  ¬ f) by (rule BoxGen)
 have  4: ⊢  □(¬ g ⟶  ¬ f) ⟶ (#True;(¬g)) ⟶ (#True;(¬f)) by (rule BoxChopImpChop)
 have  5: ⊢ (#True;(¬g)) ⟶ (#True;(¬f)) using 3 4 MP by blast
 hence 6: ⊢ ◇ (¬g)  ⟶ ◇(¬f) by (simp add: sometimes-d-def)
 hence 7: ⊢  ¬ (◇(¬f) ) ⟶ ¬( ◇ (¬g)) by auto
 from 7 show ?thesis by (simp add: always-d-def)
qed

lemma BoxImpDist:
 ⊢  □(f ⟶ g) ⟶ □ f ⟶ □ g
proof –
 have  1: ⊢ (f ⟶ g) ⟶ (¬  g ⟶ ¬  f)  by auto
 hence 2: ⊢ □(f ⟶ g) ⟶ □(¬  g ⟶ ¬  f)  by (rule BoxImpBoxRule)
 have  3: ⊢ □((¬  g) ⟶ ¬  f) ⟶  (#True; (¬  g)) ⟶ (#True; (¬  f)) by (rule BoxChopImpChop)
 have  4: ⊢ □(f ⟶ g) ⟶   (#True; (¬  g)) ⟶ (#True; (¬  f)) using 2 3 lift-imp-trans by blast
 hence 5: ⊢ □(f ⟶ g) ⟶   ◇(¬  g) ⟶ ◇(¬  f)  by (simp add: sometimes-d-def)
 hence 6: ⊢ □(f ⟶ g) ⟶   ¬( ◇(¬  f)) ⟶ ¬( ◇(¬  g))  by auto
 from 6 show ?thesis by (simp add: always-d-def)
qed
```

```
lemma DiamondEmpty:
 ⊢  ◇ empty
proof –
 have 1: ⊢ #True  by auto
 have 2: ⊢ #True;  empty  = #True  by (rule ChopEmpty)
 have 3: ⊢ #True;  empty  using 1 2 by auto
 from 3 show ?thesis by (simp add: sometimes-d-def)
qed

lemma NextEqvNext:
 assumes ⊢   f= g
 shows   ⊢ ○ f = ○ g
proof –
 have  1: ⊢ f= g using assms by auto
 hence 2: ⊢ skip;f = skip;g  by (rule RightChopEqvChop)
 from 1 show ?thesis by (metis 2 next-d-def)
qed

lemma NextAndNextImpNextRule:
 assumes ⊢ (f ∧ g) ⟶ h
 shows  ⊢ (○ f ∧ ○ g) ⟶ ○ h
using assms by (auto simp: next-defs)

lemma NextAndNextEqvNextRule:
 assumes ⊢ (f ∧ g) = h
 shows  ⊢ (○ f ∧ ○ g) = ○ h
using assms by (metis NextAndNextImpNextRule Prop11 Prop12 int-eq int-simps(20))

lemma WeakNextEqvWeakNext:
 assumes ⊢ f= g
 shows  ⊢ wnext  f = wnext  g
using assms using inteq-reflection by force

lemma DiamondImpDiamond:
 assumes ⊢ f⟶ g
 shows  ⊢ ◇ f ⟶ ◇ g
using assms by (simp add: RightChopImpChop sometimes-d-def)

lemma DiamondEqvDiamond:
 assumes ⊢ f= g
 shows  ⊢ ◇ f = ◇ g
using assms using int-eq by force

lemma BoxEqvBox:
 assumes ⊢ f= g
 shows  ⊢ □ f = □ g
using assms using inteq-reflection by force

lemma BoxAndBoxImpBoxRule:
 assumes ⊢ f ∧ g ⟶ h
 shows  ⊢ □ f ∧ □ g ⟶ □ h
using assms by (auto simp: always-defs Valid-def)

lemma BoxAndBoxEqvBoxRule:
 assumes ⊢ (f ∧ g) = h
 shows  ⊢ (□ f ∧ □ g) = □ h
using assms BoxAndBoxImpBoxRule BoxImpBoxRule by (metis int-iffD1 int-iffD2 int-iffI Prop12)

lemma ImpBoxRule:
 assumes ⊢  f ⟶ g
 shows  ⊢ □ f ⟶ □ g
using assms by (simp add: BoxImpBoxRule)

lemma BoxIntro:
 assumes ⊢  f⟶ g
        ⊢  more  ∧ f  ⟶○ f
 shows  ⊢ f⟶□ g
proof −
 have  1: ⊢  more  ∧ f  ⟶○ f  using assms by auto
 hence  2: ⊢ f⟶ (empty ∨ ○f) by (auto simp: next-defs empty-defs more-defs)
 hence  3: ⊢ f  ⟶ wnext f by (auto simp: wnext-defs empty-defs next-defs)
 hence  4: ⊢ □(f  ⟶ wnext f) by (rule BoxGen)
 have  5: ⊢ (□ (f  ⟶ wnext f)) ∧ f  ⟶ □ f  by (rule BoxInduct)
 hence  6: ⊢ (□ (f  ⟶ wnext f)) ⟶ (f  ⟶ □f) by fastforce
 have  7: ⊢ f  ⟶ □f  using 4 6 MP by blast
 have  8: ⊢ □f  ⟶ f by (rule BoxElim)
 have  9: ⊢ f = □ f using 7 8 by fastforce
 have  10: ⊢ f  ⟶ g using assms by auto
 hence 11: ⊢ □f  ⟶ □ g by (rule ImpBoxRule)
 from 7 9 11 show ?thesis  using lift-imp-trans by blast
qed

lemma NextLoop:
 assumes ⊢ f⟶ ○ f
 shows  ⊢ ¬  f
proof −
 have  1: ⊢ f⟶ ○ f  using assms by auto
 hence  2: ⊢ f  ⟶ (more ∧ wnext f) by (auto simp: more-defs wnext-defs next-defs)
 hence  3: ⊢ f  ⟶ wnext f  by auto
 hence  4: ⊢ □(f  ⟶ wnext f) by (rule BoxGen)
 have  5: ⊢ □ (f  ⟶ wnext f) ∧ f  ⟶ □ f  by (rule BoxInduct)
 hence  6: ⊢ □ (f  ⟶ wnext f)  ⟶ (f  ⟶ □f) by fastforce
 have  7: ⊢ f  ⟶ □f  using 4 6 MP by blast
 have  8: ⊢ □f  ⟶ f by (rule BoxElim)
 have  9: ⊢ f = □ f using 7 8 by fastforce
 have  10: ⊢ f  ⟶ more using 2 by auto
 hence 11: ⊢ □ f  ⟶ □ more by (rule ImpBoxRule)
 have  12: ⊢ ¬(□ more)  by (auto simp: always-defs more-defs)
 from 7 9 11 12 show ?thesis by fastforce
qed
```

```
lemma WnextEqvEmptyOrNext:
 ⊢ wnext f = (empty ∨ ○ f)
by (auto simp: empty-defs wnext-defs next-defs)

lemma NotEmptyAndNext:
 ⊢ ¬(empty ∧ ○ f)
by (auto simp: empty-defs next-defs)

lemma BoxEqvAndWnextBox:
 ⊢ □ f = (f ∧ wnext ( □ f))
proof −
 have 1: ⊢ □ f ⟶ f ∧ wnext ( □ f)
     using BoxImpNowAndWeakNext by blast
 have 2: ⊢ f ∧ wnext ( □ f) ⟶ f
     by auto
 have 3: ⊢ more ∧ (f ∧ wnext ( □ f) ) ⟶ ○ (f ∧ wnext ( □ f) )
     using 1 NextImpNext WnextEqvEmptyOrNext empty-d-def int-iffD1
     by (metis Prop01 Prop05 Prop08)
 have 4: ⊢ f ∧ wnext ( □ f) ⟶ □ f
     using 2 3 BoxIntro by blast
 from 1 4 show ?thesis by fastforce
qed

lemma BoxEqvAndEmptyOrNextBox:
 ⊢ □f = (f ∧ (empty  ∨  ○(□ f)))
using BoxEqvAndWnextBox WnextEqvEmptyOrNext by (metis int-eq)

lemma BoxEqvBoxBox:
 ⊢ □f = □ (□ f)
using BoxGen BoxInduct
by (metis BoxImpNowAndWeakNext MP int-iffI Prop09 Prop12)

lemma BoxBoxImpBox:
 ⊢ □(□h) ⟶ □ h
by (simp add: BoxElim)

lemma BoxImpBoxBox:
 ⊢ □ h  ⟶  □(□h)
by (auto simp: always-defs)

lemma DiamondIntro:
 assumes ⊢ (f ∧ ¬  g) ⟶ ○ f
 shows  ⊢ f⟶◇ g
proof −
 have  1: ⊢ f ∧ ¬  g ⟶ ○ f
       using assms by auto
 hence  2: ⊢ f ∧ ¬  g ∧ (□ (¬ g))⟶ (○ f) ∧ (□ (¬ g))
       by auto
 have  3: ⊢ (□ (¬ g)) ⟶ ¬  g
```

```
      by (rule BoxElim)
  hence  4: ⊢ □ (¬ g) = ((□ (¬ g)) ∧ ¬  g)
      using BoxImpBoxBox BoxBoxImpBox by fastforce
  have   5: ⊢ f ∧ (□ (¬ g))⟶ ○ f ∧ □ (¬ g)
      using 2 4  by fastforce
  have   6: ⊢ □ (¬ g) = ((¬ g) ∧ wnext(□ (¬ g)))
      using BoxEqvAndWnextBox by metis
  have   7: ⊢ ○ f ∧ □ (¬ g) ⟶  ○ f ∧ wnext(□ (¬ g))
      using 6 by auto
  have   8: ⊢ f ∧ (□ (¬ g))⟶ ○ f ∧ wnext(□ (¬ g))
      using 5 7 using lift-imp-trans by blast
  hence  9: ⊢ f ∧ (□ (¬ g))⟶ more ∧ wnext f ∧ wnext(□ (¬ g))
      by (auto simp: always-defs more-defs next-defs wnext-defs)
  hence 10: ⊢ f ∧ (□ (¬ g))⟶wnext f ∧ wnext(□ (¬ g))
      by auto
  hence 11: ⊢ f ∧ (□ (¬ g))⟶ wnext (f ∧ □ (¬ g))
      by (auto simp: wnext-defs always-defs next-defs)
  hence 12: ⊢ □(f ∧ (□ (¬ g))⟶ wnext (f ∧ □ (¬ g)))
      by (rule BoxGen)
  have  13: ⊢ □(f ∧ (□ (¬ g))⟶ wnext (f ∧ □ (¬ g))) ∧ f ∧ (□ (¬ g)) ⟶ □(f ∧ (□ (¬ g)))
      by (rule BoxInduct)
  hence 14: ⊢ □(f ∧ (□ (¬ g))⟶ wnext (f ∧ □ (¬ g))) ⟶ ((f ∧ (□ (¬ g))) ⟶ □(f ∧ (□ (¬ g))))
      by fastforce
  have  15: ⊢ ((f ∧ (□ (¬ g))) ⟶ □(f ∧ (□ (¬ g))))
      using 12 14 MP by blast
  have  16: ⊢ □(f ∧ (□ (¬ g))) ⟶ (f ∧ (□ (¬ g)))
      by (rule BoxElim)
  have  17: ⊢ □(f ∧ (□ (¬ g))) = (f ∧ (□ (¬ g)))
      using 16 15 by fastforce
  have  18: ⊢ (f ∧ (□ (¬ g))) ⟶ more
      using 9 by auto
  hence 19: ⊢ □(f ∧ (□ (¬ g))) ⟶ □ more
      by (rule ImpBoxRule)
  have  20: ⊢ ¬(□ more)
      by (auto simp: always-defs more-defs)
  have  21: ⊢ ¬(f ∧ (□ (¬ g)))
      using 17 19 20 by fastforce
  hence 22: ⊢ ¬ f ∨ ¬ (□ (¬ g))
      by auto
  have  23: ⊢ (¬ (□ (¬ g))) = ◇ g
      by (auto simp: always-d-def)
  from 22 23 show ?thesis by fastforce
qed


lemma DiamondIntroB:
  assumes ⊢ (f ∧ ¬  g) ⟶ ○ (f ∧ ¬ g)
  shows  ⊢ f⟶◇ g
proof –
  have 1: ⊢ (f ∧ ¬  g) ⟶ ○ (f ∧ ¬g)  using assms by auto
```

```
 hence 2: ⊢ ¬(f ∧ ¬ g)  by (rule NextLoop)
 hence 3: ⊢ f ⟶ g by auto
 have  4: ⊢ g ⟶ ◇ g by (rule NowImpDiamond)
 from 3 4 show ?thesis using lift-imp-trans by blast
qed

lemma NextContra :
 assumes ⊢ (f∧ ¬  g)⟶ (○ f ∧ ¬( ○ g))
 shows  ⊢ f ⟶ g
proof −
 have  1: ⊢ (f∧ ¬  g)⟶ (○ f ∧ ¬( ○ g))  using assms by auto
 hence 2: ⊢ ¬( f ⟶ g) ⟶ ○ ( ¬(f ⟶ g))  by (auto simp: next-defs Valid-def)
 hence 3: ⊢ ¬ ¬( f ⟶ g)  by (rule NextLoop)
 from 3 show ?thesis by auto
qed

lemma DiamondDiamondEqvDiamond:
  ⊢ ◇(◇ f) = ◇ f
proof −
 have  1: ⊢ #True;#True = #True by (auto simp: chop-defs)
 hence 2: ⊢ (#True;#True);f = #True;f  using LeftChopEqvChop by blast
 have  3: ⊢ (#True;#True);f = #True;(#True;f)  using ChopAssoc by fastforce
 from 2 3 show ?thesis by (metis inteq-reflection sometimes-d-def)
qed

lemma WeakNextDiamondInduct:
 assumes ⊢ wnext  (◇ f) ⟶ f
 shows  ⊢ f
proof −
 have  1: ⊢ wnext  (◇ f) ⟶ f  using assms by blast
 hence 2: ⊢ ¬ f ⟶ ¬( wnext  (◇ f)) by fastforce
 hence 3: ⊢ ¬ f ⟶ ○( ¬ (◇ f))  by (simp add: wnext-d-def)
 have  4: ⊢ f ⟶ ◇ f by (rule NowImpDiamond)
 hence 5: ⊢ ¬( ◇ f) ⟶ ¬ f   by auto
 have  6: ⊢ ¬f ⟶ ○( ¬f) using 3 5 using NextImpNext lift-imp-trans by blast
 hence 7: ⊢ ¬¬ f by (rule NextLoop)
 from 7 show ?thesis by auto
qed

lemma EmptyNextInducta:
 assumes ⊢ empty  ⟶ f
         ⊢ ○ f ⟶ f
 shows ⊢ f
proof −
 have  1: ⊢ empty  ⟶ f  using assms by auto
 have  2: ⊢ ○ f ⟶ f  using assms by blast
 have  3: ⊢ (empty ∨ ○ f) ⟶ f   using 1 2  by fastforce
 have  4: ⊢ wnext f = (empty ∨ ○ f) by  (rule  WnextEqvEmptyOrNext)
 hence 5: ⊢ wnext f ⟶ f using 3 by fastforce
 hence 6: ⊢ ¬f ⟶ ¬ (wnext f) by auto
 hence 7: ⊢ ¬f ⟶ ○(¬ f)  by (auto simp: wnext-d-def)
 hence 8: ⊢ ¬ ¬ f  by (rule NextLoop)
 from 8 show ?thesis by auto
qed

lemma EmptyNextInductb:
 assumes ⊢ empty  ∧ f ⟶ g
        ⊢ ○(f⟶ g) ∧ f ⟶ g
 shows  ⊢ f⟶ g
proof −
 have  1: ⊢ empty  ∧ f ⟶ g  using assms by auto
 have  2: ⊢ ○(f⟶ g) ∧ f ⟶ g  using assms by blast
 have  3: ⊢ (empty ∨ ○(f⟶ g)) ∧ f ⟶ g  using 1 2 by fastforce
 hence 4: ⊢ wnext (f⟶ g)  ∧ f ⟶ g    using WnextEqvEmptyOrNext by fastforce
 hence 5: ⊢ wnext (f⟶ g) ⟶ ( f ⟶ g)   by fastforce
 hence 6: ⊢ ¬ ( f ⟶ g) ⟶ ¬ ( wnext (f⟶ g))  by fastforce
 hence 7: ⊢ ¬ ( f ⟶ g) ⟶ ○ ( ¬(f⟶ g))  by (simp add: wnext-d-def)
 hence 8: ⊢ ¬ ¬ ( f ⟶ g) by (rule NextLoop)
 from 8 show ?thesis by auto
qed


lemma FinImpFin:
 assumes ⊢ f ⟶ g
 shows  ⊢  fin  f ⟶  fin  g
using ImpBoxRule[of TEMP (empty ⟶ f) TEMP (empty ⟶ g)] assms fin-d-def
by (smt intI intensional-rews(3) inteq-reflection Prop10)

lemma FinEqvFin:
 assumes ⊢ f = g
 shows  ⊢  fin  f =  fin  g
using assms by (simp add: FinImpFin Prop11)

lemma FinAndFinImpFinRule:
 assumes ⊢ f ∧ g ⟶ h
 shows  ⊢  fin  f ∧  fin  g ⟶  fin  h
proof −
  have ⊢ f ∧ g ⟶ h  using assms by auto
  then show ?thesis by (simp add: fin-defs Valid-def)
qed

lemma FinAndFinEqvFinRule:
 assumes ⊢ (f ∧ g) = h
 shows  ⊢  (fin  f ∧  fin  g) =  fin  h
using assms
by (simp add: FinAndFinImpFinRule FinImpFin Prop11 Prop12)

lemma HaltEqvHalt:
 assumes ⊢ f = g
 shows  ⊢  halt  f =  halt  g
```

```
proof −
 have 1: ⊢ f = g using assms by auto
 hence 2: ⊢ (empty = f) = (empty = g) by auto
 hence 3: ⊢ □(empty = f) = □ (empty = g) by (rule BoxEqvBox)
 from 3 show ?thesis by (simp add: halt-d-def)
qed

lemma BiImpDiImpDi:
 ⊢ bi (f ⟶ g) ⟶ di f ⟶ di g
proof −
 have 1: ⊢ bi (f⟶ g) ⟶ (f; #True) ⟶ (g; #True) by (rule BiChopImpChop)
 from 1 show ?thesis by (simp add: di-d-def)
qed

lemma DiImpDi:
 assumes ⊢ f⟶ g
 shows ⊢ di f ⟶ di g
proof −
 have 1: ⊢ f⟶ g using assms by auto
 hence 2: ⊢ f; #True ⟶ g; #True by (rule LeftChopImpChop)
 from 2 show ?thesis by (simp add: di-d-def)
qed

lemma BiImpBiRule:
 assumes ⊢ f⟶ g
 shows ⊢ bi f ⟶ bi g
proof −
 have 1: ⊢ f⟶ g using assms by auto
 hence 2: ⊢ ¬ g ⟶ ¬ f by auto
 hence 3: ⊢ di (¬ g) ⟶ di (¬ f) by (rule DiImpDi)
 hence 4: ⊢ ¬ ( di (¬ f)) ⟶ ¬ ( di (¬ g)) by auto
 from 4 show ?thesis by (simp add: bi-d-def)
qed

lemma DiEqvDi:
 assumes ⊢ f= g
 shows ⊢ di f = di g
proof −
 have 1: ⊢ f = g using assms by auto
 hence 2: ⊢ f; #True = g; #True by (rule LeftChopEqvChop)
 from 2 show ?thesis by (simp add: di-d-def)
qed

lemma BiEqvBi:
 assumes ⊢ f = g
 shows ⊢ bi f = bi g
proof −
 have 1: ⊢ f = g using assms by auto
 hence 2: ⊢ (¬ f) = (¬ g) by auto
 hence 3: ⊢ di (¬ f) = di (¬ g) by (rule DiEqvDi)
 hence 4: ⊢ (¬  (di (¬  f))) = (¬ ( di (¬  g)))  by auto
 from 4 show ?thesis by (simp add: bi-d-def)
qed
```

```
lemma LeftChopChopImpChopRule:
 assumes ⊢ (f; g) ⟶ g
 shows  ⊢ (f; g); h ⟶ (g; h)
proof −
 have  1: ⊢ (f; g) ⟶ g  using assms by blast
 hence 2: ⊢ (f; g); h ⟶ g; h  by (rule LeftChopImpChop)
 have  3: ⊢ f; (g; h) = (f; g); h    by (rule ChopAssoc)
 from 2 3 show ?thesis by auto
qed
```

```
lemma AndChopCommute :
⊢ (f ∧ f1); g = (f1 ∧ f); g
proof −
 have 1: ⊢ (f ∧ f1) = (f1 ∧ f)   by auto
 from 1 show ?thesis by (rule LeftChopEqvChop)
qed
```

```
lemma BiAndChopImport:
⊢  bi  f ∧ (f1; g) ⟶ (f ∧ f1); g
proof −
 have  1: ⊢ f ⟶ (f1 ⟶ f∧ f1)  by auto
 hence 2: ⊢ bi  f ⟶ bi  (f1 ⟶ f∧ f1)  by (rule BiImpBiRule)
 have  3: ⊢ bi  (f1 ⟶ (f ∧ f1)) ⟶   f1; g ⟶ (f ∧ f1); g  by (rule BiChopImpChop)
 from 2 3 show ?thesis using MP by fastforce
qed
```

```
lemma StateAndChopImport:
⊢ (init w) ∧ (f; g) ⟶ ((init w) ∧ f); g
proof −
 have  1: ⊢ (init w)⟶ bi  (init w)  by (rule StateImpBi)
 hence 2: ⊢ (init w) ∧ (f; g) ⟶ bi (init w) ∧ (f; g)  by auto
 have  3: ⊢ bi  (init w) ∧ (f; g) ⟶ ((init w) ∧ f); g   by (rule BiAndChopImport)
 from 2 3 show ?thesis using MP by fastforce
qed
```

4.4 Further Properties Di and Bi

```
lemma ImpDi:
⊢ f ⟶  di  f
proof −
 have  1: ⊢ f;  empty  = f  by (rule ChopEmpty)
 have  2: ⊢ empty  ⟶ #True  by auto
 hence 3: ⊢ f;  empty  ⟶ f; #True  by (rule RightChopImpChop)
 have 4 : ⊢ f ⟶ f; #True  using 1 3 by fastforce
 from 4 show ?thesis by (simp add: di-d-def)
qed
```

```
lemma DiState:
 ⊢ di (init w) = (init w)
proof −
 have 0: ⊢ (init (¬w)) ⟶ bi  (init (¬w))  using StateImpBi by fastforce
 hence 1: ⊢ ¬(init w) ⟶ bi ( ¬ (init w)) using Initprop(2) by (metis inteq-reflection)
 hence 2: ⊢ (¬  (init w)) ⟶ ¬ ( di (¬ ¬  (init w)))  by (simp add: bi-d-def)
 have 3: ⊢ (¬  (init w) ⟶¬  (di (¬ ¬  (init w)))) ⟶ ( di (¬ ¬  (init w)) ⟶ (init w))  by auto
 have 4: ⊢  di (¬ ¬  (init w)) ⟶ (init w)  using 2 3 MP by blast
 have 5: ⊢ (init w) ⟶ ¬ ¬  (init w)  by auto
 hence 6: ⊢  di  (init w) ⟶  di (¬ ¬  (init w))  by (rule DiImpDi)
 have 7: ⊢  di  (init w) ⟶ (init w) using 6 4 using lift-imp-trans by metis
 have 8: ⊢ (init w) ⟶  di  (init w)  by (rule ImpDi)
 from 7 8 show ?thesis by fastforce
qed

lemma StateChop:
 ⊢ (init w); f ⟶ (init w)
using DiState by (auto simp: di-defs init-defs chop-defs)

lemma StateChopExportA:
 ⊢ ((init w) ∧ f); g ⟶ (init w)
using DiState by (auto simp: init-defs chop-defs)

lemma StateAndChop:
 ⊢ ((init w) ∧ f); g = ((init w) ∧ (f; g))
by (simp add: AndChopB StateAndChopImport StateChopExportA Prop11 Prop12)

lemma StateAndChopImpChopRule:
 assumes ⊢ (init w) ∧ f ⟶ f1
 shows  ⊢ (init w) ∧ (f; g) ⟶ (f1; g)
proof −
 have 1: ⊢ (init w) ∧ f ⟶ f1  using assms by auto
 hence 2: ⊢ ((init w) ∧ f); g ⟶ f1; g  by (rule LeftChopImpChop)
 have 3: ⊢ ((init w) ∧ f); g = ((init w) ∧ (f; g))  by (rule StateAndChop)
 from 2 3 show ?thesis by fastforce
qed

lemma StateImpChopEqvChop :
 assumes ⊢(init w) ⟶ (f = f1)
 shows  ⊢ (init w) ⟶ ((f; g) = (f1; g))
proof −
 have 1: ⊢ (init w) ⟶ (f= f1) using assms by auto
 hence 2: ⊢ (init w) ∧ f ⟶ f1  by auto
 hence 3: ⊢ (init w) ∧ (f; g) ⟶ (f1; g)  by (rule StateAndChopImpChopRule)
 have 4: ⊢ (init w) ∧ f1 ⟶ f  using 1 by auto
 hence 5: ⊢ (init w) ∧ (f1; g) ⟶ (f; g) by (rule StateAndChopImpChopRule)
 from 3 5 show ?thesis by fastforce
qed
```

```
lemma ChopEqvStateAndChop:
  assumes ⊢ f = (init w) ∧ f1
  shows   ⊢ (f; g) = ((init w) ∧ (f1; g))
proof –
 have  1: ⊢ f = ((init w) ∧ f1)  using assms by auto
 hence 2: ⊢ f; g = (((init w) ∧ f1); g)  by (rule LeftChopEqvChop)
 have  3: ⊢ ((init w) ∧ f1); g = ((init w) ∧ (f1; g))  by (rule StateAndChop)
 from 2 3 show ?thesis by fastforce
qed

lemma DiIntro:
⊢ f ⟶  di  f
proof –
  have  1: ⊢ f;  empty  = f  by (rule ChopEmpty)
  have  2: ⊢  empty  ⟶ #True  by auto
  hence 3: ⊢ □( empty  ⟶ #True) by (rule BoxGen)
  have  4: ⊢ □( empty  ⟶ #True) ⟶  (f;  empty  ⟶ f; #True)  by (rule BoxChopImpChop)
  have  5: ⊢ f;  empty  ⟶ f; #True  using 3 4 MP by fastforce
  hence 6: ⊢ f;  empty  ⟶  di  f  by (simp add: di-d-def)
  from 1 6 show ?thesis by fastforce
qed

lemma BiElim:
⊢  bi  f ⟶ f
proof –
  have  1: ⊢ ¬  f ⟶  di (¬  f)  by (rule DiIntro)
  have  2: ⊢ (¬  f ⟶  di (¬  f)) ⟶ (¬ ( di (¬  f)) ⟶ f)  by auto
  have  3: ⊢ ¬  (di (¬  f)) ⟶ f  using 1 2 MP by blast
  from 3 show ?thesis by (metis bi-d-def)
qed

lemma BiContraPosImpDist:
 ⊢ bi (¬  g ⟶ ¬  f) ⟶ (bi  f) ⟶ (bi  g)
proof –
  have  1: ⊢ bi (¬  g ⟶ ¬  f) ⟶ ( di (¬  g)) ⟶ ( di (¬  f))  by (rule BiImpDiImpDi)
  hence 2: ⊢ bi (¬  g ⟶ ¬  f) ⟶ (¬ ( di (¬  f))) ⟶ (¬ ( di (¬  g)))  by auto
  from 2 show ?thesis by (metis bi-d-def)
qed

lemma BiImpDist:
⊢ bi (f⟶ g) ⟶ (bi  f) ⟶ (bi  g)
proof –
 have  1: ⊢ (f ⟶ g) ⟶ (¬  g ⟶ ¬  f)  by auto
 hence 2: ⊢ ¬ (¬  g ⟶ ¬  f) ⟶ ¬ (f ⟶ g)  by auto
 hence 3: ⊢ bi (¬ (¬  g ⟶ ¬  f) ⟶ ¬ (f ⟶ g))  by (rule BiGen)
 have  4: ⊢ bi (¬ (¬  g ⟶ ¬  f) ⟶ ¬ (f ⟶ g))
            ⟶
            bi (f ⟶ g) ⟶ bi (¬  g ⟶ ¬  f)  by (rule BiContraPosImpDist)
 have  5: ⊢ bi (f ⟶ g) ⟶ bi (¬  g ⟶ ¬  f)  using 3 4 MP by blast
 have  6: ⊢ bi (¬  g ⟶ ¬  f) ⟶ (bi  f) ⟶ (bi  g)  by (rule BiContraPosImpDist)
 from 5 6 show ?thesis  using lift-imp-trans by blast
qed

lemma IfChopEqvRule:
 assumes ⊢ f =  if_i  (init w)  then  f1  else  f2
 shows   ⊢ f; g =  if_i  (init w)  then  (f1; g)  else  (f2; g)
proof −
 have  1: ⊢ f =  if_i  (init w)  then  f1  else  f2
       using assms by auto
 hence 2: ⊢ f = (((init w) ∧ f1) ∨  ( (init (¬ w)) ∧ f2))
       by (simp add: ifthenelse-d-def init-defs Valid-def)
 hence 3: ⊢ f; g = (((init w) ∧ f1); g ∨  (  (init (¬ w)) ∧ f2); g)
       by (rule OrChopEqvRule)
 have  4: ⊢ ((init w) ∧ f1); g = ((init w) ∧ (f1; g))
       by (rule StateAndChop)
 have  5: ⊢ (  (init (¬ w)) ∧ f2); g =   ((init (¬ w)) ∧ (f2; g))
       by (rule StateAndChop)
 have  6: ⊢ f; g = (((init w) ∧ f1; g)  ∨  (  (init (¬ w)) ∧ f2; g))
       using 3 4 5 by fastforce
 from 6 show ?thesis by (simp add: ifthenelse-d-def init-defs Valid-def)
qed

lemma ChopOrEqvRule:
 assumes ⊢ g = (g1 ∨  g2)
 shows   ⊢ f; g = ((f; g1) ∨  (f; g2))
proof −
 have  1: ⊢ g = (g1 ∨  g2)  using assms by auto
 hence 2: ⊢ f; g = (f; (g1 ∨  g2))  by (rule RightChopEqvChop)
 have  3: ⊢ f; (g1 ∨  g2) = (f; g1 ∨  f; g2)  by (rule ChopOrEqv)
 from 2 3 show ?thesis by fastforce
qed

lemma EmptyOrChopEqv:
⊢ ( empty  ∨  f); g = (g ∨  (f; g))
proof −
 have 1: ⊢ ( empty  ∨  f); g = (( empty ; g) ∨  (f; g))  by (rule OrChopEqv)
 have 2: ⊢  empty ; g = g  by (rule EmptyChop)
 from 1 2 show ?thesis by fastforce
qed

lemma EmptyOrNextChopEqv:
 ⊢ ( empty  ∨  ○ f); g = (g ∨  ○(f; g))
proof −
 have 1: ⊢ ( empty  ∨  ○ f); g = (g ∨  ((○ f); g))  by (rule EmptyOrChopEqv)
 have 2: ⊢ (○ f); g = ○(f; g)  by (rule NextChop)
 from 1 2 show ?thesis by fastforce
qed

lemma EmptyOrChopImpRule:
 assumes ⊢   f ⟶  empty  ∨  f1
 shows  ⊢ f; g ⟶ g ∨ (f1; g)
proof –
 have 1: ⊢ f ⟶ empty ∨ f1 using assms by auto
 hence 2: ⊢ f; g ⟶ ( empty ∨ f1); g by (rule LeftChopImpChop)
 have 3: ⊢ ( empty ∨ f1); g = (g ∨ (f1; g)) by (rule EmptyOrChopEqv)
 from 2 3 show ?thesis by fastforce
qed

lemma EmptyOrChopEqvRule:
 assumes ⊢ f = (empty ∨ f1)
 shows  ⊢ f; g = (g ∨ (f1; g))
proof –
 have 1: ⊢ f = (empty ∨ f1) using assms by auto
 hence 2: ⊢ f; g = (( empty ∨ f1); g) by (rule LeftChopEqvChop)
 have 3: ⊢ ( empty ∨ f1); g = (g ∨ (f1; g)) by (rule EmptyOrChopEqv)
 from 2 3 show ?thesis by fastforce
qed

lemma EmptyOrNextChopImpRule:
 assumes ⊢ f ⟶ empty ∨ ○ f1
 shows  ⊢ f; g ⟶ g ∨ ○(f1; g)
proof –
 have 1: ⊢ f ⟶ empty ∨ ○ f1 using assms by auto
 hence 2: ⊢ f; g ⟶ ( empty ∨ ○ f1); g by (rule LeftChopImpChop)
 have 3: ⊢ ( empty ∨ ○ f1); g = (g ∨ ○(f1; g)) by (rule EmptyOrNextChopEqv)
 from 2 3 show ?thesis by fastforce
qed

lemma EmptyOrNextChopEqvRule:
 assumes ⊢ f = (empty ∨ ○ f1)
 shows  ⊢ f; g = (g ∨ ○(f1; g))
proof –
 have 1: ⊢ f = (empty ∨ ○ f1) using assms by auto
 hence 2: ⊢ f; g = (( empty ∨ ○ f1); g) by (rule LeftChopEqvChop)
 have 3: ⊢ ( empty ∨ ○ f1); g = (g ∨ ○(f1; g)) by (rule EmptyOrNextChopEqv)
 from 2 3 show ?thesis by fastforce
qed

lemma ChopEmptyOrImpRule:
 assumes ⊢ g ⟶ empty ∨ g1
 shows ⊢ f; g ⟶ f ∨ (f; g1)
proof –
 have 1: ⊢ g ⟶ empty ∨ g1 using assms by auto
 hence 2: ⊢ f; g ⟶ (f; empty ) ∨ (f; g1) by (rule ChopOrImpRule)
 have 3: ⊢ f; empty = f by (rule ChopEmpty)
 from 2 3 show ?thesis by fastforce
qed

lemma StateAndEmptyImpBoxState:
 ⊢ (init w) ∧ empty ⟶ □ (init w)
```

```
by (simp add: init-defs empty-defs always-defs Valid-def)

lemma BoxEqvAndBox:
⊢ □ f = (f ∧ □ f)
by (simp add: always-defs Valid-def) fastforce

lemma NotBoxImpNotOrNotNextBox:
⊢ ¬( □ f) ⟶ ¬f ∨  ¬( ○ (□ f) )
proof −
  have 1: ⊢ f ∧ (○ (□ f)) ⟶ □ f
       using BoxEqvAndEmptyOrNextBox by fastforce
  hence 2: ⊢ ¬( □ f) ⟶ ¬(f ∧ (○ (□ f)) )   by fastforce
  have 3: ⊢ (¬(f ∧ (○ (□ f)) )) = (¬f ∨ ¬( ○ (□ f) )   )  by auto
  from 2 3 show ?thesis by auto
qed

lemma BoxStateChopBoxEqvBox:
⊢   □ (init w); □ (init w) = □ (init w)
proof −
 have   1: ⊢ (□ (init w)) = ((init w) ∧ ( empty  ∨  ○(□ (init w))))
        by (rule BoxEqvAndEmptyOrNextBox)
 hence  2: ⊢ (□ (init w); □ (init w)) =
            ((init w) ∧ (( empty  ∨  ○(□ (init w))); □ (init w)))
        by (metis StateAndChop inteq-reflection)
 have   3: ⊢ (( empty  ∨  ○(□ (init w))); □ (init w)) =
           (□ (init w) ∨  ○(□ (init w); □ (init w)))
        by (rule EmptyOrNextChopEqv)
 have   4: ⊢ (□ (init w); □ (init w)) =
           ((init w) ∧ (□ (init w) ∨  ○(□ (init w); □ (init w))))
        using 2 3 by fastforce
 have   5: ⊢ ¬ (□ (init w)) ⟶ ¬  (init w) ∨  ¬( ○(□ (init w)))
        by (rule NotBoxImpNotOrNotNextBox)
 have   6: ⊢ (□ (init w); □ (init w)) ∧ ¬( □ (init w)) ⟶
           ○(□ (init w); □ (init w)) ∧ ¬( ○(□ (init w)))
        using 4 5 by fastforce
 hence  7: ⊢ □ (init w); □ (init w) ⟶ □ (init w)
       by (rule NextContra)
 have  11: ⊢ □ (init w) = ((init w) ∧ □ (init w))
       by (rule BoxEqvAndBox)
 have  12: ⊢ empty ; □ (init w) = □ (init w)
       by (rule EmptyChop)
 have  13: ⊢ ((init w) ∧  empty ); □ (init w) = ((init w) ∧ ( empty ; □ (init w)))
       by (rule StateAndChop)
 have  14: ⊢ □ (init w) = ((init w) ∧  empty ); □ (init w)
       using 11 12 13 by fastforce
 have  15: ⊢ (init w) ∧  empty  ⟶ □ (init w)
       by (rule StateAndEmptyImpBoxState)
 hence 16: ⊢ ((init w) ∧  empty ); □ (init w) ⟶ □ (init w); □ (init w)
       by (rule LeftChopImpChop)
 have  17: ⊢ □ (init w) ⟶ □ (init w); □ (init w)
```

```
    using 14 16 by fastforce
  from 7 17 show ?thesis by fastforce
qed

lemma NotBoxStateImpBoxYieldsNotBox:
 ⊢  ¬( □ (init w)) ⟶ (□ (init w)) yields (¬( □ (init w)))
proof –
 have 1: ⊢ □ (init w); □ (init w) = □ (init w)  by (rule BoxStateChopBoxEqvBox)
 have 2: ⊢ □ (init w) = (¬ ¬( □ (init w)))  by auto
 hence 3: ⊢ □ (init w); □ (init w) = □ (init w); (¬ ¬( □ (init w)))  by (rule RightChopEqvChop)
 have 4: ⊢ ¬( □ (init w)) ⟶ ¬ (□ (init w); (¬ ¬ (□ (init w))))  using 1 3 by auto
 from 4 show ?thesis by (simp add: yields-d-def)
qed

lemma StateEqvBi:
 ⊢  (init w) = bi  (init w)
proof –
  have 1: ⊢ (init w) ⟶ bi  (init w)  by (rule StateImpBi)
  have 2: ⊢ bi  (init w) ⟶ (init w)  by (rule BiElim)
  from 1 2 show ?thesis by fastforce
qed


lemma TrueChopEqvDiamond:
 ⊢  #True; f = ◇ f
by (simp add: sometimes-d-def)
```

4.5 Properties of Da and Ba

```
lemma DaEqvDtDi:
 ⊢   da  f = ◇ (di  f)
proof –
 have 1: ⊢ #True; (f; #True) = #True; (f; #True)  by auto
 hence 2: ⊢ #True; (f; #True) = #True;  di  f  by (simp add: di-d-def)
 have 3: ⊢ #True;  di  f = ◇( di  f) by (rule TrueChopEqvDiamond)
 have 4: ⊢ #True; (f; #True) = ◇( di  f)  using 2 3 by fastforce
 from 4 show ?thesis by (simp add:da-d-def)
qed

lemma DaEqvDiDt:
 ⊢   da  f =  di (◇ f)
proof –
 have 1: ⊢ #True; f = ◇ f  by (rule TrueChopEqvDiamond)
 hence 2: ⊢ (#True; f); #True = (◇ f); #True  by (rule LeftChopEqvChop)
 hence 3: ⊢ (#True; f); #True =  di( ◇ f)  by (simp add: di-d-def)
 have 4: ⊢ #True; (f; #True) = (#True; f); #True    by (rule ChopAssoc)
 have 5: ⊢ #True; (f; #True) =  di (◇ f)  using 3 4 by fastforce
 from 5 show ?thesis by (simp add: da-d-def)
qed
```

```
lemma DtDiEqvDiDt:
⊢  ◇ (di f) = di (◇ f)
by (metis ChopAssoc di-d-def sometimes-d-def)
```

```
lemma DiamondNotEqvNotBox:
⊢ ◇ (¬ f) = (¬ (□ f))
by (simp add: always-d-def)
```

```
lemma BaEqvBiBt:
⊢  ba f = bi( □ f)
proof −
 have 1: ⊢ da (¬ f) = di( ◇ (¬ f)) by (rule DaEqvDiDt)
 have 2: ⊢ ◇ (¬ f) = (¬( □ f)) by (rule DiamondNotEqvNotBox)
 hence 3: ⊢ di (◇(¬ f)) = di (¬ (□ f)) by (rule DiEqvDi)
 have 4: ⊢ da (¬ f) = di (¬( □ f)) using 1 3 by fastforce
 hence 5: ⊢ (¬ (da (¬ f))) = (¬ ( di (¬( □ f)))) by auto
 hence 6: ⊢ (¬ ( da (¬ f))) = bi( □ f) by (simp add: bi-d-def)
 from 6 show ?thesis by (simp add: ba-d-def)
qed
```

```
lemma DiNotEqvNotBi:
⊢  di (¬ f) = (¬( bi f))
proof −
 have 1: ⊢ bi f = (¬ ( di (¬ f))) by (simp add: bi-d-def)
 from 1 show ?thesis by auto
qed
```

```
lemma NotDiamondNotEqvBox:
⊢ (¬ (◇(¬ f))) = □ f
by (simp add: always-d-def)
```

```
lemma BaEqvBtBi:
⊢  ba f = □ (bi f)
proof −
 have 1: ⊢ da (¬ f) = ◇ (di (¬ f)) by (rule DaEqvDtDi)
 have 2: ⊢ di (¬ f) = (¬ (bi f)) by (rule DiNotEqvNotBi)
 hence 3: ⊢ ◇ (di (¬ f)) = ◇(¬ (bi f)) by (rule DiamondEqvDiamond)
 have 4: ⊢ (¬ (◇(¬ (bi f)))) = □(bi f) by (rule NotDiamondNotEqvBox)
 have 5: ⊢ (¬ ( da (¬ f))) = □(bi f) using 1 2 3 4 by fastforce
 from 5 show ?thesis by (simp add: ba-d-def)
qed
```

```
lemma BtBiEqvBiBt:
⊢  □ (bi f) = bi( □ f)
proof −
 have 1: ⊢  ba f = □ (bi f) by (rule BaEqvBtBi)
 have 2: ⊢  ba f = bi( □ f) by (rule BaEqvBiBt)
 from 1 2 show ?thesis by fastforce
qed
```

```
lemma BoxStateEqvBaBoxState:
 ⊢  □ (init w) =  ba (□ (init w))
proof –
 have  1: ⊢ (init w) = bi  (init w)  by (rule StateEqvBi)
 hence 2: ⊢ □ (init w) = □ (bi  (init w))  by (rule BoxEqvBox)
 have  3: ⊢ □ (bi  (init w)) = bi( □ (init w))   by (rule BtBiEqvBiBt)
 have  4: ⊢ □ (init w) = □(□ (init w))  by (rule BoxEqvBoxBox)
 hence 5: ⊢ bi( □ (init w)) = bi (□(□ (init w)))  by (rule BiEqvBi)
 have  6:  ⊢  ba( □ (init w)) = bi( □(□ (init w)))   by (rule BaEqvBiBt)
 from 2 3 5 6 show ?thesis  by fastforce
qed

lemma BaImpBi:
 ⊢   ba  f ⟶ bi  f
proof –
 have 1: ⊢  ba  f = □(bi  f)  by (rule BaEqvBtBi)
 have 2: ⊢ □(bi  f) ⟶ bi  f  by (rule BoxElim)
 from 1 2 show ?thesis using lift-imp-trans by fastforce
qed

lemma BaImpBt:
 ⊢   ba  f ⟶ □ f
proof –
 have 1: ⊢  ba  f = bi( □ f)  by (rule BaEqvBiBt)
 have 2: ⊢ bi( □ f) ⟶ □ f  by (rule BiElim)
 from 1 2 show ?thesis using lift-imp-trans by fastforce
qed

lemma DiamondImpDa:
 ⊢   ◇ f ⟶  da  f
by (metis DiIntro DiamondImpDiamond da-d-def di-d-def sometimes-d-def)

lemma DiImpDa:
 ⊢   di  f ⟶  da  f
by (metis NowImpDiamond da-d-def di-d-def sometimes-d-def)

lemma BoxAndChopImport:
 ⊢   □ h ∧ f; g ⟶ f; (h ∧ g)
proof –
 have  1: ⊢ h ⟶ g ⟶ (h∧ g)  by auto
 hence 2: ⊢ □ h ⟶ □(g ⟶ (h∧ g))  by (rule ImpBoxRule)
 have  3: ⊢ □(g ⟶ (h∧ g)) ⟶ f; g ⟶ f; (h∧ g)  by (rule BoxChopImpChop)
 from 2 3 show ?thesis by fastforce
qed

lemma BaAndChopImport:
  ⊢   ba  f ∧ (g; g1) ⟶ (f ∧ g); (f ∧ g1)
proof –
 have  1: ⊢ ba  f ⟶ bi  f by (rule BaImpBi)
 have  2: ⊢ bi  f ∧ (g; g1) ⟶ (f ∧ g); g1  by (rule BiAndChopImport)
 have  3: ⊢ ba  f ⟶ □ f by (rule BaImpBt)
 have  4: ⊢ □ f ∧ (f ∧ g); g1 ⟶ (f ∧ g); (f ∧ g1)  by (rule BoxAndChopImport)
 from 1 2 3 4 show ?thesis by fastforce
qed

lemma ChopAndCommute:
⊢   f; (g ∧ g1) = f; (g1 ∧ g)
proof –
 have 1: ⊢ (g ∧ g1) = (g1 ∧ g) by auto
 from 1 show ?thesis by (rule RightChopEqvChop)
qed

lemma ChopAndA:
⊢   f; (g ∧ g1) ⟶ f; g
proof –
 have 1: ⊢ (g ∧ g1) ⟶  g  by auto
 from 1 show ?thesis by (rule RightChopImpChop)
qed

lemma ChopAndB:
⊢   f; (g ∧ g1) ⟶ f; g1
proof –
 have 1: ⊢ (g ∧ g1) ⟶  g1  by auto
from 1 show ?thesis by (rule RightChopImpChop)
qed

lemma BoxStateAndChopEqvChop:
⊢  (□ (init w) ∧ (f; g)) = ((□ (init w) ∧ f); (□ (init w) ∧ g))
proof –
 have  1: ⊢ □ (init w) =  ba( □ (init w))
       by (rule BoxStateEqvBaBoxState)
 have  2: ⊢ ba( □ (init w)) ∧ (f; g) ⟶ (□ (init w) ∧ f); (□ (init w) ∧ g)
       by (rule BaAndChopImport)
 have  3: ⊢ □ (init w) ∧ (f; g) ⟶ (□ (init w) ∧ f); (□ (init w) ∧ g)
       using 1 2  by fastforce
 have 11: ⊢ (□ (init w) ∧ f); (□ (init w) ∧ g) ⟶ (□ (init w)); (□ (init w) ∧ g)
       by (rule AndChopA)
 have 12: ⊢ (□ (init w)); (□ (init w) ∧ g) ⟶ (□ (init w)); (□ (init w))
       by (rule ChopAndA)
 have 13: ⊢ (□ (init w)); (□ (init w)) = □ (init w)
       by (rule BoxStateChopBoxEqvBox)
 have 14: ⊢ (□ (init w) ∧ f); (□ (init w) ∧ g) ⟶ f; (□ (init w) ∧ g)
       by (rule AndChopB)
 have 15: ⊢ f; (□ (init w) ∧ g) ⟶ f; g
       by (rule ChopAndB)
 have 16: ⊢ (□ (init w) ∧ f); (□ (init w) ∧ g) ⟶ □ (init w) ∧ (f; g)
       using  11 12 13 14 15  by fastforce
 from 3 16 show ?thesis by fastforce
qed
```

```
lemma DiEqvNotBiNot:
 ⊢   di  f = (¬( bi (¬  f)))
proof −
 have  1: ⊢ bi (¬  f) = (¬ ( di (¬ ¬  f)))  by (simp add: bi-d-def)
 hence 2: ⊢  di (¬ ¬  f) = (¬( bi (¬  f)))  by auto
 have  3: ⊢ f = (¬ ¬  f)  by auto
 hence 4: ⊢  di  f =  di  (¬ ¬  f)  by (rule DiEqvDi)
 from 2 4 show ?thesis by auto
qed

lemma ChopAndBoxImport:
⊢   f; g ∧ □ h ⟶ f; (g ∧ h)
proof −
 have 1: ⊢ □ h ∧ f; g ⟶ f; (h ∧ g)  by (rule BoxAndChopImport)
 have 2: ⊢ f; (h ∧ g) = f; (g ∧ h)  by (rule ChopAndCommute)
 from 1 2 show ?thesis by fastforce
qed

lemma AndChopAndCommute:
⊢   (f ∧ g); (f1 ∧ g1) = (g ∧ f); (g1 ∧ f1)
proof −
 have 1: ⊢ (f ∧ g); (f1 ∧ g1) = (g ∧ f); (f1 ∧ g1)  by (rule AndChopCommute)
 have 2: ⊢ (g ∧ f); (f1 ∧ g1) = (g ∧ f); (g1 ∧ f1)   by (rule ChopAndCommute)
 from 1 2 show ?thesis by fastforce
qed

lemma ChopImpChop:
 assumes ⊢   f⟶ f1  ⊢ g⟶ g1
 shows   ⊢ f; g ⟶ f1; g1
proof −
 have  1: ⊢ f  ⟶ f1  using assms by auto
 hence 2: ⊢ f; g  ⟶ f1; g  by (rule LeftChopImpChop)
 have  3: ⊢ g  ⟶ g1  using assms by auto
 hence 4: ⊢ f1; g  ⟶ f1; g1  by (rule RightChopImpChop)
 from 2 4 show ?thesis by fastforce
qed

lemma ChopEqvChop:
 assumes ⊢   f= f1 ⊢ g= g1
 shows   ⊢ f; g = f1; g1
proof −
 have  1: ⊢ f = f1  using assms by auto
 hence 2: ⊢ f; g = f1; g  by (rule LeftChopEqvChop)
 have  3: ⊢ g = g1  using assms by auto
 hence 4: ⊢ f1; g = f1; g1  by (rule RightChopEqvChop)
 from 2 4 show ?thesis by fastforce
qed

lemma BoxImpBoxImpBox:
⊢ □ h ⟶ □(g ⟶ □ h ∧ g )
```

```
proof –
 have 1: ⊢ □ h ⟶ (g ⟶ □ h ∧ g )  by auto
 hence 2: ⊢ □(□ h) ⟶ □(g ⟶ □ h ∧ g )  by (rule ImpBoxRule)
 have 3: ⊢ □ h = □(□h)  by (rule BoxEqvBoxBox)
 from 2 3 show ?thesis by fastforce
qed

lemma BoxChopImpChopBox:
 ⊢   □ h ⟶ f; g ⟶ f; (□ h ∧ g)
proof –
 have 1: ⊢ □ h ⟶ □(g ⟶ □ h ∧ g )  by (rule BoxImpBoxImpBox)
 have 2: ⊢ □(g ⟶ □ h ∧ g ) ⟶ f; g ⟶ f; (□ h ∧ g)  by (rule BoxChopImpChop)
 from 1 2 show ?thesis by fastforce
qed

lemma NotChopEqvYieldsNot:
⊢   (¬ (f; g)) = f yields  (¬  g)
proof –
 have  1: ⊢ g = (¬ ¬  g)  by auto
 hence 2: ⊢ f; g = f; (¬ ¬  g)  by (rule RightChopEqvChop)
 hence 3: ⊢ (¬ (f; g)) = (¬ (f; (¬ ¬  g)))  by auto
 from 3 show ?thesis by (simp add: yields-d-def)
qed

lemma NotDiFalse:
 ⊢   ¬ ( di  #False)
proof –
 have  1: ⊢ (init #True) ⟶ bi (init #True)  by (rule StateImpBi)
 hence 2: ⊢ #True ⟶ bi #True  by (auto simp: bi-defs)
 have  3: ⊢ #True by auto
 have  4: ⊢ bi #True using 2 3 MP by auto
 hence 5: ⊢ ¬ ( di (¬ #True))  by (simp add: bi-d-def)
 have  6: ⊢ (¬ #True) =  #False  by auto
 hence 7: ⊢  di (¬ #True) =  di  #False  by (rule DiEqvDi)
 from 5 7 show ?thesis by auto
qed

lemma StateAndEmptyChop:
 ⊢   ((init w) ∧  empty ); f = ((init w) ∧ f)
proof –
 have 1: ⊢ ((init w) ∧  empty ); f = ((init w) ∧  empty ; f)  by (rule StateAndChop)
 have 2: ⊢  empty ; f = f  by (rule EmptyChop)
 from 1 2 show ?thesis by fastforce
qed

lemma StateAndNextChop:
 ⊢   ((init w) ∧ ○ f); g = ((init w) ∧ ○(f; g))
proof –
 have 1: ⊢ ((init w) ∧ ○ f); g = ((init w) ∧ (○ f); g)  by (rule StateAndChop)
 have 2: ⊢ (○ f); g = ○(f; g)  by (rule NextChop)
 from 1 2 show ?thesis by fastforce
qed

lemma NextAndEqvNextAndNext:
⊢ ○ (f ∧ g) = (○ f ∧ ○ g)
by (auto simp: next-defs)

lemma NextStateAndChop:
⊢  ○(((init w) ∧ f); g) = (○ (init w) ∧ ○(f; g))
proof −
 have 1: ⊢ ((init w) ∧ f); g = ((init w) ∧ f; g)  by (rule StateAndChop)
 hence 2: ⊢ ○(((init w) ∧ f); g) = ○((init w) ∧ f; g) by (rule NextEqvNext)
 have 3: ⊢ ○((init w) ∧ f; g) = (○ (init w) ∧ ○(f; g)) by (rule NextAndEqvNextAndNext)
 from 2 3 show ?thesis by fastforce
qed

lemma StateYieldsEqv:
⊢  ((init w) ⟶ (f yields  g)) = ((init w) ∧ f) yields  g
proof −
 have 1: ⊢ ((init w) ∧ f); (¬  g) = ((init w) ∧ f; (¬  g))  by (rule StateAndChop)
 hence 2: ⊢ ((init w) ⟶ ¬ (f; (¬  g))) = (¬ (((init w) ∧ f); (¬  g) )) by auto
 from 2 show ?thesis by (simp add: yields-d-def)
qed

lemma StateAndDi:
⊢  ((init w) ∧  di  f) =  di ((init w) ∧ f)
proof −
 have 1: ⊢ ((init w) ∧ f); #True=  ((init w) ∧ f; #True)  by (rule StateAndChop)
 from 1 show ?thesis by (metis di-d-def inteq-reflection)
qed

lemma DiNext:
⊢  di( ○ f) = ○ (di  f)
proof −
 have 1: ⊢ (○ f); #True = ○(f; #True) by (rule NextChop)
 from 1 show ?thesis by (simp add: di-d-def)
qed

lemma DiNextState:
⊢  di( ○ (init w)) = ○ (init w)
proof −
 have 1: ⊢  di( ○ (init w)) = ○( di  (init w)) by (rule DiNext)
 have 2: ⊢  di  (init w) = (init w) by (rule DiState)
 hence 3: ⊢ ○( di  (init w)) = ○ (init w) by (rule NextEqvNext)
 from 1 3 show ?thesis by fastforce
qed

lemma StateImpBiGen:
 assumes ⊢  (init w)⟶ f
 shows  ⊢ (init w) ⟶ bi  f
```

```
proof –
 have 1: ⊢ (init w) ⟶ f using assms by auto
 hence 2: ⊢ ¬ f ⟶ ¬ (init w) by auto
 hence 3: ⊢ di (¬ f) ⟶ di (¬ (init w)) by (rule DiImpDi)
 hence 4: ⊢ di (¬ f) ⟶ di (init (¬w)) by (metis Initprop(2) inteq-reflection)
 have 5: ⊢ di (init (¬ w)) = (init (¬ w)) by (rule DiState)
 have 6: ⊢ di (¬ f) ⟶ ¬ (init w) using 4 5 using Initprop(2) by fastforce
 hence 7: ⊢ (init w) ⟶ ¬ ( di (¬ f)) by auto
 from 7 show ?thesis by (simp add: bi-d-def)
qed
```

```
lemma ChopAndNotChopImp:
⊢ f; g ∧ ¬ (f; g1) ⟶ f; (g ∧ ¬ g1)
proof –
 have 1: ⊢ g ⟶ (g∧ ¬ g1) ∨ g1 by auto
 hence 2: ⊢ f; g ⟶ f; ((g∧ ¬ g1) ∨ g1) by (rule RightChopImpChop)
 have 3: ⊢ f; ((g∧ ¬ g1) ∨ g1) ⟶ (f; (g∧ ¬ g1)) ∨ (f; g1) by (rule ChopOrImp)
 have 4: ⊢ f; g ⟶ f; (g∧ ¬ g1) ∨ f; g1 using 2 3 MP by fastforce
 from 4 show ?thesis by auto
qed
```

```
lemma ChopAndYieldsImp:
⊢ f; g ∧ f yields g1 ⟶ f; (g ∧ g1)
proof –
 have 1: ⊢ g ⟶ (g∧ g1) ∨ ¬ g1 by auto
 hence 2: ⊢ f; g ⟶ f; ((g∧ g1) ∨ ¬ g1) by (rule RightChopImpChop)
 have 3: ⊢ f; ((g∧ g1) ∨ ¬ g1) ⟶ (f; (g∧ g1)) ∨ (f; (¬ g1)) by (rule ChopOrImp)
 have 4: ⊢ f; g ⟶ f; (g∧ g1) ∨ f; (¬ g1) using 2 3 MP by fastforce
 hence 5: ⊢ f; g ∧ ¬ (f; (¬ g1)) ⟶ f; (g ∧ g1) by auto
 from 5 show ?thesis by (simp add: yields-d-def)
qed
```

```
lemma ChopAndYieldsMP:
⊢ f; g ∧ f yields (g⟶ g1) ⟶ f; g1
proof –
 have 1: ⊢ f; g ∧ f yields (g⟶ g1) ⟶ f; (g ∧ (g ⟶ g1)) by (rule ChopAndYieldsImp)
 have 2: ⊢ g ∧ (g ⟶ g1) ⟶ g1 by auto
 hence 3: ⊢ f; (g ∧ (g ⟶ g1)) ⟶ f; g1 by (rule RightChopImpChop)
 from 1 3 show ?thesis by fastforce
qed
```

```
lemma OrYieldsImp:
 ⊢ (f ∨ f1) yields g = ((f yields g) ∧ (f1 yields g))
proof –
 have 1: ⊢ ((f∨ f1); (¬ g)) = ((f; (¬ g)) ∨ (f1; (¬ g))) by (rule OrChopEqv)
 hence 2: ⊢ (¬ ((f∨ f1); (¬ g))) = (¬ (f; (¬ g)) ∧ ¬(f1; (¬ g))) by auto
 from 2 show ?thesis by (simp add: yields-d-def)
qed
```

```
lemma LeftYieldsImpYields:
```

assumes ⊢ *f* ⟶ *f1*
shows ⊢ (*f1 yields g*) ⟶ (*f yields g*)
proof −
have *1*: ⊢ *f* ⟶ *f1* **using** *assms* **by** *auto*
hence *2*: ⊢ *f*; (¬ *g*) ⟶ *f1*; (¬ *g*) **by** (*rule LeftChopImpChop*)
hence *3*: ⊢ ¬ (*f1*; (¬ *g*)) ⟶ ¬ (*f*; (¬ *g*)) **by** *auto*
from *3* **show** *?thesis* **by** (*simp add*: *yields-d-def*)
qed

lemma *LeftYieldsEqvYields*:
assumes ⊢ *f* = *f1*
shows ⊢ (*f yields g*) = (*f1 yields g*)
proof −
have *1*: ⊢ *f* = *f1* **using** *assms* **by** *auto*
hence *2*: ⊢ *f*; (¬ *g*) = *f1*; (¬ *g*) **by** (*rule LeftChopEqvChop*)
hence *3*: ⊢ (¬ (*f*; (¬ *g*))) = (¬ (*f1*; (¬ *g*))) **by** *auto*
from *3* **show** *?thesis* **by** (*simp add*: *yields-d-def*)
qed

4.6 Properties of Fin

lemma *FinEqvTrueChopAndEmpty*:
⊢ *fin f* = #*True*;(*f* ∧ *empty*)
proof −
have *1*: ⊢*fin f* = □(*empty* ⟶ *f*)
by (*simp add*: *fin-d-def*)
have *2*: ⊢ □(*empty* ⟶ *f*) = (¬(◇(¬(*empty* ⟶ *f*))))
by (*simp add*: *always-d-def*)
have *3*: ⊢ (¬(*empty* ⟶ *f*)) = (¬ *f* ∧ *empty*)
by *auto*
hence *4*: ⊢ ◇(¬(*empty* ⟶ *f*)) = ◇(¬ *f* ∧ *empty*)
using *DiamondEqvDiamond* **by** *blast*
hence *5*: ⊢ ¬(◇(¬(*empty* ⟶ *f*))) = (¬(◇(¬ *f* ∧ *empty*)))
by *auto*
have *6*: ⊢ (¬(◇(¬ *f* ∧ *empty*))) = #*True*;(*f* ∧ *empty*)
using *Finprop*(*4*) *sometimes-d-def* **by** (*metis int-eq int-simps*(*4*))
from *1 2 5 6* **show** *?thesis* **by** *fastforce*
qed

lemma *DiamondFin*:
⊢ ◇(*fin w*) = *fin w*
by (*metis DiamondDiamondEqvDiamond FinEqvTrueChopAndEmpty TrueChopEqvDiamond inteq-reflection*)

lemma *ChopFinExportA*:
⊢ *f*;(*g* ∧ *fin w*) ⟶ *fin w*
using *DiamondFin*
by (*metis ChopAndB ChopImpDiamond inteq-reflection lift-imp-trans*)

lemma *FinImpBox*:

$\vdash \mathit{fin}\ w \longrightarrow \Box(\mathit{fin}\ w)$
by (*metis BoxImpBoxBox fin-d-def*)

lemma *FinAndChopImport*:
$\vdash (\mathit{fin}\ w) \wedge (f;g) \longrightarrow f;((\mathit{fin}\ w) \wedge g)$
proof −
have *1*: $\vdash \mathit{fin}\ w \longrightarrow \Box(\mathit{fin}\ w)$ **by** (*rule FinImpBox*)
hence *2*: $\vdash \mathit{fin}\ w \wedge f;g \longrightarrow \Box(\mathit{fin}\ w) \wedge (f;g)$ **by** *auto*
have *3*: $\vdash \Box(\mathit{fin}\ w) \wedge (f;g) \longrightarrow f;((\mathit{fin}\ w) \wedge g)$ **using** *BoxAndChopImport* **by** *blast*
from *2 3* **show** *?thesis* **using** *MP* **by** *fastforce*
qed

lemma *FinAndChop*:
$\vdash (f;(g \wedge \mathit{fin}\ w)) = (\mathit{fin}\ w \wedge f;g)$
using *FinAndChopImport ChopFinExportA ChopAndA ChopAndCommute* **by** *fastforce*

lemma *ChopAndEmptyEqvEmptyChopEmpty*:
$\vdash ((f;g) \wedge \mathit{empty}) = (f \wedge \mathit{empty});(g \wedge \mathit{empty})$
by (*auto simp*: *empty-defs chop-defs*)

lemma *FinAndEmpty*:
$\vdash ((\mathit{fin}\ w) \wedge \mathit{empty}) = (w \wedge \mathit{empty})$
proof −
have *1*: $\vdash ((\mathit{fin}\ w) \wedge \mathit{empty}) = (\#\mathit{True};(w \wedge \mathit{empty}) \wedge \mathit{empty})$
using *FinEqvTrueChopAndEmpty* **by** *fastforce*
have *2*: $\vdash (\#\mathit{True};(w \wedge \mathit{empty}) \wedge \mathit{empty}) = ((\#\mathit{True} \wedge \mathit{empty});(w \wedge \mathit{empty}))$
using *ChopAndEmptyEqvEmptyChopEmpty*
by (*smt int-eq int-iffD2 lift-and-com Prop10 Prop12*)
have *3*: $\vdash (\#\mathit{True} \wedge \mathit{empty});(w \wedge \mathit{empty}) = (\mathit{empty};(w \wedge \mathit{empty}))$
using *LeftChopEqvChop* **by** *fastforce*
have *4*: $\vdash (\mathit{empty};(w \wedge \mathit{empty})) = (w \wedge \mathit{empty})$
using *EmptyChop* **by** *blast*
from *1 2 3 4* **show** *?thesis* **by** *fastforce*
qed

lemma *AndFinEqvChopAndEmpty*:
$\vdash (f \wedge \mathit{fin}\ g) = f;(g \wedge \mathit{empty})$
proof −
have *1*: $\vdash (f \wedge \mathit{fin}\ g) = (f;\mathit{empty} \wedge \mathit{fin}\ g)$
using *ChopEmpty* **by** (*metis int-eq*)
have *2*: $\vdash (\mathit{fin}\ g \wedge f;\mathit{empty}) = (f;(\mathit{empty} \wedge \mathit{fin}\ g))$
using *FinAndChop* **by** *fastforce*
have *3*: $\vdash (\mathit{empty} \wedge \mathit{fin}\ g) = (\mathit{fin}\ g \wedge \mathit{empty})$
by *auto*
have *4*: $\vdash (\mathit{fin}\ g \wedge \mathit{empty}) = (g \wedge \mathit{empty})$
using *FinAndEmpty* **by** *metis*
have *5*: $\vdash (\mathit{empty} \wedge \mathit{fin}\ g) = (g \wedge \mathit{empty})$
using *3 4* **by** *auto*
hence *6*: $\vdash f;(\mathit{empty} \wedge \mathit{fin}\ g) = f;(g \wedge \mathit{empty})$
using *RightChopEqvChop* **by** *blast*

```
 from 1 2 5 show ?thesis by (metis inteq-reflection lift-and-com)
qed

lemma AndFinEqvChopStateAndEmpty:
⊢  (f ∧  fin  (init w)) = f; ((init w) ∧  empty )
using AndFinEqvChopAndEmpty by blast

lemma FinStateEqvStateAndEmptyOrNextFinState:
⊢  fin  (init w) = (((init w) ∧  empty ) ∨  ○( fin  (init w)))
proof −
 have 1: ⊢ fin (init w) = □( empty ⟶ init w)
     by (simp add: fin-d-def)
 have 2 : ⊢ □(empty ⟶ init w) =
              ((empty ⟶ init w) ∧ wnext (□ (empty ⟶ init w)))
     by (rule BoxEqvAndWnextBox)
 have 3: ⊢ fin (init w) =  ((empty ⟶ init w) ∧ wnext (fin (init w)))
     using 1 2 by (simp add: fin-d-def)
 have 4: ⊢ wnext (fin (init w)) = (empty ∨ ○ (fin (init w)))
     by (rule WnextEqvEmptyOrNext)
 have 5: ⊢ fin (init w) =  ((empty ⟶ init w) ∧ (empty ∨ ○ (fin (init w))))
     using 3 4 by fastforce
 have 6: ⊢ ((empty ⟶ init w) ∧ (empty ∨ ○ (fin (init w)))) =
           (((empty ⟶ init w) ∧ empty) ∨ ((empty ⟶ init w) ∧ ○ (fin (init w))))
     by auto
 have 7: ⊢ ((empty ⟶ init w) ∧ empty) = ((init w) ∧ empty)
     by auto
 have 8: ⊢ ((empty ⟶ init w) ∧ ○ (fin (init w))) =  ○ (fin (init w))
     by (metis 1 BoxElim DiamondFin NextDiamondImpDiamond int-eq lift-and-com
        lift-imp-trans Prop10)
 have 9: ⊢ (((empty ⟶ init w) ∧ empty) ∨ ((empty ⟶ init w) ∧ ○ (fin (init w)))) =
           ((init w) ∧  empty ) ∨  ○( fin  (init w))
     using 7 8 by auto
 from 5 6 8 9 show ?thesis  by fastforce
qed

lemma FinChopEqvOr:
 ⊢  ( fin  (init w)); f = (((init w) ∧ f) ∨  ○(( fin  (init w)); f))
proof −
 have  1: ⊢  fin  (init w) = (((init w) ∧  empty ) ∨  ○( fin  (init w)))
      by (rule FinStateEqvStateAndEmptyOrNextFinState)
 hence 2: ⊢ ( fin  (init w)); f = (((init w) ∧  empty )∨  ○( fin  (init w))); f
      by (rule LeftChopEqvChop)
 have  3: ⊢ (((init w) ∧  empty )∨  ○ (fin  (init w))); f
          = (((init w) ∧  empty ); f ∨  (○ (fin  (init w))); f)
      by (rule OrChopEqv)
 have  4: ⊢ ((init w) ∧  empty ); f = ((init w) ∧ f)
      by (rule StateAndEmptyChop)
 have  5: ⊢ (○ (fin  (init w))); f = ○(( fin  (init w)); f)
      by (rule NextChop)
 from 2 3 4 5 show ?thesis by fastforce
```

