Abstract

We mechanise the logic TLA∗ [8], an extension of Lamport’s Temporal Logic of Actions (TLA) [5] for specifying and reasoning about concurrent and reactive systems. Aiming at a framework for mechanising the verification of TLA (or TLA∗) specifications, this contribution reuses some elements from a previous axiomatic encoding of TLA in Isabelle/HOL by the second author [7], which has been part of the Isabelle distribution. In contrast to that previous work, we give here a shallow, definitional embedding, with the following highlights:

- a theory of infinite sequences, including a formalisation of the concepts of stuttering invariance central to TLA and TLA∗;
- a definition of the semantics of TLA∗, which extends TLA by a mutually-recursive definition of formulas and pre-formulas, generalising TLA action formulas;
- a substantial set of derived proof rules, including the TLA∗ axioms and Lamport’s proof rules for system verification;
- a set of examples illustrating the usage of Isabelle/TLA∗ for reasoning about systems.

Note that this work is unrelated to the ongoing development of a proof system for the specification language TLA+, which includes an encoding of TLA+ as a new Isabelle object logic [1].

A previous version of this embedding has been used heavily in the work described in [4].

Contents

1 (Infinite) Sequences .................................................................................. 3
  1.1 Some operators on sequences .............................................................. 3
    1.1.1 Properties of first and second .................................................. 4
    1.1.2 Properties of (|) ........................................................................ 4
    1.1.3 Properties of (#) ................................................................. 5
  1.2 Finite and Empty Sequences ............................................................. 5
    1.2.1 Properties of emptyseq .......................................................... 6
6 Liveness 67
   6.1 Properties of -Enabled ........................................ 68
   6.2 Fairness Properties ............................................. 69
   6.3 Stuttering Invariance ............................................ 75

7 Representing state in TLA* 75
   7.1 Temporal Quantifiers ............................................. 78

8 A simple illustrative example 79

9 Lamport’s Inc example 80

10 Refining a Buffer Specification 93
   10.1 Buffer specification ............................................. 93
   10.2 Properties of the buffer ......................................... 94
   10.3 Two FIFO buffers in a row implement a buffer ............... 98

1 (Infinite) Sequences

theory Sequence
imports Main
begin

Lamport’s Temporal Logic of Actions (TLA) is a linear-time temporal logic, and its semantics is defined over infinite sequence of states, which we simply represent by the type 'a seq, defined as an abbreviation for the type nat ⇒ 'a, where 'a is the type of sequence elements.

This theory defines some useful notions about such sequences, and in particular concepts related to stuttering (finite repetitions of states), which are important for the semantics of TLA. We identify a finite sequence with an infinite sequence that ends in infinite stuttering. In this way, we avoid the complications of having to handle both finite and infinite sequences of states: see e.g. Devillers et al [2] who discuss several variants of representing possibly infinite sequences in HOL, Isabelle and PVS.

type-synonym 'a seq = nat ⇒ 'a

1.1 Some operators on sequences

Some general functions on sequences are provided

definition first :: 'a seq ⇒ 'a
where first s ≜ s 0

definition second :: ('a seq) ⇒ 'a
where second s ≜ s 1
definition **suffix** :: 'a seq ⇒ nat ⇒ 'a seq (infixl \( |_s \) 60)
where \( s |_s i \equiv \lambda n. s(n + i) \)

definition **tail** :: 'a seq ⇒ 'a seq
where **tail** \( s \equiv s |_s 1 \)

definition **app** :: 'a ⇒ ('a seq) ⇒ ('a seq) (infixl ## 60)
where \( s ## \sigma \equiv \lambda n. \text{if } n = 0 \text{ then } s \text{ else } \sigma(n - 1) \)

\( s |_s i \) returns the suffix of sequence \( s \) from index \( i \). **first** returns the first element of a sequence while **second** returns the second element. **tail** returns the sequence starting at the second element. \( s ## \sigma \) prefixes the sequence \( \sigma \) by element \( s \).

### 1.1.1 Properties of **first** and **second**

**lemma** **first-tail-second**: \( \text{first}(\text{tail } s) = \text{second } s \)
by \((\text{simp add: first-def second-def tail-def suffix-def})\)

### 1.1.2 Properties of \( |_s \)

**lemma** **suffix-first**: \( \text{first } (s |_s n) = s + n \)
by \((\text{auto simp add: suffix-def first-def})\)

**lemma** **suffix-second**: \( \text{second } (s |_s n) = s(Suc n) \)
by \((\text{auto simp add: suffix-def second-def})\)

**lemma** **suffix-plus**: \( s |_s n |_s m = s |_s (m + n) \)
by \((\text{simp add: suffix-def add.assoc})\)

**lemma** **suffix-commute**: \( ((s |_s n) |_s m) = ((s |_s m) |_s n) \)
by \((\text{simp add: suffix-plus add.commute})\)

**lemma** **suffix-plus-com**: \( s |_s m |_s n = s |_s (m + n) \)
**proof**
- **have** \( s |_s n |_s m = s |_s (m + n) \) **by (rule suffix-plus)**
- **thus** \( s |_s m |_s n = s |_s (m + n) \) **by (simp add: suffix-commute)**
**qed**

**lemma** **suffix-zero**: \( s |_s 0 = s \)
by \((\text{simp add: suffix-def})\)

**lemma** **suffix-tail**: \( s |_s 1 = \text{tail } s \)
by \((\text{simp add: tail-def})\)

**lemma** **tail-suffix-suc**: \( s |_s (Suc n) = \text{tail } (s |_s n) \)
by \((\text{simp add: suffix-def tail-def})\)
1.1.3 Properties of (##)

lemma seq-app-second: \( (s \#\# \sigma) 1 = \sigma 0 \)
  by (simp add: app-def)

lemma seq-app-first: \( (s \#\# \sigma) 0 = s \)
  by (simp add: app-def)

lemma seq-app-first-tail: \( (\text{first } s) \#\# (\text{tail } s) = s \)
proof (rule ext)
  fix x
  show \( (\text{first } s \#\# \text{tail } s) x = s x \)
    by (simp add: first-def app-def suffix-def tail-def)
qed

lemma seq-app-tail: \( \text{tail } (x \#\# s) = s \)
  by (simp add: app-def tail-def suffix-def)

lemma seq-app-greater-than-zero: \( n > 0 \implies (s \#\# \sigma) n = \sigma (n - 1) \)
  by (simp add: app-def)

1.2 Finite and Empty Sequences

We identify finite and empty sequences and prove lemmas about them.

definition fin :: ('a seq) \Rightarrow bool
  where fin s \equiv \exists i. \forall j \geq i. s j = s i

abbreviation inf :: ('a seq) \Rightarrow bool
  where inf s \equiv \neg (fin s)

definition last :: ('a seq) \Rightarrow nat
  where last s \equiv \text{LEAST } i. (\forall j \geq i. s j = s i)

definition laststate :: ('a seq) \Rightarrow 'a
  where laststate s \equiv s (\text{last } s)

definition emptyseq :: ('a seq) \Rightarrow bool
  where emptyseq \equiv \lambda s. \forall i. s i = s 0

abbreviation notemptyseq :: ('a seq) \Rightarrow bool
  where notemptyseq s \equiv \neg (emptyseq s)

Predicate fin holds if there is an element in the sequence such that all subsequent elements are identical, i.e. the sequence is finite. Sequence.last s returns the smallest index from which on all elements of a finite sequence s are identical. Note that if s is not finite then an arbitrary number is returned. laststate returns the last element of a finite sequence. We assume that the sequence is finite when using Sequence.last and laststate. Predicate emptyseq identifies empty sequences – i.e. all states in the sequence are
identical to the initial one, while notemptyseq holds if the given sequence is not empty.

1.2.1 Properties of emptyseq

**lemma** empty-is-finite: assumes emptyseq s shows fin s
  using assms by (auto simp: fin-def emptyseq-def)

**lemma** empty-suffix-is-empty: assumes H: emptyseq s shows emptyseq (s |s n)
  proof (clarsimp simp: emptyseq-def)
    fix i
    from H have (s |s n) i = s 0 by (simp add: emptyseq-def suffix-def)
    moreover
    from H have (s |s n) 0 = s 0 by (simp add: emptyseq-def suffix-def)
    ultimately show (s |s n) i = (s |s n) 0 by simp
  qed

**lemma** suc-empty: assumes H1: emptyseq (s |s m) shows emptyseq (s |s (Suc m))
  proof –
    from H1 have emptyseq ((s |s m) |s 1) by (rule empty-suffix-is-empty)
    thus ?thesis by (simp add: suffix-plus)
  qed

**lemma** empty-suffix-exteq: assumes H:emptyseq s shows (s |s n) m = s m
  proof (unfold suffix-def)
    from H have s (m+n) = s 0 by (simp add: emptyseq-def)
    moreover
    from H have s m = s 0 by (simp add: emptyseq-def)
    ultimately show s (m + n) = s m by simp
  qed

**lemma** empty-suffix-eq: assumes H: emptyseq s shows (s |s n) = s
  proof (rule ext)
    fix m
    from H show (s |s n) m = s m by (rule empty-suffix-exteq)
  qed

**lemma** seq-empty-all: assumes H: emptyseq s shows s i = s j
  proof –
    from H have s i = s 0 by (simp add: emptyseq-def)
    moreover
    from H have s j = s 0 by (simp add: emptyseq-def)
    ultimately
    show ?thesis by simp
  qed
1.2.2 Properties of Sequence.last and laststate

lemma fin-stut-after-last: assumes H: fin s shows \( \forall j \geq \text{last } s. \ s j = s (\text{last } s) \)
proof (clarify)
  fix j
  assume j: \( j \geq \text{last } s \)
  from H obtain i where \( \forall j \geq i. \ s j = s i \) (is ?P i) by (auto simp: fin-def)
  hence ?P (last s) unfolding last-def by (rule LeastI)
  with j show s j = s (last s) by blast
qed

1.3 Stuttering Invariance

This subsection provides functions for removing stuttering steps of sequences, i.e. we formalise Lamport’s \( \natural \) operator. Our formal definition is close to that of Wahab in the PVS prover.

The key novelty with the Sequence theory, is the treatment of stuttering invariance, which enables verification of stuttering invariance of the operators derived using it. Such proofs require comparing sequences up to stuttering. Here, Lamport’s [5] method is used to mechanise the equality of sequences up to stuttering: he defines the \( \natural \) operator, which collapses a sequence by removing all stuttering steps, except possibly infinite stuttering at the end of the sequence. These are left unchanged.

definition nonstutseq :: \( (\alpha \ \text{seq}) \rightarrow \text{bool} \)
where nonstutseq s \( \equiv \forall i. s i = s \ (\text{Suc } i) \rightarrow (\forall j > i. \ s i = s j) \)

definition stutstep :: \( (\alpha \ \text{seq}) \rightarrow \text{nat} \rightarrow \text{bool} \)
where stutstep s n \( \equiv (s n = s \ (\text{Suc } n)) \)

definition nextnat :: \( (\alpha \ \text{seq}) \rightarrow \text{nat} \)
where nextnat s \( \equiv \text{if emptyseq } s \text{ then } 0 \text{ else LEAST } i. \ s i \neq s 0 \)

definition nextsuffix :: \( (\alpha \ \text{seq}) \rightarrow (\alpha \ \text{seq}) \)
where nextsuffix s \( \equiv s \parallel s (\text{nextnat } s) \)

fun next :: \( \text{nat} \rightarrow (\alpha \ \text{seq}) \rightarrow (\alpha \ \text{seq}) \) where
next 0 = id
| next (Suc n) = nextsuffix o (next n)

definition collapse :: \( (\alpha \ \text{seq}) \rightarrow (\alpha \ \text{seq}) \) (\( \natural \))
where \( \natural s \equiv \lambda n. \ (\text{next } n \ s) \ 0 \)

Predicate nonstutseq identifies sequences without any stuttering steps – except possibly for infinite stuttering at the end. Further, stutstep s n is a predicate which holds if the element after s n is equal to s n, i.e. Suc n is a stuttering step. \( \natural s \) formalises Lamport’s \( \natural \) operator. It returns the first state of the result of next n s. next n s finds suffix of the n-th change. Hence
the first element, which \( \hat{s} \) returns, is the state after the \( n^{th} \) change. \( \text{next} \ n \ s \) is defined by primitive recursion on \( n \) using function composition of function \( \text{nextsuffix} \). E.g. \( \text{next} \ 3 \ s \) equals \( \text{nextsuffix} (\text{nextsuffix} (\text{nextsuffix} s)) \). \( \text{nextsuffix} \ s \) returns the suffix of the sequence starting at the next changing state. It uses \( \text{nextnat} \) to obtain this. All the real computation is done in this function. Firstly, an empty sequence will obviously not contain any changes, and \( \emptyset \) is therefore returned. In this case \( \text{nextsuffix} \) behaves like the identify function. If the sequence is not empty then the smallest number \( i \) such that \( s \ i \) is different from the initial state is returned. This is achieved by \( \text{Least} \).

1.3.1 Properties of \( \text{nonstutseq} \)

**Lemma seq-empty-is-nonstut**

- Assumes \( H: \text{emptyseq} \ s \) shows \( \text{nonstutseq} \ s \)
  - Using \( H \) by \( \text{auto simp: nonstutseq-def seq-empty-all} \)

**Lemma notempty-exist-nonstut**

- Assumes \( H: \neg \text{emptyseq} (s \mid_s \ m) \) shows \( \exists \ i. \ s \ i \neq s \ m \land i > m \)
  - Using \( H \) proof \( \text{auto simp: emptyseq-def suffix-def} \)
    - Fix \( i \)
    - Assume \( i: s (i + m) \neq s \ m \)
    - Hence \( i \neq 0 \) by \( \text{intro notI, simp} \)
    - With \( i \) show \( \text{thesis by auto} \)

**Lemma stutnempty**

- Assumes \( H: \neg \text{stutstep} s \ n \) shows \( \neg \text{emptyseq} (s \mid_s s \ n) \)
  - Using \( H \) by \( \text{auto simp: emptyseq-def suffix-def} \)
    - Rule \( \text{ccontr} \)
      - Assume \( a2: s \ n \neq s \ 0 \) \( \text{(is ?P n)} \)
      - Hence \( \text{LEAST i. s i \neq s 0} \leq n \) by \( \text{rule Least-le} \)
      - Hence \( \neg(n < (\text{LEAST i. s i \neq s 0})) \) by \( \text{auto} \)
      - Also from \( H \) \( a1 \) have \( n < (\text{LEAST i. s i \neq s 0}) \) by \( \text{simp} \)
      - Ultimately show \( \text{False by auto} \)

1.3.2 Properties of \( \text{nextnat} \)

**Lemma nextnat-le-unch**

- Assumes \( H: n < \text{nextnat} s \) shows \( s \ n = s \ 0 \)
  - Using \( H \) by \( \text{cases emptyseq s} \)
    - Assume \( \text{emptyseq s} \)
      - Hence \( \text{nextnat s} = 0 \) by \( \text{simp add: nextnat-def} \)
      - With \( H \) show \( \text{thesis by auto} \)
  - Next
    - Assume \( \neg \text{emptyseq s} \)
      - Hence \( a1: \text{nextnat s} = (\text{LEAST i. s i \neq s 0}) \) by \( \text{simp add: nextnat-def} \)
      - Show \( \text{thesis} \)
      - Proof \( \text{(rule ccontr)} \)
        - Assume \( a2: s \ n \neq s \ 0 \) \( \text{(is ?P n)} \)
          - Hence \( (\text{LEAST i. s i \neq s 0}) \leq n \) by \( \text{rule Least-le} \)
          - Hence \( \neg(n < (\text{LEAST i. s i \neq s 0})) \) by \( \text{auto} \)
          - Also from \( H \) \( a1 \) have \( n < (\text{LEAST i. s i \neq s 0}) \) by \( \text{simp} \)
          - Ultimately show \( \text{False by auto} \)
    - Qed

**Lemma stutnempty**

- Assumes \( H: \neg \text{stutstep} s \ n \) shows \( \neg \text{emptyseq} (s \mid_s n) \)
  - Using \( H \) by \( \text{unfold emptyseq-def suffix-def} \)