```
qed

lemma FinChopEqvDiamond:
⊢  ( fin  (init w)); f = ◇ ((init w) ∧ f)
proof −
 have  1: ⊢ ( fin  (init w)) = (#True;((init w) ∧ empty))
       by (rule FinEqvTrueChopAndEmpty)
 hence 2: ⊢ ( fin  (init w));f = (#True;((init w) ∧ empty));f
       by (rule LeftChopEqvChop)
 have  3: ⊢ #True;(( (init w) ∧ empty);f)  = (#True;((init w) ∧ empty));f
       by (rule ChopAssoc)
 have  4: ⊢ #True;(( (init w) ∧ empty);f)= ◇ ( ( (init w) ∧ empty);f)
       by (simp add: sometimes-d-def)
 have  5: ⊢ ( (init w) ∧ empty);f = ((init w) ∧ f)
       using StateAndEmptyChop by blast
 hence 6: ⊢ ◇ ( ( (init w) ∧ empty);f) = ◇ ( (init w) ∧ f)
       by (rule DiamondEqvDiamond)
 from 2 3 4 6 show ?thesis by fastforce
qed

lemma NotDiamondAndNot:
⊢ ¬( ◇ ( f ∧ ¬ f))
proof −
 have 1: ⊢ (¬( ◇ ( f ∧ ¬ f))) = □(¬(f ∧ ¬f))  using NotDiamondNotEqvBox by fastforce
 have 2: ⊢ ¬(f ∧ ¬f) by simp
 have 3: ⊢ □(¬(f ∧ ¬f)) using 2 by (simp add: BoxGen)
 from 1 3 show ?thesis by fastforce
qed

lemma FinYields:
⊢   ( fin  (init w))  yields  (init w)
proof −
 have 1: ⊢ (fin (init w)); (¬(init w)) = ◇((init w) ∧ ¬(init w))  by (rule FinChopEqvDiamond)
 have 2: ⊢  ¬( ◇((init w) ∧ ¬  (init w)))  by (rule NotDiamondAndNot)
 have 3: ⊢ ¬ (( fin  (init w)); (¬  (init w)))  using 1 2 by fastforce
 from 3 show ?thesis by (simp add: yields-d-def)
qed

lemma ImpAndFinStateOrFinNotState:
⊢ f ⟶ (f ∧ fin (init w)) ∨ fin (¬ (init w))
by (simp add: fin-defs Valid-def)

lemma AndFinChopEqvStateAndChop:
⊢   (f ∧  fin  (init w)); g = f; ((init w) ∧ g)
proof −
 have   1: ⊢ ( fin  (init w)) yields  (init w)
        by (rule FinYields)
 have   2: ⊢ f ∧  fin  (init w) ⟶  fin  (init w)
        by auto
 hence  3: ⊢ ( fin  (init w)) yields  (init w) ⟶ (f ∧  fin  (init w)) yields  (init w)
```

```
    by (rule LeftYieldsImpYields)
  have  4: ⊢ (f ∧  fin  (init w)) yields  (init w)
    using 1 3 MP by fastforce
  have  5: ⊢ (f ∧  fin  (init w)); g ∧ (f ∧  fin  (init w)) yields  (init w)
          ⟶ (f ∧  fin  (init w)); (g ∧ (init w))
    by (rule ChopAndYieldsImp)
  have  6: ⊢ (f ∧  fin  (init w)); g ⟶ (f ∧  fin  (init w)); (g ∧ (init w))
    using 4 5 by fastforce
  have  7: ⊢ (f ∧  fin  (init w)); (g ∧ (init w)) ⟶ f; (g ∧ (init w))
    by (rule AndChopA)
  have  8: ⊢ g ∧ (init w) ⟶ (init w) ∧ g
    by auto
  hence  9: ⊢ f; (g ∧ (init w)) ⟶ f; ((init w) ∧ g)
    by (rule RightChopImpChop)
  have  10: ⊢ (f ∧  fin  (init w)); g ⟶ f; ((init w) ∧ g)
    using 6 7 9 by fastforce
  have  11: ⊢ f ⟶ (f ∧  fin  (init w)) ∨   fin (¬  (init w))
    by (rule ImpAndFinStateOrFinNotState)
  hence 12: ⊢ f; ((init w) ∧ g) ⟶
          ((f ∧  fin  (init w))∨  fin (¬  (init w))); ((init w) ∧ g)
    by (rule LeftChopImpChop)
  have  13: ⊢ ((f ∧  fin  (init w))∨  fin (¬  (init w))); ((init w) ∧ g)
          =
          ((f ∧  fin  (init w)); ((init w) ∧ g) ∨  ( fin (¬  (init w))); ((init w) ∧ g))
    by (rule OrChopEqv)
  have  14: ⊢ ( fin   (init (¬ w))); ((init w) ∧ g) ⟶ ◇(  (init (¬ w)) ∧ ((init w) ∧ g))
    using FinChopEqvDiamond by fastforce
  have 141: ⊢ ¬( ◇(  (init (¬ w)) ∧ ((init w) ∧ g))) ⟶
          ¬ ( ( fin   (init (¬ w))); ((init w) ∧ g))
    using 14 by fastforce
  have  15: ⊢ ¬( ◇(  (init (¬ w)) ∧ ((init w) ∧ g)))
    using NotDiamondAndNot Initprop(2) by (auto simp: sometimes-defs init-defs)
  have 151: ⊢ ¬ ( ( fin   (init (¬ w))); ((init w) ∧ g))
    using 15 141 by fastforce
  have 1511: ⊢ ( fin (¬  (init w))); ((init w) ∧ g) ⟶ #False
    using 151  by (metis Initprop(2) int-simps(14) inteq-reflection)
  have 152: ⊢ (f ∧  fin  (init w)); ((init w) ∧ g) ∨  ( fin (¬  (init w))); ((init w) ∧ g) ⟶
          (f ∧  fin  (init w)); ((init w) ∧ g)
    using 1511 by fastforce
  have  16: ⊢ f; ((init w) ∧ g) ⟶ (f ∧  fin  (init w)); ((init w) ∧ g)
    using 12 13 152 by fastforce
  have  17: ⊢ (f ∧  fin  (init w)); ((init w) ∧ g) ⟶ (f∧  fin  (init w)); g
    by (rule ChopAndB)
  have  18: ⊢ f; ((init w) ∧ g) ⟶ (f ∧  fin  (init w)); g
    using 16 17 by fastforce
  from 10 18 show ?thesis by fastforce
qed

lemma DiAndFinEqvChopState:
⊢   di (f ∧  fin  (init w)) = f; (init w)
```

```
proof −
 have 1: ⊢ (f ∧ fin(init w)); #True = f;((init w) ∧ #True)  by (rule AndFinChopEqvStateAndChop)
 have 2: ⊢ ((init w) ∧ #True) = (init w)  by auto
 hence 3: ⊢ (f; ((init w) ∧ #True)) = (f; (init w)) by (rule RightChopEqvChop)
 have 4: ⊢ (f ∧  fin  (init w)); #True = f; (init w)  using 1 3 by auto
 from 4 show ?thesis by (simp add: di-d-def)
qed

lemma FinNotStateEqvNotFinState:
⊢ fin   (init (¬ w)) = (¬( fin   (init w)))
using FinEqvTrueChopAndEmpty
by (metis (no-types, hide-lams) Finprop(4) Initprop(2) int-eq int-simps(4) int-simps(7) sometimes-d-def)

lemma BiImpFinEqvYieldsState:
⊢   bi (f ⟶  fin  (init w)) = f yields  (init w)
proof −
 have 1: ⊢  di (f ∧ fin (init (¬ w))) = f;   (init (¬ w))
       by (rule DiAndFinEqvChopState)
 have 2: ⊢ (f ∧ fin(init (¬ w)))  = (f ∧ ¬(fin(init w)))
       using FinNotStateEqvNotFinState by fastforce
 have 3: ⊢ (f ∧ ¬ (fin(init w))) = (¬ (f ⟶  fin  (init w)))
       by auto
 have 4: ⊢ (f ∧ fin(init  (¬ w))) = (¬ (f ⟶  fin(init w)))
       using 2 3 by fastforce
 hence 5: ⊢  di (f ∧  fin   (init (¬ w))) =  di (¬ (f ⟶ fin(init w)))
       by (rule DiEqvDi)
 have 6: ⊢  di (¬ (f ⟶  fin  (init w))) = (¬( bi (f ⟶ fin(init w))))
       by (rule DiNotEqvNotBi)
 have 7: ⊢ ¬ (bi (f ⟶  fin  (init w))) = f;(init (¬ w))
       using 1  5 6 Initprop by fastforce
 hence 8: ⊢ bi (f ⟶  fin  (init w)) = (¬ (f; (¬  (init w))))
       by (metis Initprop(2) int-eq int-simps(7))
 from 8 show ?thesis by (simp add: yields-d-def)
qed

lemma StateImpYields:
 assumes ⊢   (init w) ∧ f ⟶  fin  (init w1)
 shows   ⊢ (init w) ⟶ (f yields  (init w1))
proof −
 have 1: ⊢ (init w) ∧ f ⟶  fin  (init w1)  using assms by auto
 hence 2: ⊢ (init w) ⟶ (f ⟶  fin  (init w1))  by auto
 hence 3: ⊢ (init w) ⟶ bi (f ⟶  fin  (init w1))  by (rule StateImpBiGen)
 have 4: ⊢ bi (f ⟶  fin  (init w1)) = f yields  (init w1)  by (rule BiImpFinEqvYieldsState)
 from 3 4 show ?thesis by fastforce
qed

lemma StateAndYieldsImpYields:
 assumes ⊢   (init w) ∧ f ⟶ f1
 shows   ⊢ (init w) ∧ (f1 yields  g) ⟶ (f yields  g)
proof −
```

```
 have 1: ⊢ (init w) ∧ f ⟶ f1 using assms by auto
 hence 2: ⊢ (init w) ∧ (f; (¬ g)) ⟶ f1; (¬ g) by (rule StateAndChopImpChopRule)
 hence 3: ⊢ (init w) ∧ ¬ (f1; (¬ g)) ⟶ ¬ (f; (¬ g)) by auto
 from 3 show ?thesis by (simp add: yields-d-def)
qed

lemma AndYieldsA:
⊢ f yields g ⟶ (f ∧ f1) yields g
proof –
 have 1: ⊢ f ∧ f1 ⟶ f by auto
 from 1 show ?thesis by (rule LeftYieldsImpYields)
qed

lemma AndYieldsB:
 ⊢ f1 yields g ⟶ (f ∧ f1) yields g
proof –
 have 1: ⊢ f ∧ f1 ⟶ f1 by auto
 from 1 show ?thesis by (rule LeftYieldsImpYields)
qed

lemma RightYieldsImpYields:
 assumes ⊢ g⟶ g1
 shows ⊢ (f yields g) ⟶ (f yields g1)
proof –
 have 1: ⊢ g ⟶ g1 using assms by auto
 hence 2: ⊢ ¬ g1 ⟶ ¬ g by auto
 hence 3: ⊢ f; (¬ g1) ⟶ f; (¬ g) by (rule RightChopImpChop)
 hence 4: ⊢ ¬ (f; (¬ g)) ⟶ ¬ (f; (¬ g1)) by auto
 from 4 show ?thesis by (simp add: yields-d-def)
qed

lemma RightYieldsEqvYields:
 assumes ⊢ g= g1
 shows ⊢ (f yields g) = (f yields g1)
proof –
 have 1: ⊢ g = g1 using assms by auto
 hence 2: ⊢ (¬ g) = (¬ g1) by auto
 hence 3: ⊢ f; (¬ g) = f; (¬ g1) by (rule RightChopEqvChop)
 hence 4: ⊢ (¬ (f; (¬ g))) = (¬ (f; (¬ g1))) by auto
 from 4 show ?thesis by (simp add: yields-d-def)
qed

lemma BoxImpYields:
⊢ □ g ⟶ f yields g
proof –
 have 1: ⊢ f; (¬ g) ⟶ ◇(¬ g) by (rule ChopImpDiamond)
 hence 2: ⊢ ¬ (◇(¬ g)) ⟶ ¬ (f; (¬ g)) by auto
 from 2 show ?thesis by (simp add: yields-d-def always-d-def)
qed
```

```
lemma BoxEqvTrueYields:
⊢  □ f = #True yields  f
proof –
 have  1: ⊢ #True; (¬  f) = ◇ (¬  f)  by (rule TrueChopEqvDiamond)
 hence 2: ⊢ (¬ (#True; (¬  f))) = (¬( ◇ (¬  f)))  by auto
 have  3: ⊢ □ f = (¬ ( ◇ (¬  f)))  by (simp add: always-d-def)
 have  4: ⊢ □ f = (¬ (#True; (¬  f)))  using 2 3 by fastforce
 from 4 show ?thesis by (simp add: yields-d-def)
qed

lemma YieldsGen:
 assumes ⊢  g
 shows  ⊢ f yields  g
proof –
  have  1: ⊢ g  using assms by auto
  hence 2: ⊢ □ g  by (rule BoxGen)
  have  3: ⊢ □ g ⟶ f yields  g  by (rule BoxImpYields)
  from 2 3 show ?thesis using MP by fastforce
qed

lemma YieldsAndYieldsEqvYieldsAnd:
 ⊢  ((f yields  g) ∧ (f  yields  g1)) = f yields (g ∧ g1)
proof –
 have  1: ⊢ f; (¬  g ∨  ¬  g1) = ((f; (¬  g)) ∨  (f; (¬  g1)))  by (rule ChopOrEqv)
 hence 2: ⊢ ((f; (¬  g)) ∨  (f; (¬  g1))) = f; (¬  g ∨  ¬  g1)  by auto
 have  3: ⊢ (¬  g ∨  ¬  g1)  =  (¬ (g ∧ g1))  by auto
 hence 4: ⊢ f; (¬  g ∨  ¬  g1)  =  f; (¬ (g ∧ g1))  by (rule RightChopEqvChop)
 have  5: ⊢ (f; (¬  g)) ∨  (f; (¬  g1)) = f; (¬ (g ∧ g1))  using 2 4 by fastforce
 hence 6: ⊢ (¬ (f; (¬  g)) ∧ ¬ (f; (¬  g1))) = (¬ (f; (¬ (g ∧ g1))))  by (auto simp: chop-defs)
 from 6 show ?thesis by (simp add: yields-d-def)
qed

lemma YieldsAndYieldsImpAndYieldsAnd:
⊢  (f yields  g) ∧ (f1 yields  g1) ⟶ (f∧ f1) yields (g∧ g1)
proof –
 have 1: ⊢ f yields  g ⟶ (f ∧ f1) yields  g
      by (rule AndYieldsA)
 have 2: ⊢ f1 yields  g1 ⟶ (f ∧ f1) yields  g1
      by (rule AndYieldsB)
 have 3: ⊢ ((f ∧ f1) yields  g ∧ (f ∧ f1) yields  g1) = (f ∧ f1) yields  (g ∧ g1)
      by (rule YieldsAndYieldsEqvYieldsAnd)
 from 1 2 3 show ?thesis by fastforce
qed

lemma YieldsYieldsEqvChopYields:
⊢  f yields (g yields  h) = (f; g) yields h
proof –
 have  1: ⊢ f; (g; (¬  h)) = (f; g); (¬  h)    by (rule ChopAssoc)
 hence 2: ⊢ f; (g; (¬  h)) = (f; g); (¬  h)  by auto
 have  3: ⊢ g; (¬  h) = (¬ ¬ (g; (¬  h)))  by auto
 hence 4: ⊢ f; (g; (¬ h)) = f; (¬ ¬ (g; (¬ h))) by (rule RightChopEqvChop)
 have 5: ⊢ f; (¬ ¬ (g; (¬ h))) = (f; g); (¬ h) using 2 4 by auto
 hence 6: ⊢ f; (¬ (g yields h)) = (f; g); (¬ h) by (simp add: yields-d-def)
 hence 7: ⊢ (¬ (f; (¬ (g yields h)))) = (¬ ((f; g); (¬ h))) by auto
 from 7 show ?thesis by (simp add: yields-d-def)
qed

lemma EmptyYields:
⊢ empty yields f = f
proof −
 have 1: ⊢ empty ; (¬ f) = (¬ f) by (rule EmptyChop)
 hence 2: ⊢ (¬ ( empty ; (¬ f))) = f by auto
 from 2 show ?thesis by (simp add: yields-d-def)
qed

lemma NextYields:
⊢ (○ f) yields g = wnext (f yields g)
proof −
 have 1: ⊢ (○ f); (¬ g) = ○(f; (¬ g)) by (rule NextChop)
 hence 2: ⊢ (¬ ((○ f); (¬ g))) = (¬ (○(f; (¬ g)))) by auto
 hence 3: ⊢ (○ f) yields g = (¬ (○(f; (¬ g)))) by (simp add: yields-d-def)
 have 4: ⊢ (¬( ○(f; (¬ g)))) = wnext (¬ (f; (¬ g))) by (auto simp: wnext-d-def)
 have 5: ⊢ (○ f) yields g = wnext (¬ (f; (¬ g))) using 3 4 by fastforce
 from 5 show ?thesis by (simp add: yields-d-def)
qed

lemma SkipChopEqvNext:
 ⊢ skip ; f = ○ f
by (simp add: next-d-def)

lemma SkipYieldsEqvWeakNext:
⊢ skip yields f = wnext f
proof −
 have 1: ⊢ skip ; (¬ f) = ○(¬ f) by (rule SkipChopEqvNext)
 hence 2: ⊢ (¬ ( skip ; (¬ f))) = (¬( ○(¬ f))) by auto
 have 3: ⊢ (¬ (○(¬ f))) = wnext f by (auto simp: wnext-d-def)
 have 4: ⊢ (¬ ( skip ; (¬ f))) = wnext f using 2 3 by fastforce
 from 4 show ?thesis by (simp add: yields-d-def)
qed

lemma NextImpSkipYields:
⊢ ○ f ⟶ skip yields f
proof −
 have 1: ⊢ ○ f ⟶ wnext f using WnextEqvEmptyOrNext by fastforce
 have 2: ⊢ skip yields f = wnext f by (rule SkipYieldsEqvWeakNext)
 from 1 2 show ?thesis by fastforce
qed

lemma MoreEqvSkipChopTrue:
⊢ more = skip ; #True
```

```
proof –
 have 1: ⊢ skip ; #True = ○#True by (rule SkipChopEqvNext)
 hence 2: ⊢ ○#True = skip ; #True by auto
 from 2 show ?thesis by (simp add: more-d-def)
qed

lemma MoreChopImpMore:
⊢ more ; f ⟶ more
proof –
 have 1: ⊢ (○#True); f = ○(#True; f) by (rule NextChop)
 have 2: ⊢ ○(#True; f) ⟶ more by (auto simp: more-defs next-defs)
 have 3: ⊢ (○#True; f) ⟶ more using 1 2 by fastforce
 from 3 show ?thesis by (metis more-d-def)
qed

lemma ChopMoreImpMore:
⊢ f; more ⟶ more
proof –
 have 1: ⊢ f; more ⟶ ◇ more by (rule ChopImpDiamond)
 have 2: ⊢ ◇ more ⟶ more by (auto simp: more-defs sometimes-defs)
 from 1 2 show ?thesis by fastforce
qed

lemma MoreChopEqvNextDiamond:
⊢ more ; f = ○(◇ f)
proof –
 have 1: ⊢ more ; f = (○ #True); f by (simp add: more-d-def)
 have 2: ⊢ (○#True); f = ○(#True; f) by (rule NextChop)
 have 3: ⊢ more ; f = ○(#True; f) using 1 2 by fastforce
 from 3 show ?thesis by (simp add: sometimes-d-def)
qed

lemma WeakNextBoxImpMoreYields:
⊢ more yields f = wnext( □ f)
proof –
 have 1: ⊢ more ; (¬ f) = ○(◇ (¬f)) by (rule MoreChopEqvNextDiamond)
 have 2: ⊢ ○(◇ (¬f)) = ○(¬(□f)) by (auto simp: always-d-def)
 have 3: ⊢ ○(¬(□f)) = (¬ ( wnext( □ f) )) by (auto simp: wnext-d-def)
 have 4: ⊢ more ; (¬ f) = (¬(more yields f)) by (simp add: yields-d-def)
 from 1 2 3 4 show ?thesis by fastforce
qed

lemma NotEqvYieldsMore:
⊢ (¬ f) = f yields more
proof –
 have 1: ⊢ f; empty = f by (rule ChopEmpty)
 hence 2: ⊢ (¬ (f; empty )) = (¬ f) by auto
 have 3: ⊢ empty = (¬ more) by (auto simp: empty-d-def)
 hence 4: ⊢ f; empty = f; (¬ more) by (rule RightChopEqvChop)
 hence 5: ⊢ (¬ (f; empty )) = (¬ (f; (¬ more ))) by auto
 have 6: ⊢ (¬ f) = (¬ (f; (¬ more) )) using 2 5 by fastforce
 from 6 show ?thesis by (metis yields-d-def)
qed

lemma LeftChopImpMoreRule:
 assumes ⊢ f ⟶ more
 shows ⊢ f; g ⟶ more
proof –
 have 1: ⊢ f ⟶ more using assms by auto
 hence 2: ⊢ f; g ⟶ more ; g by (rule LeftChopImpChop)
 have 3: ⊢ more ; g ⟶ more by (rule MoreChopImpMore)
 from 2 3 show ?thesis using lift-imp-trans by blast
qed

lemma RightChopImpMoreRule:
 assumes ⊢ g ⟶ more
 shows ⊢ f; g ⟶ more
proof –
 have 1: ⊢ g ⟶ more using assms by auto
 hence 2: ⊢ f; g ⟶ f; more by (rule RightChopImpChop)
 have 3: ⊢ f; more ⟶ more by (rule ChopMoreImpMore)
 from 2 3 show ?thesis using lift-imp-trans by blast
qed

lemma NotDiEqvBiNot:
⊢ (¬ ( di f)) = bi (¬ f)
proof –
 have 1: ⊢ f = (¬ ¬ f) by auto
 hence 2: ⊢ di f = di (¬ ¬ f) by (rule DiEqvDi)
 hence 3: ⊢ (¬ ( di f)) = (¬ ( di (¬ ¬ f))) by auto
 from 3 show ?thesis by (simp add: bi-d-def)
qed

lemma ChopImpDi:
⊢ f; g ⟶ di f
proof –
 have 1: ⊢ g ⟶ #True by auto
 hence 2: ⊢ f; g ⟶ f; #True by (rule RightChopImpChop)
 from 2 show ?thesis by (simp add: di-d-def)
qed

lemma TrueEqvTrueChopTrue:
⊢ #True = #True; #True
proof –
 have 1: ⊢ #True; #True ⟶ #True by auto
 have 2: ⊢ #True ⟶ di #True by (rule DiIntro)
 hence 3: ⊢ #True ⟶ #True; #True by (simp add: di-d-def)
 from 1 3 show ?thesis by auto
qed
```

```
lemma DiEqvDiDi:
⊢  di f = di ( di f)
proof –
 have 1: ⊢ #True = #True; #True by (rule TrueEqvTrueChopTrue)
 hence 2: ⊢ f; #True = f; (#True; #True) by (rule RightChopEqvChop)
 have 3: ⊢ f; (#True; #True)= (f; #True); #True  by (rule ChopAssoc)
 have 4: ⊢ f; #True = (f; #True); #True using 2 3 by fastforce
 from 4 show ?thesis by (metis di-d-def)
qed
```

```
lemma BiEqvBiBi:
⊢  bi f = bi( bi f)
proof –
 have 1: ⊢ di (¬ f) = di( di (¬ f)) by (rule DiEqvDiDi)
 have 2: ⊢ di (¬ f) = (¬ ( bi f)) by (rule DiNotEqvNotBi)
 hence 3: ⊢ di ( di (¬ f)) = di (¬ ( bi f)) by (rule DiEqvDi)
 have 4: ⊢ di (¬ f) = di (¬( bi f)) using 1 3 by fastforce
 hence 5: ⊢ (¬ ( di (¬ f))) = (¬ ( di (¬ ( bi f)))) by fastforce
 from 5 show ?thesis by (metis bi-d-def)
qed
```

```
lemma DiOrEqv:
⊢  di (f ∨ g) = (di f ∨ di g)
proof –
 have 1: ⊢ (f∨ g); #True = (f; #True ∨ g; #True) by (rule OrChopEqv)
 from 1 show ?thesis by (simp add: di-d-def)
qed
```

```
lemma DiAndA:
⊢  di (f ∧ g) ⟶ di f
proof –
 have 1: ⊢ (f ∧ g); #True ⟶ f; #True by (rule AndChopA)
 from 1 show ?thesis by (simp add: di-d-def)
qed
```

```
lemma DiAndB:
⊢  di (f ∧ g) ⟶ di g
proof –
 have 1: ⊢ (f ∧ g); #True ⟶ g; #True by (rule AndChopB)
 from 1 show ?thesis by (simp add: di-d-def)
qed
```

```
lemma DiAndImpAnd:
⊢  di (f ∧ g) ⟶ di f ∧ di g
proof –
 have 1: ⊢ di (f ∧ g) ⟶ di f by (rule DiAndA)
 have 2: ⊢ di (f ∧ g) ⟶ di g by (rule DiAndB)
 from 1 2 show ?thesis by fastforce
qed
```

```
lemma DiSkipEqvMore:
 ⊢   di skip = more
proof -
 have 1: ⊢ skip ; #True = ○#True by (rule SkipChopEqvNext)
 have 2: ⊢ ○#True = more by (auto simp: more-d-def)
 have 3: ⊢ skip ; #True = more using 1 2 by fastforce
 from 3 show ?thesis by (simp add: di-d-def)
qed

lemma DiMoreEqvMore:
 ⊢   di more = more
proof -
 have 1: ⊢ di (○ #True ) = ○( di #True) by (rule DiNext)
 have 2: ⊢ ○( di #True) ⟶ more by (auto simp: next-defs di-defs more-defs)
 have 3: ⊢ di( ○ #True) ⟶ more using 1 2 by fastforce
 hence 4: ⊢ di more ⟶ more by (simp add: more-d-def)
 have 5: ⊢ more ⟶ di more by (rule ImpDi)
 from 4 5 show ?thesis by fastforce
qed

lemma DiIfEqvRule:
 assumes ⊢  f = if_i (init w) then g else h
 shows  ⊢ di f = if_i (init w) then ( di g) else (di h)
proof -
 have 1: ⊢ f = if_i (init w) then g else h using assms by auto
 hence 2: ⊢ f; #True = if_i (init w) then (g; #True) else (h; #True) by (rule IfChopEqvRule)
 from 2 show ?thesis by (simp add: di-d-def)
qed

lemma DiEmpty:
 ⊢   di empty
proof -
 have 1: ⊢ #True  by auto
 have 2: ⊢ empty ; #True = #True by (rule EmptyChop)
 have 3: ⊢ empty ; #True using 1 2 by auto
 from 3 show ?thesis by (simp add: di-d-def)
qed

lemma DaNotEqvNotBa:
 ⊢   da (¬ f) = (¬ ( ba f))
proof -
 have 1: ⊢ ba f = (¬ ( da (¬ f))) by (simp add: ba-d-def)
 from 1 show ?thesis by fastforce
qed


lemma DaEqvDa:
 assumes ⊢  f = g
 shows ⊢ da f = da g
using assms using int-eq by force
```

```
lemma DaEqvNotBaNot:
⊢   da  f = (¬ ( ba (¬  f)))
proof −
 have 1: ⊢  ba (¬  f) = (¬ ( da (¬ ¬  f))) by (simp add: ba-d-def)
 hence 2: ⊢  da (¬ ¬  f) = (¬(  ba (¬  f)))  by fastforce
 have 3: ⊢ f = (¬ ¬  f)  by simp
 hence 4: ⊢  da  f =  da  (¬ ¬  f)  by (rule DaEqvDa)
 from 2 4 show ?thesis by simp
qed

lemma BaElim:
⊢   ba  f ⟶ f
proof −
 have 1: ⊢  ba  f = □(bi  f)  by (rule BaEqvBtBi)
 have 2: ⊢ bi  f  ⟶ f  by (rule BiElim)
 hence 3: ⊢ □(bi  f  ⟶ f)  by (rule BoxGen)
 have 4: ⊢ □(bi  f  ⟶ f) ⟶ □(bi  f) ⟶ □ f  by (rule BoxImpDist)
 have 5: ⊢ □(bi  f) ⟶ □ f  using 3 4 MP by fastforce
 have 6: ⊢ □ f  ⟶ f  by (rule BoxElim)
 from 1 5 6 show ?thesis  using BaImpBt lift-imp-trans by metis
qed

lemma DaIntro:
⊢  f ⟶  da  f
proof −
 have 1: ⊢  ba  (¬  f) ⟶ (¬  f)   by (rule BaElim)
 hence 2: ⊢ ¬ ¬  f  ⟶ ¬ (  ba  (¬  f))  by fastforce
 have 3: ⊢ f = (¬ ¬  f)  by simp
 have 4: ⊢  da  f = (¬  ( ba  (¬  f)))  by (rule DaEqvNotBaNot)
 from 2 3 4 show ?thesis by fastforce
qed

lemma BaGen:
 assumes ⊢   f
 shows  ⊢  ba  f
proof −
 have 1: ⊢   f  using assms by auto
 hence 2: ⊢ □ f  by (rule BoxGen)
 hence 3: ⊢ bi( □ f)  by (rule BiGen)
 have 4: ⊢  ba  f = bi (□ f)  by (rule BaEqvBiBt)
 from 3 4 show ?thesis by fastforce
qed

lemma BaImpDist:
⊢  ba (f ⟶ g) ⟶  ba  f ⟶  ba  g
proof −
 have 1: ⊢ bi (f ⟶ g) ⟶ (bi  f ⟶ bi  g)  by (rule BiImpDist)
 hence 2: ⊢ □(bi (f ⟶ g) ⟶ (bi  f ⟶ bi  g))  by (rule BoxGen)
 have 3: ⊢ □(bi (f ⟶ g) ⟶ (bi  f ⟶ bi  g))
```

```
          ⟶
          (□ (bi (f ⟶ g)) ⟶ (□(bi  f) ⟶ □(bi  g)))
          by (meson 2 BoxImpDist MP lift-imp-trans Prop01 Prop05 Prop09)
 have  4: ⊢ □(bi (f ⟶ g)) ⟶ (□(bi  f) ⟶ □(bi  g))  using 2 3 MP by fastforce
 have  5: ⊢  ba (f⟶ g) = □(bi (f ⟶ g))  by (rule BaEqvBtBi)
 have  6: ⊢  ba  f = □(bi  f)  by (rule BaEqvBtBi)
 have  7: ⊢  ba  g = □(bi  g)  by (rule BaEqvBtBi)
 from 4 5 6 7 show ?thesis by fastforce
qed

lemma BaAndEqv:
⊢   ba (f ∧ g) =  (ba  f ∧  ba  g)
proof −
 have  1: ⊢    ba (f ∧ g) =   □(bi (f ∧ g))
      by (rule BaEqvBtBi)
 have  2: ⊢  bi (f ∧ g) = (bi f ∧ bi g)
      by (auto simp: bi-defs)
 hence 3: ⊢ □(bi (f ∧ g)) = □(bi f ∧ bi g)
      using BoxEqvBox by blast
 have  4: ⊢ □(bi f ∧ bi g)= (□(bi f)  ∧ □(bi g))
      by (metis 2 BoxAndBoxEqvBoxRule inteq-reflection)
 have  5: ⊢ba  f = □(bi  f)
      by (rule BaEqvBtBi)
 have  6: ⊢  ba  g = □(bi  g)
      by  (rule BaEqvBtBi)
 from 1 3 4 5 6 show ?thesis by  fastforce
qed

lemma BaImpBaEqvBa:
⊢   ba (f = g) ⟶  (ba  f =  ba  g)
proof −
 have  1: ⊢  ba (f ⟶ g) ⟶  ba  f ⟶  ba  g  by (rule BaImpDist)
 have  2: ⊢  ba (g ⟶ f) ⟶  ba  g ⟶  ba  f  by (rule BaImpDist)
 have  3: ⊢ ba (f = g) = ba ((f ⟶ g) ∧ (g ⟶ f))  by (auto simp: ba-defs)
 have  4: ⊢ ba ((f ⟶ g) ∧ (g ⟶ f)) = (ba((f ⟶ g)) ∧ ba((g ⟶ f)))  by (rule BaAndEqv)
 have  5: ⊢ ((ba  f ⟶  ba  g) ∧ (ba  g ⟶  ba  f)) = (ba  f =  ba  g)  by auto
 from 1 2 3 4 5 show ?thesis  by fastforce
qed

lemma BaImpBa:
 assumes ⊢   f ⟶ g
 shows  ⊢ba  f ⟶  ba  g
using BaGen BaImpDist MP assms by metis

lemma BaEqvBa:
 assumes ⊢   f = g
 shows  ⊢ ba  f =  ba  g
using BaGen BaImpBaEqvBa MP assms by metis

lemma DaImpDa:
 assumes ⊢  f ⟶ g
 shows  ⊢ da  f ⟶  da  g
using assms by (metis DaEqvDtDi DiAndB DiamondImpDiamond inteq-reflection Prop10)

lemma DiamondEqvDiamondDiamond:
⊢ ◇ f = ◇ (◇ f)
proof −
 have 1: ⊢ ◇ (◇ f) = #True;(#True;f)
     by (simp add: sometimes-d-def)
 have 2: ⊢ #True;(#True;f)  = (#True;#True);f
     by (rule ChopAssoc)
 have 3: ⊢ (#True;#True);f  = #True;f
     using LeftChopEqvChop TrueEqvTrueChopTrue by (metis int-eq)
 have 4: ⊢ #True;f  = ◇f
     by (simp add: sometimes-d-def)
 from 1 2 3 4 show ?thesis by fastforce
qed

lemma DaEqvDaDa:
 ⊢  da  f =  da (  da  f)
proof −
 have  1: ⊢  da  f = ◇( di  f)
      by (rule DaEqvDtDi)
 have  2: ⊢  di  f =  (di ( di  f))
      by (rule DiEqvDiDi)
 hence 3: ⊢ ◇ ( di  f) = ◇ (di  (di  f))
      by (rule DiamondEqvDiamond)
 have  4: ⊢ ◇  (di  f) = ◇(◇ (di  (di  f)))
      using DiamondEqvDiamondDiamond DiEqvDiDi using 3 by fastforce
 have  5: ⊢ ◇  (di  (di  f)) =  di (◇  (di  f))
      by (rule DtDiEqvDiDt)
 hence 6: ⊢ ◇(◇  (di  (di  f))) = ◇ (di (◇  (di  f)))
      by (rule DiamondEqvDiamond)
 have  7: ⊢  da  f = ◇ (di( ◇ ( di  f)))
      using 1 3 4 6 by fastforce
 have  8: ⊢  da (◇ ( di  f)) = ◇( di (◇  (di  f)))
      by (rule DaEqvDtDi)
 have  9: ⊢  da ( da  f) =  da (◇  (di  f))
      using 1 by (rule DaEqvDa)
 from 7 8 9 show ?thesis by fastforce
qed


lemma BaEqvBaBa:
⊢   ba  f =  ba  (ba  f)
proof −
 have 1: ⊢  da (¬  f) =  da  (da (¬  f)) by (rule DaEqvDaDa)
 have 2: ⊢  da  (da (¬  f)) = (¬  (ba (¬  (da (¬  f))))) by (rule DaEqvNotBaNot)
 have 3: ⊢ (¬  (da  (da (¬  f)))) =  ba (¬  (da (¬  f))) by (auto simp: ba-d-def)
 have 4: ⊢ (¬  (da (¬  f))) =  ba (¬  (da (¬  f))) using 1 2 3 by fastforce
 from 4 show ?thesis by (metis ba-d-def)
qed



lemma BaLeftChopImpChop:
⊢   ba (f⟶ f1) ⟶ f; g ⟶ f1; g
proof −
 have 1: ⊢  ba (f ⟶ f1) ⟶ bi (f ⟶ f1) by (rule BaImpBi)
 have 2: ⊢ bi (f ⟶ f1) ⟶ f; g ⟶ f1; g by (rule BiChopImpChop)
 from 1 2 show ?thesis by fastforce
qed

lemma BaRightChopImpChop:
⊢   ba (g⟶ g1) ⟶ f; g ⟶ f; g1
proof −
 have 1: ⊢  ba (g ⟶ g1) ⟶ □(g ⟶ g1) by (rule BaImpBt)
 have 2: ⊢ □(g ⟶ g1) ⟶ f; g ⟶ f; g1 by (rule BoxChopImpChop)
 from 1 2 show ?thesis by fastforce
qed

lemma ChopAndBaImport:
 ⊢   (f; f1) ∧  ba  g ⟶ (f ∧ g); (f1 ∧ g)
proof −
 have 1: ⊢  ba  g ∧ (f; f1) ⟶ (g ∧ f); (g ∧ f1) by (rule BaAndChopImport)
 have 2: ⊢ (g ∧ f); (g ∧ f1) = (f ∧ g); (f1 ∧ g) by (rule AndChopAndCommute)
 from 1 2 show ?thesis by fastforce
qed

lemma BaImpBaImpBaAnd:
⊢ ba h ⟶ ba(g ⟶ ba h ∧ g )
proof −
 have  1: ⊢ ba h ⟶ (g ⟶ ba h ∧ g )  by fastforce
 hence 2: ⊢ ba(ba h) ⟶ ba(g ⟶ ba h ∧ g )  by (rule BaImpBa)
 have  3: ⊢ ba h = ba(ba h)  by (rule BaEqvBaBa)
 from 2 3 show ?thesis by fastforce
qed



lemma BaChopImpChopBa:
⊢   ba  f ⟶ g; g1 ⟶ g; (( ba  f) ∧ g1)
proof −
 have 1: ⊢  ba  f ⟶  ba (g1 ⟶ (ba f) ∧ g1 ) by (rule BaImpBaImpBaAnd)
 have 2: ⊢  ba (g1 ⟶  ba  f ∧ g1 ) ⟶ g; g1 ⟶ g; ( ba  f ∧ g1) by (rule BaRightChopImpChop)
 from 1 2 show ?thesis by fastforce
qed

lemma DiNotBaImpNotBa:
 ⊢   di (¬ ( ba  f)) ⟶ ¬ ( ba  f)
proof −
```

```
 have 1: ⊢ ba f = ba( ba f) by (rule BaEqvBaBa)
 have 2: ⊢ ba ( ba f) ⟶ bi ( ba f) by (rule BaImpBi)
 have 3: ⊢ ba f ⟶ bi ( ba f) using 1 2 by fastforce
 hence 4: ⊢ ba f ⟶ ¬ ( di (¬ ( ba f))) by (simp add: bi-d-def)
 from 4 show ?thesis by fastforce
qed

lemma NotBaChopImpNotBa:
⊢ (¬ ( ba f)); g ⟶ ¬ ( ba f)
proof −
 have 1: ⊢ (¬ ( ba f)); g ⟶ di (¬ ( ba f)) by (rule ChopImpDi)
 have 2: ⊢ di (¬ ( ba f)) ⟶ ¬ ( ba f) by (rule DiNotBaImpNotBa)
 from 1 2 show ?thesis using lift-imp-trans by blast
qed

lemma DiamondFinImpFin:
 ⊢ ◇ (fin f) ⟶ fin f
proof −
 have 1: ⊢ fin f = #True;(f ∧ empty)
      by (rule FinEqvTrueChopAndEmpty)
 hence 2: ⊢ ◇ (fin f) = #True;(#True;(f ∧ empty))
      by (metis ChopEqvChop TrueEqvTrueChopTrue inteq-reflection sometimes-d-def)
 have 3: ⊢ #True;(#True;(f ∧ empty)) = (#True;#True);(f ∧ empty)
      by (rule ChopAssoc)
 have 4: ⊢ (#True;#True);(f ∧ empty) = #True;(f ∧ empty)
      using TrueEqvTrueChopTrue using LeftChopEqvChop by (metis int-eq)
 from 1 2 3 4 show ?thesis by fastforce
qed

lemma ChopFinImpFin:
 ⊢ f; fin (init w) ⟶ fin (init w)
proof −
 have 1: ⊢ f; fin (init w) ⟶ ◇ ( fin (init w)) by (rule ChopImpDiamond)
 have 2: ⊢ ◇ (fin (init w)) ⟶ fin (init w) by (rule DiamondFinImpFin)
 from 1 2 show ?thesis using lift-imp-trans by blast
qed


lemma FinImpYieldsFin:
⊢ fin (init w) ⟶ f yields ( fin (init w))
proof −
 have 1: ⊢ f; fin (init (¬ w)) ⟶ fin (init (¬ w))
      by (rule ChopFinImpFin)
 have 2: ⊢ fin (init (¬ w)) = (¬ ( fin (init w)))
      using FinNotStateEqvNotFinState by blast
 hence 3: ⊢ f; fin (init (¬ w)) = f; (¬ ( fin (init w)))
      by (rule RightChopEqvChop)
 have 4: ⊢ f; (¬ ( fin (init w))) ⟶ ¬ ( fin (init w))
      using 1 2 3 by fastforce
 hence 5: ⊢ fin (init w) ⟶ ¬ (f; (¬ ( fin (init w))))
```

```
      by fastforce
  from 5 show ?thesis by (simp add: yields-d-def)
qed


lemma ChopAndFin:
  ⊢ ((f; g) ∧  fin  (init w)) = f; (g ∧  fin  (init w))
proof −
  have  1: ⊢  fin  (init w) ⟶ f yields ( fin  (init w))
      by (rule FinImpYieldsFin)
  hence 2: ⊢ (f; g) ∧  fin  (init w) ⟶ (f; g) ∧ f yields  ( fin  (init w))
      by auto
  have  3: ⊢ (f; g) ∧ f yields  ( fin  (init w)) ⟶ f; (g ∧  fin  (init w))
      by (rule ChopAndYieldsImp)
  have  4: ⊢ (f; g) ∧  fin  (init w) ⟶ f; (g ∧  fin  (init w))
      using 2 3  by fastforce
  have 11: ⊢ f; (g ∧  fin  (init w)) ⟶ f; g
     by (rule ChopAndA)
  have 12: ⊢ f; (g ∧  fin  (init w)) ⟶ f;  fin  (init w)
     by (rule ChopAndB)
  have 13: ⊢ f;  fin  (init w) ⟶ ◇ ( fin  (init w))
     by (rule ChopImpDiamond)
  have 14: ⊢ ◇( fin  (init w)) ⟶  fin  (init w)
     by (rule DiamondFinImpFin)
  have 15: ⊢ f; (g ∧  fin  (init w)) ⟶ (f; g) ∧  fin  (init w)
     using 11 12 13 14 by fastforce
  from 4 15 show ?thesis by fastforce
qed

lemma ChopAndNotFin:
  ⊢  (f; g ∧ ¬ ( fin  (init w))) = f; (g ∧ ¬ ( fin  (init w)))
proof −
  have  1: ⊢ (f; g ∧  fin   (init (¬ w))) = f; (g ∧  fin   (init (¬ w)))
      by (rule ChopAndFin)
  have  2: ⊢  fin   (init (¬ w)) = (¬  ( fin  (init w) ))
      using FinNotStateEqvNotFinState by blast
  hence 3: ⊢ (g ∧  fin   (init (¬ w))) = (g ∧ ¬(  fin  (init w)))
      by auto
  hence 4: ⊢ f; (g ∧  fin    (init (¬ w))) = f; (g ∧ ¬ ( fin  (init w)))
      by (rule RightChopEqvChop)
  from 1 2 4 show ?thesis by fastforce
qed

lemma FinChopChain:
⊢  ((init w)⟶  fin  (init w1)); ((init w1) ⟶  fin  (init w2))
      ⟶ ((init w) ⟶  fin  (init w2))
proof −
  have  1: ⊢  (init w) ∧  ((init w)⟶  fin  (init w1)); ((init w1) ⟶  fin  (init w2))
          ⟶
          ( (init w) ∧ ((init w)⟶ fin  (init w1))); ((init w1)⟶  fin  (init w2))
      by (rule StateAndChopImport)
  have 2: ⊢ (init w) ∧ ((init w) ⟶ fin (init w1)) ⟶ fin (init w1)
      by auto
  have 3: ⊢ ((init w) ∧ ((init w)⟶ fin (init w1))); ((init w1) ⟶ fin (init w2))
          ⟶
          ( fin (init w1)); ((init w1) ⟶ fin (init w2))
      using 2 by (rule LeftChopImpChop)
  have 4: ⊢ ( fin (init w1)); ((init w1) ⟶ fin (init w2)) =
           ◇((init w1) ∧ ((init w1) ⟶ fin (init w2)))
      by (rule FinChopEqvDiamond)
  have 41: ⊢ ((init w1) ∧ ((init w1) ⟶ fin (init w2))) ⟶ fin (init w2)
      by auto
  have 42: ⊢ ◇((init w1) ∧ ((init w1) ⟶ fin (init w2))) ⟶ ◇ ( fin (init w2))
      using 41 DiamondImpDiamond by blast
  have 5: ⊢ ◇( fin( init w2)) ⟶ fin (init w2)
      using DiamondFinImpFin by blast
  have 6: ⊢ (init w) ∧ ((init w)⟶ fin (init w1)); ((init w1) ⟶ fin (init w2))
             ⟶ fin (init w2)
      using 1 3 4 5 42 by fastforce
  from 6 show ?thesis by fastforce
qed

lemma ChopRule:
  assumes ⊢ (init w) ∧ f ⟶ fin (init w1)
          ⊢ (init w1)∧ f1 ⟶ fin (init w2)
  shows ⊢ (init w) ∧ (f; f1) ⟶ fin (init w2)
proof −
  have 1: ⊢ (init w) ∧ (f; f1) ⟶ ((init w) ∧ f); f1 by (rule StateAndChopImport)
  have 2: ⊢ (init w) ∧ f ⟶ fin (init w1) using assms by auto
  hence 3: ⊢ ((init w) ∧ f); f1 ⟶ ( fin (init w1)); f1 by (rule LeftChopImpChop)
  have 4: ⊢ ( fin (init w1)); f1 = ◇((init w1) ∧ f1) by (rule FinChopEqvDiamond)
  have 5: ⊢ (init w1) ∧ f1 ⟶ fin (init w2) using assms by auto
  hence 6: ⊢ ◇((init w1) ∧ f1) ⟶ ◇ (fin (init w2)) by (rule DiamondImpDiamond)
  have 7: ⊢ ◇( fin (init w2)) ⟶ fin (init w2) using DiamondFinImpFin by blast
  from 1 3 4 6 7 show ?thesis by fastforce
qed


lemma ChopRep:
  assumes ⊢ (init w) ∧ f ⟶ f1 ∧ fin (init w1)
          ⊢ (init w1) ∧ g ⟶ g1
  shows ⊢ (init w) ∧ (f; g) ⟶ (f1; g1)
proof −
  have 1: ⊢ (init w) ∧ f ⟶ f1 ∧ fin (init w1) using assms by auto
  hence 2: ⊢ (init w) ∧ (f; g) ⟶ (f1 ∧ fin (init w1)); g by (rule StateAndChopImpChopRule)
  have 3: ⊢ (f1 ∧ fin (init w1)); g = f1; ((init w1) ∧ g) by (rule AndFinChopEqvStateAndChop)
  have 4: ⊢ (init w1)∧ g ⟶ g1 using assms by auto
  hence 5: ⊢ f1; ((init w1) ∧ g) ⟶ f1; g1 by (rule RightChopImpChop)
  from 2 3 5 show ?thesis by fastforce
qed
```

```
lemma ChopRepAndFin:
 assumes ⊢  (init w) ∧ f ⟶ f1 ∧  fin  (init w1)
         ⊢  (init w1) ∧ g ⟶ g1 ∧  fin  (init w2)
 shows  ⊢  (init w) ∧ (f; g) ⟶ (f1; g1) ∧  fin  (init w2)
proof −
 have 1: ⊢ (init w) ∧ f ⟶ f1 ∧  fin  (init w1) using assms by auto
 have 2: ⊢ (init w1) ∧ g ⟶ g1 ∧  fin  (init w2) using assms by auto
 have 3: ⊢ (init w) ∧ (f; g) ⟶ f1; (g1 ∧  fin  (init w2)) using 1 2 by (rule ChopRep)
 have 4: ⊢ f1; (g1 ∧  fin  (init w2)) ⟶ f1; g1 by (rule ChopAndA)
 have 5: ⊢ f1; (g1 ∧  fin  (init w2)) ⟶ f1;  fin  (init w2) by (rule ChopAndB)
 have 6: ⊢ f1;  fin  (init w2) ⟶  fin  (init w2) by (rule ChopFinImpFin)
 from 1 2 3 4 5 6 show ?thesis using ChopRep ChopRule by fastforce
qed

lemma TrueChopMoreEqvMore:
⊢ #True ; more = more
by (metis ChopMoreImpMore NowImpDiamond TrueChopEqvDiamond int-eq int-iffI)


lemma MoreChopLoop:
 assumes ⊢  f ⟶  more ; f
 shows  ⊢ ¬ f
proof −
 have  1: ⊢ f ⟶  more ; f
        using assms by auto
 hence 11: ⊢ ◇ f ⟶ ◇ (more;f)
       by (rule DiamondImpDiamond)
 have 12: ⊢ ◇ (more;f) = #True;(more;f)
       by (simp add: sometimes-d-def)
 have 13: ⊢ #True;(more;f) = (#True;more);f
       by (rule ChopAssoc)
 have 14: ⊢  ◇ (more;f) = more;f
       using TrueChopMoreEqvMore 12 13  by (metis int-eq)
 have  2: ⊢  more ; f = ○(◇ f)
       by  (rule MoreChopEqvNextDiamond)
 have  3: ⊢  ◇ f ⟶ ○(◇ f)
       using 11 14 2 by fastforce
 hence  4: ⊢ ¬ (◇ f)
       by (rule NextLoop)
 have  5: ⊢ ¬ (◇ f) ⟶ ¬ f
       using NowImpDiamond by fastforce
 from 4 5 show ?thesis  using MP by blast
qed

lemma MoreChopContra:
 assumes ⊢  f ∧ ¬  g ⟶ ( more ; (f ∧ ¬  g))
 shows  ⊢ f ⟶ g
proof −
 have 1: ⊢ f ∧ ¬  g ⟶ ( more ; (f ∧ ¬  g)) using assms by auto
 hence 2: ⊢ ¬ (f ∧ ¬  g) by (rule MoreChopLoop)
 from 2 show ?thesis by auto
qed

lemma ChopLoop:
 assumes ⊢   f ⟶ g;f
         ⊢ g ⟶  more
 shows  ⊢ ¬  f
proof –
have  1: ⊢ f ⟶ g; f using assms by auto
have  2: ⊢ g ⟶  more using assms by auto
hence 3: ⊢ g; f ⟶  more ; f by (rule LeftChopImpChop)
have  4: ⊢ f ⟶  more ; f using 1 3 by fastforce
from 4 show ?thesis using MoreChopLoop by auto
qed

lemma ChopContra:
 assumes ⊢   f ∧ ¬  g ⟶ h; f ∧ ¬ (h; g)
          ⊢ h ⟶  more
 shows   ⊢ f ⟶ g
proof –
have  1: ⊢ f ∧ ¬  g ⟶ h; f ∧ ¬ (h; g) using assms by auto
have  2: ⊢ h ⟶  more using assms by auto
have  3: ⊢ h; f ∧ ¬ (h; g) ⟶ h; (f ∧ ¬  g) by (rule ChopAndNotChopImp)
have  4: ⊢ h; (f ∧ ¬  g) ⟶  more ; (f ∧ ¬  g) using 2 by (rule LeftChopImpChop)
have  5: ⊢ f ∧ ¬  g ⟶  more ; (f ∧ ¬  g) using 1 3 4 by fastforce
from 5 show ?thesis using MoreChopContra by auto
qed
```

4.7 Properties of Chopstar and Chopplus

```
lemma EmptyImpCS:
⊢ empty  ⟶ f⋆
proof –
 have 1: ⊢ f⋆ = (empty ∨ (f ∧ more);f⋆) by (rule ChopstarEqv)
 have 2: ⊢ empty ⟶ empty ∨ (f ∧ more);f⋆ by auto
 from 1 2 show ?thesis by fastforce
qed

lemma CSEqvOrChopCS:
⊢   f⋆ =  (empty  ∨  (f; f⋆))
proof –
 have 1: ⊢ f⋆ = (empty ∨ (f ∧ more);f⋆) by (rule ChopstarEqv)
 have 2: ⊢ (f ∧ more);f⋆ ⟶ f;f⋆ by (rule AndChopA)
 have 3: ⊢ f⋆ ⟶  empty  ∨  f; f⋆ using 1 2 by (metis int-iffD1 Prop08)
 have 4: ⊢ empty  ⟶ f⋆ by (rule EmptyImpCS)
 have 5: ⊢ f ⟶ empty ∨ (f ∧ more) by (auto simp: empty-d-def)
 have 6: ⊢ f; f⋆ ⟶ f⋆ ∨  (f ∧  more ); f⋆ using 5 by (rule EmptyOrChopImpRule)
 have 7: ⊢ f⋆ ⟶ empty ∨ (f ∧ more);f⋆ using 1 by fastforce
 have 8: ⊢ f; f⋆ ⟶ empty ∨  (f ∧  more ); f⋆  using 6 7 by fastforce
```

hence *9*: $\vdash f ; f^{\star} \longrightarrow f^{\star}$ **using** *1* **by** *fastforce*
have *10*: $\vdash \mathit{empty} \ \lor \ f ; f^{\star} \longrightarrow f^{\star}$ **using** *9 4* **by** *fastforce*
from *3 10* **show** *?thesis* **by** *fastforce*
qed

lemma *CSAndMoreEqvAndMoreChop*:
$\vdash \ (f^{\star} \land \ \mathit{more}) \ = (f \land \ \mathit{more} \ ); \ f^{\star}$
proof −
have *1*: $\vdash (\ \mathit{empty} \ \lor \ (f \land \ \mathit{more} \ ); \ f^{\star}) \land \ \mathit{more} \longrightarrow (f \land \ \mathit{more} \ ); \ f^{\star}$
by (*auto simp*: *empty-d-def*)
have *2*: $\vdash f^{\star} = (\mathit{empty} \lor (f \land \mathit{more});f^{\star})$
by (*rule ChopstarEqv*)
have *3*: $\vdash f^{\star} \land \ \mathit{more} \ \longrightarrow (f \land \ \mathit{more} \ ); \ f^{\star}$
using *1 2* **by** *fastforce*
have *4*: $\vdash (f \land \ \mathit{more} \ ); \ f^{\star} \longrightarrow f^{\star}$
using *2* **by** *fastforce*
have *5*: $\vdash (f \land \ \mathit{more} \ ) \longrightarrow \mathit{more}$
by *auto*
hence *6*: $\vdash (f \land \ \mathit{more} \ ); \ f^{\star} \longrightarrow \ \mathit{more}$
by (*rule LeftChopImpMoreRule*)
have *7*: $\vdash (f \land \ \mathit{more} \ ); \ f^{\star} \longrightarrow f^{\star} \land \ \mathit{more}$
using *4 6* **by** *fastforce*
from *3 7* **show** *?thesis* **by** *fastforce*
qed

lemma *CSAndMoreImpChopCS*:
$\vdash \ \ f^{\star} \land \ \mathit{more} \ \longrightarrow f ; f^{\star}$
proof −
have *1*: $\vdash (f^{\star} \land \ \mathit{more}) \ = (f \land \ \mathit{more} \ ); \ f^{\star}$ **by** (*rule CSAndMoreEqvAndMoreChop*)
have *2*: $\vdash (f \land \ \mathit{more} \ ); \ f^{\star} \longrightarrow f ; \ f^{\star}$ **by** (*rule AndChopA*)
from *1 2* **show** *?thesis* **by** *fastforce*
qed

lemma *NotAndMoreEqvEmptyOr*:
$\vdash \neg \ (f \land \mathit{more}) = (\mathit{empty} \lor \neg f)$
by (*auto simp*: *empty-d-def*)

lemma *MoreAndEmptyOrEqvMoreAnd*:
$\vdash (\mathit{more} \land (\mathit{empty} \lor \neg f)) = (\mathit{more} \land \neg \ f)$
by (*auto simp*: *empty-d-def*)

lemma *CSMoreNotImpChopCSAndMore*:
$\vdash \ \ f^{\star} \land \ \mathit{more} \ \land \neg \ \ f \ \longrightarrow (f \land \ \mathit{more} \ ); \ (f^{\star} \land \ \mathit{more} \ )$
proof −
have *1*: $\vdash (f^{\star} \land \ \mathit{more}) \ = (f \land \ \mathit{more} \ ); \ f^{\star}$
by (*rule CSAndMoreEqvAndMoreChop*)
have *2*: $\vdash \ \mathit{empty} \ \lor \ \ \mathit{more}$
by (*auto simp*: *empty-d-def*)
hence *3*: $\vdash f^{\star} \longrightarrow \ \mathit{empty} \ \lor \ (f^{\star} \land \ \mathit{more} \ )$

by *auto*
hence *4*: $\vdash (f \wedge \mathit{more}); f^\star \longrightarrow (f \wedge \mathit{more}) \vee ((f \wedge \mathit{more}); (f^\star \wedge \mathit{more}))$
by (*rule ChopEmptyOrImpRule*)
hence *5*: $\vdash (f \wedge \mathit{more}); f^\star \wedge \neg(f \wedge \mathit{more}) \longrightarrow ((f \wedge \mathit{more}); (f^\star \wedge \mathit{more}))$
by *fastforce*
have *6*: $\vdash (f \wedge \mathit{more}); f^\star = ((f \wedge \mathit{more}); f^\star \wedge \mathit{more})$ **using** *1*
by *auto*
have *7*: $\vdash ((f \wedge \mathit{more}); f^\star \wedge \neg(f \wedge \mathit{more})) = ((f \wedge \mathit{more}); f^\star \wedge \mathit{more} \wedge \neg(f \wedge \mathit{more}))$
using *6* **by** *auto*
have *8*: $\vdash (f \wedge \mathit{more}); f^\star \wedge \mathit{more} \wedge \neg f \longrightarrow (f \wedge \mathit{more}); (f^\star \wedge \mathit{more})$
using *5 7* **by** *auto*
have *9*: $\vdash (f^\star \wedge \mathit{more} \wedge \neg f) = ((f^\star \wedge \mathit{more}) \wedge (\mathit{more} \wedge \neg f))$
by *auto*
have *10*: $\vdash ((f^\star \wedge \mathit{more}) \wedge (\mathit{more} \wedge \neg f)) = ((f \wedge \mathit{more}); f^\star \wedge (\mathit{more} \wedge \neg f))$
using *1* **by** *fastforce*
from *1 8 9 10* **show** *?thesis* **by** *fastforce*
qed

lemma *CSAndMoreImpCSChop*:
$\vdash f^\star \wedge \mathit{more} \longrightarrow f^\star; f$
proof −
have *1*: $\vdash (f^\star \wedge \mathit{more}) = (f \wedge \mathit{more}); f^\star$
by (*rule CSAndMoreEqvAndMoreChop*)
have *2*: $\vdash \mathit{empty} \vee \mathit{more}$
by (*auto simp*: *empty-d-def*)
hence *3*: $\vdash f^\star \longrightarrow \mathit{empty} \vee (f^\star \wedge \mathit{more})$
by *auto*
hence *4*: $\vdash (f \wedge \mathit{more}); f^\star \longrightarrow$
$(f \wedge \mathit{more}) \vee ((f \wedge \mathit{more}); (f^\star \wedge \mathit{more}))$
by (*rule ChopEmptyOrImpRule*)
have *5*: $\vdash f^\star \wedge \mathit{more} \wedge \neg f \longrightarrow (f \wedge \mathit{more}); (f^\star \wedge \mathit{more})$
by (*rule CSMoreNotImpChopCSAndMore*)
have *6*: $\vdash f^\star = (\mathit{empty} \vee (f \wedge \mathit{more}); f^\star)$
by (*rule ChopstarEqv*)
hence *7*: $\vdash f^\star; f = (f \vee ((f \wedge \mathit{more}); f^\star); f)$
by (*rule EmptyOrChopEqvRule*)
have *8*: $\vdash (f \wedge \mathit{more}); (f^\star; f) = ((f \wedge \mathit{more}); f^\star); f$
by (*rule ChopAssoc*)
have *9*: $\vdash (f^\star \wedge \mathit{more}) \wedge \neg (f^\star; f) \longrightarrow$
$(f \wedge \mathit{more}); (f^\star \wedge \mathit{more}) \wedge \neg ((f \wedge \mathit{more}); (f^\star; f))$
using *5 7 8* **by** *fastforce*
have *10*: $\vdash f \wedge \mathit{more} \longrightarrow \mathit{more}$
by *auto*
from *9 10* **show** *?thesis* **by** (*rule ChopContra*)
qed

lemma *NotEmptyEqvMore*:
$\vdash (\neg \mathit{empty}) = \mathit{more}$
by (*simp add*: *empty-d-def*)

lemma *NotCSImpMore*:
$\vdash \ \neg\ (f^\star) \longrightarrow \mathit{more}$
proof –
have *1*: $\vdash \mathit{empty} \longrightarrow (f^\star)$ **using** *EmptyImpCS* **by** *blast*
hence *2*: $\vdash \ \neg\ \mathit{empty} \vee (f^\star)$ **by** *fastforce*
from *2* **show** *?thesis* **using** *1 NotEmptyEqvMore* **by** *fastforce*
qed

lemma *CSChopCSImpCS*:
$\vdash \ f^\star;\ f^\star \longrightarrow f^\star$
proof –
have *1*: $\vdash f^\star = (\mathit{empty} \ \vee\ (f \wedge \ \mathit{more}\ );\ f^\star)$
by (*rule ChopstarEqv*)
hence *2*: $\vdash f^\star;\ f^\star = (f^\star \vee \ ((f \wedge \ \mathit{more}\ );\ f^\star);\ f^\star)$
by (*rule EmptyOrChopEqvRule*)
have *21*: $\vdash f^\star;\ f^\star \wedge \neg\ (f^\star) \longrightarrow ((f \wedge \ \mathit{more}\ );\ f^\star);\ f^\star$
using *2* **by** *auto*
have *22*: $\vdash \neg\ (f^\star) = (\neg \mathit{empty} \wedge \neg\ ((f \wedge \ \mathit{more}\ );\ f^\star))$
using *1* **by** *fastforce*
have *23*: $\vdash \ \neg\ (f^\star) \longrightarrow \neg\ ((f \wedge \ \mathit{more}\ );\ f^\star)$
using *2 22* **by** *fastforce*
have *24*: $\vdash \ f^\star;\ f^\star \wedge \neg\ (f^\star) \longrightarrow \neg\ (f^\star)$
by *auto*
have *25*: $\vdash \ f^\star;\ f^\star \wedge \neg\ (f^\star) \longrightarrow \neg\ ((f \wedge \ \mathit{more}\ );\ f^\star)$
using *23 24 MP* **by** *auto*
have *3*: $\vdash f^\star;\ f^\star \wedge \neg\ (f^\star) \longrightarrow ((f \wedge \ \mathit{more}\ );\ f^\star);\ f^\star \wedge \neg\ ((f \wedge \ \mathit{more}\ );\ f^\star)$
using *21 25* **by** *fastforce*
have *4*: $\vdash (f \wedge \ \mathit{more}\ );(\ f^\star;\ f^\star) = ((f \wedge \ \mathit{more}\ );\ f^\star);\ f^\star$
by (*rule ChopAssoc*)
have *5*: $\vdash f^\star;\ f^\star \wedge \neg\ (f^\star) \longrightarrow (f \wedge \ \mathit{more}\ );\ (f^\star;\ f^\star) \wedge \neg\ ((f \wedge \ \mathit{more}\ );\ f^\star)$
using *3 4* **by** *fastforce*
have *6*: $\vdash f \wedge \ \mathit{more} \ \longrightarrow \ \mathit{more}$
by *auto*
from *5 6* **show** *?thesis* **using** *ChopContra* **by** *blast*
qed

lemma *ImpChopPlus*:
$\vdash \ f \longrightarrow f;f^\star$
proof –
have *1*: $\vdash f^\star = (\mathit{empty} \vee \ f;\ f^\star)$ **by** (*rule CSEqvOrChopCS*)
hence *2*: $\vdash f;f^\star = (f;\mathit{empty} \vee \ f;\ (f;f^\star))$ **using** *ChopOrEqvRule* **by** *blast*
have *3*: $\vdash f;\mathit{empty} = f$ **using** *ChopEmpty* **by** *blast*
from *2 3* **show** *?thesis* **by** *fastforce*
qed

lemma *ImpCS*:
$\vdash \ f \longrightarrow f^\star$
proof –
have *1*: $\vdash f \ \longrightarrow f;f^\star$ **by** (*rule ImpChopPlus*)

```
 hence 2: ⊢ f  ⟶  empty  ∨  f;f⋆  by auto
 from 2 show ?thesis using CSEqvOrChopCS by fastforce
qed

lemma CSChopImpCS:
⊢  f⋆; f ⟶ f⋆
proof −
 have  1: ⊢ f ⟶ f⋆  by (rule ImpCS)
 hence 2: ⊢ f⋆; f ⟶ f⋆; f⋆  by (rule RightChopImpChop)
 have  3: ⊢ f⋆; f⋆ ⟶ f⋆  by (rule CSChopCSImpCS)
 from 2 3 show ?thesis using lift-imp-trans by blast
qed


lemma ChopPlusImpCS:
 ⊢  f;f⋆  ⟶ f⋆
proof −
 have 1: ⊢  f;f⋆ ⟶ empty  ∨ f;f⋆ by auto
 from 1 show ?thesis using CSEqvOrChopCS by fastforce
qed

lemma CSChopEqvOrChopPlusChop:
⊢  f⋆; g = (g ∨  (f;f⋆) ; g)
proof −
 have 1: ⊢ f⋆ =  (empty  ∨  f;f⋆)  by (rule CSEqvOrChopCS)
 from 1 show ?thesis using EmptyOrChopEqvRule by blast
qed

lemma CSElim:
 assumes ⊢   empty  ⟶ g
         ⊢ (f ∧  more ); g ⟶ g
 shows  ⊢ f⋆ ⟶ g
proof −
 have  1: ⊢ f⋆ =  (empty  ∨  (f ∧  more ); f⋆)
       by (rule ChopstarEqv)
 have  2: ⊢  empty  ⟶ g
       using assms by blast
 have  3: ⊢ (f ∧  more ); g ⟶ g
       using assms by blast
 have 31: ⊢  ¬ g  ⟶ more
       using 2 by (auto simp: empty-d-def)
 have 32: ⊢ ¬ g  ⟶ ¬ ((f ∧  more ); g)
       using 3 by fastforce
 have 33: ⊢ f⋆ ∧ more ⟶ (f ∧  more ); f⋆
       using 1 using CSAndMoreEqvAndMoreChop by fastforce
 have 34: ⊢ f⋆ ∧ ¬  g ⟶ f⋆ ∧ more
       using 31 by auto
 have 35: ⊢ f⋆ ∧ ¬  g ⟶ (f ∧  more ); f⋆
       using 33 34 by fastforce
 have 36: ⊢ f⋆ ∧ ¬  g ⟶ ¬ ((f ∧  more ); g)
```

```
     using 32 by auto
 have 4: ⊢ f⋆ ∧ ¬ g ⟶ (f ∧ more ); f⋆ ∧ ¬ ((f ∧ more ); g)
     using 35 36 by fastforce
 have 5: ⊢ f ∧ more ⟶ more
     by auto
 from 4 5 show ?thesis using ChopContra by blast
qed


lemma CSCSImpCS:
⊢ (f⋆)⋆ ⟶ f⋆
proof −
 have 1: ⊢ empty ⟶ f⋆ by (rule EmptyImpCS)
 have 2: ⊢ (f⋆ ∧ more ); f⋆ ⟶ f⋆; f⋆ by (rule AndChopA)
 have 3: ⊢ f⋆; f⋆ ⟶ f⋆ by (rule CSChopCSImpCS)
 have 4: ⊢ (f⋆ ∧ more ); f⋆ ⟶ f⋆ using 2 3 lift-imp-trans by blast
 from 1 4 show ?thesis using CSElim by blast
qed

lemma RightEmptyOrChopEqv:
⊢ g;( empty ∨ f) = (g ∨ (g; f))
proof −
 have 1: ⊢ g;( empty ∨ f) = (g;empty ∨ g;f) by (rule ChopOrEqv)
 have 2: ⊢ g;empty = g by (rule ChopEmpty)
 from 1 2 show ?thesis by fastforce
qed

lemma RightEmptyOrChopEqvRule:
 assumes ⊢ f = (empty ∨ f1)
 shows ⊢ g;f = (g ∨ (g;f1))
proof −
 have 1: ⊢ f = (empty ∨ f1) using assms by auto
 hence 2: ⊢ g;f = g;(empty ∨ f1) by (rule RightChopEqvChop)
 have 3: ⊢ g;(empty ∨ f1) = (g ∨ (g;f1)) by (rule RightEmptyOrChopEqv)
 from 2 3 show ?thesis by fastforce
qed

lemma ChopPlusEqvOrChopChopPlus:
⊢ (f;f⋆) = (f ∨ f; (f;f⋆))
proof −
 have 1: ⊢ f⋆ = (empty ∨ f;f⋆) by (rule CSEqvOrChopCS)
 from 1 show ?thesis by (rule RightEmptyOrChopEqvRule)
qed

lemma CSAndEmptyEqvEmpty:
⊢ ((f⋆) ∧ empty) = empty
using EmptyImpCS by fastforce

lemma NotAndMoreChopAndEmpty:
⊢ ¬(((f ∧ more);g) ∧ empty)
```

```
by (metis AndChopA ChopEmpty LeftChopImpMoreRule Prop01 empty-d-def int-simps(14)
        int-simps(25) int-simps(4) inteq-reflection lift-and-com)

lemma NotChopAndMoreAndEmpty:
 ⊢ ¬((f;(g∧ more)) ∧ empty)
by (metis (no-types, lifting) NotAndMoreChopAndEmpty REmptyEqvEmpty RMoreEqvMore RevChop
        ReverseEqv inteq-reflection rev-fun1 rev-fun2)

lemma ChopCsAndEmptyEqvAndEmpty:
  ⊢ ((f;f⋆) ∧ empty) = (f ∧ empty)
proof −
 have 1: ⊢ ((f;f⋆) ∧ empty) = (f∧ empty);(f⋆ ∧ empty)
      using ChopAndEmptyEqvEmptyChopEmpty by blast
 have 2: ⊢ (f∧ empty);(f⋆ ∧ empty) = (f∧ empty);empty
      using CSAndEmptyEqvEmpty using RightChopEqvChop by blast
 have 3: ⊢ (f∧ empty);empty = (f∧ empty)
      by (rule ChopEmpty)
 from 1 2 3 show ?thesis by fastforce
qed

lemma AndMoreChopAndMoreEqvAndMoreChop:
 ⊢ ((f ∧ more);g ∧ more) = (f ∧ more);g
using ChopImpDi DiAndB DiMoreEqvMore by fastforce

lemma ChopPlusEqv:
 ⊢ (f;f⋆)  = (f ∨  (f∧  more ); (f;f⋆))
proof −
 have  1: ⊢ f⋆ =  (empty  ∨  (f ∧  more ); f⋆)
       by (rule ChopstarEqv)
 have  2: ⊢ f⋆ =  (empty  ∨ f;f⋆)
       by (rule CSEqvOrChopCS)
 hence 3: ⊢   (empty  ∨  f;f⋆)  = (empty  ∨  (f ∧  more );f⋆)
       using 1 2 by fastforce
 have  4: ⊢ (f ∧  more );(f⋆) = (f ∧  more );(empty  ∨  f;f⋆)
       using 2 using RightChopEqvChop by blast
 hence 5: ⊢   empty  ∨  f;f⋆  = empty  ∨  (f ∧  more );(empty  ∨  f;f⋆)
       using 3 4 by fastforce
 have  6: ⊢ (f ∧  more ); ( empty  ∨  f;f⋆ ) =
           ((f ∧  more );  empty  ∨  (f ∧  more ); (f;f⋆))
       using ChopOrEqv by blast
 have  7: ⊢ (f ∧  more );  empty  = (f ∧  more)
       using ChopEmpty by blast
 have  8: ⊢  (empty  ∨  f;f⋆)  =
            (empty  ∨  (f ∧  more ) ∨  (f ∧  more ); (f;f⋆))
       using 5 6 7 by (metis 2 3 inteq-reflection)
 have  9: ⊢  ((empty  ∨  f;f⋆) ∧ more) =  (f;f⋆ ∧ more)
       by (auto simp: empty-d-def)
 have 10: ⊢ ((empty  ∨  (f ∧  more ) ∨  (f ∧  more ); (f;f⋆)) ∧ more) =
          (((f ∧  more ) ∨  (f ∧  more ); (f;f⋆)) ∧ more)
       by (auto simp: empty-d-def)
 have 11: ⊢ (((f ∧  more ) ∨  (f ∧  more ); (f;f⋆)) ∧ more) =
            ((f ∧  more ) ∨  (f ∧  more ); (f;f⋆))
       using 10 6 7 int-eq
       using AndMoreChopAndMoreEqvAndMoreChop by fastforce
 have 12: ⊢ (f;f⋆ ∧ more) = ((f ∧  more ) ∨  (f ∧  more ); (f;f⋆))
       using 8 9 10 11 by fastforce
 have 13: ⊢ (f;f⋆ ∧ empty) = (f ∧ empty)
       by (rule ChopCsAndEmptyEqvAndEmpty)
 have 14: ⊢ ((f ∧  more ) ∨  (f ∧  more ); (f;f⋆) ∨ (f ∧ empty)) =
            (f ∨  (f ∧  more );(f;f⋆))
       by (auto simp: empty-d-def)
 have 15: ⊢ f;f⋆ = (( f;f⋆ ∧ empty) ∨ ( f;f⋆ ∧ more))
       by (auto simp: empty-d-def)
 from 12 13 14 15 show ?thesis by fastforce
qed


lemma ChopPlusImpChopPlus:
 assumes ⊢   f ⟶ g
 shows ⊢  f;f⋆ ⟶ g;g⋆
proof –
 have 1: ⊢ f ⟶ g
      using assms by auto
 have 2: ⊢ f;f⋆  = (f ∨  (f∧  more ); (f;f⋆))
      by (rule ChopPlusEqv)
 have 3: ⊢ g;g⋆  = (g ∨  (g ∧  more );(g;g⋆))
      by (rule ChopPlusEqv)
 have 4: ⊢  f;f⋆  ∧ ¬ (g;g⋆ ) ⟶ ((f∧  more ); (f;f⋆) ) ∧ ¬ ((g ∧  more ); (g;g⋆) )
      using 1 2 3 by fastforce
 have 5: ⊢ f ∧  more  ⟶ g ∧  more  using 1
      by auto
 have 6: ⊢ (f∧  more ); (f;f⋆)  ⟶ (g ∧  more ); (f;f⋆)
      using 5 by (rule LeftChopImpChop)
 have 7: ⊢ f;f⋆  ∧ ¬ (g;g⋆ ) ⟶
          ((g∧  more ); (f;f⋆) ) ∧ ¬ ((g ∧  more ); (g;g⋆) )
      using 4 6 by fastforce
 have 8: ⊢ g ∧  more  ⟶  more
      by auto
 from 7 8 show ?thesis using ChopContra by blast
qed

lemma ChopChopPlusImpChopPlus:
⊢   f; (f;f⋆)  ⟶ f;f⋆
proof –
 have  1: ⊢  empty  ∨   more  by (auto simp: empty-d-def)
 hence 2: ⊢ f ⟶  empty  ∨  (f ∧  more )  by auto
 hence 3: ⊢ f; (f;f⋆)   ⟶ (f;f⋆)  ∨  (f ∧  more );(f;f⋆)  by (rule EmptyOrChopImpRule)
 have  4: ⊢  f;f⋆ = (f ∨  (f∧  more ); (f;f⋆))  by (rule ChopPlusEqv)
 hence 5: ⊢ (f∧  more ); (f;f⋆)  ⟶ f;f⋆  by auto
 from 3 5 show ?thesis using ChopPlusImpCS RightChopImpChop by blast
```

```
qed

lemma CSImpCS:
 assumes ⊢  f ⟶ g
 shows  ⊢ f⋆ ⟶ g⋆
proof −
 have 1: ⊢ f ⟶ g  using assms by auto
 hence 2: ⊢ f;f⋆ ⟶ g;g⋆  by (rule ChopPlusImpChopPlus)
 hence 3: ⊢  empty ∨ f;f⋆  ⟶  empty ∨  g;g⋆  by auto
 from 2 3 show ?thesis using CSEqvOrChopCS by (metis inteq-reflection)
qed

lemma ChopPlusIntro:
 assumes ⊢  f ∧ ¬ g ⟶ (g ∧ more ); f
 shows  ⊢ f ⟶ g;g⋆
proof −
 have 1: ⊢ f ∧ ¬ g ⟶ (g ∧ more ); f  using assms by auto
 have 2: ⊢ g;g⋆  = (g ∨ (g ∧ more ); (g;g⋆))  by (rule ChopPlusEqv)
 have 3: ⊢ f ∧ ¬ (g;g⋆ ) ⟶
        (g ∧ more ); f ∧ ¬ ((g ∧ more ); (g;g⋆) )  using 1 2 by fastforce
 have 4: ⊢ g ∧ more ⟶ more  by auto
 from 3 4 show ?thesis using ChopContra by blast
qed

lemma ChopPlusElim:
 assumes ⊢  f ⟶ g
        ⊢ (f∧ more ); g ⟶ g
 shows  ⊢ f;f⋆  ⟶ g
proof −
 have  1: ⊢ f;f⋆  = (f ∨ (f∧ more ); (f;f⋆))  by (rule ChopPlusEqv)
 have  2: ⊢ f ⟶ g  using assms by blast
 hence 21: ⊢ ¬ g ⟶ ¬ f  by auto
 have  3: ⊢ (f∧ more ); g ⟶ g  using assms by blast
 hence 31: ⊢ ¬ g ⟶ ¬ ((f∧ more ); g)  by fastforce
 hence 32: ⊢ f;f⋆ ∧ ¬ g ⟶ ¬ ((f∧ more ); g)  by auto
 have  33: ⊢f;f⋆ ∧ ¬ g ⟶ (f ∧ more ); (f;f⋆) using 1 21 by fastforce
 have  4: ⊢ f;f⋆ ∧ ¬ g ⟶
          (f ∧ more ); (f;f⋆) ∧ ¬ ((f∧ more ); g)  using 31 33 by fastforce
 have  5: ⊢ f ∧ more ⟶ more  by auto
 from  4 5 show ?thesis using ChopContra by blast
qed

lemma ChopPlusElimWithoutMore:
 assumes ⊢  f ⟶ g
        ⊢ f; g ⟶ g
 shows  ⊢ f;f⋆  ⟶ g
proof −
 have 1: ⊢ f ⟶ g  using assms by blast
 have 2: ⊢ (f; g) ⟶ g  using assms by blast
 have 3: ⊢ (f ∧ more ); g ⟶ f; g  by (rule AndChopA)
 have 4: ⊢ (f ∧  more ); g ⟶ g  using 2 3 lift-imp-trans  by blast
 from 1 4 show ?thesis using ChopPlusElim by blast
qed
```

```
lemma ChopPlusEqvChopPlus:
 assumes ⊢  f = g
 shows  ⊢ f;f⋆  = g;g⋆
proof –
 have  1: ⊢ f = g  using assms by auto
 hence 2: ⊢ f ⟶ g  by auto
 hence 3: ⊢ f;f⋆  ⟶ g;g⋆  by (rule ChopPlusImpChopPlus)
 have  4: ⊢ g ⟶ f  using 1 by auto
 hence 5: ⊢  g;g⋆ ⟶ f;f⋆  by (rule ChopPlusImpChopPlus)
 from 3 5 show ?thesis by fastforce
qed
```

```
lemma CSEqvCS:
 assumes ⊢  f = g
 shows  ⊢ f⋆ = g⋆
proof –
 have  1: ⊢ f = g  using assms by auto
 hence 2: ⊢ f;f⋆  = g;g⋆ by (rule ChopPlusEqvChopPlus)
 hence 3: ⊢  (empty  ∨ f;f⋆)   =  (empty  ∨  g;g⋆)  by auto
 from 3 show ?thesis using CSEqvOrChopCS by (metis int-eq)
qed
```

```
lemma AndCSA:
⊢  (f ∧ g)⋆ ⟶ f⋆
proof –
 have 1: ⊢ f ∧ g ⟶ f  by auto
 from 1 show ?thesis using CSImpCS by blast
qed
```

```
lemma AndCSB:
⊢  (f ∧ g)⋆ ⟶ g⋆
proof –
 have 1: ⊢ f ∧ g ⟶ g  by auto
 from 1 show ?thesis using CSImpCS by blast
qed
```

```
lemma CSIntro:
 assumes ⊢  f ∧  more  ⟶ (g ∧  more ); f
 shows  ⊢ f ⟶ g⋆
proof –
 have   1: ⊢ f ∧  more  ⟶ (g ∧  more ); f
       using assms by auto
 have   2: ⊢  more  = (¬  empty)
       by (auto simp: empty-d-def)
 have   3: ⊢ f ∧ ¬  empty  ⟶ (g ∧  more ); f
       using 1 2 by fastforce
 have  4: ⊢ g⋆ =  (empty  ∨  (g ∧  more ); g⋆)
       by (rule ChopstarEqv)
 hence 41: ⊢ (¬(empty  ∨  (g ∧  more ); g⋆)) = (¬empty ∧ ¬((g ∧  more ); g⋆))
       by fastforce
 have  411: ⊢ (¬empty ∧ ¬((g ∧  more ); g⋆)) = (more ∧ ¬((g ∧  more ); g⋆))
       using NotEmptyEqvMore by fastforce
 have  42: ⊢ ¬( g⋆)  = (more ∧ ¬((g ∧  more ); g⋆))
       using 4 41 411 by fastforce
 have  43: ⊢f ∧ ¬( g⋆) ⟶ f ∧ more ∧ ¬ ((g ∧  more ); g⋆)
       using 42 by fastforce
 have  44: ⊢ f ∧ more ∧ ¬ ((g ∧  more ); g⋆) ⟶ (g ∧  more ); f ∧ ¬ ((g ∧  more ); g⋆)
       using 3 43 1 by auto
 have  5: ⊢ f ∧ ¬ ( g⋆) ⟶
           (g ∧  more ); f ∧ ¬ ((g ∧  more ); g⋆)
       using 43 44 lift-imp-trans by fastforce
 have  6: ⊢ g ∧  more  ⟶  more
       by auto
 from 5 6 show ?thesis using  ChopContra by blast
qed

lemma CSElimWithoutMore:
 assumes ⊢   empty  ⟶ g
         ⊢ f; g ⟶ g
 shows  ⊢ f⋆ ⟶ g
proof −
 have 1: ⊢  empty  ⟶ g using assms by blast
 have 2: ⊢ f; g ⟶ g  using assms by blast
 have 3: ⊢ (f ∧  more ); g ⟶ f; g  by (rule AndChopA)
 have 4: ⊢ (f ∧  more ); g ⟶ g  using 2 3 lift-imp-trans by blast
 from 1 4 show ?thesis using CSElim by blast
qed

lemma ChopAssocB:
⊢ (f;g);h = f;(g;h)
using ChopAssoc by fastforce

lemma CSChopEqvChopOrRule:
 assumes ⊢   f= (g⋆; h)
 shows  ⊢ f = ((g; f) ∨  h)
proof −
 have  1: ⊢ f = (g⋆; h)  using assms by auto
 have  2: ⊢ g⋆ =  (empty  ∨  (g; g⋆))  by (rule CSEqvOrChopCS)
 hence  3: ⊢ g⋆; h = (h ∨  ((g; g⋆); h))  by (rule EmptyOrChopEqvRule)
 have  4: ⊢ (g; g⋆); h =  g; (g⋆; h)    by (rule ChopAssocB)
 hence 41: ⊢ g⋆; h = (h ∨ g; (g⋆; h)) using 3 by fastforce
 have  5: ⊢ g; f = g; (g⋆; h)  using 1 by (rule RightChopEqvChop)
 hence  6: ⊢ (g⋆; h) = (h ∨  g; f) using 41 by fastforce
 hence 61: ⊢ (g⋆; h) = ((g; f) ∨ h)  by auto
 from 1 61 show ?thesis  by fastforce
qed
```

lemma *CSChopIntroRule*:
assumes $\vdash\ f \wedge \neg\ h \longrightarrow g;\ f$
$\vdash g \longrightarrow\ \mathit{more}$
shows $\vdash f \longrightarrow g^{\star};\ h$
proof –
have *1*: $\vdash f \wedge \neg\ h \longrightarrow g;\ f$
using *assms* **by** *blast*
have *2*: $\vdash g \longrightarrow\ \mathit{more}$
using *assms* **by** *blast*
hence *3*: $\vdash g \longrightarrow g \wedge\ \mathit{more}$
by *auto*
hence *4*: $\vdash g;\ f \longrightarrow (g \wedge\ \mathit{more}\ );\ f$
by (*rule LeftChopImpChop*)
have *5*: $\vdash f \longrightarrow (g \wedge\ \mathit{more}\ );\ f \vee\ h$
using *1 4* **by** *fastforce*
have *6*: $\vdash g^{\star} =\ (\mathit{empty}\ \vee\ (g \wedge\ \mathit{more}\ );\ g^{\star})$
by (*rule ChopstarEqv*)
hence *7*: $\vdash (g^{\star});\ h = (h \vee\ ((g \wedge\ \mathit{more}\ );\ g^{\star});\ h)$
by (*rule EmptyOrChopEqvRule*)
have *8*: $\vdash ((g \wedge\ \mathit{more}\ );\ g^{\star});\ h = (g \wedge\ \mathit{more}\ );\ (g^{\star};\ h)$
by (*rule ChopAssocB*)
have *9*: $\vdash (g^{\star});\ h = (h \vee\ (g \wedge\ \mathit{more}\ );\ (g^{\star};\ h))$
using *7 8* **by** *fastforce*
have *10*: $\vdash f \wedge \neg\ (g^{\star};\ h) \longrightarrow (g \wedge \mathit{more});\ f \wedge \neg\ ((g \wedge \mathit{more});\ (g^{\star};\ h))$
using *5 9* **by** *fastforce*
have *11*: $\vdash g \wedge\ \mathit{more}\ \longrightarrow\ \mathit{more}$
by *fastforce*
from *10 11* **show** *?thesis* **using** *ChopContra* **by** *blast*
qed

lemma *DiamondAndEmptyEqvAndEmpty*:
$\vdash (\Diamond\ f \wedge \mathit{empty}) = (f \wedge \mathit{empty})$
by (*auto simp*: *sometimes-defs empty-defs*)

lemma *InitAndEmptyEqvAndEmpty*:
$\vdash ((\mathit{init}\ w) \wedge \mathit{empty}) = (w \wedge \mathit{empty})$
proof –
have *1*: $\vdash ((\mathit{init}\ w) \wedge \mathit{empty}) = ((w \wedge \mathit{empty});\#\mathit{True} \wedge \mathit{empty})$
by (*metis init-d-def int-eq lift-and-com*)
have *2*: $\vdash ((w \wedge \mathit{empty});\#\mathit{True} \wedge \mathit{empty}) = (w \wedge \mathit{empty});(\#\mathit{True} \wedge \mathit{empty})$
by (*meson AndChopA ChopAndA ChopAndEmptyEqvEmptyChopEmpty lift-imp-trans Prop11 Prop12*)
have *3*: $\vdash (w \wedge \mathit{empty});(\#\mathit{True} \wedge \mathit{empty}) = (w \wedge \mathit{empty});\mathit{empty}$
using *RightChopEqvChop* **by** *fastforce*
have *4*: $\vdash (w \wedge \mathit{empty});\mathit{empty} = (w \wedge \mathit{empty})$
using *ChopEmpty* **by** *blast*
from *1 2 3 4* **show** *?thesis* **by** *fastforce*
qed

```
lemma InitAndNotBoxInitImpNotEmpty:
⊢ init w ∧ ¬( □ (init w)) ⟶ ¬  empty
proof −
 have 1: ⊢ ((init w) ∧ empty) = (w ∧ empty)
     by (rule InitAndEmptyEqvAndEmpty)
 have 2: ⊢ (¬( □ (init w)) ∧ empty) = (◇ (¬ (init w)) ∧ empty)
     by (auto simp: always-d-def)
 have 3: ⊢ (◇ (¬ (init w)) ∧ empty) = (¬ (init w) ∧ empty)
     by (simp add: DiamondAndEmptyEqvAndEmpty)
 have 4: ⊢ (¬ (init w)) = (init (¬ w))  using Initprop(2) by blast
 have 5: ⊢ (¬ (init w) ∧ empty) = (¬ w ∧ empty)
     using 4 InitAndEmptyEqvAndEmpty by (metis inteq-reflection)
 have 6: ⊢ (¬( □ (init w)) ∧ empty) = (¬ w ∧ empty)
     using 2 3 5 by fastforce
 have 7: ⊢ ¬(init w ∧ ¬( □ (init w)) ∧ empty)
     using 1 6 by fastforce
 from 7 show ?thesis by auto
qed

lemma BoxImpTrueChopAndEmpty:
⊢ □ f ⟶ #True;(f ∧ empty)
using BoxAndChopImport Finprop(3) by fastforce

lemma BoxInitAndMoreImpBoxInitAndMoreAndFinInit:
⊢ □( init w) ∧  more  ⟶ (□ (init w) ∧  more ) ∧  fin ( init w)
proof −
 have 1: ⊢ fin ( init w) = #True ; (init w ∧ empty) using FinEqvTrueChopAndEmpty by blast
 have 2: ⊢ □( init w) ⟶ #True;(init w ∧ empty)  by (rule BoxImpTrueChopAndEmpty)
 from 1 2 show ?thesis by fastforce
qed

lemma CSImpBox:
 assumes ⊢ f ⟶  empty  ∨  (□ (init w) ∧  more ); f
 shows  ⊢ init w ∧ f ⟶ □ (init w)
proof −
 have  1: ⊢ f ⟶  empty  ∨  (□( init w) ∧  more ); f
      using assms by auto
 have  2: ⊢ init w ∧ ¬( □ (init w)) ⟶ ¬  empty
      by (rule InitAndNotBoxInitImpNotEmpty)
 have  3: ⊢ init w ∧ f ∧ ¬( □ (init w)) ⟶ (□ (init w) ∧  more ); f
      using 1 2 by fastforce
 have  4: ⊢ □ (init w) ∧  more  ⟶ (□ (init w) ∧  more ) ∧  fin ( init w)
      by (rule BoxInitAndMoreImpBoxInitAndMoreAndFinInit)
 hence 5: ⊢ (□( init w) ∧  more ); f ⟶ ((□ (init w) ∧  more ) ∧  fin ( init w) ); f
      by (rule LeftChopImpChop)
 have  6: ⊢ ((□ (init w) ∧  more ) ∧  fin ( init w) ); f =
           (□( init w) ∧  more ); (init w ∧ f)
      by (rule AndFinChopEqvStateAndChop)
 have  7: ⊢ ¬( □( init w)) ⟶ (□ (init w)) yields (¬( □ (init w)))
```

```
      by (rule NotBoxStateImpBoxYieldsNotBox)
 have 8: ⊢ (□( init w)) yields (¬ (□ (init w))) ⟶
          (□ (init w) ∧  more ) yields (¬( □( init w)))
      by (rule AndYieldsA)
 have 9: ⊢ (□( init w) ∧  more ); (init w ∧ f ) ∧ (□( init w) ∧  more ) yields (¬( □ (init w)))
          ⟶
          (□ (init w) ∧  more ); ((init w ∧ f ) ∧ ¬ (□ (init w)))
      by (rule ChopAndYieldsImp)
 have 10: ⊢ (init w ∧ f ) ∧ ¬( □ (init w)) ⟶
          (□( init w) ∧  more ); ((init w ∧ f ) ∧ ¬( □ (init w)))
      using 3 5 6 7 8 9 by fastforce
 have 11: ⊢ (□( init w) ∧  more ); ((init w ∧ f ) ∧ ¬( □ (init w))) ⟶
           more ; ((init w ∧ f ) ∧ ¬( □ (init w)) )
      by (rule AndChopB)
 have 12: ⊢ (init w ∧ f ) ∧ ¬( □ (init w)) ⟶
          more ; ((init w ∧ f ) ∧ ¬( □ (init w)) )
      using 10 11 by fastforce
 from 12 show ?thesis using MoreChopContra by blast
qed

lemma BoxCSEqvBox:
⊢  (init w ∧ (□( init w))⋆) = □ (init w)
proof −
 have 1: ⊢ (□ (init w))⋆ =  (empty  ∨  (□ (init w) ∧  more ); (□ (init w))⋆)
      by (rule ChopstarEqv)
 hence 2: ⊢ (□ (init w))⋆ ⟶  empty  ∨  (□ (init w) ∧  more ); (□ (init w))⋆
      by fastforce
 hence 3: ⊢ init w ∧ (□ (init w))⋆ ⟶ □ (init w)
      by (rule CSImpBox)
 have 11: ⊢ □ (init w) ⟶ (init w)
      using BoxElim by blast
 have 12: ⊢ □( init w) ⟶ (□ (init w))⋆
      by (rule ImpCS)
 have 13: ⊢ □ (init w) ⟶ init w ∧ (□ (init w))⋆
      using 11 12 by fastforce
 from 3 13 show ?thesis by fastforce
qed

lemma BoxStateAndCSEqvCS:
⊢  (□( init w) ∧ f⋆) = (init w ∧ (□( init w) ∧ f )⋆)
proof −
 have 1: ⊢ □ (init w) ⟶ init w
      using BoxElim by blast
 have 2: ⊢ (f⋆ ∧  more)  = (f ∧  more ); f⋆
      by (rule CSAndMoreEqvAndMoreChop)
 have 3: ⊢ (□( init w) ∧ ((f ∧  more ); f⋆)) =
        ((□ (init w) ∧ f ∧  more ); (□ (init w) ∧ f⋆))
      by (rule BoxStateAndChopEqvChop)
 have 4: ⊢ □ (init w) ∧ f ∧  more  ⟶ (□ (init w) ∧ f ) ∧  more
      by auto
 hence 5: ⊢ (□ (init w) ∧ f ∧  more ); (□ (init w) ∧ f⋆) ⟶
           ((□ (init w) ∧ f) ∧  more ); (□ (init w) ∧ f⋆)
      by (rule LeftChopImpChop)
 have  6: ⊢ (□( init w) ∧ f⋆) ∧  more  ⟶
           ((□ (init w) ∧ f) ∧  more ); (□ (init w) ∧ f⋆)
      using 2 3 5 by fastforce
 hence 7: ⊢ □ (init w) ∧ f⋆ ⟶ (□ (init w) ∧ f)⋆
      by (rule CSIntro)
 have 71: ⊢ init w ∧ □ (init w) ∧ f⋆ ⟶ init w ∧ (□ (init w) ∧ f)⋆
      using 7 by fastforce
 have  8: ⊢ □( init w) ∧ f⋆ ⟶ init w ∧ (□ (init w) ∧ f)⋆
      using 1 71  by fastforce
 have 11: ⊢ (□ (init w) ∧ f)⋆ ⟶ (□ (init w))⋆
      by (rule AndCSA)
 have 12: ⊢ (init w ∧ (□ (init w))⋆) = □ (init w)
      by (rule BoxCSEqvBox)
 have 13: ⊢ (□ (init w) ∧ f)⋆ ⟶ f⋆
      by (rule AndCSB)
 have 14: ⊢ init w ∧ (□ (init w) ∧ f)⋆ ⟶ init w ∧ (□ (init w))⋆ ∧ f⋆
      using 11 13 by fastforce
 have 15: ⊢ init w ∧ (□ (init w))⋆ ∧ f⋆ ⟶ □ (init w) ∧ f⋆
      using 12 by auto
 have 16: ⊢ init w ∧ (□ (init w) ∧ f)⋆ ⟶ □ (init w) ∧ f⋆
      using 14 15 lift-imp-trans by blast
 from 8 16 show ?thesis by fastforce
qed

lemma BaCSImpCS:
⊢   ba (f ⟶ g) ⟶ f⋆ ⟶ g⋆
proof –
 have   1: ⊢ f⋆ =  (empty  ∨  (f ∧  more ); f⋆)
       by (rule ChopstarEqv)
 have   2: ⊢ g⋆ =  (empty  ∨  (g ∧  more ); g⋆)
       by (rule ChopstarEqv)
 have  21: ⊢ ¬(g⋆) =  (¬empty  ∧ ¬( (g ∧  more ); g⋆))
       using 2 by fastforce
 hence 22: ⊢ ¬(g⋆) =  (more  ∧ ¬( (g ∧  more ); g⋆))
       using NotEmptyEqvMore by fastforce
 have   3: ⊢ f⋆ ∧ ¬ (g⋆) ⟶
           (empty  ∨ (f ∧  more ); f⋆) ∧ more ∧ ¬ ((g ∧  more ); g⋆)
       using 1 22 by fastforce
 have  31: ⊢ ((empty  ∨ (f ∧  more ); f⋆) ∧ more) = ((f ∧  more ); f⋆ ∧ more)
       by (auto simp: empty-d-def)
 have  32: ⊢ f⋆ ∧ ¬ (g⋆) ⟶ (f ∧  more ); f⋆ ∧ ¬ ((g ∧  more ); g⋆)
       using 3 31 by fastforce
 have   4: ⊢ (f ⟶ g) ⟶ (f ∧  more  ⟶ g ∧  more )
       by auto
 hence  5: ⊢  ba (f ⟶ g) ⟶  ba (f ∧  more  ⟶ g ∧  more )
       by (rule BaImpBa)
 have   6: ⊢  ba (f ∧  more  ⟶ g ∧  more ) ⟶
```

```
        (f ∧  more ); f⋆ ⟶  (g ∧  more ); f⋆
      by (rule BaLeftChopImpChop)
  have  7: ⊢  ba (f ⟶ g) ∧ (f ∧ more ); f⋆ ⟶ (g ∧  more ); f⋆
      using 5 6 by fastforce
  have  8: ⊢ (g ∧  more ); f⋆ ∧ ¬ ((g ∧  more ); g⋆)
        ⟶  (g ∧  more ); (f⋆ ∧ ¬ (g⋆))
      by (rule ChopAndNotChopImp)
  have  9: ⊢ (g ∧  more ); (f⋆ ∧ ¬ (g⋆)) ⟶  more ; (f⋆ ∧ ¬ (g⋆))
      by (rule AndChopB)
  have  10: ⊢  ba (f ⟶ g) ⟶ more ; (f⋆ ∧ ¬ (g⋆)) ⟶
          more ; ( ba (f ⟶ g) ∧ f⋆ ∧ ¬ (g⋆))
      by (rule BaChopImpChopBa)
  have  11: ⊢  ba (f ⟶ g) ∧ f⋆ ∧ ¬ (g⋆) ⟶
          more ; ( ba (f ⟶ g) ∧ f⋆ ∧ ¬ (g⋆))
      using 32 7 8 9 10 by fastforce
  hence 12: ⊢ ¬ ( (ba (f ⟶ g)) ∧ (f⋆) ∧ (¬ (g⋆)))
      using MoreChopLoop by blast
  from 12 show ?thesis using  MP by fastforce
qed

lemma BaCSEqvCS:
⊢   ba (f = g) ⟶ (f⋆ = g⋆)
proof −
  have 1: ⊢ ba (f = g) = (ba (f ⟶ g) ∧ ba (g ⟶ f))  by (auto simp: ba-defs)
  have 2: ⊢ ba (f ⟶ g) ⟶ (f⋆ ⟶ g⋆)  by (rule BaCSImpCS)
  have 3: ⊢ ba (g ⟶ f) ⟶ (g⋆ ⟶ f⋆)  by (rule BaCSImpCS)
  have 4: ⊢ ba (f = g) ⟶ (f⋆ ⟶ g⋆) ∧  (g⋆ ⟶ f⋆)  using 1 2 3 by fastforce
  have 5: ⊢ ((f⋆ ⟶ g⋆) ∧ (g⋆ ⟶ f⋆)) = (f⋆ = g⋆)  by auto
  from 4 5 show ?thesis by auto
qed

lemma BaAndCSImport:
⊢   ba  f ∧ g⋆ ⟶ (f ∧ g)⋆
proof −
  have  1: ⊢ f ⟶ (g ⟶ f ∧ g)  by auto
  hence 2: ⊢  ba  f ⟶  ba  (g ⟶ f ∧ g) by (rule BaImpBa)
  have  3: ⊢  ba  (g ⟶ f ∧ g) ⟶ g⋆ ⟶ (f ∧ g)⋆  by (rule BaCSImpCS)
  from 2 3 show ?thesis by fastforce
qed

lemma CSSkip:
⊢ skip⋆
by (metis ChopPlusImpCS EmptyImpCS EmptyNextInducta next-d-def)
```

4.8 Properties of While

```
lemma WhileEqvIf:
⊢ while  (init w)  do  f =  ifᵢ  (init w)  then  (f; ( while  (init w)  do  f))  else   empty
proof −
  have  1: ⊢  while  (init w)  do  f = (((init w) ∧ f)⋆ ∧  fin (¬  (init w)))
```

```
     by (simp add: while-d-def)
 have  2: ⊢ (init w ∧ f)⋆ =  (empty  ∨  ((init w ∧ f); (init w ∧ f)⋆))
     by (rule CSEqvOrChopCS)
 have 21: ⊢ (((init w) ∧ f)⋆ ∧  fin (¬  (init w))) =
          ((empty  ∨  ((init w ∧ f); (init w ∧ f)⋆)) ∧  fin (¬  (init w)))
     using 2 by fastforce
 have 22: ⊢ ((empty  ∨  ((init w ∧ f); (init w ∧ f)⋆)) ∧  fin (¬  (init w))) =
         (( empty ∧ fin (¬(init w))) ∨ ( ((init w ∧ f); (init w ∧ f)⋆) ∧  fin (¬  (init w))))
     by auto
 have  3: ⊢  (empty  ∧  fin (¬ ( init w))) = (¬ ( init w) ∧  empty)
     using AndFinEqvChopAndEmpty EmptyChop by (metis int-eq)
 have  4: ⊢ (init w ∧ f); (init w ∧ f)⋆ = (init w ∧ (f; (init w ∧ f)⋆))
     by (rule StateAndChop)
 have 41: ⊢ (((init w ∧ f); (init w ∧ f)⋆) ∧  fin (¬  (init w))) =
          (init w ∧ (f; (init w ∧ f)⋆) ∧  fin (¬  (init w)))
     using 4 by auto
 have 42: ⊢ (init w ∧ (f; (init w ∧ f)⋆) ∧  fin (¬  (init w))) =
          (init w ∧ (f; (init w ∧ f)⋆) ∧  fin   (init (¬ w)))
     using Initprop(2) by (metis StateAndEmptyChop int-eq)
 have  5: ⊢ ((f; ((init w ∧ f)⋆)) ∧  (fin  ( init (¬ w))))
          = (f; ((init w ∧ f)⋆ ∧  (fin  ( init (¬ w)))))
     by (rule ChopAndFin)
 have 51: ⊢ (f; ((init w ∧ f)⋆ ∧  (fin  ( init (¬ w))))) =
          (f; ((init w ∧ f)⋆ ∧  (fin (¬ ( init  w)))))
     using Initprop(2) by (smt RightChopEqvChop int-eq lift-and-com)
 have 52: ⊢ (init w ∧ (f; (init w ∧ f)⋆) ∧  fin (¬  (init w))) =
        (init w ∧ (f; ((init w ∧ f)⋆ ∧  fin (¬ ( init  w)))))
     using 42 5 51 by fastforce
 have  6: ⊢ (f; ((init w ∧ f)⋆ ∧  fin (¬ ( init w)))) = f;  while  (init w)  do  f
     by (simp add: while-d-def)
 have 61: ⊢ (init w ∧ (f; ((init w ∧ f)⋆ ∧  fin (¬ ( init  w))))) =
         (init w ∧  (f;  while  (init w)  do  f))  using 6
     by auto
 have 62: ⊢ ( empty ∧  fin (¬  (init w))) ∨ ( ((init w ∧ f); (init w ∧ f)⋆) ∧  fin (¬  (init w)))
          = (¬ ( init w) ∧  empty ) ∨  (init w ∧ (f;  while  (init w)  do  f))
     using 21 22 3 4 52 61 by fastforce
 have  7: ⊢  while  (init w)  do  f
         = ((¬ ( init w) ∧  empty ) ∨  (init w ∧ (f;  while  (init w)  do  f)))
     using 1 21 22 62
     by (metis 3 41 42 5 51 inteq-reflection)
 have 71: ⊢ if_i  (init w)  then  (f; ( while  (init w)  do  f))  else   empty =
           ((¬ ( init w) ∧  empty ) ∨  (init w ∧ (f;  while  (init w)  do  f)))
     by (auto simp: ifthenelse-d-def)
 from 7 71 show ?thesis by fastforce
qed

lemma WhileChopEqvIf:
⊢  ( while ( init w) do f); g =  if_i(init w) then (f; ((while ( init w) do f); g))  else  g
proof −
 have  1: ⊢  while  (init w)  do  f =
          ifᵢ (init w) then (f; ( while (init w) do f)) else empty
     by (rule WhileEqvIf)
 hence 2: ⊢ ( while (init w) do f); g =
          ifᵢ (init w) then ((f; while (init w) do f); g) else (empty ; g)
     by (rule IfChopEqvRule)
 have 3: ⊢ empty ; g = g
     by (rule EmptyChop)
 have 4: ⊢ ifᵢ (init w) then ((f; while (init w) do f); g) else (empty ; g) =
         ifᵢ (init w) then ((f; while (init w) do f); g) else g
     using 3 using inteq-reflection by fastforce
 have 5: ⊢ ((f; while (init w) do f); g) = (f; (while (init w) do f; g))
     by (rule ChopAssocB)
 have 6: ⊢ ifᵢ (init w) then ((f; while (init w) do f); g) else g =
         ifᵢ (init w) then (f; ((while (init w) do f); g)) else g
     using 5 using inteq-reflection by fastforce
 from 1 2 4 6 show ?thesis by fastforce
qed

lemma WhileChopEqvIfRule:
 assumes ⊢ f = ( while (init w) do g); h
 shows ⊢ f = ifᵢ (init w) then (g; f) else h
proof −
 have 1: ⊢ f = ( while (init w) do g); h
    using assms by auto
 have 2: ⊢ ( while (init w) do g); h =
         ifᵢ (init w) then (g; (( while (init w) do g); h)) else h
    by (rule WhileChopEqvIf)
 have 3: ⊢ (g; f) = (g; (( while (init w) do g); h))
    using 1 by (rule RightChopEqvChop)
 have 4: ⊢ (g; (( while (init w) do g); h)) = (g; f)
    using 3 by auto
 have 5: ⊢ ifᵢ (init w) then (g; (( while (init w) do g); h)) else h =
        ifᵢ (init w) then (g; f) else h
    using 4 using inteq-reflection by fastforce
 from 1 2 5 show ?thesis by fastforce
qed

lemma WhileImpFin:
⊢ while (init w) do f ⟶ fin (¬ ( init w))
proof −
 have 1: ⊢ (init w ∧ f)⋆ ∧ fin (¬ ( init w)) ⟶ fin (¬ ( init w)) by auto
 from 1 show ?thesis by (simp add: while-d-def)
qed

lemma WhileEqvEmptyOrChopWhile:
⊢ while (init w) do f = ((¬ (init w) ∧ empty) ∨ (init w ∧ (f ∧ more );while (init w) do f))
proof −
 have 1: ⊢ (init w ∧ f)⋆ = (empty ∨ ((init w ∧ f)∧ more ); (init w ∧ f)⋆)
     by (rule ChopstarEqv)
 have 2: ⊢ ((init w ∧ f) ∧ more) = (init w ∧ (f ∧ more ))
```

```
      by auto
 hence 3: ⊢ ((init w ∧ f)∧  more ); (init w ∧ f)⋆  = (init w ∧ f ∧  more ); (init w ∧ f)⋆
      by (rule LeftChopEqvChop)
 have  4: ⊢ (init w ∧ f)⋆  =  (empty  ∨  (init w ∧ f ∧  more ); (init w ∧ f)⋆)
      using 1 3 by fastforce
 have  5: ⊢ ((init w ∧ f)⋆ ∧  fin (¬ ( init w)))  =
          (( empty  ∧  fin  (¬ (init w))) ∨
          ((init w ∧ f ∧  more ); (init w ∧ f)⋆∧ fin (¬ ( init w))))
      using 1  4 by fastforce
 have  6: ⊢  (empty  ∧  fin  (¬ ( init w))) = (¬ ( init w) ∧  empty)
      using AndFinEqvChopAndEmpty EmptyChop by (metis int-eq)
 have  7: ⊢ (init w ∧ f ∧  more ); (init w ∧ f)⋆ = (init w ∧ (f ∧  more ); (init w ∧ f)⋆)
      by (rule StateAndChop)
 have  8: ⊢ (((f ∧  more ); (init w ∧ f)⋆) ∧  fin  ( init (¬ w)))  =
         ((f ∧  more ); ((init w ∧ f)⋆ ∧  fin (  init (¬ w))))
      by (rule ChopAndFin)
 have 81: ⊢ fin (  init (¬ w)) = fin (¬ ( init w))
      using FinEqvFin Initprop(2) by fastforce
 have 82: ⊢ ((f ∧  more ); (init w ∧ f)⋆ ∧  fin (¬ ( init w))) =
         ((f ∧  more ); ((init w ∧ f)⋆ ∧  fin (¬ ( init w))))
      using 8 81
      by (metis inteq-reflection)
 have  9: ⊢ ((init w ∧ f)⋆ ∧  fin (¬ ( init w)))  =
         ((¬ ( init w) ∧  empty ) ∨
             (init w ∧ (f ∧  more ); ((init w ∧ f)⋆ ∧  fin (¬ ( init w)))))
      using 5 6 7 82  by fastforce
 from 9 show ?thesis by (simp add: while-d-def)
qed

lemma WhileIntro:
 assumes ⊢   ¬ ( init w) ∧ f ⟶  empty
         ⊢ init w ∧ f ⟶ (g ∧  more ); f
 shows  ⊢ f⟶  while ( init w) do  g
proof −
 have  1: ⊢   ¬ ( init w) ∧ f ⟶  empty
      using assms by blast
 have  2: ⊢ init w ∧ f ⟶ (g ∧  more ); f
      using assms by blast
 have  3: ⊢  while ( init w)  do  g =
         ((¬  (init w) ∧  empty ) ∨  (init w ∧ (g ∧  more ); while  (init w)  do  g))
      by (rule WhileEqvEmptyOrChopWhile)
 hence 31: ⊢ ¬ ( while  (init w)  do  g) =
         (¬( (¬  (init w) ∧  empty ) ∨  (init w ∧ (g ∧  more );  while  (init w)  do  g)))
      by fastforce
 hence 32: ⊢ (f ∧ ¬ ( while  (init w)  do  g)) =
           (f ∧ ¬( (¬(init w) ∧ empty) ∨  (init w ∧ (g ∧  more );  while  (init w)  do  g)))
      by fastforce
 have  33: ⊢ (f ∧ ¬( (¬(init w) ∧ empty) ∨ (init w ∧ (g ∧ more);  while  (init w)  do  g))) =
           (f ∧ ¬(¬(init w) ∧  empty ) ∧ ¬(init w ∧ (g ∧  more );  while  (init w)  do  g))
      by auto
 have 34: ⊢ (f ∧ ¬(¬(init w) ∧ empty) ∧ ¬((init w) ∧ ((g ∧ more ); while (init w) do g))) =
           (f ∧ ( (init w) ∨ more ) ∧ (¬(init w) ∨ ¬((g ∧ more); while (init w) do g)))
      by (auto simp: empty-d-def)
 have 35: ⊢ (f ∧ ((init w) ∨ more) ∧ (¬(init w) ∨ ¬((g ∧ more);while (init w) do g))) =
           ((f ∧ (init w) ∧ ¬ ((g ∧ more ); while ( init w) do g)) ∨
           (f ∧ (init w) ∧ ¬(init w)) ∨
           (f ∧ more ∧ ¬ ((g ∧ more ); while ( init w) do g)) ∨
           (f ∧ more ∧ ¬(init w)))
      by auto
 have 36: ⊢ (f ∧ ¬ ( while (init w) do g)) =
           ((f ∧ (init w) ∧ ¬ ((g ∧ more ); while ( init w) do g)) ∨
           (f ∧ (init w) ∧ ¬(init w)) ∨
           (f ∧ more ∧ ¬ ((g ∧ more ); while ( init w) do g)) ∨
           (f ∧ more ∧ ¬(init w))) using 32 33 34 35 by fastforce
 have 37: ⊢ ¬(f ∧ more ∧ ¬(init w))
      using 1 by (auto simp: empty-d-def)
 have 38: ⊢ (f ∧ more ∧ ¬ ((g ∧ more ); while ( init w) do g)) ⟶
          ((g ∧ more ); f ∧ ¬ ((g ∧ more ); while ( init w) do g))
      using 1 2 by (auto simp: empty-d-def Valid-def)
 have 39: ⊢ (f ∧ (init w) ∧ ¬ ((g ∧ more ); while ( init w) do g)) ⟶
          ((g ∧ more ); f ∧ ¬ ((g ∧ more ); while ( init w) do g))
      using 2 by auto
 have 40: ⊢ ((f ∧ (init w) ∧ ¬ ((g ∧ more ); while ( init w) do g)) ∨
           (f ∧ (init w) ∧ ¬(init w)) ∨
           (f ∧ more ∧ ¬ ((g ∧ more ); while ( init w) do g)) ∨
           (f ∧ more ∧ ¬(init w))) ⟶
           (g ∧ more ); f ∧ ¬ ((g ∧ more ); while ( init w) do g)
       using 39 38 37 38 by fastforce
 have  4: ⊢ f ∧ ¬ ( while (init w) do g) ⟶
           (g ∧ more ); f ∧ ¬ ((g ∧ more ); while ( init w) do g)
       using 36 40 by fastforce
 have  5: ⊢ g ∧ more ⟶ more
       by auto
 from 4 5 show ?thesis using ChopContra by blast
qed

lemma WhileElim:
 assumes ⊢ ¬ ( init w) ∧ empty ⟶ g
         ⊢ init w ∧ (f ∧ more ); g ⟶ g
 shows ⊢ while (init w) do f ⟶ g
proof −
 have  1: ⊢ while ( init w) do f =
          ((¬ ( init w) ∧ empty ) ∨ (init w ∧ (f ∧ more ); while ( init w) do f))
       by (rule WhileEqvEmptyOrChopWhile)
 hence 11: ⊢ ((while ( init w) do f) ∧ ¬ g) =
          (((¬(init w) ∧ empty) ∨ (init w ∧ (f ∧ more);while (init w) do f)) ∧ ¬ g)
      by auto
 have  2: ⊢ ¬ ( init w) ∧ empty ⟶ g
       using assms by blast
 hence 21: ⊢ ¬ g ⟶ ¬(¬ ( init w) ∧ empty)
```

```
       by auto
 have  22: ⊢  ((¬ (init w) ∧ empty) ∨ (init w ∧ (f ∧ more);while (init w) do f)) ∧ ¬  g ⟶
            (init w ∧ (f ∧  more );  while ( init w)  do  f)
       using 21 by auto
 have  23: ⊢ (while ( init w)  do  f) ∧ ¬ g ⟶
            (init w ∧ (f ∧  more );  while ( init w)  do  f) ∧ ¬  g
       using 11 21 by fastforce
 have   3: ⊢ (init w) ∧ ((f ∧  more ); g) ⟶ g
       using assms by blast
 hence 31: ⊢ ¬ g ⟶ ¬((init w) ∧ ((f ∧  more ); g))
       by fastforce
 have  32: ⊢ (init w ∧ (f ∧  more );  while ( init w)  do  f) ∧ ¬ g ⟶
           (((f ∧  more );  (while  (init w)  do  f)) ∧ ¬ ((f ∧  more ); g)) ∧ ¬g
       using 31 by auto
 have   4: ⊢  (while  (init w)  do  f) ∧ ¬  g ⟶
            ((f ∧  more );  (while  (init w)  do  f)) ∧ ¬ ((f ∧  more ); g)
       using 23 32 by fastforce
 have   5: ⊢ f ∧  more  ⟶  more
       by auto
 from 4 5 show ?thesis using ChopContra by blast
qed

lemma BaWhileImpWhile:
⊢   ba (f ⟶ g) ⟶ ( while  (init w)  do  f) ⟶ ( while  (init w)  do  g)
proof −
 have  1: ⊢ (f ⟶ g) ⟶ ((init w ∧ f)  ⟶ (init w ∧ g))
      by auto
 hence 2: ⊢  ba (f ⟶ g) ⟶  ba ((init w ∧ f)  ⟶ (init w∧ g))
      by (rule BaImpBa)
 have  3:  ⊢  ba ((init w ∧ f)  ⟶ (init w∧ g)) ⟶ ((init w∧ f)⋆ ⟶ (init w∧ g)⋆)
      by (rule BaCSImpCS)
 have  4: ⊢  ba (f ⟶ g) ⟶ ((init w∧ f)⋆∧ fin (¬ ( init w)) ⟶ (init w∧ g)⋆∧ fin (¬ (init w)))
      using 2 3 by fastforce
 from 4 show ?thesis by (simp add: while-d-def)
qed

lemma WhileImpWhile:
 assumes ⊢   f⟶ g
 shows ⊢ ( while  (init w)  do  f) ⟶ ( while  (init w)  do  g)
proof −
 have  1: ⊢ f ⟶ g
      using assms by auto
 hence 2: ⊢  ba (f ⟶ g)
      by (rule BaGen)
 have  3: ⊢  ba (f ⟶ g) ⟶ ( while  (init w)  do  f) ⟶ ( while  (init w)  do  g)
      by (rule BaWhileImpWhile)
 from 2 3 show ?thesis using MP by blast
qed
```

4.9 Properties of Halt

```
lemma WnextAndMoreEqvNext:
⊢ (wnext f ∧ more) = ○ f
by (auto simp: wnext-defs more-defs next-defs)

lemma BoxStateAndEmptyEqvStateAndEmpty:
⊢ (□(empty = (init w)) ∧ empty) = ((init w) ∧ empty)
by (auto simp: always-defs init-defs empty-defs)

lemma BoxEmptyEqvIStateqvEmptyAndStateOrNotStateNext:
⊢ □(empty = (init w)) = ((empty ∧ init w) ∨ (¬(init w) ∧ ○(□(empty = (init w)))))
proof −
 have 1: ⊢ □(empty = (init w)) =
          ((□(empty = (init w)) ∧ empty) ∨ (□(empty = (init w)) ∧ more))
      by (auto simp: empty-d-def)
 have 2: ⊢ (□(empty = (init w)) ∧ empty) = ((init w) ∧ empty)
      using BoxStateAndEmptyEqvStateAndEmpty by blast
 have 3: ⊢ □(empty = (init w))  = ((empty = (init w)) ∧ wnext(□(empty = (init w))))
      using BoxEqvAndWnextBox by blast
 hence 4: ⊢ (□(empty = (init w)) ∧ more)  =
            (((empty = (init w)) ∧ more) ∧ (wnext(□(empty = (init w))) ∧ more))
      by auto
 have 5: ⊢ ((empty = (init w)) ∧ more) = (¬(init w) ∧ more)
      by (auto simp: empty-d-def)
 have 6: ⊢ (wnext(□(empty = (init w))) ∧ more) = ○(□(empty = (init w)))
      using WnextAndMoreEqvNext by metis
 have 7: ⊢ (□(empty = (init w)) ∧ more) =
          ((¬ (init w) ∧ more) ∧ (wnext(□(empty = (init w))) ∧ more) )
      using 4 5 by fastforce
 have 8: ⊢ ((¬ (init w) ∧ more) ∧ (wnext(□(empty = (init w))) ∧ more) ) =
          ((¬ (init w)) ∧ (wnext(□(empty = (init w))) ∧ more) ) by auto
 have 9: ⊢ ((¬ (init w)) ∧ (wnext(□(empty = (init w))) ∧ more) ) =
           ((¬ (init w)) ∧ ○(□(empty = (init w))))  using 8 6 by auto
 have 10: ⊢ □(empty = (init w)) = (((init w) ∧ empty) ∨ (□(empty = (init w)) ∧ more) )
      using 1 2 by fastforce
 from 7 9 10 show ?thesis by fastforce
qed

lemma HaltStateEqvIfStateThenEmptyElseNext:
⊢  halt(  init w) =  if_i  (init w)  then   empty   else ( ○( halt ( init w)))
proof −
 have 1: ⊢ halt(  init w) = □(empty = (init w))
     by (simp add: halt-d-def)
 have 2: ⊢ □(empty = (init w)) =
          ((empty ∧ init w) ∨ (¬(init w) ∧ ○(□(empty = (init w)))))
     by (rule BoxEmptyEqvIStateqvEmptyAndStateOrNotStateNext)
 have 21: ⊢ ((empty ∧ init w) ∨ (¬(init w) ∧ ○(□(empty = (init w))))) =
          ((init w ∧ empty) ∨ (¬(init w) ∧ ○(□(empty = (init w)))))
     by auto
 have 22: ⊢ ○(halt ( init w)) = ○(□(empty = (init w)))
```

```
     using NextEqvNext using 1 by blast
 have 3: ⊢ ifᵢ (init w) then empty else ( ○( halt ( init w))) =
          ((init w ∧ empty ) ∨ (¬(init w) ∧ ○( halt ( init w))))
     by (simp add: ifthenelse-d-def)
 from 1 2 21 22 3 show ?thesis by fastforce
qed

lemma HaltChopEqv:
⊢ ((halt ( init w)) ; f) = (ifᵢ (init w) then ( f ) else (○( (halt ( init w)); f)))
proof −
 have 1: ⊢ halt(init w) =
           (ifᵢ (init w) then empty else ( ○( halt ( init w))))
      by (rule HaltStateEqvIfStateThenEmptyElseNext)
 hence 2: ⊢ ((halt(init w));f) =
           (ifᵢ (init w) then (empty;f) else ( ○( halt ( init w));f))
      by (rule IfChopEqvRule)
 have 3: ⊢ empty ; f = f
      by (rule EmptyChop)
 have 4: ⊢ (○ (halt ( init w))); f = ○( halt ( init w); f)
      by (rule NextChop)
 from 2 3 4 show ?thesis by (metis inteq-reflection)
qed

lemma AndHaltChopImp:
⊢ init w ∧ ( halt ( init w); f) ⟶ f
proof −
 have 1: ⊢ halt ( init w); f = ifᵢ (init w) then f else ( ○( halt ( init w); f))
     by (rule HaltChopEqv)
 have 2: ⊢ init w ∧ ifᵢ (init w) then f else ( ○( halt ( init w); f)) ⟶ f
     by (auto simp: ifthenelse-d-def)
 from 1 2 show ?thesis by fastforce
qed

lemma NotAndHaltChopImpNext:
⊢ ¬ ( init w) ∧ ( halt (init w); f) ⟶ ○( halt ( init w); f)
proof −
 have 1: ⊢ halt ( init w); f = ifᵢ (init w) then f else ( ○( halt ( init w); f))
     by (rule HaltChopEqv)
 have 2: ⊢ ¬ ( init w) ∧ ifᵢ (init w) then f else ( ○( halt ( init w); f)) ⟶
          ○( halt ( init w); f)
     by (auto simp: ifthenelse-d-def)
from 1 2 show ?thesis by fastforce
qed

lemma NotAndHaltChopImpSkipYields:
⊢ ¬ ( init w) ∧ ( halt ( init w); f) ⟶ skip yields ( halt (init w); f)
proof −
 have 1: ⊢ ¬ ( init w) ∧ ( halt ( init w); f) ⟶ ○( halt ( init w); f)
     by (rule NotAndHaltChopImpNext)
 have 2: ⊢ ○( halt ( init w); f) ⟶ skip yields ( halt ( init w); f)
```

```
    by (rule NextImpSkipYields)
 from 1 2 show ?thesis by fastforce
qed

lemma TrueChopAndEmptyEqvChopAndEmpty:
 ⊢ ((#True;(f ∧ empty)) ∧ g) = (g;(f∧ empty))
using AndFinEqvChopAndEmpty FinEqvTrueChopAndEmpty by (metis int-eq lift-and-com)

lemma WprevEqvEmptyOrPrev:
⊢ wprev f = (empty ∨ prev f)
by  (auto simp: wprev-defs empty-defs prev-defs)


lemma NotChopSkipEqvMoreAndNotChopSkip:
 ⊢ (¬ f);skip = (more ∧ ¬(f;skip))
proof −
 have 1: ⊢ wprev f = (empty ∨ prev f) using WprevEqvEmptyOrPrev by auto
 hence 2: ⊢ (¬(wprev f)) = (¬(empty ∨ prev f))  by auto
 have 3: ⊢ ¬(wprev f) = ((¬ f);skip)  by (simp add: wprev-d-def prev-d-def)
 have 31: ⊢ (empty ∨ prev f) = (empty ∨ (f;skip)) by (simp add: prev-d-def)
 have 32: ⊢ (empty ∨ (f;skip)) = (¬more ∨ ¬¬(f;skip)) by (simp add: empty-d-def)
 have 33: ⊢ (¬more ∨ ¬¬(f;skip)) = (¬(more ∧ ¬(f;skip))) by fastforce
 have 34: ⊢ (empty ∨ prev f) = (¬(more ∧ ¬(f;skip))) using 31 32 33 by (metis int-eq)
 have 4: ⊢ ¬(empty ∨ prev f) = (more ∧ ¬(f;skip)) using 34 by fastforce
 from 2 3 4 show ?thesis by fastforce
qed


lemma HaltChopImpNotHaltChopNot:
⊢    halt ( init w); f ⟶ ¬ ( halt ( init w); (¬  f))
proof −
 have   1: ⊢  halt  (init w); f =  if_i  (init w)  then  f  else ( ○( halt ( init w); f))
        by (rule HaltChopEqv)
 have   2: ⊢ if_i  (init w)  then  f  else ( ○( halt ( init w); f)) ⟶
            ( ((init w) ⟶ f) ∧ ( ¬(init w) ⟶ ( ○( halt ( init w); f))))
        by (rule IfThenElseImp)
 have   3: ⊢  halt  (init w); (¬f) =
             if_i  (init w)  then  (¬f)  else ( ○( halt ( init w); (¬f)))
        by (rule HaltChopEqv)
 have   4: ⊢ if_i  (init w)  then  (¬f)  else ( ○( halt ( init w); (¬f))) ⟶
            ( ((init w) ⟶ ¬f) ∧ ( ¬(init w) ⟶ ( ○( halt ( init w); (¬f)))))
        by (rule IfThenElseImp)
 have   5: ⊢ halt  (init w); f ∧ halt  (init w); (¬f) ⟶
          ( ((init w) ⟶ f) ∧ ( ¬(init w) ⟶ ( ○( halt ( init w); f)))) ∧
          ( ((init w) ⟶ ¬f) ∧ ( ¬(init w) ⟶ ( ○( halt ( init w); (¬f)))))
        using 1 2 3 4 by fastforce
 have   6: ⊢ ( ((init w) ⟶ f) ∧ ( ¬(init w) ⟶ ( ○( halt ( init w); f)))) ∧
          ( ((init w) ⟶ ¬f) ∧ ( ¬(init w) ⟶ ( ○( halt ( init w); (¬f))))) ⟶
          ( ○( halt ( init w); f)) ∧ ( ○( halt ( init w); (¬f)))
        by auto
 have  7: ⊢ halt (init w); f ∧ halt (init w); (¬f) ⟶
          ( ○( halt ( init w); f)) ∧ ( ○( halt ( init w); (¬f)))
       using 5 6 lift-imp-trans by blast
 have  8: ⊢ (( ○( halt ( init w); f)) ∧ ( ○( halt ( init w); (¬f)))) =
           ○ (halt ( init w); f ∧ halt ( init w); (¬f))
       using NextAndEqvNextAndNext by fastforce
 have  9: ⊢ halt (init w); f ∧ halt (init w); (¬f) ⟶
            ○ (halt ( init w); f ∧ halt ( init w); (¬f))
       using 7 8 by fastforce
 hence 10: ⊢ ¬(halt (init w); f ∧ halt (init w); (¬f))
       using NextLoop by blast
 from 10 show ?thesis by auto
qed

lemma HaltChopImpHaltYields:
⊢  halt ( init w); f ⟶ ( halt ( init w)) yields f
proof –
 have 1: ⊢ halt ( init w); f ⟶ ¬ ( halt ( init w); (¬ f)) by (rule HaltChopImpNotHaltChopNot)
 from 1 show ?thesis by (simp add: yields-d-def)
qed

lemma HaltChopAnd:
⊢ ( halt (init w)); f ∧ ( halt (init w)); g ⟶ ( halt ( init w)); (f ∧ g)
proof –
 have  1: ⊢ ( halt (init w)); g ⟶ ( halt (init w)) yields g by (rule HaltChopImpHaltYields)
 hence 2: ⊢ ( halt (init w)); f ∧ ( halt (init w)); g ⟶
           ( halt (init w)); f ∧ ( halt (init w)) yields g by auto
 have  3: ⊢ ( halt (init w)); f ∧ ( halt (init w)) yields g⟶
           ( halt (init w)); (f ∧ g) by (rule ChopAndYieldsImp)
 from 2 3 show ?thesis by fastforce
qed

lemma HaltAndChopAndHaltChopImpHaltAndChopAnd:
⊢  ( halt (init w) ∧ f); f1 ∧ ( halt (init w); g) ⟶ ( halt ( init w) ∧ f); (f1 ∧ g)
proof –
 have  1: ⊢ f1 ⟶ ¬ g ∨ (f1 ∧ g)
       by auto
 hence 2: ⊢ ( halt (init w) ∧ f); f1 ⟶
           ( halt (init w) ∧ f); (¬ g) ∨ (( halt (init w) ∧ f); (f1 ∧ g))
       by (rule ChopOrImpRule)
 have  3: ⊢ ( halt (init w) ∧ f); (¬ g) ⟶ halt (init w); (¬ g)
       by (rule AndChopA)
 have  31: ⊢ ( halt (init w) ∧ f); f1 ⟶
           halt (init w); (¬ g) ∨ (( halt (init w) ∧ f); (f1 ∧ g))
       using 23 by fastforce
 have  4: ⊢ halt (init w); g ⟶ ¬ ( halt (init w); (¬ g))
       by (rule HaltChopImpNotHaltChopNot)
 hence 41: ⊢ ( halt (init w); (¬ g)) ⟶ ¬(halt (init w); g)
       by auto
 have  42: ⊢ ( halt (init w) ∧ f); f1 ⟶
```

```
          ¬( halt (init w); g) ∨ (( halt (init w) ∧ f); (f1 ∧ g))
       using 31 41 by fastforce
 from 42 show ?thesis by auto
qed

lemma HaltImpBoxYields:
⊢ ( halt (init w)); f ⟶ (□(¬ ( init w))) yields (( halt (init w)); f)
proof −
 have  1: ⊢ (□ (¬ (init w))); (¬ ( halt (init w); f)) ⟶ di (□ (¬ (init w)))
       by (rule ChopImpDi)
 have  2: ⊢ □ (¬ (init w)) ⟶ ¬ (init w)
       by (rule BoxElim)
 hence 3: ⊢ di (□ (¬ (init w))) ⟶ di (¬ (init w))
       by (rule DiImpDi)
 have  4: ⊢ di (init (¬ w)) = (init (¬w))
       by (rule DiState)
 have  41: ⊢ (init (¬ w)) = (¬ (init w))
       using Initprop(2) by fastforce
 have  42: ⊢ di (¬ (init w)) = (¬(init w))
       using 4 41 by (metis inteq-reflection)
 have  5: ⊢ ((□(¬ (init w))); (¬ ( halt (init w); f))) ⟶ ¬ ( init w)
       using 1 2 42 using 3 by fastforce
 hence 51: ⊢ ( halt (init w); f) ∧ ((□(¬ (init w))); (¬ ( halt (init w); f))) ⟶
          ( halt (init w); f) ∧ ¬ ( init w)
       by fastforce
 have  6: ⊢ halt (init w); f = if_i (init w) then f else (○( halt (init w); f))
       by (rule HaltChopEqv)
 hence 61: ⊢ (halt (init w); f ∧ ¬ ( init w)) =
            ((if_i (init w) then f else (○( halt (init w); f))) ∧ ¬ ( init w))
       using 6 by auto
 have  62: ⊢ (if_i (init w) then f else (○( halt (init w); f))) ∧
            ¬ ( init w) ⟶ (○( halt (init w); f))
       by (auto simp: ifthenelse-d-def)
 have  63: ⊢ halt (init w); f ∧ ¬ ( init w) ⟶ (○( halt (init w); f))
       using 61 62 by fastforce
 have  7: ⊢ ( halt (init w); f) ∧ (□(¬ (init w))); (¬ ( halt (init w); f)) ⟶
           ○(( halt (init w)); f)
       using 51 63 using lift-imp-trans by blast
 have  8: ⊢ □ (¬ (init w)) ⟶ empty ∨ ○(□(¬( init w)))
       using BoxBoxImpBox BoxEqvAndEmptyOrNextBox by fastforce
 hence 9: ⊢ ((□ (¬ ( init w))); (¬ ( halt (init w); f))) ⟶
          ¬ ( halt (init w); f) ∨ ○((□(¬ (init w))); (¬ ( halt (init w); f)))
       by (rule EmptyOrNextChopImpRule)
 hence 10: ⊢ (( halt (init w)); f) ∧ (□ (¬ (init w))); (¬ ( halt (init w); f)) ⟶
          ○((□(¬ (init w))); (¬ ( halt (init w); f)))
       by fastforce
 have  11: ⊢ ( halt (init w)); f ∧ (□ (¬ (init w))); (¬ ( halt (init w); f)) ⟶
          ○(( halt (init w)); f) ∧ ○((□(¬ (init w))); (¬ ( halt (init w); f)))
       using 7 10 by fastforce
 have  12: ⊢ ○(( halt (init w)); f) ∧ ○((□(¬ (init w))); (¬ ( halt (init w); f)))
```

```
        ⟶ ○((( halt (init w)); f) ∧ ((□(¬ (init w))); (¬ ( halt (init w); f))))
      using NextAndEqvNextAndNext by fastforce
  have 13: ⊢ ( halt (init w)); f ∧ (□ (¬ (init w))); (¬ ( halt (init w); f)) ⟶
          ○((( halt (init w)); f) ∧ ((□(¬ (init w))); (¬ ( halt (init w); f))))
      using 11 12 by fastforce
  hence 14: ⊢ ¬ (( halt (init w)); f ∧ (□ (¬ (init w))); (¬ ( halt (init w); f)))
      using NextLoop by blast
  hence 15: ⊢ ( halt (init w)); f ⟶ ¬ ((□ (¬ (init w))); (¬ ( halt (init w); f)))
      by auto
  from 15 show ?thesis by (simp add: yields-d-def)
qed
```

4.10 Properties of Groups of chops

```
lemma NestedChopImpChop:
  assumes ⊢  init w ∧ f ⟶ g; (init w1 ∧ f1)
          ⊢ init w1 ∧ f1 ⟶ g1; (init w2 ∧ f2)
  shows  ⊢ init w ∧ f ⟶ g; (g1; (init w2 ∧ f2))
proof –
  have  1: ⊢ init w ∧ f ⟶ g; (init w1 ∧ f1)  using assms(1) by auto
  have  2: ⊢ init w1 ∧ f1 ⟶ g1; (init w2 ∧ f2)  using assms(2) by auto
  hence 3: ⊢ g; (init w1 ∧ f1) ⟶ g; (g1; (init w2 ∧ f2))  by (rule RightChopImpChop)
  from 1 3 show ?thesis by fastforce
qed
```

4.11 Properties of Time Reversal

```
lemma RNot:
 ⊢ (¬f)^r = (¬ f^r)
by (simp add: rev-fun1)

lemma RRNot:
 ⊢ (¬(f^r))^r = (¬f)
by (metis EqvReverseReverse int-eq rev-fun1)

lemma RTrue:
 ⊢ (#True)^r = #True
using rev-const by fastforce

lemma ROr:
 ⊢ (f ∨ g)^r = (f^r ∨ g^r)
by (simp add: rev-fun2)

lemma RROr:
 ⊢ (f^r ∨ g^r)^r = (f ∨ g)
proof –
  have 1: ⊢ (f^r ∨ g^r)^r = ((f^r)^r ∨ (g^r)^r) using ROr by blast
  have 2: ⊢ ((f^r)^r ∨ (g^r)^r) = (f ∨ g) using EqvReverseReverse by (metis inteq-reflection)
  from 1 2 show ?thesis by fastforce
qed
```