8
from $H$ have $s \ (\text{Suc} \ n) \neq s \ n$ by (auto simp add: stutstep-def)

hence $s \ (1+n) \neq s \ (0+n)$ by simp

thus $\neg \ (\forall \ i. \ s \ (i+n) = s \ (0+n))$ by blast

qed

lemma notstutstep-nexnat1:

assumes $H$: $\neg$ stutstep $s \ n$

shows $\text{nextrat} \ (s \ |_s \ n) = 1$

proof -

from $H$ have $h'$: $\text{nextrat} \ (s \ |_s \ n) = (\text{LEAST} \ i. \ (s \ |_s \ n) \ i \neq (s \ |_s \ n) \ 0)$

by (auto simp add: nextrat-def stutnempty)

from $H$ have $s \ (\text{Suc} \ n) \neq s \ n$ by (auto simp add: stutstep-def)

hence $(s \ |_s \ n) \ 1 \neq (s \ |_s \ n) \ 0$ (is ?P 1) by (auto simp add: suffix-def)

hence $\text{Least} \ ?P \leq 1$ by (rule Least-le)

hence $g1$: $\text{Least} \ ?P = 0 \lor \text{Least} \ ?P = 1$ by auto

with $h'$ have $g1'$: $\text{nextrat} \ (s \ |_s \ n) = 0 \lor \text{nextrat} \ (s \ |_s \ n) = 1$ by auto

also have $\text{nextrat} \ (s \ |_s \ n) \neq 0$

proof -

from $H$ have $\neg$ emptyseq $\ (s \ |_s \ n)$ by (rule stutnempty)

then obtain $i$ where $(s \ |_s \ n) \ i \neq (s \ |_s \ n) \ 0$ by (auto simp add: emptyseq-def)

hence $(s \ |_s \ n) \ (\text{LEAST} \ i. \ (s \ |_s \ n) \ i \neq (s \ |_s \ n) \ 0) \neq (s \ |_s \ n) \ 0$ by (rule Least1)

with $h'$ have $g2$: $(s \ |_s \ n) \ (\text{nextrat} \ (s \ |_s \ n)) \neq (s \ |_s \ n) \ 0$ by auto

show $(\text{nextrat} \ (s \ |_s \ n)) \neq 0$

proof

assume $(\text{nextrat} \ (s \ |_s \ n)) = 0$

with $g2$ show False by simp

qed

ultimately show $\text{nextrat} \ (s \ |_s \ n) = 1$ by auto

qed

lemma stutstep-notempty-notempty:

assumes $h1$: emptyseq $(s \ |_s \ \text{Suc} \ n)$ (is emptyseq ?sn)

and $h2$: stutstep $s \ n$

shows emptyseq $(s \ |_s \ n)$ (is emptyseq ?s)

proof (auto simp: emptyseq-def)

fix $k$

show $?s \ k = ?s \ 0$

proof (cases $k$)

  assume $k = 0$ thus ?thesis by simp

next

  fix $m$

  assume $k: \ k = \text{Suc} \ m$

  hence $?s \ k = ?s \ m$ by (simp add: suffix-def)

  also from $h1$ have $\ldots$ $= ?s \ 0$ by (simp add: emptyseq-def)

  also from $h2$ have $\ldots = s \ n$ by (simp add: suffix-def stutstep-def)

  finally show ?thesis by (simp add: suffix-def)

qed

qed
lemma `stutstep-empty-suc`:
  assumes `stutstep` `s` `n`
  shows `emptyseq` `s` `Suc` `n` = `emptyseq` `s` `n`
using assms by (auto elim: `stutstep-notempty-notempty` `Suc-empty`)

lemma `stutstep-notempty-sucnextnat`:
  assumes `h1`: `¬` `emptyseq` `s` `Suc` `n` and `h2`: `stutstep` `s` `n`
  shows `nextnat` `s` `n` = `Suc` `(nextnat` `s` `Suc` `n`))
proof –
  from `h2` have `g1`: `¬` `(s` `(Suc` `n)` `≠` `s` `Suc` `n`) by (auto simp add: `stutstep-def`)
  from `h1` obtain `i` where `s` `(i` `+` `n)` `≠` `s` `n` by (auto simp: `emptyseq` `suffix-def`)
  with `h2` have `g2`: `s` `(i` `+` `n)` `≠` `s` `Suc` `n` by (simp add: `stutstep-empty-suc`)
  from `g2` `g1` have `(LEAST` `n` `?P` `n` `=` `Suc` (LEAST` `n` ?P` `Suc` `n`)) by (rule Least-Suc)
  from `g2` `g1` have `(LEAST` `i` `s` `(i` `+` `n)` `≠` `s` `Suc` `n`) = `Suc` (LEAST` `i` `s` `(Suc` `i` `+` `n)` `≠` `s` `Suc` `n`))
    by (rule Least-Suc)
  hence `G1`: `(LEAST` `i` `s` `(i` `+` `n)` `≠` `s` `Suc` `n`) = `Suc` (LEAST` `i` `s` `(i` `+` `Suc` `n)` `≠` `s` `Suc` `n`)) by auto
  from `h1` `h2` have `¬` `emptyseq` `s` `Suc` `n` by (simp add: `stutstep-empty-suc`)
  hence `nextnat` `s` `Suc` `n` = (LEAST` `i` `s` `(Suc` `n`) `i` `≠` `s` `Suc` `n` `0`)
    by (auto simp add: `nextnat-def`)
  hence `g1`: `nextnat` `s` `Suc` `n` = (LEAST` `i` `s` `(Suc` `n`) `i` `≠` `s` `Suc` `n` `0`)
    by (auto simp add: `nextnat-def`)
  from `h1` have `nextnat` `s` `Suc` `n` = (LEAST` `i` `s` `(Suc` `n`) `i` `≠` `s` `Suc` `n` `0`)
    by (auto simp add: `nextnat-def`)
  hence `g2`: `nextnat` `s` `Suc` `n` = (LEAST` `i` `s` `(Suc` `n`) `i` `≠` `s` `Suc` `n`)
    by (auto simp add: `suffix-def`)
  with `h2` have `g2`: `nextnat` `s` `Suc` `n` = (LEAST` `i` `s` `(Suc` `n`) `i` `≠` `s` `Suc` `n`)
    by (auto simp add: `stutstep-def`)
  from `G1` `g1` `g2` show `?thesis` by auto
qed

lemma `nextnat-empty-neq`: assumes `H`: `¬` `emptyseq` `s` shows `s` `Suc` `n` `≠` `s` `0`
proof –
  from `H` have `a1`: `nextnat` `s` `n` = (LEAST` `i` `s` `i` `≠` `s` `0`)
    by (simp add: `nextnat-def`)
  from `H` obtain `i` where `s` `i` `≠` `s` `0` by (auto simp: `emptyseq`)
  hence `s` (LEAST` `i` `s` `i` `≠` `s` `0`) `≠` `s` `0` by (rule LeastI)
    with `a1` show `?thesis` by auto
qed

lemma `nextnat-empty-gzero`: assumes `H`: `¬` `emptyseq` `s` shows `nextnat` `s` > `0`
proof –
  from `H` have `a1`: `s` `Suc` `n` `≠` `s` `0` by (rule `nextnat-empty-neq`)
  have `nextnat` `s` `≠` `0` by (rule `nextnat-empty-neq`)
  assume `nextnat` `s` `=` `0`
    with `a1` show `False` by simp
qed
thus \text{nextnat} \; s > 0 \; \text{by simp}
qd

1.3.3 Properties of \text{nextsuffix}

\textbf{lemma empty-nextsuffix:}
\begin{enumerate}
\item assumes \(H: \text{emptyseq} \; s\) shows \(\text{nextsuffix} \; s = s\)
\item using \(H\) by (simp add: \text{nextsuffix-def nextnat-def})
\end{enumerate}

\textbf{lemma empty-nextsuffix-id:}
\begin{enumerate}
\item assumes \(H: \text{emptyseq} \; s\) shows \(\text{nextsuffix} \; s = \text{id} \; s\)
\item using \(H\) by (simp add: \text{empty-nextsuffix})
\end{enumerate}

\textbf{lemma notstutstep-nextsuffix1:}
\begin{enumerate}
\item assumes \(H: \neg \text{stutstep} \; s \; n\) shows \(\text{nextsuffix} \; (s \mid s \mid s \mid n) = s \mid s \mid s \mid n \mid (\text{Suc} \; n)\)
\item proof (unfold \text{nextsuffix-def})
\item show \((s \mid s \mid s \mid (\text{nextnat} \; (s \mid s \mid n))) = s \mid s \mid (\text{Suc} \; n)\)
\end{enumerate}

\textbf{lemma notempty-nextnotzero:}
\begin{enumerate}
\item assumes \(H: \neg \text{emptyseq} \; s\) shows \((\text{next} \; (\text{Suc} \; 0) \; s) \; 0 \neq s \; 0\)
\item proof –
\end{enumerate}
from H have g1: s (nextnat s) ≠ s 0 by (rule nextnat-empty-neq)
have next (Suc 0) s = nextsuffix s by auto
hence (next (Suc 0) s) 0 = s (nextnat s) by (simp add: nextsuffix-def suffix-def)
with g1 show ?thesis by simp
qed

lemma next-ex-id: ∃ i. s i = (next m s) 0
proof –
have ∃ i. (s |s i) = (next m s)
proof (induct m)
have s |s 0 = next 0 s by simp
thus ∃ i. (s |s i) = (next 0 s) ..
next
fix m
assume a1: ∃ i. (s |s i) = (next m s)
then obtain i where a1': (s |s i) = (next m s).. have next (Suc m) s = nextsuffix (next m s) by auto
hence next (Suc m) s = (next m s) |s (nextnat (next m s)) by (simp add: nextsuffix-def)
hence ∃ i. next (Suc m) s = (next m s) |s i (next nat (Suc m) s) by (simp add: nextsuffix-def)
hence ∃ i. next (Suc m) s = (next m s) |s i ..
then obtain j where next (Suc m) s = (next m s) |s j ..
with a1' have next (Suc m) s = (s |s i) |s j by simp
hence next (Suc m) s = (s |s (j+i)) by (simp add: suffix-plus)
hence (s |s (j+i)) = next (Suc m) s by simp
thus ∃ i. (s |s i) = (next (Suc m) s) ..
qed
then obtain i where (s |s i) = (next m s) ..
hence (s |s i) 0 = (next m s) 0 by auto
hence s i = (next m s) 0 by (auto simp add: suffix-def)
thus ?thesis ..
qed

1.3.5 Properties of ♯

lemma emptyseq-collapse-eq: assumes A1: emptyseq s shows ♯ s = s
proof (unfold collapse-def, rule ext)
fix n
from A1 have next n s = s by (rule next-empty)
moreover
from A1 have s n = s 0 by (simp add: emptyseq-def)
ultimately
show (next n s) 0 = s n by simp
qed

lemma empty-collapse-empty:
  assumes H: emptyseq s shows emptyseq (♯ s)
  using H by (simp add: emptyseq-collapse-eq)

lemma collapse-empty-empty:
assumes $H$: emptyseq ($\emptyset s$) shows emptyseq $s$
proof (rule ccontr)
  assume $a1$: $\neg$emptyseq $s$
  from $H$ have $\forall i. (\text{next } i \; s) \; 0 = s \; 0$ by (simp add: collapse-def emptyseq-def)
  moreover
  from $a1$ have $(\text{next } (\text{Suc } 0) \; s) \; 0 \neq s \; 0$ by (rule notempty-nextnotzero)
  ultimately show False by blast
qed

lemma collapse-empty-iff-empty [simp]: emptyseq ($\emptyset s$) = emptyseq $s$
  by (auto elim: empty-collapse-empty collapse-empty-empty)

1.4 Similarity of Sequences
Since adding or removing stuttering steps does not change the validity of
a stuttering-invariant formula, equality is often too strong, and the weaker
equality up to stuttering is sufficient. This is often called similarity ($\approx$) of
sequences in the literature, and is required to show that logical operators
are stuttering invariant. This is mechanised as:

definition seqsimilar :: (′a seq) ⇒ (′a seq) ⇒ bool (infixl $\approx$ 50)
where $\sigma \approx \tau \equiv (\narrow s) = (\narrow t)$

1.4.1 Properties of ($\approx$)

lemma seqsim-refl [iff]: $s \approx s$
  by (simp add: seqsimilar-def)

lemma seqsim-sym: assumes $H$: $s \approx t$ shows $t \approx s$
  using $H$ by (simp add: seqsimilar-def)

lemma seqeq-imp-sim: assumes $H$: $s = t$ shows $s \approx t$
  using $H$ by simp

lemma seqsim-trans [trans]: assumes $h1$: $s \approx t$ and $h2$: $t \approx z$ shows $s \approx z$
  using assms by (simp add: seqsimilar-def)

theorem sim-first: assumes $H$: $s \approx t$ shows first $s = \text{first } t$
proof −
  from $H$ have $(\narrow s) \; 0 = (\narrow t) \; 0$ by (simp add: seqsimilar-def)
  thus $\?thesis$ by (simp add: collapse-def first-def)
qed

lemmas sim-first2 = sim-first[unfolded first-def]

lemma tail-sim-second: assumes $H$: tail $s \approx \text{tail } t$ shows second $s = \text{second } t$
proof −
  from $H$ have first (tail $s$) = first (tail $t$) by (simp add: sim-first)
  thus second $s = \text{second } t$ by (simp add: first-tail-second)
lemma seqsimilarI:
assumes 1: first s = first t and 2: nextsuffix s ≈ nextsuffix t
shows s ≈ t
unfolding seqsimilar-def collapse-def
proof
fix n
show next n s 0 = next n t 0
proof (cases n)
assume n = 0
with 1 show ?thesis by (simp add: first-def)
next
fix m
assume m: n = Suc m
from 2 have next m (nextsuffix s) 0 = next m (nextsuffix t) 0
unfolding seqsimilar-def collapse-def by (rule fun-cong)
with m show ?thesis by (simp add: next-suffix-com)
qed
qed

lemma seqsim-empty-empty:
assumes H1: s ≈ t and H2: emptyseq s shows emptyseq t
proof −
from H2 have emptyseq (♮ s) by simp
with H1 have emptyseq (♮ t) by (simp add: seqsimilar-def)
thus ?thesis by simp
qed

lemma seqsim-empty-iff-empty:
assumes H: s ≈ t shows emptyseq s = emptyseq t
proof
assume emptyseq s with H show emptyseq t by (rule seqsim-empty-empty)
next
assume t: emptyseq t
from H have t ≈ s by (rule seqsim-sym)
from this t show emptyseq s by (rule seqsim-empty-empty)
qed

lemma seq-empty-eq:
assumes H1: s 0 = t 0 and H2: emptyseq s and H3: emptyseq t
shows s = t
proof (rule ext)
fix n
from assms have t n = s n by (auto simp: emptyseq-def)
thus s n = t n by simp
qed

lemma seqsim-notstatstep:
assumes $H: \neg \text{stutstep } s \ n$ shows $(s \ |_s (\text{Suc } n)) \approx \text{nextsuffix } (s \ |_s n)$
using $H$ by (simp add: notstutstep-nextsuffix1)

lemma stat-nexsuffix-suc:
assumes $H: \text{stutstep } s \ n$ shows $\text{nextsuffix } (s \ |_s n) = \text{nextsuffix } (s \ |_s (\text{Suc } n))$
proof (cases emptyseq $(s \ |_s n)$)
case True
hence $g1: \text{nextsuffix } (s \ |_s n) = (s \ |_s n)$ by (simp add: nextsuffix-def nextnat-def)
from True have $g2: \text{nextsuffix } (s \ |_s \text{Suc } n) = (s \ |_s \text{Suc } n)$
  by (simp add: suc-empty nextsuffix-def nextnat-def)
have $(s \ |_s n) = (s \ |_s \text{Suc } n)$
proof
  fix $x$
from True have $(s \ |_s n) = (s \ |_s n)$
  unfolding emptyseq-def suffix-def
  by (blast+)
thus $(s \ |_s n) x = (s \ |_s \text{Suc } n) x$ by (simp add: suffix-def)
qed
with $g1$ $g2$ show $\text{thesis}$ by auto
next
case False
with $H$ have $(\text{nextnat } (s \ |_s n)) = \text{Suc } (\text{nextnat } (s \ |_s \text{Suc } n))$
  by (simp add: statstep-notempty-sucrextnat)
thus $\text{thesis}$
  by (simp add: nextsuffix-def suffix-plus)
qed

lemma seqsim-suffix-seqsim:
assumes $H: s \approx t$ shows $\text{nextsuffix } s \approx \text{nextsuffix } t$
unfolding seqsimilar-def collapse-def
proof
  fix $n$
from $H$ have $(\text{next } (\text{Suc } n) \ s) \ 0 = (\text{next } (\text{Suc } n) \ t) \ 0$
  unfolding seqsimilar-def collapse-def by (rule fun-cong)
thus $\text{next } n ((\text{nextsuffix } s) \ 0 = (\text{next } n (\text{nextsuffix } t) \ 0$
  by (simp add: next-suffix-com)
qed

lemma seqsim-statstep:
assumes $H: \text{stutstep } s \ n$ shows $(s \ |_s (\text{Suc } n)) \approx (s \ |_s n)$ (is $\text{?sn} \approx \text{?s}$)
unfolding seqsimilar-def collapse-def
proof
  fix $m$
  show $\text{next } m ((s \ |_s \text{Suc } n) \ 0 = (\text{next } m (s \ |_s n) \ 0$
  proof (cases $m$)
    assume $m=0$
    with $H$ show $\text{thesis}$ by (simp add: suffix-def stubstep-def)
next
  fix $k$
  assume $m: m = \text{Suc } k$
with \( H \) have next \( m \ (s \ |_s \ Suc \ n) = next \ k \ (nextsuffix \ (s \ |_s \ n)) \)
\[ by \ (simp \ add: \ stut-next-suf \ next-suffix-com) \]
moreover from \( m \) have next \( m \ (s \ |_s \ n) = next \ k \ (nextsuffix \ (s \ |_s \ n)) \)
\[ by \ (simp \ add: \ next-suffix-com) \]
ultimately show next \( m \ (s \ |_s \ Suc \ n) \ \theta = next \ m \ (s \ |_s \ n) \ \theta \) by simp
qed

lemma addfeqstut: stutstep \((\text{first} \ t) \text{##} \ t \) \(\theta\)
\[ by \ (simp \ add: \ first-def \ stutstep-def \ app-def \ suffix-def) \]

lemma addfeqsim: \((\text{first} \ t) \text{##} \ t \) \(\approx \ t\)
proof –
  have stutstep \((\text{first} \ t) \text{##} \ t \) \(\theta\) by \(\text{rule} \ addfeqstat\)
  hence \((\text{first} \ t) \text{##} \ t \) \(\approx \ ((\text{first} \ t) \text{##} \ t) \ |_s \ 0\) by \(\text{rule} \ seqsim-stutstep\)
  hence tail \((\text{first} \ t) \text{##} \ t \) \(\approx \ ((\text{first} \ t) \text{##} \ t) \ |_s \ 0\) by \(\text{simp} \ add: \ suffix-def \ tail-def\)
  hence \((\text{first} \ t) \text{##} \ t \) \(\approx \ t\) by \(\text{simp} \ add: \ tail-def \ app-def \ suffix-def\)
  thus \(\theta\)thesis by \(\text{rule} \ seqsim-sym\)
qed