```
lemma RAnd:
 ⊢ (f ∧ g)^r = (f^r ∧ g^r)
by (simp add: rev-fun2)

lemma RRAnd:
 ⊢ (f^r ∧ g^r)^r = (f ∧ g)
proof −
 have 1: ⊢ (f^r ∧ g^r)^r = ((f^r)^r ∧ (g^r)^r) using RAnd by blast
 have 2: ⊢ ((f^r)^r ∧ (g^r)^r) = (f ∧ g) using EqvReverseReverse by (metis inteq-reflection)
 from 1 2 show ?thesis by fastforce
qed

lemma RImpRule:
 assumes ⊢ f ⟶ g
 shows  ⊢ f^r ⟶ g^r
using assms by (simp add: Valid-def reverse-d-def)

sledgehammer-params [minimize=true,preplay-timeout=10,timeout=60,verbose=true,
                    provers=z3 vampire cvc4 e spass ]

lemma RAndEmptyEqvAndEmpty:
⊢ (f ∧ empty)^r = (f ∧ empty)
apply (simp add: Valid-def empty-defs reverse-d-def)
by (metis interval-st-intlen intrev.simps(1))

lemma RNextEqvPrev:
⊢ (○ f)^r = prev (f^r)
by (metis RevChop RevSkip inteq-reflection next-d-def prev-d-def)

lemma RRNextEqvPrev:
⊢ (○ (f^r))^r = prev (f)
proof −
 have 1: ⊢ (○ (f^r))^r = prev ((f^r)^r) using RNextEqvPrev by blast
 have 2: ⊢ prev ((f^r)^r) = prev f using EqvReverseReverse  by (metis inteq-reflection)
 from 1 2 show ?thesis by fastforce
qed

lemma RWNextEqvWPrev:
⊢ (wnext f)^r = wprev(f^r)
by (smt  RNextEqvPrev REmptyEqvEmpty WnextEqvEmptyOrNext WprevEqvEmptyOrPrev int-eq rev-fun2)

lemma RRWNextEqvWPrev:
⊢ (wnext (f^r))^r = wprev(f)
proof −
 have 1: ⊢ (wnext (f^r))^r = wprev ((f^r)^r) using RWNextEqvWPrev by blast
 have 2: ⊢ wprev ((f^r)^r) = wprev f using EqvReverseReverse by (metis inteq-reflection)
 from 1 2 show ?thesis by fastforce
qed
```

lemma *RPrevEqvNext*:
$\vdash (\mathit{prev}\ f)^r = \bigcirc\ (f^r)$
by (*metis RevChop RevSkip inteq-reflection next-d-def prev-d-def*)

lemma *RRPrevEqvNext*:
$\vdash (\mathit{prev}\ (f^r))^r = \bigcirc\ (f)$
proof −
have *1*: $\vdash (\mathit{prev}\ (f^r))^r = \bigcirc\ ((f^r)^r)$ **using** *RPrevEqvNext* **by** *blast*
have *2*: $\vdash \bigcirc\ ((f^r)^r) = \bigcirc\ f$ **using** *EqvReverseReverse* **by** (*metis inteq-reflection*)
from *1 2* **show** *?thesis* **by** *fastforce*
qed

lemma *RWPrevEqvWNext*:
$\vdash (\mathit{wprev}\ f)^r = \mathit{wnext}(f^r)$
by (*metis EqvReverseReverse RRWNextEqvWPrev int-eq*)

lemma *RRWPrevEqvWNext*:
$\vdash (\mathit{wprev}\ (f^r))^r = \mathit{wnext}(f)$
proof −
have *1*: $\vdash (\mathit{wprev}\ (f^r))^r = \mathit{wnext}\ ((f^r)^r)$ **using** *RWPrevEqvWNext* **by** *blast*
have *2*: $\vdash \mathit{wnext}\ ((f^r)^r) = \mathit{wnext}\ f$ **using** *EqvReverseReverse* **by** (*metis inteq-reflection*)
from *1 2* **show** *?thesis* **by** *fastforce*
qed

lemma *RDiamondEqvDi*:
$\vdash (\Diamond f)^r = \mathit{di}\ (f^r)$
by (*simp add*: *di-d-def sometimes-d-def*, *metis RevChop RTrue inteq-reflection*)

lemma *RRDiamondEqvDi*:
$\vdash (\Diamond(f^r))^r = \mathit{di}\ (f)$
proof −
have *1*: $\vdash (\Diamond\ (f^r))^r = \mathit{di}\ ((f^r)^r)$ **using** *RDiamondEqvDi* **by** *blast*
have *2*: $\vdash \mathit{di}\ ((f^r)^r) = \mathit{di}\ f$ **using** *EqvReverseReverse* **by** (*metis inteq-reflection*)
from *1 2* **show** *?thesis* **by** *fastforce*
qed

lemma *RBoxEqvBi*:
$\vdash (\Box\ f)^r = \mathit{bi}\ (f^r)$
by (*simp add*: *always-d-def bi-d-def*, *metis RDiamondEqvDi int-eq rev-fun1*)

lemma *RRBoxEqvBi*:
$\vdash (\Box\ (f^r))^r = \mathit{bi}\ (f)$
proof −
have *1*: $\vdash (\Box\ (f^r))^r = \mathit{bi}\ ((f^r)^r)$ **using** *RBoxEqvBi* **by** *blast*
have *2*: $\vdash \mathit{bi}\ ((f^r)^r) = \mathit{bi}\ f$ **using** *EqvReverseReverse* **by** (*metis inteq-reflection*)
from *1 2* **show** *?thesis* **by** *fastforce*
qed

lemma *RDiEqvDiamond*:
$\vdash (di\ f)^r = \Diamond\ (f^r)$
by (*simp add*: *di-d-def sometimes-d-def*, *metis RevChop RTrue inteq-reflection*)

lemma *RRDiEqvDiamond*:
$\vdash (di\ (f^r))^r = \Diamond\ (f)$
proof −
have *1*: $\vdash (di\ (f^r))^r = \Diamond\ ((f^r)^r)$ **using** *RDiEqvDiamond* **by** *blast*
have *2*: $\vdash \Diamond\ ((f^r)^r) = \Diamond\ f$ **using** *EqvReverseReverse* **by** (*metis inteq-reflection*)
from *1 2* **show** *?thesis* **by** *fastforce*
qed

lemma *RBiEqvBox*:
$\vdash (bi\ f)^r = \Box\ (f^r)$
by (*simp add*: *always-d-def bi-d-def*, *metis RDiEqvDiamond rev-fun1 int-eq*)

lemma *RRBiEqvBox*:
$\vdash (bi\ (f^r))^r = \Box\ (f)$
proof −
have *1*: $\vdash (bi\ (f^r))^r = \Box\ ((f^r)^r)$ **using** *RBiEqvBox* **by** *blast*
have *2*: $\vdash \Box\ ((f^r)^r) = \Box\ f$ **using** *EqvReverseReverse* **by** (*metis inteq-reflection*)
from *1 2* **show** *?thesis* **by** *fastforce*
qed

lemma *RDaEqvDa*:
$\vdash (da\ f)^r = da(f^r)$
proof −
have *1*: $\vdash (\#True;(f;\#True))^r = (f;\#True)^r;\ \#True^r$ **using** *RevChop* **by** *blast*
have *2*: $\vdash (f;\#True)^r;\ \#True^r = \ (f;\#True)^r;\ \#True$ **using** *RTrue RightChopEqvChop* **by** *blast*
have *3*: $\vdash (f;\#True)^r;\ \#True = (\#True^r;f^r);\#True$ **by** (*simp add*: *RevChop LeftChopEqvChop*)
have *4*: $\vdash (\#True^r;f^r);\#True = (\#True;f^r);\#True$ **by** (*metis 3 RTrue int-eq*)
have *5*: $\vdash (\#True;f^r);\#True = \#True;(f^r;\#True)$ **using** *ChopAssocB* **by** *blast*
have *6*: $\vdash (\#True;(f;\#True))^r = \#True;(f^r;\#True)$ **using** *1 2 3 4 5* **by** *fastforce*
from *6* **show** *?thesis* **by** (*simp add*: *da-d-def*)
qed

lemma *RRDaEqvDa*:
$\vdash (da\ (f^r))^r = da(f)$
proof −
have *1*: $\vdash (da\ (f^r))^r = da\ ((f^r)^r)$ **using** *RDaEqvDa* **by** *blast*
have *2*: $\vdash da\ ((f^r)^r) = da\ f$ **using** *EqvReverseReverse* **by** (*metis inteq-reflection*)
from *1 2* **show** *?thesis* **by** *fastforce*
qed

lemma *RBaEqvBa*:
$\vdash (ba\ f)^r = ba(f^r)$
by (*simp add*: *ba-d-def*, *metis RDaEqvDa int-eq rev-fun1*)

lemma *RRBaEqvBa*:

$\vdash (ba\ (f^r))^r = ba(f)$
proof –
 have *1*: $\vdash (ba\ (f^r))^r = ba\ ((f^r)^r)$ **using** *RBaEqvBa* **by** *blast*
 have *2*: $\vdash ba\ ((f^r)^r) = ba\ f$ **using** *EqvReverseReverse* **by** (*metis inteq-reflection*)
 from *1 2* **show** *?thesis* **by** *fastforce*
qed

lemma *ChopCsImpCSChop*:
$\vdash f;f^\star \longrightarrow f^\star;f$
by (*meson CSChopEqvChopOrRule CSChopEqvOrChopPlusChop ChopAssocB ChopPlusElimWithoutMore EmptyYields Prop03 Prop04 Prop06*)

lemma *CSChopImpChopCS*:
$\vdash f^\star;f \longrightarrow f;f^\star$
proof –
 have *1*: $\vdash (f^r);(f^r)^\star \longrightarrow (f^r)^\star;(f^r)$
  using *ChopCsImpCSChop* **by** *blast*
 hence *2*: $\vdash ((f^r);(f^r)^\star \longrightarrow (f^r)^\star;(f^r)\ )^r$
  using *ReverseEqv* **by** *blast*
 have *3*: $\vdash (((f^r);(f^r)^\star \longrightarrow (f^r)^\star;(f^r)\ )^r) = (\ ((f^r);(f^r)^\star)^r \longrightarrow ((f^r)^\star;(f^r))^r)$
  by (*smt 1 2 RImpRule Valid-def unl-lift2*)
 have *4*: $\vdash ((f^r);(f^r)^\star)^r = ((f^r)^\star\ )^r;\ (f^r)^r$
  by (*simp add*: *RevChop*)
 have *5*: $\vdash ((f^r)^\star\ )^r;\ (f^r)^r = ((f^r)^r)^\star;(f^r)^r$
  by (*simp add*: *LeftChopEqvChop RevChopstar*)
 have *6*: $\vdash (f^r)^r = f$
  using *EqvReverseReverse* **by** *blast*
 have *7*: $\vdash ((f^r)^r)^\star;(f^r)^r = f^\star;f$
  using *6 CSEqvCS ChopEqvChop* **by** *blast*
 have *8*: $\vdash ((f^r);(f^r)^\star)^r = f^\star;f$
  using *7 5* **using** *4* **by** *fastforce*
 have *9*: $\vdash ((f^r)^\star;(f^r))^r = (f^r)^r;((f^r)^\star)^r$
  by (*simp add*: *RevChop*)
 have *10*: $\vdash (f^r)^r;((f^r)^\star)^r = (f^r)^r;\ ((f^r)^r)^\star$
  by (*simp add*: *RevChopstar RightChopEqvChop*)
 have *11*: $\vdash (f^r)^r;\ ((f^r)^r)^\star = f;f^\star$
  using *6 ChopPlusEqvChopPlus* **by** *blast*
 have *12*: $\vdash ((f^r);(f^r)^\star)^r = f;f^\star$
  using *9 10 11* **by** (*metis 4 5 ChopCsImpCSChop RImpRule int-eq int-iffI*)
 from *2 3 8 12* **show** *?thesis* **by** *fastforce*
qed

lemma *CSChopEqvChopCS*:
$\vdash f;f^\star = f^\star;f$
using *ChopCsImpCSChop CSChopImpChopCS* **by** *fastforce*

lemma *TrueChopSkipEqvSkipChopTrue*:
$\vdash\ \#True;skip = skip;\#True$
proof –
 have *1*: $\vdash skip;skip^\star = skip^\star;skip$ **using** *CSChopEqvChopCS* **by** *blast*

have *2*: $\vdash$ *skip** = #*True* **using** *CSSkip* **by** *simp*
have *3*: $\vdash$ *skip*;*skip** = *skip*;#*True* **using** *2* **using** *RightChopEqvChop* **by** *blast*
have *4*: $\vdash$ *skip**;*skip* = #*True*;*skip* **using** *2* **using** *LeftChopEqvChop* **by** *blast*
from *1 3 4* **show** *?thesis* **by** *fastforce*
qed

lemma *RInitEqvFin*:
$\vdash$ (*init f*)^r = *fin*(*f*)
proof –
have *1*: $\vdash$ (*init f*)^r = ((*f* $\wedge$ *empty*);#*True*)^r
by (*metis AndChopCommute REqvRule init-d-def*)
have *2*: $\vdash$ ((*f* $\wedge$ *empty*);#*True*)^r = (#*True*;(*f* $\wedge$ *empty*)^r)
using *RTrue* **by** (*metis RevChop int-eq*)
have *3*: $\vdash$ #*True*;(*f* $\wedge$ *empty*)^r = #*True*;(*f*^{r} $\wedge$ *empty*)
by (*metis RAnd REmptyEqvEmpty RightChopEqvChop int-eq*)
have *4*: $\vdash$ #*True*;(*f*^{r} $\wedge$ *empty*) = #*True*;(*f* $\wedge$ *empty*)
using *RAndEmptyEqvAndEmpty*
by (*metis REmptyEqvEmpty RightChopEqvChop all-rev-eq*(*3*) *int-eq*)
have *5*: $\vdash$ #*True*;(*f* $\wedge$ *empty*) = *fin*(*f*)
using *FinEqvTrueChopAndEmpty* **by** *fastforce*
from *1 2 3 4 5* **show** *?thesis* **by** *fastforce*
qed

lemma *RFinEqvInit*:
$\vdash$ (*fin f*)^r = *init* (*f*)
proof –
have *1*: $\vdash$ *fin f* = #*True*;(*f* $\wedge$ *empty*)
using *FinEqvTrueChopAndEmpty* **by** *auto*
have *2*: $\vdash$ (*fin f*)^r = (#*True*;(*f* $\wedge$ *empty*))^r
using *1 REqvRule* **by** *blast*
have *3*: $\vdash$ (#*True*;(*f* $\wedge$ *empty*))^r = (*f* $\wedge$ *empty*)^r;#*True*
using *RTrue* **by** (*metis RevChop int-eq*)
have *4*: $\vdash$ (*f*$\wedge$ *empty*)^r;#*True* = (*f*^{r} $\wedge$ *empty*);#*True*
using *LeftChopEqvChop RAnd REmptyEqvEmpty* **by** (*metis int-eq*)
have *5*: $\vdash$ (*f*$\wedge$ *empty*)^r;#*True* = (*f*$\wedge$ *empty*);#*True*
by (*simp add*: *RAndEmptyEqvAndEmpty LeftChopEqvChop*)
have *6*: $\vdash$ (*f* $\wedge$ *empty*);#*True* = *init*(*f*)
by (*simp add*: *AndChopCommute init-d-def*)
from *1 2 3 4 5 6* **show** *?thesis* **by** *fastforce*
qed

lemma *RHaltEqvInitonly*:
$\vdash$ (*halt f*)^r = *initonly* (*f*^{r})
proof –
have *1*: $\vdash$ (*halt f*)^r = ($\Box$ (*empty* = *f*))^r **by** (*simp add*: *halt-d-def*)
have *2*: $\vdash$ ($\Box$ (*empty* = *f*))^r = *bi* ((*empty* = *f*)^r) **by** (*simp add*: *RBoxEqvBi*)

```
 have 3: ⊢ (empty = f)^r = (empty = f^r) by (metis REmptyEqvEmpty inteq-reflection rev-fun2)
 hence 4: ⊢ bi ( (empty = f)^r) = bi(empty = f^r) by (simp add: BiEqvBi)
 have 5: ⊢ bi(empty = f^r) = initonly(f^r) by (simp add: initonly-d-def)
 from 1 2 4 5 show ?thesis by fastforce
qed

lemma RInitonlyEqvHalt:
 ⊢ (initonly f)^r = halt(f^r)
proof –
 have 1: ⊢ (initonly f)^r = (bi (empty = f))^r by (simp add: initonly-d-def)
 have 2: ⊢ (bi (empty = f))^r = □((empty = f)^r) by (simp add: RBiEqvBox)
 have 3: ⊢ (empty = f)^r = (empty = f^r) by (metis REmptyEqvEmpty inteq-reflection rev-fun2)
 hence 4: ⊢ □ ((empty = f)^r) = □(empty = f^r) by (simp add: BoxEqvBox)
 have 5: ⊢ □(empty = f^r) = halt(f^r) by (simp add: halt-d-def)
 from 1 2 4 5 show ?thesis by fastforce
qed

lemma RRHaltEqvInitonly:
 ⊢ (halt (f^r))^r = initonly (f)
proof –
 have 1: ⊢ (halt (f^r))^r = initonly ((f^r)^r) using RHaltEqvInitonly by blast
 have 2: ⊢ initonly ((f^r)^r) = initonly(f)  using EqvReverseReverse by (metis inteq-reflection)
 from 1 2 show ?thesis by fastforce
qed

lemma RRInitonlyEqvHalt :
 ⊢ (initonly (f^r))^r = halt(f)
proof –
 have 1: ⊢ (initonly (f^r))^r = halt((f^r)^r) using RInitonlyEqvHalt by blast
 have 2: ⊢ halt((f^r)^r) = halt(f) using EqvReverseReverse by (metis inteq-reflection)
 from 1 2 show ?thesis by fastforce
qed

lemma RKeepEqvKeep :
 ⊢ (keep f)^r = keep(f^r)
proof –
 have 1: ⊢ (keep f)^r = (ba(skip ⟶ f))^r by (simp add: keep-d-def)
 have 2: ⊢ (ba(skip ⟶ f))^r = ba((skip ⟶ f)^r) by (simp add:RBaEqvBa)
 have 3: ⊢ (skip ⟶ f)^r = (skip ⟶ f^r) by (metis all-rev-eq(12) rev-fun2)
 hence 4: ⊢ ba((skip ⟶ f)^r) = ba(skip ⟶ f^r) by (simp add: BaEqvBa)
 have 5: ⊢ ba(skip ⟶ f^r) = keep(f^r) by (simp add: keep-d-def)
 from 1 2 4 5 show ?thesis by fastforce
qed

lemma RRKeepEqvKeep :
 ⊢ (keep (f^r))^r = keep(f)
proof –
 have 1: ⊢ (keep (f^r))^r = keep( (f^r)^r ) using RKeepEqvKeep by blast
 have 2: ⊢ keep( (f^r)^r ) = keep(f) using EqvReverseReverse by (metis inteq-reflection)
 from 1 2 show ?thesis by fastforce
```

```
qed

lemma NextDiamondEqvDiamondNext:
 ⊢ ○( ◇ f) = ◇(○f)
proof −
 have 1: ⊢ #True;skip = skip;#True  by (rule TrueChopSkipEqvSkipChopTrue)
 hence 2: ⊢ (#True;skip);f =( skip;#True);f using LeftChopEqvChop by blast
 have 3: ⊢ (#True;skip);f = #True;(skip;f)  by (simp add: ChopAssocB)
 have 4: ⊢ ( skip;#True);f = skip;(#True;f)  by (simp add: ChopAssocB)
 from 2 3 4 show ?thesis by (metis int-eq next-d-def sometimes-d-def)
qed

lemma WeakNextBoxInduct:
 assumes ⊢ wnext (□ f) ⟶ f
 shows ⊢ f
proof −
 have  1: ⊢ wnext (□ f) ⟶ f using assms by blast
 hence 2: ⊢ ¬ f ⟶ ¬ ( wnext (□ f))  by fastforce
 hence 3: ⊢ ¬ f ⟶  ○ (¬ (□ f)) by (simp add: wnext-d-def)
 have  4: ⊢ (¬ (□ f)) =  (◇ (¬ f)) by (auto simp: always-d-def)
 hence 5: ⊢ ○(¬ (□ f)) = ○ (◇ (¬ f)) using NextEqvNext by blast
 have  6: ⊢ ¬ f ⟶  ○ (◇ (¬ f)) using 3 5 by fastforce
 have  7: ⊢ ○ (◇ (¬ f)) = ◇( ○ (¬ f))  using NextDiamondEqvDiamondNext by blast
 have  8: ⊢ ¬ f ⟶  ◇( ○ (¬ f))  using 6 7 by fastforce
 have  9: ⊢ ◇(¬ f) ⟶  ◇(◇( ○ (¬ f))) using 8 DiamondImpDiamond by blast
 have 10: ⊢ ◇(◇( ○ (¬ f))) = ◇( ○ (¬ f)) using DiamondDiamondEqvDiamond by blast
 have 11: ⊢ ◇(¬ f) ⟶ ◇( ○ (¬ f)) using 9 10 by fastforce
 have 12: ⊢ ◇(¬ f) ⟶ ○ (◇ (¬ f)) using 7 11 by fastforce
 hence 13: ⊢ ¬( ◇(¬ f)) using NextLoop by blast
 hence 14: ⊢ □f by (simp add: always-d-def)
 have 15: ⊢ □f ⟶ f using BoxElim by blast
 from 14 15 show ?thesis using MP by blast
qed

lemma RassignEqvTAssign:
⊢ ($v = e)^r = (v ← e^r)
proof −
 have 1: ⊢ ($v = e)^r = (($v)^r = e^r) by (simp add: rev-fun2)
 have 2: ⊢ (($v)^r = e^r) = ((!v) = e^r) by (simp add: all-rev-eq(8))
 have 3: ⊢ ((!v) = e^r) = (v ← e^r) by (simp add: intI temporal-assign-d-def)
 from 1 2 3 show ?thesis by fastforce
qed

lemma RTAssignEqvAssign:
⊢ (v ← e)^r = ($v = e^r)
proof −
 have 1: ⊢ (v ← e)^r = (!v = e)^r by (simp add: REqvRule intI temporal-assign-d-def)
 have 2: ⊢ (!v = e)^r = ($v = e^r) by (metis all-rev-eq(11) rev-fun2)
 from 1 2 show ?thesis by fastforce
qed
```

lemma *RNextAssignEqvPrevAssign*:
$\vdash (v := e)^r = (v =: e^r)$
proof −
 have *1*: $\vdash (v := e)^r = (v\$ = e)^r$ **by** (*simp add*: *REqvRule intI next-assign-d-def*)
 have *2*: $\vdash (v\$ = e)^r = (v! = e^r)$ **by** (*metis all-rev-eq*(*9*) *rev-fun2*)
 have *3*: $\vdash (v! = e^r) = (v =: e^r)$ **by** (*simp add*: *prev-assign-d-def*)
 from *1 2 3* **show** *?thesis* **by** *fastforce*
qed

lemma *RPrevAssignEqvNextAssign*:
$\vdash (v =: e)^r = (v := e^r)$
proof −
 have *1*: $\vdash (v =: e)^r = (v! = e)^r$ **by** (*simp add*: *REqvRule intI prev-assign-d-def*)
 have *2*: $\vdash (v! = e)^r = (v\$ = e^r)$ **by** (*metis all-rev-eq*(*10*) *rev-fun2*)
 have *3*: $\vdash (v\$ = e^r) = (v := e^r)$ **by** (*simp add*: *next-assign-d-def*)
 from *1 2 3* **show** *?thesis* **by** *fastforce*
qed

lemma *RGetsEqvBaSkipImp*:
$\vdash (v \ gets \ e)^r = ba(skip \longrightarrow (\$v = e^r))$
proof −
 have *1*: $\vdash (v \ gets \ e)^r = (ba(skip \longrightarrow (!v = e)))^r$
  using *gets-d-def temporal-assign-d-def keep-d-def REqvRule*
  by (*metis Prop04 ba-d-def int-simps*(*15*))
 have *2*: $\vdash (ba(skip \longrightarrow (!v = e)))^r = ba \ (\ (skip \longrightarrow (!v = e))^r \ )$
  by (*simp add*: *RBaEqvBa*)
 have *3*: $\vdash (skip \longrightarrow (!v = e))^r = (skip \longrightarrow (\$v = e^r))$
  by (*simp add*: *all-rev-eq*(*11*) *all-rev-eq*(*12*) *all-rev-eq*(*3*))
 hence *4*: $\vdash ba \ ((skip \longrightarrow (!v = e))^r) = ba \ (skip \longrightarrow (\$v = e^r))$
  by (*simp add*: *BaEqvBa*)
 from *1 2 4* **show** *?thesis* **by** *fastforce*
qed

lemma *RIfThenElse*:
$\vdash (if_i \ f0 \ then \ f1 \ else \ f2)^r = if_i \ (f0^r) \ then \ (f1^r) \ else \ (f2^r)$
by (*simp add*: *all-rev-eq*(*2*) *all-rev-eq*(*3*) *ifthenelse-d-def*)

lemma *RWhile*:
$\vdash (init \ f \wedge while \ f0 \ do \ f1)^r = (\ fin(f) \wedge ((f0^r) \wedge (f1^r))^\star \wedge init \ (\neg(f0)) \ )$
proof −
 have *1*: $\vdash (init \ f \wedge while \ f0 \ do \ f1)^r = (\ init \ f \wedge (f0 \wedge f1)^\star \wedge fin \ (\neg f0) \ )^r$
  by (*simp add*: *while-d-def*)
 have *2*: $\vdash (\ init \ f \wedge (f0 \wedge f1)^\star \wedge fin \ (\neg f0) \ )^r = ((init \ f)^r \wedge ((f0 \wedge f1)^\star)^r \wedge (fin \ (\neg f0))^r)$
  by (*simp add*: *all-rev-eq*(*3*))
 have *3*: $\vdash (init \ f)^r = fin(f)$
  by (*simp add*: *RInitEqvFin*)
 have *4*: $\vdash ((f0 \wedge f1)^\star)^r = ((f0^r) \wedge (f1^r))^\star$
  by (*metis RevChopstar all-rev-eq*(*3*))
 have *5*: $\vdash (fin \ (\neg f0))^r = init \ (\neg(f0))$

```
      by (metis RFinEqvInit)
  have 6: ⊢ ((init f)^r ∧ ((f0 ∧ f1)^⋆)^r ∧ (fin (¬f0))^r) =
             ( fin(f) ∧ ((f0^r) ∧ (f1^r))^⋆ ∧ init (¬(f0)) ) using 3 4 5 by fastforce
  from 1 2 6 show ?thesis by fastforce
qed

end


theory FOTheorems
  imports
    Theorems
begin
```

5 First Order ITL theorems

We give the proofs of a list of first order ITL theorems.

```
lemmas EExI-unl = EExI[unlift-rule] — w ⊨ F x ⟹ w ⊨ (∃∃ x. F x)


lemma EExNoDep:
 ⊢ (∃∃ x. G) = G
proof –
  have 1: ⊢ G ⟶ (∃∃ x. G) by (meson EExI)
  have 2: ⋀ x. ⊢ G ⟶ G by simp
  have 3: ⊢ (∃∃ x. G) ⟶ G  using 2 by (meson EExE)
  from 1 3 show ?thesis using int-iffI by blast
qed

lemma AAxNoDep:
 ⊢ (∀∀ x. G) = G
using EExNoDep AAxDef EExE EExI
by (smt Valid-def exist-state-d-def intensional-rews(2) intensional-rews(3))

lemma EExEqvRule:
  assumes ⋀ x. ⊢ F x = G x
  shows  ⊢ (∃∃ x. F x) = (∃∃ x. G x)
by (metis EExE EExI assms int-iffD1 int-iffD2 int-iffI lift-imp-trans)

lemma AAxImpEEx:
 ⊢ (∀∀ x. F x) ⟶ (∃∃ x. F x)
by (simp add: exist-state-d-def forall-state-d-def intI)

lemma EExImpRule:
  assumes ⊢ F x ⟶ G x
  shows  ⊢ (∃∃ x. F x ⟶  G x)
using assms by (meson MP EExI)

lemma EExImpRuleDist:
  assumes ⊢ F x ⟶ G x
  shows  ⊢ (∀∀ x. F x) ⟶ (∃∃ x. G x)
proof −
  have 1: ⊢ (F x)  ⟶ (∃∃ x. G x) using EExI assms lift-imp-trans by blast
  have 2: ⊢  ¬(F x) ∨ (∃∃ x. G x) using 1 by auto
  have 3: ⊢ ¬(F x) ⟶ (∃∃ x. ¬(F x)) by (meson EExI)
  have 4: ⊢ (∃∃ x. ¬(F x)) = (¬(∀∀ x. F x)) using AAxDef by fastforce
  from 2 3 4 show ?thesis by fastforce
qed

lemma EExImpNoDepDist:
  assumes ⊢ R ⟶ G x
  shows  ⊢ R ⟶ (∃∃ x. G x)
using assms by (metis EExI lift-imp-trans)

lemma EExOrDist-1:
 ⊢ (∃∃ x. H x) ⟶ (∃∃ x. (F x) ∨ (H x))
proof −
  have 1: ⋀ x. ⊢ H x ⟶ F x ∨ H x  by (simp add: Valid-def)
  have 2: ⋀ x. ⊢ F x ∨ H x ⟶ (∃∃ x. (F x) ∨ (H x))  by (meson EExI)
  have 3: ⋀ x. ⊢ H x ⟶ (∃∃ x. (F x) ∨ (H x)) using 1 2 by (meson lift-imp-trans)
  from 3 show ?thesis using EExE by blast
qed

lemma EExOrDist-2:
 ⊢ (∃∃ x. F x) ⟶ (∃∃ x. (F x) ∨ (H x))
proof −
  have 1: ⋀ x. ⊢ F x ⟶ F x ∨ H x  by (simp add: Valid-def)
  have 2: ⋀ x. ⊢ F x ∨ H x ⟶ (∃∃ x. (F x) ∨ (H x))  by (meson EExI)
  have 3: ⋀ x. ⊢ F x ⟶ (∃∃ x. (F x) ∨ (H x)) using 1 2 by (meson lift-imp-trans)
  from 3 show ?thesis using EExE by blast
qed

lemma EExOrDist-3:
⊢ (∃∃ x. F x) ∨ (∃∃ x. H x) ⟶ (∃∃ x. (F x) ∨ (H x))
using EExOrDist-2 EExOrDist-1 by fastforce

lemma EExOrDist-4:
  ⊢ (∃∃ x. (F x) ∨ (H x)) ⟶ (∃∃ x. F x) ∨ (∃∃ x. H x)
proof −
  have 1: ⋀ x. ⊢ (F x) ∨ (H x) ⟶ (∃∃ x. F x) ∨ (∃∃ x. H x)
  by (simp add: EExI-unl intI)
  from 1 show ?thesis by (simp add: EExE)
qed

lemma EExOrDist:
⊢ ((∃∃ x. F x) ∨ (∃∃ x. H x)) = (∃∃ x. (F x) ∨ (H x))
using  EExOrDist-3 EExOrDist-4 by fastforce

lemma EExOrImport-1:
```

```
⊢ G ⟶ (∃∃ x. G ∨ (F x))
by (simp add: EExI-unI Valid-def)

lemma EExOrImport-2:
⊢ (∃∃ x. F x) ⟶ (∃∃ x. G ∨ (F x))
by (simp add: EExOrDist-1)

lemma EExOrImport-3:
⊢ (G ∨ (∃∃ x. F x)) ⟶ (∃∃ x. G ∨ (F x))
using EExOrImport-1 EExOrImport-2  by fastforce

lemma EExOrImport-4:
⊢ (∃∃ x. G ∨ F x) ⟶ (G ∨ (∃∃ x. F x))
proof –
 have 1: ⋀ x. ⊢ G ∨ F x ⟶ G ∨ (∃∃ x. F x) by (meson EExI int-iffD2 int-simps(27) Prop04 Prop08)
 from 1 show ?thesis by (simp add: EExE)
qed

lemma EExOrImport:
⊢ (G ∨ (∃∃ x. F x)) = (∃∃ x. G ∨ F x)
by (metis EExOrImport-3 EExOrImport-4 int-iffI)

lemma EExAndImport-1:
⊢ G ∧ (∃∃ x. F x) ⟶ (∃∃ x. G ∧ F x)
proof –
 have 1: ⊢ (G ∧ (∃∃ x. F x) ⟶ (∃∃ x. G ∧ F x))=
            ((∃∃ x. F x) ⟶ (G ⟶ (∃∃ x. G ∧ F x)))  by fastforce
 have 2: ⋀ x. ⊢ F x ⟶ (G ⟶ (∃∃ x. G ∧ F x)) by (metis EExI int-eq lift-and-com Prop09)
 hence 3: ⊢ (∃∃ x. F x) ⟶ (G ⟶ (∃∃ x. G ∧ F x))   by (simp add: EExE)
 from 1 3 show ?thesis by auto
qed

lemma EExAndImport-2:
⊢ (∃∃ x. G ∧ F x) ⟶ G ∧ (∃∃ x. F x)
proof –
 have 1: ⋀ x. ⊢ G ∧ F x ⟶ G ∧ (∃∃ x. F x)
 by (metis EExI int-iffD2 lift-and-com lift-imp-trans Prop12)
 from 1 show ?thesis by (simp add: EExE)
qed

lemma EExAndImport:
⊢ (G ∧ (∃∃ x. F x)) = (∃∃ x. G ∧ F x)
by (simp add: EExAndImport-1 EExAndImport-2 int-iffI)


lemma EExAndDist:
 assumes ⊢ F x ∧ G x
 shows   ⊢ (∃∃ x. F x) ∧ (∃∃ x. G x)
proof –
 have 1: ⊢ F x  using assms by fastforce
 have 2: ⊢ G x  using assms by fastforce
 have 3: ⊢ (∃∃ x. F x) using 1 by (meson EExI MP)
 have 4: ⊢ (∃∃ x. G x) using 2 by (meson EExI MP)
 from 3 4 show ?thesis by fastforce
qed


lemma EExAndNoDepDist:
 assumes ⊢ R ∧ G x
 shows   ⊢ R ∧ (∃∃ x. G x)
proof −
 have 1: ⊢ R  using assms by fastforce
 have 2: ⊢ G x using assms by fastforce
 have 3: ⊢ (∃∃ x. G x) using 2 by (meson EExI MP)
 from 1 3 show ?thesis by fastforce
qed

lemma Spec:
⊢ (∀∀ x. F x) ⟶ F x
proof −
 have 1: ⊢ ¬(F x) ⟶ (∃∃ x. ¬(F x)) by (meson EExI)
 have 2: ⊢ ¬(∃∃ x. ¬(F x)) ⟶ F x using 1 by auto
 from 2 show ?thesis using AAxDef by fastforce
qed

lemma AAxE:
 assumes ⊢ (∀∀ x. F x)
         ⊢ F x ⟶ R
 shows   ⊢ R
using MP Spec assms(1) assms(2) by blast

lemma AAxI:
 assumes ⋀ x. ⊢ F x
 shows   ⊢ (∀∀ x. F x)
unfolding AAxDef
using AAxDef EExE assms
by (smt Valid-def int-simps(15) unl-lift unl-lift2)

lemma AAxEqvRule:
 assumes ⋀ x. ⊢ F x = G x
 shows   ⊢ (∀∀ x. F x) = (∀∀ x. G x)
by (metis (mono-tags, lifting) AAxDef EExEqvRule assms int-iffD1 int-iffI
    inteq-reflection lift-imp-neg)

lemma AAxAndDist:
⊢ (∀∀ x. (F x) ∧ (G x)) = ((∀∀ x. F x) ∧ (∀∀ x. G x))
proof −
 have 1: ⊢ ((∃∃ x. ¬(F x)) ∨ (∃∃ x. ¬(G x))) = (∃∃ x. ¬(F x) ∨ ¬(G x)) by (simp add:EExOrDist)
 have 2: ⊢ ((∃∃ x. ¬(F x))) = (¬(∀∀ x. F x) )  using AAxDef by fastforce
 have 3: ⊢ ((∃∃ x. ¬(G x))) = (¬(∀∀ x. G x) )  using AAxDef by fastforce
 have 4: ⊢ ((∃∃ x. ¬(F x)) ∨ (∃∃ x. ¬(G x))) = (¬(∀∀ x. F x) ∨ ¬(∀∀ x. G x) )
     using 2 3 by fastforce
 have 5: ⋀ x . ⊢ (¬(F x) ∨ ¬(G x)) = (¬((F x) ∧ (G x))) by auto
 have 6: ⊢ (∃∃ x. ¬(F x) ∨ ¬(G x)) = (∃∃ x. ¬((F x) ∧ (G x)))  using 5 by (simp add: EExEqvRule)
 have 7: ⊢ (∃∃ x. ¬((F x) ∧ (G x))) = (¬(∀∀ x. (F x) ∧ (G x))) using AAxDef by fastforce
 have 8: ⊢ (¬(∀∀ x. F x) ∨ ¬(∀∀ x. G x) ) = (¬( (∀∀ x. F x) ∧ (∀∀ x. G x))) by fastforce
 have 9: ⊢ (¬( (∀∀ x. F x) ∧ (∀∀ x. G x))) = (¬(∀∀ x. (F x) ∧ (G x)))
     using 1 4 6 7 8 by fastforce
 from 9 show ?thesis by fastforce
qed

lemma AAxAndImport:
⊢ (G ∧ (∀∀ x. F x)) = (∀∀ x. G ∧ F x)
proof –
 have 1: ⊢ (¬ G ∨ (∃∃ x. ¬(F x))) = (∃∃ x. ¬ G ∨ ¬(F x))  by (simp add: EExOrImport)
 have 2: ⊢ ( (∃∃ x. ¬(F x))) = (¬((∀∀ x. F x))) using AAxDef by fastforce
 have 3: ⊢ ( ¬ G ∨ (∃∃ x. ¬(F x))) = (¬(G ∧ (∀∀ x. F x))) using 2 by fastforce
 have 4: ⋀ x. ⊢ (¬ G ∨ ¬(F x)) = (¬(G ∧ F x)) by auto
 have 5: ⊢ (∃∃ x. ¬ G ∨ ¬(F x)) = (∃∃ x. ¬(G ∧ F x)) using 4 by (simp add: EExEqvRule)
 have 6: ⊢ (∃∃ x. ¬(G ∧ F x)) = (¬(∀∀ x. G ∧ F x)) using AAxDef by fastforce
 have 7: ⊢ (¬(G ∧ (∀∀ x. F x))) = (¬(∀∀ x. G ∧ F x)) using 1 3 5 6 by fastforce
 from 7 show ?thesis by fastforce
qed

lemma AAxOrImport:
⊢ (G ∨ (∀∀ x. F x)) = (∀∀ x. G ∨ F x)
proof –
 have 1: ⊢ (¬ G ∧ (∃∃ x. ¬(F x))) = (∃∃ x. ¬ G ∧ ¬(F x)) by (simp add: EExAndImport)
 have 2: ⊢ (∃∃ x. ¬(F x)) =  (¬((∀∀ x. F x))) using AAxDef by fastforce
 have 3: ⊢ ( ¬ G ∧ (∃∃ x. ¬(F x))) = (¬(G ∨ (∀∀ x. F x))) using 2 by fastforce
 have 4: ⋀ x. ⊢ (¬ G ∧ ¬(F x)) = (¬(G ∨ F x)) by auto
 have 5: ⊢ (∃∃ x. ¬ G ∧ ¬(F x)) = (∃∃ x. ¬(G ∨ F x)) using 4 by (simp add: EExEqvRule)
 have 6: ⊢ (∃∃ x. ¬(G ∨ F x)) = (¬(∀∀ x. G ∨ F x)) using AAxDef by fastforce
 have 7: ⊢ (¬(G ∨ (∀∀ x. F x))) = (¬(∀∀ x. G ∨ F x)) using 1 3 5 6 by fastforce
 from 7 show ?thesis by auto
qed

lemma EExImpChopRule:
 assumes ⊢ F x ⟶ G x
 shows  ⊢ (∃∃ x. H;(F x) ⟶ H;(G x))
using RightChopImpChop EExImpRule assms by (smt MP EExI)

lemma EExChopRight:
⊢ (∃∃ x. (F x);F1) ⟶ (∃∃ x. F x);F1
proof –
 have 1: ⋀x. ⊢ (F x);F1 ⟶ (∃∃ x. F x);F1 by (simp add: EExI LeftChopImpChop)
 from 1 show ?thesis by (simp add: EExE)
qed

lemma EExChopRightNoDep:
```

$\vdash (\exists\exists\ x.\ (F\ x);F1) = (\exists\exists\ x.\ (F\ x));F1$
by (*simp add*: *exist-state-d-def Valid-def chop-defs*, *auto*)

lemma *EExChopLeft* :
$\vdash (\exists\exists\ x.\ F1;(F\ x)\ ) \longrightarrow F1;(\exists\exists\ x.\ F\ x)$
proof −
have *1*: $\bigwedge x.\ \vdash F1;(F\ x) \longrightarrow F1;(\exists\exists\ x.\ F\ x)$ **by** (*simp add*: *EExI RightChopImpChop*)
from *1* **show** *?thesis* **by** (*simp add*: *EExE*)
qed

lemma *EExChopLeftNoDep*:
$\vdash (\exists\exists\ x.\ F1;(F\ x)\ ) = F1;(\exists\exists\ x.\ F\ x)$
by (*simp add*: *exist-state-d-def Valid-def chop-defs*, *auto*)

lemma *EExEExChopEqvEExEExChop*:
$\vdash (\exists\exists\ v.\ (\exists\exists\ y.\ (F2\ v);(F3\ y)\ )) = (\exists\exists\ y.\ (\exists\exists\ v.\ (F2\ v);(F3\ y)\ ))$
by (*simp add*: *exist-state-d-def Valid-def chop-defs*, *blast*)

lemma *EExEExChopEqvEExChopEExA*:
$\vdash (\exists\exists\ v.\ (\exists\exists\ y.\ (F2\ v);(F3\ y)\ )) = (\exists\exists\ v.\ (F2\ v);(\exists\exists\ y.\ (F3\ y)\ )\ )$
by (*simp add*: *exist-state-d-def Valid-def chop-defs*, *blast*)

lemma *EExEExChopEqvEExChopEExB*:
$\vdash (\exists\exists\ y.\ (\exists\exists\ v.\ (F2\ v);(F3\ y)\ )) = (\exists\exists\ y.\ (\exists\exists\ v.\ (F2\ v));\ (F3\ y))$
by (*simp add*: *exist-state-d-def Valid-def chop-defs*, *blast*)

lemma *EExEExChopEqvEExChopEExC*:
$\vdash (\exists\exists\ v.\ (\exists\exists\ y.\ (F2\ v);(F3\ y)\ )) = (\exists\exists\ v.\ (F2\ v));(\exists\exists\ y.\ (F3\ y))$
by (*metis EExChopRightNoDep EExEExChopEqvEExChopEExA EExNoDep Prop04*)

lemma *AAxRev*:
$\vdash (\forall\forall\ x.\ F\ x)^r = (\forall\forall\ x.\ (F\ x)^r)$
proof −
have *1*: $\vdash (\forall\forall\ x.\ F\ x) = (\neg(\exists\exists\ x.\ \neg(F\ x)))$ **using** *AAxDef* **by** *blast*
have *2*: $\vdash (\forall\forall\ x.\ F\ x)^r = (\neg(\exists\exists\ x.\ \neg(F\ x)))^r$ **using** *REqvRule 1* **by** *blast*
have *3*: $\vdash (\neg(\exists\exists\ x.\ \neg(F\ x)))^r = (\neg((\exists\exists\ x.(\neg\ (F\ x)))^r))$ **by** (*simp add*: *rev-fun1*)
have *4*: $\vdash ((\exists\exists\ x.(\neg\ (F\ x)))^r) = ((\exists\exists\ x.(\neg\ (F\ x))^r))$ **by** (*simp add*: *EExRev*)
hence *5*: $\vdash (\neg((\exists\exists\ x.(\neg\ (F\ x)))^r)) = (\neg(\exists\exists\ x.(\neg\ (F\ x))^r))$ **by** *auto*
have *51*: $\bigwedge x.\ \vdash (\neg\ (F\ x))^r = (\neg(\ (F\ x)^r))$ **by** (*simp add*: *rev-fun1*)
hence *52*: $\vdash (\exists\exists\ x.(\neg\ (F\ x))^r) = (\exists\exists\ x.\neg(\ (F\ x)^r))$ **using** *EExEqvRule* **by** *fastforce*
hence *6*: $\vdash (\neg(\exists\exists\ x.(\neg\ (F\ x))^r)) = (\neg(\exists\exists\ x.\neg(\ (F\ x)^r)))$ **by** *fastforce*
have *7*: $\vdash (\neg(\exists\exists\ x.\neg(\ (F\ x)^r))) = (\forall\forall\ x.\ (F\ x)^r)$ **using** *AAxDef* **by** *fastforce*
from *1 2 3 5 6 7* **show** *?thesis* **by** *fastforce*
qed

lemma *ExLen*:
$\vdash \exists\ n.\ len(n)$
by (*simp add*: *Valid-def len-defs*)

lemma *CSPowerChop*:

$\vdash (f^{\star}) = (\exists\, n.\ powerchop\ f\ n)$
by (*simp add*: *chopstar-eqv-power-chop Valid-def*)

lemma *ExChopRightNoDep*:
$\vdash (\exists\ x.\ (F\ x);F1) = (\exists\ x.\ (F\ x));F1$
by (*simp add*: *Valid-def chop-defs*, *auto*)

lemma *ExChopLeftNoDep*:
$\vdash (\exists\ x.\ F1;(F\ x)\ ) = F1;(\exists\ x.\ F\ x)$
by (*simp add*: *Valid-def chop-defs*, *auto*)

lemma *ExExEqvExEx*:
$\vdash (\exists\ x.\ (\exists\ y.\ (F1\ x);(F2\ y))) = (\exists\ y.\ (\exists\ x.\ (F1\ x);(F2\ y)))$
by (*simp add*: *Valid-def chop-defs*, *auto*)

end

theory *First*
imports
Theorems
begin

6 The First Occurrence Operator in ITL

Runtime verification (RV) has gained significant interest in recent years. The behaviour of a program can be verified in real time by analysing its evolving trace. This approach has two significant benefits over static verification techniques such as model checking. Firstly, it is only necessary to verify actual execution paths rather than all possible paths. Secondly, it is possible to react at runtime should the program diverge from its specified behaviour. RV does not replace traditional verification techniques but it does provide an extra layer of security.

Linear Temporal Logic (LTL) is a popular formalism for writing specifications from which RV monitors can be derived automatically. By contrast, Interval Temporal Logic (ITL) has not been as widely represented in this field despite being more expressive and compositional. The principal issue is efficiency. ITL uses non-deterministic operators to construct sequential and iterative specifications (chop and chop-star, respectively) and these introduce combinatorial complexity. Approaches to mitigate this include using a deterministic subset of ITL or adapting the semantics to include a deterministic chop operator. This thesis proposes an alternative approach, wholly within existing ITL, and based upon a new, derived operator called “first occurrence”.

A theory of first occurrence is developed and used to derive an algebra of RV monitors.

6.1 Definitions

6.1.1 Definitions Strict Initial and Final

definition *bs-d* :: $('a::world)\ formula \Rightarrow\ 'a\ formula$

where
bs-d f ≡ *LIFT*(*empty* ∨ ((*bi f*) ; *skip*))

definition *bt-d* :: (*'a*::*world*) *formula* ⇒ *'a formula*
where
bt-d f ≡ *LIFT*(*empty* ∨ (*skip*;(□ *f*)))

syntax
-bs-d :: *lift* ⇒ *lift* ((*bs* -) [*88*] *87*)
-bt-d :: *lift* ⇒ *lift* ((*bt* -) [*88*] *87*)

syntax (*ASCII*)
-bs-d :: *lift* ⇒ *lift* ((*bs* -) [*88*] *87*)
-bt-d :: *lift* ⇒ *lift* ((*bt* -) [*88*] *87*)

translations
-bs-d ⇌ *CONST bs-d*
-bt-d ⇌ *CONST bt-d*

definition *ds-d* :: (*'a*::*world*) *formula* ⇒ *'a formula*
where
ds-d f ≡ *LIFT* (¬ (*bs* (¬ *f*)))

definition *dt-d* :: (*'a*::*world*) *formula* ⇒ *'a formula*
where
dt-d f ≡ *LIFT* (¬ (*bt* (¬ *f*)))

syntax
-ds-d :: *lift* ⇒ *lift* ((*ds* -) [*88*] *87*)
-dt-d :: *lift* ⇒ *lift* ((*dt* -) [*88*] *87*)

syntax (*ASCII*)
-ds-d :: *lift* ⇒ *lift* ((*ds* -) [*88*] *87*)
-dt-d :: *lift* ⇒ *lift* ((*dt* -) [*88*] *87*)

translations
-ds-d ⇌ *CONST ds-d*
-dt-d ⇌ *CONST dt-d*

6.1.2 Definition First and Last Operators

definition *first-d* :: (*'a*::*world*) *formula* ⇒ *'a formula*
where
first-d f ≡ *LIFT* (*f* ∧ (*bs* (¬ *f*)))

definition *last-d* :: (*'a*::*world*) *formula* ⇒ *'a formula*
where
last-d f ≡ *LIFT* (*f* ∧ (*bt* (¬ *f*)))

syntax
 -first-d :: *lift* ⇒ *lift* ((▷ -) [*88*] *87*)
 -last-d :: *lift* ⇒ *lift* ((◁ -) [*88*] *87*)

syntax (*ASCII*)
 -first-d :: *lift* ⇒ *lift* ((*first* -) [*88*] *87*)
 -last-d :: *lift* ⇒ *lift* ((*last* -) [*88*] *87*)

translations
 -first-d ⇌ *CONST first-d*
 -last-d ⇌ *CONST last-d*

6.2 First and Time Reversal

lemma *BsEqvRule*:
 assumes ⊢ *f* = *g*
 shows ⊢ *bs f* = *bs g*
proof −
 have *1*: ⊢ *f* = *g* **using** *assms* **by** *auto*
 hence *2*: ⊢ *bi*(*f*) = *bi*(*g*) **by** (*simp add*: *BiEqvBi*)
 hence *3*: ⊢ *bi*(*f*);*skip* = *bi*(*g*);*skip* **by** (*simp add*: *LeftChopEqvChop*)
 hence *4*: ⊢ (*empty* ∨ *bi*(*f*);*skip*) = (*empty* ∨ *bi*(*g*);*skip*) **by** *auto*
 hence *5*: ⊢ *bs*(*f*) = *bs*(*g*) **by** (*simp add*: *bs-d-def*)
 from *1 2 3 4 5* **show** *?thesis* **by** *auto*
qed

lemma *BtEqvRule*:
 assumes ⊢ *f* = *g*
 shows ⊢ *bt f* = *bt g*
proof −
 have *1*: ⊢ *f* = *g* **using** *assms* **by** *auto*
 hence *2*: ⊢ □(*f*) = □(*g*) **by** (*simp add*: *BoxEqvBox*)
 hence *3*: ⊢ *skip*;□(*f*) = *skip*;□(*g*) **using** *RightChopEqvChop* **by** *blast*
 hence *4*: ⊢ (*empty* ∨ *skip*;□(*f*)) = (*empty* ∨ *skip*;□(*g*)) **by** *auto*
 hence *5*: ⊢ *bt*(*f*) = *bt*(*g*) **by** (*simp add*: *bt-d-def*)
 from *1 2 3 4 5* **show** *?thesis* **by** *auto*
qed

lemma *FstEqvRule*:
 assumes ⊢ *f* = *g*
 shows ⊢ ▷*f* = ▷*g*
proof −
 have *1*: ⊢ *f* = *g* **using** *assms* **by** *auto*
 hence *2*: ⊢ (¬ *f*) = (¬ *g*) **by** *auto*
 hence *3*: ⊢ *bs*(¬ *f*) = *bs*(¬ *g*) **by** (*simp add*: *BsEqvRule*)
 hence *4*: ⊢ (*f* ∧ *bs*(¬ *f*)) = (*g* ∧ *bs*(¬ *g*)) **using** *1* **by** *fastforce*
 from *4* **show** *?thesis* **by** (*simp add*:*first-d-def*)
qed

lemma *LstEqvRule*:

assumes $\vdash f = g$
shows $\vdash \lhd f = \lhd g$
proof −
have *1*: $\vdash f = g$ **using** *assms* **by** *auto*
hence *2*: $\vdash (\neg f) = (\neg g)$ **by** *auto*
hence *3*: $\vdash \mathit{bt}(\neg f) = \mathit{bt}(\neg g)$ **by** (*simp add*: *BtEqvRule*)
hence *4*: $\vdash (f \wedge \mathit{bt}(\neg f)) = (g \wedge \mathit{bt}(\neg g))$ **using** *1* **by** *fastforce*
from *4* **show** *?thesis* **by** (*simp add*:*last-d-def*)
qed

lemma *RBsEqvBt*:
$\vdash (\mathit{bs}\ f)^r = (\mathit{bt}\ (f^r))$
proof −
have *1*: $\vdash (\mathit{bs}\ f)^r = (\mathit{empty} \vee ((\mathit{bi}\ f)\ ;\ \mathit{skip}))^r$
by (*simp add*: *bs-d-def*)
have *2*: $\vdash (\mathit{empty} \vee ((\mathit{bi}\ f)\ ;\ \mathit{skip}))^r = (\mathit{empty}^r \vee ((\mathit{bi}\ f)\ ;\ \mathit{skip})^r)$
using *ROr* **by** *blast*
have *3*: $\vdash (\mathit{empty}^r \vee ((\mathit{bi}\ f)\ ;\ \mathit{skip})^r) = (\mathit{empty} \vee (\mathit{skip}^r;(\mathit{bi}\ f)^r))$
using *REmptyEqvEmpty RevChop* **by** *fastforce*
have *4*: $\vdash (\mathit{empty} \vee (\mathit{skip}^r;(\mathit{bi}\ f)^r)) = (\mathit{empty} \vee (\mathit{skip};\Box\ (f^r)))$
by (*metis 3 RBiEqvBox RevSkip int-eq*)
have *5*: $\vdash (\mathit{empty} \vee (\mathit{skip};\Box\ (f^r))) = (\mathit{bt}\ (f^r))$
by (*simp add*: *bt-d-def*)
from *1 2 3 4 5* **show** *?thesis* **by** *fastforce*
qed

lemma *RRBsEqvBt*:
$\vdash (\mathit{bs}\ (f^r))^r = (\mathit{bt}\ (f))$
proof −
have *1*: $\vdash (\mathit{bs}\ (f^r))^r = \mathit{bt}\ ((f^r)^r)$ **using** *RBsEqvBt* **by** *blast*
have *2*: $\vdash \mathit{bt}\ ((f^r)^r) = \mathit{bt}\ f$ **using** *EqvReverseReverse* **using** *BtEqvRule* **by** *blast*
from *1 2* **show** *?thesis* **by** *fastforce*
qed

lemma *RBtEqvBs*:
$\vdash (\mathit{bt}\ f)^r = (\mathit{bs}\ (f^r))$
proof −
have *1*: $\vdash (\mathit{bt}\ f)^r = (\mathit{empty} \vee (\mathit{skip};\Box\ f))^r$
by (*simp add*: *bt-d-def*)
have *2*: $\vdash (\mathit{empty} \vee (\mathit{skip};\Box\ f))^r = (\mathit{empty}^r \vee (\mathit{skip};\Box\ f)^r)$
using *ROr* **by** *blast*
have *3*: $\vdash (\mathit{empty}^r \vee (\mathit{skip};\Box\ f)^r) = (\mathit{empty} \vee (\Box\ f)^r;\mathit{skip}^r)$
using *REmptyEqvEmpty RevChop* **by** *fastforce*
have *4*: $\vdash (\mathit{empty} \vee (\Box\ f)^r;\mathit{skip}^r) = (\mathit{empty} \vee (\mathit{bi}\ (f^r));\mathit{skip})$
by (*metis 3 RBoxEqvBi RevSkip int-eq*)
have *5*: $\vdash (\mathit{empty} \vee (\mathit{bi}\ (f^r));\mathit{skip}) = (\mathit{bs}\ (f^r))$
by (*simp add*: *bs-d-def*)
from *1 2 3 4 5* **show** *?thesis* **by** *fastforce*
qed

lemma *RRBtEqvBs*:
$\vdash (bt\ (f^r))^r = (bs\ (f))$
proof –
have *1*: $\vdash (bt\ (f^r))^r = bs\ ((f^r)^r)$ **using** *RBtEqvBs* **by** *blast*
have *2*: $\vdash bs\ ((f^r)^r) = bs\ f$ **using** *EqvReverseReverse* **using** *BsEqvRule* **by** *blast*
from *1 2* **show** *?thesis* **by** *fastforce*
qed

lemma *RFirstEqvLast*:
$\vdash (\rhd\ f)^r = (\lhd\ (f^r))$
proof –
have *1*: $\vdash (\rhd\ f)^r = (f \wedge bs(\neg\ f))^r$ **by** (*simp add*: *first-d-def*)
have *2*: $\vdash (f \wedge bs(\neg\ f))^r = (f^r \wedge (bs\ (\neg\ f))^r)$ **using** *RAnd* **by** *blast*
have *3*: $\vdash (f^r \wedge (bs\ (\neg\ f))^r) = (f^r \wedge bt\ ((\neg\ f)^r))$ **using** *RBsEqvBt* **by** *fastforce*
have *4*: $\vdash (f^r \wedge bt\ ((\neg\ f)^r)) = (f^r \wedge bt\ (\neg(f^r)))$ **using** *RNot int-eq* **by** *fastforce*
have *5*: $\vdash (f^r \wedge bt\ (\neg(f^r))) = (\lhd\ (f^r))$ **by** (*simp add*: *last-d-def*)
from *1 2 3 4 5* **show** *?thesis* **by** *fastforce*
qed

lemma *RRFirstEqvLast*:
$\vdash (\rhd\ (f^r))^r = (\lhd\ (f))$
proof –
have *1*: $\vdash (\rhd\ (f^r))^r = \lhd\ ((f^r)^r)$ **using** *RFirstEqvLast* **by** *blast*
have *2*: $\vdash \lhd\ ((f^r)^r) = \lhd\ f$ **using** *EqvReverseReverse* **using** *LstEqvRule* **by** *blast*
from *1 2* **show** *?thesis* **by** *fastforce*
qed

lemma *RLastEqvFirst*:
$\vdash (\lhd\ f)^r = (\rhd\ (f^r))$
proof –
have *1*: $\vdash (\lhd\ f)^r = (f \wedge bt(\neg\ f))^r$ **by** (*simp add*: *last-d-def*)
have *2*: $\vdash (f \wedge bt(\neg\ f))^r = (f^r \wedge (bt\ (\neg f))^r)$ **using** *RAnd* **by** *blast*
have *3*: $\vdash (f^r \wedge (bt\ (\neg f))^r) = (f^r \wedge bs\ (\ (\neg f)^r))$ **using** *RBtEqvBs* **by** *fastforce*
have *4*: $\vdash (f^r \wedge bs\ (\ (\neg f)^r)) = (f^r \wedge bs(\neg(f^r)))$ **using** *RNot int-eq* **by** *fastforce*
have *5*: $\vdash (f^r \wedge bs(\neg(f^r))) = (\rhd\ (f^r))$ **by** (*simp add*: *first-d-def*)
from *1 2 3 4 5* **show** *?thesis* **by** *fastforce*
qed

lemma *RRLastEqvFirst*:
$\vdash (\lhd\ (f^r))^r = (\rhd\ (f))$
proof –
have *1*: $\vdash (\lhd\ (f^r))^r = \rhd\ ((f^r)^r)$ **using** *RLastEqvFirst* **by** *blast*
have *2*: $\vdash \rhd\ ((f^r)^r) = \rhd\ f$ **using** *EqvReverseReverse* **using** *FstEqvRule* **by** *blast*
from *1 2* **show** *?thesis* **by** *fastforce*
qed

6.3 Semantic Theorems

6.3.1 Semantics First and Last Operators

lemma *FstAndBisem*:

```
 (intlen σ >0 ∧ (σ ⊨ f) ∧ ( σ ⊨  bi (¬f);skip)) =
  (intlen σ >0 ∧ (σ ⊨ f) ∧ (∀ ia<intlen (σ). (prefix ia  σ ⊨ ¬f)) )
apply (simp add: chop-defs bi-defs skip-defs)
apply (simp add: interval-prefix-length interval-suffix-length)
proof −
 have 1: (0 < intlen σ ∧ (σ ⊨ f) ∧
          (∃ i. (i ≤ intlen σ ⟶ (∀ ia≤i. ¬ (prefix ia (prefix i σ) ⊨ f)) ∧
          intlen σ − i = Suc 0) ∧ i ≤ intlen σ)
         ) =
         (0 < intlen σ ∧ (σ ⊨ f) ∧
          (∃ i. (i ≤ intlen σ ⟶ (∀ ia≤i. ¬ (prefix ia (prefix i σ) ⊨ f)) ∧
          i = intlen σ − Suc 0) ∧ i ≤ intlen σ)
         )
     by auto
 also have ... =
         (0 < intlen σ ∧ (σ ⊨ f) ∧
          (∀ ia≤(intlen σ − Suc 0). ¬ (prefix ia (prefix (intlen σ − Suc 0) σ) ⊨ f) )
         )
     using diff-le-self by blast
 also have ... =
         (intlen σ >0 ∧ (σ ⊨ f) ∧
           (∀ ia<intlen ( σ). ¬ (prefix ia (prefix (intlen σ − Suc 0) σ) ⊨ f))
         )  by (metis Suc-pred less-Suc-eq-le)
 also have ... =
         (intlen σ >0 ∧ (σ ⊨ f) ∧
          (∀ ia<intlen ( σ). (prefix ia (prefix (intlen σ − Suc 0) σ) ⊨ ¬f))
         )
     by auto
 also have ... =
         (intlen σ >0 ∧ (σ ⊨ f) ∧ (∀ ia<intlen ( σ). ¬ (prefix ia  σ ⊨ f)))
         by (simp add: interval-pref-pref-help)
 finally show (0 < intlen σ ∧ (σ ⊨ f) ∧
               (∃ i. (i ≤ intlen σ ⟶ (∀ ia≤i. ¬ (prefix ia (prefix i σ) ⊨ f)) ∧
                                 intlen σ − i = Suc 0) ∧ i ≤ intlen σ)
              ) =
              (0 < intlen σ ∧ (σ ⊨ f) ∧ (∀ ia<intlen σ. ¬ (prefix ia σ ⊨ f)))  .
qed

lemma Fstsem-0:
 (σ ⊨ ▷f) =
  (
   ( σ ⊨ f ∧ empty) ∨ (intlen σ >0  ∧ (σ ⊨ f) ∧ ( σ ⊨  bi (¬f);skip))
  )
apply (simp add: first-d-def bs-d-def) using empty-defs by auto

lemma Emptysem:
 (σ ⊨ f ∧ empty) = ((σ ⊨ f) ∧ intlen σ = 0)
using empty-defs by auto

lemma Fstsem:
```

```
(σ ⊨ ▷f) =
 (
  ( (σ ⊨ f) ∧ intlen σ = 0) ∨
  ( intlen σ>0 ∧ (σ ⊨ f) ∧ (∀ ia<intlen (σ). (prefix ia  σ ⊨ ¬f)))
 )
using Fstsem-0 Emptysem FstAndBisem by metis

lemma Lstsem:
 (σ ⊨ ◁ f) =
  ( ( (σ ⊨ f) ∧ intlen σ = 0) ∨
    ( intlen σ >0 ∧ (σ ⊨ f) ∧ (∀ ia < intlen σ. (suffix ((intlen σ) −ia)  σ ⊨ ¬f)) )
  )

proof −
 have (σ ⊨ ◁ f) = (σ ⊨ (▷ (f^r))^r)
      using  RRFirstEqvLast by fastforce
 also have  ... = (intrev σ ⊨ ▷ (f^r))
      by (metis reverse-d-def)
 also have  ... =
  (
   ( (intrev σ ⊨ f^r) ∧ intlen (intrev σ) = 0) ∨
   ( intlen (intrev σ)>0 ∧ (intrev σ ⊨ f^r) ∧
     (∀ ia<intlen (intrev σ). (prefix ia  (intrev σ) ⊨ ¬(f^r))))
  )
      using Fstsem by blast
 also have  ... =
  (
   ( ( σ ⊨ f) ∧ intlen (σ) = 0) ∨
   ( intlen (σ)>0 ∧ ( σ ⊨ f) ∧
     (∀ ia<intlen (intrev σ). (prefix ia  (intrev σ) ⊨ (¬(f))^r)))
  )
  by (simp add: reverse-d-def)
 also have  ... =
  (
   ( ( σ ⊨ f) ∧ intlen (σ) = 0) ∨
   ( intlen (σ)>0 ∧ ( σ ⊨ f) ∧
     (∀ ia<intlen (intrev σ). (intrev (prefix ia  (intrev σ)) ⊨ (¬(f)))))
  )
  by (simp add: reverse-d-def)
 also have  ... =
  (
   ( ( σ ⊨ f) ∧ intlen (σ) = 0) ∨
   ( intlen (σ)>0 ∧ ( σ ⊨ f) ∧
     (∀ ia<intlen (σ). ( (suffix ((intlen σ) − ia)  (σ)) ⊨ (¬(f)))))
  )
      by (simp add: interval-intrev-prefix)
 finally show
  (σ ⊨ ◁ f) =
  ( ( (σ ⊨ f) ∧ intlen σ = 0) ∨
    ( intlen σ >0 ∧ (σ ⊨ f) ∧
     (∀ ia < intlen σ. (suffix ((intlen σ) −ia)  σ ⊨ ¬f )) )
  )  .
qed
```

6.3.2 Various Semantic Lemmas

```
lemma DiLensem:
 (σ ⊨ di (f ∧ len(i))) =
   ( (prefix i σ ⊨ f ) ∧ i≤intlen σ)
apply (simp add: di-defs len-defs)
using interval-prefix-length-good by auto

lemma PrefixFstsem:
 ( (prefix i σ ⊨ ▷f ) ∧ i≤intlen σ) =
  ( i≤intlen σ ∧
   (
    ( (prefix i σ ⊨ f ) ∧ i = 0) ∨
    ( i>0 ∧ (prefix i σ ⊨ f ) ∧ (∀ ia<i. (prefix ia σ ⊨ ¬f )))
   )
  )
proof −
 have 1: ( ((prefix i σ) ⊨ ▷f )) =
  (
   ( ((prefix i σ) ⊨ f ) ∧ intlen (prefix i σ) = 0) ∨
   ( intlen (prefix i σ)>0 ∧ ((prefix i σ) ⊨ f ) ∧
     (∀ ia<intlen (prefix i σ). (prefix ia  (prefix i σ) ⊨ ¬f )) )
  )
       using Fstsem by blast
 hence 2: ( ((prefix i σ) ⊨ ▷f ) ∧ i≤intlen σ) =
  (  i≤intlen σ ∧ (
   ( ((prefix i σ) ⊨ f ) ∧ intlen (prefix i σ) = 0) ∨
   ( intlen (prefix i σ)>0 ∧ ((prefix i σ) ⊨ f ) ∧
     (∀ ia<intlen (prefix i σ). (prefix ia  (prefix i σ) ⊨ ¬f )) )
    )
  )
       by auto
 hence 3: ( ((prefix i σ) ⊨ ▷f ) ∧ i≤intlen σ) =
  (  i≤intlen σ ∧ (
   ( ((prefix i σ) ⊨ f ) ∧ i = 0) ∨
   ( i>0 ∧ ((prefix i σ) ⊨ f ) ∧ (∀ ia<i. (prefix ia  (prefix i σ) ⊨ ¬f )))
    )
  )
       by auto
 hence 4: ( ((prefix i σ) ⊨ ▷f ) ∧ i≤intlen σ) =
  (  i≤intlen σ ∧ (
   ( ((prefix i σ) ⊨ f ) ∧ i = 0) ∨
   ( i>0 ∧ ((prefix i σ) ⊨ f ) ∧ (∀ ia<i. (prefix ia σ ⊨ ¬f )))
    )
  )
      using interval-pref-pref-3 using less-imp-add-positive by fastforce
 from 4 show ?thesis by auto
qed

lemma PrefixFstAndsem:
 ( (prefix i σ ⊨ ▷f ∧ g) ∧ i≤intlen σ) =
  ( i≤intlen σ ∧
   (
    ( (prefix i σ ⊨ f ∧ g ) ∧ i = 0) ∨
    ( i>0 ∧ (prefix i σ ⊨ f ∧ g) ∧ (∀ ia<i. (prefix ia σ ⊨ ¬f)))
   )
 )
using PrefixFstsem by (metis unl-lift2)

lemma DiLenFstsem:
 (σ ⊨ di (▷f ∧ len(i))) =
   ( i≤intlen σ ∧
   (
    ( (prefix i σ ⊨ f) ∧ i = 0) ∨
    ( i>0 ∧ (prefix i σ ⊨ f) ∧ (∀ ia<i. (prefix ia σ ⊨ ¬f)))
   )
  )
by (simp add: DiLensem PrefixFstsem)

lemma DiLenFstAndsem:
 (σ ⊨ di ((▷f ∧ g) ∧ len(i))) =
   ( i≤intlen σ ∧
   (
    ( (prefix i σ ⊨ f ∧ g) ∧ i = 0) ∨
    ( i>0 ∧ (prefix i σ ⊨ f ∧ g) ∧ (∀ ia<i. (prefix ia σ ⊨ ¬f)))
   )
  )
using DiLensem PrefixFstAndsem by metis

lemma FstLenSamesem:
 ( ( i≤intlen σ ∧
   (
    ( (prefix i σ ⊨ f) ∧ i = 0) ∨
    ( i>0 ∧ (prefix i σ ⊨ f) ∧ (∀ ia<i. (prefix ia σ ⊨ ¬f)))
   )
  ) ∧
  ( j≤intlen σ ∧
   (
    ( (prefix j σ ⊨ f) ∧ j = 0) ∨
    ( j>0 ∧ (prefix j σ ⊨ f) ∧ (∀ ia<j. (prefix ia σ ⊨ ¬f)))
   )
  )
) ⟶ (i=j)

by (metis not-less-iff-gr-or-eq unl-lift)
```

6.4 Theorems

6.4.1 Fixed length intervals

```
lemma LenZeroEqvEmpty:
 ⊢ len(0) = empty
by simp
```

```
lemma LenOneEqvSkip:
 ⊢ len(1) = skip
by (simp add: len-d-def ChopEmpty)
```

```
lemma LenNPlusOneA:
 ⊢  len(n+1) = skip;len(n)
by simp
```

```
lemma LenEqvLenChopLen:
 ⊢ len(i+j) = len(i);len(j)
proof
 (induct i)
 case 0
 then show ?case
 by (metis EmptyChop comm-monoid-add-class.add-0 int-eq len-d.simps(1))
 next
 case (Suc i)
 then show ?case
 by (metis ChopAssoc add-Suc inteq-reflection len-d.simps(2))
qed
```

```
lemma LenNPlusOneB:
 ⊢ len(n+1) = len(n);skip
proof −
 have  1: ⊢ len(n+1) = len(n);len(1) by (rule LenEqvLenChopLen)
 have  2: ⊢ len(1) = skip by (rule LenOneEqvSkip)
 hence 3: ⊢ len(n);len(1) = len(n);skip using RightChopEqvChop by blast
 from 1 3 show ?thesis by fastforce
qed
```

```
lemma LenCommute:
 ⊢ (skip;(len n)) = (len n);skip
proof
  (induct n)
  case 0
  then show ?case using EmptyChop ChopEmpty len-0 by (metis int-eq)
  next
  case (Suc n)
  then show ?case using ChopAssoc len-Suc by (metis inteq-reflection)
qed
```

```
lemma SkipTrueEqvTrueSkip:
 ⊢ skip;#True = #True;skip
```

```
using TrueChopSkipEqvSkipChopTrue by fastforce
```

```
lemma PowerCommute:
 ⊢ (f;(power f n)) = ((power f n);f)
proof
  (induct n)
  case 0
  then show ?case using EmptyChop ChopEmpty pow-0 by (metis int-eq)
  next
  case (Suc n)
  then show ?case using ChopAssoc pow-Suc by (metis inteq-reflection)
qed
```

```
lemma PowerRev:
 ⊢ (power skip n)^r = (power skip n)
proof
 (induct n)
 case 0
 then show ?case using REmptyEqvEmpty by auto
 next
 case (Suc n)
 then show ?case using PowerCommute RevChop pow-Suc by (metis RevSkip int-eq)
qed
```

```
lemma RLenEqvLen:
 ⊢ (len k)^r = (len k)
proof
  (induct k)
  case 0
  then show ?case using REmptyEqvEmpty by auto
  next
  case (Suc k)
  then show ?case using LenCommute RevChop len-Suc by (metis RevSkip int-eq)
qed
```

```
lemma PowerSkipEqvLen:
 ⊢ (power skip n) = (len n)
proof
  (induct n)
  case 0
  then show ?case by auto
  next
  case (Suc n)
  then show ?case by (metis LenEqvLenChopLen Suc-eq-plus1 int-eq len-Suc pow-Suc)
qed
```

```
lemma ExistsLen:
 ⊢ ∃ k. len(k)
by (simp add: len-defs Valid-def)
```

```
lemma AndExistsLen:
 ⊢ f = (f ∧ (∃ k. len(k)))
using ExistsLen by fastforce

lemma AndExistsLenChop:
 ⊢ (f;g) =  (∃ k. (f ∧ len(k));g)
by (simp add: Valid-def len-defs chop-defs)

lemma AndExistsLenChopR:
 ⊢ (f;g) = (∃ k. f;(g ∧ len(k)))
by (simp add: Valid-def len-defs chop-defs)

lemma LFixedAndDistr:
 ⊢ ((f0 ∧ len(k));g0 ∧ (f1 ∧ len(k));g1) = ((f0 ∧ f1) ∧ len(k));(g0 ∧ g1)
apply (simp add: Valid-def len-defs chop-defs interval-prefix-length interval-suffix-length)
by blast

lemma RFixedAndDistr:
 ⊢ (f0;(g0 ∧ len(k)) ∧ f1;(g1 ∧ len(k))) = (f0 ∧ f1);((g0 ∧ g1) ∧ len(k))
apply (simp add: Valid-def len-defs chop-defs interval-prefix-length interval-suffix-length)
by (metis diff-diff-cancel)

lemma LFixedAndDistrA:
 ⊢ ((f0 ∧ len(k));g0 ∧ (f1 ∧ len(k));g0) = ((f0 ∧ f1) ∧ len(k));g0
proof −
 have 1: ⊢ ((f0 ∧ len(k));g0 ∧ (f1 ∧ len(k));g0) = ((f0 ∧ f1) ∧ len(k));(g0 ∧ g0)
     by (rule LFixedAndDistr)
 have 2: ⊢ ((f0 ∧ f1) ∧ len(k));(g0 ∧ g0) = ((f0 ∧ f1) ∧ len(k));g0
     by auto
 from 1 2 show ?thesis by fastforce
qed

lemma LFixedAndDistrB:
 ⊢ ((f0 ∧ len(k));g0 ∧ (f0 ∧ len(k));g1) = (f0 ∧ len(k));(g0 ∧ g1)
proof −
 have 1: ⊢ ((f0 ∧ len(k));g0 ∧ (f0 ∧ len(k));g1) = ((f0 ∧ f0) ∧ len(k));(g0 ∧ g1)
     by (rule LFixedAndDistr)
 have 2: ⊢ ((f0 ∧ f0) ∧ len(k));(g0 ∧ g1) = (f0 ∧ len(k));(g0 ∧ g1)
    by auto
 from 1 2 show ?thesis by fastforce
qed

lemma LFixedAndDistrB1:
 ⊢ (len(k);f ∧ len(k);g) = len(k);(f ∧ g)
proof −
 have 1: ⊢ len(k);f = (#True ∧ len(k));f
     by auto
 have 2: ⊢ len(k);g = (#True ∧ len(k));g
     by auto
 have 3: ⊢ (len(k);f ∧ len(k);g) = ((#True ∧ len(k));f ∧ (#True ∧ len(k));g)
```

```
     using 1 2 by auto
 have 4: ⊢ ((#True ∧ len(k));f ∧ (#True ∧ len(k));g) = (#True ∧ len(k));(f ∧ g)
     using LFixedAndDistrB by blast
 have 5: ⊢ (#True ∧ len(k));(f ∧ g) = (len(k));(f ∧ g)
     by auto
 from 1 2 3 4 5 show ?thesis by auto
qed

lemma RFixedAndDistrA:
⊢ (f0;(g0 ∧ len(k)) ∧ f0;(g1 ∧ len(k))) = f0;((g0 ∧ g1) ∧ len(k))
proof −
 have 1: ⊢ (f0;(g0 ∧ len(k)) ∧ f0;(g1 ∧ len(k))) = (f0 ∧ f0);((g0 ∧ g1) ∧ len(k))
     by (rule RFixedAndDistr)
 have 2: ⊢ (f0 ∧ f0);((g0 ∧ g1) ∧ len(k)) = f0;((g0 ∧ g1) ∧ len(k))
     by auto
from 1 2 show ?thesis by fastforce
qed

lemma RFixedAndDistrB:
⊢ (f0;(g0 ∧ len(k)) ∧ f1;(g0 ∧ len(k))) = (f0 ∧ f1);(g0 ∧ len(k))
proof −
 have 1: ⊢ (f0;(g0 ∧ len(k)) ∧ f1;(g0 ∧ len(k))) = (f0 ∧ f1);((g0 ∧ g0) ∧ len(k))
     by (rule RFixedAndDistr)
 have 2: ⊢ (f0 ∧ f1);((g0 ∧ g0) ∧ len(k)) = (f0 ∧ f1);(g0 ∧ len(k))
     by auto
from 1 2 show ?thesis by fastforce
qed

lemma ChopSkipAndChopSkip:
⊢ (f0;skip ∧ f1;skip) = (f0 ∧ f1);skip
proof −
 have 1: ⊢ (f0;(#True ∧ len(1)) ∧ f1;(#True ∧ len(1))) = (f0 ∧ f1);(#True ∧ len(1))
     by (rule RFixedAndDistrB)
 have 2: ⊢ (#True ∧ len(1)) = skip
     using LenOneEqvSkip by fastforce
 hence 3: ⊢ f0;(#True ∧ len(1)) = f0;skip
     using RightChopEqvChop by blast
 have 4: ⊢ f1;(#True ∧ len(1)) = f1;skip
     using 2 RightChopEqvChop by blast
 have 5: ⊢ (f0;(#True ∧ len(1)) ∧ f1;(#True ∧ len(1))) = (f0;skip ∧ f1;skip)
     using 3 4 by fastforce
 have 6: ⊢ (f0 ∧ f1);(#True ∧ len(1)) = (f0 ∧ f1);skip
     using 2 RightChopEqvChop by blast
 from 1 5 6 show ?thesis by fastforce
qed

lemma BiAndChopSkipEqv:
⊢ (bi (f ∧ g));skip = ((bi f);skip ∧ (bi g);skip)
proof −
 have 1: ⊢ bi (f ∧ g) = ((bi f) ∧ (bi g))
```

```
     by (simp add: bi-defs Valid-def , auto)
 hence 2: ⊢ (bi (f ∧ g));skip = (bi f ∧ bi g);skip
     by (rule LeftChopEqvChop)
 have 3: ⊢ (bi f ∧ bi g);skip = ((bi f );skip ∧ (bi g);skip)
     using ChopSkipAndChopSkip  by fastforce
 from 2 3 show ?thesis by fastforce
qed

lemma DiAndChopSkipEqv:
⊢ (di (f ∧ g));skip ⟶ (di f );skip ∧ (di g);skip
proof −
 have 1: ⊢ di (f ∧ g) ⟶ (di f ) ∧ (di g)
     by (simp add: DiAndImpAnd)
 hence 2: ⊢ (di (f ∧ g));skip ⟶ (di f ∧ di g);skip
     by (rule LeftChopImpChop)
 have 3: ⊢ (di f ∧ di g);skip = ((di f );skip ∧ (di g);skip)
     using ChopSkipAndChopSkip by fastforce
 from 2 3 show ?thesis by fastforce
qed

lemma ChopEmptyAndEmpty:
⊢ (f ;g ∧ empty) = (f ∧ g ∧ empty)
apply (simp add: Valid-def chop-defs empty-defs)
by (metis interval-prefix-intlen interval-suffix-zero le-zero-eq)

lemma ChopSkipImpMore:
⊢ f ;skip ⟶ more
using ChopImpDiamond MoreEqvSkipChopTrue SkipTrueEqvTrueSkip TrueChopEqvDiamond by fastforce

lemma MoreEqvMoreChopTrue:
⊢ more = more;#True
proof −
 have 1: ⊢ more = skip;#True
     using MoreEqvSkipChopTrue by blast
 have 2: ⊢ #True = #True;#True
     by (simp add: Valid-def chop-defs, auto)
 hence 3: ⊢ skip;#True = skip;(#True;#True)
     using RightChopEqvChop by blast
 have 4: ⊢ skip;(#True;#True) = (skip;#True);#True
     using ChopAssoc by blast
 have 5: ⊢ (skip;#True);#True = more;#True
     using MoreEqvSkipChopTrue by (simp add: more-d-def next-d-def )
 from 1 3 4 5 show ?thesis by fastforce
qed


lemma NotNotChopSkip:
⊢ (¬((¬ f ) ;skip)) = (empty ∨ (f ;skip))
by (metis WprevEqvEmptyOrPrev prev-d-def wprev-d-def )
```

lemma *NotChopFixed*:
$\vdash (\neg(f;(g \wedge len(k)))) = (\neg(\Diamond(g \wedge len(k))) \vee ((\neg f);(g \wedge len(k))))$
apply (*simp add*: *len-defs Valid-def sometimes-defs chop-defs interval-suffix-length*)
by (*smt diff-diff-cancel*)

lemma *NotFixedChop*:
$\vdash (\neg((g \wedge len(k));f)) = (\neg(di(g \wedge len(k))) \vee ((g \wedge len(k));(\neg f)))$
by (*simp add*: *len-defs Valid-def di-defs chop-defs interval-prefix-length*, *auto*)

lemma *NotChopNotSkip*:
$\vdash (\neg(f;skip)) = (empty \vee ((\neg f);skip))$
proof –
have *1*: $\vdash (\neg((\neg(\neg f));skip)) = (empty \vee ((\neg f);skip))$ **using** *NotNotChopSkip* **by** *blast*
have *2*: $\vdash (\neg((\neg(\neg f));skip)) = (\neg(f;skip))$ **by** *auto*
from *1 2* **show** *?thesis* **by** *auto*
qed

6.4.2 Additional ITL theorems

lemma *BiOrBiImpBiOr*:
$\vdash bi\ f \vee bi\ g \longrightarrow bi(f \vee g)$
proof –
have *1*: $\vdash f \longrightarrow f \vee g$ **by** *auto*
hence *2*: $\vdash bi\ f \longrightarrow bi(f \vee g)$ **by** (*rule BiImpBiRule*)
have *3*: $\vdash g \longrightarrow f \vee g$ **by** *auto*
hence *4*: $\vdash bi\ g \longrightarrow bi(f \vee g)$ **by** (*rule BiImpBiRule*)
from *2 4* **show** *?thesis* **by** *fastforce*
qed

lemma *MoreAndBiImpBiChopSkip*:
$\vdash more \wedge bi\ f \longrightarrow (bi\ f);skip$
proof –
have *1*: $\vdash (bi\ f);skip = ((\neg(di\ (\neg f)));skip)$ **by** (*simp add*: *bi-d-def*)
have *2*: $\vdash (\neg(\ (\neg(di\ (\neg f)));skip)) = (empty \vee (di\ (\neg f));skip)$ **by** (*rule NotNotChopSkip*)
have *3*: $\vdash empty \longrightarrow empty \vee di\ (\neg f)$ **by** *auto*
have *4*: $\vdash (di\ (\neg f));skip \longrightarrow di\ (\neg f)$ **using** *ChopImpDi DiEqvDiDi* **by** *fastforce*
hence *5*: $\vdash (di\ (\neg f));skip \longrightarrow empty \vee di\ (\neg f)$ **by** (*rule Prop05*)
have *6*: $\vdash \neg(\ (\neg(di\ (\neg f)));skip) \longrightarrow empty \vee di\ (\neg f)$ **using** *2 3 5* **by** *fastforce*
hence *7*: $\vdash \neg(empty \vee di\ (\neg f)) \longrightarrow \neg(\neg(\ (\neg(di\ (\neg f)));skip))$ **by** *fastforce*
have *8*: $\vdash (\neg(\ \neg(\ (\neg(di\ (\neg f)));skip))) = ((\neg(di\ (\neg f)));skip)$ **by** *auto*
have *9*: $\vdash (\neg(empty \vee di\ (\neg f))) = (more \wedge \neg(\ di\ (\neg f)))$
using *NotAndMoreEqvEmptyOr* **by** *fastforce*
have *10*: $\vdash (more \wedge \neg(\ di\ (\neg f))) = (more \wedge bi\ f)$ **by** (*simp add*: *bi-d-def*)
from *1 6 7 8 9 10* **show** *?thesis* **by** (*metis int-eq*)
qed

lemma *DiChopImpDiB*:
$\vdash di(f;g) \longrightarrow di\ f$
proof –
have *1*: $\vdash f\ ;\ (g;\#True) \longrightarrow di\ f$ **by** (*rule ChopImpDi*)

```
 have 2: ⊢ f ; (g;#True) = (f;g);#True  by (rule ChopAssoc)
 from 1 2 show ?thesis by (metis di-d-def int-eq)
qed

lemma BiBiOrImpBi:
⊢ bi ( bi f ∨ bi g) ⟶ bi f ∨ bi g
using BiElim by auto

lemma BiImpBiBiOr:
⊢ bi f  ⟶ bi ( bi f ∨ bi g)
proof −
 have 1: ⊢ bi f ⟶ bi f ∨ bi g by auto
 hence 2: ⊢ bi (bi f) ⟶ bi(bi f ∨ bi g)  using BiImpBiRule by blast
 have 3: ⊢ bi (bi f) = bi f using BiEqvBiBi by fastforce
 from 2 3 show ?thesis by fastforce
qed

lemma BiImpBiBiOrB:
⊢ bi g  ⟶ bi ( bi f ∨ bi g)
proof −
 have 1: ⊢ bi g ⟶ bi f ∨ bi g by auto
 hence 2: ⊢ bi (bi g) ⟶ bi(bi f ∨ bi g)  using BiImpBiRule by blast
 have 3: ⊢ bi (bi g) = bi g using BiEqvBiBi by fastforce
 from 2 3 show ?thesis by fastforce
qed

lemma BiBiOrEqvBi:
⊢ bi ( bi f ∨ bi g) = bi f ∨ bi g
proof −
 have 1: ⊢ bi ( bi f ∨ bi g) ⟶ bi f ∨ bi g  by (rule BiBiOrImpBi)
 have 2: ⊢ bi f  ⟶ bi ( bi f ∨ bi g)  by (rule BiImpBiBiOr)
 have 3: ⊢ bi g  ⟶ bi ( bi f ∨ bi g)  by (rule BiImpBiBiOrB)
 have 4: ⊢ bi f  ∨ bi g ⟶ bi ( bi f ∨ bi g)  using 2 3 by fastforce
 from 1 4 show ?thesis by fastforce
qed

lemma DiEqvOrDiChopSkipA:
⊢ di f = (f ∨ di(f;skip))
proof −
 have 1: ⊢ di f = f ;#True  by (simp add: di-d-def)
 hence 2: ⊢ di f = f; ( empty ∨ more) by (simp add: empty-d-def)
 hence 3: ⊢f; ( empty ∨ more) = (f;empty ∨ f;more)  using ChopOrEqv by blast
 have 4: ⊢ f;empty = f by (rule ChopEmpty)
 have 5: ⊢ more = skip;#True using MoreEqvSkipChopTrue by blast
 hence 6: ⊢ f;more = f;(skip;#True) using RightChopEqvChop by blast
 have 7: ⊢ f;(skip;#True) = (f;skip);#True by (rule ChopAssoc)
 from 2 3 4 6 7 show ?thesis by (metis di-d-def int-eq)
qed

lemma DiEqvOrDiChopSkipB:
```

```
⊢ di f = (f ∨ (di f);skip)
proof −
 have 1: ⊢ (di f) = (f ∨ di(f;skip)) by (rule DiEqvOrDiChopSkipA)
 have 2: ⊢  di(f;skip) = (f;skip);#True  by (simp add: di-d-def)
 have 3: ⊢ (f;skip);#True = f;(skip;#True) by (rule ChopAssocB)
 have 4: ⊢ di(f;skip) = f;(skip;#True)  using 2 3 by fastforce
 have 5: ⊢ skip;#True = #True;skip by (rule SkipTrueEqvTrueSkip)
 hence 6: ⊢ f;(skip;#True) = f;(#True;skip) using RightChopEqvChop by blast
 have 7: ⊢ di(f;skip) = f;(#True;skip) using 4 6 by fastforce
 have 8: ⊢ f;(#True;skip) = (f;#True);skip by (rule ChopAssoc)
 have 9: ⊢ (f;#True);skip = (di f);skip by (simp add: di-d-def)
 have 10 : ⊢ di(f;skip) =(di f);skip  using 7 8 9 by fastforce
 hence 11: ⊢ (f ∨ di(f;skip)) = (f ∨ (di f);skip)  by auto
 from 1 11 show ?thesis by fastforce
qed

lemma BiEqvAndEmptyOrBiChopSkip:
⊢ bi f = (f ∧ (empty ∨ (bi f);skip))
proof −
 have 1: ⊢ di (¬ f) = (¬ f ∨ (di (¬ f);skip)) by (rule DiEqvOrDiChopSkipB)
 have 2: ⊢ di (¬ f) = (¬( bi f )) by (rule DiNotEqvNotBi)
 have 3: ⊢ (¬( bi f )) =  (¬ f ∨ (di (¬ f);skip))  using 1 2 by fastforce
 hence 4: ⊢ bi f = (¬(¬ f ∨ (di (¬ f);skip)))  by auto
 have 5: ⊢ (¬(¬ f ∨ (di (¬ f);skip))) = (f ∧ ¬(di (¬ f);skip))  by auto
 have 6: ⊢ di (¬ f);skip = ((¬(bi f));skip)  by (simp add: 2 LeftChopEqvChop)
 hence 7: ⊢ (¬(di (¬ f);skip)) = (¬((¬(bi f));skip))  by auto
 have 8: ⊢ (¬((¬(bi f));skip)) = (empty ∨ (bi f);skip)  using NotNotChopSkip by blast
 hence 9: ⊢ (f ∧ ¬(di (¬ f);skip)) = (f ∧ (empty ∨ (bi f);skip))  using 7 8 by fastforce
 from 4 5 9 show ?thesis by fastforce
qed

lemma DiDiAndEqvDi:
⊢ di ( di f ∧ di g) = (di f ∧ di g)
proof −
 have 1: ⊢ bi ( bi (¬ f) ∨ bi (¬ g)) = (bi (¬f) ∨ bi (¬ g))
     by (meson BiBiOrImpBi BiImpBiBiOr BiImpBiBiOrB Prop02 int-iffI)
 have 2: ⊢ bi (¬f) = (¬ (di f))
     by (simp add: bi-d-def)
 have 3: ⊢ bi (¬g) = (¬ (di g))
     by (simp add: bi-d-def)
 have 4: ⊢ (bi (¬f) ∨ bi (¬ g)) = (¬ (di f) ∨ ¬ (di g))
     using 2 3 by fastforce
 have 5: ⊢ (¬ (di f) ∨ ¬ (di g)) = (¬(di f ∧ di g))
     by auto
 have 6: ⊢ bi ( bi (¬ f) ∨ bi (¬ g)) = (¬(di f ∧ di g))
     using 1 5 4 by fastforce
 hence 7: ⊢ (¬(bi ( bi (¬ f) ∨ bi (¬ g)))) = (di f ∧ di g)
     by auto
 have 8 : ⊢ (¬(bi ( bi (¬ f) ∨ bi (¬ g)))) = di ( ¬(bi (¬ f) ∨ bi (¬ g)))
     using DiNotEqvNotBi  by fastforce
```

have *9* : $\vdash (\neg(bi\ (\neg\ f) \vee bi\ (\neg\ g))) = (di\ f \wedge di\ g)$
using *1 7* **by** *fastforce*
hence *10*: $\vdash di\ (\ \neg(bi\ (\neg\ f) \vee bi\ (\neg\ g))) = di\ (\ di\ f \wedge di\ g)$
using *DiEqvDi* **by** *blast*
from *7 8 10* **show** *?thesis* **by** *fastforce*
qed

lemma *BiInduct*:
$\vdash bi(f \longrightarrow wprev\ f) \wedge f \longrightarrow bi\ f$
proof –
have *1*: $\vdash \Box((f^r) \longrightarrow wnext(f^r)) \wedge f^r \longrightarrow \Box(f^r)$ **using** *BoxInduct* **by** *blast*
hence *2*: $\vdash (\Box((f^r) \longrightarrow wnext(f^r)) \wedge f^r \longrightarrow \Box(f^r))^r$ **using** *ReverseEqv* **by** *blast*
have *3*: $\vdash ((f^r)^r\ ) = f$ **by** (*simp add*: *EqvReverseReverse*)
have *4*: $\vdash (\Box\ (f^r))^r = bi\ (f)$ **using** *RRBoxEqvBi* **by** *blast*
have *5*: $\vdash ((f^r) \longrightarrow wnext(f^r))^r = (\ (f^r)^r \longrightarrow (wnext(f^r))^r\ )$ **by** (*simp add*: *rev-fun2*)
have *6*: $\vdash (wnext(f^r))^r = wprev(f)$ **using** *RRWNextEqvWPrev* **by** *blast*
have *7*: $\vdash (\ (f^r)^r \longrightarrow (wnext(f^r))^r\ ) = (\ f \longrightarrow wprev(f)\ )$ **using** *6 3* **by** *fastforce*
have *8*: $\vdash bi(\ (f^r)^r \longrightarrow (wnext(f^r))^r\ ) = bi(\ f \longrightarrow wprev(f)\ )$ **using** *7 3 BiEqvBi* **by** *blast*
have *9*: $\vdash (\Box((f^r) \longrightarrow wnext(f^r)))^r = bi\ (\ ((f^r) \longrightarrow wnext(f^r))^r\ )$ **using** *RBoxEqvBi* **by** *blast*
have *10*: $\vdash (\Box((f^r) \longrightarrow wnext(f^r)))^r = bi(\ f \longrightarrow wprev(f)\ )$ **using** *8 9 5 int-eq* **by** *fastforce*
have *11*: $\vdash (\Box((f^r) \longrightarrow wnext(f^r)) \wedge f^r \longrightarrow \Box(f^r))^r =$
$(((\Box((f^r) \longrightarrow wnext(f^r)))^r \wedge (f^r)^r \longrightarrow (\Box(f^r))^r))$ **by** (*metis int-eq rev-fun2*)
have *12*: $\vdash ((\Box((f^r) \longrightarrow wnext(f^r)))^r \wedge (f^r)^r \longrightarrow (\Box(f^r))^r) =$
$(bi(\ f \longrightarrow wprev(f)\ ) \wedge f \longrightarrow bi\ f\ )$ **using** *8 3 4 10* **by** *fastforce*
from *2 11 12* **show** *?thesis* **using** *MP* **by** *fastforce*
qed

lemma *PrevLoop*:
assumes $\vdash f \longrightarrow prev\ f$
shows $\vdash \neg\ f$
proof –
have *1*: $\vdash f \longrightarrow prev\ f$ **using** *assms* **by** *auto*
hence *2*: $\vdash f \longrightarrow (\ more \wedge wprev\ f)$
by (*smt intI int-eq more-defs prev-defs Prop10 unl-lift2 wprev-defs*)
hence *3*: $\vdash f \longrightarrow wprev\ f$ **by** *auto*
hence *4*: $\vdash bi(f \longrightarrow wprev\ f)$ **by** (*rule BiGen*)
have *5*: $\vdash bi(f \longrightarrow wprev\ f) \wedge f \longrightarrow bi\ f$ **by** (*rule BiInduct*)
hence *6*: $\vdash bi(f \longrightarrow wprev\ f) \longrightarrow (f \longrightarrow bi\ f)$ **by** *fastforce*
have *7*: $\vdash (f \longrightarrow bi\ f)$ **using** *4 6 MP* **by** *blast*
have *8*: $\vdash bi\ f \longrightarrow f$ **by** (*rule BiElim*)
have *9*: $\vdash f = bi\ f$ **using** *7 8* **by** *fastforce*
have *10*: $\vdash f \longrightarrow more$ **using** *2* **by** *auto*
hence *11*: $\vdash bi\ f \longrightarrow\ bi\ more$ **using** *BiImpBiRule* **by** *blast*
have *12*: $\vdash \neg(bi\ more)$ **using** *DiEmpty bi-d-def empty-d-def* **by** (*simp add*: *bi-d-def empty-d-def*)
from *7 9 11 12* **show** *?thesis* **using** *MP* **by** *fastforce*
qed

lemma *PrevImpNotPrevNot*:
$\vdash prev\ f \longrightarrow \neg(prev\ (\neg\ f))$
by (*metis* (*no-types*, *lifting*) *NextImpNotNextNot RPrevEqvNext ReverseEqv inteq-reflection*

```
   rev-fun1 rev-fun2)

lemma BiEqvAndWprevBi:
⊢ bi f = (f ∧ wprev(bi f))
using BoxEqvAndWnextBox
by (metis (no-types, lifting) RBiEqvBox RRAnd RRBoxEqvBi RWPrevEqvWNext int-eq)


lemma DiIntroLoop:
 assumes ⊢ (f ∧ ¬ g) ⟶ prev f
 shows ⊢ f ⟶ di g
using assms DiamondIntro
by (metis (no-types, lifting) RDiEqvDiamond RPrevEqvNext ReverseEqv inteq-reflection
   rev-fun2 rev-fun1)


lemma DiEqvOrChopMore:
⊢ di f = (f ∨ f;more)
proof −
 have 1: ⊢ di f = f;#True by (simp add: di-d-def)
 hence 2: ⊢ di f = f; (empty ∨ more)  by (simp add: empty-d-def)
 have  3: ⊢ f; (empty ∨ more) = (f;empty ∨ f;more)  by (simp add: ChopOrEqv)
 have  4: ⊢ f;empty = f  by (rule ChopEmpty)
 from 2 3 4 show ?thesis by fastforce
qed


lemma DiAndDiEqvDiAndDiOrDiAndDi:
⊢ (di f ∧ di g) = (di(f ∧ di g) ∨ di(g ∧ di f))
proof −
 have 1: ⊢ di f  = (f ∨ f;more)
     using DiEqvOrChopMore by blast
 have 2: ⊢ di g  = (g ∨ g;more)
     using DiEqvOrChopMore by blast
 have 3: ⊢ (di f ∧ di g) = ((f ∨ f;more) ∧ (g ∨ g;more))
     using 1 2 by fastforce
 have 4: ⊢ ((f ∨ f;more) ∧ (g ∨ g;more)) =
          ((f ∧ g) ∨ ( f ∧ g;more) ∨ (g ∧ f;more) ∨ (f;more ∧ g;more ))
     by auto
 have 5: ⊢ more = #True;skip
     using MoreEqvSkipChopTrue SkipTrueEqvTrueSkip by fastforce
 hence 6: ⊢ f;more = f;(#True;skip)
     using RightChopEqvChop by blast
 have  7: ⊢ f;(#True;skip) = (f;#True);skip
     by (rule ChopAssoc)
 have  8: ⊢ f;more = prev (di f)
     using 6 7 by (metis di-d-def int-eq prev-d-def)
 have  9: ⊢ g;more = g;(#True;skip)
     using 5  RightChopEqvChop by blast
 have 10: ⊢ g;(#True;skip) = (g;#True);skip
```

```
    by (rule ChopAssoc)
have 11: ⊢ g;more = prev (di g)
    using 9 10 by (metis di-d-def int-eq prev-d-def)
have 12: ⊢ (f;more ∧ g;more) = (prev (di f) ∧ prev (di g))
    using 8 11 by fastforce
hence 13: ⊢ (f;more ∧ g;more) = prev (di f ∧ di g)
    by (metis ChopSkipAndChopSkip int-eq prev-d-def)
have 14: ⊢ (di f ∧ di g) =
        ((f ∧ g) ∨ ( f ∧ g;more) ∨ (g ∧ f;more)) ∨ (f;more ∧ g;more)
    using 3 4 by auto
have 15: ⊢ (di f ∧ di g) =
        ((f ∧ g) ∨ ( f ∧ g;more) ∨ (g ∧ f;more)) ∨ prev (di f ∧ di g)
    using 13 14 by fastforce
hence 16: ⊢ (di f ∧ di g) ⟶
        ((f ∧ g) ∨ ( f ∧ g;more) ∨ (g ∧ f;more)) ∨ prev (di f ∧ di g)
    by fastforce
hence 17: ⊢ (di f ∧ di g) ∧ ¬((f ∧ g) ∨ ( f ∧ g;more) ∨ (g ∧ f;more)) ⟶
          prev (di f ∧ di g)
    by fastforce
hence 18: ⊢ (di f ∧ di g) ⟶ di((f ∧ g) ∨ ( f ∧ g;more) ∨ (g ∧ f;more))
    using DiIntroLoop by blast
have  19: ⊢ di((f ∧ g) ∨ ( f ∧ g;more) ∨ (g ∧ f;more)) =
         (di(f ∧ g) ∨ di( f ∧ g;more) ∨ di(g ∧ f;more))
    by (meson DiOrEqv Prop06)
have  20: ⊢ f   ⟶ di f
    using DiIntro by blast
hence 21: ⊢ f ∧ g ⟶ g ∧ di f
    by auto
hence 22: ⊢ di(f ∧ g) ⟶ di(g ∧ di f)
    using DiImpDi by blast
hence 23: ⊢ di(f ∧ g) ⟶ di(g ∧ di f) ∨ di(f ∧ di g)
    by auto
have  24: ⊢ g;more ⟶ di g
    by (simp add: ChopImpDi)
hence 25: ⊢ f ∧ g;more ⟶ f ∧ di g
    by auto
hence 26: ⊢ di(f ∧ g;more) ⟶ di(f ∧ di g)
    using DiImpDi by blast
hence 27: ⊢ di(f ∧ g;more) ⟶    di(f ∧ di g) ∨ di(g ∧ di f)
    by auto
have  28: ⊢ f;more ⟶ di f
    by (simp add: ChopImpDi)
hence 29: ⊢ g ∧ f;more ⟶ g ∧ di f
    by auto
hence 30: ⊢ di(g ∧ f;more) ⟶ di(g ∧ di f)
    using DiImpDi by blast
hence 31: ⊢ di(g ∧ f;more) ⟶  di(f ∧ di g) ∨  di(g ∧ di f)
    by auto
have  32: ⊢  di(f ∧ g) ∨ di( f ∧ g;more) ∨ di(g ∧ f;more) ⟶
          di(f ∧ di g) ∨  di(g ∧ di f)
    using 23 27 31 by fastforce
  have 33: ⊢ di((f ∧ g) ∨ ( f ∧ g;more) ∨ (g ∧ f;more)) ⟶
           di(f ∧ di g) ∨  di(g ∧ di f)
    using 19 32 by fastforce
  have 34: ⊢ (di f ∧ di g) ⟶ di(f ∧ di g) ∨  di(g ∧ di f)
    using 18 33 by fastforce
  have 35: ⊢  f  ⟶ di f
    using DiIntro by blast
  hence 36: ⊢ f ∧ di g ⟶ di f ∧ di g
    by auto
  hence 37: ⊢ di (f ∧ di g) ⟶ di (di f ∧ di g)
    using DiImpDi by blast
  have 38: ⊢ di (di f ∧ di g) = (di f ∧ di g)
    using DiDiAndEqvDi by blast
  have 39: ⊢ di (f ∧ di g) ⟶ di f ∧ di g
    using 37 38 by fastforce
  have 40: ⊢  g  ⟶ di g
    using DiIntro by blast
  hence 41: ⊢   g ∧ di f ⟶ di f ∧ di g
    by auto
  hence 42: ⊢ di (g ∧ di f) ⟶ di (di f ∧ di g)
    using DiImpDi by blast
  have 43: ⊢ di (di f ∧ di g) = (di f ∧ di g)
    using DiDiAndEqvDi by fastforce
  have 44: ⊢ di (g ∧ di f) ⟶ di f ∧ di g
    using 42 43 by fastforce
  have 45: ⊢ di (f ∧ di g) ∨ di (g ∧ di f) ⟶ di f ∧ di g
    using 39 44  by fastforce
  from 34 45 show ?thesis by fastforce
qed

lemma BoxStateEqvBiFinState:
⊢ □ (init w) = bi (fin (init w))
proof −
  have 1: ⊢ ◇ (¬ (init w)) = #True ; (¬(init w))
    by (simp add: sometimes-d-def)
  have 2: ⊢ ◇ (init(¬ w)) = #True ; init (¬ w)
    by (simp add: sometimes-d-def)
  have 3: ⊢ di (#True ∧ fin (init (¬ w))) = #True ; init (¬ w)
    using DiAndFinEqvChopState by blast
  have 4: ⊢ ◇ (init(¬ w)) = di (#True ∧ fin (init (¬ w)))
    using 1 2 3 by fastforce
  have 5: ⊢ (¬ (◇ (init(¬ w)))) = (¬ (di (#True ∧ fin (init (¬ w)))))
    using 4 by fastforce
  have 6: ⊢ □ (init w) = (¬ (di (#True ∧ fin (init (¬ w)))))
    using 5 always-d-def Initprop(2) by (metis int-eq)
  have 7: ⊢ □ (init w) = bi (¬ (fin (init (¬ w))))
    using 6 by (simp add: bi-d-def)
  have 8: ⊢ init (¬ w) = (¬ (init w))
    using Initprop(2) by fastforce
  have 9: ⊢ fin (init (¬ w)) = fin (¬ (init w))
      using 8 FinEqvFin by blast
  have 10: ⊢ fin (init (¬ w)) = (¬ (fin (init w)))
      using 8 FinNotStateEqvNotFinState FinEqvFin by blast
  have 11: ⊢ (¬ (fin (init (¬ w)))) = (fin (init w))
      using 10  by fastforce
  have 12: ⊢ bi (¬ (fin (init (¬ w)))) = bi (fin (init w))
      using 11  by (simp add: BiEqvBi)
  have 13: ⊢ □ (init w) = bi (fin (init w))
      using 7 12 by fastforce
  from 13 show ?thesis by simp
qed

lemma DiamondStateEqvDiFinState:
 ⊢ ◇ (init w) = di (fin (init w))
proof −
  have 1: ⊢ □ (init (¬ w)) = bi (fin (init (¬ w)))
      using BoxStateEqvBiFinState by blast
  have 2: ⊢ (¬ (□ (init (¬ w)))) = (¬ (bi (fin (init (¬ w)))))
      using 1 by auto
  have 3: ⊢ ◇ (¬ (init (¬ w))) = di (¬ (fin (init (¬ w))))
      using 2 by (simp add: always-d-def bi-d-def)
  have 4: ⊢ ◇ (init  w) = di (¬ (fin (init (¬ w))))
      by (metis 3 DiEqvNotBiNot DiState Initprop(2) StateEqvBi int-eq)
  have 5: ⊢ ◇ (init  w) = di (fin (init w)) using 4 FinNotStateEqvNotFinState
      by (metis DiEqvNotBiNot DiNotEqvNotBi inteq-reflection)
  from 1 2 3 4 5 show ?thesis by simp
qed

lemma OrDiEqvDi:
 ⊢ (f ∨ di f) = di f
proof −
  have 1: ⊢ f ⟶ di f using DiIntro by blast
 from 1 show ?thesis by auto
qed

lemma AndDiEqv:
 ⊢ (f ∧ di f) = f
proof −
  have 1: ⊢ f ⟶ di f using DiIntro by blast
 from 1 show ?thesis by auto
qed

lemma BiEmptyEqvEmpty:
 ⊢ bi empty = empty
proof −
 have 1: ⊢ bi empty = (¬ (di (¬ empty))) by (simp add: bi-d-def)
 have 2: ⊢ (¬ (di (¬ empty))) = (¬ ((¬ empty);#True)) by (simp add: di-d-def)
 have 3: ⊢ (¬ ((¬ empty);#True)) = (¬ (more;#True)) by (simp add: empty-d-def)
 have 4: ⊢ more;#True =  more using MoreEqvMoreChopTrue by auto
 hence 5: ⊢ (¬(more;#True)) =  (¬ more) by fastforce
 from 1 2 3 5 show ?thesis using NotEmptyEqvMore by fastforce
qed

lemma EmptyChopSkipInduct:
 assumes ⊢ empty ⟶ f
         ⊢ prev f ⟶ f
 shows  ⊢ f
proof −
 have 1: ⊢ empty  ⟶ f using assms(1) by auto
 have 2: ⊢ prev f ⟶ f  using assms(2) by blast
 have 3: ⊢ (empty ∨ prev f) ⟶ f using 1 2  by fastforce
 have 4: ⊢ wprev f = (empty ∨ prev f) by (simp add: WprevEqvEmptyOrPrev)
 hence 5: ⊢ wprev f ⟶ f using 3 by fastforce
 hence 6: ⊢ ¬f ⟶ ¬ (wprev f)  by fastforce
 hence 7: ⊢ ¬f ⟶ prev (¬ f)  by (simp add: wprev-d-def)
 hence 8: ⊢ ¬ ¬ f  by (rule PrevLoop)
 from 8 show ?thesis by auto
qed

lemma MoreImpImpChopSkipEqv:
⊢ more ⟶ ( (f⟶g);skip = ((f;skip)⟶(g;skip)) )
proof −
 have 01: ⊢ (f⟶g) = (¬ f ∨ g) by auto
 hence 02: ⊢ (f⟶g);skip = (¬ f ∨ g);skip  by (simp add: LeftChopEqvChop)
 hence 1: ⊢ (more ∧ (f⟶g);skip) = (more ∧  (¬ f ∨ g);skip) by fastforce
 have 2: ⊢ (¬ f ∨ g);skip = ((¬ f);skip ∨ g;skip)
      using OrChopEqv by auto
 hence 3: ⊢ (more ∧  (¬ f ∨ g);skip) = (more ∧ ((¬ f);skip ∨ g;skip))
      by auto
 have 4: ⊢ (¬((¬ f);skip)) = (empty ∨ (f;skip))
      using NotNotChopSkip by blast
 hence 5: ⊢ ((¬ f);skip) = (¬(empty ∨ (f;skip)))
      by fastforce
 have 6: ⊢ ¬(empty ∨ (f;skip)) = (more ∧ ¬(f;skip))
      using 5 NotChopSkipEqvMoreAndNotChopSkip by fastforce
 have 7: ⊢ ((¬ f);skip ∨ g;skip) = ( (more ∧ ¬(f;skip)) ∨ g;skip)
      using 5 6 by fastforce
 hence 8: ⊢ (more ∧(¬ f;skip ∨ g;skip)) = (more ∧ ( (more ∧ ¬(f;skip)) ∨ g;skip))
      by auto
 have 9: ⊢ (more ∧ ( (more ∧ ¬(f;skip)) ∨ g;skip)) = (more ∧ (¬(f;skip) ∨ g;skip))
      by auto
 have 10: ⊢ (more ∧ (¬(f;skip) ∨ g;skip)) = (more ∧ ((f;skip)⟶(g;skip)))
      by auto
 have 11: ⊢ (more ∧ (f⟶g);skip) = (more ∧ ((f;skip)⟶(g;skip)))
      using 1 2 3 8 9 10  7 by fastforce
 from 11 show ?thesis using MP by fastforce
qed

lemma MoreImpImpPrevEqv:
```

⊢ *more* ⟶ (*prev*(*f* ⟶*g*) = (*prev f* ⟶ *prev g*))
by (*simp add*: *MoreImpImpChopSkipEqv prev-d-def*)

lemma *BiBoxNotEqvNotTrueChopChopTrue*:
⊢ *bi*(□ (¬ *f*)) = (¬((#*True*;*f*);#*True*))
by (*simp add*: *bi-d-def always-d-def di-d-def sometimes-d-def*)

lemma *DiAndEmptyEqvAndEmpty*:
⊢ (*di f* ∧ *empty*) = (*f* ∧ *empty*)
proof −
have *1* : ⊢ *di f* = (*f* ∨ *di f*;*skip*)
  using *DiEqvOrDiChopSkipB* **by** *blast*
hence *2*: ⊢ (*di f* ∧ *empty*) = ((*f* ∨ *di f*;*skip*) ∧ *empty*)
  by *fastforce*
have *3* : ⊢ ((*f* ∨ *di f*;*skip*) ∧ *empty*) = ((*f* ∧ *empty*) ∨ (*di f*;*skip* ∧ *empty*))
  by *auto*
have *4*: ⊢ ¬(*di f*;*skip* ∧ *empty*)
  by (*metis AndChopB AndDiEqv ChopAndEmptyEqvEmptyChopEmpty DiEmpty MoreEqvSkipChopTrue TrueChopSkipEqvSkipChopTrue empty-d-def int-eq int-eq-true int-simps*(*14*) *int-simps*(*21*) *lift-and-com*)
hence *5* : ⊢ ((*f* ∧ *empty*) ∨ (*di f*;*skip* ∧ *empty*)) = (*f* ∧ *empty*)
  by *auto*
from *2 3 5* **show** *?thesis* **by** *fastforce*
qed

6.4.3 Strict initial intervals

lemma *DsMoreDi*:
⊢ *ds f* = (*more* ∧ (*di f*);*skip*)
proof −
have *1*: ⊢ *ds f* = (¬(*bs* (¬ *f*)))
  by (*simp add*: *ds-d-def*)
have *2*: ⊢ (¬(*bs* (¬ *f*))) = (¬(*empty* ∨ (*bi* (¬ *f*));*skip*))
  by (*simp add*: *bs-d-def*)
have *3*: ⊢ (¬(*empty* ∨ (*bi* (¬ *f*));*skip*)) = (¬*empty* ∧ ¬((*bi* (¬*f*));*skip*))
  by *auto*
have *4*: ⊢ (¬*empty* ∧ ¬((*bi* (¬*f*));*skip*)) = (*more* ∧ ¬((*bi* (¬*f*));*skip*))
  using *NotEmptyEqvMore* **by** *auto*
have *5*: ⊢ (*more* ∧ ¬((*bi* (¬*f*));*skip*)) = (*more* ∧ ¬(¬(*di f*);*skip*))
  by (*metis DiEqvNotBiNot DiIntro DiSkipEqvMore NotChopSkipEqvMoreAndNotChopSkip Prop10 RightChopImpMoreRule int-simps*(*4*) *inteq-reflection lift-and-com*)
have *6*: ⊢ (*more* ∧ ¬((¬(*di f*));*skip*)) = (*more* ∧ (*empty* ∨ (*di f*);*skip*))
  using *NotNotChopSkip* **by** *fastforce*
have *7*: ⊢ (*more* ∧ (*empty* ∨ (*di f*);*skip*)) = (*more* ∧ (*di f*);*skip*)
  using *NotEmptyEqvMore* **by** *auto*
from *1 2 3 4 5 6 7* **show** *?thesis* **by** *fastforce*
qed

lemma *DsDi*:

```
⊢ ds f = (di f);skip
proof –
 have 1: ⊢ ds f = (more ∧ (di f);skip) by (rule DsMoreDi)
 have 2: ⊢ (di f);skip ⟶ more  by (metis DiIntro DiSkipEqvMore RightChopImpMoreRule int-eq)
 hence 3: ⊢ (more ∧ (di f);skip) = (di f);skip by auto
 from 1 2 show ?thesis by fastforce
qed

lemma BsEqvNotDsNot:
⊢ bs f = (¬(ds (¬ f)))
proof –
 have 1: ⊢ ds (¬ f)  = (more ∧ (di (¬ f));skip)
      by (rule DsMoreDi)
 hence 2: ⊢ (¬(ds (¬ f)))  = (¬(more ∧ (di (¬ f));skip))
      by auto
 have 3: ⊢ (¬(more ∧ (di (¬ f));skip)) = (empty ∨ ¬((di (¬ f));skip))
      using NotEmptyEqvMore by auto
 have 4: ⊢ (empty ∨ ¬((di (¬ f));skip)) = (empty ∨ ¬((¬(bi f));skip))
      using DiNotEqvNotBi by (metis 3 inteq-reflection)
 have 5: ⊢  (¬((¬(bi f));skip)) = (empty ∨ (bi f);skip)
      by (rule NotNotChopSkip)
 hence 6: ⊢ (empty ∨ ¬((¬(bi f));skip)) = (empty ∨ (bi f);skip)
      by auto
 from 2 3 4 6 show ?thesis by (metis bs-d-def inteq-reflection)
qed

lemma NotBsEqvDsNot:
⊢ (¬(bs f)) = ds (¬ f)
proof –
 have 1: ⊢ bs f = (¬(ds (¬ f))) by (rule BsEqvNotDsNot)
 hence 2: ⊢ (¬( bs f)) = (¬¬(ds (¬ f)))  by auto
 from 2 show ?thesis by auto
qed

lemma NotDsEqvBsNot:
⊢ (¬ (ds f)) = bs (¬ f)
proof –
 have 1: ⊢ (¬(ds  f)) = (¬¬(bs (¬f))) by (simp add: ds-d-def)
 from 1 show ?thesis by auto
qed

lemma NotDsAndEmpty:
⊢ ¬(ds f ∧ empty)
proof –
 have 1: ⊢ ds f = (more ∧ (di f);skip) by (rule DsMoreDi)
 have 2: ⊢ more ∧ (di f);skip ∧ empty ⟶ #False using NotEmptyEqvMore by auto
 from 1 2 show ?thesis by fastforce
qed

lemma BsMoreEqvEmpty:
```

```
⊢ bs more = empty
proof –
 have 1: ⊢ bs more = (empty ∨ (bi more);skip) by (simp add: bs-d-def)
 have 2: ⊢ bi more ⟶ #False using DiEmpty NotEmptyEqvMore by (simp add: bi-d-def empty-d-def)
 hence 3: ⊢ (bi more);skip ⟶ #False;skip  using LeftChopImpChop by blast
 have 31: ⊢ #False;skip ⟶ #False  by (simp add: Valid-def skip-defs chop-defs)
 have 32: ⊢ (bi more);skip ⟶ #False  using 3 31 by fastforce
 hence 4: ⊢ (empty ∨ ((bi more);skip)) = empty  by fastforce
 from 1 4 show ?thesis by fastforce
qed

lemma  BsAndEqv:
⊢ (bs f ∧ bs g) = bs(f ∧ g)
proof –
 have 1: ⊢ bs f = (empty ∨ (bi f) ;skip)
     by (simp add: bs-d-def)
 have 2: ⊢ bs g = (empty ∨ (bi g) ;skip)
     by (simp add: bs-d-def)
 have 3: ⊢ (bs f ∧ bs g) = ((empty ∨ (bi f) ;skip) ∧ (empty ∨ (bi g) ;skip))
     using 1 2 by fastforce
 have 4: ⊢ ((empty ∨ (bi f) ;skip) ∧ (empty ∨ (bi g) ;skip)) =
           (empty ∨ ((bi f) ;skip ∧ (bi g) ;skip))
     by auto
 have 5: ⊢ (((bi f) ;skip ∧ (bi g) ;skip)) = bi(f ∧ g);skip
     using BiAndChopSkipEqv  by fastforce
 hence 6: ⊢ (empty ∨ ((bi f) ;skip ∧ (bi g) ;skip)) = (empty ∨ bi(f ∧ g);skip)
     by auto
 from 3 4 6 show ?thesis by (metis bs-d-def inteq-reflection)
qed

lemma DsEqvRule:
 assumes ⊢ f = g
 shows ⊢ ds f = ds g
using assms using int-eq by force

lemma DsOrEqv:
⊢ (ds f ∨ ds g) = ds (f ∨ g)
proof –
 have 1: ⊢ ds f = (¬(bs (¬ f))) by (simp add: ds-d-def)
 have 2: ⊢ ds g = (¬(bs (¬ g))) by (simp add: ds-d-def)
 have 3: ⊢ (ds f ∨ ds g) = (¬(bs (¬ f)) ∨ ¬(bs (¬ g))) using 1 2 by fastforce
 have 4: ⊢ (¬(bs (¬ f)) ∨ ¬(bs (¬ g))) = (¬(bs (¬ f) ∧ bs (¬ g))) by auto
 have 5: ⊢ (bs (¬ f) ∧ bs (¬ g)) = bs( ¬ f ∧ ¬g) by (rule BsAndEqv)
 hence 6: ⊢ (¬(bs (¬ f) ∧ bs (¬ g))) = (¬(bs( ¬ f ∧ ¬g)))  by auto
 have 7: ⊢ (¬(bs( ¬ f ∧ ¬g))) = ds (¬(¬ f ∧ ¬g))  by (rule NotBsEqvDsNot)
 have 8: ⊢ (¬(¬ f ∧ ¬g)) = (f ∨ g) by auto
 hence 9: ⊢ ds(¬(¬ f ∧ ¬g)) = ds (f ∨ g)  by (rule DsEqvRule)
 from 3 4 6 7 9 show ?thesis by fastforce
qed
```

```
lemma BsOrImp:
⊢ bs f ∨ bs g ⟶ bs(f ∨ g)
proof −
 have 1: ⊢ bi f ∨ bi g ⟶ bi(f ∨ g)
     by (rule BiOrBiImpBiOr)
 hence 2: ⊢ (bi f ∨ bi g);skip ⟶ (bi(f ∨ g));skip
     by (rule LeftChopImpChop)
 have 3: ⊢ (bi f);skip ∨ (bi g);skip ⟶ (bi(f ∨ g));skip
     using 1 OrChopEqv 2 by fastforce
 hence 4: ⊢ empty ∨ (bi f);skip ∨ (bi g);skip ⟶  empty ∨ (bi(f ∨ g));skip
     by auto
 hence 5: ⊢ (empty ∨ (bi f);skip) ∨ (empty ∨ (bi g);skip) ⟶  empty ∨ (bi(f ∨ g));skip
     by auto
 from 5 show ?thesis by (simp add: bs-d-def)
qed

lemma DsAndImp:
⊢ ds (f ∧ g) ⟶ ds f ∧ ds g
proof −
 have 1: ⊢ bs (¬f) ∨ bs (¬g) ⟶ bs(¬f ∨ ¬g) by (rule BsOrImp)
 have 2: ⊢ (¬f ∨ ¬g) = (¬(f ∧ g)) by auto
 hence 3: ⊢ bs(¬f ∨ ¬g) = bs (¬(f ∧ g)) by (rule BsEqvRule)
 have 4: ⊢ bs (¬f) ∨ bs (¬g) ⟶ bs (¬(f ∧ g)) using 1 3 by fastforce
 have 5: ⊢ bs (¬f) = (¬(ds f)) using NotDsEqvBsNot by fastforce
 have 6: ⊢ bs (¬g) = (¬(ds g)) using NotDsEqvBsNot by fastforce
 have 7: ⊢ bs (¬(f ∧ g)) =  (¬(ds (f ∧ g))) using NotDsEqvBsNot by fastforce
 have 8: ⊢  ¬(ds f) ∨  ¬(ds g) ⟶  ¬(ds (f ∧ g)) using 4 5 6 7 by fastforce
 hence 9: ⊢ ¬(ds f ∧ ds g)  ⟶  ¬(ds (f ∧ g)) by auto
 from 9 show ?thesis by auto
qed

lemma DsAndImpElimL:
⊢ ds (f ∧ g) ⟶ ds f
using DsAndImp by fastforce

lemma DsAndImpElimR:
⊢ ds (f ∧ g) ⟶ ds g
using DsAndImp by fastforce

lemma BiImpBs:
⊢ bi f ⟶ bs f
proof −
 have 1: ⊢ empty ⟶ empty ∨ (bi f);skip by auto
 hence 2: ⊢ empty ∧ bi f ⟶ empty ∨ (bi f);skip by auto
 have 2: ⊢ more ∧ bi f ⟶ (bi f);skip by (rule MoreAndBiImpBiChopSkip)
 hence 3: ⊢ more ∧ bi f ⟶ empty ∨ (bi f);skip  by auto
 have 4: ⊢ bi f = ((bi f ∧ empty) ∨ (bi f ∧ more)) by  (simp add: empty-d-def, auto)
 have 5: ⊢ (empty ∨ (bi f);skip) = bs f  by (simp add: bs-d-def)
 from 2 3 4 5 show ?thesis by fastforce
qed
```

```
lemma BsImpBsBs:
⊢ bs f ⟶ bs ( bs f)
proof −
 have 1: ⊢ bi f ⟶ bs f by (rule BiImpBs)
 hence 2: ⊢ bi (bi f) ⟶ bi(bs f) by (rule BiImpBiRule)
 hence 3: ⊢ (bi f) ⟶ bi(bs f) using BiEqvBiBi by fastforce
 hence 4: ⊢ (bi f);skip ⟶ (bi(bs f));skip by (rule LeftChopImpChop)
 hence 5: ⊢ empty ∨ (bi f);skip ⟶ empty ∨ (bi(bs f));skip by auto
 from 5 show ?thesis by (simp add: bs-d-def)
qed

lemma DsImpDi:
⊢ ds f ⟶ di f
proof −
 have 1: ⊢ bi (¬ f) ⟶ bs (¬ f) by (rule BiImpBs)
 hence 2: ⊢ ¬(bs (¬ f)) ⟶ ¬(bi (¬ f))  by fastforce
 from 2 show ?thesis using NotBsEqvDsNot DiEqvNotBiNot by fastforce
qed

lemma BsImpBsRule:
 assumes ⊢ f ⟶ g
 shows ⊢ bs f  ⟶ bs g
proof −
 have 1: ⊢ f ⟶ g using assms by auto
 hence 2: ⊢ bi f ⟶ bi g  by (rule BiImpBiRule)
 hence 3: ⊢ (bi f);skip ⟶ (bi g);skip  by (rule LeftChopImpChop)
 hence 4: ⊢ empty ∨ (bi f);skip ⟶ empty ∨ (bi g);skip by auto
 from 4 show ?thesis by (simp add: bs-d-def)
qed

lemma DsChopImpDsB:
⊢ ds (f;g) ⟶ ds f
proof −
 have 1: ⊢ di(f;g) ⟶ di f by (rule DiChopImpDiB)
 hence 2: ⊢ (di(f;g));skip ⟶ (di f);skip  by (rule LeftChopImpChop)
 from 2 show ?thesis using DsDi by fastforce
qed

lemma NotBsImpBsNotChop:
⊢ bs (¬ f) ⟶ bs ( ¬(f;g))
proof −
 have 1: ⊢ ds (f;g) ⟶ ds f by (rule DsChopImpDsB)
 hence 2: ⊢ ¬(ds f) ⟶ ¬(ds (f;g)) by fastforce
 from 2 show ?thesis using NotDsEqvBsNot by fastforce
qed


lemma BsOrBsEqvBsBiOrBi:
⊢ (bs f ∨ bs g) = bs(bi f ∨ bi g)
```

```
proof −
 have 1: ⊢ (bs f ∨ bs g) = ((empty ∨ (bi f);skip) ∨ (empty ∨ (bi g);skip))
     by (simp add: bs-d-def)
 have 2: ⊢ ((empty ∨ (bi f);skip) ∨ (empty ∨ (bi g);skip)) = (empty ∨ (bi f);skip ∨ (bi g);skip)
     by auto
 have 3: ⊢ ((bi f);skip ∨ (bi g);skip) = (bi f ∨ bi g);skip
     using OrChopEqv by fastforce
 hence 4: ⊢ (empty ∨ (bi f);skip ∨ (bi g);skip) = (empty ∨ (bi f ∨ bi g);skip)
     by auto
 have 5: ⊢ (bi f ∨ bi g) = bi ( bi f ∨ bi g)
     by (meson BiBiOrImpBi BiImpBiBiOr BiImpBiBiOrB Prop02 int-iffI)
 hence 6: ⊢ (bi f ∨ bi g);skip = bi ( bi f ∨ bi g);skip
     by (simp add: LeftChopEqvChop)
 hence 7: ⊢ (empty ∨ bi ( bi f ∨ bi g);skip) = (empty ∨ (bi f ∨ bi g);skip)
     by auto
 have 8: ⊢ (empty ∨ (bi f ∨ bi g);skip) = bs(bi f ∨ bi g) using bs-d-def
     by (metis 4 5 inteq-reflection)
 from 1 2 4 8 show ?thesis by (metis inteq-reflection)
qed

lemma DiOrDsEqvDi:
⊢ di f ∨ ds f = di f
proof −
 have 1: ⊢ di f ⟶ di f ∨ ds f by auto
 have 2: ⊢ di f ⟶ di f by auto
 have 3: ⊢ ds f ⟶ di f by (rule DsImpDi)
 have 4: ⊢ di f ∨ ds f ⟶ di f using 2 3 by auto
 from 1 4 show ?thesis by auto
qed

lemma DiAndDsEqvDs:
⊢ (di f ∧ ds f) = ds f
proof −
 have 1: ⊢ di f ∧ ds f ⟶ ds f by auto
 have 2: ⊢ ds f ⟶ ds f  by auto
 have 3: ⊢ ds f ⟶ di f by (rule DsImpDi)
 have 4: ⊢ ds f ⟶ di f ∧ ds f using 2 3 by auto
 from 1 4 show ?thesis by auto
qed

lemma OrDsEqvDi:
⊢ (f ∨ ds f) = di f
proof −
 have 1: ⊢ ds f =  (di f);skip by (rule DsDi)
 hence 2: ⊢ (f ∨ ds f) = (f ∨ (di f);skip) by auto
 from 2 show ?thesis using DiEqvOrDiChopSkipB by fastforce
qed

lemma AndBsEqvBi:
⊢ (f ∧ bs f) = bi f
```

```
proof –
 have 1: ⊢ (f ∧ bs f) = (f ∧ (empty ∨ (bi f);skip)) by (simp add: bs-d-def)
 from 1 show ?thesis using BiEqvAndEmptyOrBiChopSkip by fastforce
qed

lemma BsEqvBsBi:
⊢ bs f = bs (bi f)
proof –
 have 1: ⊢ bs f = (empty ∨ (bi f);skip) by (simp add: bs-d-def)
 have 2: ⊢ bi f = bi ( bi f) by (rule BiEqvBiBi)
 hence 3: ⊢ (bi f);skip = bi (bi f);skip using LeftChopEqvChop by blast
 hence 4: ⊢ (empty ∨ (bi f);skip) = (empty ∨ bi (bi f);skip) by auto
 from 1 4 show ?thesis by (simp add: bs-d-def)
qed

lemma StateImpBs:
⊢ init w ⟶ bs (init w)
proof –
 have 1: ⊢ init w = bi (init w) by (rule StateEqvBi)
 have 2: ⊢ bi (init w) ⟶ bs (init w) by (rule BiImpBs)
 from 1 2 show ?thesis using StateImpBi by fastforce
qed

lemma DsAndDsEqvDsAndDiOrDsAndDi:
⊢ (ds f ∧ ds g) =( ds (f ∧ di g) ∨ ds(g ∧ di f))
proof –
 have 1: ⊢ (di f ∧ di g) = (di(f ∧ di g) ∨ di(g ∧ di f))
      by (rule DiAndDiEqvDiAndDiOrDiAndDi)
 hence 2: ⊢ (di f ∧ di g);skip = (di(f ∧ di g) ∨ di(g ∧ di f));skip
      by (rule LeftChopEqvChop)
 have 3: ⊢ (di f ∧ di g);skip = ((di f);skip ∧ (di g);skip)
      using ChopSkipAndChopSkip by fastforce
 have 4: ⊢ ((di f);skip ∧ (di g);skip) = (di(f ∧ di g) ∨ di(g ∧ di f));skip
      using 2 3 by fastforce
 have 5: ⊢ (di(f ∧ di g) ∨ di(g ∧ di f));skip = (di(f ∧ di g);skip ∨ di(g ∧ di f);skip)
      using OrChopEqv by blast
 have 6: ⊢ ds f = (di f);skip
      using DsDi by blast
 have 7: ⊢ ds g = (di g);skip
      using DsDi by blast
 have 8: ⊢ ((di f);skip ∧ (di g);skip) = (ds f ∧ ds g)
      using 6 7 by fastforce
 have 9: ⊢ ds(f ∧ di g) = di(f ∧ di g);skip
      using DsDi by blast
 have 10: ⊢ ds(g ∧ di f) = di(g ∧ di f);skip
      using DsDi by blast
 have 11: ⊢ (di(f ∧ di g);skip ∨ di(g ∧ di f);skip) = (ds(f ∧ di g) ∨ ds(g ∧ di f))
      using 9 10 by fastforce
 from 4 5 8 11 show ?thesis by fastforce
qed
```

```
lemma BsEqvBiMoreImpChop:
⊢ bs f = bi(more ⟶ f;skip)
proof −
 have 1: ⊢ bs f = (empty ∨ (bi f;skip))
     by (simp add: bs-d-def)
 have 2: ⊢ (empty ∨ (bi f;skip)) = ((¬(¬(bi f));skip))
     using NotNotChopSkip by fastforce
 have 3: ⊢ ¬((¬(bi f));skip) = (¬(di (¬ f);skip))
     by (simp add: bi-d-def)
 have 4: ⊢ (¬(di (¬ f);skip)) = (¬(((¬ f) ;#True);skip))
     by (simp add:di-d-def)
 have 5: ⊢ (¬(((¬ f) ;#True);skip)) = (¬((¬ f) ;(#True;skip)))
     using ChopAssocB by fastforce
 have 6: ⊢ (¬((¬ f) ;(#True;skip))) = (¬((¬ f) ;(skip;#True)))
     using SkipTrueEqvTrueSkip TrueChopSkipEqvSkipChopTrue RightChopEqvChop by fastforce
 have 7: ⊢ (¬((¬ f) ;(skip;#True))) = (¬(((¬ f) ;skip);#True))
     using ChopAssoc by fastforce
 have 8: ⊢ (¬(((¬ f) ;skip);#True)) = (¬(di ((¬ f);skip)))
     by (simp add: di-d-def)
 have 9: ⊢ (¬(di ((¬ f);skip))) = bi (¬((¬ f) ;skip))
     using NotDiEqvBiNot by blast
 have 10: ⊢ bi (¬((¬ f) ;skip)) = bi( empty ∨ (f;skip))
     using NotNotChopSkip using BiEqvBi by blast
 have 11: ⊢ bi( empty ∨ (f;skip)) = bi( ¬ more ∨ (f;skip))
     by (simp add: empty-d-def)
 have 12: ⊢ ( ¬ more ∨ (f;skip)) = (more ⟶ f;skip) by auto
 have 13: ⊢ bi( ¬ more ∨ (f;skip)) = bi(more ⟶ f;skip) using 12 using BiEqvBi by blast
 have 14: ⊢ bs f = (¬ (((¬ f);skip);#True)) using 1 2 3 4 5 6 7 by fastforce
 have 15: ⊢ (¬ (((¬ f);skip);#True)) = bi(more ⟶ f;skip) using 8 9 10 11 13 by fastforce
 from 14 15 show ?thesis by fastforce
qed

lemma BoxMoreStateEqvBsFinState:
⊢ □(more ⟶ ¬ (init w)) = bs(¬(fin(init w)))
proof −
 have 1: ⊢ □(more ⟶ ¬ (init w)) = (¬(◇(¬(more ⟶ ¬ (init w)))))
     by (simp add: always-d-def)
 have 01: ⊢ (¬(more ⟶ ¬ (init w))) = (init w ∧  more) by auto
 hence 2: ⊢ ¬(◇(¬(more ⟶ ¬ (init w)))) = (¬(#True;(init w ∧  more)))
     by (metis int-eq int-iffD1 int-simps(14) int-simps(6) sometimes-d-def)
 have 3: ⊢ more = #True; skip
     using MoreEqvSkipChopTrue SkipTrueEqvTrueSkip by fastforce
 have 4: ⊢ (init w ∧  more) = (init w ∧ (#True; skip))
     using 3 by auto
 have 5: ⊢ (init w ∧ (#True; skip))  = ((init w ∧ empty);(#True;skip))
     using StateAndEmptyChop by fastforce
 have 6: ⊢ (init w ∧  more) = ((init w ∧ empty);(#True;skip))
     using 4 5 by fastforce
 have 7: ⊢  (#True;(init w ∧  more)) = (#True;((init w ∧ empty);(#True;skip)) )
```

using *6 RightChopEqvChop* **by** *blast*

have *8*: $\vdash (\#True;((init\ w \wedge empty);(\#True;skip))\ ) =$
$(((\#True;(init\ w \wedge empty));(\#True;skip))\ )$
using *ChopAssoc* **by** *blast*

have *9*: $\vdash (((\#True;(init\ w \wedge empty));(\#True;skip))\ ) =$
$((((\#True;(init\ w \wedge empty));\#True);skip)\ )$
using *ChopAssoc* **by** *blast*

have *10*: $\vdash (\#True;(init\ w \wedge\ more)) =$
$((((\#True;(init\ w \wedge empty));\#True);skip)\ )$
using *7 8 9* **by** *fastforce*

hence *11*: $\vdash (\neg(\#True;(init\ w \wedge\ more))) =$
$(\neg((((\#True;(init\ w \wedge empty));\#True);skip)\ ))$
by *auto*

have *12*: $\vdash \neg((((\#True;(init\ w \wedge empty));\#True);skip)\ ) =$
$empty \vee (\neg((\#True;(init\ w \wedge empty));\#True);skip)$
using *NotChopNotSkip* **by** *fastforce*

have *13*: $\vdash (\neg((\#True;(init\ w \wedge empty));\#True)) = bi(\Box\ (\neg(init\ w \wedge empty)))$
using *BiBoxNotEqvNotTrueChopChopTrue* **by** *fastforce*

hence *14*: $\vdash (\neg((\#True;(init\ w \wedge empty));\#True));skip =$
$(bi(\Box\ (\neg(init\ w \wedge empty))));skip$
using *RightChopEqvChop* **by** (*simp add*: *LeftChopEqvChop*)

hence *15*: $\vdash empty \vee (\neg((\#True;(init\ w \wedge empty));\#True));skip =$
$empty\ \vee(bi(\Box\ (\neg(init\ w \wedge empty))));skip$
by *auto*

have *16*: $\vdash (\neg((((\#True;(init\ w \wedge empty));\#True);skip)\ )) =$
$(empty \vee (bi(\Box\ (\neg(init\ w \wedge empty)));skip))$
using *12 15* **using** *14 NotChopNotSkip int-eq* **by** *fastforce*

have *171*: $\vdash (\neg(init\ w \wedge empty)) = (\neg(init\ w) \vee \neg empty)$
by *auto*

hence *172*: $\vdash \Box(\neg(init\ w \wedge empty)) = \Box(\neg(init\ w) \vee \neg empty)$
by (*simp add*: *BoxEqvBox*)

hence *173*: $\vdash bi(\Box(\neg(init\ w \wedge empty))) = bi(\Box(\neg(init\ w) \vee \neg empty))$
by (*simp add*: *BiEqvBi*)

hence *174*: $\vdash bi(\Box(\neg(init\ w \wedge empty)));skip = bi(\Box(\neg(init\ w) \vee \neg empty));skip$
using *LeftChopEqvChop* **by** *blast*

hence *17*: $\vdash (empty \vee (bi(\Box\ (\neg(init\ w \wedge empty)));skip)) =$
$(empty \vee (bi(\Box\ (\neg(init\ w) \vee \neg empty));skip))$
by *auto*

have *181*: $\vdash\ (\neg(init\ w) \vee \neg empty) =\ (\neg empty \vee \neg(init\ w)\ \ )$
by *auto*

hence *18*: $\vdash \Box\ (\neg(init\ w) \vee \neg empty) = \Box\ (\neg empty \vee \neg(init\ w)\ \ )$
by (*simp add*: *BoxEqvBox*)

have *191*: $\vdash\ (\neg empty \vee \neg(init\ w)\ \ ) = (empty \longrightarrow \neg(init\ w)\ )$
by *auto*

hence *19*: $\vdash \Box\ (\neg empty \vee \neg(init\ w)\ \ ) = \Box(empty \longrightarrow \neg(init\ w)\ )$
by (*simp add*: *BoxEqvBox*)

have *20*: $\vdash \Box(empty \longrightarrow \neg(init\ w)\ ) = fin\ (\neg(init\ w)\ )$
by (*simp add*: *fin-d-def*)

have *21*: $\vdash fin\ (\neg(init\ w)\ ) = (\neg(fin\ (init\ w)))$
using *FinEqvFin FinNotStateEqvNotFinState Initprop*(*2*) **by** *fastforce*

```
 have 22: ⊢ bi(□ (¬(init w) ∨ ¬empty)) = bi (¬(fin (init w)))
     using 18 19 20 21 BiEqvBi by (metis int-eq)
 hence 23: ⊢ (bi(□ (¬(init w) ∨ ¬empty)));skip = (bi (¬(fin (init w))));skip
     using RightChopEqvChop by (simp add: LeftChopEqvChop)
 hence 24: ⊢ (empty ∨ (bi(□ (¬(init w) ∨ ¬empty)));skip) =
           (empty ∨ (bi (¬(fin (init w))));skip)
     by auto
 hence 25: ⊢ (empty ∨ (bi (¬(fin (init w))));skip) = bs(¬(fin (init w)))
     by (simp add:bs-d-def)
 from 1 2 11 16 17 24  25 show ?thesis by fastforce
qed

lemma BsFalseEqvEmpty:
⊢ bs #False = empty
proof −
 have 1: ⊢ bs #False = (empty ∨ bi #False;skip)
     by (simp add: bs-d-def)
 have 2: ⊢ ¬(bi #False;skip)
     by (metis BiEqvAndWprevBi MoreEqvSkipChopTrue NotChopSkipEqvMoreAndNotChopSkip
         SkipTrueEqvTrueSkip int-eq int-iffD1 int-simps(14) int-simps(19) int-simps(2)
         int-simps(21))
 from 1 2 show ?thesis by fastforce
qed
```

6.4.4 First occurrence

```
lemma FstWithAndImp:
⊢ ▷f ∧ g ⟶ ▷ ( f ∧ g)
proof −
 have 1: ⊢ (▷f ∧ g) = ((f ∧ (bs (¬f))) ∧ g)
     by (simp add: first-d-def)
 have 2: ⊢ ((f ∧ (bs (¬f))) ∧ g) = (f ∧ ¬(ds f) ∧ g)
     using NotDsEqvBsNot by fastforce
 have 3: ⊢ ¬(ds f) ⟶ ¬(ds(f ∧ g))
     using DsAndImpElimL by fastforce
 hence 4: ⊢ f ∧ ¬(ds f) ∧ g ⟶ f ∧ g ∧ ¬(ds(f ∧ g))
     by auto
 have  5: ⊢ (f ∧ g ∧ ¬(ds(f ∧ g))) = ((f ∧ g) ∧ (bs (¬(f ∧ g))))
     using NotDsEqvBsNot by fastforce
 have  6: ⊢ ((f ∧ g) ∧ (bs (¬(f ∧ g)))) = ▷(f ∧ g)
     by (simp add: first-d-def)
 from 1 2 4 5 6 show ?thesis by fastforce
qed

lemma FstWithOrEqv:
⊢ ▷(f ∨ g) = ((▷f ∧ bs (¬g)) ∨ (▷g ∧ bs (¬f)))
proof −
 have 1: ⊢ ▷(f ∨ g) = ((f ∨ g) ∧ bs (¬(f ∨ g)))
     by (simp add: first-d-def)
 have 2: ⊢  (¬(f ∨ g)) =  (¬f ∧ ¬g)
```