lemma addfirststat:
assumes \(H\): first \(s = \text{second} \ s\) shows \(s \approx \text{tail} \ s\)
proof –
  have \((\text{first} \ s) \text{##} \ (\text{tail} \ s) = s\) by \(\text{rule} \ seq-app-first-tail\)
  from \(H\) have \((\text{first} \ s) = \text{first} \ (\text{tail} \ s)\)
\[ by \ (simp \ add: \ first-def \ second-def \ tail-def \ suffix-def) \]
  hence \((\text{first} \ s) \text{##} \ (\text{tail} \ s) \approx (\text{tail} \ s)\) by \(\text{simp} \ add: \ addfeqsim\)
  with \((\text{first} \ s) \text{##} \ (\text{tail} \ s) \approx (\text{tail} \ s)\) by \(\text{simp} \ add: \ addfeqsim\)
qed

lemma app-seqsimilar:
assumes \(h1\): \(s \approx t\) shows \(x \text{##} s \approx (x \text{##} t)\)
proof (cases stutstep \((x \text{##} s) \ |_s \ 0\))
  case True
  from \(h1\) have first \(s = \text{first} \ t\) by \(\text{rule} \ sim-first\)
  with True have \(a2\): stutstep \((x \text{##} t) \ |_s \ 0\)
\[ by \ (simp \ add: \ stutstep-def \ first-def \ app-def) \]
  from True have \((x \text{##} s) \ |_s \ 0\) \(\approx (x \text{##} t) \ |_s \ 0\) by \(\text{rule} \ seqsim-stutstep\)
  hence tail \((x \text{##} s) \approx (x \text{##} t)\) by \(\text{simp} \ add: \ tail-def \ suffix-def\)
  hence \(s \approx (x \text{##} s)\) by \(\text{simp} \ add: \ app-def \ tail-def \ suffix-def\)
  from \(a2\) have \((x \text{##} t) \ |_s \ 0\) \(\approx (x \text{##} t) \ |_s \ 0\) by \(\text{rule} \ seqsim-stutstep\)
  hence tail \((x \text{##} t) \approx (x \text{##} t)\) by \(\text{simp} \ add: \ tail-def \ suffix-def\)
  hence \(g2\): \(t \approx (x \text{##} t)\) by \(\text{simp} \ add: \ app-def \ tail-def \ suffix-def\)
  from \(h1\) \(g2\) have \(s \approx (x \text{##} t)\) by \(\text{rule} \ seqsim-trans\)
  from this [\(\text{THEN} \ seqsim-sym\] \(g1\) show \((x \text{##} s) \approx (x \text{##} t)\)
\[ by \ (rule \ seqsim-sym[of \ seqsim-trans]) \]
next
  case False
  from \(h1\) have first \(s = \text{first} \ t\) by \(\text{rule} \ sim-first\)
with \textbf{False} have \( a2 : \neg \text{stattest}\ (x \# \# t) \ 0 \)
by (simp add: stattest-def first-def app-def)

from \textbf{False} have \((x \# \# s) \mid_s (\text{Suc} \ 0)) \sim nextsuffix\ ((x \# \# s) \mid_s 0)\)
by (rule seqsim-notstutstep)

hence \((\text{tail}\ (x \# \# s)) \sim nextsuffix\ (x \# \# s)\)
by (simp add: tail-def)

hence \(g1 : s \approx nextsuffix\ (x \# \# s)\) by (simp add: seq-app-tail)
from \textbf{a2} have \((x \# \# t) \mid_s (\text{Suc} \ 0)) \sim nextsuffix\ ((x \# \# t) \mid_s 0)\)
by (rule seqsim-notstutstep)

hence \((\text{tail}\ (x \# \# t)) \approx nextsuffix\ (x \# \# t)\) by (simp add: tail-def)

hence \(g2 : t \approx nextsuffix\ (x \# \# t)\) by (simp add: seq-app-tail)

with \(h1 \) have \(s \approx nextsuffix\ (x \# \# t)\) by (rule seqsim-trans)
from \textbf{this[THEN seqsim-sym]} \(g1 \) have \(g3 : nextsuffix\ (x \# \# s) \approx nextsuffix\ (x \# \# t)\)
by (rule seqsim-sym[OF seqsim-trans])

have \(\text{first}\ (x \# \# s) = \text{first}\ (x \# \# t)\) by (simp add: first-def app-def)
from this \textbf{g3} show \(?thesis\) by (rule seqsimilarI)

qed

If two sequences are similar then for any suffix of one of them there exists a similar suffix of the other one. We will prove a stronger result below.

\textbf{lemma simstep-disj1: assumes} \(H: s \approx t\) \textbf{shows} \(\exists\ m.\ ((s \mid_s n) \approx (t \mid_s m))\)

\textbf{proof (induct \(n\))}
from \(H \) have \((s \mid_s 0) \approx (t \mid_s 0))\) by auto
thus \(\exists\ m.\ ((s \mid_s 0) \approx (t \mid_s m))\)

next

fix \(n\)

assume \(\exists\ m.\ ((s \mid_s n) \approx (t \mid_s m))\)
then obtain \(m\) where \(a1' : (s \mid_s n) \approx (t \mid_s m)\)

show \(\exists\ m.\ ((s \mid_s (\text{Suc} \ n)) \approx (t \mid_s m))\)

\textbf{proof (cases stattest \(s\ \mid\ n\))}

\textbf{case True}

hence \((s \mid_s (\text{Suc} \ n)) \approx (s \mid_s n)\) by (rule seqsim-stutstep)
from \textbf{this a1'} have \((s \mid_s (\text{Suc} \ n)) \approx (t \mid_s m)\) by (rule seqsim-trans)
thus \(?thesis\)

next

\textbf{case False}

hence \((s \mid_s (\text{Suc} \ n)) \approx nextsuffix\ (s \mid_s n)\) by (rule seqsim-notstutstep)

moreover

from \textbf{a1'} have \(nextsuffix\ (s \mid_s n) \approx nextsuffix\ (t \mid_s m)\)
by (simp add: seqsim-suffix-seqsim)

ultimately have \((s \mid_s (\text{Suc} \ n)) \approx nextsuffix\ (t \mid_s m)\) by (rule seqsim-trans)

hence \((s \mid_s (\text{Suc} \ n)) \approx t \mid_s (m + (\text{nextnat}\ (t \mid_s m)))\)
by (simp add: nextsuffix-def suffix-plus-com)
thus \(\exists\ m.\ (s \mid_s (\text{Suc} \ n)) \approx t \mid_s m\)

qed

qed

\textbf{lemma nextnat-le-seqsim}:
assumes $n$: $n < \text{nextnat } s$ shows $s \approx (s |_s n)$
proof (cases emptyseq $s$)
case True — case impossible
with $n$ show $\theta$thesis by (simp add: nextnat-def)
next
case False
from $n$ show $\theta$thesis
proof (induct $n$)
show $s \approx (s |_s 0)$ by simp
next
fix $n$
assume $a2$: $n < \text{nextnat } s$ and $a3$: $\text{Suc } n < \text{nextnat } s$
from $a3$ have $g1$: $s (\text{Suc } n) = s 0$ by (rule nextnat-le-unch)
from $a3$ have $a3'$: $n < \text{nextnat } s$ by simp
hence $s n = s 0$ by (rule nextnat-le-unch)
with $g1$ have $g2$: $(s |_s n) = (s |_s (\text{Suc } n))$ by (rule seqsim-stutstep[THEN seqsim-sym])
with $a3'$ $a2$ show $s \approx (s |_s (\text{Suc } n))$ by (auto elim: seqsim-trans)
qed qed

lemma seqsim-prev-nextnat: $s \approx s |_s ((\text{nextnat } s) - 1)$
proof (cases emptyseq $s$)
case True
hence $s \approx s |_s 0$ by (simp add: nextnat-def)
thus $\theta$thesis by simp
next
case False
hence $\neg \text{emptyseq } (s |_s n)$ by (rule stutnempty)
with $H$ have $a2$: $\neg \text{emptyseq } (t |_s m)$ by (simp add: seqsim-empty-iff-empty)
qed

Given a suffix $s |_s n$ of some sequence $s$ that is similar to some suffix $t |_s m$ of sequence $t$, there exists some suffix $t |_s m'$ of $t$ such that $s |_s n$ and $t |_s m'$ are similar and also $s |_s (n+1)$ is similar to either $t |_s m'$ or to $t |_s (m'+1)$.

lemma seqsim-suffix-suc:
assumes $H$: $s |_s n \approx t |_s m$
shows $\exists m'. s |_s n \approx t |_s m' \land ((s |_s \text{Suc } n \approx t |_s \text{Suc } m') \lor (s |_s \text{Suc } n \approx t |_s m'))$
proof (cases stutstep $s \ n$)
case True
hence $s |_s \text{Suc } n \approx s |_s n$ by (rule seqsim-stutstep)
from this $H$ have $s |_s \text{Suc } n \approx t |_s m$ by (rule seqsim-trans)
with $H$ show $\theta$thesis by blast
next
case False
hence $\neg \text{emptyseq } (s |_s n)$ by (rule stutnempty)
with $H$ have $a2$: $\neg \text{emptyseq } (t |_s m)$ by (simp add: seqsim-empty-iff-empty)

qed
hence \( g_4 \): \( \text{nextsuffix} \ (t \ |_s \ m) = (t \ |_s \ m) \ |_s \ (\text{nextnat} \ (t \ |_s \ m) - 1) \)

by \((\text{simp add: nextnat-empty-gzero nextsuffix-def})\)

have \( g_3 \): \( (t \ |_s \ m) \approx (t \ |_s \ m) \ |_s \ (\text{nextnat} \ (t \ |_s \ m) - 1) \)

by \((\text{rule seqsim-prev-nextnat})\)

with \( H \) have \( G_1 \): \( s \ |_s \ n \approx (t \ |_s \ m) \ |_s \ (\text{nextnat} \ (t \ |_s \ m) - 1) \)

by \((\text{rule seqsim-trans})\)

from False have \( G_1' \): \( s \ |_s \ Suc n \approx (t \ |_s \ m) \ |_s \ (\text{nextnat} \ (t \ |_s \ m) - 1) \)

by \((\text{rule notstutstep-nextsuffix1 [THEN sym]})\)

from \( H \) have \( \text{nextsuffix} \ (s \ |_s \ n) \approx \text{nextsuffix} \ (t \ |_s \ m) \)

by \((\text{rule seqsim-suffix-seqsim})\)

with \( G_1 \ G_1' \) \( g_4 \)

have \( s \ |_s \ n \approx t \ |_s \ (m + (\text{nextnat} \ (t \ |_s \ m) - 1)) \)

\& \( s \ |_s \ (Suc n) \approx t \ |_s \ (Suc m + (\text{nextnat} \ (t \ |_s \ m) - 1)) \)

by \((\text{simp add: suffix-plus-com})\)

thus \?thesis by blast

qed

The following main result about similar sequences shows that if \( s \approx t \) holds
then for any suffix \( s \ |_s \ n \) of \( s \) there exists a suffix \( t \ |_s \ m \) such that

- \( s \ |_s \ n \) and \( t \ |_s \ m \) are similar, and
- \( s \ |_s \ (n+1) \) is similar to either \( t \ |_s \ (m+1) \) or \( t \ |_s \ m \).

The idea is to pick the largest \( m \) such that \( s \ |_s \ n \approx t \ |_s \ m \) (or some such \( m \)
if \( s \ |_s \ n \) is empty).

theorem sim-step:

assumes \( H \): \( s \approx t \)

shows \( \exists \ m. \ s \ |_s \ n \approx t \ |_s \ m \ \& (s \ |_s \ Suc n \approx t \ |_s \ Suc m) \vee (s \ |_s \ Suc n \approx t \ |_s \ m) \)\)

(is \( \exists \ m. \ ?Sim n m \))

proof \((\text{induct } n)\)

from \( H \) have \( s \ |_s \ 0 \approx t \ |_s \ 0 \) by simp

thus \( \exists \ m. \ ?Sim 0 m \) by \((\text{rule seqsim-suffix-suc})\)

next

fix \( n \)

assume \( \exists \ m. \ ?Sim n m \)

hence \( \exists k. \ s \ |_s \ Suc n \approx t \ |_s \ k \) by blast

thus \( \exists \ m. \ ?Sim \ (Suc n) m \) by \((\text{blast dest: seqsim-suffix-suc})\)

qed

end

2 Representing Intensional Logic

theory Intensional

imports Main

begin
In higher-order logic, every proof rule has a corresponding tautology, i.e. the deduction theorem holds. Isabelle/HOL implements this since object-level implication ($\rightarrow$) and meta-level entailment ($\Rightarrow$) commute, viz. the proof rule $\text{impI}: (\forall P \Rightarrow \forall Q) \Rightarrow \forall P \rightarrow \forall Q$. However, the deduction theorem does not hold for most modal and temporal logics [6, page 95][7]. For example $A \vdash \Box A$ holds, meaning that if $A$ holds in any world, then it always holds. However, $\vdash A \rightarrow \Box A$, stating that $A$ always holds if it initially holds, is not valid.

Merz [7] overcame this problem by creating an Intensional logic. It exploits Isabelle’s axiomatic type class feature [9] by creating a type class world, which provides Skolem constants to associate formulas with the world they hold in. The class is trivial, not requiring any axioms.

2.1 Abstract Syntax and Definitions

```
type-synonym ('w, 'a) expr = 'w ⇒ 'a

type-synonym 'w form = ('w, bool) expr
```

The intention is that 'a will be used for unlifted types (class type), while 'w is lifted (class world).

```
definition Valid :: ('w::world) form ⇒ bool
  where Valid A ≡ ∀ w. A w
```

```
definition const :: 'a ⇒ ('w::world, 'a) expr
  where unl-con: const c w ≡ c
```

```
definition lift :: ['a ⇒ 'b, ('w::world, 'a) expr] ⇒ ('w,'b) expr
  where unl-lift: lift f x w ≡ f (x w)
```

```
definition lift2 :: ['a ⇒ 'b ⇒ 'c, ('w::world,'a) expr, ('w,'b) expr] ⇒ ('w,'c) expr
  where unl-lift2: lift2 f x y w ≡ f (x w) (y w)
```

```
definition lift3 :: ['a ⇒ 'b ⇒ 'c ⇒ 'd, ('w::world,'a) expr, ('w,'b) expr, ('w,'c) expr] ⇒ ('w,'d) expr
  where unl-lift3: lift3 f x y z w ≡ f (x w) (y w) (z w)
```

```
definition lift4 :: ['a ⇒ 'b ⇒ 'c ⇒ 'd ⇒ 'e, ('w::world,'a) expr, ('w,'b) expr, ('w,'c) expr, ('w,'d) expr] ⇒ ('w,'e) expr
  where unl-lift4: lift4 f x y z w z z w ≡ f (x w) (y w) (z w) (zz w)
```

Valid $F$ asserts that the lifted formula $F$ holds everywhere. const allows lifting of a constant, while lift through lift4 allow functions with arity 1–4
to be lifted. (Note that there is no way to define a generic lifting operator for functions of arbitrary arity.)

**definition** $\text{RAll} :: (\tau \Rightarrow (\text{')w::world}) \Rightarrow \text{'w form}$ (\text{\textbf{binder}} \text{Rall} 10)

\text{where} \ \text{unl-Rall} : (\text{Rall} \ x. \ A \ x) \ w \equiv \forall x. \ A \ x \ w

**definition** $\text{REx} :: (\tau \Rightarrow (\text{')w::world}) \Rightarrow \text{'w form}$ (\text{\textbf{binder}} \text{Rex} 10)

\text{where} \ \text{unl-REx} : (\text{REx} \ x. \ A \ x) \ w \equiv \exists x. \ A \ x \ w

**definition** $\text{REx1} :: (\tau \Rightarrow (\text{')w::world}) \Rightarrow \text{'w form}$ (\text{\textbf{binder}} \text{REx1} 10)

\text{where} \ \text{unl-REx1} : (\text{REx1} \ x. \ A \ x) \ w \equiv \exists! x. \ A \ x \ w

$\text{RAll}$, $\text{REx}$ and $\text{REx1}$ introduces “rigid” quantification over values (of non-world types) within “intensional” formulas. $\text{RAll}$ is universal quantification, $\text{REx}$ is existential quantification. $\text{REx1}$ requires unique existence.

We declare the “unlifting rules” as rewrite rules that will be applied automatically.

**lemmas** intensional-rews\[\text{simp} =

\text{unl-con unl-lift unl-lift2 unl-lift3 unl-lift4}

\text{unl-Rall unl-REx unl-REx1}

### 2.2 Concrete Syntax

**nonterminal**

\text{-lift} \ \text{and} \ \text{liftargs}

The non-terminal \text{lift} represents lifted expressions. The idea is to use Isabelle’s macro mechanism to convert between the concrete and abstract syntax.

**syntax**

\begin{align*}
\text{:: id} & \Rightarrow \text{lift} \quad (-) \\
\text{:: longid} & \Rightarrow \text{lift} \quad (-) \\
\text{:: var} & \Rightarrow \text{lift} \quad (-) \\
\text{-applC} & \Rightarrow [\text{lift}, \text{cargs}] \Rightarrow \text{lift} \quad \langle 1/- \rangle [1000, 1000] 999 \\
\text{-lambda} & \Rightarrow [\text{lift} \Rightarrow \text{lift}] \quad \langle'-'\rangle \\
\text{-constrain} & \Rightarrow [\text{lift}, \text{type}] \Rightarrow \text{lift} \quad \langle::-\rangle [4, 0] 3 \\
\text{-liftargs} & \Rightarrow [\text{lift} \Rightarrow \text{liftargs}] \quad (-) \\
\text{-Valid} & \Rightarrow [\text{lift} \Rightarrow \text{bool}] \quad \langle 0 - \rangle 5 \\
\text{-holdsAt} & \Rightarrow [\text{a}, \text{lift}] \Rightarrow \text{bool} \quad \langle [\text{=} \rangle [100,10] 10 \\
\end{align*}

$LIFT \Rightarrow \text{lift} \Rightarrow \text{'}a \quad LIFT$ \\
\begin{align*}
\text{-const} & \Rightarrow \text{'}a \Rightarrow \text{lift} \quad \langle (#-) [1000] 999 \\
\text{-lift} & \Rightarrow [\text{'a}, \text{lift}] \Rightarrow \text{lift} \quad \langle (-,-) [1000] 999 \\
\end{align*}
-lift2 :: ["a, lift, lift] ⇒ lift ((\<\-, \to\>) [1000] 999)
-lift3 :: ["a, lift, lift, lift] ⇒ lift ((\<\-, \to\>) [1000] 999)
-lift4 :: ["a, lift, lift, lift, lift] ⇒ lift ((\<\-, \to\-) [1000] 999)

-translations

-const ⇒ CONST const

-translations

-lift ⇒ CONST lift
-lift2 ⇒ CONST lift2
-lift3 ⇒ CONST lift3
-lift4 ⇒ CONST lift4
-Valid ⇒ CONST Valid

-translations

-∀ x A ⇒ Rall x. A
\[ -REx \ x \ A \quad \Rightarrow \ Rex \ x. \ A \]
\[ -REx1 \ x \ A \quad \Rightarrow \ Rex! \ x. \ A \]

**translations**

- \( -ARAll \rightarrow -RAll \)
- \( -AREx \rightarrow -REx \)
- \( -AREx1 \rightarrow -REx1 \)

\[ w \models A \rightarrow A \ w \]

\[ LIFT \ A \rightarrow A :: \Rightarrow \]

**translations**

- \( \text{liftEqu} \quad \Rightarrow \quad \text{lift} \ 2 \ (=) \)
- \( \text{liftNeq} \ u \ v \quad \Rightarrow \quad \text{liftNot} \ 2 \ (\text{liftEqu} \ u \ v) \)
- \( \text{liftNot} \quad \Rightarrow \quad \text{lift} \ 2 \ (\text{CONST} \ \text{Not}) \)
- \( \text{liftAnd} \quad \Rightarrow \quad \text{lift} \ 2 \ (&) \)
- \( \text{liftOr} \quad \Rightarrow \quad \text{lift} \ 2 \ (\text{(||)}) \)
- \( \text{liftImp} \quad \Rightarrow \quad \text{lift} \ 2 \ (\text{(-|->)}) \)
- \( \text{liftIf} \quad \Rightarrow \quad \text{lift} \ 3 \ (\text{CONST} \ \text{If}) \)
- \( \text{liftPlus} \quad \Rightarrow \quad \text{lift} \ 2 \ (+) \)
- \( \text{liftMinus} \quad \Rightarrow \quad \text{lift} \ 2 \ (-) \)
- \( \text{liftTimes} \quad \Rightarrow \quad \text{lift} \ 2 \ ((\ast)) \)
- \( \text{liftDiv} \quad \Rightarrow \quad \text{lift} \ 2 \ (\text{div}) \)
- \( \text{liftMod} \quad \Rightarrow \quad \text{lift} \ 2 \ (\text{mod}) \)
- \( \text{liftLess} \quad \Rightarrow \quad \text{lift} \ 2 \ (<) \)
- \( \text{liftLeq} \quad \Rightarrow \quad \text{lift} \ 2 \ (\leq) \)
- \( \text{liftMem} \quad \Rightarrow \quad \text{lift} \ 2 \ (: \) \)
- \( \text{liftNotMem} \ x \ xs \quad \Rightarrow \quad \text{liftNot} \ 2 \ (\text{liftMem} \ x \ xs) \)

**translations**

- \( \text{liftFinset} \ 2 \ 3 \ (\text{liftargs} \ x \ xs) \quad \Rightarrow \quad \text{lift} \ 2 \ \text{CONST} \ \text{insert} \ x \ (\text{liftFinset} \ xs) \)
- \( \text{liftFinset} \ x \quad \Rightarrow \quad \text{lift} \ 2 \ \text{CONST} \ \text{insert} \ x \ (\text{const} \ (\text{CONST} \ \text{Set} \ \text{empty})) \)
- \( \text{liftPair} \ x \ 2 \ 3 \ (\text{liftargs} \ y \ z) \quad \Rightarrow \quad \text{liftPair} \ x \ (\text{liftPair} \ y \ z) \)
- \( \text{liftPair} \quad \Rightarrow \quad \text{lift} \ 2 \ \text{CONST} \ \text{Pair} \)
- \( \text{liftCons} \quad \Rightarrow \quad \text{lift} \ 2 \ \text{CONST} \ \text{Cons} \)
- \( \text{liftApp} \quad \Rightarrow \quad \text{lift} \ 2 \ (\emptyset) \)
- \( \text{liftList} \ 2 \ 3 \ (\text{liftargs} \ x \ xs) \quad \Rightarrow \quad \text{liftCons} \ x \ (\text{liftList} \ xs) \)
- \( \text{liftList} \ x \quad \Rightarrow \quad \text{liftCons} \ x \ (\text{const} \ []) \)

\[ w \models \lnot \ A \leftarrow \text{liftNot} \ A \ w \]
\[ w \models A \wedge B \leftarrow \text{liftAnd} \ A \ B \ w \]
\[ w \models A \lor B \leftarrow \text{liftOr} \ A \ B \ w \]
\[ w \models A \rightarrow B \leftarrow \text{liftImp} \ A \ B \ w \]
\[ w \models u = v \leftarrow \text{liftEqu} \ u \ v \ w \]
\[ w \models \forall x. \ A \leftarrow \text{RAAll} \ x \ A \ w \]
\[ w \models \exists x. \ A \leftarrow \text{REx} \ x \ A \ w \]
\[ w \models \exists! x. \ A \leftarrow \text{REx1} \ x \ A \ w \]

**syntax (ASCII)**

23
2.3 Lemmas and Tactics

lemma intD[dest]: ⊨ A → w |= A

proof -
  assume a: A
  from a have ∀ w. w |= A by (auto simp add: Valid-def)
  thus ?thesis ..
  qed

lemma intI [intro!]: assumes P1:(∀ w. w |= A) shows ⊨ A
  using assms by (auto simp: Valid-def)

Basic unlifting introduces a parameter w and applies basic rewrites, e.g ⊨ F = G becomes F w = G w and ⊨ F → G becomes F w → G w.

method-setup int-unlift = ⟨
  Scan.succeed (fn ctxt => SIMPLE-METHOD'
    (resolve-tac ctxt (2EX técn intI) THEN' rewrite-goal-tac ctxt @{thms intensional-rews}))
⟩ method to unlift and followed by intensional rewrites

lemma inteq-reflection: assumes P1: ⊨ x=y shows (x ≡ y)

proof -
  from P1 have P2: ∀ w. x w = y w by (unfold Valid-def unI-lift2)
  hence P3:x=y by blast
  thus x ≡ y by (rule eq-reflection)
  qed

lemma int-simps:
  ⊨ (x=x) = #True
  ⊨ (¬ #True) = #False
  ⊨ (¬ #False) = #True
  ⊨ (¬¬ P) = P
  ⊨ ((¬ P) = P) = #False
  ⊨ (P = (¬P)) = #False
  ⊨ (P ≠ Q) = (P = (¬ Q))
  ⊨ (#True=P) = P

24
\(\vdash (P \rightarrow \#\text{True}) = P\)
\(\vdash (\#\text{True} \rightarrow P) = P\)
\(\vdash (\#\text{False} \rightarrow P) = \#\text{True}\)
\(\vdash (P \rightarrow \#\text{True}) = \#\text{True}\)
\(\vdash (P \rightarrow P) = \#\text{True}\)
\(\vdash (P \rightarrow \#\text{False}) = (\neg P)\)
\(\vdash (P \rightarrow \neg P) = (\neg P)\)
\(\vdash (P \land \#\text{True}) = P\)
\(\vdash (\#\text{True} \land P) = P\)
\(\vdash (P \land \#\text{False}) = \#\text{False}\)
\(\vdash (\#\text{False} \land P) = \#\text{False}\)
\(\vdash (P \land \neg P) = P\)
\(\vdash (P \land \neg P) = \#\text{False}\)
\(\vdash (\neg P \land P) = \#\text{False}\)
\(\vdash (P \lor \#\text{True}) = \#\text{True}\)
\(\vdash (\#\text{True} \lor P) = \#\text{True}\)
\(\vdash (P \lor \#\text{False}) = P\)
\(\vdash (\#\text{False} \lor P) = P\)
\(\vdash (P \lor P) = P\)
\(\vdash (P \lor \neg P) = \#\text{True}\)
\(\vdash (\neg P \lor P) = \#\text{True}\)
\(\vdash (\forall x. P) = P\)
\(\vdash (\exists x. P) = P\)

by auto

**lemmas** intensional-simps[simp] = int-simps[THEN inteq-reflection]

**method-setup** int-rewrite = ⟨⟨Scan.succeed (fn ctxt => SIMPLE-METHOD (rewrite-goal-tac ctxt \{thms
intensional-simps\})))⟩⟩

rewrite method at intensional level

**lemma** Not-Rall: \(\vdash (\neg (\forall x. F x)) = (\exists x. \neg F x)\)
by auto

**lemma** Not-Rex: \(\vdash (\neg (\exists x. F x)) = (\forall x. \neg F x)\)
by auto

**lemma** TrueW [simp]: \(\vdash \#\text{True}\)
by auto

**lemma** int-eg: \(\vdash X = Y \implies X = Y\)
by (auto simp: inteq-reflection)

**lemma** int-iffI:
assumes \(\vdash F \rightarrow G\) and \(\vdash G \rightarrow F\)
shows \(\vdash F = G\)
using assms by force
lemma int-iffD1: assumes $h: \vdash F = G$ shows $\vdash F \rightarrow G$
  using $h$ by auto

lemma int-iffD2: assumes $h: \vdash F = G$ shows $\vdash G \rightarrow F$
  using $h$ by auto

lemma lift-imp-trans: assumes $\vdash A \rightarrow B$ and $\vdash B \rightarrow C$
  shows $\vdash A \rightarrow C$
  using assms by force

lemma lift-imp-neg: assumes $\vdash A \rightarrow B$ shows $\vdash \neg B \rightarrow \neg A$
  using assms by auto

lemma lift-and-com: $\vdash (A \land B) = (B \land A)$
  by auto

end

3 Semantics

theory Semantics
imports Sequence Intensional
begin

This theory mechanises a shallow embedding of TLA\(^*\) using the Sequence and Intensional theories. A shallow embedding represents TLA\(^*\) using Isabelle/HOL predicates, while a deep embedding would represent TLA\(^*\) formulas and pre-formulas as mutually inductive datatypes\(^1\). The choice of a shallow over a deep embedding is motivated 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 both formulas and pre-formulas. Finally, since our target is system verification rather than proving meta-properties of TLA\(^*\), which requires a deep embedding, a shallow embedding is more fit for purpose.

3.1 Types of Formulas

To mechanise the TLA\(^*\) semantics, the following type abbreviations are used:

<table>
<thead>
<tr>
<th>Type-synonym</th>
<th>Abbreviation</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>type-synonym</td>
<td>('a,'b) formfun = 'a seq =&gt; 'b</td>
<td></td>
</tr>
<tr>
<td>type-synonym</td>
<td>'a formula = ('a,bool) formfun</td>
<td></td>
</tr>
<tr>
<td>type-synonym</td>
<td>('a,'b) stfun = 'a =&gt; 'b</td>
<td></td>
</tr>
<tr>
<td>type-synonym</td>
<td>'a stpred = ('a,bool) stfun</td>
<td></td>
</tr>
</tbody>
</table>

\(^1\) See e.g. [10] for a discussion about deep vs. shallow embeddings in Isabelle/HOL.
instance
  \texttt{fun :: (type,type) world ..}

instance
  \texttt{prod :: (type,type) world ..}

Pair and function 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.

### 3.2 Semantics of TLA*

The semantics of TLA* is defined.

**definition** \texttt{always :: ('a::world) formula ⇒ 'a formula}

\texttt{where always F ≡ \lambda s. \forall n. (s \mid s n) \models F}

**definition** \texttt{nexts :: ('a::world) formula ⇒ 'a formula}

\texttt{where nexts F ≡ \lambda s. \text{tail } s \mid = F}

**definition** \texttt{before :: ('a::world,'b) stfun ⇒ ('a,'b) formfun}

\texttt{where before f ≡ \lambda s. \text{first } s \mid = f}

**definition** \texttt{after :: ('a::world,'b) stfun ⇒ ('a,'b) formfun}

\texttt{where after f ≡ \lambda s. \text{second } s \mid = f}

**definition** \texttt{unch :: ('a::world,'b) stfun ⇒ 'a formula}

\texttt{where unch v ≡ \lambda s. \text{unch } v \models = (after v) = (before v)}

**definition** \texttt{action :: ('a::world) formula ⇒ ('a,'b) stfun ⇒ 'a formula}

\texttt{where action P v ≡ \lambda s. \forall i. ((s \mid s i) \models P) \lor ((s \mid s i) \models unch v)}

### 3.2.1 Concrete Syntax

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

**syntax**

- \texttt{always :: lift ⇒ lift ((□-) [90] 90)}
- \texttt{nexts :: lift ⇒ lift ((〇-) [90] 90)}
- \texttt{action :: [lift, lift] ⇒ lift ((□-) [90]) [20,1000] 90)}
- \texttt{-before :: lift ⇒ lift (({-}) [100] 99)}
- \texttt{after :: lift ⇒ lift (({-}) [100] 99)}
- \texttt{prime :: lift ⇒ lift (({-}) [100] 99)}
- \texttt{unch :: lift ⇒ lift ((Unchanged) [100] 99)}
- \texttt{TEMP :: lift ⇒ 'b ((TEMP -))}

**syntax (ASCII)**

- \texttt{always :: lift ⇒ lift ((-)] [90] 90)}
- \texttt{nexts :: lift ⇒ lift ((Next) [90] 90)}
3.3 Abbreviations

Some standard temporal abbreviations, with their concrete syntax.

**Definition**  \( \text{actrans} : (\text{a}::\text{world}) \text{ formula} \Rightarrow (\text{a}, \text{b}) \text{ stfun} \Rightarrow \text{a formula} \)

**where** \( \text{actrans} \ P \ v \equiv \text{TEMP}(P \lor \text{unch} \ v) \)

**Definition**  \( \text{eventually} : (\text{a}::\text{world}) \text{ formula} \Rightarrow \text{a formula} \)

**where** \( \text{eventually} \ F \equiv \text{LIFT}(\neg \Box(\neg F)) \)

**Definition**  \( \text{angle-action} : (\text{a}::\text{world}) \text{ formula} \Rightarrow (\text{a}, \text{b}) \text{ stfun} \Rightarrow \text{a formula} \)

**where** \( \text{angle-action} \ P \ v \equiv \text{LIFT}(\neg \Box(\neg P)) \)

**Definition**  \( \text{angle-actrans} : (\text{a}::\text{world}) \text{ formula} \Rightarrow (\text{a}, \text{b}) \text{ stfun} \Rightarrow \text{a formula} \)

**where** \( \text{angle-actrans} \ P \ v \equiv \text{TEMP}(\neg \text{actrans(\text{LIFT}(\neg P)) \ v}) \)

**Definition**  \( \text{leadsto} : (\text{a}::\text{world}) \text{ formula} \Rightarrow \text{a formula} \Rightarrow \text{a formula} \)

**where** \( \text{leadsto} \ P \ Q \equiv \text{LIFT} \Box(P \rightarrow \text{eventually} \ Q) \)

### 3.3.1 Concrete Syntax

**Syntax** (ASCII)

**actrans** :: \([\text{lift, lift}] \Rightarrow \text{lift (([\neg ]\text{‘}(-)) \hspace{1em} [20,1000] \hspace{1em} 90)}\)

**eventually** :: \([\text{lift, lift}] \Rightarrow \text{lift ((<<>\text{‘}(-)) \hspace{1em} [20,1000] \hspace{1em} 90)}\)

**angle-action** :: \([\text{lift, lift}] \Rightarrow \text{lift (((\nabla\text{‘})\text{‘}(-)) \hspace{1em} [20,1000] \hspace{1em} 90)}\)

**angle-actrans** :: \([\text{lift, lift}] \Rightarrow \text{lift (((\nabla\text{‘}(-)) \hspace{1em} [20,1000] \hspace{1em} 90)}\)

**leadsto** :: \([\text{lift, lift}] \Rightarrow \text{lift ((~\sim \text{‘}(-)) \hspace{1em} [26,25] \hspace{1em} 25)}\)

**Translations**

**actrans** = \text{CONST actrans}

**eventually** = \text{CONST eventually}

**angle-action** = \text{CONST angle-action}
3.4 Properties of Operators

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

**Lemma eventually-defs**: $(w \models \Diamond F) = (\exists \ n. (w|_n \models F))$

**Proof**: (simp add: eventually-def always-def)

**Lemma angle-action-defs**: $(w \models \Diamond\langle P \rangle \cdot v \not\models A)$

**Proof**: (simp add: angle-action-def action-def unch-def)

**Lemma unch-defs**: $(w \models \text{Unchanged } v) = ((w|_\text{second } w) \models v) = ((w|_\text{first } w) \models v)$

**Proof**: (simp add: unch-def before-def nexts-def after-def tail-def suffix-def first-def second-def)

**Lemma linalw**:

**Assumes** $h1$: $a \leq b$ and $h2$: $(w|_a \models A)$

**Shows** $(w|_b \models A)$

**Proof**: (clarsimp simp: always-def)

3.5 Invariance Under Stuttering

A key feature of TLA* is that specification at different abstraction levels can be compared. The soundness of this relies on the stuttering invariance of formulas. Since the embedding is shallow, it cannot be shown that a generic TLA* formula is stuttering invariant. However, this section will show that each operator is stuttering invariant or preserves stuttering invariance in an appropriate sense, which can be used to show stuttering invariance for given specifications.