```
    by auto
 hence 3: ⊢ bs (¬(f ∨ g)) =  bs (¬f ∧ ¬g)
    using BsEqvRule by blast
 have  4: ⊢  bs (¬f ∧ ¬g) = (bs (¬ f) ∧ bs (¬ g))
    using BsAndEqv  by fastforce
 have  5: ⊢ ((f ∨ g) ∧ bs (¬(f ∨ g))) = ((f ∨ g) ∧ bs (¬ f) ∧ bs (¬ g))
    using 3 4 by fastforce
 have  6: ⊢ ((f ∨ g) ∧ bs (¬ f) ∧ bs (¬ g)) =
          (((f ∧ bs (¬ f)) ∧ bs (¬ g)) ∨ (g ∧ bs (¬ f) ∧ bs (¬ g)))
    by auto
 have  7: ⊢ ((f ∧ bs (¬ f)) ∧ bs (¬ g)) = (▷f ∧  bs (¬ g))
    by (simp add: first-d-def)
 have  8: ⊢ (g ∧ bs (¬ f) ∧ bs (¬ g)) = ((g ∧ bs (¬ g)) ∧ bs (¬ f))
    by auto
 have  9: ⊢ ((g ∧ bs (¬ g)) ∧ bs (¬ f)) = (▷g ∧  bs (¬ f))
    by (simp add: first-d-def)
 have 10: ⊢ ((f ∧ bs (¬ f)) ∧ bs (¬ g)) ∨ (g ∧ bs (¬ f) ∧ bs (¬ g)) =
          (▷f ∧  bs (¬ g)) ∨ (▷g ∧  bs (¬ f))
    using 7 8 9 by fastforce
 from 1 5 6 10 show ?thesis by (metis 7 8 9 int-eq)
qed

lemma FstFstAndEqvFstAnd:
⊢ ▷(▷f ∧ g) = (▷f ∧ g)
proof −
  have 1: ⊢ (▷f ∧ g) = ((f ∧ (bs (¬f))) ∧ g) by (simp add: first-d-def)
  hence 2: ⊢ ▷f ∧ g ⟶ (bs (¬f)) by auto
  hence 3: ⊢ ▷f ∧ g ⟶ ▷f ∧ g ∧ (bs (¬f))  by auto
  have 4: ⊢ ¬ f ⟶ ¬ f ∨ ¬(bs (¬f)) ∨ ¬ g by auto
  hence 5: ⊢ bs (¬ f) ⟶ bs(¬ f ∨ ¬(bs (¬f)) ∨ ¬ g) using BsImpBsRule by blast
  have 6: ⊢ (¬ f ∨ ¬(bs (¬f)) ∨ ¬ g) = (¬(f ∧ bs (¬ f) ∧ g)) by auto
  hence 7: ⊢ bs(¬ f ∨ ¬(bs (¬f)) ∨ ¬ g) = bs(¬(f ∧ bs (¬ f) ∧ g)) using BsEqvRule by blast
  have 8: ⊢ ((f ∧ bs (¬ f)) ∧ g) = (▷f ∧ g)    by (simp add: first-d-def)
  hence 9: ⊢ (¬(f ∧ bs (¬ f) ∧ g)) = (¬(▷f ∧ g)) by auto
  hence 10: ⊢ bs (¬(f ∧ bs (¬ f) ∧ g)) = bs (¬(▷f ∧ g)) using BsEqvRule by blast
  have 11: ⊢ ▷f ∧ g ⟶ (▷f ∧ g) ∧   bs (¬(▷f ∧ g)) using 3 5 7 10 by fastforce
  hence 12: ⊢ ▷f ∧ g ⟶ ▷(▷f ∧ g) by (simp add: first-d-def)
  have 13: ⊢ ▷(▷f ∧ g) = ((▷f ∧ g) ∧   bs (¬(▷f ∧ g)))  by (simp add: first-d-def)
  hence 14: ⊢ ▷(▷f ∧ g) ⟶ ▷f ∧ g by auto
 from 12 14 show ?thesis by fastforce
qed

lemma FstTrue:
⊢ ▷ #True = empty
proof −
 have 1: ⊢ ▷ #True = (#True ∧ bs (¬ #True))
    by (simp add: first-d-def)
 have 2: ⊢ bs (¬ #True) = (empty ∨ (bi (¬ #True));skip)
    by (simp add: bs-d-def)
 have 3: ⊢ ¬(bi (¬ #True))
```

```
    using BiElim by fastforce
  have 4: ⊢ ¬((bi (¬ #True));skip)
    by (metis AndChopA BiEqvAndEmptyOrBiChopSkip MoreEqvSkipChopTrue
       NotChopSkipEqvMoreAndNotChopSkip SkipTrueEqvTrueSkip int-eq int-simps(14) int-simps(21))
  have 5: ⊢ bs (¬ #True) = empty
    using 2 4 by fastforce
  from 1 5 show ?thesis by fastforce
qed

lemma FstFalse:
 ⊢ ¬(▷ #False)
proof −
  have 1: ⊢ ▷ #False = (#False ∧ bs #True) by (simp add: first-d-def)
  from 1 show ?thesis by auto
qed

lemma FstChopFalseEqvFalse:
 ⊢ ¬(▷f ; #False)
by (simp add:Valid-def chop-defs)

lemma FstEmpty:
 ⊢ ▷ empty = empty
proof −
  have 1: ⊢ ▷ empty = (empty ∧ bs (¬ empty)) by (simp add: first-d-def)
  have 2: ⊢ bs (¬ empty) = (empty ∨ bi (¬ empty);skip) by (simp add: bs-d-def)
  from 1 2 show ?thesis by fastforce
qed

lemma FstAndEmptyEqvAndEmpty:
 ⊢ (▷f ∧ empty) = (f ∧ empty)
proof −
  have 1: ⊢ (▷f ∧ empty) = ((f ∧ bs (¬ f)) ∧ empty) by (simp add: first-d-def)
  have 2: ⊢ bs (¬ f) = (empty ∨ bi (¬f);skip) by (simp add: bs-d-def)
  from 1 2 show ?thesis by fastforce
qed

lemma FstEmptyOrEqvEmpty:
 ⊢ ▷(empty ∨ f) = empty
proof −
  have 1: ⊢ ▷(empty ∨ f) = ((▷empty ∧ bs (¬ f)) ∨ (▷f ∧ bs (¬ empty))) using FstWithOrEqv by blast
  have 2: ⊢  (¬ empty) = more  by (simp add: empty-d-def)
  hence 3: ⊢ bs (¬ empty) = bs more using BsEqvRule by blast
  have 4: ⊢ bs more = empty using BsMoreEqvEmpty by blast
  have 5: ⊢ (▷f ∧ bs (¬ empty)) = (▷f ∧ empty) using 3 4 by fastforce
  have  6: ⊢ ▷empty = empty using FstEmpty by blast
  hence 7: ⊢ (▷empty ∧ bs (¬ f)) = (empty ∧ bs (¬ f)) by auto
  have 8: ⊢ (empty ∧  bs (¬ f)) = (empty ∧(empty ∨ bi (¬ f);skip)) by (simp add:bs-d-def)
  have 9: ⊢ (empty ∧(empty ∨ bi (¬ f);skip)) = empty by auto
  have 10: ⊢ (empty ∧  bs (¬ f)) = empty  using 8 9 by auto
  have 11: ⊢ ((▷empty ∧ bs (¬ f)) ∨ (▷f ∧ bs (¬ empty))) =
       (empty ∨ (▷f ∧ empty)) using 7 10 5 by fastforce
  have 12: ⊢ (empty ∨ (▷f ∧ empty)) = empty by auto
  from 1 11 12 show ?thesis by fastforce
qed

lemma FstChopEmptyEqvFstChopFstEmpty:
 ⊢ (▷f;g ∧ empty) = (▷f;▷g ∧ empty)
proof −
  have 1: ⊢ (▷f;g ∧ empty) = (▷f ∧ g ∧ empty) using ChopEmptyAndEmpty by blast
  have 2: ⊢ (▷g ∧ empty) = (g ∧ empty)  using FstAndEmptyEqvAndEmpty by blast
  hence 3: ⊢ (▷f ∧ g ∧ empty) = (▷f ∧ ▷g ∧ empty) by auto
  have 4: ⊢ (▷f;▷g ∧ empty) = (▷f ∧ ▷g ∧ empty) using ChopEmptyAndEmpty by blast
  from 1 3 4 show ?thesis by fastforce
qed

lemma FstMoreEqvSkip:
 ⊢ ▷ more = skip
proof −
  have 1: ⊢ ▷ more = (more ∧ bs (¬ more))  by (simp add: first-d-def)
  have 2: ⊢ (more ∧ bs (¬ more)) = (more ∧ (empty ∨ bi (¬ more);skip))  by (simp add:bs-d-def)
  have 3: ⊢ (more ∧ (empty ∨ bi (¬ more);skip)) = (more ∧ bi (¬ more);skip) using empty-d-def
  using MoreAndEmptyOrEqvMoreAnd by fastforce
  have 4: ⊢ (more ∧ ((bi (¬ more));skip)) = ((bi (¬ more));skip) using ChopSkipImpMore by fastforce
  have 5: ⊢ ((bi (¬ more));skip) = bi empty;skip by (simp add: empty-d-def)
  have 6: ⊢ bi empty = empty using BiEmptyEqvEmpty by auto
  hence 7: ⊢ bi empty;skip = empty;skip using LeftChopEqvChop by blast
  have 8: ⊢ empty;skip = skip  using EmptyChop by blast
  from 1 2 3 4 5 7 8 show ?thesis by (metis int-eq)
qed

lemma FstEqvBsNotAndDi:
⊢ ▷f = (bs (¬ f) ∧ di f)
proof −
  have 1: ⊢ bs (¬ f)  = (¬(ds f))  by (simp add: ds-d-def)
  hence 2: ⊢ (bs (¬ f)  ∧ di f) = (¬(ds f) ∧ di f) by auto
  have 3: ⊢  di f = (ds f ∨ f) using OrDsEqvDi by fastforce
  hence 4 : ⊢ (¬(ds f)  ∧ di f) = (¬(ds f)  ∧ (ds f ∨ f)) by auto
  have 5 : ⊢ (¬(ds f)  ∧ (ds f ∨ f)) = (¬(ds f)  ∧ f) by auto
  have 6 : ⊢ (¬(ds f)  ∧ f) = (f ∧  bs (¬ f)) using 1 by auto
  from 2 4 5 6 show ?thesis by (metis first-d-def int-eq)
qed

lemma FstOrDiEqvDi:
 ⊢ (▷f ∨ di f) = di f
proof −
  have 1: ⊢ (▷f ∨ di f) = ((f ∧ bs (¬f)) ∨ di f) by (simp add: first-d-def)
  have 2: ⊢ ((f ∧ bs (¬f)) ∨ di f) = ((f ∨ di f) ∧ (bs (¬ f) ∨ di f)) by auto
  have 3: ⊢ (f ∨ di f) = di f
  by (metis 2 DiIntro RRDiamondEqvDi int-eq Prop02 Prop03 Prop11 Prop12)
  hence 4: ⊢ ((f ∨ di f) ∧ (bs (¬ f) ∨ di f)) = (di f ∧ (bs (¬ f) ∨ di f)) by auto
```

have *5*: ⊢ (*di f* ∧ (*bs* (¬ *f*) ∨ *di f*)) = *di f* **by** *auto*
from *1 2 4 5* **show** *?thesis* **by** *fastforce*
qed

lemma *FstAndDiEqvFst*:
⊢ (▷*f* ∧ *di f*) = ▷*f*
proof −
have *1*: ⊢ (▷*f* ∧ *di f*) = ((*f* ∧ *bs* (¬*f*)) ∧ *di f*) **by** (*simp add*: *first-d-def*)
have *2*: ⊢ (*f* ∧ *di f*) = *f* **by** (*meson DiIntro Prop10 Prop11*)
hence *3*: ⊢ (*f* ∧ *bs* (¬*f*) ∧ *di f*) = (*f* ∧ *bs* (¬*f*)) **by** *auto*
from *1 3* **show** *?thesis* **by** (*metis first-d-def int-iffD2 int-iffI Prop12*)
qed

lemma *DiEqvDiFst*:
⊢ *di f* = *di* (▷ *f*)
proof −
have *1*: ⊢ *di* (▷ *f*) = *di* (*f* ∧ *bs* (¬*f*))
by (*simp add*: *first-d-def*)
have *2*: ⊢ *di* (*f* ∧ *bs* (¬*f*)) ⟶ *di f* ∧ *di* (*bs* (¬ *f*))
using *DiAndImpAnd* **by** *auto*
hence *3*: ⊢ *di* (*f* ∧ *bs* (¬*f*)) ⟶ *di f*
by *auto*
have *4*: ⊢ *di* (▷ *f*) ⟶ *di f* **using** *1 3*
by *fastforce*
have *5*: ⊢ (*di f* ∧ *empty*) = (*f* ∧ *empty*)
using *DiAndEmptyEqvAndEmpty* **by** *blast*
have *6*: ⊢ (▷ *f* ∧ *empty*) = (*f* ∧ *empty*)
using *FstAndEmptyEqvAndEmpty* **by** *auto*
have *7*: ⊢ *di f* ∧ *empty* ⟶ ▷*f*
using *5 6* **by** *fastforce*
have *8*: ⊢ ▷*f* ⟶ *di* (▷ *f*)
using *DiIntro* **by** *auto*
have *9*: ⊢ *di f* ∧ *empty* ⟶ *di* (▷ *f*)
using *7 8* **using** *lift-imp-trans* **by** *blast*
hence *10*: ⊢ *empty* ⟶ (*di f* ⟶ *di* (▷ *f*))
by *auto*
have *11*: ⊢ *prev* (*di f* ⟶ *di* (▷ *f*))⟶ *more*
by (*simp add*: *ChopSkipImpMore prev-d-def*)
have *12*: ⊢ *more* ⟶(*prev* (*di f* ⟶ *di* (▷ *f*)) = (*prev*(*di f*) ⟶ *prev*(*di* (▷*f*))))
using *MoreImpImpPrevEqv* **by** *auto*
have *13*: ⊢ (*more* ∧ *prev* (*di f* ⟶ *di* (▷ *f*))) = (*more* ∧ (*prev*(*di f*) ⟶ *prev*(*di* (▷*f*))))
using *12* **by** *fastforce*
have *14*: ⊢ *prev* (*di f* ⟶ *di* (▷ *f*)) = (*more* ∧ (*prev*(*di f*) ⟶ *prev*(*di* (▷*f*))))
using *11 13* **by** *fastforce*
have *15*: ⊢ *di f* = (*f* ∨ *ds f*)
using *OrDsEqvDi* **by** *fastforce*
have *16*: ⊢ *di f* = (*di f* ∧ (*bs* (¬ *f*) ∨ ¬(*bs* (¬ *f*))))
by *auto*
have *17*: ⊢ (*di f* ∧ (*bs* (¬ *f*) ∨ ¬(*bs* (¬ *f*)))) = ((*di f* ∧ *bs* (¬ *f*)) ∨ (*di f* ∧ ¬(*bs* (¬ *f*))))
by *auto*

have *18*: $\vdash (\mathit{di}\ f \wedge \mathit{bs}\ (\neg f)) = ((f \vee \mathit{ds}\ f) \wedge \mathit{bs}\ (\neg f))$
  using *15* **by** *auto*
have *19*: $\vdash ((f \vee \mathit{ds}\ f) \wedge \mathit{bs}\ (\neg f)) = ((f \wedge \mathit{bs}\ (\neg f)) \vee (\mathit{ds}\ f \wedge \mathit{bs}\ (\neg f)))$
  by *auto*
have *20*: $\vdash \neg(\mathit{ds}\ f \wedge \mathit{bs}\ (\neg f))$
  by (*simp add*: *ds-d-def*)
have *21*: $\vdash ((f \wedge \mathit{bs}\ (\neg f)) \vee (\mathit{ds}\ f \wedge \mathit{bs}\ (\neg f))) = (f \wedge \mathit{bs}\ (\neg f))$
  using *20* **by** *auto*
have *22*: $\vdash (\mathit{di}\ f \wedge \mathit{bs}\ (\neg f)) = (f \wedge \mathit{bs}\ (\neg f))$
  using *18 19 21* **by** *fastforce*
have *23*: $\vdash (f \wedge \mathit{bs}\ (\neg f)) = \rhd f$
  by (*simp add*: *first-d-def*)
have *24*: $\vdash (\rhd f) \longrightarrow \mathit{di}\ (\rhd f)$
  using *DiIntro* **by** *auto*
have *25*: $\vdash (f \wedge \mathit{bs}\ (\neg f)) \longrightarrow \mathit{di}\ (\rhd f)$
  using *23 24* **by** *fastforce*
have *26*: $\vdash (\mathit{di}\ f \wedge \mathit{bs}\ (\neg f)) \longrightarrow \mathit{di}\ (\rhd f)$
  using *25 22* **by** *fastforce*
hence *27*: $\vdash (\mathit{di}\ f \wedge \mathit{bs}\ (\neg f) \wedge (\mathit{prev}\ (\mathit{di}\ f \longrightarrow \mathit{di}\ (\rhd f)))) \longrightarrow \mathit{di}\ (\rhd f)$
  by *auto*
have *28*: $\vdash (\mathit{di}\ f \wedge \neg(\mathit{bs}\ (\neg f))) = (\mathit{di}\ f \wedge \mathit{ds}\ f)$
  by (*simp add*: *ds-d-def*)
hence *29*: $\vdash (\mathit{di}\ f \wedge \neg(\mathit{bs}\ (\neg f)) \wedge (\mathit{prev}\ (\mathit{di}\ f \longrightarrow \mathit{di}\ (\rhd f)))) =$
    $(\mathit{di}\ f \wedge \mathit{ds}\ f \wedge (\mathit{prev}\ (\mathit{di}\ f \longrightarrow \mathit{di}\ (\rhd f))))$
  by *auto*
have *30*: $\vdash \mathit{ds}\ f = \mathit{prev}(\mathit{di}\ f)$
  using *DsDi* **by** (*metis prev-d-def*)
hence *31*: $\vdash (\mathit{di}\ f \wedge \mathit{ds}\ f \wedge (\mathit{prev}\ (\mathit{di}\ f \longrightarrow \mathit{di}\ (\rhd f)))) =$
    $(\mathit{di}\ f \wedge \mathit{prev}(\mathit{di}\ f) \wedge (\mathit{prev}\ (\mathit{di}\ f \longrightarrow \mathit{di}\ (\rhd f))))$
  by *auto*
have *32*: $\vdash \mathit{prev}\ (\ \mathit{di}\ f \longrightarrow \mathit{di}\ (\rhd f)) \longrightarrow (\mathit{prev}(\mathit{di}\ f) \longrightarrow \mathit{prev}(\mathit{di}\ (\rhd f)))$
  using *14* **by** *auto*
hence *33*: $\vdash \mathit{di}\ f \wedge \mathit{prev}(\mathit{di}\ f) \wedge \mathit{prev}\ (\ \mathit{di}\ f \longrightarrow \mathit{di}\ (\rhd f)) \longrightarrow$
    $\mathit{di}\ f \wedge \mathit{prev}(\mathit{di}\ f) \wedge (\mathit{prev}(\mathit{di}\ f) \longrightarrow \mathit{prev}(\mathit{di}\ (\rhd f)))$
  by *auto*
have *34*: $\vdash \mathit{di}\ f \wedge \mathit{prev}(\mathit{di}\ f) \wedge (\mathit{prev}(\mathit{di}\ f) \longrightarrow \mathit{prev}(\mathit{di}\ (\rhd f))) \longrightarrow \mathit{prev}(\mathit{di}\ (\rhd f))$
  by *auto*
have *35*: $\vdash \mathit{prev}(\mathit{di}\ (\rhd f)) = (\mathit{di}\ (\rhd f));\mathit{skip}$
  by (*simp add*: *prev-d-def*)
have *36*: $\vdash (\mathit{di}\ (\rhd f));\mathit{skip} \longrightarrow \mathit{di}(\mathit{di}\ (\rhd f))$
  using *ChopImpDi* **by** *auto*
have *37*: $\vdash \mathit{di}(\mathit{di}\ (\rhd f)) = \mathit{di}\ (\rhd f)$
  using *DiEqvDiDi* **by** *fastforce*
have *38*: $\vdash \mathit{di}\ f \wedge \mathit{prev}(\mathit{di}\ f) \wedge (\mathit{prev}(\mathit{di}\ f) \longrightarrow \mathit{prev}(\mathit{di}\ (\rhd f))) \longrightarrow \mathit{di}\ (\rhd f)$
  using *37 36 35 34* **by** *fastforce*
have *39*: $\vdash \mathit{di}\ f \wedge \neg(\mathit{bs}\ (\neg f)) \wedge (\mathit{prev}\ (\mathit{di}\ f \longrightarrow \mathit{di}\ (\rhd f))) \longrightarrow \mathit{di}\ (\rhd f)$
  using *29 31 33 38* **by** *fastforce*
hence *40*: $\vdash \neg(\mathit{bs}\ (\neg f)) \wedge (\mathit{prev}\ (\mathit{di}\ f \longrightarrow \mathit{di}\ (\rhd f))) \longrightarrow (\mathit{di}\ f \longrightarrow \mathit{di}\ (\rhd f))$
  by *fastforce*
have *41*: $\vdash \mathit{bs}\ (\neg f) \wedge (\mathit{prev}\ (\mathit{di}\ f \longrightarrow \mathit{di}\ (\rhd f))) \longrightarrow (\mathit{di}\ f \longrightarrow \mathit{di}\ (\rhd f))$

using *27* **by** *fastforce*
have *42*: $\vdash (\neg(bs\ (\neg\ f)) \vee bs\ (\neg\ f)) \wedge (prev\ (di\ f\ \longrightarrow\ di\ (\rhd\ f))) \longrightarrow (di\ f\ \longrightarrow\ di\ (\rhd f))$
using *40 41* **by** *fastforce*
have *43*: $\vdash (\neg(bs\ (\neg\ f)) \vee bs\ (\neg\ f))$
by *auto*
have *44*: $\vdash (prev\ (di\ f\ \longrightarrow\ di\ (\rhd\ f))) \longrightarrow (di\ f\ \longrightarrow\ di\ (\rhd f))$
using *42 43* **by** *fastforce*
have *45*: $\vdash di\ f\ \longrightarrow\ di\ (\rhd f)$
using *10 44 EmptyChopSkipInduct* **by** *blast*
from *4 45* **show** *?thesis* **by** *fastforce*
qed

lemma *FstDiEqvFst*:
$\vdash \rhd(di\ f) = \rhd f$
proof –
have *1*: $\vdash \rhd(di\ f) = (di\ f \wedge bs\ (\neg\ (di\ f)))$ **by** (*simp add*: *first-d-def*)
have *2*: $\vdash\ (\neg\ (di\ f)) = bi\ (\neg\ f)$ **by** (*simp add*: *NotDiEqvBiNot*)
hence *3*: $\vdash bs\ (\neg\ (di\ f)) = bs\ (bi\ (\neg\ f))$ **using** *BsEqvRule* **by** *blast*
have *4*: $\vdash bs\ (bi\ (\neg\ f)) = bs\ (\ \neg\ f)$ **using** *BsEqvBsBi* **by** *fastforce*
hence *5*: $\vdash (di\ f \wedge bs\ (\neg\ (di\ f))) = (di\ f \wedge bs\ (\ \neg\ f))$ **using** *3* **by** *fastforce*
have *6* : $\vdash di\ f = (f \vee ds\ f)$ **using** *OrDsEqvDi* **by** *fastforce*
hence *7*: $\vdash (di\ f \wedge bs\ (\ \neg\ f)) = ((f \vee ds\ f) \wedge bs\ (\ \neg\ f))$ **by** *auto*
have *8*: $\vdash ((f \vee ds\ f) \wedge bs\ (\ \neg\ f)) = ((f \wedge bs\ (\ \neg\ f)) \vee (ds\ f \wedge bs\ (\ \neg\ f)))$ **by** *auto*
have *9*: $\vdash \neg(ds\ f \wedge bs\ (\ \neg\ f))$ **by** (*simp add*: *ds-d-def*)
have *10*: $\vdash (f \wedge bs\ (\ \neg\ f)) = \rhd f$ **by** (*simp add*: *first-d-def*)
have *11*: $\vdash ((f \wedge bs\ (\ \neg\ f)) \vee (ds\ f \wedge bs\ (\ \neg\ f))) = \rhd f$ **using** *9 10* **by** *fastforce*
from *1 5 7 8 11* **show** *?thesis* **by** (*metis int-eq*)
qed

lemma *DiAndFstOrEqvFstOrDiAnd*:
$\vdash (di\ f \wedge (\rhd f \vee g)) = (\rhd f \vee (di\ f \wedge g))$
proof –
have *1*: $\vdash (di\ f \wedge (\rhd f \vee g)) = (\rhd f \wedge di\ f) \vee (di\ f \wedge g)$ **by** *auto*
have *2*: $\vdash (\rhd f \wedge di\ f) = \rhd f$ **using** *FstAndDiEqvFst* **by** *blast*
from *1 2* **show** *?thesis* **by** *auto*
qed

lemma *DiOrFstAndEqvDi*:
$\vdash di\ f \vee (\rhd f \wedge g) = di\ f$
proof –
have *1*: $\vdash (di\ f \vee (\rhd f \wedge g)) = ((\rhd f \vee di\ f) \wedge (di\ f \vee g))$ **by** *auto*
have *2*: $\vdash (\rhd f \vee di\ f) = di\ f$ **using** *FstOrDiEqvDi* **by** *blast*
from *1 2* **show** *?thesis* **by** *auto*
qed

lemma *FstDiAndDiEqv*:
$\vdash \rhd(di\ f \wedge di\ g) = ((\rhd f \wedge di\ g) \vee (\rhd g \wedge di\ f))$
proof –
have *1*: $\vdash \rhd(di\ f \wedge di\ g) = ((di\ f \wedge di\ g) \wedge bs\ (\neg(di\ f \wedge di\ g)))$ **by** (*simp add*: *first-d-def*)
have *2*: $\vdash (\neg(di\ f \wedge di\ g)) = (bi\ (\neg f) \vee bi\ (\neg g))$ **by** (*simp add*: *bi-d-def*, *auto*)

```
 hence 3: ⊢ bs (¬(di f ∧ di g)) = bs(bi (¬f) ∨ bi (¬g)) using BsEqvRule by blast
 hence 4: ⊢  ((di f ∧ di g) ∧ bs (¬(di f ∧ di g))) =
            (di f ∧ di g ∧ bs(bi (¬f) ∨ bi (¬g)))  by auto
 have 5: ⊢ (bs (¬f) ∨ bs (¬g)) = bs(bi (¬f) ∨ bi (¬g)) using BsOrBsEqvBsBiOrBi by blast
 hence 6: ⊢ (di f ∧ di g ∧ bs(bi (¬f) ∨ bi (¬g))) =
           (di f ∧ di g ∧(bs (¬f) ∨ bs (¬g))) by auto
 have 7: ⊢ (di f ∧ di g ∧(bs (¬f) ∨ bs (¬g))) =
          ((  bs (¬f) ∧ di f ∧ di g) ∨ (di f ∧ bs (¬g) ∧ di g  )) by auto
 have 8: ⊢ ▷f =  (bs (¬f)  ∧ di f) using FstEqvBsNotAndDi by blast
 hence 9: ⊢ (bs (¬f) ∧ di f ∧ di g) = (▷f ∧ di g) by auto
 have 10: ⊢ ▷g =  (bs (¬g)  ∧ di g) using FstEqvBsNotAndDi by blast
 hence 11: ⊢ (di f ∧ bs (¬g) ∧ di g) = (di f ∧ ▷g) by auto
 have 12: ⊢ (di f ∧ di g ∧(bs (¬f) ∨ bs (¬g))) =
           ((▷f ∧ di g) ∨ (di f ∧ ▷g)) using 7 9 11 by (metis int-eq)
 from 1 4 6 12 show ?thesis using inteq-reflection lift-and-com by fastforce
qed

lemma BiNotFstEqvBiNot:
⊢ bi (¬ (▷f)) = bi (¬ f)
proof −
 have 1: ⊢ di f = di (▷f) using DiEqvDiFst by blast
 hence 2: ⊢ (¬(di f)) = (¬( di (▷f))) by auto
 from 1 2 show ?thesis using NotDiEqvBiNot by fastforce
qed

lemma BsNotFstEqvBsNot:
⊢ bs (¬ (▷f)) = bs (¬ f)
proof −
 have 1: ⊢ bs (¬ (▷f)) = (empty ∨ bi (¬ (▷f));skip)  by (simp add: bs-d-def)
 have 2: ⊢ bi (¬ (▷f)) = bi (¬ f)  using BiNotFstEqvBiNot by blast
 hence 3: ⊢ bi (¬ (▷f));skip = bi (¬ f);skip using LeftChopEqvChop by blast
 hence 4: ⊢ (empty ∨ bi (¬ (▷f));skip) = (empty ∨ bi (¬ f);skip) by auto
 from 1 4 show ?thesis by (simp add: bs-d-def)
qed

lemma FstState:
⊢ ▷ (init w) = (empty ∧ init w)
proof −
 have 1: ⊢ ▷ (init w) = (init w ∧ bs (¬(init w))) by (simp add: first-d-def)
 hence 2: ⊢ ▷ (init w) ⟶ init w by auto
 have 3: ⊢ init w ⟶ bs (init w) using StateImpBs by auto
 have 4: ⊢ ▷ (init w) ⟶ bs (init w) using 2 3 by fastforce
 have 5: ⊢ ▷ (init w) ⟶ bs (¬(init w)) using 1 by auto
 have 6: ⊢ ▷ (init w) ⟶ bs (init w) ∧ bs (¬(init w))  using 4 5 by fastforce
 have 7: ⊢ (bs (init w) ∧ bs (¬(init w))) = (bs((init w) ∧ ¬(init w)))  using BsAndEqv by blast
 have 8: ⊢ ((init w) ∧ ¬(init w)) =  #False  by auto
 hence 9: ⊢ (bs((init w) ∧ ¬(init w))) =  bs #False using BsEqvRule by blast
 have 10: ⊢ bs #False = empty using BsFalseEqvEmpty by auto
 have 11: ⊢ ▷ (init w) ⟶ empty using 10 9 7 6 by fastforce
 have 12: ⊢ ▷ (init w) ⟶ empty ∧ init w using 11 2 by fastforce
 have 13: ⊢ empty ∧ init w ⟶ empty by auto
 hence 14: ⊢ empty ∧ init w ⟶ empty ∨ bi (¬(init w));skip by auto
 hence 15: ⊢ empty ∧ init w ⟶ bs (¬(init w)) by (simp add: bs-d-def)
 have 16: ⊢ empty ∧ init w ⟶ init w  by auto
 have 17: ⊢ empty ∧ init w ⟶ init w ∧ bs (¬(init w)) using 16 15 by auto
 hence 18: ⊢ empty ∧ init w ⟶ ▷(init w) by (simp add: first-d-def)
 from 12 18 show ?thesis by fastforce
qed

lemma FstStateAndBsNotEmpty:
⊢ (▷ (init w) ∧ bs (¬ empty)) = ▷ (init w)
proof −
 have 1: ⊢ (▷ (init w) ∧ bs (¬ empty)) = (▷ (init w) ∧ bs more)
     using BsEqvRule NotEmptyEqvMore by (simp add: empty-d-def)
 have 2: ⊢ (▷ (init w) ∧ bs more) = (▷ (init w) ∧ empty)
     using BsMoreEqvEmpty by fastforce
 have 3: ⊢ ▷ (init w)  =  (empty ∧ (init w))
     using FstState by blast
 hence 4: ⊢ (▷ (init w)  ∧ empty) =  (empty ∧ (init w)  ∧ empty)
     by auto
 have 5: ⊢ (empty ∧ (init w)  ∧ empty) = (empty ∧ (init w))
     by auto
 have 6: ⊢ (empty ∧ (init w)) = ▷(init w)
     using FstState by fastforce
 from 1 2 4 5 6 show ?thesis by fastforce
qed

lemma FstStateImpFstStateOr:
⊢ ▷(init w) ⟶ ▷(init w ∨ f)
proof −
 have 1: ⊢ ▷(init w) = (empty ∧ init w)
     using FstState by blast
 have 2: ⊢  (empty ∧ init w) =  (empty ∧ (empty ∨ bi (¬ f);skip) ∧ init w)
     by auto
 have 3: ⊢  (empty ∧ (empty ∨ bi (¬ f);skip) ∧ init w) =
           (empty ∧ bs (¬ f) ∧ init w)
     by (simp add: bs-d-def)
 have 4: ⊢  (empty ∧ bs (¬ f) ∧ init w) = (empty ∧ init w ∧ bs (¬ f))
     by auto
 have 5: ⊢ (empty ∧ init w)  = ▷ (init w)
     using FstState  by fastforce
 hence 6: ⊢ (empty ∧ init w ∧ bs (¬ f)) = (▷ (init w) ∧ bs (¬ f))
     by auto
 have 7: ⊢ ▷ (init w) ∧ bs (¬ f) ⟶ (▷ (init w) ∧ bs (¬ f)) ∨ (▷f ∧ bs (¬(init w)))
     by auto
 have 8: ⊢ ▷(init w ∨ f) = ((▷ (init w) ∧ bs (¬ f)) ∨ (▷f ∧ bs (¬(init w))))
     using FstWithOrEqv by blast
 from 1 2 3 4 5 6 7 8 show ?thesis by fastforce
qed
```

```
lemma FstLenSame:
  (∀ σ. (σ ⊨ di (▷f ∧ len(i)) ∧ di (▷f ∧ len(j))) ⟶ (i=j))
by (simp add: DiLenFstsem FstLenSamesem)

lemma FstLenSame-1:
  ⊢ di (▷f ∧ len(i)) ∧ di (▷f ∧ len(j)) ⟶ (#i=#j)
using FstLenSame Valid-def by fastforce

lemma FstAndLenSame:
  (∀ σ. (σ ⊨ di ((▷f ∧ g1) ∧ len(i)) ∧ di ((▷f ∧ g2) ∧ len(j))) ⟶ (i=j))
apply (simp  add: DiLenFstAndsem)
using linorder-neqE-nat by blast

lemma FstAndLenSame-1:
  ⊢ di ((▷f ∧ g1) ∧ len(i)) ∧ di ((▷f ∧ g2) ∧ len(j)) ⟶ (#i=#j)
using FstAndLenSame Valid-def by fastforce

lemma FstLenSameChop:
  (∀ σ. (σ ⊨ ((▷f ∧ g1) ∧ len(i));h1 ∧ ((▷f ∧ g2) ∧ len(j));h2) ⟶ (i=j))
proof
  fix σ
  show (σ ⊨ ((▷f ∧ g1) ∧ len(i));h1 ∧ ((▷f ∧ g2) ∧ len(j));h2) ⟶ (i=j)
  proof
  assume 0: (σ ⊨ ((▷f ∧ g1) ∧ len(i));h1 ∧ ((▷f ∧ g2) ∧ len(j));h2)
   have 1: (σ ⊨ ((▷f ∧ g1) ∧ len(i));h1) using 0 by auto
   have 2: (σ ⊨ ((▷f ∧ g1) ∧ len(i));h1) ⟶
          (σ ⊨ ((▷f ∧ g1) ∧ len(i));#True) by (metis ChopImpDi Valid-def di-d-def unl-lift2)
   have 3: (σ ⊨ di((▷f ∧ g1) ∧ len(i))) using 1 2 by (simp add: di-d-def)
   have 4: (σ ⊨ ((▷f ∧ g2) ∧ len(j));h2) using 0 by auto
   have 5: (σ ⊨ ((▷f ∧ g2) ∧ len(j));h2) ⟶
          (σ ⊨ ((▷f ∧ g2) ∧ len(j));#True) by (metis ChopImpDi Valid-def di-d-def unl-lift2)
   have 6: (σ ⊨ di((▷f ∧ g2) ∧ len(j))) using 4 5 by (simp add: di-d-def)
   have 7: (σ ⊨ di((▷f ∧ g1) ∧ len(i)) ∧ di((▷f ∧ g2) ∧ len(j))) using 3 6 by auto
   thus (i=j) using FstAndLenSame by blast
   qed
qed

lemma FstLenSameChop-1:
  ⊢ ((▷f ∧ g1) ∧ len(i));h1 ∧ ((▷f ∧ g2) ∧ len(j));h2 ⟶ (#i=#j)
using FstLenSameChop Valid-def by fastforce

lemma DiImpExistsOneDiLenAndFst:
 (∀σ. (σ⊨ di f) ⟶ (∃! k. (σ ⊨ di( ▷f ∧ len(k)))))
proof
 fix σ
 show (σ⊨ di f) ⟶ (∃! k. (σ ⊨ di( ▷f ∧ len(k))))
 proof
  assume 0: (σ⊨ di f)
  have 1: (σ ⊨ di (▷ f))
```

```
      using 0 DiEqvDiFst Valid-def by force
  have 2: (σ ⊨ ▷ f) = ( (σ⊨▷ f) ∧ (∃ k. (σ ⊨ len(k))))
      using AndExistsLen[of TEMP ▷ f] by (simp add: Valid-def)
  have 3: ((σ⊨▷ f) ∧ (∃ k. (σ ⊨ len(k)))) =
          (∃ k.  (σ⊨▷ f) ∧  (σ ⊨ len(k)))
      by auto
  have 4: (σ ⊨ di(▷ f)) = (∃ k. (σ ⊨ di(▷f ∧ len(k))))
      using 2 3 by (metis 1 DiLensem di-defs)
  have 5: (∃ k. (σ ⊨ di(▷f ∧ len(k))))
      using 1 using 4 by auto
  then obtain i where 6: (σ ⊨ di(▷f ∧ len(i)))  by blast
  from 5 obtain j where 7: (σ ⊨ di(▷f ∧ len(j)))  by blast
  have 8: (σ ⊨ di(▷f ∧ len(i))) ∧ (σ ⊨ di(▷f ∧ len(j)))
      using 6 7 by auto
  hence 9: (σ ⊨ di(▷f ∧ len(i)) ∧ di(▷f ∧ len(j)))
      by simp
  hence 10: i=j
      using FstLenSame by blast
  have 11: ⋀j. (σ ⊨ di(▷f ∧ len(j))) ⟶ (j=i)
      using 9 10 using FstLenSame by auto
  thus (∃! k. (σ ⊨ di(  ▷f ∧ len(k) )))
      using 11 5 by blast
  qed
qed

lemma DiImpExistsOneDiLenAndFst-1:
 ⊢ di f ⟶ (∃! k. ( di( ▷f ∧ len(k))))
using Valid-def DiImpExistsOneDiLenAndFst by fastforce

lemma LFstAndDist-help:
 (σ ⊨((▷f ∧ g1) ∧ len(k));h1  ∧ ((▷f ∧ g2) ∧ len(k));h2) =
  (σ ⊨(((▷f ∧ g1) ∧ (▷f ∧ g2)) ∧ len(k));(h1 ∧ h2) )
using LFixedAndDistr by fastforce

lemma LFstAndDist-help-1:
 (∃  k. (σ ⊨((▷f ∧ g1) ∧ len(k));h1  ∧ ((▷f ∧ g2) ∧ len(k));h2)) =
          (∃  k. (σ ⊨(((▷f ∧ g1) ∧ (▷f ∧ g2)) ∧ len(k));(h1 ∧ h2) ))
proof
  assume 0: ∃ k. σ ⊨  ((▷ f ∧ g1) ∧ len k) ; h1  ∧  ((▷ f ∧ g2) ∧ len k) ; h2
  obtain k where 1:  σ ⊨  ((▷ f ∧ g1) ∧ len k) ; h1  ∧  ((▷ f ∧ g2) ∧ len k) ; h2
  using 0 by auto
  hence 2: (σ ⊨(((▷f ∧ g1) ∧ (▷f ∧ g2)) ∧ len(k));(h1 ∧ h2))
  using  LFstAndDist-help by blast
  show (∃  k. (σ ⊨(((▷f ∧ g1) ∧ (▷f ∧ g2)) ∧ len(k));(h1 ∧ h2) ))
  using 2 by auto
  next
  assume 3: (∃  k. (σ ⊨(((▷f ∧ g1) ∧ (▷f ∧ g2)) ∧ len(k));(h1 ∧ h2) ))
  obtain k where 4: (σ ⊨(((▷f ∧ g1) ∧ (▷f ∧ g2)) ∧ len(k));(h1 ∧ h2) )
  using 3 by auto
  hence 5: (σ ⊨((▷f ∧ g1) ∧ len(k));h1  ∧ ((▷f ∧ g2) ∧ len(k));h2)
  using  LFstAndDist-help by blast
  show (∃ k. (σ ⊨((▷f ∧ g1) ∧ len(k));h1  ∧ ((▷f ∧ g2) ∧ len(k));h2))
  using 5 by auto
qed

lemma LFstAndDistrsem:
 (∀ σ. (σ ⊨((▷f ∧ g1);h1 ∧ (▷f ∧ g2);h2) = (▷f ∧ g1 ∧ g2);(h1 ∧ h2)))
proof
 fix σ
 show (σ ⊨((▷f ∧ g1);h1 ∧ (▷f ∧ g2);h2) = (▷f ∧ g1 ∧ g2);(h1 ∧ h2))
 proof −
  have 1: (σ ⊨(▷f ∧ g1);h1) = (∃ i. (σ ⊨((▷f ∧ g1) ∧ len(i));h1) )
  using AndExistsLenChop[of TEMP (▷ f ∧ g1)] by fastforce
  have 2: (σ ⊨(▷f ∧ g2);h2) = (∃ j. (σ ⊨((▷f ∧ g2) ∧ len(j));h2) )
  using AndExistsLenChop[of TEMP (▷f ∧ g2)] by fastforce
  have 3: (σ ⊨(▷f ∧ g1);h1 ∧ (▷f ∧ g2);h2) =
        ( (∃ i j. (σ ⊨((▷f ∧ g1) ∧ len(i));h1  ∧
                        ((▷f ∧ g2) ∧ len(j));h2) )
        )
  using 1 2 by auto
  have 4: ( (∃ i j. (σ ⊨((▷f ∧ g1) ∧ len(i));h1  ∧
                        ((▷f ∧ g2) ∧ len(j));h2) )
        ) =
        ( (∃ k. (σ ⊨((▷f ∧ g1) ∧ len(k));h1  ∧
                        ((▷f ∧ g2) ∧ len(k));h2) )
        )
  using FstLenSameChop by blast
  have 5: (∃ k. (σ ⊨((▷f ∧ g1) ∧ len(k));h1  ∧ ((▷f ∧ g2) ∧ len(k));h2)) =
          (∃ k. (σ ⊨(((▷f ∧ g1) ∧ (▷f ∧ g2)) ∧ len(k));(h1 ∧ h2) ))
  using LFstAndDist-help-1 by blast
  have 6 :  (∃ k. (σ ⊨(((▷f ∧ g1) ∧ (▷f ∧ g2)) ∧ len(k));(h1 ∧ h2) )) =
           (σ ⊨((▷f ∧ g1) ∧ (▷f ∧ g2));(h1 ∧ h2))
  using AndExistsLenChop[of TEMP ((▷ f ∧ g1) ∧ ▷ f ∧ g2)] by fastforce
  have 7 : (σ ⊨((▷f ∧ g1) ∧ (▷f ∧ g2));(h1 ∧ h2)) =
          (σ ⊨(▷f ∧ g1 ∧ g2);(h1 ∧ h2))
  by (simp add: chop-defs, auto)
  from 3 4 5 6 7 show ?thesis by auto
  qed
qed

lemma LFstAndDistr:
 ⊢ ((▷f ∧ g1);h1 ∧ (▷f ∧ g2);h2) = (▷f ∧ g1 ∧ g2);(h1 ∧ h2)
using  LFstAndDistrsem by fastforce

lemma LFstAndDistrA:
 ⊢ ((▷f ∧ g1);h ∧ (▷f ∧ g2);h) = (▷f ∧ g1 ∧ g2);h
proof −
 have 1: ⊢ ((▷f ∧ g1);h ∧ (▷f ∧ g2);h) = (▷f ∧ g1 ∧ g2);(h ∧ h)  using LFstAndDistr by blast
 have 2: ⊢ (▷f ∧ g1 ∧ g2);(h ∧ h) = (▷f ∧ g1 ∧ g2);h  by auto
 from 1 2 show ?thesis by auto
```

qed

lemma *LFstAndDistrB*:
$\vdash ((\rhd f \wedge g);h1 \wedge (\rhd f \wedge g);h2) = (\rhd f \wedge g);(h1 \wedge h2)$
proof –
have *1*: $\vdash ((\rhd f \wedge g);h1 \wedge (\rhd f \wedge g);h2) = (\rhd f \wedge g \wedge g);(h1 \wedge h2)$ **using** *LFstAndDistr* **by** *blast*
have *2*: $\vdash (\rhd f \wedge g \wedge g);(h1 \wedge h2) = (\rhd f \wedge g);(h1 \wedge h2)$ **by** *auto*
from *1 2* **show** *?thesis* **by** *auto*
qed

lemma *LFstAndDistrC*:
$\vdash ((\rhd f\ );h1 \wedge (\rhd f\ );h2) = (\rhd f);(h1 \wedge h2)$
proof –
have *1*: $\vdash ((\rhd f \wedge \#True);h1 \wedge (\rhd f \wedge \#True);h2) = (\rhd f \wedge \#True \wedge \#True);(h1 \wedge h2)$
using *LFstAndDistr* **by** *blast*
have *2*: $\vdash (\rhd f \wedge \#True);h1 = (\rhd f\ );h1$
by *auto*
have *3*: $\vdash (\rhd f \wedge \#True);h2 = (\rhd f\ );h2$
by *auto*
have *4*: $\vdash (\rhd f \wedge \#True \wedge \#True);(h1 \wedge h2) = (\rhd f);(h1 \wedge h2)$
by *auto*
from *1 2 3 4* **show** *?thesis* **by** *auto*
qed

lemma *LFstAndDistrD*:
$\vdash (di(\rhd f \wedge g1) \wedge di(\rhd f \wedge g2)) = di(\rhd f \wedge g1 \wedge g2)$
proof –
have *1*: $\vdash ((\rhd f \wedge g1);\#True \wedge (\rhd f \wedge g2);\#True) = (\rhd f \wedge g1 \wedge g2);(\#True \wedge \#True)$
using *LFstAndDistr* **by** *blast*
have *2*: $\vdash (\rhd f \wedge g1);\#True = di(\rhd f \wedge g1)$
by (*simp add*: *di-d-def*)
have *3*: $\vdash (\rhd f \wedge g2);\#True = di(\rhd f \wedge g2)$
by (*simp add*: *di-d-def*)
have *4*: $\vdash (\rhd f \wedge g1 \wedge g2);(\#True \wedge \#True) = di(\rhd f \wedge g1 \wedge g2)$
by (*simp add*: *di-d-def*)
from *1 2 3 4* **show** *?thesis* **by** *fastforce*
qed

lemma *LstAndDistr*:
$\vdash (h1;(\lhd f \wedge g1) \wedge h2;(\lhd f \wedge g2)) = (h1 \wedge h2);(\lhd f \wedge g1 \wedge g2)$
proof –
have *1*: $\vdash ((\rhd(f^r) \wedge g1^r);(h1^r) \wedge (\rhd(f^r) \wedge (g2^r));(h2^r)) =$
$(\rhd(f^r) \wedge (g1^r) \wedge (g2^r));((h1^r) \wedge (h2^r))$
using *LFstAndDistr* **by** *blast*
hence *2*: $\vdash ((\rhd(f^r) \wedge g1^r);(h1^r) \wedge (\rhd(f^r) \wedge (g2^r));(h2^r))^r =$
$((\rhd(f^r) \wedge (g1^r) \wedge (g2^r));((h1^r) \wedge (h2^r)))^r$
using *1 REqvRule* **by** *blast*
have *3*: $\vdash (((\rhd(f^r) \wedge g1^r);(h1^r))^r \wedge ((\rhd(f^r) \wedge (g2^r));(h2^r))^r) =$
$((\rhd(f^r) \wedge g1^r);(h1^r) \wedge (\rhd(f^r) \wedge (g2^r));(h2^r))^r$

using *RAnd* **by** *fastforce*
have *4*: $\vdash ((h1^r)^r;(\rhd(f^r) \wedge g1^r)^r \wedge (h2^r)^r;(\rhd(f^r) \wedge (g2^r))^r) =$
$(((\rhd(f^r) \wedge g1^r);(h1^r))^r \wedge ((\rhd(f^r) \wedge (g2^r));(h2^r))^r)$

using *RevChop* **by** *fastforce*
have *5*: $\vdash (h1^r)^r = h1$
using *EqvReverseReverse* **by** *blast*
have *6*: $\vdash (h2^r)^r = h2$
using *EqvReverseReverse* **by** *blast*
have *7*: $\vdash (g1^r)^r = g1$
using *EqvReverseReverse* **by** *blast*
have *8*: $\vdash (g2^r)^r = g2$
using *EqvReverseReverse* **by** *blast*
have *9*: $\vdash (f^r)^r = f$
using *EqvReverseReverse* **by** *blast*
have *10*: $\vdash (\rhd(f^r) \wedge g1^r)^r = ((\rhd(f^r))^r \wedge (g1^r)^r)$
using *RAnd* **by** *blast*
have *11*: $\vdash (\rhd(f^r) \wedge g2^r)^r = ((\rhd(f^r))^r \wedge (g2^r)^r)$
using *RAnd* **by** *blast*
have *12*: $\vdash (\rhd(f^r))^r = \lhd (f)$
using *RRFirstEqvLast* **by** *blast*
have *13*: $\vdash ((\rhd(f^r))^r \wedge (g1^r)^r) = (\lhd f \wedge g1)$
using *12 7* **by** *fastforce*
have *14*: $\vdash ((\rhd(f^r))^r \wedge (g2^r)^r) = (\lhd f \wedge g2)$
using *12 8* **by** *fastforce*
have *15*: $\vdash (h1;(\lhd f \wedge g1) \wedge h2;(\lhd f \wedge g2)) =$
$((h1^r)^r;(\rhd(f^r) \wedge g1^r)^r \wedge (h2^r)^r;(\rhd(f^r) \wedge (g2^r))^r)$

using *14 13 10 11 5 6* **by** (*metis 4 int-eq*)
have *16*: $\vdash (((\rhd(f^r) \wedge (g1^r) \wedge (g2^r));((h1^r) \wedge (h2^r)))^r) =$
$((h1^r) \wedge (h2^r))^r;((\rhd (f^r)) \wedge (g1^r) \wedge (g2^r))^r$
by (*simp add*: *RevChop*)
have *17*: $\vdash ((\rhd (f^r)) \wedge (g1^r) \wedge (g2)^r)^r = ((\rhd (f^r))^r \wedge (g1^r)^r \wedge (g2^r)^r)$
by (*metis inteq-reflection rev-fun2*)
have *18*: $\vdash ((\rhd (f^r))^r \wedge (g1^r)^r \wedge (g2^r)^r) = (\lhd f \wedge g1 \wedge g2)$
using *12 7 8* **by** *fastforce*
have *19*: $\vdash ((h1^r) \wedge (h2^r))^r = (h1 \wedge h2)$
using *RRAnd* **by** *auto*
have *20*: $\vdash ((h1^r) \wedge (h2^r))^r;((\rhd (f^r)) \wedge (g1^r) \wedge (g2^r))^r =$
$(h1 \wedge h2);(\lhd f \wedge g1 \wedge g2)$
using *19 17 18* **using** *ChopEqvChop* **by** (*metis int-eq*)
from *15 4 3 2 16 20* **show** *?thesis* **using** *int-eq* **by** *metis*
qed

lemma *LstAndDistrA*:
$\vdash (h;(\lhd f \wedge g1) \wedge h;(\lhd f \wedge g2)) = h;(\lhd f \wedge g1 \wedge g2)$
proof –
have *1*: $\vdash (h;(\lhd f \wedge g1) \wedge h;(\lhd f \wedge g2)) = (h \wedge h);(\lhd f \wedge g1 \wedge g2)$
using *LstAndDistr* **by** *blast*
have *2*: $\vdash (h \wedge h);(\lhd f \wedge g1 \wedge g2) = h;(\lhd f \wedge g1 \wedge g2)$

```
     by auto
 from 1 2 show ?thesis by auto
qed

lemma LstAndDistrB:
⊢ (h1;(◁ f ∧ g) ∧ h2;(◁ f ∧ g)) = (h1 ∧ h2);(◁ f ∧ g)
proof –
 have 1: ⊢ (h1;(◁ f ∧ g) ∧ h2;(◁ f ∧ g)) = (h1 ∧ h2);(◁ f ∧ g ∧ g)
     using LstAndDistr by blast
 have 2: ⊢ (h1 ∧ h2);(◁ f ∧ g ∧ g) = (h1 ∧ h2);(◁ f ∧ g)
     by auto
 from 1 2 show ?thesis by auto
qed

lemma LstAndDistrC:
⊢ (h1;(◁ f ) ∧ h2;(◁ f )) = (h1 ∧ h2);(◁ f )
proof –
 have 1: ⊢ (h1;(◁ f ∧ #True) ∧ h2;(◁ f ∧ #True)) = (h1 ∧ h2);(◁ f ∧ #True ∧ #True  )
     using LstAndDistr by blast
 have 2: ⊢ (h1 ∧ h2);(◁ f ∧ #True ∧ #True  ) = (h1 ∧ h2);(◁ f )
     by auto
 have 3: ⊢ h1;(◁ f ∧ #True) = h1;(◁ f )
     by auto
 have 4: ⊢ h2;(◁ f ∧ #True) = h2;(◁ f )
     by auto
 from 1 2 3 4 show ?thesis by auto
qed

lemma LstAndDistrD:
⊢ (◇(◁ f ∧ g1) ∧ ◇(◁ f ∧ g2)) = ◇(◁ f ∧ g1 ∧ g2)
proof –
 have 1: ⊢ (#True;(◁ f ∧ g1) ∧ #True;(◁ f ∧ g2)) = (#True ∧ #True );(◁ f ∧ g1 ∧ g2)
     using LstAndDistr by blast
 have 2: ⊢ (#True ∧ #True );(◁ f ∧ g1 ∧ g2) = ◇(◁ f ∧ g1 ∧ g2)
     by (simp add: sometimes-d-def)
 have 3: ⊢ #True;(◁ f ∧ g1) = ◇(◁ f ∧ g1)
     by (simp add: sometimes-d-def)
 have 4: ⊢ #True;(◁ f ∧ g2) = ◇(◁ f ∧ g2)
     by (simp add: sometimes-d-def)
 from 1 2 3 4 show ?thesis by fastforce
qed

lemma NotFstChop:
⊢ (¬(▷f ;g)) = (¬(di (▷f)) ∨ (▷f;(¬g)))
proof –
 have  1: ⊢ g ⟶ #True by auto
 hence 2: ⊢ ▷f;g ⟶ ▷f;#True using RightChopImpChop by blast
 hence 3: ⊢ ▷f;g ⟶ di(▷f) by (simp add:di-d-def)
 hence 4: ⊢ ¬(di(▷f)) ⟶ ¬(▷f;g) by auto
 have  5: ⊢ (▷f; (¬g) ⟶ ¬(▷f;g)) = ((▷f; (¬g)) ∧ (▷f;g) ⟶ #False ) by auto
 have 6: ⊢ ((▷f; (¬g)) ∧ (▷f;g)) = ▷f;(¬g ∧ g)  using LFstAndDistrC by blast
 have 7: ⊢ ¬(▷f;(¬g ∧ g))  by (simp add: FstChopFalseEqvFalse)
 have 8: ⊢ ▷f; (¬g) ⟶ ¬(▷f;g)  using 5 6 7 by fastforce
 have 9: ⊢ ¬(di(▷f)) ∨ (▷f; (¬g)) ⟶ ¬(▷f;g)  using 4 8 by fastforce
 have 10: ⊢ di(▷f) ∨ ¬(di(▷f)) by auto
 hence 11: ⊢ (▷f;#True) ∨ ¬(di(▷f))  by (simp add: di-d-def)
 hence 12: ⊢ (▷f;(g ∨ ¬g)) ∨ ¬(di(▷f)) by  auto
 have 13: ⊢ (▷f;(g ∨ ¬g)) = ((▷f;g) ∨ (▷f;(¬g)))    using ChopOrEqv by fastforce
 have 14: ⊢ ((▷f;g) ∨ (▷f;(¬g))) ∨ ¬(di(▷f))   using 12 13 by fastforce
 hence 15: ⊢ ¬(▷f;g) ⟶ ¬(di(▷f)) ∨ (▷f; (¬g))  by auto
 from 9 15 show ?thesis by fastforce
qed

lemma BsNotFstChop:
 ⊢ bs(¬(▷f;g)) = (empty ∨ ¬(di(▷f)) ∨ (▷f;bs(¬g)))
proof −
 have 1: ⊢ bs(¬(▷f;g))= (empty ∨ bi(¬(▷f;g));skip)
     by (simp add:bs-d-def)
 have 2: ⊢ (empty ∨ bi(¬(▷f;g));skip) = (empty ∨ (¬(di(▷f;g)));skip)
     by (metis 1 NotDiEqvBiNot int-eq)
 have 3: ⊢ (empty ∨ (¬(di(▷f;g)));skip) = (empty ∨ (¬((▷f;g);#True));skip)
     by (simp add: di-d-def)
 have 4: ⊢  (¬((▷f;g);#True));skip =  (¬(▷f;(g;#True)));skip
     by (metis ChopAssocB LeftChopEqvChop int-simps(15) inteq-reflection)
 hence 5: ⊢ (empty ∨ (¬((▷f;g);#True));skip) =  (empty ∨ (¬(▷f;(g;#True)));skip)
     by auto
 have 6: ⊢ (empty ∨ (¬(▷f;(g;#True)));skip) = (empty ∨ (¬(▷f;di(g)));skip)
     by (simp add: di-d-def)
 have 7: ⊢ (empty ∨ (¬(▷f;di(g)));skip) = (empty ∨ ¬(¬((¬(▷f;di(g)));skip)))
     by auto
 have 8: ⊢ ¬(¬((¬(▷f;di(g)));skip)) =  (¬(empty ∨ (▷f;di(g));skip ))
     using NotNotChopSkip  by fastforce
 hence 9: ⊢ (empty ∨ ¬(¬((¬(▷f;di(g)));skip))) =  (empty ∨ ¬(empty ∨ (▷f;di(g));skip ))
     by auto
 have 10: ⊢ (empty ∨ ¬(empty ∨ (▷f;di(g));skip )) = (empty ∨ (more ∧ ¬((▷f;di(g));skip )))
     by (meson 6 7 9 NotChopSkipEqvMoreAndNotChopSkip Prop04 Prop06)
 have 11: ⊢ (empty ∨ (more ∧ ¬((▷f;di(g));skip ))) = (empty ∨ ¬((▷f;di(g));skip ))
     by (simp add: empty-d-def, auto)
 have 12: ⊢  (empty ∨ ¬((▷f;di(g));skip )) =  (empty ∨ ¬(▷f;(di(g);skip) ))
     using ChopAssocB 11 by fastforce
 have 13: ⊢  (¬(▷f;(di(g);skip) )) =  (¬(▷f;(ds(g)) ))
     using DsDi using RightChopEqvChop by fastforce
 hence 14: ⊢ (empty ∨ ¬(▷f;(di(g);skip) )) =  (empty ∨ ¬(▷f;(ds(g)) ))
     by auto
 have 15: ⊢ (empty ∨ ¬(▷f;(ds(g)) )) = (empty ∨ ¬(di (▷f)) ∨ (▷f;(¬(ds g))))
     using NotFstChop by fastforce
 have 16: ⊢  (▷f;(¬(ds g)))=   (▷f;(bs (¬g)))
     using NotDsEqvBsNot RightChopEqvChop by blast
 hence 17: ⊢ ((empty ∨ ¬(di (▷f))) ∨ (▷f;(¬(ds g)))) = ((empty ∨ ¬(di (▷f))) ∨  (▷f;(bs (¬g))))
     by auto
```

from *1 2 3 5 6 7 9 10 11 12 14 15 17* **show** *?thesis* **by** *fastforce*
qed

lemma *FstFstChopEqvFstChopFst*:
$\vdash \rhd(\rhd f;g) = \rhd f;\rhd g$
proof –
have *1*: $\vdash \rhd(\rhd f;g) = ((\rhd f;g) \wedge \mathit{bs}\ (\neg(\rhd f;g)))$
by (*simp add*: *first-d-def*)
have *2*: $\vdash \mathit{bs}\ (\neg(\rhd f;g)) = (\mathit{empty} \vee \neg(\mathit{di}(\rhd f)) \vee (\rhd f;\mathit{bs}(\neg g)))$
using *BsNotFstChop* **by** *auto*
hence *3*: $\vdash ((\rhd f;g) \wedge \mathit{bs}\ (\neg(\rhd f;g))) = ((\rhd f;g) \wedge (\mathit{empty} \vee \neg(\mathit{di}(\rhd f)) \vee (\rhd f;\mathit{bs}(\neg g))))$
by *auto*
have *4*: $\vdash ((\rhd f;g) \wedge (\mathit{empty} \vee \neg(\mathit{di}(\rhd f)) \vee (\rhd f;\mathit{bs}(\neg g)))) =$
$(((\rhd f;g) \wedge \mathit{empty}) \vee ((\rhd f;g) \wedge \neg(\mathit{di}(\rhd f))) \vee ((\rhd f;g) \wedge (\rhd f;\mathit{bs}(\neg g))))$
by *auto*
have *5*: $\vdash \neg((\rhd f;g) \wedge \neg(\mathit{di}(\rhd f)))$
using *ChopImpDi* **by** *fastforce*
hence *6*: $\vdash (((\rhd f;g) \wedge \mathit{empty}) \vee ((\rhd f;g) \wedge \neg(\mathit{di}(\rhd f))) \vee ((\rhd f;g) \wedge (\rhd f;\mathit{bs}(\neg g)))) =$
$(((\rhd f;g) \wedge \mathit{empty}) \vee ((\rhd f;g) \wedge (\rhd f;\mathit{bs}(\neg g))))$
by *auto*
have *7*: $\vdash ((\rhd f;g) \wedge (\rhd f;(\mathit{bs}(\neg g)))) = ((\rhd f;(g \wedge (\mathit{bs}(\neg g)))))$
using *LFstAndDistrC* **by** *blast*
hence *8*: $\vdash (((\rhd f;g) \wedge \mathit{empty}) \vee ((\rhd f;g) \wedge (\rhd f;(\mathit{bs}(\neg g))))) =$
$(((\rhd f;g) \wedge \mathit{empty}) \vee ((\rhd f;(g \wedge (\mathit{bs}(\neg g))))))$
by *auto*
have *9*: $\vdash (((\rhd f;g) \wedge \mathit{empty}) \vee ((\rhd f;(g \wedge (\mathit{bs}(\neg g)))))) = (((\rhd f;g) \wedge \mathit{empty}) \vee \rhd f;\rhd g)$
by (*simp add*: *first-d-def*)
have *10*: $\vdash ((\rhd f;g) \wedge \mathit{empty}) = ((\rhd f;\rhd g) \wedge \mathit{empty})$
using *FstChopEmptyEqvFstChopFstEmpty* **by** *blast*
hence *11*: $\vdash (((\rhd f;g) \wedge \mathit{empty}) \vee \rhd f;\rhd g) = (((\rhd f;\rhd g) \wedge \mathit{empty}) \vee \rhd f;\rhd g)$
by *auto*
have *12*: $\vdash (((\rhd f;\rhd g) \wedge \mathit{empty}) \vee \rhd f;\rhd g) = \rhd f;\rhd g$
by *auto*
from *1 3 4 6 8 9 11 12* **show** *?thesis* **by** (*metis inteq-reflection*)
qed

lemma *FstFixFst*:
$\vdash \rhd(\rhd f) = \rhd f$
proof –
have *1*: $\vdash \rhd f = (\rhd f);\mathit{empty}$ **using** *ChopEmpty* **by** (*metis int-eq*)
hence *2*: $\vdash \rhd(\ \rhd f\ ) = \rhd((\rhd f);\mathit{empty})$ **using** *FstEqvRule* **by** *blast*
have *3*: $\vdash \rhd((\rhd f);\mathit{empty}) = \rhd f;\rhd \mathit{empty}$ **using** *FstFstChopEqvFstChopFst* **by** *auto*
have *4*: $\vdash \rhd f;\rhd \mathit{empty} = \rhd f;\mathit{empty}$ **using** *FstEmpty* **using** *RightChopEqvChop* **by** *blast*
have *5*: $\vdash \rhd f;\mathit{empty} = \rhd f$ **using** *ChopEmpty* **by** *blast*
from *2 3 4 5* **show** *?thesis* **by** *fastforce*
qed

lemma *FstCSEqvEmpty*:
$\vdash \rhd(f^{\star}) = \mathit{empty}$
proof –

```
 have 1: ⊢ ▷(f⋆) = ▷(empty ∨ ((f ∧ more);f⋆)) using ChopstarEqv FstEqvRule by blast
 from 1 show ?thesis using FstEmptyOrEqvEmpty by fastforce
qed

lemma FstIterFixFst:
⊢ power (▷ f) n = ▷(power (▷ f) n)
proof
 (induct n)
 case 0
 then show ?case
 proof –
  have 1: ⊢ power (▷ f) 0 = empty by auto
  have 2: ⊢ empty = ▷ empty using FstEmpty by auto
  have 3: ⊢ ▷ empty = ▷(power (▷f) 0) by auto
  from 1 2 3 show ?thesis by auto
 qed
 next
  case (Suc n)
  then show ?case
  proof –
   have 4: ⊢ (power (▷ f) (Suc n)) = (▷ f) ;(power (▷f) n)
   by (simp)
   have 5: ⊢ (▷ f) ;(power (▷f) n) = (▷ f) ; ▷ (power (▷ f) n)
   using RightChopEqvChop Suc.hyps by blast
   have 6: ⊢ (▷ f) ; ▷ (power (▷ f) n) = ▷(▷f;(power (▷ f) n))
   using FstFstChopEqvFstChopFst by fastforce
   have 7: ⊢ ▷(▷f;(power (▷ f) n)) = ▷(power (▷ f) (Suc n))
   by simp
   from 4 5 6 7 show ?thesis by fastforce
  qed
qed

lemma DsImpNotFst:
⊢ ds f ⟶ (¬(▷f))
proof –
 have 1: ⊢ (ds f ∧ ▷f) = (ds f ∧ (f ∧ bs (¬ f))) by (simp add: first-d-def)
 have 2: ⊢ (ds f ∧ (f ∧ bs (¬ f))) = (ds f ∧ f ∧ ¬(ds f))  using NotDsEqvBsNot by fastforce
 from 1 2 show ?thesis by fastforce
qed

lemma FstLenAndEqvLenAnd:
⊢ ▷(len(k) ∧ f) = (len(k) ∧ f)
proof –
 have 1: ⊢ len(k) ∧ f ∧ ds(len(k)∧ f) ⟶ ds (len(k))
      using DsAndImpElimL by fastforce
 hence 2: ⊢ len(k) ∧ f ∧ ds(len(k)∧ f) ⟶ (di(len(k)));skip
      using DsDi by fastforce
 hence 3: ⊢ len(k) ∧ f ∧ ds(len(k)∧ f) ⟶ ((len(k);#True));skip
      by (simp add: di-d-def)
 hence 4: ⊢ len(k) ∧ f ∧ ds(len(k)∧ f) ⟶ (len(k);(#True;skip))
```

```
    using ChopAssocB by fastforce
 hence 5: ⊢ len(k) ∧ f ∧ ds(len(k)∧ f) ⟶ (len(k);(skip;#True))
    using SkipTrueEqvTrueSkip TrueChopSkipEqvSkipChopTrue RightChopEqvChop by fastforce
 hence 6: ⊢ len(k) ∧ f ∧ ds(len(k)∧ f) ⟶ (len(k);(skip;#True)) ∧ len(k)
    by auto
 hence 7: ⊢ len(k) ∧ f ∧ ds(len(k)∧ f) ⟶ (len(k);(skip;#True)) ∧ len(k);empty
    using ChopEmpty by (metis int-eq)
 hence 8: ⊢ len(k) ∧ f ∧ ds(len(k)∧ f) ⟶ (len(k);((skip;#True) ∧ empty))
    using LFixedAndDistrB1 by fastforce
 have 9: ⊢ ¬(len(k);((skip;#True) ∧ empty))
    by (simp add: empty-d-def more-d-def next-d-def chop-defs Valid-def)
 have 10: ⊢ len(k) ∧ f ⟶ ¬(ds(len(k)∧ f))
    using 8 9 by fastforce
 hence 11: ⊢ len(k) ∧ f ⟶ bs (¬(len(k)∧ f))
    using NotDsEqvBsNot by fastforce
 hence 12: ⊢ len(k) ∧ f ⟶ (len(k) ∧ f) ∧ bs (¬(len(k)∧ f))
    by auto
 hence 13: ⊢ len(k) ∧ f ⟶ ▷(len(k) ∧ f)
    by (simp add: first-d-def)
 have 14: ⊢ ▷(len(k) ∧ f) ⟶ len(k) ∧ f
    by (simp add: first-d-def,auto)
 from 13 14 show ?thesis by fastforce
qed

lemma FstAndElimL:
⊢ ▷f ⟶ f
by (simp add: first-d-def, auto)

lemma FstImpNotDiChopSkip:
⊢ ▷f ⟶ ¬(di f;skip)
proof –
 have 1: ⊢ ▷f ⟶ bs (¬ f) by (simp add: first-d-def,auto)
 hence 2: ⊢ ▷f ⟶ ¬(ds f) using NotDsEqvBsNot by fastforce
 have 3: ⊢ ds f = di f ; skip using DsDi by blast
 hence 4: ⊢ (¬(ds f)) = (¬(di f;skip)) by auto
 from 2 4 show ?thesis by fastforce
qed

lemma FstImpNotDiChopSkipB:
⊢ ▷f ⟶ ¬(di (f;skip))
proof –
 have 1: ⊢ ▷f ⟶ bs (¬ f)
    by (simp add: first-d-def,auto)
 hence 2: ⊢ ▷f ⟶ ¬(ds f)
    using NotDsEqvBsNot by fastforce
 have 3: ⊢ ds f = di f ; skip
    using DsDi by blast
 have 4: ⊢ di f ; skip = (f;#True);skip
    by (simp add: di-d-def)
 have 5: ⊢(f;#True);skip = f;(#True;skip)
```