Formula $F$ is stuttering invariant if for any two similar behaviours (i.e., sequences of states), $F$ holds in one iff it holds in the other. The definition is generalised to arbitrary expressions, and not just predicates.

**Definition stutinv**: $(a, b) \text{ formfun } \Rightarrow \text{ bool}$

**Where** $stutinv F \equiv \forall \sigma. \tau. \sigma \approx \tau \longrightarrow (\sigma \models F) = (\tau \models F)$

The requirement for stuttering invariance is too strong for pre-formulas. For example, an action formula specifies a relation between the first two states of a behaviour, and will rarely be satisfied by a stuttering step. This is why pre-formulas are “protected” by (square or angle) brackets in TLA*:
the only place a pre-formula $P$ can be used is inside an action: \( \square[P]\). To show that \( \square[P]\) is stuttering invariant, must be shown that a slightly weaker predicate holds for $P$. For example, if $P$ contains a term of the form \( \circ \circ Q \), then it is not a well-formed pre-formula, thus \( \square[P]\) is not stuttering invariant. This weaker version of stuttering invariance has been named near stuttering invariance.

**definition** \( \text{nstutinv} :: (\text{a}', \text{b}') \rightarrow \text{bool} \)

**where** \( \text{nstutinv} \equiv \forall \sigma \tau. (\text{first } \sigma = \text{first } \tau) \land (\text{tail } \sigma) \approx (\text{tail } \tau) \rightarrow (\sigma \models P) = (\tau \models P) \)

**syntax**

\[-\text{stutinv} :: \text{lift} \Rightarrow \text{bool} ((\text{STUTINV} -) [40] 40)\]

\[-\text{nstutinv} :: \text{lift} \Rightarrow \text{bool} ((\text{NSTUTINV} -) [40] 40)\]

**translations**

\[-\text{stutinv} := \text{CONST stutinv}\]

\[-\text{nstutinv} := \text{CONST nstutinv}\]

Predicate \( \text{STUTINV } F \) formalises stuttering invariance for formula $F$. That is if two sequences are similar $s \approx t$ (equal up to stuttering) then the validity of $F$ under both $s$ and $t$ are equivalent. Predicate \( \text{NSTUTINV } P \) should be read as nearly stuttering invariant – and is required for some stuttering invariance proofs.

**lemma** \( \text{stutinv-strictly-stronger} :\)

**assumes** \( h : \text{STUTINV } F \)

**shows** \( \text{NSTUTINV } F \)

**unfolding** \( \text{nstutinv-def} \)

**proof** (clarify)

**fix** \( s \text{ t :: nat} \Rightarrow 'a\)

**assume** \( a1 : \text{first } s = \text{first } t \text{ and } a2 : (\text{tail } s) \approx (\text{tail } t)\)

**have** \( s \approx t\)

**proof**

**have** \( t g1 : (\text{first } s) \#\# (\text{tail } s) = s \text{ by (rule seq-app-first-tail)}\)

**have** \( t g2 : (\text{first } t) \#\# (\text{tail } t) = t \text{ by (rule seq-app-first-tail)}\)

**with** \( a1 \text{ have } t g2' : (\text{first } s) \#\# (\text{tail } t) = t \text{ by simp}\)

**from** \( a2 \text{ have } (\text{first } s) \#\# (\text{tail } s) \approx (\text{first } s) \#\# (\text{tail } t) \text{ by (rule app-seqsimilar)}\)

**with** \( t g1 \text{ t g2' show } \text{thesis by simp}\)

**qed**

**with** \( h \text{ show } (s \models F) = (t \models F) \text{ by (simp add: stutinv-def)}\)

**qed**

### 3.5.1 Properties of \( -\text{stutinv} \)

This subsection proves stuttering invariance, preservation of stuttering invariance and introduction of stuttering invariance for different formulas. First, state predicates are stuttering invariant.

**theorem** \( \text{stut-before} : \text{STUTINV } $F\)

**proof** (clarsimp simp simp: stutinv-def)
fix $s \cdot t :: \text{'}a \text{ seq}\\assume a1: s \approx t\\\text{hence (first } s) = (\text{first } t) \text{ by (rule sim-first) }\\\text{thus } (s \models F) = (t \models F) \text{ by (simp add: before-def) }\\\text{qed}\\\\\text{lemma nstut-after: NSTUTINV } F\$\\\text{proof (clarsimp simp: nstutinv-def) }\\\text{fix } s \cdot t :: \text{'}a \text{ seq}\\\text{assume a1: tail } s \approx \text{tail } t\\\text{thus } (s \mathbin{|} F) = (t \mathbin{|} F) \text{ by (simp add: after-def tail-sim-second) }\\\text{qed}\\\\\text{The always operator preserves stuttering invariance. }\\\text{theorem stut-always: assumes H:STUTINV } F \text{ shows STUTINV } \Box F\\\text{proof (clarsimp simp: stutinv-def) }\\\text{fix } s \cdot t :: \text{'}a \text{ seq}\\\\text{assume a2: } s \approx t\\\text{show } (s \mathbin{|} (\Box F)) = (t \mathbin{|} (\Box F))\\\text{proof }\\\text{assume a1: } t \models \Box F\\\text{show } s \models \Box F\\\text{proof (clarsimp simp: always-def) }\\\text{fix } n\\\from a2[THEN sim-step] \text{ obtain } m \text{ where } m: s \mathbin{|}_s n \approx t \mathbin{|}_s m \text{ by blast }\\\from a1 \text{ have } (t \mathbin{|}_s m) \models F \text{ by (simp add: always-def) }\\\text{with } H \text{ m show } (s \mathbin{|}_s n) \models F \text{ by (simp add: stutinv-def) }\\\text{qed}\\\text{next}\\\text{assume a1: } s \models (\Box F)\\\text{show } t \models (\Box F)\\\text{proof (clarsimp simp: always-def) }\\\text{fix } n\\\from a2[THEN seqsim-sym, THEN sim-step] \text{ obtain } m \text{ where } m: t \mathbin{|}_s n \approx s \mathbin{|}_s m \text{ by blast }\\\from a1 \text{ have } (s \mathbin{|}_s m) \models F \text{ by (simp add: always-def) }\\\text{with } H \text{ m show } (t \mathbin{|}_s n) \models F \text{ by (simp add: stutinv-def) }\\\text{qed}\\\text{qed}\\\text{qed}\\\\\text{Assuming that formula } P \text{ is nearly stuttering invariant then } \Box [P] \text{-v will be stuttering invariant. }\\\text{lemma stut-action-lemma: }\\\text{assumes H: NSTUTINV } P \text{ and st: } s \approx t \text{ and } P: t \models \Box [P] \text{-v }\\\text{shows } s \models \Box [P] \text{-v }\\\text{proof (clarsimp simp: action-def) }\\\text{fix } n\\\text{assume } \lnot ((s \mathbin{|}_s n) \models \text{Unchanged } v)
The lemmas below show that propositional and predicate operators preserve

\[ \text{hence } v : v \ (s \ (\text{Suc } n)) \neq v \ (s \ n) \]

by \(\text{(simp add: unch-defs first-def second-def suffix-def)}\)

from \(\text{st} \ [\text{THEN} \ \text{sim-step}] \ \text{obtain } m \ \text{where}
\]

\[ a2' : \ s \ |_s n \approx t \ |_s m \]

\& \ (s |_s \text{Suc } n \approx t |_s \text{Suc } m \lor s |_s \text{Suc } n \approx t |_s m) \ldots \]

\[ \text{hence } g1 : \ (s |_s n \approx t |_s m) \ \text{by simp} \]

\[ \text{hence } g1'' : \ \text{first } (s |_s n) = \text{first } (t |_s m) \ \text{by simp add: sim-first} \]

\[ \text{hence } g1'' : \ s \ n = t \ m \ \text{by simp add: suffix-def first-def} \]

from \(a2' \ \text{have } g2 : \ s \ |_s \text{Suc } n \approx t \ |_s \text{Suc } m \lor s \ |_s \text{Suc } n \approx t \ |_s m \ \text{by simp} \]

\[ \text{from } P \ \text{have } a1'' : \ ((t |_s m) \models P) \lor ((t |_s m) \models \text{Unchanged } v) \ \text{by simp add: action-def} \]

\[ \text{from } g2 \ \text{show } (s |_s n) \models P \]

\[ \text{proof} \]

\[ \text{assume } s \ |_s \text{Suc } n \approx t \ |_s m \]

\[ \text{hence } \text{first } (s |_s \text{Suc } n) = \text{first } (t |_s m) \ \text{by simp add: sim-first} \]

\[ \text{hence } s \ (\text{Suc } n) = t \ m \ \text{by simp add: suffix-def first-def} \]

\[ \text{with } g1' \ v \ \text{show } ?\text{thesis by simp — by contradiction} \]

next

\[ \text{assume } a3 : \ s \ |_s \text{Suc } n \approx t \ |_s \text{Suc } m \]

\[ \text{hence } \text{first } (s |_s \text{Suc } n) = \text{first } (t |_s \text{Suc } m) \ \text{by simp add: sim-first} \]

\[ \text{hence } a3' : s \ (\text{Suc } n) = t \ (\text{Suc } m) \ \text{by simp add: suffix-def first-def} \]

from \(a1'' \ \text{show } ?\text{thesis} \)

\[ \text{proof} \]

\[ \text{assume } (t |_s m) \models \text{Unchanged } v \]

\[ \text{hence } v \ (t \ (\text{Suc } m)) = v \ (t \ m) \]

\[ \text{by simp add: unch-defs first-def second-def suffix-def} \]

\[ \text{with } g1'' \ a3' \ v \ \text{show } ?\text{thesis by simp — again, by contradiction} \]

next

\[ \text{assume } a4 : (t |_s m) \models P \]

\[ \text{from } a3 \ \text{have } \text{tail } (s |_s n) \approx \text{tail } (t |_s m) \ \text{by simp add: tail-def suffix-plus} \]

\[ \text{with } H \ g1'' \ a4 \ \text{show } ?\text{thesis by auto simp: nstutinv-def} \]

\[ \text{qed} \]

\[ \text{qed} \]

\[ \text{theorem } \text{stut-action: assumes } H : \text{NSTUTINV } P \ \text{shows } \text{STUTINV } \Box[P]-v \]

\[ \text{proof} \ (\text{clarsimp simp: stutinv-def}) \]

\[ \text{fix } s \ t :: 'a \text{ seq} \]

\[ \text{assume } \text{st} : s \approx t \]

\[ \text{show } (s \models \Box[P]-v) = (t \models \Box[P]-v) \]

\[ \text{proof} \]

\[ \text{assume } t \models \Box[P]-v \]

\[ \text{with } H \ \text{st show } s \models \Box[P]-v \ \text{by (rule stat-action-lemma)} \]

next

\[ \text{assume } s \models \Box[P]-v \]

\[ \text{with } H \ \text{st[THEN seqsim-sym] show } t \models \Box[P]-v \ \text{by (rule stat-action-lemma)} \]

\[ \text{qed} \]

\[ \text{qed} \]

The lemmas below shows that propositional and predicate operators preserve
stuttering invariance.

**lemma** stut-and: \([\text{STUTINV } F; \text{STUTINV } G] \Rightarrow \text{STUTINV } (F \land G)\)  
by (simp add: stutinv-def)

**lemma** stut-or: \([\text{STUTINV } F; \text{STUTINV } G] \Rightarrow \text{STUTINV } (F \lor G)\)  
by (simp add: stutinv-def)

**lemma** stut-imp: \([\text{STUTINV } F; \text{STUTINV } G] \Rightarrow \text{STUTINV } (F \rightarrow G)\)  
by (simp add: stutinv-def)

**lemma** stut-eq: \([\text{STUTINV } F; \text{STUTINV } G] \Rightarrow \text{STUTINV } (F = G)\)  
by (simp add: stutinv-def)

**lemma** stut-noteq: \([\text{STUTINV } F; \text{STUTINV } G] \Rightarrow \text{STUTINV } (F \neq G)\)  
by (simp add: stutinv-def)

**lemma** stut-not: \(\text{STUTINV } F \Rightarrow \text{STUTINV } (\neg F)\)  
by (simp add: stutinv-def)

**lemma** stut-all: \((\forall x. \text{STUTINV } (F x)) \Rightarrow \text{STUTINV } (\forall x. F x)\)  
by (simp add: stutinv-def)

**lemma** stut-ex: \((\exists x. \text{STUTINV } (F x)) \Rightarrow \text{STUTINV } (\exists x. F x)\)  
by (simp add: stutinv-def)

**lemma** stut-const: \(\text{STUTINV } \# c\)  
by (simp add: stutinv-def)

**lemma** stut-fun1: \(\text{STUTINV } X \Rightarrow \text{STUTINV } (f <X>)\)  
by (simp add: stutinv-def)

**lemma** stut-fun2: \([\text{STUTINV } X; \text{STUTINV } Y] \Rightarrow \text{STUTINV } (f <X,Y>)\)  
by (simp add: stutinv-def)

**lemma** stut-fun3: \([\text{STUTINV } X; \text{STUTINV } Y; \text{STUTINV } Z] \Rightarrow \text{STUTINV } (f <X,Y,Z>)\)  
by (simp add: stutinv-def)

**lemma** stut-fun4: \([\text{STUTINV } X; \text{STUTINV } Y; \text{STUTINV } Z; \text{STUTINV } W] \Rightarrow \text{STUTINV } (f <X,Y,Z,W>)\)  
by (simp add: stutinv-def)

**lemma** stut-plus: \([\text{STUTINV } x; \text{STUTINV } y] \Rightarrow \text{STUTINV } (x+y)\)  
by (simp add: stutinv-def)

**3.5.2 Properties of -nstutinv**

This subsection shows analogous properties about near stuttering invariance.
If a formula $F$ is stuttering invariant then $\circ F$ is nearly stuttering invariant.

**Lemma nstut-nexts:** Assumes $H$: $\text{STUTINV } F$ shows $\text{NSTUTINV } \circ F$

Using $H$ by $(\text{simp add: stutter-inv_def nstut-inv_def nexts-def})$

The lemmas below show that propositional and predicate operators preserve near stuttering invariance.

**Lemma nstut-and:** $\text{[NSTUTINV } F;\text{NSTUTINV } G]\implies\text{NSTUTINV } (F \land G)$

by $(\text{auto simp: nstut-inv-def})$

**Lemma nstut-or:** $\text{[NSTUTINV } F;\text{NSTUTINV } G]\implies\text{NSTUTINV } (F \lor G)$

by $(\text{auto simp: nstut-inv-def})$

**Lemma nstut-imp:** $\text{[NSTUTINV } F;\text{NSTUTINV } G]\implies\text{NSTUTINV } (F \rightarrow G)$

by $(\text{auto simp: nstut-inv-def})$

**Lemma nstut-eq:** $\text{[NSTUTINV } F;\text{NSTUTINV } G]\implies\text{NSTUTINV } (F = G)$

by $(\text{force simp: nstut-inv-def})$

**Lemma nstut-not:** $\text{NSTUTINV } F\implies\text{NSTUTINV } (\neg F)$

by $(\text{auto simp: nstut-inv-def})$

**Lemma nstut-noteq:** $\text{[NSTUTINV } F;\text{NSTUTINV } G]\implies\text{NSTUTINV } (F \neq G)$

by $(\text{simp add: nstut-eq nstut-not})$

**Lemma nstut-all:** $(\forall x.\text{NSTUTINV } (F x))\implies\text{NSTUTINV } (\forall x. F x)$

by $(\text{auto simp: nstut-inv-def})$

**Lemma nstut-ex:** $(\exists x.\text{NSTUTINV } (F x))\implies\text{NSTUTINV } (\exists x. F x)$

by $(\text{auto simp: nstut-inv-def})$

**Lemma nstut-const:** $\text{NSTUTINV } #c$

by $(\text{auto simp: nstut-inv-def})$

**Lemma nstut-fun1:** $\text{NSTUTINV } X\implies\text{NSTUTINV } (f <X>)$

by $(\text{force simp: nstut-inv-def})$

**Lemma nstut-fun2:** $\text{[NSTUTINV } X;\text{NSTUTINV } Y]\implies\text{NSTUTINV } (f <X,Y>)$

by $(\text{force simp: nstut-inv-def})$

**Lemma nstut-fun3:** $\text{[NSTUTINV } X;\text{NSTUTINV } Y;\text{NSTUTINV } Z]\implies\text{NSTUTINV } (f <X,Y,Z>)$

by $(\text{force simp: nstut-inv-def})$

**Lemma nstut-fun4:** $\text{[NSTUTINV } X;\text{NSTUTINV } Y;\text{NSTUTINV } Z;\text{NSTUTINV } W]\implies\text{NSTUTINV } (f <X,Y,Z,W>)$

by $(\text{force simp: nstut-inv-def})$

**Lemma nstut-plus:** $\text{[NSTUTINV } x;\text{NSTUTINV } y]\implies\text{NSTUTINV } (x + y)$

by $(\text{simp add: nstut-fun2})$
3.5.3 Abbreviations

We show the obvious fact that the same properties holds for abbreviated operators.

**lemmas nstut-before = stat-before[THEN statinv-strictly-stronger]**

**lemma nstut-unch: NSTUTINV (Unchanged v)**

**proof** (unfold unch-def)
- **have g1: NSTUTINV v$ by** (rule nstut-after)
- **have NSTUTINV $v by** (rule stat-before[THEN statinv-strictly-stronger])
- **with g1 show NSTUTINV (v$ = $v) by** (rule nstut-eq)

qed

Formulas $[P]\cdot v$ are not TLA* formulas by themselves, but we need to reason about them when they appear wrapped inside $\square[-].v$. We only require that it preserves nearly stuttering invariance. Observe that $[P]\cdot v$ trivially holds for a stuttering step, so it cannot be stuttering invariant.

**lemma nstut-actrans: NSTUTINV P $\Rightarrow$ NSTUTINV $[P]\cdot v**

by (simp add: actrans-def nstut-unch nstut-or)

**lemma stat-eventually: STUTINV F $\Rightarrow$ STUTINV $\lozenge F**

by (simp add: eventually-def stut-not stut-always)

**lemma stat-leadsto: [STUTINV F; STUTINV G] $\Rightarrow$ STUTINV (F $\leadsto$ G)**

by (simp add: leadsto-def stut-always stut-eventually stut-imp)

**lemma stat-angle-action: NSTUTINV P $\Rightarrow$ STUTINV $\Diamond (P)\cdot v**

by (simp add: angle-action-def nstut-not stut-action stut-not)

**lemma nstut-angle-actrans: NSTUTINV P $\Rightarrow$ NSTUTINV $\langle P \rangle\cdot v**

by (simp add: angle-actrans-def nstut-not nstut-actrans)

**lemmas stutinvs = stat-before stat-always stat-action**

**stut-and stat-or stat-imp stat-eq stat-noteq stat-not**

**stut-all stat-ex stat-eventually stat-leadsto stat-angle-action stat-const**

**stut-fun1 stat-fun2 stat-fun3 stat-fun4**

**lemmas nstutinvs = nstut-after nstut-nexts nstut-actrans**

**nstut-unch nstut-and nstut-or nstut-imp nstut-eq nstut-noteq nstut-not**

**nstut-all nstut-ex nstut-angle-actrans stutinv-strictly-stronger**

**nstut-fun1 nstut-fun2 nstut-fun3 nstut-fun4 stutinvs[THEN statinv-strictly-stronger]**

**lemmas bothstutinvs = stutinvs nstutinvs**

end
4 Reasoning about PreFormulas

theory PreFormulas
imports Semantics
begin

Semantic separation of formulas and pre-formulas requires a deep embedding. We introduce a syntactically distinct notion of validity, written $\sim A$, for pre-formulas. Although it is semantically identical to $\vdash A$, it helps users distinguish pre-formulas from formulas in TLA$^*$ proofs.

definition PreValid :: ('w::world) form $\Rightarrow$ bool
where PreValid $A \equiv \forall w. \ w \models A$

syntax
  -$\sim A$ :: lift $\Rightarrow$ bool   ((|$\sim \cdot$) 5)

translations
  -$\sim A$ $\Rightarrow$ CONST PreValid

lemma prefD [dest]: $\sim A \Longrightarrow w \models A$
  by (simp add: PreValid-def)

lemma prefI [intro!]: $(\forall w. w \models A) \Longrightarrow \sim A$
  by (simp add: PreValid-def)

method-setup pref-unlift = ⟨⟨ Scan.succeed (fn ctxt $=>$ SIMPLE-METHOD'  
  (resolve-tac ctxt @{| thms prefI | THEN' rewrite-goal-tac ctxt @{| thms intensional-rews |}}) 
) ⟩⟩

int-unlift for PreFormulas

lemma pref-eq-reflection: assumes P1: $\sim x=y$ shows $(x \equiv y)$
  using P1 by (intro eq-reflection) force

lemma pref-True[simp]: $\sim \# True$
  by auto

lemma pref-eq: $\sim X = Y \Longrightarrow X = Y$
  by (auto simp: pref-eq-reflection)

lemma pref-iffI:
  assumes $\sim F \rightarrow G$ and $\sim G \rightarrow F$
  shows $\sim F = G$
  using assms by force

lemma pref-iffD1: assumes $\sim F = G$ shows $\sim F \rightarrow G$
  using assms by auto

lemma pref-iffD2: assumes $\sim F = G$ shows $\sim G \rightarrow F$
  using assms by auto

36
lemma unl-pref-imp:
assumes \( \neg F \rightarrow G \) shows \( \wedge w \models F \implies w \models G \)
using assms by auto

lemma pref-imp-trans:
assumes \( \neg F \rightarrow G \) and \( \neg G \rightarrow H \)
shows \( \neg F \rightarrow H \)
using assms by force

4.1 Lemmas about Unchanged

Many of the TLA\(^*\) axioms only require a state function witness which leaves
the state space unchanged. An obvious witness is the \( id \) function. The
lemmas require that the given formula is invariant under stuttering.

lemma pre-id-unch: assumes \( h: \text{stutinv } F \)
shows \( \neg F \wedge \text{Unchanged } id \rightarrow \bigcirc F \)
proof (pref-unlift, clarify)
fix \( s \)
assume \( a1: s \models F \) and \( a2: s \models \text{Unchanged } id \)
from \( a2 \) have \((id \text{ (second } s) = id \text{ (first } s))\) by (simp add: unch-defs)

hence \( s \approx (\text{tail } s) \) by (simp add: addfirststut)
with \( h \) \( a1 \) have \((\text{tail } s) \models F \) by (simp add: statinv-def)
thus \( s \models \bigcirc F \) by (unfold nexts-def)
qed

lemma pre-ex-unch:
assumes \( h: \text{stutinv } F \)
shows \( \exists (v::'a::\text{world} \Rightarrow 'a). \ (\neg F \wedge \text{Unchanged } v \rightarrow \bigcirc F) \)
using pre-id-unch[OF \( h \)] by blast

lemma unch-pair: \( \neg \text{Unchanged } (x,y) = (\text{Unchanged } x \wedge \text{Unchanged } y) \)
by (auto simp: nexts-def before-def after-def unch-def)

lemmas unch-eq1 = unch-pair[THEN pref-eq]
lemmas unch-eq2 = unch-pair[THEN pref-eq-reflection]

lemma angle-actrans-sem: \( \neg \langle F \rangle \cdot v = (F \wedge v\$ \neq v\$) \)
by (auto simp: angle-actrans-def actrans-def unch-def)

lemmas angle-actrans-sem-eq = angle-actrans-sem[THEN pref-eq]

4.2 Lemmas about after

lemma after-const: \( \neg (\#e)\' = \#e \)
by (auto simp: nexts-def before-def after-def)

lemma after-fun1: \( \neg f\langle x\rangle' = f\langle x'\rangle \)
by (auto simp: nexts-def before-def after-def)
### 4.3 Lemmas about before

**Lemma after-fun2**: \( f<x,y> \sim f <x',y'> \)

by (auto simp: nexts-def before-def after-def)

**Lemma after-fun3**: \( f<x,y,z> \sim f <x',y',z'> \)

by (auto simp: nexts-def before-def after-def)

**Lemma after-fun4**: \( f<x,y,z,zz> \sim f <x',y',z',zz'> \)

by (auto simp: nexts-def before-def after-def)

**Lemma after-forall**: \( \forall x. P x \sim \forall x. (P x)' \)

by (auto simp: nexts-def before-def after-def)

**Lemma after-exists**: \( \exists x. P x \sim \exists x. (P x)' \)

by (auto simp: nexts-def before-def after-def)

**Lemma after-exists1**: \( \exists! x. P x \sim \exists! x. (P x)' \)

by (auto simp: nexts-def before-def after-def)

**Lemmas all-after** = after-const after-fun1 after-fun2 after-fun3 after-fun4 after-forall after-exists after-exists1

**Lemmas all-after-unl** = all-after[THEN prefD]

**Lemmas all-after-eq** = all-after[THEN prefeq-reflection]

**Lemma before-const**: \( \# c = \# c \)

by (auto simp: before-def)

**Lemma before-fun1**: \( f<x> = f <x> \)

by (auto simp: before-def)

**Lemma before-fun2**: \( f<x,y> = f <x,y> \)

by (auto simp: before-def)

**Lemma before-fun3**: \( f<x,y,z> = f <x,y,z> \)

by (auto simp: before-def)

**Lemma before-fun4**: \( f<x,y,z,zz> = f <x,y,z,zz> \)

by (auto simp: before-def)

**Lemma before-forall**: \( \forall x. P x = \forall x. (P x) \)

by (auto simp: before-def)

**Lemma before-exists**: \( \exists x. P x = \exists x. (P x) \)

by (auto simp: before-def)

**Lemma before-exists1**: \( \exists! x. P x = \exists! x. (P x) \)
lemmas all-before = before-const before-fun1 before-fun2 before-fun3 before-fun4 before-forall before-exists before-exists1

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

4.4 Some general properties

lemma angle-actrans-conj: \( \neg ((F \land G) \mathbin{-v}) = ((F) \mathbin{-v} \land (G) \mathbin{-v}) \)
  by (auto simp: angle-actrans-def actrans-def unch-def)

lemma angle-actrans-disj: \( \neg ((F \lor G) \mathbin{-v}) = ((F) \mathbin{-v} \lor (G) \mathbin{-v}) \)
  by (auto simp: angle-actrans-def actrans-def unch-def)

lemma int-eq-true: \( \vdash P \implies \vdash P = \#True \)
  by auto

lemma pref-eq-true: \( \neg P \implies \neg P = \#True \)
  by auto

4.5 Unlifting attributes and methods

Attribute which unlifts an intensional formula or preformula
ML \( \langle\langle \text{fun unl-rewr ctxt thm = let val unl = (thm RS \{\text{thm intD}\}}) \text{ handle THM -\=> (thm RS \{\text{thm prefD}\}}) handle THM -\=> thm
  val rewr = rewrite-rule ctxt \{\text{thms intensional-rews}\}
in unl |> rewr end; \rangle\rangle \)
attribute-setup unlifted = \( \langle\langle \text{Scan.succeed (Thm.rule-attribute [] (unl-rewr o Context.proof-of))}\rangle\rangle \) unlift intensional formulas

attribute-setup unlift-rule = \( \langle\langle \text{Scan.succeed (Thm.rule-attribute [] (Context.proof-of #> (fn ctxt => Object-Logic.rulify ctxt o unl-rewr ctxt)))}\rangle\rangle \) unlift and rulify intensional formulas

Attribute which turns an intensional formula or preformula into a rewrite rule. Formulas \( F \) that are not equalities are turned into \( F \equiv \#True \).
ML \( \langle\langle \rangle\rangle \)
fun int-rewr thm =
  (thm RS {thm inteq-reflection})
handle THM - => (thm RS {thm prefeq-reflection})
handle THM - => ((thm RS {thm int-eq-true}) RS {thm inteq-reflection})
handle THM - => ((thm RS {thm pre-eq-true}) RS {thm prefeq-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 or preformulas

attribute-setup int-rewrite = ⟨⟨ Scan.succeed (Thm.rule-attribute [] (fn - =>
  int-rewr)) ⟩⟩
produce rewrites from intensional formulas or preformulas
end

5 A Proof System for TLA*

theory Rules
imports PreFormulas
begin

We prove soundness of the proof system of TLA*, from which the system
verification rules from Lamport’s original TLA paper will be derived. This
theory is still state-independent, thus state-dependent enableness proofs,
required for proofs based on fairness assumptions, and flexible quantification,
are not discussed here.

The TLA* paper [8] suggest both a heterogeneous and a homogenous proof
system for TLA*. The homogeneous version eliminates the auxiliary def-
initions from the Preformula theory, creating a single provability relation.
This axiomatisation is based on the fact that a pre-formula can only be used
via the sq rule. In a nutshell, sq is applied to pax1 to pax5, and nex, pre
and pmp are changed to accommodate this. It is argued that while the het-
erogenous version is easier to understand, the homogenous system avoids
the introduction of an auxiliary provability relation. However, the price to
pay is that reasoning about pre-formulas (in particular, actions) has to be
performed in the scope of temporal operators such as □[P]-v, which is no-
tationally quite heavy, We prefer here the heterogeneous approach, which
exposes the pre-formulas and lets us use standard HOL rules more directly.

5.1 The Basic Axioms

theorem fmp: assumes ⊢ F and ⊢ F → G shows ⊢ G
using assms\{unlifted\} by auto

theorem pmp: assumes \(" F \) and \(\neg \ F \rightarrow G\) shows \(\neg G\)
using assms\{unlifted\} by auto

theorem sq: assumes \(\neg \ P\) shows \(\Box[\neg P]\)-v
using assms\{unlifted\} by (auto simp: action-def)

theorem sq: assumes \(\neg \ P\) shows \(\Box[\neg P]\)-v
using assms by auto

theorem sq: assumes \(\neg \ P\) shows \(\Box[\neg P]\)-v
using assms by auto

theorem ax0: \(\neg \ # \ True\)
by auto

theorem ax1: \(\Box \neg F \rightarrow F\)
proof (clarsimp simp: always-def)
fix \(w\)
assume \(\forall n. (w |s n) \models F\)
hence \((w |s 0) \models F\).
thus \(w \models F\) by simp
qed

theorem ax2: \(\Box \neg F \rightarrow \Box[\neg F]\)-v
by (auto simp: always-def suffix-plus)

theorem ax3:
assumes \(\neg \ F \wedge \text{Unchanged } v \rightarrow \Diamond F\)
shows \(\Box[\neg F \rightarrow \Diamond F]\)-v \(\rightarrow (F \rightarrow \Box F)\)
proof (clarsimp simp: always-def)
fix \(w\) \(n\)
assume \(a1: w \models \Box[\neg F \rightarrow \Diamond F]\)-v \(\text{and } a2: w \models F\)
show \((w |s n) \models F\)
proof (induct \(n\))
from \(a2\) show \((w |s 0) \models F\) by simp
next
fix \(m\)
assume \(a3: (w |s m) \models F\)
with \(a1\) \(H[\text{unlifted}]\) show \((w |s (Suc m)) \models F\)
by (auto simp: nexts-def action-def tail-suffix-suc)
qed
qed

theorem ax4: \(\Box[\neg P \rightarrow Q]\)-v \(\rightarrow (\Box[P]\)-v \(\rightarrow \Box[Q]\)-v)\)
by (force simp: action-def)

theorem ax5: \(\Box[v' \neq \$v]\)-v

41
by (auto simp: action-def unch-def)

**Theorem pax0:** \( \sim \neq \text{True} \)
by auto

**Theorem pax1 [simp-und]:** \( \sim (\neg F) = (\neg F) \)
by (auto simp: nexts-def)

**Theorem pax2:**
\[
\sim (F \rightarrow G) \rightarrow (\neg F \rightarrow \neg G)
\]
by (auto simp: nexts-def)

**Theorem pax3:**
\[
\sim \square F \rightarrow \# \square F
\]
by (auto simp: always-def action-def tail-def suffix-plus)

**Theorem pax4:**
\[
\sim \square [P]-v = ([P]-v \land \square [P]-v)
\]
proof (auto)

Fix \( w \)
assert \( w \models \square [P]-v \)
from this unfolded action-def have \( ((w \mid_s 0) \models P) \lor ((w \mid_s 0) \models \text{Unchanged} v) \)
thus \( w \models [P]-v \) by (simp add: actrans-def)
next
fix \( w \)
assert \( w \models \square [P]-v \)
thus \( w \models \square [P]-v \) by (auto simp: nexts-def action-def tail-def suffix-plus)
next
fix \( w \)
assert \( 1: w \models [P]-v \) and \( 2: w \models \square [P]-v \)
show \( w \models \square [P]-v \)
proof (auto simp: action-def)

Fix \( i \)
assert \( 3: \neg (w \mid_s i) \models \text{Unchanged} v \)
show \( (w \mid_s i) \models P \)
proof (cases \( i \))

Assume \( i = 0 \)
with \( 1 \) \& \( 3 \) show \( ?\text{thesis} \) by (simp add: actrans-def)
next
fix \( j \)
assert \( i = \text{Suc} j \)
with \( 2 \) \& \( 3 \) show \( ?\text{thesis} \) by (auto simp: nexts-def action-def tail-def suffix-plus)
qed
qed

**Theorem pax5:**
\[
\sim \neg \square F \rightarrow \square [\neg F]-v
\]
by (auto simp: nexts-def always-def action-def tail-def suffix-plus)

Theorem to show that universal quantification distributes over the always operator. Since the TLA\(^*\) paper only addresses the propositional fragment,
this theorem does not appear there.

**Theorem allT:** \(\vdash (\forall x. \Box (F x)) = (\Box (\forall x. F x))\)
by \((\text{auto simp: always-def})\)

**Theorem allActT:** \(\vdash (\forall x. \Box [F x] \cdot v) = (\Box [\forall x. F x] \cdot v)\)
by \((\text{force simp: action-def})\)

### 5.2 Derived Theorems

This section includes some derived theorems based on the axioms, taken from the TLA* paper [8]. We mimic the proofs given there and avoid semantic reasoning whenever possible.

The \(\text{alw}\) theorem of [8] states that if \(F\) holds in all worlds then it always holds, i.e. \(F \models \Box F\). However, the derivation of this theorem (using the proof rules above) relies on access of the set of free variables (FV), which is not available in a shallow encoding.

However, we can prove a similar rule \(\text{alw2}\) using an additional hypothesis \(\sim F \land \text{Unchanged } v \rightarrow \Diamond F\).

**Theorem alw2:**
assumes \(h1: \vdash F\) and \(h2: \sim F \land \text{Unchanged } v \rightarrow \Diamond F\)
shows \(\vdash \Box F\)

**Proof**
- from \(h1\) have \(g2: \sim F \rightarrow \Diamond F\) by \((\text{rule nex})\)
  hence \(g3: \sim F \rightarrow \Diamond F\) by \(\text{auto}\)
  hence \(g4: \vdash \Box [\Diamond (F \rightarrow \Diamond F)] \cdot v \rightarrow F \rightarrow \Box F\) by \((\text{rule ax3})\)
  from \(h2\) have \(\vdash \Box [\Diamond (F \rightarrow \Diamond F)] \cdot v \rightarrow F \rightarrow \Box F\) by \((\text{rule sq})\)
  with \(g4[\text{unlifted}]\) have \(g5: F \rightarrow \Box F\) by \(\text{auto}\)
  with \(h1[\text{unlifted}]\) show \(\text{thesis}\) by \(\text{auto}\)

**Qed**

Similar theorem, assuming that \(F\) is stuttering invariant.