```
    using ChopAssocB by blast
 have 6: ⊢ f;(#True;skip) = f;(skip;#True)
    using SkipTrueEqvTrueSkip using TrueChopSkipEqvSkipChopTrue RightChopEqvChop by blast
 have 7: ⊢ f;(skip;#True) = (f;skip);#True
    using ChopAssoc by blast
 have 8: ⊢ (f;skip);#True = di(f;skip)
    by (simp add: di-d-def)
 have 9: ⊢  (¬(ds f)) = (¬(di(f;skip)))
    using 3 4 5 6 7 8 by fastforce
 from 2 9 show ?thesis by fastforce
qed

lemma FstImpDiEqv:
⊢ ▷f ⟶ (di f = f)
proof −
 have 1: ⊢ ▷f  ⟶ ¬(di f;skip) using FstImpNotDiChopSkip by blast
 have 2: ⊢ di f ⟶ f ∨ (di f;skip)  using DiEqvOrDiChopSkipB by fastforce
 have 3: ⊢ ▷f  ∧ di f  ⟶ (f ∨ (di f;skip)) ∧ ¬(di f;skip)  using 1 2 by fastforce
 have 4: ⊢ ((f ∨ (di f;skip)) ∧ ¬(di f;skip)) = (f ∧ ¬(di f;skip)) by auto
 have 5: ⊢  ▷f  ∧ di f  ⟶ f ∧ ¬(di f;skip)  using 3 4 by fastforce
 hence 6: ⊢  ▷f  ∧ di f  ⟶ f  by fastforce
 hence 7: ⊢ ▷f  ⟶ (di f  ⟶ f) using FstAndElimL by fastforce
 have 8: ⊢ f  ⟶ di f  using DiIntro by auto
 hence 9: ⊢ ▷f  ⟶ (f  ⟶ (di f)) by auto
 from 7 9 show ?thesis by fastforce
qed

lemma FstAndDiFstAndEqvFstAnd:
⊢ (▷f ∧  di(▷f ∧ g)) = (▷f ∧ g)
proof −
 have 1: ⊢ ▷f ∧  di(▷f ∧ g) ⟶ ▷ f
    by auto
 have 2: ⊢ ▷f ∧  di(▷f ∧ g) ⟶ di(▷f ∧ g)
    by auto
 have 3: ⊢ di(▷f ∧ g) = ((▷f ∧ g) ∨ di((▷f ∧ g);skip))
    using DiEqvOrDiChopSkipA by blast
 have 4: ⊢ di((▷f ∧ g);skip) = ((▷f ∧ g);skip);#True
    by (simp add: di-d-def)
 have 5: ⊢ ▷f ∧  di(▷f ∧ g) ⟶ (▷f ∧ g) ∨ ((▷f ∧ g);skip);#True
    using 2 3 4 by fastforce
 have 6: ⊢ ▷f ∧ g ⟶ f
    using FstAndElimL by fastforce
 hence 7: ⊢ ((▷f ∧ g);skip);#True  ⟶ (f;skip);#True
    by (simp add: LeftChopImpChop)
 hence 8: ⊢ ((▷f ∧ g);skip);#True  ⟶ di(f;skip)
    by (simp add: di-d-def)
 have  9: ⊢ ▷f ⟶ ¬(di(f;skip))
    using FstImpNotDiChopSkipB by  blast
 have 10: ⊢ ▷f ∧  di(▷f ∧ g) ⟶ ((▷f ∧ g) ∨ di(f;skip))
    using 5 8  by fastforce
 have 11: ⊢ ▷f ∧  di(▷f ∧ g) ⟶ ¬(di(f;skip)) ∧ ((▷f ∧ g) ∨ di(f;skip))
     using 9 10 1 by fastforce
 have 12: ⊢ (¬(di(f;skip)) ∧ ((▷f ∧ g) ∨ di(f;skip))) = (¬(di(f;skip)) ∧ ((▷f ∧ g)))
     by auto
 have 13: ⊢ ▷f ∧  di(▷f ∧ g) ⟶ (▷f ∧ g)
     using 11 12 by auto
 have 14: ⊢ (▷f ∧ g) ⟶ ▷f
     by auto
 hence 15: ⊢ (▷f ∧ g) ⟶ di(▷f ∧ g )
     using DiIntro by auto
 have 16: ⊢ (▷f ∧ g) ⟶ ▷f ∧ di(▷f ∧ g)
     using 14 15 by auto
 from 13 16 show ?thesis by fastforce
qed

lemma FstAndDiImpBsNotAndDi:
⊢ (▷f ∧ di g) ⟶ (bs (¬(di f ∧ g)))
proof −
 have 1: ⊢ (▷f ∧ di g) ∧ ¬(bs (¬(di f ∧ g))) ⟶ ds(di f ∧ g)
     by (simp add: ds-d-def,auto)
 hence 2: ⊢ (▷f ∧ di g) ∧ ¬(bs (¬(di f ∧ g))) ⟶ ds(di f)
     using DsAndImp by fastforce
 hence 3: ⊢ (▷f ∧ di g) ∧ ¬(bs (¬(di f ∧ g))) ⟶ di(di f);skip
     using DsDi by fastforce
 hence 4: ⊢ (▷f ∧ di g) ∧ ¬(bs (¬(di f ∧ g))) ⟶ di f;skip
     using DiEqvDiDi by (metis int-eq)
 hence 5: ⊢ (▷f ∧ di g) ∧ ¬(bs (¬(di f ∧ g))) ⟶ ds f
     using DsDi by fastforce
 hence 6: ⊢  (▷f ∧ di g) ∧ ¬(bs (¬(di f ∧ g))) ⟶¬ (▷ f)
     using DsImpNotFst by fastforce
 from 6 show ?thesis by auto
qed

lemma FstFstOrEqvFstOrL:
⊢ ▷(▷f ∨ g) = ▷(f ∨ g)
proof −
 have 1: ⊢ ▷(f ∨ g) = ((f ∨ g) ∧ bs (¬(f ∨ g)))
     by (simp add: first-d-def)
 have 2: ⊢  (¬(f ∨ g)) =  (¬f ∧ ¬ g)
     by auto
 hence 3: ⊢  bs(¬(f ∨ g)) =  bs (¬f ∧ ¬ g)
     using BsEqvRule by blast
 have 4: ⊢ bs (¬f ∧ ¬ g) = (bs (¬ f) ∧ bs (¬ g))
     using BsAndEqv by fastforce
 hence 5: ⊢ ((f ∨ g) ∧ bs(¬(f ∨ g))) =  ((f ∨ g) ∧ bs (¬ f) ∧ bs (¬ g))
     using 4 3 by fastforce
 have 6: ⊢ ((f ∨ g) ∧ bs (¬ f) ∧ bs (¬ g)) =
           (((f ∧ bs (¬ f)) ∨ (g ∧ bs (¬ f))) ∧ bs (¬ g))
     by auto
 have 7: ⊢ (((f ∧ bs (¬ f)) ∨ (g ∧ bs (¬ f))) ∧ bs (¬ g)) =
           ((▷f ∨ (g ∧ bs (¬ f))) ∧ bs (¬ g))
     by (simp add: first-d-def)
 have 8: ⊢ ((▷f ∨ (g ∧ bs (¬ f))) ∧ bs (¬ g)) =
           (((▷f ∨ g) ∧ (▷f ∨ bs (¬f))) ∧ bs (¬ g))
     by auto
 have 9: ⊢ (((▷f ∨ g) ∧ (▷f ∨ bs (¬f))) ∧ bs (¬ g)) =
           (((▷f ∨ g) ∧ ((f ∧ bs (¬ f)) ∨ bs (¬f))) ∧ bs (¬ g))
     by (simp add: first-d-def)
 have 10: ⊢ (((▷f ∨ g) ∧ ((f ∧ bs (¬ f)) ∨ bs (¬f))) ∧ bs (¬ g)) =
            ((▷f ∨ g) ∧ bs (¬ f) ∧ bs (¬ g))
     by auto
 have 11: ⊢ ((▷f ∨ g) ∧ bs (¬ f) ∧ bs (¬ g)) =
             ((▷f ∨ g) ∧ bs(¬(▷f))∧ bs (¬ g))
     using BsNotFstEqvBsNot by fastforce
 have 12: ⊢ ((▷f ∨ g) ∧ bs(¬(▷f))∧ bs (¬ g)) =
            ((▷f ∨ g) ∧ bs (¬(▷f) ∧ ¬ g))
     using BsAndEqv by fastforce
 have 13: ⊢  (¬(▷f) ∧ ¬ g) = (¬(▷f ∨ g))
     by auto
 hence 14: ⊢ bs (¬(▷f) ∧ ¬ g) = bs (¬(▷f ∨ g))
     using BsEqvRule by blast
 hence 15: ⊢ ((▷f ∨ g) ∧ bs (¬(▷f) ∧ ¬ g)) = ((▷f ∨ g) ∧ bs (¬(▷f ∨ g)))
     by auto
 have 16:  ⊢ ((▷f ∨ g) ∧ bs (¬(▷f ∨ g))) = ▷(▷f ∨ g)
     by (simp add: first-d-def)
 from 16 15 12 11 10 9 8 7 6 5 1 show ?thesis by (metis int-eq)
qed

lemma FstFstOrEqvFstOrR:
     ⊢ ▷(f ∨ ▷g)= ▷(f ∨ g)
proof −
 have 1: ⊢ (f ∨ ▷g)= (▷g ∨ f) by auto
 hence 2: ⊢ ▷ (f ∨ ▷g)= ▷(▷g ∨ f)  using FstEqvRule by blast
 have 3: ⊢ ▷(▷g ∨ f) = ▷(g ∨ f) using FstFstOrEqvFstOrL by blast
 have 4: ⊢ (g ∨ f) = (f ∨ g) by auto
 hence 5: ⊢ ▷(g ∨ f) = ▷(f ∨ g)  using FstEqvRule by blast
 from 2 3 5 show ?thesis by fastforce
qed

lemma FstFstOrEqvFstOr:
     ⊢ ▷(▷f ∨ ▷g) = ▷(f ∨ g)
proof −
 have 1: ⊢  ▷(▷f ∨ ▷g) = ▷(f ∨ ▷g) using FstFstOrEqvFstOrL by blast
 have 2: ⊢ ▷(f ∨ ▷g) = ▷(f ∨ g) using FstFstOrEqvFstOrR by blast
 from 1 2 show ?thesis by fastforce
qed

lemma FstLenEqvLen:
     ⊢ ▷( len(k)) = len(k)
proof −
```

```
 have 1: ⊢ ▷( len(k) ∧ #True) = (len(k) ∧ #True) using FstLenAndEqvLenAnd by blast
 have 2: ⊢ (len(k) ∧ #True) = len(k) by auto
 hence 3: ⊢ ▷( len(k) ∧ #True) = ▷(len(k))  using FstEqvRule by blast
 from 1 2 3 show ?thesis by auto
qed

lemma FstSkip:
⊢ ▷ skip = skip
proof −
 have 1: ⊢ skip = len(1)   using LenOneEqvSkip by fastforce
 hence 2: ⊢ ▷skip = ▷ (len(1)) using FstEqvRule by blast
 have 3: ⊢ ▷( len(1)) = len(1) using FstLenEqvLen by blast
 from 1 2 3 show ?thesis using LenOneEqvSkip by fastforce
qed

lemma HaltStateEqvFstFinState:
⊢ halt (init w) = ▷ (fin (init w))
proof −
 have 1: ⊢ halt (init w) = □(empty = (init w)) by (simp add: halt-d-def)
 have 21: ⊢ (empty = (init w)) = (((empty ⟶ (init w)) ∧ ((init w) ⟶ empty)))
     by auto
 hence 2: ⊢ □(empty = (init w)) = (□((empty ⟶ (init w)) ∧ ((init w) ⟶ empty)))
     by (simp add: BoxEqvBox)
 have 3: ⊢ (□((empty ⟶ (init w)) ∧ ((init w) ⟶ empty))) =
          (□((empty ⟶ (init w))) ∧ □((init w) ⟶ empty))
     by (metis 21 BoxAndBoxEqvBoxRule int-eq)
 have 4: ⊢ ((init w) ⟶ empty) = (more ⟶ ¬(init w))
     by (simp add: empty-d-def,auto)
 hence 5: ⊢ □ ((init w) ⟶ empty) = □ (more ⟶ ¬(init w)) using BoxEqvBox by blast
 have 6: ⊢ □ (more ⟶ ¬(init w)) = bs(¬(fin(init w))) using BoxMoreStateEqvBsFinState by blast
 have 7: ⊢ □((empty ⟶ (init w))) = fin(init w) by (simp add: fin-d-def)
 have 8: ⊢  (□((empty ⟶ (init w))) ∧ □((init w) ⟶ empty)) =
           (fin(init w) ∧ bs(¬(fin(init w)))) using 5 6 7 by fastforce
 from 1 2 3 8 show ?thesis by (metis first-d-def inteq-reflection)
qed

lemma FstLenEqvLenFst:
⊢ ▷(len k ; f) = len k ; ▷ f
proof −
  have 1: ⊢ len k ; f = ▷(len k) ; f  using FstLenEqvLen LeftChopEqvChop by fastforce
  have 2: ⊢ ▷(len k ; f) = ▷ (▷(len k) ; f)  using 1 FstEqvRule by blast
  have 3: ⊢ ▷ (▷(len k) ; f) = ▷(len k) ; ▷f using FstFstChopEqvFstChopFst by blast
  have 4: ⊢ ▷(len k) ; ▷f = len k ; ▷f using FstLenEqvLen LeftChopEqvChop by fastforce
  from 2 3 4 show ?thesis by fastforce
qed

lemma FstNextEqvNextFst:
⊢ ▷(○ f) = ○( ▷ f)
proof −
  have 1: ⊢ ▷(○ f) = ▷(skip ; f)  using FstEqvRule by (simp add: next-d-def)
  have 2: ⊢ skip ; f = ▷skip ; f  using FstSkip using LeftChopEqvChop by fastforce
  have 3: ⊢ ▷(skip ; f) = ▷ (▷skip ; f)  using 2 FstEqvRule LeftChopEqvChop by blast
  have 4: ⊢ ▷ (▷skip ; f) = ▷skip ; ▷f  using 3 FstFstChopEqvFstChopFst by blast
  have 5: ⊢ ▷skip ; ▷f = skip ; ▷f  using 4 FstSkip LeftChopEqvChop by blast
  have 6: ⊢ skip ; ▷f  =  ○( ▷ f) by (simp add: next-d-def)
  from 1 2 3 4 5 6 show ?thesis by fastforce
qed

lemma FstDiamondStateEqvHalt:
⊢ ▷ (◇ (init w)) = halt (init w)
proof −
  have 1: ⊢ ◇ (init w) = ◇ ((init w) ∧ #True) by simp
  have 2: ⊢ fin (init w) ; #True = ◇ ((init w) ∧ #True) using 1 FinChopEqvDiamond by blast
  have 3: ⊢ fin (init w) ; #True = di (fin (init w)) by (simp add: di-d-def)
  have 4: ⊢ (◇ (init w)) = (di (fin (init w))) using 1 2 3 by fastforce
  have 5: ⊢ ▷ (◇ (init w)) = ▷ (di (fin (init w))) using 4 FstEqvRule by blast
  hence 6: ⊢ ▷ (◇ (init w)) = ▷ (fin (init w)) using FstDiEqvFst by fastforce
  hence 7: ⊢ ▷ (◇ (init w)) = halt (init w) using HaltStateEqvFstFinState by fastforce
  from 7 show ?thesis by simp
qed

lemma FstBoxStateEqvStateAndEmpty:
⊢ ▷ (□ (init w)) = ((init w) ∧ empty)
proof −
  have 1: ⊢ ((init w) ∧ (□ (init w))⋆) = □ (init w)
  using BoxCSEqvBox by blast
  have 2: ⊢ □ (init w) = ((init w) ∧ (□ (init w))⋆)
  using 1 by auto
  hence 3: ⊢ □ (init w) = ((init w) ∧ (□ (init w))⋆)
  by blast
  have 4: ⊢ ((init w) ∧ empty) ; (□ (init w))⋆ = ((init w) ∧ (□ (init w))⋆)
  using StateAndEmptyChop by blast
  have 5: ⊢ ((init w) ∧ (□ (init w))⋆)  =  ((init w) ∧ empty) ; (□ (init w))⋆
  using 4 by fastforce
  have 6: ⊢ □ (init w) = ((init w) ∧ empty) ; (□ (init w))⋆
  using 3 5 by fastforce
  have 7: ⊢ ((init w) ∧ empty) ; (□ (init w))⋆ = ▷ (init w) ; (□ (init w))⋆
  using FstState by (metis AndChopCommute int-eq)
  have 8: ⊢ □ (init w) = ▷ (init w) ; (□ (init w))⋆
  using 6 7 by fastforce
  have 9: ⊢ ▷ (□ (init w)) = ▷ (▷ (init w) ; (□ (init w))⋆)
  using 8 FstEqvRule by blast
  have 10: ⊢ ▷ (▷ (init w) ; (□ (init w))⋆) = ▷ (init w) ;  ▷ ((□ (init w))⋆)
  using FstFstChopEqvFstChopFst by blast
  have 11: ⊢ ▷ (init w) ;  ▷ ((□ (init w))⋆) = ▷ (init w) ; empty
  using RightChopEqvChop FstCSEqvEmpty by blast
  have 12: ⊢ ▷ (init w) ; empty = ▷ (init w)
  using RightChopEqvChop ChopEmpty by blast
  have 13: ⊢ ▷ (init w) = ((init w) ∧ empty)
  using FstState by fastforce
  from 9 10 11 12 13 show ?thesis by fastforce
qed

lemma FstAndFstStarEqvFst:
⊢ (▷f ∧ (▷f)⋆) = ▷f
proof −
  have 1: ⊢ (▷f)⋆ = (empty ∨ (▷f);(▷f)⋆)
      using CSEqvOrChopCS by fastforce
  have 2: ⊢ ((▷f)⋆ ∧ ▷f) = ((empty ∨ (▷f);(▷f)⋆) ∧ ▷f)
      using 1 by fastforce
  have 3: ⊢ ((empty ∨ (▷f);(▷f)⋆) ∧ ▷f) = ((empty ∧ ▷f) ∨ ((▷f);(▷f)⋆ ∧ ▷f))
      by auto
  have 4: ⊢ ((▷f)⋆ ∧ ▷f) = ((empty ∧ ▷f) ∨ ((▷f);(▷f)⋆ ∧ ▷f))
      using 2 3 by fastforce
  have 5: ⊢ ((▷f);(▷f)⋆ ∧ ▷f) = ((▷f);(▷f)⋆ ∧ ▷f;empty)
      using ChopEmpty by (metis inteq-reflection)
  have 6: ⊢ ((▷f);(▷f)⋆ ∧ ▷f;empty) = (▷f);((▷f)⋆ ∧ empty)
      using LFstAndDistrC by blast
  have 7: ⊢ ((▷f)⋆ ∧ empty) = empty
      using EmptyImpCS by fastforce
  have 8: ⊢ (▷f);((▷f)⋆ ∧ empty) = ▷f
      using 7 ChopEmpty by (metis inteq-reflection)
  have 9: ⊢ ((▷f);(▷f)⋆ ∧ ▷f) = ▷f
      using 5 6 8 by fastforce
  have 10: ⊢ ((▷f)⋆ ∧ ▷f) = ((empty ∧ ▷f) ∨ ▷f)
      using 4 9 by fastforce
  have 11: ⊢ ((empty ∧ ▷f) ∨ ▷f) = ▷f
      by auto
  have 12:  ⊢ ((▷f)⋆ ∧ ▷f) = ▷f
      using 10 11 by fastforce
  from 12 show ?thesis by auto
qed

lemma HaltStateEqvFstHaltState:
  ⊢ halt(init(w)) = ▷(halt(init(w)))
proof −
  have 1:  ⊢ halt (init w) = ▷ (fin (init w))
      by (simp add: HaltStateEqvFstFinState)
  have 2:  ⊢ ▷ (fin (init w)) = ▷ (▷ (fin (init w)))
    using FstEqvRule FstFixFst by fastforce
  have 3:  ⊢ ▷ (▷ (fin (init w))) = ▷(halt(init(w)))
     using FstEqvRule HaltStateEqvFstFinState by fastforce
    from 1 2 3 show ?thesis by fastforce
qed


lemma DiHaltAndDiHaltAndEqvDiHaltAndAnd:
  ⊢ (di(halt (init w) ∧ f) ∧ di(halt (init w) ∧ g)) = di(halt (init w) ∧ f ∧ g)
proof −
  have 1: ⊢ (di(halt (init w) ∧ f) ∧ di(halt (init w) ∧ g)) =
          (di(▷(fin (init w)) ∧ f ) ∧ di (▷(fin (init w)) ∧ g))
      using HaltStateEqvFstFinState by (metis LFstAndDistrD inteq-reflection)
  have 2: ⊢ (di(▷(fin (init w)) ∧ f ) ∧ di(▷(fin (init w)) ∧ g)) =
          di(▷(fin (init w)) ∧ f ∧ g)
      using LFstAndDistrD by fastforce
  have 3: ⊢ di(▷(fin (init w)) ∧ f ∧ g)  = di(halt (init w) ∧ f ∧ g)
      using HaltStateEqvFstFinState  by (metis DiEqvDi int-eq lift-and-com)
  from 1 2 3 show ?thesis using int-eq by metis
qed
```

```
lemma counter-ex-lhs:
⊢ ((▷(len(5)) ∧ ▷(len(2))) ; (len(5) ∨ len(2))) = #False
proof −
 have 1: ⊢ ((▷(len(5)) ∧ ▷(len(2))) ; (len(5) ∨ len(2))) =
          (len(5) ∧ len(2)); (len(5) ∨ len(2))
      by (metis FstLenAndEqvLenAnd FstLenEqvLen LeftChopEqvChop inteq-reflection)
 have 2: ⊢ (len(5) ∧ len(2)) = #False
     by (simp add: Valid-def len-defs)
 have 3: ⊢ ((len(5) ∧ len(2)); (len(5) ∨ len(2))) = (#False;(len(5) ∨ len(2)))
     by (simp add: 2 LeftChopEqvChop)
 have 4: ⊢ (#False;(len(5) ∨ len(2))) = #False
     by (simp add: Valid-def chop-defs)
 from 1 3 4 show ?thesis by fastforce
qed
```

```
lemma counter-ex-rhs:
⊢ ((▷ (len(5)) ; (len(5) ∨ len(2))) ∧ (▷ (len(2)) ; (len(5) ∨ len(2)))) = len(7)
proof −
 have 1: ⊢ (▷ (len(5)) ; (len(5) ∨ len(2))) =
           len(5);(len(5) ∨ len(2))
     using FstLenEqvLen LeftChopEqvChop by blast
 have 2: ⊢ (▷ (len(2)) ; (len(5) ∨ len(2))) =
           len(2) ;(len(5) ∨ len(2))
     using FstLenEqvLen LeftChopEqvChop by blast
 have 3: ⊢ len(5);(len(5) ∨ len(2)) =
          ((len(5);len(5)) ∨ (len(5);len(2)))
     by (simp add: ChopOrEqv)
 have 4: ⊢ ((len(5);len(5)) ∨ (len(5);len(2))) =
          (len(10) ∨ len(7))
     using LenEqvLenChopLen inteq-reflection by fastforce
 have 5: ⊢ len(2) ;(len(5) ∨ len(2)) =
          ((len(2);len(5)) ∨ (len(2);len(2)))
     by (simp add: ChopOrEqv)
 have 6: ⊢ ((len(2);len(5)) ∨ (len(2);len(2))) =
          (len(7) ∨ len(4))
     using LenEqvLenChopLen inteq-reflection by fastforce
 have 7: ⊢ ((len(10) ∨ len(7)) ∧ (len(7) ∨ len(4))) =
          ((len(7) ∨ len(10)) ∧ (len(7) ∨ len(4)))
     by fastforce
 have 8: ⊢ ((len(7) ∨ len(10)) ∧ (len(7) ∨ len(4))) =
           (len(7) ∨ (len(10) ∧ len(4)))
     by fastforce
 have 9: ⊢ (len(10) ∧ len(4)) = #False
     by (simp add: Valid-def len-defs)
 have 10 : ⊢ (len(7) ∨ (len(10) ∧ len(4))) = len(7)
     using 9 by auto
 have 11: ⊢ ((▷ (len(5)) ; (len(5) ∨ len(2))) ∧ (▷ (len(2)) ; (len(5) ∨ len(2)))) =
           (len(5);(len(5) ∨ len(2)) ∧ len(2) ;(len(5) ∨ len(2)))
     using 1 2 by fastforce
 have 12: ⊢ (len(5);(len(5) ∨ len(2)) ∧ len(2) ;(len(5) ∨ len(2))) = len(7)
        using 10 3 4 5 6
     by fastforce
 from 11 12 show ?thesis by fastforce
qed



end


theory Monitor
imports First

begin
```

7 Monitors

The RV monitors language is introduced plus the algebraic properties of the monitor operators.

7.1 Syntax

```
datatype ('a ::world) monitor =
   mFIRST-d 'a formula             ((FIRST -) [84] 83)
 | mUPTO-d 'a monitor 'a monitor ((- UPTO -) [84,84] 83)
 | mTHRU-d 'a monitor 'a monitor ((- THRU -) [84,84] 83)
 | mTHEN-d 'a monitor 'a monitor ((- THEN -) [84,84] 83)
 | mWITH-d 'a monitor 'a formula ((- WITH -) [84,84] 83)


fun MON :: ('a::world) monitor ⇒ 'a formula
where  (MON (FIRST f))  = LIFT(▷ f)
     | (MON (a UPTO b)) = LIFT(▷((MON a) ∨ (MON b) ))
     | (MON (a THRU b)) = LIFT(▷(di(MON a) ∧ di(MON b)))
     | (MON (a THEN b)) = LIFT((MON a);(MON b))
     | (MON (a WITH f)) = LIFT((MON a) ∧ f)

syntax
```

```
 -MON :: 'a monitor ⇒ lift ((𝓜 -) [80] 80)

translations
 -MON == CONST MON

definition eq-d :: ('a:: world) monitor ⇒ 'a monitor ⇒ bool (( - ≃ -) [84,84] 83)
where
 eq-d a b ≡ (⊢ (𝓜 a) = (𝓜 b))

lemma MonEqRefl:
 a ≃ a
by (simp add: eq-d-def)

lemma MonEqSym:
 assumes a ≃ b
 shows   b ≃ a
using assms by (metis eq-d-def inteq-reflection)

lemma MonEqTrans:
 assumes a ≃ b
         b ≃ c
 shows   a ≃ c
using assms(1) assms(2) by (metis eq-d-def inteq-reflection)

lemma MonEq:
 (a ≃ b) = (⊢ (𝓜 a) = (𝓜 b))
by (simp add: eq-d-def)

lemma MonEqSubstWith:
 assumes a ≃ b
 shows   (a WITH f) ≃ (b WITH f)
using assms by (metis MON.simps(5) eq-d-def inteq-reflection lift-and-com)

lemma MonEqSubstThen:
 assumes a1 ≃ b1
         a2 ≃ b2
 shows   (a1 THEN a2) ≃ (b1 THEN b2)
using assms(1) assms(2) by (simp add: ChopEqvChop eq-d-def)

lemma MonEqSubstUpto:
 assumes a1 ≃ b1
         a2 ≃ b2
 shows   (a1 UPTO a2) ≃ (b1 UPTO b2)
using assms(1) assms(2) by (metis (mono-tags, lifting) MON.simps(2) eq-d-def int-eq MonEqRefl)

lemma MonEqSubstThru:
 assumes a1 ≃ b1
         a2 ≃ b2
 shows   (a1 THRU a2) ≃ (b1 THRU b2)
using assms(1) assms(2) by (metis (mono-tags, lifting) MON.simps(3) eq-d-def int-eq MonEqRefl)
```

7.2 Derived Monitors

definition *HALT-d* :: (*'a* :: *world*) *formula* ⇒ *'a monitor*
where *HALT-d w* ≡ *FIRST*(*LIFT*(*fin* (*init w*)))

definition *LEN-d* :: *nat* ⇒ (*'a* ::*world*) *monitor*
where
LEN-d k ≡ *FIRST* (*LIFT*(*len k*))

definition *EMPTY-d* :: (*'a*:: *world*) *monitor*
where
EMPTY-d ≡ *FIRST* (*LIFT*(*empty*))

definition *SKIP-d* :: (*'a*:: *world*) *monitor*
where
SKIP-d ≡ *FIRST* (*LIFT* (*skip*))

syntax
-HALT-d :: *lift* ⇒ *'a monitor* ((*HALT* -) [*84*] *83*)
-LEN-d :: *nat* ⇒ *'a monitor* ((*LEN* -) [*84*] *83*)
-EMPTY-d :: *'a monitor* ((*EMPTY*))
-SKIP-d :: *'a monitor* ((*SKIP*))

syntax (*ASCII*)
-HALT-d :: *lift* ⇒ *'a monitor* ((*HALT* -) [*84*] *83*)
-LEN-d :: *nat* ⇒ *'a monitor* ((*LEN* -) [*84*] *83*)
-EMPTY-d :: *'a monitor* ((*EMPTY*))
-SKIP-d :: *'a monitor* ((*SKIP*))

translations
-HALT-d ⇌ *CONST HALT-d*
-LEN-d ⇌ *CONST LEN-d*
-EMPTY-d ⇌ *CONST EMPTY-d*
-SKIP-d ⇌ *CONST SKIP-d*

definition *GUARD-d* :: (*'a*::*world*) *formula* ⇒ *'a monitor*
where
GUARD-d w ≡ (*EMPTY WITH LIFT*(*init w*))

primrec *TIMES-d* :: (*'a* :: *world*) *monitor* ⇒ *nat* ⇒ *'a monitor*
where
TIMES-0 : *TIMES-d a 0* = *EMPTY*
| *TIMES-Suc*: *TIMES-d a* (*Suc k*) = (*a THEN* (*TIMES-d a k*))

syntax
-GUARD-d :: *lift* ⇒ *'a monitor* ((*GUARD* -) [*84*] *83*)
-TIMES-d :: [*'a monitor*,*nat*] ⇒ *'a monitor* ((- *TIMES* -) [*84*,*84*] *83*)

syntax (*ASCII*)

```
 -GUARD-d :: lift ⇒ 'a monitor            ((GUARD -) [84] 83)
 -TIMES-d :: ['a monitor,nat] ⇒ 'a monitor ((- TIMES -) [84,84] 83)

translations
  -GUARD-d ⇌ CONST GUARD-d
  -TIMES-d ⇌ CONST TIMES-d


definition FAIL-d :: ('a:: world) monitor
where
 FAIL-d ≡ GUARD (#False)

definition ALWAYS-d :: ('a :: world) monitor ⇒ 'a formula ⇒ 'a monitor
where
 ALWAYS-d a w ≡  (a WITH LIFT((bi (fin (init w)))))

definition SOMETIME-d :: ('a :: world) monitor ⇒ 'a formula ⇒ 'a monitor
where
 SOMETIME-d a w ≡  (a WITH LIFT((di (fin (init w)))))

definition LIMIT-d :: ('a :: world) formula ⇒ 'a formula
where
 LIMIT-d f ≡ LIFT(bs (¬ f))

definition UNTIL-d :: ('a :: world) formula ⇒ 'a formula ⇒ 'a monitor
  where
 UNTIL-d w1 w2 ≡ (HALT w2) WITH (LIFT(bm w1))

syntax
 -FAIL-d     :: 'a monitor                    (FAIL)
 -ALWAYS-d   :: ['a monitor,lift] ⇒ 'a monitor ((- ALWAYS -) [84,84] 83)
 -SOMETIME-d :: ['a monitor,lift] ⇒ 'a monitor ((- SOMETIME -) [84,84] 83)
 -LIMIT-d    :: lift ⇒ lift                  ((Limit -) [84] 83)
 -UNTIL-d    :: [lift,lift] ⇒ 'a monitor       ((- UNTIL -) [84,84] 83)

syntax (ASCII)
 -FAIL-d     :: 'a monitor                    (FAIL)
 -ALWAYS-d   :: ['a monitor,lift] ⇒ 'a monitor ((- ALWAYS -) [84,84] 83)
 -SOMETIME-d :: ['a monitor,lift] ⇒ 'a monitor ((- SOMETIME -) [84,84] 83)
 -LIMIT-d    :: lift ⇒ lift                  ((Limit -) [84] 83)
 -UNTIL-d    :: [lift,lift] ⇒ 'a monitor       ((- UNTIL -) [84,84] 83)

translations
 -FAIL-d     ⇌ CONST FAIL-d
 -ALWAYS-d   ⇌ CONST ALWAYS-d
 -SOMETIME-d ⇌ CONST SOMETIME-d
 -LIMIT-d    ⇌ CONST LIMIT-d
 -UNTIL-d    ⇌ CONST UNTIL-d
```

```
definition WITHIN-d :: ('a :: world) monitor ⇒ 'a formula ⇒ 'a monitor
where
 WITHIN-d a f ≡ (a WITH LIFT(Limit f))

syntax
 -WITHIN-d :: ['a monitor,lift] ⇒ 'a monitor ((- WITHIN -) [84,84] 83)

syntax (ASCII)
 -WITHIN-d :: ['a monitor,lift] ⇒ 'a monitor ((- WITHIN -) [84,84] 83)

translations
 -WITHIN-d ⇌ CONST WITHIN-d


definition AND-d :: ('a :: world) monitor ⇒ 'a monitor ⇒ 'a monitor
where
  AND-d a b ≡ (a WITH LIFT(ℳ b))

definition ITERATE-d :: ('a :: world) monitor ⇒ 'a monitor ⇒ 'a monitor
where
  ITERATE-d a b ≡ (a WITH (LIFT (ℳ b)⋆))


syntax
 -AND-d     :: ['a monitor,'a monitor] ⇒ 'a monitor ((- AND -) [84,84] 83)
 -ITERATE-d :: ['a monitor,'a monitor] ⇒ 'a monitor ((- ITERATE -) [84,84] 83)

syntax (ASCII)
 -AND-d     :: ['a monitor,'a monitor] ⇒ 'a monitor ((- AND -) [84,84] 83)
 -ITERATE-d :: ['a monitor,'a monitor] ⇒ 'a monitor ((- ITERATE -) [84,84] 83)

translations
 -AND-d     ⇌ CONST AND-d
 -ITERATE-d ⇌ CONST ITERATE-d


definition STAR-d :: ('a :: world) monitor ⇒ 'a formula ⇒ 'a monitor
where
 STAR-d a f ≡ ((FIRST LIFT(◇ f)) ITERATE (a))

definition REPEAT-d :: ('a :: world) monitor ⇒ 'a formula ⇒ 'a monitor
where
  REPEAT-d a w ≡ ((HALT w) ITERATE (a WITH LIFT(keep(¬ (init w)))))

syntax
 -STAR-d   :: ['a monitor,lift] ⇒ 'a monitor ((- STAR -) [84,84] 83)
 -REPEAT-d :: ['a monitor,lift] ⇒ 'a monitor (( - REPEATUNTIL -) [84,84] 83)

syntax (ASCII)
 -STAR-d   :: ['a monitor,lift] ⇒ 'a monitor ((- STAR -) [84,84] 83)
 -REPEAT-d :: ['a monitor,lift] ⇒ 'a monitor (( - REPEATUNTIL -) [84,84] 83)
```

```
translations
 -STAR-d   ⇌ CONST STAR-d
 -REPEAT-d ⇌ CONST REPEAT-d
```

7.3 Monitor Laws

```
lemma MFixFst:
⊢ (ℳ a) = ▷ (ℳ a)
proof
 (induct a )
 case (mFIRST-d x)
 then show ?case
  proof −
   have 1: ⊢ (ℳ (FIRST x)) = ▷ x  by simp
   have 2: ⊢ ▷ x = ▷ (▷ x) using FstFixFst by fastforce
   have 3: ⊢  ▷ (▷ x) = ▷(ℳ (FIRST x)) by simp
   from 1 2 3 show ?thesis by fastforce
  qed
 next
  case (mUPTO-d a1 a2)
  then show ?case
  proof −
  have 1: ⊢ (ℳ (a1 UPTO a2)) = ▷( (ℳ a1) ∨ (ℳ a2))
       by (simp )
  have 2: ⊢  ▷( (ℳ a1) ∨ (ℳ a2)) = ▷(▷((ℳ a1) ∨ (ℳ a2)))
       using FstFixFst by fastforce
  have 3: ⊢ ▷(▷((ℳ a1) ∨ (ℳ a2))) = ▷(ℳ (a1 UPTO a2))
       using 2 by simp
  from 1 2 3 show ?thesis by fastforce
 qed
next
  case (mTHRU-d a1 a2)
  then show ?case
  proof −
   have 1: ⊢ (ℳ (a1 THRU a2)) = ▷( di (ℳ a1) ∧ di(ℳ a2))
        by (simp)
   have 2: ⊢ ▷( di (ℳ a1) ∧ di(ℳ a2)) = ▷(▷(di(ℳ a1) ∧ di(ℳ a2)))
        using FstFixFst by fastforce
   have 3: ⊢ ▷(▷( di (ℳ a1) ∧ di(ℳ a2))) = ▷(ℳ (a1 THRU a2))
        using 2 by simp
   from 1 2 3 show ?thesis by fastforce
 qed
next
  case (mTHEN-d a1 a2)
  then show ?case
  proof −
   have 1: ⊢ (ℳ (a1 THEN a2)) = (ℳ a1) ; (ℳ a2)
        by (simp)
   have 2: ⊢ (ℳ a1) ; (ℳ a2) = ▷(ℳ a1) ; ▷(ℳ a2)
        using ChopEqvChop mTHEN-d.hyps(1) mTHEN-d.hyps(2) by blast
   have 3: ⊢ ▷(ℳ a1) ; ▷(ℳ a2) = ▷(▷(ℳ a1) ; (ℳ a2))
        using FstFstChopEqvFstChopFst by fastforce
   have 4: ⊢ ▷(▷(ℳ a1) ; (ℳ a2)) = ▷((ℳ a1) ; (ℳ a2))
        using FstEqvRule LeftChopEqvChop mTHEN-d.hyps(1) by (metis inteq-reflection)
   have 5: ⊢ ▷((ℳ a1) ; (ℳ a2)) = ▷(ℳ (a1 THEN a2))
        using 4 by simp
   from 1 2 3 4 5 show ?thesis by fastforce
  qed
next
  case (mWITH-d a x2)
  then show ?case
  proof −
   have 1: ⊢ (ℳ (a WITH x2)) = ((ℳ a) ∧ ( x2))
        by (simp )
   have 2: ⊢ ((ℳ a) ∧ ( x2)) = (▷(ℳ a) ∧ ( x2))
        using mWITH-d.hyps by fastforce
   have 3: ⊢ (▷(ℳ a) ∧ ( x2)) = ▷(▷(ℳ a) ∧ ( x2))
        using FstFstAndEqvFstAnd by fastforce
   have 4: ⊢ ▷(▷(ℳ a) ∧ ( x2)) = ▷((ℳ a) ∧ ( x2))
        using 2 FstEqvRule by fastforce
   have 5: ⊢ ▷((ℳ a) ∧ ( x2)) = ▷(ℳ (a WITH x2))
        using 4 by simp
   from 1 2 3 4 5 show ?thesis by (metis inteq-reflection)
 qed
qed

lemma MGuardFalseEqvFalse:
⊢  ℳ(GUARD #False) = #False
proof −
  have 1: ⊢ ℳ(GUARD #False) = ℳ(EMPTY WITH LIFT(init #False)) by (simp add: GUARD-d-def)
  have 2: ⊢ ℳ(EMPTY WITH LIFT(init #False)) = (ℳ(EMPTY) ∧ (init #False)) by (simp )
  have 3: ⊢ #False = (init #False) by (simp add:init-defs Valid-def)
  have 4: ⊢ (ℳ(EMPTY) ∧ (init #False)) = (ℳ(EMPTY) ∧ #False) using 3 by auto
  have 5: ⊢ (ℳ(EMPTY) ∧ #False) = #False by simp
  have 6: ⊢ (ℳ(EMPTY) ∧ (init #False)) = #False using 4 5 by simp
  have 7: ⊢ ℳ(EMPTY WITH LIFT(init #False)) = #False using 2 6 by fastforce
  have 8: ⊢ ℳ(GUARD #False) = #False using 1 7 by fastforce
  from 8 show ?thesis by auto
qed

lemma MFirstFalseEqvFalse:
⊢  ℳ(FIRST LIFT #False) = #False
proof −
  have 1: ⊢ ℳ(FIRST LIFT #False) = ▷ #False by (simp )
  have 2:  ⊢ ℳ(FIRST LIFT #False) = #False using FstFalse  by fastforce
 from 2 show ?thesis by auto
qed
```

```
lemma MFailAlt:
 ⊢  𝓜 FAIL = #False
proof –
  have 1: ⊢  𝓜 FAIL =  𝓜 (GUARD (#False))  by (simp add: FAIL-d-def)
  have 2: ⊢ 𝓜(GUARD (#False)) = #False using MGuardFalseEqvFalse by auto
 from 1 2 show ?thesis by fastforce
qed

lemma MFailEqvFirstFalseWithinEmpty:
   FAIL ≃  ((FIRST LIFT #False) WITHIN empty)
proof –
  have 1: ⊢ 𝓜 ( (FIRST LIFT #False) WITHIN ( empty )) =
            𝓜((FIRST LIFT #False) WITH LIFT(Limit empty) )
       by (simp add: WITHIN-d-def)
  have 2: ⊢  𝓜((FIRST LIFT #False) WITH LIFT(Limit empty) ) =
             (𝓜(FIRST LIFT #False) ∧ (Limit empty ))
       by (simp )
  have 3: ⊢  𝓜((FIRST LIFT #False) WITH LIFT(Limit empty) ) = #False
       using MFirstFalseEqvFalse by auto
  have 4: ⊢ 𝓜( (FIRST LIFT #False) WITHIN ( empty )) = #False
       using 1 3 by fastforce
  have 5: ⊢ 𝓜( FAIL) = #False
       using MFailAlt by simp
 from 4 5 show ?thesis using MonEq by (metis int-eq)
 qed

lemma MEmptyAlt:
 ⊢  𝓜 EMPTY = empty
proof –
  have 1: ⊢ 𝓜 ( EMPTY) = 𝓜( (FIRST LIFT empty)) by (simp add: EMPTY-d-def)
  have 2: ⊢ 𝓜 ( (FIRST LIFT empty)) = ▷ empty by (simp)
  have 3: ⊢ ▷ empty = empty using FstEmpty by auto
 from 1 2 3 show ?thesis by fastforce
qed

lemma MSkipAlt:
 ⊢  𝓜 SKIP = skip
proof –
  have 1: ⊢  𝓜 SKIP = 𝓜 (FIRST LIFT skip) by (simp add: SKIP-d-def)
  have 2: ⊢ 𝓜 (FIRST LIFT skip) = ▷ skip by (simp)
  have 3: ⊢ ▷ skip = skip using FstSkip by simp
 from 1 2 3 show ?thesis by fastforce
qed

lemma MGuardAlt:
 ⊢  𝓜 (GUARD(w)) = (empty ∧ init w)
proof –
  have 1: ⊢  𝓜(GUARD(w)) = 𝓜(EMPTY WITH ( LIFT (init w))) by (simp add:GUARD-d-def)
  have 2: ⊢ 𝓜(EMPTY WITH ( LIFT(init w))) = (𝓜( EMPTY) ∧ ( init w)) by (simp)
  have 3: ⊢ (𝓜( EMPTY) ∧ ( init w)) = (empty ∧ ( init w)) using MEmptyAlt by fastforce
  have 4: ⊢  (empty ∧ ( init w)) = (empty ∧ init w) by simp
 from 1 2 3 4 show ?thesis by fastforce
qed

lemma MLengthAlt:
 ⊢ 𝓜 (LEN(k)) = len(k)
proof −
  have 1: ⊢ 𝓜(LEN(k)) = 𝓜(FIRST LIFT(len(k))) by (simp add:LEN-d-def)
  have 2: ⊢ 𝓜(FIRST LIFT(len(k))) = ▷(len(k)) by (simp)
  have 3: ⊢ ▷(len(k)) = len(k) using FstLenEqvLen by blast
 from 1 2 3 show ?thesis by fastforce
qed

lemma MAlwaysAlt:
  ⊢ 𝓜(a ALWAYS w) = (𝓜(a) ∧ □ (init w))
proof −
  have 1: ⊢ 𝓜(a ALWAYS w) = 𝓜(a WITH LIFT(bi (fin (init w))))
       by (simp add: ALWAYS-d-def)
  have 2: ⊢ 𝓜(a WITH LIFT(bi (fin (init w)))) = (𝓜(a) ∧ (bi (fin (init w))))
       by (simp)
  have 3: ⊢ (𝓜(a) ∧ (bi (fin (init w)))) = (𝓜(a) ∧ □ (init w))
      using BoxStateEqvBiFinState by fastforce
  from 1 2 3 show ?thesis by fastforce
qed


lemma MSometimeAlt:
 ⊢ 𝓜(a SOMETIME w) = (𝓜(a) ∧ ◇ (init w))
proof −
  have 1: ⊢ 𝓜(a SOMETIME w) = 𝓜(a WITH LIFT(di (fin (init w))))
       by (simp add: SOMETIME-d-def)
  have 2: ⊢ 𝓜(a WITH LIFT(di (fin (init w)))) = (𝓜(a) ∧ (di (fin (init w))))
       by (simp)
  have 3: ⊢ 𝓜(a WITH LIFT(di (fin (init w)))) = (𝓜(a) ∧  ◇ (init w))
       using DiamondStateEqvDiFinState  by fastforce
  from 1 2 3 show ?thesis by fastforce
qed

lemma MWithinAlt:
 ⊢ 𝓜(a WITHIN f) = (𝓜(a) ∧ (bs (¬ f)))
proof −
  have 1: ⊢ 𝓜(a WITHIN f) = 𝓜(a WITH LIFT(bs (¬ f)))
       by (simp add: WITHIN-d-def LIMIT-d-def)
  have 2: ⊢ 𝓜(a WITH LIFT(bs (¬ f))) = (𝓜(a) ∧ (bs (¬ f)))
       by (simp)
  from 1 2 show ?thesis by fastforce
qed

lemma MTimesAlt:
 ⊢ 𝓜(a TIMES k) = power (𝓜(a)) k
```

```
proof
 (induct k)
 case 0
 then show ?case
  proof –
   have 1: ⊢  𝓜 (a TIMES 0) =  𝓜 EMPTY by simp
   have 2: ⊢  𝓜 EMPTY = empty  using MEmptyAlt by simp
   have 3: ⊢ empty = power (𝓜 a) 0 by simp
   from 1 2 3 show ?thesis by auto
  qed
 next
 case (Suc k)
 then show ?case
  proof –
   have 1: ⊢  𝓜( a TIMES Suc k) = 𝓜(a THEN (a TIMES k) )
        by simp
   have 2: ⊢ 𝓜(a THEN (a TIMES k) ) = (𝓜 a);(𝓜 (a TIMES k))
        by (simp)
   have 3: ⊢ (𝓜 a);(𝓜(a TIMES k)) = (𝓜 a);(power (𝓜 a) k)
        using RightChopEqvChop Suc.hyps by blast
   have 4: ⊢ (𝓜 a);(power (𝓜 a) k) = power (𝓜 a) (Suc k)
        by simp
   from 1 2 3 4 show ?thesis by fastforce
  qed
 qed

lemma MUptoAlt:
⊢ 𝓜(a UPTO b) = (( (𝓜 a) ∧ bi (¬(𝓜 b))) ∨ ((𝓜 b) ∧ bi (¬(𝓜 a))) ∨ ((𝓜 a) ∧ (𝓜 b)))
proof –
  have 1: ⊢ 𝓜 (a UPTO b) = ▷((𝓜 a) ∨ (𝓜 b))
        by (simp)
  have 2: ⊢ ▷((𝓜 a) ∨ (𝓜 b)) = ((▷(𝓜 a) ∧  (bs (¬(𝓜 b)))) ∨ (▷(𝓜 b) ∧  (bs (¬(𝓜 a)))))
       using FstWithOrEqv by blast
  have 3: ⊢ ((▷(𝓜 a) ∧  (bs (¬(𝓜 b)))) ∨ (▷(𝓜 b) ∧  (bs (¬(𝓜 a))))) =
          (((𝓜 a) ∧ ((𝓜 b) ∨ ¬(𝓜 b)) ∧  (bs (¬(𝓜 b)))) ∨
          ((𝓜 b) ∧ ((𝓜 a) ∨ ¬(𝓜 a)) ∧  (bs (¬(𝓜 a)))))
       using MFixFst by fastforce
  have 4: ⊢ (((𝓜 a) ∧ ((𝓜 b) ∨ ¬(𝓜 b)) ∧  (bs (¬(𝓜 b)))) ∨
          ((𝓜 b) ∧ ((𝓜 a) ∨ ¬(𝓜 a)) ∧  (bs (¬(𝓜 a))))) =
          (((𝓜 a) ∧ ( ((𝓜 b) ∧ bs (¬(𝓜 b))) ∨ (¬(𝓜 b) ∧ bs (¬(𝓜 b)))) ) ∨
          ((𝓜 b) ∧ ( ((𝓜 a) ∧ bs (¬(𝓜 a))) ∨ (¬(𝓜 a) ∧ bs (¬(𝓜 a)))) ))
       by auto
  have 5: ⊢ (((𝓜 a) ∧ ( ((𝓜 b) ∧ bs (¬(𝓜 b))) ∨ (¬(𝓜 b) ∧ bs (¬(𝓜 b)))) ) ∨
          ((𝓜 b) ∧ ( ((𝓜 a) ∧ bs (¬(𝓜 a))) ∨ (¬(𝓜 a) ∧ bs (¬(𝓜 a)))) )) =
          (((𝓜 a) ∧ ( (▷(𝓜 b)) ∨ (¬(𝓜 b) ∧ bs (¬(𝓜 b)))) ) ∨
          ((𝓜 b) ∧ ( (▷(𝓜 a)) ∨ (¬(𝓜 a) ∧ bs (¬(𝓜 a)))) ))
       by (simp add: first-d-def)
  have 6: ⊢ (((𝓜 a) ∧ ( (▷(𝓜 b)) ∨ (¬(𝓜 b) ∧ bs (¬(𝓜 b)))) ) ∨
          ((𝓜 b) ∧ ( (▷(𝓜 a)) ∨ (¬(𝓜 a) ∧ bs (¬(𝓜 a)))) )) =
          (((𝓜 a) ∧ ( ((𝓜 b)) ∨ (¬(𝓜 b) ∧ bs (¬(𝓜 b)))) ) ∨
          ((ℳ b) ∧ ( ((ℳ a)) ∨ (¬(ℳ a) ∧ bs (¬(ℳ a)))) ))
      using MFixFst by fastforce
  have 7: ⊢ (¬(ℳ b) ∧ bs (¬(ℳ b))) = bi(¬(ℳ b))
      using AndBsEqvBi by blast
  have 8: ⊢ (¬(ℳ a) ∧ bs (¬(ℳ a))) = bi(¬(ℳ a))
      using AndBsEqvBi by blast
  have 9: ⊢ (((ℳ a) ∧ ( ((ℳ b)) ∨ ((¬(ℳ b)) ∧ bs(¬(ℳ b)))) ) ∨
          ((ℳ b) ∧ ( ((ℳ a)) ∨ ((¬(ℳ a)) ∧ bs(¬(ℳ a)))) )) =
          (((ℳ a) ∧ ( ((ℳ b)) ∨ ( bi(¬(ℳ b)))) ) ∨
          ((ℳ b) ∧ ( ((ℳ a)) ∨ ( bi(¬(ℳ a)))) ))
      using 7 8 by fastforce
  have 10: ⊢(((ℳ a) ∧ ( ((ℳ b)) ∨ ( bi(¬(ℳ b)))) ) ∨
          ((ℳ b) ∧ ( ((ℳ a)) ∨ ( bi(¬(ℳ a)))) )) =
          ((( (ℳ a) ∧ (ℳ b)) ∨ ( (ℳ a) ∧ bi(¬(ℳ b)))) ∨
          (( (ℳ b) ∧ (ℳ a)) ∨ ( (ℳ b) ∧ bi(¬(ℳ a)))))
      by auto
  have 11: ⊢ ((( (ℳ a) ∧ (ℳ b)) ∨ ( (ℳ a) ∧ bi(¬(ℳ b)))) ∨
          (( (ℳ b) ∧ (ℳ a)) ∨ ( (ℳ b) ∧ bi(¬(ℳ a))))) =
         (( (ℳ a) ∧ bi (¬(ℳ b))) ∨ ((ℳ b) ∧ bi (¬(ℳ a))) ∨ ((ℳ a) ∧ (ℳ b)))
      by auto
 from 1 2 3 4 5 6 9 10 11 show ?thesis by (metis int-eq)
qed

lemma MThruAlt:
⊢ ℳ(a THRU b) = (((ℳ a) ∧ di(ℳ b)) ∨ ((ℳ b) ∧ di(ℳ a)))
proof −
  have 1: ⊢ ℳ(a THRU b) = ▷(di(ℳ a) ∧ di(ℳ b))
      by (simp)
  have 2: ⊢ ▷(di(ℳ a) ∧ di(ℳ b)) = ((▷(ℳ a) ∧ di(ℳ b)) ∨ (▷(ℳ b) ∧ di(ℳ a)))
      using FstDiAndDiEqv by auto
  have 3: ⊢ ((▷(ℳ a) ∧ di(ℳ b)) ∨ (▷(ℳ b) ∧ di(ℳ a))) =
          (((ℳ a) ∧ di(ℳ b)) ∨ ((ℳ b) ∧ di(ℳ a)))
      using MFixFst by fastforce
 from 1 2 3 show ?thesis by fastforce
qed

lemma MHaltAlt:
⊢ ℳ(HALT w) = halt(init w)
proof −
  have 1: ⊢ ℳ(HALT w) = ℳ(FIRST LIFT(fin (init  w))) by (simp add: HALT-d-def)
  have 2: ⊢ ℳ(FIRST LIFT(fin (init  w))) = ▷ (fin (init  w)) by (simp)
  have 3: ⊢ ▷ (fin (init  w)) = halt(init w) using HaltStateEqvFstFinState by fastforce
  from 1 2 3 show ?thesis by fastforce
qed

lemma MFailUpto:
(FAIL UPTO a) ≃ ( a)
proof −
  have 1: ⊢  ℳ(FAIL UPTO a) = ▷( (ℳ FAIL) ∨ (ℳ a)) by (simp)
  have 2: ⊢  (ℳ FAIL ∨ ℳ a) = (#False ∨ ℳ a) using MFailAlt by auto
  have 3: ⊢ ▷(𝓜 FAIL ∨ (𝓜 a)) = ▷(#False ∨ (𝓜 a)) using 2 FstEqvRule by blast
  have 4: ⊢ (#False ∨ (𝓜 a)) = 𝓜 a by simp
  have 5: ⊢ ▷(#False ∨ (𝓜 a)) =  ▷(𝓜 a) using 4 FstEqvRule by blast
  have 6: ⊢ ▷(𝓜 a) = 𝓜 a using MFixFst by fastforce
  from 1 2 3 4 5 6 show ?thesis using MonEq by (metis int-eq)
qed

lemma MFailThru:
  (FAIL THRU ( a)) ≃  FAIL
proof −
  have 1: ⊢ 𝓜 (FAIL THRU ( a)) = ▷(di(𝓜 FAIL) ∧ di(𝓜 a))
       by (simp)
  have 2: ⊢ ▷(di (𝓜 FAIL) ∧ di(𝓜 a)) = ▷(di (#False) ∧ di(𝓜 a))
       using MFailAlt by (metis 1 int-eq)
  have 3: ⊢ di #False = #False
       by (simp add: di-defs Valid-def)
  hence 4: ⊢ ▷(di (#False) ∧ di(𝓜 a)) = ▷( (#False) ∧ di(𝓜 a))
       by (metis 2 inteq-reflection)
  have 5: ⊢ ▷( (#False) ∧ di(𝓜 a)) = ▷#False
       using FstEqvRule by fastforce
  have 6: ⊢ ▷#False = #False using FstFalse
       by auto
  have 7: ⊢ #False = 𝓜 FAIL
       using MFailAlt by auto
 from 1 2 4 5 6 7 show ?thesis using MonEq by (metis int-eq)
qed

lemma MFailAnd:
 (FAIL AND a) ≃  FAIL
proof −
  have 1: ⊢  𝓜 (FAIL AND a) = (𝓜 FAIL ∧ (𝓜 a)) by (simp add: AND-d-def)
  have 2: ⊢ (𝓜 FAIL ∧ (𝓜 a)) = (#False ∧ (𝓜 a)) using MFailAlt by fastforce
  have 3: ⊢ (#False ∧ (𝓜 a)) = #False by auto
  have 4: ⊢  𝓜(FAIL AND a) = #False using 1 2 3 by fastforce
  have 5: ⊢ #False =  𝓜 FAIL using MFailAlt by auto
  from 1 2 3 4 5 show ?thesis using MonEq by (metis int-eq)
qed

lemma MThenFail:
  (a THEN FAIL) ≃ FAIL
proof −
  have 1: ⊢  𝓜 (a THEN FAIL) = (𝓜 a);(𝓜 FAIL) by (simp)
  have 2: ⊢ (𝓜 a);(𝓜 FAIL) = (𝓜 a);#False  by (simp add: MFailAlt RightChopEqvChop)
  have 3: ⊢ (𝓜 a);#False = #False by (simp add: chop-d-def Valid-def)
  have 4: ⊢ #False =  𝓜 FAIL using MFailAlt by auto
 from 1 2 3 4 show ?thesis using MonEq by (metis int-eq)
qed

lemma MFailThen:
  ( FAIL THEN a) ≃  FAIL
```

```
proof –
  have 1: ⊢ ℳ( FAIL THEN a) = (ℳ FAIL);(ℳ a) by (simp)
  have 2: ⊢ (ℳ FAIL);(ℳ a) = #False;(ℳ a)  using MFailAlt using LeftChopEqvChop by blast
  have 3: ⊢ #False;(ℳ a) = #False by (simp add: chop-d-def Valid-def)
  have 4: ⊢ #False = ℳ FAIL using MFailAlt by auto
 from 1 2 3 4 show ?thesis  using MonEq by (metis int-eq)
qed

lemma MFailWith:
  ( FAIL WITH f) ≃  FAIL
proof –
  have 1: ⊢ ℳ (FAIL WITH f) = ((ℳ FAIL) ∧ f) by (simp)
  have 2: ⊢ ((ℳ FAIL) ∧ f) = (#False ∧ f) using MFailAlt by auto
  have 3: ⊢ (#False ∧ f) = #False by simp
  have 4: ⊢ #False =  ℳ FAIL using MFailAlt by auto
 from 1 2 3 4 show ?thesis using MonEq by (metis int-eq)
qed

lemma MWithFalse:
  (a WITH (LIFT(#False))) ≃  FAIL
proof –
  have 1: ⊢ ℳ (a WITH LIFT(#False)) = ((ℳ a) ∧ #False) by (simp)
  have 2: ⊢ ((ℳ a) ∧ #False) = ℳ FAIL using MFailAlt by auto
 from 1 2 show ?thesis using MonEq by (metis int-eq)
qed

lemma MWithTrue:
  (a WITH (LIFT(#True))) ≃  a
proof –
  have 1: ⊢ ℳ (a WITH LIFT(#True)) = ((ℳ a) ∧ #True) by (simp)
  have 2: ⊢ ((ℳ a) ∧ #True) = ℳ a by simp
 from 1 2 show ?thesis using MonEq by (metis int-eq)
qed

lemma MEmptyUpto:
  (EMPTY UPTO a) ≃  EMPTY
proof –
  have 1: ⊢  ℳ (EMPTY UPTO a) = ▷(ℳ EMPTY ∨ (ℳ a)) by (simp)
  have 2: ⊢ ℳ EMPTY  = empty  using MEmptyAlt by auto
 hence 3: ⊢  (ℳ EMPTY ∨ (ℳ a)) = (empty ∨ (ℳ a)) by auto
 hence 4: ⊢  ▷(ℳ EMPTY ∨ ℳ a) =  ▷(empty ∨ ℳ a) using FstEqvRule by blast
  have 5: ⊢ ▷(empty ∨ ℳ a) = empty using FstEmptyOrEqvEmpty by blast
  have 6: ⊢ empty = ℳ EMPTY using MEmptyAlt by auto
 from 1 4 5 6 show ?thesis using MonEq by (metis int-eq)
qed

lemma MEmptyThru:
  (EMPTY THRU a) ≃ ( a)
proof –
  have 1: ⊢  ℳ(EMPTY THRU a) = ▷(di(ℳ EMPTY) ∧ di(ℳ a)) by (simp)
  have 2: ⊢ di(ℳ EMPTY) = di empty  using MEmptyAlt DiEqvDi by blast
 hence 3: ⊢ (di(ℳ EMPTY) ∧ di(ℳ a)) = (di empty ∧ di(ℳ a)) by auto
 hence 4: ⊢ (di empty ∧ di(ℳ a)) = di(ℳ a) using DiEmpty by auto
  have 5: ⊢ (di(ℳ EMPTY) ∧ di(ℳ a)) = di(ℳ a) using 3 4 by fastforce
 hence 6: ⊢ ▷(di(ℳ EMPTY) ∧ di(ℳ a)) = ▷(di(ℳ a)) using FstEqvRule by blast
  have 7: ⊢ ▷(di(ℳ a)) = ▷(ℳ a) using FstDiEqvFst by blast
  have 8: ⊢ ▷(ℳ a) = (ℳ a) using MFixFst by fastforce
 from 1 6 7 8 show ?thesis using MonEq by (metis int-eq)
qed

lemma MThenEmpty:
 ( a THEN EMPTY) ≃ (a)
proof −
  have 1: ⊢ ℳ( a THEN EMPTY) = (ℳ a); (ℳ EMPTY) by (simp)
  have 2: ⊢ (ℳ a); (ℳ EMPTY) = (ℳ a); empty by (simp add: MEmptyAlt RightChopEqvChop)
  have 3: ⊢ (ℳ a); empty = (ℳ a) using ChopEmpty by auto
 from 1 2 3 show ?thesis using MonEq by (metis int-eq)
qed

lemma MEmptyThen:
 ( EMPTY THEN a) ≃ a
proof −
  have 1: ⊢ ℳ( EMPTY THEN a) = (ℳ EMPTY);(ℳ a) by (simp)
  have 2: ⊢ (ℳ EMPTY);(ℳ a) = empty;(ℳ a) by (simp add: MEmptyAlt LeftChopEqvChop)
  have 3: ⊢ empty;(ℳ a) = (ℳ a) by (simp add: EmptyChop)
 from 1 2 3 show ?thesis using MonEq by (metis int-eq)
qed

lemma MEmptyIterate:
 ( EMPTY ITERATE b) ≃  EMPTY
proof −
  have 1: ⊢ ℳ( EMPTY ITERATE b) = ℳ( EMPTY WITH LIFT(ℳ b)⋆)
       by (simp add: ITERATE-d-def)
  have 2: ⊢ ℳ (EMPTY WITH LIFT(ℳ b)⋆) = (ℳ EMPTY ∧ (ℳ b)⋆)
       by (simp)
  have 3: ⊢ (ℳ EMPTY ∧ (ℳ b)⋆)  = (empty ∧ (ℳ b)⋆)
       using MEmptyAlt by auto
  have 4: ⊢ (empty ∧ (ℳ b)⋆)  = (empty ∧ (empty ∨ (((ℳ b) ∧ more);(ℳ b)⋆)))
       using ChopstarEqv by fastforce
  have 5: ⊢ (empty ∧ (empty ∨ (((ℳ b) ∧ more);(ℳ b)⋆))) = empty
       by auto
  have 6: ⊢  ℳ(EMPTY ITERATE b) =  ℳ EMPTY
       using 1 2 3 4 5 MEmptyAlt by fastforce
 from 6 show ?thesis using MonEq by (metis int-eq)
qed

lemma MIterateIdemp:
 (a ITERATE a) ≃  (a)
proof −
  have 1: ⊢  ℳ(a ITERATE  a) =  ℳ (a WITH LIFT(ℳ a)⋆) by (simp add: ITERATE-d-def)
```

have *2*: $\vdash\ \mathcal{M}(a\ \mathit{WITH}\ \mathit{LIFT}(\mathcal{M}\ a)^\star) = ((\mathcal{M}\ a) \wedge (\mathcal{M}\ a)^\star)$ **by** (*simp*)
have *3*: $\vdash ((\mathcal{M}\ a) \wedge (\mathcal{M}\ a)^\star) = (\triangleright(\mathcal{M}\ a) \wedge (\triangleright(\mathcal{M}\ a))^\star)$ **using** *MFixFst*
by (*metis ImpCS inteq-reflection Prop10*)
have *4*: $\vdash (\triangleright(\mathcal{M}\ a) \wedge (\triangleright(\mathcal{M}\ a))^\star) = \triangleright(\mathcal{M}\ a)$ **using** *FstAndFstStarEqvFst* **by** *fastforce*
have *5*: $\vdash \triangleright(\mathcal{M}\ a) = \mathcal{M}\ a$ **using** *MFixFst* **by** *fastforce*
from *1 2 3 4 5* **show** *?thesis* **using** *MonEq* **by** (*metis int-eq*)
qed

lemma *MUptoIdemp*:
$(a\ \mathit{UPTO}\ a) \simeq (a)$
proof –
have *1*: $\vdash\ \mathcal{M}(a\ \mathit{UPTO}\ a) = \triangleright((\mathcal{M}\ a) \vee (\mathcal{M}\ a))$ **by** *auto*
have *2*: $\vdash \triangleright((\mathcal{M}\ a) \vee (\mathcal{M}\ a)) = \triangleright(\mathcal{M}\ a)$ **using** *FstEqvRule* **by** *fastforce*
have *3*: $\vdash \triangleright(\mathcal{M}\ a) = (\mathcal{M}\ a)$ **using** *MFixFst* **by** *fastforce*
from *1 2 3* **show** *?thesis* **using** *MonEq* **by** (*metis int-eq*)
qed

lemma *MThruIdemp*:
$(a\ \mathit{THRU}\ a) \simeq (a)$
proof –
have *1*: $\vdash\ \mathcal{M}(a\ \mathit{THRU}\ a) = \triangleright(\mathit{di}(\mathcal{M}\ a) \wedge \mathit{di}(\mathcal{M}\ a))$ **by** *auto*
have *2*: $\vdash \triangleright(\mathit{di}(\mathcal{M}\ a) \wedge \mathit{di}(\mathcal{M}\ a)) = \triangleright(\mathit{di}\ (\mathcal{M}\ a))$ **using** *FstEqvRule* **by** *fastforce*
have *3*: $\vdash \triangleright(\mathit{di}\ (\mathcal{M}\ a)) = \triangleright(\mathcal{M}\ a)$ **using** *FstDiEqvFst* **by** *blast*
have *4*: $\vdash \triangleright(\mathcal{M}\ a) = (\mathcal{M}\ a)$ **using** *MFixFst* **by** *fastforce*
from *1 2 3 4* **show** *?thesis* **using** *MonEq* **by** (*metis int-eq*)
qed

lemma *MAndIdemp*:
$(a\ \mathit{AND}\ a) \simeq (a)$
proof –
have *1*: $\vdash \mathcal{M}(a\ \mathit{AND}\ a) = ((\mathcal{M}\ a) \wedge (\mathcal{M}\ a))$ **by** (*simp add*: *AND-d-def*)
have *2*: $\vdash ((\mathcal{M}\ a) \wedge (\mathcal{M}\ a)) = (\mathcal{M}\ a)$ **by** *fastforce*
from *1 2* **show** *?thesis* **using** *MonEq* **by** (*metis int-eq*)
qed

lemma *MWithIdemp*:
$((a\ \mathit{WITH}\ f)\ \ \mathit{WITH}\ f) \simeq (a\ \mathit{WITH}\ f)$
proof –
have *1*: $\vdash \mathcal{M}((a\ \mathit{WITH}\ f)\ \ \mathit{WITH}\ f) = (((\mathcal{M}\ a) \wedge (\ f)) \wedge (\ f))$ **by** *auto*
have *2*: $\vdash (((\mathcal{M}\ a) \wedge (\ f)) \wedge (\ f)) = ((\mathcal{M}\ a) \wedge (\ f))$ **by** *fastforce*
have *3*: $\vdash ((\mathcal{M}\ a) \wedge (\ f)) = \mathcal{M}(a\ \mathit{WITH}\ f)$ **by** *auto*
from *1 2 3* **show** *?thesis* **using** *MonEq* **by** (*metis int-eq*)
qed

lemma *MUptoCommut*:
$(a\ \mathit{UPTO}\ b) \simeq (b\ \mathit{UPTO}\ a)$
proof –
have *1*: $\vdash \mathcal{M}(a\ \mathit{UPTO}\ b) = \triangleright((\mathcal{M}\ a) \vee (\mathcal{M}\ b))$ **by** (*simp*)
have *2*: $\vdash ((\mathcal{M}\ a) \vee (\mathcal{M}\ b)) = ((\mathcal{M}\ b) \vee (\mathcal{M}\ a))$ **by** *auto*
hence *3*: $\vdash \triangleright((\mathcal{M}\ a) \vee (\mathcal{M}\ b)) = \triangleright((\mathcal{M}\ b) \vee (\mathcal{M}\ a))$ **using** *FstEqvRule* **by** *blast*