**Theorem alw3:**
assumes \(h1: \vdash F\) and \(h2: \text{stutinv } F\)
shows \(\vdash \Box F\)

**Proof**
- from \(h2\) have \(\sim F \land \text{Unchanged } id \rightarrow \Diamond F\) by \((\text{rule pre-id-unch})\)
  with \(h1\) show \(\text{thesis}\) by \((\text{rule alw2})\)

**Qed**

In a deep embedding, we could prove that all (proper) TLA* formulas are stuttering invariant and then get rid of the second hypothesis of rule \(\text{alw3}\).

In fact, the rule is even true for pre-formulas, as shown by the following rule, whose proof relies on semantical reasoning.

**Theorem alw:** assumes \(H1: \vdash F\) shows \(\vdash \Box F\)
using \(H1\) by \((\text{auto simp: always-def})\)
\textbf{theorem} \textit{alw-valid-iff-valid}: (\vdash \Box F) = (\vdash F)

\textbf{proof}
\begin{itemize}
    \item \textit{assume} \vdash \Box F
    \item \textit{from this ax1 show} \vdash F \textit{by (rule fmp)}
\end{itemize}
\textbf{qed (rule alw)}

[8] proves the following theorem using the deduction theorem of TLA*: (\vdash F \Longrightarrow \vdash G) \Longrightarrow \vdash [F \longrightarrow G], which can only be proved by induction on the formula structure, in a deep embedding.

\textbf{theorem} \textit{T1 [simp-unl]}: \vdash \Box \Box F = \Box F

\textbf{proof (auto simp: always-def suffix-plus)}
\begin{itemize}
    \item \textbf{fix} w n
    \item \textit{assume} \forall m k. (w \mid s (k+m)) \models F
    \item \textbf{hence} (w \mid s (n+0)) \models F \textit{by blast}
    \item \textbf{thus} (w \mid s n) \models F \textit{by simp}
\end{itemize}
\textbf{qed}

\textbf{theorem} \textit{T2 [simp-unl]}: \vdash \Box \Box [P] = \Box [P]

\textbf{proof –}
\begin{itemize}
    \item \textbf{have} 1: \vdash \Box [P] = [P] by \textit{force}
    \item \textit{hence} \vdash \Box [P] \Longrightarrow [P] \textit{by (rule sq)}
    \item \textbf{moreover}
    \item \textbf{have} \vdash [P] \Longrightarrow [P] \textit{by (rule ax3) (auto elim: \textit{unlift-rule})}
    \item \textbf{moreover}
    \item \textbf{ultimately show} \textbf{thesis by force}
\end{itemize}
\textbf{qed}

\textbf{theorem} \textit{T3 [simp-unl]}: \vdash [P] = [P]

\textbf{proof –}
\begin{itemize}
    \item \textbf{have} \vdash \Box [P] \Longrightarrow [P] \textit{by (auto simp: actrans-def)}
    \item \textit{hence} \vdash \Box [P] \textit{by (rule sq)}
    \item \textbf{with ax4 have} \vdash \Box [P] \textit{by force}
    \item \textbf{moreover}
    \item \textbf{ultimately show} \textbf{thesis by force}
\end{itemize}
\textbf{qed}

\textbf{theorem} \textit{M2}:
\begin{itemize}
    \item \textbf{assumes} h: \vdash \Box F \longrightarrow G
    \item \textbf{shows} \vdash [P] \longrightarrow [P]
\end{itemize}
\textbf{using sq OF h ax4 by force}

\textbf{theorem} \textit{N1}:
\begin{itemize}
    \item \textbf{assumes} h: \vdash F \longrightarrow G
    \item \textbf{shows} \vdash \Box [P] \longrightarrow [P]
\end{itemize}
\textbf{forces intro ax4 [unlift-rule]}
by (rule pmp[OF nex[OF h] pax2])

theorem T4: \( \vdash \Box[P]-v \rightarrow \Box[[P]-v]-w \)

proof –
  have \( \vdash \Box[\Box[P]-v] \rightarrow \Box[[\Box[P]-v]-w \) by (rule ax2)
  moreover
  from pax4 have \( \sim \Box[\Box[P]-v] \rightarrow [P]-v \) unfolding T2[int-rewrite] by force
  hence \( \vdash \Box[[\Box[P]-v]-w \rightarrow \Box[[P]-v]-w \) by (rule M2)
  ultimately show \( \sim \text{thesis} \) unfolding T2[int-rewrite] by (rule lift-imp-trans)
qed

theorem T5: \( \vdash \Box[[P]-w]-v \rightarrow \Box[[P]-v]-w \)

proof –
  have \( \vdash \Box[[P]-w]-v \rightarrow [[P]-v]-w \) by (auto simp: actrans-def)
  hence \( \vdash \Box[[P]-w]-v \rightarrow \Box[[P]-v]-w \) by (rule M2)
  with T4 show \( \sim \text{thesis} \) unfolding T2[int-rewrite] by (rule lift-imp-trans)
qed

theorem T6: \( \vdash \Box F \rightarrow \Box[\Box F]-v \)

proof –
  from ax1 have \( \sim \Box[\Box F] \rightarrow F \) by (rule nex)
  with pax2 have \( \sim \Box F \rightarrow F \) by force
  with pax3 have \( \sim F \rightarrow F \) by (rule pref-imp-trans)
  hence \( \vdash \Box[\Box F]-v \rightarrow \Box[\Box F]-v \) by (rule M2)
  with ax2 show \( \sim \text{thesis} \) by (rule lift-imp-trans)
qed

theorem T7:
  assumes h: \( \sim F \land \text{Unchanged} v \rightarrow \Box F \)
  shows \( \sim (F \land \Box F) = \Box F \)

proof –
  have \( \vdash \Box F \rightarrow F \rightarrow \Box F]-v \) by (rule sq) auto
  with ax4 have \( \vdash \Box[\Box F]-v \rightarrow \Box[(F \rightarrow \Box F)]-v \) by force
  with ax3[OF h, unfolded] have \( \vdash \Box[\Box F]-v \rightarrow (F \rightarrow \Box F) \) by force
  with pax5 have \( \sim F \land \Box F \rightarrow F \land F \) by force
  with ax1[of TEMP F,unlifted] pax3[of TEMP F,unlifted] show \( \sim \text{thesis} \) by force
qed

theorem T8: \( \sim (\Box F \land G) = (\Box F \land \Box G) \)

proof –
  have \( \sim (\Box F \land G) \rightarrow \Box F \) by (rule N1) auto
  moreover
  have \( \sim (F \land G) \rightarrow \Box G \) by (rule N1) auto
  moreover
  have \( \vdash F \rightarrow G \rightarrow F \land G \) by auto
  from nex[OF this] have \( \sim \Box F \rightarrow \Box G \rightarrow (F \land G) \)
    by (force intro: pax2[unlift-rule])
  ultimately show \( \sim \text{thesis} \) by force
qed
lemma T9: \( \neg \Box [A] \implies [A] \)
using pax4 by force

theorem H1:
assumes h1: \( \vdash \Box P \)
and h2: \( \vdash (P \rightarrow Q) \)
shows \( \vdash \Box Q \)
using assms ax4 [unlifted] by force

theorem H2: assumes h1: \( \vdash F \)
shows \( \vdash \Box F \)
using h1 by (blast dest: pre sq)

theorem H3:
assumes h1: \( \vdash (P \rightarrow Q) \)
and h2: \( \vdash (Q \rightarrow R) \)
shows \( \vdash (P \rightarrow R) \)
proof
have \( \neg \neg (P \rightarrow Q) \rightarrow (Q \rightarrow R) \rightarrow (P \rightarrow R) \)
by auto
hence \( \vdash \Box(P \rightarrow Q) \rightarrow (Q \rightarrow R) \rightarrow (P \rightarrow R) \)
by (rule sq)
with h1 have \( \vdash \Box(Q \rightarrow R) \rightarrow (P \rightarrow R) \)
by (rule H1)
with h2 show \(?thesis\)
by (rule H1)
qed

theorem H4: \( \vdash \Box[P] \rightarrow P \)
proof
have \( \neg v \neq \neg v \rightarrow ([P] \rightarrow P) \)
by (auto simp: unch-def actrans-def)
hence \( \vdash \Box(v \neq \neg v \rightarrow ([P] \rightarrow P)) \)
by (rule sq)
with ax5 show \(?thesis\)
by (rule H1)
qed

5.3 Some other useful derived theorems

theorem P1: \( \neg \Box F \rightarrow \Diamond F \)
proof
have \( \neg \Diamond F \rightarrow \Diamond F \)
by (rule N1[OF ax1])
with pax3 show \(?thesis\)
by (rule pref-imp-trans)
qed

theorem P2: \( \neg \Box F \rightarrow F \land \Diamond F \)
using ax1[of F] P1[of F] by force

theorem P4: \( \vdash \Box[F] \rightarrow \Box[F] \)
proof
have \( \vdash \Box[F] \rightarrow \Box[F] \)
by (rule M2[OF ax1])
with ax2 show \(?thesis\)
by (rule lift-imp-trans)
qed
theorem \( P5: \vdash \Box[P] \rightarrow \Box[\Box[P] \rightarrow w] \)
proof
have \( \vdash \Box[\Box[P] \rightarrow w] \rightarrow \Box[\Box[P] \rightarrow w] \) by (rule \( P4 \))
thus \( \text{thesis} \) by (unfold \( T2[\text{int-rewrite}] \))
qed 

theorem \( M0: \vdash \Box F \rightarrow \Box[F \rightarrow \Diamond F] \)
proof
from \( P1 \) have \( \vdash \neg \Box F \rightarrow F \rightarrow \Diamond F \) by force
hence \( \vdash \Box[\Box F] \rightarrow \Box[F \rightarrow \Diamond F] \) by (rule \( M2 \))
with \( ax2 \) show \( \text{thesis} \) by (rule \( \text{lift-imp-trans} \))
qed 

theorem \( M1: \vdash \Box F \rightarrow \Box[F \land \Diamond F] \)
proof
have \( \vdash \neg \Box F \rightarrow F \land \Diamond F \) by (rule \( P2 \))
hence \( \vdash \Box[\Box F] \rightarrow \Box[F \land \Diamond F] \) by (rule \( M2 \))
with \( ax2 \) show \( \text{thesis} \) by (rule \( \text{lift-imp-trans} \))
qed 

theorem \( M3: \text{assumes} \ h; \vdash F \rightarrow \Box[\Diamond F] \)
using \( \text{alw}[OF \ h] \) \( T6 \) by (rule \( \text{fmp} \))

lemma \( M4: \vdash \Box[\Diamond(F \land G)] = (\Diamond F \land \Diamond G) \)
by (rule \( \text{sq}[OF \ T8] \))

theorem \( M5: \vdash \Box[\Box[P] \rightarrow w] \rightarrow \Box[\Box[P] \rightarrow w] \)
proof (rule \( \text{sq} \))
show \( \vdash \Box[P] \rightarrow \Diamond[P] \) by (auto simp: \( pax4[\text{unlifted}] \))
qed 

theorem \( M6: \vdash \Box[F \land G] \rightarrow \Box[F] \land \Box[G] \)
proof
have \( \vdash \Box[F \land G] \rightarrow \Box[F] \land \Box[G] \) by (rule \( M2 \)) auto
moreover
have \( \vdash \Box[F \land G] \rightarrow \Box[G] \) by (rule \( M2 \)) auto
ultimately show \( \text{thesis} \) by force
qed 

theorem \( M7: \vdash \Box[F \land G] \rightarrow \Box[F] \land \Box[G] \)
proof
have \( \vdash \Box[F] \rightarrow G \rightarrow F \land G \) by auto
hence \( \vdash \Box[F] \rightarrow \Box[G \rightarrow F \land G] \) by (rule \( M2 \))
with \( ax4 \) show \( \text{thesis} \) by force
qed 

theorem \( M8: \vdash \Box[F \land G] = (\Box[F] \land \Box[G] \land \Diamond) \)
by (rule \( \text{int-iffI}[OF \ M6 \ M7] \))
theorem M9: \[ \neg \Box F \rightarrow F \land \Box F \]
using \[\text{pre}[\text{OF ax1}\{\text{of } F\}] \text{ pax3}\{\text{of } F}\] by force

theorem M10:
assumes \( h : \neg F \land \text{Unchanged } v \rightarrow \Diamond F \)
shows \( \neg F \land \Diamond F \rightarrow \Box F \)
using \( T7[\text{OF } h]\) by auto

theorem M11:
assumes \( h : \neg [A]-f \rightarrow [B]-g \)
shows \( \vdash \Box [A]-f \rightarrow [B]-g \)
proof –
from \( h \) have \( \vdash [A]-f \rightarrow [B]-g \) by (rule M2)
with \( T4 \) show \( \text{thesis by force} \)
qed

theorem M12: \( \vdash ([A]-f \land [B]-g) = \Box [A]-f \land [B]-g \rightarrow (f,g) \)
proof –
have \( \vdash [A]-f \land [B]-g \rightarrow [A]-f \)
by (auto simp: \( M8[\text{int-rewrite}] \) elim: \( T4[\text{unlift-rule}] \))
moreover
have \( \neg [A]-f \land [B]-g \rightarrow [A]-f \)
by (auto simp: \( \text{actrans-def unch-def all-before-eq all-after-eq} \))
hence \( \vdash [A]-f \land [B]-g \rightarrow [A]-f \) by (rule M11)
moreover
have \( \neg [A]-f \land [B]-g \rightarrow [B]-g \)
by (auto simp: \( \text{actrans-def unch-def all-before-eq all-after-eq} \))
hence \( \vdash [A]-f \land [B]-g \rightarrow [B]-g \)
by (rule M11)
ultimately show \( \text{thesis by force} \)
qed

We now derive Lamport’s 6 simple temporal logic rules (STL1)-(STL6) [5].
Firstly, STL1 is the same as \( \vdash \neg F \equiv \vdash \Box \neg F \) derived above.

lemmas STL1 = alw

STL2 and STL3 have also already been derived.

lemmas STL2 = ax1

lemmas STL3 = T1

As with the derivation of \( \vdash \neg F \equiv \vdash \Box \neg F \), a purely syntactic derivation of (STL4) relies on an additional argument – either using Unchanged or STUTINV.

theorem STL4-2:
assumes \( h1 : \vdash F \rightarrow G \text{ and } h2 : \neg G \land \text{Unchanged } v \rightarrow \Diamond G \)
shows \( \vdash \Box F \rightarrow \Box G \)
proof –
from ax1[of F] h1 have ⊢ □F → G by (rule lift-imp-trans)
moreover
from h1 have | φ F → ○G by (rule N1)
hence | φ F → ○G by auto
hence ⊢ [□F]-v → □G → ○G]-v by (rule M2)
with T6 have ⊢ □F → □G → □G by (rule lift-imp-trans)
moreover
from h2 have ⊢ □G → □G] by (rule ax3)
ultimately
show ?thesis by force
qed

lemma STL4-3:
assumes h1: ⊢ F → G and h2: STUTINV G
shows ⊢ □F → □G
using h1 h2[THEN pre-id-unch] by (rule STL4-2)

Of course, the original rule can be derived semantically

lemma STL4: assumes h: ⊢ F → G shows ⊢ □F → □G
using h by (force simp: always-def)

Dual rule for ◇

lemma STL4-eve: assumes h: ⊢ F → G shows ⊢ ◇F → ◇G
using h by (force simp: eventually-defs)

Similarly, a purely syntactic derivation of (STL5) requires extra hypotheses.

theorem STL5-2:
assumes h1: | φ F ∧ Unchanged f → φ F
and h2: | φ G ∧ Unchanged g → φ G
shows ⊢ [□(F ∧ G) = (□F ∧ □G)]
proof (rule int-iffI)
have ⊢ F ∧ G → F by auto
from this h1 have ⊢ [□(F ∧ G) → □F by (rule STL4-2)]
moreover
have ⊢ F ∧ G → G by auto
from this h2 have ⊢ □(F ∧ G) → □G by (rule STL4-2)
ultimately show ⊢ □(F ∧ G) → □F ∧ □G by force
next
have | φ Unchanged (f,g) → Unchanged f ∧ Unchanged g by (auto simp: unch-defs)
with h1[unlifted] h2[unlifted] T8[of F G, unlifted]
have h3: | φ (F ∧ G) ∧ Unchanged (f,g) → φ(F ∧ G) by force
from ax1[of F] ax1[of G] have ⊢ □F ∧ □G → F ∧ G by force
moreover
from ax2[of F] ax2[of G] have ⊢ □F ∧ □G → □[□F]-φ(f,g) ∧ □[□G]-φ(f,g)
by force
with M8 have ⊢ □F ∧ □G → □[□F]-φ(f,g) by force
moreover
from P1[of F] P1[of G] have | φ F ∧ φ G → F ∧ G → φ(F ∧ G)
unfolding $T8[\text{int-rewrite}]$ by force

hence $\vdash \square F \land \square G \neg(f,g) \rightarrow \square(F \land G) \rightarrow \bigcirc(F \land G)\neg(f,g)$ by (rule $M2$

from this ax3[OF h3] have $\vdash \square F \land \square G \rightarrow F \rightarrow \square(F \land G)$

by (rule lift-imp-trans)

ultimately show $\vdash \square F \land \square G \rightarrow \square(F \land G)$ by force

qed

**Theorem STL5-21:**

assumes $h1$: stutinv $F$ and $h2$: stutinv $G$

shows $\vdash \square (F \land G) = (\square F \land \square G)$

using $h1[\text{THEN pre-id-unch}]$ $h2[\text{THEN pre-id-unch}]$ by (rule $STL5-2$

We also derive STL5 semantically.