```
  have 4: ⊢ ▷((𝓜 b) ∨ (𝓜 a)) = 𝓜(b UPTO a) by auto
 from 1 3 4 show ?thesis using MonEq by (metis int-eq)
qed

lemma MThruCommut:
 (a THRU b) ≃ (b THRU a)
proof −
  have 1: ⊢ 𝓜(a THRU b) = ▷(di(𝓜 a) ∧ di(𝓜 b)) by (simp)
  have 2: ⊢ (di(𝓜 a) ∧ di(𝓜 b)) = (di(𝓜 b) ∧ di(𝓜 a))  by auto
 hence 3: ⊢ ▷(di(𝓜 a) ∧ di(𝓜 b)) = ▷(di(𝓜 b) ∧ di(𝓜 a)) using FstEqvRule by blast
  have 4: ⊢ ▷(di(𝓜 b) ∧ di(𝓜 a)) = 𝓜(b THRU a) by auto
 from 1 3 4 show ?thesis using MonEq by (metis int-eq)
qed

lemma MAndCommut:
 (a AND b) ≃ (b AND a)
proof −
  have 1: ⊢ 𝓜(a AND b) = ((𝓜 a) ∧ (𝓜 b)) by (simp add: AND-d-def)
  have 2: ⊢ ((𝓜 a) ∧ (𝓜 b)) = ((𝓜 b) ∧ (𝓜 a))  by auto
  have 3: ⊢ ((𝓜 b) ∧ (𝓜 a)) = 𝓜(b AND a) by (simp add: AND-d-def)
 from 1 2 3 show ?thesis using MonEq by (metis int-eq)
qed

lemma MWithCommut:
 ((a WITH f) WITH g) ≃ ((a WITH g) WITH f)
proof −
  have 1: ⊢ 𝓜((a WITH f) WITH g) = (((𝓜 a) ∧ (f)) ∧ (g))  by auto
  have 2: ⊢ (((𝓜 a) ∧ (f)) ∧ (g)) = (((𝓜 a) ∧ (g)) ∧ (f)) by auto
  have 3: ⊢ (((𝓜 a) ∧ (g)) ∧ (f)) = 𝓜((a WITH g) WITH f) by auto
 from 1 2  3 show ?thesis using MonEq by (metis int-eq)
qed

lemma MWithAbsorp:
 ((a WITH f) WITH g) ≃ (a WITH LIFT(f ∧ g))
proof −
  have 1: ⊢ 𝓜((a WITH f) WITH g) = (((𝓜 a) ∧ (f)) ∧ (g)) by auto
  have 2: ⊢ (((𝓜 a) ∧ (f)) ∧ (g)) =  ((𝓜 a) ∧ (f ∧ g)) by auto
 from 1 2 show ?thesis by (simp add: MonEq)
qed

lemma MUptoAssoc:
 ((a UPTO b) UPTO c) ≃ (a UPTO (b UPTO c))
proof −
  have 1: ⊢ 𝓜((a UPTO b) UPTO c) = ▷(𝓜(a UPTO b) ∨ (𝓜 c))
       by (simp)
  have 2: ⊢ ▷(𝓜(a UPTO b) ∨ (𝓜 c)) = ▷(▷((𝓜 a) ∨ (𝓜 b)) ∨ (𝓜 c))
       by auto
  have 3: ⊢ ▷(▷((𝓜 a) ∨ (𝓜 b)) ∨ (𝓜 c)) = ▷(((𝓜 a) ∨ (𝓜 b)) ∨ (𝓜 c))
       using FstFstOrEqvFstOrL by blast
  have 4: ⊢ (((𝓜 a) ∨ (𝓜 b)) ∨ (𝓜 c)) = ((𝓜 a) ∨ ((𝓜 b) ∨ (𝓜 c)))
```

```
      by auto
  hence 5: ⊢ ▷(((𝓜 a) ∨ (𝓜 b)) ∨ (𝓜 c)) = ▷((𝓜 a) ∨ ((𝓜 b) ∨ (𝓜 c)))
      using FstEqvRule by blast
  have 6: ⊢ ▷((𝓜 a) ∨ ((𝓜 b) ∨ (𝓜 c))) = ▷((𝓜 a) ∨ ▷((𝓜 b) ∨ (𝓜 c)))
      using FstFstOrEqvFstOrR  by fastforce
  have 7: ⊢ ▷((𝓜 a) ∨ ▷((𝓜 b) ∨ (𝓜 c))) = ▷((𝓜 a) ∨ 𝓜(b UPTO c))
      by auto
  have 8: ⊢ ▷((𝓜 a) ∨ 𝓜(b UPTO c)) = 𝓜(a UPTO (b UPTO c))
      by auto
 from 1 2 3 5 6 7 8 show ?thesis using MonEq by (metis int-eq)
qed

lemma MThruAssoc:
 ((a THRU b) THRU c) ≃ (a THRU (b THRU c))
proof −
  have 1: ⊢ 𝓜((a THRU b) THRU c) = ▷(di(▷(di(𝓜 a) ∧ di(𝓜 b))) ∧ di(𝓜 c))
      by auto
  have 2: ⊢ di(▷(di(𝓜 a) ∧ di(𝓜 b))) = di((di(𝓜 a) ∧ di(𝓜 b)))
      using DiEqvDiFst by fastforce
  have 3: ⊢ di((di(𝓜 a) ∧ di(𝓜 b))) = (di(𝓜 a) ∧ di(𝓜 b))
      using DiDiAndEqvDi by blast
  have 4: ⊢ di(▷(di(𝓜 a) ∧ di(𝓜 b))) = (di(𝓜 a) ∧ di(𝓜 b))
      using 2 3 by fastforce
  hence 5: ⊢ (di(▷(di(𝓜 a) ∧ di(𝓜 b)))  ∧ di(𝓜 c)) = (di(𝓜 a) ∧ di(𝓜 b)  ∧ di(𝓜 c))
      by auto
  have 6: ⊢ (di(𝓜 b)  ∧ di(𝓜 c)) = di (di(𝓜 b)  ∧ di(𝓜 c))
      using DiDiAndEqvDi by fastforce
  have 7: ⊢ di (di(𝓜 b)  ∧ di(𝓜 c)) = di (▷(di(𝓜 b)  ∧ di(𝓜 c)))
      using DiEqvDiFst by blast
  have 8: ⊢ (di(𝓜 b)  ∧ di(𝓜 c)) = di (▷(di(𝓜 b)  ∧ di(𝓜 c)))
      using 6 7 by fastforce
  hence 9: ⊢ (di(𝓜 a) ∧ di(𝓜 b)  ∧ di(𝓜 c)) = (di(𝓜 a) ∧ di (▷(di(𝓜 b)  ∧ di(𝓜 c))))
      by auto
  have 10: ⊢ (di(▷(di(𝓜 a) ∧ di(𝓜 b)))  ∧ di(𝓜 c)) =
           (di(𝓜 a) ∧ di (▷(di(𝓜 b)  ∧ di(𝓜 c))))
      using 5 9 by fastforce
  hence 11: ⊢ ▷(di(▷(di(𝓜 a) ∧ di(𝓜 b)))  ∧ di(𝓜 c)) =
           ▷(di(𝓜 a) ∧ di (▷(di(𝓜 b)  ∧ di(𝓜 c))))
      using FstEqvRule by fastforce
  have 12: ⊢ ▷(di(𝓜 a) ∧ di (▷(di(𝓜 b)  ∧ di(𝓜 c)))) = 𝓜(a THRU (b THRU c))
      by auto
 from 1 11 12 show ?thesis using MonEq by (metis int-eq)
qed

lemma MAndAssoc:
 ((a AND b) AND c) ≃ (a AND (b AND c))
proof −
  have 1: ⊢ 𝓜((a AND b) AND c) = ((𝓜 a) ∧ (𝓜 b) ∧ (𝓜 c))
      using AND-d-def by (metis MON.simps(5) MWithAbsorp eq-d-def)
  have 2: ⊢ ((𝓜 a) ∧ (𝓜 b) ∧ (𝓜 c)) = 𝓜 (a AND (b AND c))
```

```
     using AND-d-def by (simp add: AND-d-def)
 from 1 2 show ?thesis using MonEq by (metis int-eq)
qed

lemma MThenAssoc:
 ((a THEN b) THEN c) ≃ (a THEN (b THEN c))
proof −
  have 1: ⊢ ℳ((a THEN b) THEN c) = ((ℳ a);(ℳ b));(ℳ c)  by auto
  have 2: ⊢ ((ℳ a);(ℳ b));(ℳ c) = (ℳ a);((ℳ b);(ℳ c)) using ChopAssocB by blast
  have 3: ⊢ (ℳ a);((ℳ b);(ℳ c)) = ℳ(a THEN (b THEN c)) by auto
 from 1 2 3 show ?thesis using MonEq by (metis int-eq)
qed

lemma MUptoThruAbsorp:
 (a UPTO (a THRU b)) ≃  a
proof −
  have 1: ⊢ ℳ(a UPTO (a THRU b)) = ▷((ℳ a) ∨ ▷(di(ℳ a) ∧ di(ℳ b) ))
     by simp
  have 2: ⊢ ▷((ℳ a) ∨ ▷(di(ℳ a) ∧ di(ℳ b) )) =
          ▷((ℳ a) ∨ (di(ℳ a) ∧ di(ℳ b) ))
     using FstFstOrEqvFstOrR by auto
  have 3: ⊢  ((ℳ a) ∨ (di(ℳ a) ∧ di(ℳ b) )) =
           (((ℳ a) ∨ di(ℳ a)) ∧ ((ℳ a) ∨ di(ℳ b)))
     by auto
  have 4: ⊢ (((ℳ a) ∨ di(ℳ a)) ∧ ((ℳ a) ∨ di(ℳ b))) =
          ((di(ℳ a)) ∧ ((ℳ a) ∨ di(ℳ b)))
     using OrDiEqvDi by fastforce
  have 5: ⊢ ((ℳ a) ∨ (di(ℳ a) ∧ di(ℳ b) )) =
          ((di(ℳ a)) ∧ ((ℳ a) ∨ di(ℳ b)))
     using 3 4 by auto
  hence 6: ⊢ ▷ ((ℳ a) ∨ (di(ℳ a) ∧ di(ℳ b) )) =
           ▷ ((di(ℳ a)) ∧ ((ℳ a) ∨ di(ℳ b)))
     using FstEqvRule by blast
  have 7: ⊢ ▷ ((di(ℳ a)) ∧ ((ℳ a) ∨ di(ℳ b))) =
          ((di(ℳ a)) ∧ ((ℳ a) ∨ di(ℳ b)) ∧
          bs (¬( (di(ℳ a)) ∧ ((ℳ a) ∨ di(ℳ b)))))
     by (simp add: first-d-def, auto)
  have 8: ⊢ ((di(ℳ a)) ∧ ((ℳ a) ∨ di(ℳ b))) =
          ((di(ℳ a) ∧ (ℳ a)) ∨ (di(ℳ a) ∧ di(ℳ b)))
     by auto
  hence 9: ⊢ (¬((di(ℳ a)) ∧ ((ℳ a) ∨ di(ℳ b)))) =
          (¬((di(ℳ a) ∧ (ℳ a)) ∨ (di(ℳ a) ∧ di(ℳ b))))
     by fastforce
  have 10: ⊢ (¬((di(ℳ a) ∧ (ℳ a)) ∨ (di(ℳ a) ∧ di(ℳ b)))) =
           (¬(((ℳ a)) ∨ (di(ℳ a) ∧ di(ℳ b))))
     using AndDiEqv using 5 by auto
  have 11: ⊢ (¬(((ℳ a)) ∨ (di(ℳ a) ∧ di(ℳ b)))) =
            (¬(ℳ a) ∧ ¬(di(ℳ a) ∧ di(ℳ b)))
     by auto
  have 12: ⊢ (¬((di(ℳ a)) ∧ ((ℳ a) ∨ di(ℳ b)))) =
```

(¬(ℳ a) ∧ ¬(di(ℳ a) ∧ di(ℳ b)))
using *9 10 11* **by** *auto*
hence *13*: ⊢ bs (¬((di(ℳ a)) ∧ ((ℳ a) ∨ di(ℳ b)))) =
bs (¬(ℳ a) ∧ ¬(di(ℳ a) ∧ di(ℳ b)))
using *BsEqvRule* **by** *blast*
have *14*: ⊢ bs ((¬(ℳ a)) ∧ ¬(di(ℳ a) ∧ di(ℳ b))) =
(bs ((¬(ℳ a))) ∧ bs(¬(di(ℳ a) ∧ di(ℳ b))))
using *BsAndEqv* **by** *fastforce*
have *141*: ⊢ bs (¬((di(ℳ a)) ∧ ((ℳ a) ∨ di(ℳ b)))) =
(bs ((¬(ℳ a))) ∧ bs(¬(di(ℳ a) ∧ di(ℳ b))))
using *13 14* **by** *fastforce*
hence *15*: ⊢ ((di(ℳ a)) ∧ ((ℳ a) ∨ di(ℳ b)) ∧
bs (¬((di(ℳ a)) ∧ ((ℳ a) ∨ di(ℳ b))))) =
((di(ℳ a)) ∧ ((ℳ a) ∨ di(ℳ b)) ∧
bs ((¬(ℳ a))) ∧ bs(¬(di(ℳ a) ∧ di(ℳ b))))
by *auto*
have *16*: ⊢ ((di(ℳ a)) ∧ ((ℳ a) ∨ di(ℳ b)) ∧
bs ((¬(ℳ a))) ∧ bs(¬(di(ℳ a) ∧ di(ℳ b)))) =
((bs ((¬(ℳ a))) ∧ di(ℳ a)) ∧ ((ℳ a) ∨ di(ℳ b)) ∧
bs(¬(di(ℳ a) ∧ di(ℳ b))))
by *auto*
have *17*: ⊢ ((bs ((¬(ℳ a))) ∧ di(ℳ a)) ∧ ((ℳ a) ∨ di(ℳ b)) ∧
bs(¬(di(ℳ a) ∧ di(ℳ b)))) =
((▷(ℳ a)) ∧ ((ℳ a) ∨ di(ℳ b)) ∧
bs(¬(di(ℳ a) ∧ di(ℳ b))))
using *FstEqvBsNotAndDi* **by** *fastforce*
have *18*: ⊢ ((▷(ℳ a)) ∧ ((ℳ a) ∨ di(ℳ b)) ∧
bs(¬(di(ℳ a) ∧ di(ℳ b)))) =
(((ℳ a)) ∧ ((ℳ a) ∨ di(ℳ b)) ∧
bs(¬(di(ℳ a) ∧ di(ℳ b))))
using *MFixFst* **by** *fastforce*
have *19*: ⊢ (((ℳ a)) ∧ ((ℳ a) ∨ di(ℳ b)) ∧
bs(¬(di(ℳ a) ∧ di(ℳ b)))) =
(((ℳ a)) ∧ bs(¬(di(ℳ a) ∧ di(ℳ b))))
by *auto*
have *20*: ⊢ (¬(di(ℳ a) ∧ di(ℳ b))) = (¬(di(ℳ a)) ∨ ¬(di(ℳ b)))
by *auto*
have *21*: ⊢ (¬(di(ℳ a)) ∨ ¬(di(ℳ b))) = ((bi (¬(ℳ a))) ∨ (bi (¬(ℳ b))))
by (*simp add*: *bi-d-def*)
have *22*: ⊢ (¬(di(ℳ a) ∧ di(ℳ b))) = ((bi (¬(ℳ a))) ∨ (bi (¬(ℳ b))))
using *20 21* **by** *auto*
hence *23*: ⊢ bs (¬(di(ℳ a) ∧ di(ℳ b))) = bs ((bi (¬(ℳ a))) ∨ (bi(¬(ℳ b))))
using *BsEqvRule* **by** *blast*
have *24*: ⊢ bs ((bi (¬(ℳ a))) ∨ (bi (¬(ℳ b)))) = bs (¬(ℳ a)) ∨ bs (¬(ℳ b))
using *BsOrBsEqvBsBiOrBi* **by** *fastforce*
have *25*: ⊢ bs (¬(di(ℳ a) ∧ di(ℳ b))) = (bs (¬(ℳ a)) ∨ bs (¬(ℳ b)))
using *23 24* **using** *BsOrBsEqvBsBiOrBi* **by** *fastforce*
hence *26*: ⊢ ((ℳ a) ∧ bs (¬(di(ℳ a) ∧ di(ℳ b)))) =
((ℳ a) ∧ (bs (¬(ℳ a)) ∨ bs (¬(ℳ b))))
by *auto*

```
  have 27: ⊢ ((ℳ a) ∧ (bs (¬(ℳ a)) ∨ bs (¬(ℳ b)))) =
          (▷(ℳ a) ∧ (bs (¬(ℳ a)) ∨ bs (¬(ℳ b))))
      using MFixFst by fastforce
  have 28: ⊢ (▷(ℳ a) ∧ (bs (¬(ℳ a)) ∨ bs (¬(ℳ b)))) =
           ((ℳ a) ∧ bs (¬(ℳ a)) ∧ (bs (¬(ℳ a)) ∨ bs (¬(ℳ b))))
      by (simp add: first-d-def, auto)
  have 29: ⊢ ((ℳ a) ∧ bs (¬(ℳ a)) ∧ (bs (¬(ℳ a)) ∨ bs (¬(ℳ b)))) =
           ((ℳ a) ∧ bs (¬(ℳ a)))
      by auto
  have 30: ⊢ ((ℳ a) ∧ bs (¬(ℳ a))) = ▷(ℳ a)
      by (simp add: first-d-def)
  have 31: ⊢ ▷(ℳ a) = (ℳ a)
      using MFixFst by fastforce
  have 32: ⊢ ℳ(a UPTO (a THRU b)) =
          ((di(ℳ a)) ∧ ((ℳ a) ∨ di(ℳ b)) ∧
          bs (¬( (di(ℳ a)) ∧ ((ℳ a) ∨ di(ℳ b)))))
      using 1 2 6 7 by fastforce
  have 33: ⊢ ((di(ℳ a)) ∧ ((ℳ a) ∨ di(ℳ b)) ∧
          bs (¬( (di(ℳ a)) ∧ ((ℳ a) ∨ di(ℳ b))))) =
           (((ℳ a)) ∧  bs(¬(di(ℳ a) ∧ di(ℳ b))))
      using 15 16 17 18 19 by (metis int-eq)
  have 34: ⊢ (((ℳ a)) ∧  bs(¬(di(ℳ a) ∧ di(ℳ b)))) = (ℳ a)
      using 26 27 28 29 30 31 by (metis int-eq)
 from 32 33 34 show ?thesis using MonEq by (metis int-eq)
qed

lemma MThruUptoAbsorp:
 (a THRU (a UPTO b)) ≃ (a)
proof −
  have 1: ⊢ ℳ(a THRU (a UPTO b)) = ▷(di(ℳ a) ∧ di(▷((ℳ a) ∨ (ℳ b))))
      by simp
  have 2: ⊢  ▷(di(ℳ a) ∧ di(▷((ℳ a) ∨ (ℳ b)))) =
            ▷(di(ℳ a) ∧ di(((ℳ a) ∨ (ℳ b))))
    by (metis DiEqvDiFst FstEqvRule inteq-reflection lift-and-com)
  have 3: ⊢ ▷(di(ℳ a) ∧ di(((ℳ a) ∨ (ℳ b)))) =
          ▷(di(ℳ a) ∧ (di(ℳ a) ∨ di(ℳ b)))
     by (metis DiOrEqv FstEqvRule inteq-reflection lift-and-com)
  have 4: ⊢ (di(ℳ a) ∧ (di(ℳ a) ∨ di(ℳ b))) = (di(ℳ a))
       by auto
  hence 5: ⊢ ▷(di(ℳ a) ∧ (di(ℳ a) ∨ di(ℳ b))) = ▷(di(ℳ a))
      using FstEqvRule by blast
  have 6: ⊢ ▷(di(ℳ a)) = ▷(ℳ a)
      using FstDiEqvFst by blast
  have 7: ⊢ ▷(ℳ a) = (ℳ a)
      using MFixFst by fastforce
 from 1 2 3 5 6 7 show ?thesis using MonEq by (metis int-eq)
qed

lemma MUptoThruDistrib:
 (a UPTO (b THRU c)) ≃ ((a UPTO b) THRU (a UPTO c))
```

```
proof −
  have 1: ⊢ 𝓜((a UPTO b) THRU (a UPTO c)) =
           ▷( di(▷((𝓜 a) ∨ (𝓜 b))) ∧ di(▷((𝓜 a) ∨ (𝓜 c))) )
      by simp
  have 2: ⊢ ( di(▷((𝓜 a) ∨ (𝓜 b))) ∧ di(▷((𝓜 a) ∨ (𝓜 c))) ) =
           ( di(((𝓜 a) ∨ (𝓜 b))) ∧ di(((𝓜 a) ∨ (𝓜 c))) )
      using DiEqvDiFst by fastforce
  have 3: ⊢ ( di(((𝓜 a) ∨ (𝓜 b))) ∧ di(((𝓜 a) ∨ (𝓜 c))) ) =
           (( di(𝓜 a) ∨ di(𝓜 b)) ∧ (di(𝓜 a) ∨ di(𝓜 c)))
      using DiOrEqv by fastforce
  have 4: ⊢ (( di(𝓜 a) ∨ di(𝓜 b)) ∧ (di(𝓜 a) ∨ di(𝓜 c))) =
          (di(𝓜 a) ∨ (di(𝓜 b) ∧ di(𝓜 c)))
      by auto
  have 5: ⊢ ( di(▷((𝓜 a) ∨ (𝓜 b))) ∧ di(▷((𝓜 a) ∨ (𝓜 c))) ) =
           (di(𝓜 a) ∨ (di(𝓜 b) ∧ di(𝓜 c)))
       using 2 3 4 by fastforce
  hence 6: ⊢ ▷( di(▷((𝓜 a) ∨ (𝓜 b))) ∧ di(▷((𝓜 a) ∨ (𝓜 c))) ) =
           ▷( di(𝓜 a) ∨ (di(𝓜 b) ∧ di(𝓜 c)) )
      using FstEqvRule by blast
  have 7: ⊢ ▷( di(𝓜 a) ∨ (di(𝓜 b) ∧ di(𝓜 c)) ) =
           ▷( ▷(di(𝓜 a)) ∨ ▷(di(𝓜 b) ∧ di(𝓜 c)) )
      using FstFstOrEqvFstOr by fastforce
  have 8: ⊢ ▷(di(𝓜 a)) = ▷((𝓜 a))
      using FstDiEqvFst  by blast
  have 9: ⊢ ▷((𝓜 a)) = (𝓜 a)
      using MFixFst by fastforce
  have 10: ⊢ ▷(di(𝓜 a)) = (𝓜 a)
      using 8 9 by fastforce
  hence 11: ⊢ (▷(di(𝓜 a)) ∨ ▷(di(𝓜 b) ∧ di(𝓜 c))) =
            ((𝓜 a) ∨ ▷(di(𝓜 b) ∧ di(𝓜 c)))
      by auto
  hence 12: ⊢ ▷(▷(di(𝓜 a)) ∨ ▷(di(𝓜 b) ∧ di(𝓜 c))) =
            ▷((𝓜 a) ∨ ▷(di(𝓜 b) ∧ di(𝓜 c)))
      using FstEqvRule by blast
  have 13: ⊢▷((𝓜 a) ∨ ▷(di(𝓜 b) ∧ di(𝓜 c))) = 𝓜(a UPTO (b THRU c))
      by simp
 from 1 6 7 12 13 show ?thesis using MonEq by (metis int-eq)
qed

lemma MThruUptoDistrib:
 (a THRU (b UPTO c)) ≃ ((a THRU b) UPTO (a THRU c))
proof −
  have 1: ⊢ 𝓜((a THRU b) UPTO (a THRU c)) =
         ▷(▷(di(𝓜 a) ∧ di(𝓜 b)) ∨ ▷(di(𝓜 a) ∧ di(𝓜 c)))
         by simp
  have 2: ⊢ ▷(▷(di(𝓜 a) ∧ di(𝓜 b)) ∨ ▷(di(𝓜 a) ∧ di(𝓜 c))) =
           ▷((di(𝓜 a) ∧ di(𝓜 b)) ∨ (di(𝓜 a) ∧ di(𝓜 c)))
         using FstFstOrEqvFstOr by auto
  have 3: ⊢ ((di(𝓜 a) ∧ di(𝓜 b)) ∨ (di(𝓜 a) ∧ di(𝓜 c))) =
           (di(𝓜 a) ∧ (di(𝓜 b) ∨ di(𝓜 c))) by auto
  have 4: ⊢ (di(ℳ a) ∧ (di(ℳ b) ∨ di(ℳ c))) =
           (di(ℳ a) ∧ di( (ℳ b) ∨ (ℳ c))) using DiOrEqv by fastforce
  have 5: ⊢ (di(ℳ a) ∧ di( (ℳ b) ∨ (ℳ c))) =
           (di(ℳ a) ∧ di(▷( (ℳ b) ∨ (ℳ c)))) using DiEqvDiFst by fastforce
  have 6: ⊢ ((di(ℳ a) ∧ di(ℳ b)) ∨ (di(ℳ a) ∧ di(ℳ c))) =
           (di(ℳ a) ∧ di(▷( (ℳ b) ∨ (ℳ c)))) using 3 4 5 by fastforce
 hence 7: ⊢ ▷((di(ℳ a) ∧ di(ℳ b)) ∨ (di(ℳ a) ∧ di(ℳ c))) =
           ▷(di(ℳ a) ∧ di(▷( (ℳ b) ∨ (ℳ c))))  using FstEqvRule by blast
  have 8: ⊢ ▷(di(ℳ a) ∧ di(▷( (ℳ b) ∨ (ℳ c)))) =
           ℳ(a THRU (b UPTO c)) by simp
 from 1 2 7 8 show ?thesis using MonEq by (metis int-eq)
qed

lemma MThruUptoRDistrib:
 ((a THRU b) UPTO c) ≃ ((a UPTO c) THRU (b UPTO c))
proof −
  have 1: ((a THRU b) UPTO c) ≃ (c UPTO (a THRU b))
      using MUptoCommut by auto
  have 2: (c UPTO (a THRU b)) ≃ ((c UPTO a) THRU (c UPTO b))
      using MUptoThruDistrib by auto
  have 3: (c UPTO a) ≃ (a UPTO c)
      using MUptoCommut by auto
  have 4: (c UPTO b) ≃ (b UPTO c)
      using MUptoCommut by auto
  have 5:  ((c UPTO a) THRU (c UPTO b)) ≃ ((a UPTO c) THRU (c UPTO b))
      using 3  by (simp add: MonEqRefl MonEqSubstThru)
  have 6:  ((a UPTO c) THRU (c UPTO b)) ≃ ((c UPTO b) THRU (a UPTO c))
      using MThruCommut by auto
  have 7:  ((c UPTO b) THRU (a UPTO c)) ≃ ((b UPTO c) THRU (a UPTO c))
      using 4 by (simp add: MonEqRefl MonEqSubstThru)
  from 1 2 5 6 7 show ?thesis using MThruCommut MonEq by (metis int-eq)
qed

lemma MUptoThruRDistrib:
 ((a UPTO b) THRU c) ≃ ((a THRU c) UPTO (b THRU c))
proof −
  have 1: ((a UPTO b) THRU c) ≃ (c THRU (a UPTO b))
      using MThruCommut by auto
  have 2: (c THRU (a UPTO b)) ≃ ((c THRU a) UPTO (c THRU b))
      using MThruUptoDistrib by auto
  have 3: (c THRU a) ≃ (a THRU c)
      using MThruCommut by auto
  have 4: (c THRU b) ≃ (b THRU c)
      using MThruCommut by auto
  have 5:  ((c THRU a) UPTO (c THRU b)) ≃ ((a THRU c) UPTO (c THRU b))
      using 3 by (simp add: MonEqRefl MonEqSubstUpto)
  have 6:  ((a THRU c) UPTO (c THRU b)) ≃ ((c THRU b) UPTO (a THRU c))
      using MUptoCommut by auto
  have 7:  ((c THRU b) UPTO (a THRU c)) ≃ ((b THRU c) UPTO (a THRU c))
      using 4 by (simp add: MonEqRefl MonEqSubstUpto)
  from 1 2 5 6 7 show ?thesis using MUptoCommut MonEq by (metis int-eq)
qed

lemma MWithAndDistrib:
 ((a AND b) WITH f) ≃ ((a WITH f) AND (b WITH f))
proof −
  have 1: ⊢ 𝓜((a AND b) WITH f) = (𝓜(a AND b) ∧ f)
       by (simp)
  have 2: ⊢ 𝓜(a AND b) = 𝓜(a WITH LIFT(𝓜 b))
       by (simp add: AND-d-def)
  have 3: ⊢ (𝓜(a AND b) ∧ f) = (𝓜(a WITH LIFT(𝓜 b)) ∧ f)
       using 2 by auto
  have 4: ⊢ 𝓜(a WITH (LIFT( (𝓜 b) ∧ f))) = (𝓜(a) ∧ 𝓜(b) ∧ f)
       by simp
  have 5: ⊢ (𝓜(a) ∧ 𝓜(b) ∧ f) = ((𝓜(a) ∧ f) ∧ (𝓜(b) ∧ f))
       by auto
  have 6: ⊢ ((𝓜(a) ∧ f) ∧ (𝓜(b) ∧ f)) = (𝓜(a WITH f) ∧ 𝓜( b WITH f))
       by simp
  have 7: ⊢ (𝓜(a WITH f) ∧ 𝓜(b WITH f)) = 𝓜((a WITH f) WITH LIFT(𝓜(b WITH f)))
       by simp
  have 8: ⊢ 𝓜((a WITH f) WITH LIFT(𝓜(b WITH f))) = 𝓜((a WITH f) AND (b WITH f))
       by (simp add: AND-d-def)
  from 1 2 3 4 5 6 7 8 show ?thesis using MonEq by (metis AND-d-def MWithAbsorp int-eq)
qed

lemma MHaltWithAndDistrib:
 (((HALT w) WITH f) AND ((HALT w) WITH g)) ≃ ((HALT w) WITH LIFT(f ∧ g))
proof −
  have 1: ⊢ 𝓜(((HALT w) WITH f) AND ((HALT w) WITH g)) =
             𝓜(((HALT w) WITH f) WITH LIFT(𝓜((HALT w) WITH g)))
       by (simp add: AND-d-def)
  have 2: ⊢ 𝓜(((HALT w) WITH f) WITH LIFT(𝓜((HALT w) WITH g))) =
             (𝓜(HALT w) ∧ f ∧ 𝓜(HALT w) ∧ g)
       by auto
  have 3: ⊢ (𝓜(HALT w) ∧ f ∧ 𝓜(HALT w) ∧ g) = (𝓜(HALT w) ∧ f ∧ g)
       by auto
  have 4: ⊢ (𝓜(HALT w) ∧ f ∧ g) = 𝓜((HALT w) WITH LIFT(f ∧ g))
       by auto
  from 1 2 3 4 show ?thesis using MonEq by (metis int-eq)
qed

lemma MHaltWithUptoHaltWithEqvHaltWithOr:
 (((HALT w) WITH f) UPTO ((HALT w) WITH g)) ≃ ((HALT w) WITH LIFT(f ∨ g))
proof −
  have 1: ⊢ 𝓜(((HALT w) WITH f) UPTO ((HALT w) WITH g)) =
             ▷(𝓜((HALT w) WITH f) ∨ 𝓜((HALT w) WITH g))
       by (simp)
  have 2: ⊢ ▷(𝓜((HALT w) WITH f) ∨ 𝓜((HALT w) WITH g)) =
             ▷((𝓜(HALT w) ∧ f) ∨ (𝓜(HALT w) ∧ g))
       by auto
  have 3: ⊢ ((ℳ(HALT w) ∧ f) ∨ (ℳ(HALT w) ∧ g)) = (ℳ(HALT w) ∧ (f ∨ g))
      by auto
  have 4: ⊢ ▷((ℳ(HALT w) ∧ f) ∨ (ℳ(HALT w) ∧ g)) = ▷(ℳ(HALT w) ∧ (f ∨ g))
      using 3 FstEqvRule by fastforce
  have 5: ⊢ ▷(ℳ(HALT w) ∧ (f ∨ g)) = ▷(ℳ((HALT w) WITH LIFT(f ∨ g)))
      by simp
  have 6: ⊢ ℳ(((HALT w) WITH LIFT(f ∨ g))) = ▷(ℳ((HALT w) WITH LIFT(f ∨ g)))
      using MFixFst by blast
  from 1 2 3 4 5 6 show ?thesis using MonEq by (metis int-eq)
qed

lemma MHaltWithThruHaltWithEqvHaltWithAndHaltWith:
 (((HALT w) WITH f) THRU ((HALT w) WITH g)) ≃ (((HALT w) WITH f) AND ((HALT w) WITH g))
proof −
  have 1: ⊢ ℳ(((HALT w) WITH f) THRU ((HALT w) WITH g)) =
          ▷( di(ℳ(HALT w) ∧ f) ∧ di(ℳ(HALT w) ∧ g) )
      by simp
  have 2: ⊢ (di(ℳ(HALT w) ∧ f) ∧ di(ℳ(HALT w) ∧ g)) =
          (di(halt(init w) ∧ f) ∧ di(halt(init w) ∧ g))
      using MHaltAlt DiEqvDi
      by (metis (no-types, lifting) inteq-reflection lift-and-com)
  have 3: ⊢ (di(halt(init w) ∧ f) ∧ di(halt(init w) ∧ g)) =
          di(halt(init w) ∧ f ∧ g)
      using DiHaltAndDiHaltAndEqvDiHaltAndAnd by fastforce
  have 4: ⊢ di( halt(init w) ∧ f ∧ g) = di(ℳ(HALT w) ∧ f ∧ g)
     by (metis DiEqvDi MHaltAlt inteq-reflection lift-and-com)
  have 5: ⊢ (di(ℳ(HALT w) ∧ f) ∧ di(ℳ(HALT w) ∧ g)) = di(ℳ(HALT w) ∧ f ∧ g)
      using 2 3 4 by fastforce
  have 6: ⊢ ▷(di(ℳ(HALT w) ∧ f) ∧ di(ℳ(HALT w) ∧ g)) = ▷(di(ℳ(HALT w) ∧ f ∧ g))
      using 5 FstEqvRule by blast
  have 7: ⊢ ▷(di(ℳ(HALT w) ∧ f ∧ g)) = ▷(ℳ(HALT w) ∧ f ∧ g)
      using FstDiEqvFst by fastforce
  have 8: ⊢ ▷(ℳ(HALT w) ∧ f ∧ g) = ▷(ℳ((HALT w) WITH LIFT(f ∧ g)))
      by simp
  have 9: ⊢ ℳ((HALT w) WITH LIFT(f ∧ g)) = ▷(ℳ((HALT w) WITH LIFT(f ∧ g)))
      using MFixFst by blast
  have 10: ⊢ ℳ(((HALT w) WITH f) THRU ((HALT w) WITH g)) = ℳ((HALT w) WITH LIFT(f ∧ g))
      using 1 2 3 4 5 6 7 8 9 int-eq by metis
  have 11: ⊢ ℳ(((HALT w) WITH f) AND ((HALT w) WITH g)) = ℳ((HALT w) WITH LIFT(f ∧ g))
      using MHaltWithAndDistrib using eq-d-def by blast
  have 12: ⊢ ℳ((HALT w) WITH LIFT(f ∧ g)) = ℳ(((HALT w) WITH f) AND ((HALT w) WITH g))
      using 11 by fastforce
  from 10 12 show ?thesis using MonEq by (metis int-eq)
qed

lemma MThenAndDistrib:
 (a THEN (b AND c)) ≃ ((a THEN b) AND (a THEN c))
proof −
  have 1: ⊢ ℳ(a THEN (b AND c)) = (ℳ(a)) ; (ℳ(b AND c))
      by simp
  have 2: ⊢ (𝓜(a)) ; (𝓜(b AND c)) = (𝓜(a)) ; (𝓜(b) ∧ 𝓜(c))
       by (simp add: AND-d-def)
  have 3: ⊢ (𝓜(a)) ; (𝓜(b) ∧ 𝓜(c)) =  ▷ (𝓜(a)) ; (𝓜(b) ∧ 𝓜(c))
       using MFixFst LeftChopEqvChop by blast
  have 4: ⊢ ▷ (𝓜(a)) ; (𝓜(b) ∧ 𝓜(c)) =  ((▷ (𝓜(a)) ; (𝓜(b))) ∧ (▷ (𝓜(a)) ; (𝓜(c))))
    using LFstAndDistrC by fastforce
  have 5: ⊢((▷ (𝓜(a)) ; (𝓜(b))) ∧ (▷ (𝓜(a)) ; (𝓜(c)))) =
          (( (𝓜(a)) ; (𝓜(b))) ∧ ( (𝓜(a)) ; (𝓜(c))) )  using MFixFst
          by (metis 4 inteq-reflection)
  have 6: ⊢ (( (𝓜(a)) ; (𝓜(b))) ∧ ( (𝓜(a)) ; (𝓜(c))) ) =
           (𝓜(a THEN b) ∧ 𝓜(a THEN c))
       by simp
  have 7: ⊢ (𝓜(a THEN b) ∧ 𝓜(a THEN c)) = 𝓜((a THEN b) AND (a THEN c))
        by (simp add: AND-d-def)
  from 1 2 3 4 5 6 7 show ?thesis using MonEq by (metis int-eq)
qed

lemma MThenUptoDistrib:
 (a THEN (b UPTO c)) ≃ ((a THEN b) UPTO (a THEN c))
proof −
 have 1: ⊢ (𝓜 (a THEN (b UPTO c))) = ((𝓜 a);(▷((𝓜 b) ∨ (𝓜 c))))
      by simp
 have 2: ⊢ ((𝓜 a);(▷((𝓜 b) ∨ (𝓜 c)))) = (▷(𝓜 a);(▷((𝓜 b) ∨ (𝓜 c))))
      by (simp add: MFixFst LeftChopEqvChop)
 have 3: ⊢ (▷(𝓜 a);(▷((𝓜 b) ∨ (𝓜 c)))) = ((▷(▷(𝓜 a);((𝓜 b) ∨ (𝓜 c)))))
      using FstFstChopEqvFstChopFst by fastforce
 have 4: ⊢ ▷(𝓜 a);((𝓜 b) ∨ (𝓜 c)) = (𝓜 a);((𝓜 b) ∨ (𝓜 c))
      using MFixFst by (metis LeftChopEqvChop inteq-reflection)
 have 5: ⊢ (𝓜 a);((𝓜 b) ∨ (𝓜 c)) = ((𝓜 a);(𝓜 b) ∨ (𝓜 a);(𝓜 c))
      by (simp add: ChopOrEqv)
 have 6: ⊢ ((𝓜 a);(𝓜 b) ∨ (𝓜 a);(𝓜 c)) = (𝓜(a THEN b) ∨ 𝓜(a THEN c))
      by simp
 have 7: ⊢ ▷(𝓜 a);((𝓜 b) ∨ (𝓜 c)) = (𝓜(a THEN b) ∨ 𝓜(a THEN c))
      using 6 5 4 by fastforce
 have 8: ⊢ ▷(▷(𝓜 a);((𝓜 b) ∨ (𝓜 c))) = ▷(𝓜(a THEN b) ∨ 𝓜(a THEN c))
      using 7 by (simp add: FstEqvRule)
 have 9: ⊢ ▷(𝓜(a THEN b) ∨ 𝓜(a THEN c)) = 𝓜((a THEN b) UPTO (a THEN c))
      by simp
 from 9 7 1 2 3 show ?thesis by (metis eq-d-def inteq-reflection)
qed

lemma MThenThruDistrib:
 (a THEN (b THRU c)) ≃ ((a THEN b) THRU (a THEN c))
proof −
 have 1: ⊢ 𝓜(a THEN (b THRU c)) = (𝓜 a);▷(di(𝓜 b) ∧ di(𝓜 c))
      by simp
 have 2: ⊢ (𝓜 a);▷(di(𝓜 b) ∧ di(𝓜 c)) = ▷(𝓜 a);▷(di(𝓜 b) ∧ di(𝓜 c))
      by (simp add: MFixFst LeftChopEqvChop)
 have 3: ⊢ ▷(𝓜 a);▷(di(𝓜 b) ∧ di(𝓜 c)) = ▷(▷(𝓜 a);(di(𝓜 b) ∧ di(𝓜 c) ))
      using FstFstChopEqvFstChopFst by fastforce
 have 4: ⊢ ▷(𝓜 a);(di(𝓜 b) ∧ di(𝓜 c) ) = (▷(𝓜 a);di(𝓜 b) ∧ ▷(𝓜 a);di(𝓜 c))
     by (meson LFstAndDistrC Prop11)
 have 5: ⊢ (▷(𝓜 a);di(𝓜 b) ∧ ▷(𝓜 a);di(𝓜 c)) = ((𝓜 a);di(𝓜 b) ∧ (𝓜 a);di(𝓜 c))
     using MFixFst by (metis 4 int-eq)
 have 6: ⊢ (𝓜 a);di(𝓜 b) = (𝓜 a);((𝓜 b);#True)
     by (simp add: di-d-def)
 have 7: ⊢ (𝓜 a);((𝓜 b);#True) = ((𝓜 a);(𝓜 b));#True
     by (simp add: ChopAssoc)
 have 8: ⊢ ((𝓜 a);(𝓜 b));#True = di((𝓜 a);(𝓜 b))
     by (simp add: di-d-def)
 have 9: ⊢ (𝓜 a);di(𝓜 b) = di((𝓜 a);(𝓜 b))
     using 8 7 6 by fastforce
 have 10: ⊢ (𝓜 a);di(𝓜 c) = (𝓜 a);((𝓜 c);#True)
     by (simp add: di-d-def)
 have 11: ⊢ (𝓜 a);((𝓜 c);#True) = ((𝓜 a);(𝓜 c));#True
     by (simp add: ChopAssoc)
 have 12: ⊢ ((𝓜 a);(𝓜 c));#True = di((𝓜 a);(𝓜 c))
     by (simp add: di-d-def)
 have 13: ⊢ (𝓜 a);di(𝓜 c) = di((𝓜 a);(𝓜 c))
     using 12 11 10 by fastforce
 have 14: ⊢ ((𝓜 a);di(𝓜 b) ∧ (𝓜 a);di(𝓜 c)) = (di((𝓜 a);(𝓜 b)) ∧ di((𝓜 a);(𝓜 c)))
     using 13 9 by fastforce
 have 15: ⊢ (di((𝓜 a);(𝓜 b)) ∧ di((𝓜 a);(𝓜 c))) = (di(𝓜(a THEN b)) ∧ di(𝓜(a THEN c)))
     by simp
 have 16: ⊢ ▷(𝓜 a);(di(𝓜 b) ∧ di(𝓜 c) ) = (di(𝓜(a THEN b)) ∧ di(𝓜(a THEN c)))
     using 15 14 4 5 by fastforce
 have 17: ⊢ ▷(▷(𝓜 a);(di(𝓜 b) ∧ di(𝓜 c) )) = ▷(di(𝓜(a THEN b)) ∧ di(𝓜(a THEN c)))
     using 16 by (simp add: FstEqvRule)
 have 18: ⊢ ▷(di(𝓜(a THEN b)) ∧ di(𝓜(a THEN c))) = 𝓜((a THEN b) THRU (a THEN c))
     by simp
 from 18 16 1 2 3 show ?thesis by (metis eq-d-def int-eq)
qed
```

```
end
```

```
theory Example
imports
  FOTheorems
begin
```

8 Examples

8.1 Example 1

```
definition F1 :: nat statefun ⇒ temporal
where  F1 w ≡ TEMP □ ( #0 ≤ $w )
```

definition *Init1* :: *nat statefun* ⇒ *temporal*
where *Init1 w* ≡ *TEMP* $*w* = #*0*

lemma *init1*:
(⟨*s0*,*s1*,*s2*⟩ ⊨ *len*(*2*) ∧ *Init1 w*) = ((*w s0*) = *0*)
by (*simp add*: *Init1-def current-val-d-def len-defs*)

lemma *exist-test-F1* :
⊢ ∃∃ *w*. *F1 w*
proof −
have *1*: ⋀ *w*. ⊢ *F1 w* **by** (*simp add*: *always-defs current-val-d-def F1-def Valid-def*)
from *1* **show** *?thesis* **using** *EExI*[*unlift-rule*] **by** *blast*
qed

8.2 Example 2

locale *Test* =
fixes *v* :: *state* ⇒ *nat*
fixes *v1* :: *state* ⇒ *nat*
fixes *y* :: *state* ⇒ *bool*
fixes *z* :: *state* ⇒ *int*
fixes *F2* :: *nat statefun* ⇒ *temporal*
fixes *F3* :: *bool statefun* ⇒ *temporal*
fixes *F4* :: *int statefun* ⇒ *temporal*
fixes *F5* :: *nat statefun* ⇒ *temporal*
fixes *Init2* :: *nat statefun* ⇒ *temporal*
fixes *Init3* :: *bool statefun* ⇒ *temporal*
defines *F2* ≡ (λ *v*. *TEMP* □ (#*0* ≤ $*v*))
defines *F3* ≡ (λ *p*. *TEMP* □ ($*p* ∨ ¬ $*p*))
defines *F4* ≡ (λ *z*. *TEMP* □ (#*0*≤ $*z* ∨ $*z* < #*0*))
defines *F5* ≡ (λ *v*. *TEMP* $*v*=#*0* ∧ *v gets* $*v*+#*1*)
defines *Init2* ≡ (λ *v*. *TEMP* $*v* = #*0*)
defines *Init3* ≡ (λ *p*. *TEMP* $*p*)

lemma (**in** *Test*) *currentval-test* :
(*s* ⊨ ($*v* = #*0*)) = ((*v* (*nth s 0*)) = *0*)
by (*simp add*: *current-val-d-def*)

lemma (**in** *Test*) *nextempty-test* :
(⟨*s0*⟩ ⊨ *v*$) = (ϵ *x*. *x*=*x*)
by (*simp add*: *next-val-d-def*)

lemma (**in** *Test*) *nextempty-test-1* :
(⟨*s0*⟩ ⊨ *v*$ = *v*$)
by *simp*

lemma (**in** *Test*) *nextempty-test-2* :
(⟨*s0*⟩ ⊨ *v*$ = *v1*$)

```
by (simp add: Test.nextempty-test)

lemma (in Test) nextcurrent-test:
  (⟨s0,s1⟩ ⊨ skip ∧ ($v=#0) ∧ (v$=$v+#1)) = (((v s0) = 0) ∧ ((v s1) = 1  ))
unfolding current-val-d-def next-val-d-def skip-defs by auto

lemma (in Test) nextcurrentfinpenult-test:
  (⟨s0,s1,s2,s3⟩ ⊨ len(3) ∧ v =: !v−#1 ∧ v ← #3 ∧ $v=#0 ∧ v := $v+#1 ) =
     (((v s0) = 0) ∧ ((v s1) = 1 ∧ (v (s2)) = 2 ∧ ((v s3) = 3   )))
unfolding current-val-d-def next-val-d-def fin-val-d-def penult-val-d-def
          next-assign-d-def prev-assign-d-def temporal-assign-d-def len-defs by auto


lemma (in Test) stable-test:
 (⟨s0,s1,s2,s3⟩ ⊨ len(3) ∧ stable v ∧ $v=#0) =
   ((v s0) = 0 ∧ (v s1) = 0 ∧ (v s2) = 0 ∧ (v s3) = 0)
by (auto simp: stable-defs len-defs
           current-val-d-def next-val-d-def Nitpick.case-nat-unfold)

lemma (in Test) revnextcurrentfinpenult-test:
  (⟨s0,s1,s2,s3⟩ ⊨ (len 3 ∧ v! = !v−#1 ∧ !v = #3 ∧ $v=#0 ∧ v$=$v+#1)^r) =
     (((v s3) = 0) ∧ ((v s2) = 1 ∧ (v (s1)) = 2 ∧ ((v s0) = 3   )))
unfolding reverse-d-def len-defs current-val-d-def next-val-d-def
          penult-val-d-def fin-val-d-def  by auto

lemma (in Test) exist-test-F2 :
 ⊢ ∃∃  v. F2 v
proof −
 have 1: ⊢ F2 v by (simp add: always-defs current-val-d-def F2-def Valid-def)
 from 1 show ?thesis using EExI[unlift-rule] by blast
qed

lemma (in Test) exist-test-F3 :
 ⊢ ∃∃  y. F3 y
proof −
 have 1: ⊢ F3 y by (simp add: always-defs current-val-d-def F3-def Valid-def)
 from 1 show ?thesis using EExI[unlift-rule] by blast
qed
```

8.3 Example 3

```
locale Test1 =
 fixes v :: state ⇒ nat
 fixes F5 :: nat statefun ⇒ nat ⇒ temporal
 defines F5 ≡ (λ v n. TEMP $v=#0 ∧ v gets $v+#1 ∧ fin($v=#n))



lemma (in Test1) test-E-F5-1:
 (
           x (Interval.nth w (0::nat)) = (0::nat) ∧
```

```
          (∀ i<intlen w. x (Interval.nth w (Suc i)) = Suc (x (Interval.nth w i))) ∧
          x (Interval.nth w (intlen w)) = n) ⟶
  (
          x (Interval.nth w (0::nat)) = (0::nat) ∧
          (∀ i≤intlen w. x (Interval.nth w (i)) = i) ∧
          x (Interval.nth w (intlen w)) = n)
apply simp
proof
  assume 0: x (Interval.nth w (0::nat)) = (0::nat) ∧
   (∀ i<intlen w. x (Interval.nth w (Suc i)) = Suc (x (Interval.nth w i))) ∧
   x (Interval.nth w (intlen w)) = n
  have 1: x (Interval.nth w (0::nat)) = (0::nat)  using 0 by auto
  have 2: x (Interval.nth w (intlen w)) = n  using 0 by auto
  have 3: (∀ i<intlen w. x (Interval.nth w (Suc i)) = Suc (x (Interval.nth w i))) using 0 by auto
  show ∀ i≤intlen w. x (Interval.nth w i) = i
  proof
    fix i
    show i ≤ intlen w ⟶ x (Interval.nth w i) = i
    proof
     (induct i)
     case 0
     then show ?case using 1 by simp
     next
     case (Suc i)
     then show ?case by (simp add: 3)
    qed
  qed
qed

lemma (in Test1) test-E-F5-2:
 (
          x (Interval.nth w (0::nat)) = (0::nat) ∧
          (∀ i≤intlen w. x (Interval.nth w (i)) = i) ∧
          x (Interval.nth w (intlen w)) = n) ⟶ (
          x (Interval.nth w (0::nat)) = (0::nat) ∧
          (∀ i<intlen w. x (Interval.nth w (Suc i)) = Suc (x (Interval.nth w i))) ∧
          x (Interval.nth w (intlen w)) = n)

by simp

lemma (in Test1) test-E-F5-3:
 (
          x (Interval.nth w (0::nat)) = (0::nat) ∧
          (∀ i<intlen w. x (Interval.nth w (Suc i)) = Suc (x (Interval.nth w i))) ∧
          x (Interval.nth w (intlen w)) = n) =
  (
          x (Interval.nth w (0::nat)) = (0::nat) ∧
          (∀ i≤intlen w. x (Interval.nth w (i)) = i) ∧
          x (Interval.nth w (intlen w)) = n)
using test-E-F5-1 test-E-F5-2 by auto
```

```
lemma (in Test1) test-E-F5-4:
 (∃x::state ⇒ nat.
          x (Interval.nth w (0::nat)) = (0::nat) ∧
          (∀i<intlen w. x (Interval.nth w (Suc i)) = Suc (x (Interval.nth w i))) ∧
          x (Interval.nth w (intlen w)) = n) =
   (∃x::state ⇒ nat.
          x (Interval.nth w (0::nat)) = (0::nat) ∧
          (∀i≤intlen w. x (Interval.nth w (i)) = i) ∧
          x (Interval.nth w (intlen w)) = n)
by (simp add: Test1.test-E-F5-3)
```

```
lemma (in Test1) test-E-F5:
 ⊢ (∃∃ v. (F5 v n)) ⟶ (len n)
apply (simp add: Valid-def F5-def exist-state-d-def gets-defs current-val-d-def
                 fin-defs sub-def len-defs)
proof
  fix w
  show (∃x::state ⇒ nat.
          x (Interval.nth w (0::nat)) = (0::nat) ∧
          (∀i<intlen w. x (Interval.nth w (Suc i)) = Suc (x (Interval.nth w i))) ∧
          x (Interval.nth w (intlen w)) = n) ⟶
      (intlen w = n)
  proof −
   have 1: (∃x::state ⇒ nat.
          x (Interval.nth w (0::nat)) = (0::nat) ∧
          (∀i<intlen w. x (Interval.nth w (Suc i)) = Suc (x (Interval.nth w i))) ∧
          x (Interval.nth w (intlen w)) = n) =
          (∃x::state ⇒ nat.
          x (Interval.nth w (0::nat)) = (0::nat) ∧
          (∀i≤intlen w. x (Interval.nth w (i)) = i) ∧
          x (Interval.nth w (intlen w)) = n)  using test-E-F5-4 by auto
   have 2: (∃x::state ⇒ nat.
          x (Interval.nth w (0::nat)) = (0::nat) ∧
          (∀i≤intlen w. x (Interval.nth w (i)) = i) ∧
          x (Interval.nth w (intlen w)) = n) ⟶ (intlen w =n)
          by auto
   from 1 2 show ?thesis by auto
 qed
qed
```

8.4 Example 4

```
locale Testrev =
 fixes x :: state ⇒ nat
 fixes F1 :: nat statefun ⇒ temporal
 defines F1 ≡ (λ v. TEMP $v=#0 ∧ skip ∧ v:= $v+#1 )
```

```
lemma (in Testrev) testrev1:
  (σ ⊨ F1 (x)) = (intlen σ = 1 ∧ (x (nth σ 0)) = 0 ∧ (x (nth σ 1)) = 1)
by (simp add: F1-def skip-defs next-assign-d-def next-val-d-def current-val-d-def, auto)

lemma (in Testrev) testrev2:
  (σ ⊨ (F1 (x))^r) = (intlen σ = 1 ∧ (x (nth σ 0)) = 1 ∧ (x (nth σ 1)) = 0)
proof −
 have  (σ ⊨ (F1 (x))^r) = (σ ⊨ ($x=#0 ∧ skip ∧ x := $x+#1)^r)
      by (simp add: F1-def)
 also have  ... =
         (σ ⊨ (($x=#0)^r ∧ skip^r ∧ (x := $x+#1)^r))
      by (simp add: all-rev-eq)
 also have ... =
         (σ ⊨ ((!x=#0) ∧ skip ∧ (x! = !x+#1)))
      by (smt RRAnd all-rev-eq(1) all-rev-eq(10) all-rev-eq(11) all-rev-eq(12)
                   all-rev-eq(3) int-eq next-assign-d-def)
 also have  ... =
         (σ ⊨ ((x$=#0) ∧ skip ∧ ($x = x$+#1)))
      by (simp add: skip-defs next-val-d-def finval-defs penultval-defs current-val-d-def, auto)
 also have ... =
         (intlen σ = 1 ∧ (x (nth σ 0)) = 1 ∧ (x (nth σ 1)) = 0)
      by (simp add: skip-defs next-val-d-def current-val-d-def, auto)
 finally show (σ ⊨ (F1 (x))^r) = (intlen σ = 1 ∧ (x (nth σ 0)) = 1 ∧ (x (nth σ 1)) = 0) .
qed
```

8.5 Example 5

```
lemma revnextcurrentfinpenult:
⊢  (v$ = $v)^r = (v!=!v)
proof −
 have 1: ⊢ (v$ = $v)^r = ( (v$)^r = ($v)^r) by (simp add: rev-fun2)
 have 2: ⊢ ((v$)^r =  (v!)) by (simp add: rev-next)
 have 3: ⊢ (($v)^r =  (!v)) by (simp add: rev-current)
 have 4:  ⊢  (( (v$)^r = ($v)^r) = ( (v!) = (!v) )) by (metis 1 2 3 inteq-reflection)
 from 1 4 show ?thesis by fastforce
qed



end


theory MonitorExample
imports
  FOTheorems Monitor
begin
```

9 Example

```
locale Test =
 fixes v :: state ⇒ nat
 fixes y :: state ⇒ bool
 fixes z :: state ⇒ nat
 fixes F2 :: nat statefun ⇒ temporal
 fixes F3 :: bool statefun ⇒ temporal
 fixes F4 :: nat statefun ⇒ temporal
 fixes F5 :: nat statefun ⇒ temporal
 fixes Init2 :: nat statefun ⇒ temporal
 fixes Init3 :: bool statefun ⇒ temporal
 fixes Mon1  :: state  monitor
 fixes Mon2  :: state monitor
 fixes Mon3  :: state monitor
 fixes Mon4  :: state monitor
 fixes Mon5  :: state monitor
 fixes Mon6  :: state monitor
 defines F2 ≡ (λ v. TEMP □ ( #0 ≤ $v ))
 defines F3 ≡ (λ p. TEMP □ ( $p ∨ ¬ $p))
 defines F4 ≡ (λ z. TEMP $z=#0 ∧ z gets $z+#1)
 defines F5 ≡ (λ z. TEMP fin($z=#4))
 defines Init2 ≡ (λ v. TEMP $v = #0)
 defines Init3 ≡ (λ p. TEMP $p)
 defines Mon1 ≡ FIRST( F2 v )
 defines Mon2 ≡ EMPTY UPTO Mon1
 defines Mon3 ≡ Mon1 WITH (F2 v)
 defines Mon4 ≡ Mon2 THEN Mon1
 defines Mon5 ≡ Mon3 THRU Mon4
 defines Mon6 ≡ (FIRST F4 z) WITH (F5 z)

lemma (in Test) test:
 ⊢ ℳ(Mon1) =  empty
proof −
 have 1: ⊢ ℳ(Mon1) = ▷(□ ( #0 ≤ $v ))
     using F2-def Mon1-def by fastforce
 have 2: ⊢ □ ( #0 ≤ $v )
     by (simp add: Valid-def always-defs current-val-d-def)
 have 3: ⊢ ▷(□ ( #0 ≤ $v )) = empty
     using 2 by (metis FstTrue int-eq int-eq-true)
 from 1 2 3 show ?thesis by fastforce
qed

lemma (in Test) test1:
⊢ ℳ(Mon2) = empty
proof −
 have 1: ⊢ ℳ(Mon2) = ℳ(EMPTY UPTO Mon1)
     using Mon2-def by fastforce
 have 2: ⊢ ℳ(EMPTY UPTO Mon1) = ▷(ℳ(EMPTY) ∨ ℳ(Mon1))
     by fastforce
 have 3: ⊢ ▷(ℳ(EMPTY) ∨ ℳ(Mon1)) = ▷(empty ∨ empty)
     using test by (metis 2 MEmptyAlt int-eq)
 have 4: ⊢ ▷(empty ∨ empty) = empty
     using FstEmptyOrEqvEmpty by blast
 from 1 2 3 4 show ?thesis by fastforce
qed

lemma (in Test) test2:
⊢ 𝓜(Mon3) = empty
proof –
 have 1: ⊢ 𝓜(Mon3) = 𝓜(Mon1 WITH (F2 v)) using Mon3-def by fastforce
 have 2: ⊢ 𝓜(Mon1 WITH (F2 v)) = (𝓜(Mon1) ∧ (F2 v)) by fastforce
 have 3: ⊢ (𝓜(Mon1) ∧ (F2 v)) = (empty ∧ (F2 v))  using test by fastforce
 have 4: ⊢ (F2 v) by (simp add: F2-def Valid-def always-defs current-val-d-def)
 have 5: ⊢ (empty ∧ (F2 v)) = empty using 4 by fastforce
 from 1 2 3 5 show ?thesis by fastforce
qed

lemma (in Test) test3:
⊢ 𝓜(Mon4) = empty
proof –
 have 1: ⊢ 𝓜(Mon4) = 𝓜(Mon2 THEN Mon1)
      using Mon4-def by fastforce
 have 2: ⊢ 𝓜(Mon2 THEN Mon1) = ( 𝓜(Mon2) ) ;( 𝓜(Mon1) )
      by fastforce
 have 3: ⊢ ( 𝓜(Mon2) ) ;( 𝓜(Mon1) ) = empty;empty
      using test test1 using ChopEqvChop by blast
 have 4: ⊢ empty; empty = empty
      by (simp add: ChopEmpty)
 from 1 2 3 4 show ?thesis by fastforce
qed

lemma (in Test) test4:
⊢ 𝓜(Mon5) = empty
proof –
 have 1: ⊢ 𝓜(Mon5) = 𝓜(Mon3 THRU Mon4)
      using Mon5-def by fastforce
 have 2: ⊢ 𝓜(Mon3 THRU Mon4) = ▷(di(𝓜(Mon3)) ∧ di(𝓜(Mon4)))
      by fastforce
 have 3: ⊢ (di(𝓜(Mon3)) ∧ di(𝓜(Mon4))) = (di(empty) ∧ di(empty))
      using test3 test2 by (metis inteq-reflection lift-and-com)
 hence 4: ⊢ ▷(di(𝓜(Mon3)) ∧ di(𝓜(Mon4))) = ▷(di(empty) ∧ di(empty))
      by (simp add: FstEqvRule)
 have 5: ⊢ ▷(di(empty) ∧ di(empty)) = ▷(di(empty))
      by simp
 have 6: ⊢ ▷(di(empty)) = empty
      using FstDiEqvFst FstEmpty by fastforce
 from 6 5 4 2 1 show ?thesis by fastforce
qed

lemma (in Test) test5:
⊢ 𝓜 (Mon6) = (▷($z=#0  ∧ z gets $z+#1) ∧ fin($z=#4) )
proof –
 have 1: ⊢ 𝓜(Mon6) = (𝓜(FIRST F4 z) ∧ (F5 z))
```

```
     using Mon6-def by fastforce
 have 2: ⊢ (ℳ(FIRST F4 z) ∧ (F5 z)) = (▷(F4 z) ∧ fin($z=#4))
     using F5-def by fastforce
 have 3: ⊢ (▷(F4 z) ∧ fin($z=#4)) = (▷($z=#0 ∧  z gets $z+#1) ∧ fin($z=#4) )
     using F4-def by fastforce
 from 1 2 3 show ?thesis by fastforce
qed

lemma (in Test) test5-1:
⊢ ▷($z=#0 ∧  z gets $z+#1) ∧ fin($z=#4)  ⟶
   ▷(($z=#0 ∧ z gets $z+#1) ∧ fin($z=#4))

using FstWithAndImp by blast

lemma (in Test) test5-2:
 (s ⊨ ($z=#0 ∧ z gets $z+#1) ∧ fin($z=#4)) =
  (z (nth s 0) =0  ∧ (∀  i< intlen s. z (nth s (Suc i)) = Suc(z (nth s i))) ∧
   z (nth s (intlen s)) = 4)
by (simp add: gets-defs fin-defs current-val-d-def sub-def)

lemma (in Test) test5-3:
  (z (nth s 0) =0  ∧ (∀  i< intlen s. z (nth s (Suc i)) = Suc(z (nth s i))) ∧
   z (nth s (intlen s)) = 4)
  ⟹
 (z (nth s 0) =0  ∧ (∀  i≤ intlen s. z (nth s i) = i)
  ∧ z (nth s (intlen s)) = 4)

proof −
  assume 0: (z (nth s 0) =0  ∧ (∀  i< intlen s. z (nth s (Suc i)) = Suc(z (nth s i))) ∧
   z (nth s (intlen s)) = 4)
  show (z (nth s 0) =0  ∧ (∀  i≤ intlen s. z (nth s i) = i)
  ∧ z (nth s (intlen s)) = 4)
  proof −
    have 1: z (nth s 0) =0  using 0 by auto
    have 2: z (nth s (intlen s)) = 4 using 0 by auto
    have 3: (∀  i≤ intlen s. z (nth s i) = i)
    proof
      fix i
      show i ≤ intlen s ⟶ z (Interval.nth s i) = i
      proof
       (induct i)
       case 0
       then show ?case by (simp add: 1)
       next
       case (Suc i)
       then show ?case by (simp add: 0)
       qed
    qed
    from 1 2 3 show ?thesis by auto
   qed
```

```
qed

lemma (in Test) test5-4:
   (z (nth s 0) =0  ∧ (∀  i≤ intlen s. z (nth s i) = i)
  ∧ z (nth s (intlen s)) = 4) ⟹
    (z (nth s 0) =0  ∧ (∀  i< intlen s. z (nth s (Suc i)) = Suc(z (nth s i))) ∧
   z (nth s (intlen s)) = 4)

proof −
 assume 0: (z (nth s 0) =0  ∧ (∀  i≤ intlen s. z (nth s i) = i)
  ∧ z (nth s (intlen s)) = 4)
 show (z (nth s 0) =0  ∧ (∀  i< intlen s. z (nth s (Suc i)) = Suc(z (nth s i))) ∧
   z (nth s (intlen s)) = 4)
 proof −
  have 1: z (nth s 0) =0  using 0 by auto
  have 2: z (nth s (intlen s)) = 4 using 0 by auto
  have 3: (∀  i< intlen s. z (nth s (Suc i)) = Suc(z (nth s i))) by (simp add: 0)
  from 1 2 3 show ?thesis by auto
 qed
qed

lemma (in Test) test5-5:
  (z (nth s 0) =0  ∧ (∀  i< intlen s. z (nth s (Suc i)) = Suc(z (nth s i))) ∧
   z (nth s (intlen s)) = 4)
  =
 (z (nth s 0) =0  ∧ (∀  i≤ intlen s. z (nth s i) = i)
  ∧ z (nth s (intlen s)) = 4)

using   test5-3 test5-4 by blast

lemma (in Test) test5-6 :
 (z (nth s 0) =0  ∧ (∀  i≤ intlen s. z (nth s i) = i)
  ∧ z (nth s (intlen s)) = 4) =
 (intlen s =4 ∧ (∀  i≤ intlen s. z (nth s i) = i) )

by auto

lemma (in Test) test5-7 :
 (s ⊨ ($z=#0 ∧ z gets $z+#1) ∧ fin($z=#4)) =
  (intlen s =4 ∧ (∀  i≤ intlen s. z (nth s i) = i) )
using test5-6 test5-5 test5-2 by fastforce

lemma (in Test) test5-8 :
 (s ⊨ ▷(($z=#0 ∧ z gets $z+#1) ∧ fin($z=#4))) =
  (
   ( (s ⊨ ($z=#0 ∧ z gets $z+#1) ∧ fin($z=#4)) ∧ intlen s =0) ∨
   ( 0 < intlen s  ∧ (s ⊨ $z=#0 ∧ z gets $z+#1 ∧ fin($z=#4)) ∧
     (∀  ia < intlen s. (prefix ia s ⊨ ¬(($z=#0 ∧ z gets $z+#1) ∧ fin($z=#4)))))
  )
```

```
using Fstsem[of TEMP ($z=#0 ∧ z gets $z+#1) ∧ fin($z=#4)]
by simp

lemma (in Test) test5-9 :
 ¬( (s ⊨ ($z=#0 ∧ z gets $z+#1) ∧ fin($z=#4)) ∧ intlen s =0)
using test5-7 by simp

lemma (in Test) test5-10:
   (s ⊨ ($z=#0 ∧ z gets $z+#1) ∧ fin($z=#4))
   ⟹
    0 < intlen s  ∧
   (∀  ia < intlen s. (prefix ia s ⊨ ¬(($z=#0 ∧ z gets $z+#1) ∧ fin($z=#4))))

proof −
 assume 0: s ⊨ ($z=#0 ∧ z gets $z+#1) ∧ fin($z=#4)
 show 0 < intlen s  ∧
       (∀  ia < intlen s. (prefix ia s ⊨ ¬(($z=#0 ∧ z gets $z+#1) ∧ fin($z=#4))))
 proof −
  have 1: 0 < intlen s using test5-7 0 by simp
  have 2: (∀  ia < intlen s. (prefix ia s ⊨ ¬(($z=#0 ∧ z gets $z+#1) ∧ fin($z=#4))))
  proof
   fix ia
   show ia < intlen s ⟶
       (prefix ia s ⊨ ¬ (($z = #0 ∧ z gets $z + #1) ∧ fin ($z = #4)))
   proof −
   have 1: (prefix ia s ⊨ ¬ (($z = #0 ∧ z gets $z + #1) ∧ fin ($z = #4))) =
            (¬((prefix ia s ⊨  (($z = #0 ∧ z gets $z + #1) ∧ fin ($z = #4)))))
             by auto
   have 2: (prefix ia s ⊨  (($z = #0 ∧ z gets $z + #1) ∧ fin ($z = #4))) =
             (intlen (prefix ia s) = 4 ∧ (∀  i≤ intlen(prefix ia s) . z (nth (prefix ia s) i) = i))
             using test5-7 by simp
   have 3: ia < intlen s ⟶ ¬(intlen (prefix ia s) = 4 ∧
                              (∀  i≤ intlen(prefix ia s) . z (nth (prefix ia s) i) = i))
              using 0 using test5-7 by auto
   from 1 2 3 show ?thesis by blast
   qed
  qed
  from 1 2 show ?thesis by auto
 qed
qed

lemma (in Test) test5-11 :
  (s ⊨ ▷(($z=#0 ∧ z gets $z+#1) ∧ fin($z=#4))) =
  (s ⊨ ($z=#0 ∧ z gets $z+#1) ∧ fin($z=#4))
using test5-8 test5-9 test5-10 by fastforce

lemma (in Test) test5-12 :
 ⊢ ▷(($z=#0 ∧ z gets $z+#1) ∧ fin($z=#4)) = (($z=#0 ∧ z gets $z+#1) ∧ fin($z=#4))
using test5-11 by (simp add: Valid-def)
```

end

References

[1] A. Cau, B. Moszkowski, and D. Smallwood. The deep embedding of ITL in Isabelle/HOL, 2018. http://antonio-cau.co.uk/ITL/itlhomepagese6.html.

[2] G. Grov and S. Merz. A Definitional Encoding of TLA* in Isabelle/HOL. *Archive of Formal Proofs*, 2011. https://www.isa-afp.org/entries/TLA.html, Formal proof development.

[3] S. Merz. An Encoding of TLA in Isabelle. http://www.pst.informatik.uni-muenchen.de/~merz/isabelle/. Part of the Isabelle distribution., 1998.

[4] B. Moszkowski. A Hierarchical Completeness Proof for Propositional Interval Temporal Logic with Finite Time. *Journal of Applied Non-Classical Logics*, 14(1–2):55–104, 2004.

[5] B. C. Moszkowski. Imperative reasoning in interval temporal logic. Technical report, University of Newcastle upon Tyne, 1996.

[6] M. Wildmoser and T. Nipkow. Certifying Machine Code Safety: Shallow versus Deep Embedding. In K. Slind, A. Bunker, and G. Gopalakrishnan, editors, *Theorem Proving in Higher Order Logics (TPHOLs 2004)*, volume 3223 of *LNCS*, pages 305–320. Springer, 2004.