**Lemma STL5:**

$\vdash \square (F \land G) = (\square F \land \square G)$

by (auto simp: always-def)

Elimination rule corresponding to $STL5$ in unlifted form.

**Lemma box-conjE:**

assumes $s \models \square F$ and $s \models \square G$ and $s \models \square(F \land G) \Rightarrow P$

shows $P$

using assms by (auto simp: STL5[unlifted])

**Lemma box-thin:**

assumes $h1$: $s \models \square F$ and $h2$: PROP $W$

shows PROP $W$

using $h2$. 

Finally, we derive STL6 (only semantically)

**Lemma STL6:**

$\vdash \Box (F \land G) = (\Box F \land \Box G)$

**Proof**

fix $w$

assume $a1$: $w \models \Box F$ and $a2$: $w \models \Box G$

from $a1$ obtain $nf$ where $nf$: $(w \mid_s nf) \models \Box F$ by (auto simp: eventually-defs)

from $a2$ obtain $ng$ where $ng$: $(w \mid_s ng) \models \Box G$ by (auto simp: eventually-defs)

let $?n = \max nf ng$

have $nf \leq ?n$ by simp

from this $nf$ have $(w \mid_s ?n) \models \Box F$ by (rule linalw)

moreover

have $ng \leq ?n$ by simp

from this $ng$ have $(w \mid_s ?n) \models \Box G$ by (rule linalw)

ultimately

have $(w \mid_s ?n) \models \Box(F \land G)$ by (rule box-conjE)

thus $w \models \Box(F \land G)$ by (auto simp: eventually-defs)

next

fix $w$

assume $h$: $w \models \Box(F \land G)$

have $\vdash F \land G \rightarrow F$ by auto

hence $\vdash \Box(F \land G) \rightarrow \Box F$ by (rule $STL4$-eve[OF $STL4$])

with $h$ show $w \models \Box F$ by auto

50
next
fix \( w \)
assume \( h: \models \lozenge \Box (F \land G) \)
have \( \vdash F \land G \to G \) by auto
hence \( \vdash \lozenge (F \land G) \to \lozenge \Box G \) by (rule STL4-eve[OF STL4])
with \( h \) show \( w \models \lozenge \Box G \) by auto
qed

lemma MM0: \( \vdash \Box (F \to G) \to \Box F \to \Box G \)
proof –
have \( \vdash \Box (F \land (F \to G)) \to \Box G \) by (rule STL4) auto
thus ?thesis by (auto simp: STL5[int-rewrite])
qed

lemma MM1: assumes \( h: \vdash F = G \) shows \( \vdash \Box F = \Box G \)
by (auto simp: h[int-rewrite])

theorem MM2: \( \vdash \Box A \land \Box (B \to C) \to \Box (A \land B \to C) \)
proof –
have \( \vdash \Box (A \land (B \to C)) \to \Box (A \land B \to C) \) by (rule STL4) auto
thus ?thesis by (auto simp: STL5[int-rewrite])
qed

theorem MM3: \( \vdash \Box \neg A \to \Box (A \land B \to C) \)
by (rule STL4) auto

theorem MM4[simp-unl]: \( \vdash \Box \# F = \# F \)
proof (cases \( F \))
assume \( F \)
hence \( 1: \vdash \# F \) by auto
hence \( \vdash \Box \# F \) by (rule alw)
with \( 1 \) show ?thesis by force
next
assume \( \neg F \)
hence \( 1: \vdash \neg \# F \) by auto
from \( ax1 \) have \( \vdash \neg \# F \to \neg \Box \# F \) by (rule lift-imp-neg)
with \( 1 \) show ?thesis by force
qed

theorem MM4b[simp-unl]: \( \vdash \Box \neg \# F = \neg \# F \)
proof –
have \( \vdash \neg \# F = \# (\neg F) \) by auto
hence \( \vdash \Box \neg \# F = \Box \# (\neg F) \) by (rule MM1)
thus ?thesis by auto
qed

theorem MM5: \( \vdash \Box F \lor \Box G \to \Box (F \lor G) \)
proof –
have \( \vdash \Box F \to \Box (F \lor G) \) by (rule STL4) auto
moreover
have \( \vdash \Box G \rightarrow \Box (F \lor G) \) by (rule STL4) auto
ultimately show \( ? \text{thesis by force} \)
qed

theorem MM6: \( \vdash \Box F \lor \Box G \rightarrow \Box (\Box F \lor \Box G) \)
proof
have \( \vdash \Box \Box F \lor \Box \Box G \rightarrow \Box (\Box F \lor \Box G) \) by (rule MM5)
thus \( ? \text{thesis by simp} \)
qed

lemma MM10:
assumes \( \vdash \neg F = G \) shows \( \vdash \Box[v] F = \Box[v] G \)
by (auto simp: \( h[v] \text{[unlifted]} \))

lemma MM9:
assumes \( \vdash F = G \) shows \( \vdash \Box[v] F = \Box[v] G \)
by (rule MM10[OF pre[OF \( h \text{[OF h]} \)])

theorem MM11: \( \vdash \Box [\neg (P \land Q)] -v \rightarrow \Box [P] -v \rightarrow \Box [P \land \neg Q] -v \)
proof
have \( \vdash \Box [\neg (P \land Q)] -v \rightarrow \Box [P \rightarrow P \land \neg Q] -v \) by (rule M2) auto
from this ax4 show \( ? \text{thesis by (rule lift-imp-trans)} \)
qed

theorem MM12[simp-und]: \( \vdash \Box[v] [\Box[v] P] -v = \Box[v] P \)
proof (rule int-iffI)
have \( \vdash \Box[v] [\Box[v] P] -v = \Box[v] P \) by (auto simp: pax4[unlifted])
hence \( \vdash \Box[v] [\Box[v] P] -v = \Box[v] P \) by (rule M2)
thus \( \vdash \Box[v] [\Box[v] P] -v = \Box[v] P \) by (unfold T3[unlifted])
next
have \( \vdash \Box[v] [\Box[v] P] -v = \Box[v] [\Box[v] P] -v \) by (rule ax2)
thus \( \vdash \Box[v] P -v = \Box[v] [\Box[v] P] -v \) by auto
qed

5.4 Theorems about the eventually operator
— rules to push negation inside modal operators, sometimes useful for rewriting

theorem dualization:
\( \vdash \neg \Box F = \Diamond \neg F \)
\( \vdash \neg \Diamond F = \Box \neg F \)
\( \vdash \neg \Box [A] -v = \Diamond (\neg A) -v \)
\( \vdash \neg \Diamond (A) -v = \Box (\neg A) -v \)

unfolding eventually-def angle-action-def by simp-all

lemmas dualization-rew = dualization[int-rewrite]
lemmas dualization-und = dualization[unlifted]

theorem E1: \( \vdash \Diamond (F \lor G) = (\Diamond F \lor \Diamond G) \)
proof
  have ⊢ □¬(F ∨ G) = □(¬F ∧ ¬G) by (rule MM1) auto
  thus thesis unfolding eventually-def STL5[int-rewrite] by force
qed

theorem E3: ⊢ F −→ ♦F
  unfolding eventually-def by (force dest: ax1[unlift-rule])

theorem E4: ⊢ □F −→ ♦F
  by (rule lift-imp-trans[OF ax1 E3])

theorem E5: ⊢ □F −→ ♦□F
  proof
    have ⊢ □□F −→ □♦F by (rule STL4[OF E4])
    thus thesis by simp
  qed

theorem E6: ⊢ □F −→ ♦□F
  using E4[of TEMP □F] by simp

theorem E7:
  assumes h: ▼¬F ∧ Unchanged v −→ □¬F
  shows ▼ ♦F −→ F ∨ □♦F
  proof
    from h have ▼¬F ∧ □□¬F −→ □¬F by (rule M10)
    thus thesis by (auto simp: eventually-def)
  qed

theorem E8: ⊢ ♦(F −→ G) −→ □F −→ ♦G
  proof
    have □(F ∧ ¬G) −→ □¬(F −→ G) by (rule STL4) auto
    thus thesis unfolding eventually-def STL5[int-rewrite] by auto
  qed

theorem E9: ⊢ □(F −→ G) −→ ♦F −→ ♦G
  proof
    have □(F −→ G) −→ □(¬G −→ ¬F) by (rule STL4) auto
    with MM0[of TEMP ¬G TEMP ¬F] show thesis unfolding eventually-def
    by force
  qed

theorem E10[simp-unl]: ⊢ ♦♦F = ♦F
  by (simp add: eventually-def)

theorem E22:
  assumes h: ⊢ F = G
  shows ⊢ ♦F = ♦G
  by (auto simp: h[int-rewrite])
### Theorem E15 (simp-unl): \( \Diamond \# F \Rightarrow \# F \)
by (simp add: eventually-def)

### Theorem E15b (simp-unl): \( \Diamond \neg \# F \Rightarrow \neg \# F \)
by (simp add: eventually-def)

### Theorem E16:
\( \Diamond \Box F \Rightarrow \Diamond F \)
by (rule STL4-eve[OF ax1])

An action version of STL6

#### Lemma STL6-act:
\( \Diamond \Box (\Box F \wedge \Box G) = (\Diamond \Box F \wedge \Diamond \Box G) \)

**Proof** –
- have \( \Diamond (\Box (\Box F \wedge \Box G)) = \Box (\Box (\Box F \wedge \Box G)) \)
  by (rule E22[OF STL5])
- thus \( \text{thesis} \)
  by (auto simp: STL6[int-rewrite])

**Qed**

#### Lemma SE1:
\( \Box F \wedge \Diamond G \Rightarrow \Diamond (\Box F \wedge G) \)

**Proof** –
- have \( \Diamond (\Box F \wedge G) \Rightarrow \Box (\Box F \wedge \neg G) \)
  by (rule STL4) auto
- with MM0 show \( \text{thesis} \)
  by (force simp: eventually-def)

**Qed**

#### Lemma SE2:
\( \Box F \wedge \Diamond G \Rightarrow \Diamond (F \wedge G) \)

**Proof** –
- have \( \Diamond (\Box F \wedge G) \Rightarrow \Box (F \Rightarrow \neg G) \)
  by (auto elim: ax1[unlift-rule])
- hence \( \Diamond (\Box F \wedge G) \Rightarrow \Diamond (F \wedge G) \)
  by (rule STL4-eve)
- with SE1 show \( \text{thesis} \)
  by (rule lift-imp-trans)

**Qed**

#### Lemma SE3:
\( \Box F \wedge \Diamond G \Rightarrow \Diamond (G \wedge F) \)

**Proof** –
- have \( \Diamond (F \wedge G) \Rightarrow \Diamond (G \Rightarrow F) \)
  by (rule STL4-eve) auto
- with SE2 show \( \text{thesis} \)
  by (rule lift-imp-trans)

**Qed**

#### Lemma SE4:
assumes \( h1: s \models \Box F \) and \( h2: s \models \Diamond G \) and \( h3: \vdash \Box F \wedge G \Rightarrow H \)
shows \( s \models \Diamond H \)
using \( h1 \) \( h2 \) \( h3 \)[THEN STL4-eve] SE1 by force

### Theorem E17:
\( \Diamond \Box \Diamond F \Rightarrow \Box \Diamond F \)
by (rule STL4[OF STL4-eve[OF ax1]])

### Theorem E18:
\( \Diamond \Box \Diamond F \Rightarrow \Diamond \Box F \)
by (rule ax1)

### Theorem E19:
\( \Diamond \Diamond F \Rightarrow \Box \Diamond \Diamond F \)

**Proof** –

54
have \( \vdash (\Box F \land \neg \Box F) = \# False \) by auto

hence \( \vdash \Diamond (\Box F \land \neg \Box F) = \Diamond \# False \) by (rule E22 [OF MM1])

thus ?thesis unfolding STL6 [int-rewrite] by (auto simp: eventually-def)

qed

theorem E20: \( \vdash \Diamond \Box F \rightarrow \Box \Diamond F \)
by (rule lift-imp-trans [OF E19 E17])

theorem E21 [simp-und]: \( \vdash \Box \Diamond F = \Diamond \Box F \)

using E21 unfolding eventually-def by force

lemma E28: \( \vdash \Diamond \Box F \land \Box \Diamond G \rightarrow \Box (F \land G) \)

proof -
  have \( \vdash \Diamond (\Box F \land \Diamond G) \rightarrow \Box (F \land G) \) by (rule STL4-eve [OF STL4 [OF SE2]])
  thus ?thesis by (simp add: STL6 [int-rewrite])

qed

lemma E23: \( \neg \Diamond F \rightarrow \Diamond F \)

using P1 by (force simp: eventually-def)

lemma E24: \( \vdash \Diamond Q \rightarrow \Box [\Diamond Q] \cdot \cdot \cdot \)

by (rule lift-imp-trans [OF E20 P4])

lemma E25: \( \vdash \Diamond (A \cdot \cdot \cdot) \rightarrow \Diamond A \)

using P4 by (force simp: eventually-def angle-action-def)

lemma E26: \( \vdash \Box (A \cdot \cdot \cdot) \rightarrow \Box A \)

by (rule STL4 [OF E25])

lemma allBox: \( (s \models \Box (\forall x. F x)) = (\forall x. s \models \Box (F x)) \)
unfolding allT [unlifted] ..

lemma E29: \( \neg \Diamond F \rightarrow \Diamond F \)

unfolding eventually-def using pax3 by force

lemma E30:
  assumes h1: \( \vdash F \rightarrow \Box F \) and h2: \( \vdash \Diamond F \)
  shows \( \vdash \Diamond \Box F \)
  using h2 h1 [THEN STL4-eve] by (rule fmp)

lemma E31: \( \vdash \Diamond F \rightarrow \Box (F \land \Diamond F) \land \Diamond F \rightarrow \Diamond F \)

proof -
  have \( \vdash \Diamond (F \rightarrow \Box F) \land \Diamond F \rightarrow \Diamond (\Box F \land F) \) by (rule SE1)
  moreover
  have \( \vdash \Diamond F \land F \rightarrow \Box F \) using ax1 [of TEMP F \rightarrow \Box F] by auto

55
hence ⊢ ◻(◻(F → ◻F) ∧ F) → ◻◻F by (rule STL4-eve)
ultimately show thesis by (rule lift-imp-trans)
qed

lemma allActBox: (s |= ◻[(∀ x. F x)]-v) = (∀ x. s |= ◻[(F x)]-v)
unfolding allActT[unlifted] ..

theorem exEE: ⊢ (∃ x. ◻(F x)) = ◻((∃ x. F x))
proof -
  have ⊢ ¬(∃ x. ◻(F x)) = ¬◻((∃ x. F x))
  by (auto simp: eventually-def Not-Rex[int-rewrite] allBox)
thus thesis by force
qed

theorem exActE: ⊢ (∃ x. ◻⟨F x⟩)-v = ◻⟨(∃ x. F x)⟩-v
proof -
  have ⊢ ¬(∃ x. ◻⟨F x⟩)-v = ¬◻⟨(∃ x. F x)⟩-v
  by (auto simp: angle-action-def Not-Rex[int-rewrite] allActBox)
thus thesis by force
qed

5.5 Theorems about the leadsto operator

theorem LT1: ⊢ F ⇝ F
unfolding leadsto-def by (rule alw[OF E3])

theorem LT2: assumes h: ⊢ F ⟹ G shows ⊢ F ⟹ ◻G
by (rule lift-imp-trans[OF h E3])

theorem LT3: assumes h: ⊢ F ⟹ G shows ⊢ F ⟳ G
unfolding leadsto-def by (rule alw[OF LT2[OF h]])

theorem LT4: ⊢ F ⟳ (F ⇝ G) ⟹ ◻G
unfolding leadsto-def using ax1[of TEMP F ⟹ ◻G] by auto

theorem LT5: ⊢ ◻(F ⟳ ◻G) ⟹ ◻F ⟹ ◻G
using E9[of F TEMP ◻G] by simp

theorem LT6: ⊢ ◻F ⟳ (F ⇝ G) ⟹ ◻G
unfolding leadsto-def using LT5[of F G] by auto

theorem LT9[simp-und]: ⊢ ◻(F ⟳ G) = (F ⇝ G)
by (auto simp: leadsto-def)

theorem LT7: ⊢ ◻(F ⟳ G) ⟹ ◻F ⟳ ◻G
proof -
  have ⊢ ◻(F ⟳ G) ⟹ ◻(F ⟳ G) ⟹ ◻G by (rule STL4[OF LT6])
  from lift-imp-trans[OF this MM0] show thesis by simp
qed

56
theorem LT8: \( \vdash \Box \Diamond G \rightarrow (F \rightarrow G) \)
Theorem LT8 has been unfolded leadsto-def by (rule STL4) auto

theorem LT13: \( \vdash (F \rightarrow G) \rightarrow (G \rightarrow H) \rightarrow (F \rightarrow H) \)
Proof -
  have \( \vdash \Diamond G \rightarrow (G \rightarrow H) \rightarrow \Diamond H \) by (rule LT6)
  hence \( \vdash \Box (F \rightarrow \Diamond G) \rightarrow \Box ((G \rightarrow H) \rightarrow (F \rightarrow \Diamond H)) \) by (intro STL4) auto
  from lift-imp-trans[OF this MM0] show \(?thesis\) by (simp add: leadsto-def)
qed

theorem LT11: \( \vdash (F \rightarrow G) \rightarrow (F \rightarrow (G \vee H)) \)
Proof -
  have \( \vdash G \rightarrow (G \vee H) \) by (rule LT3) auto
  with LT13[of F G TEMP (G \vee H)] show \(?thesis\) by force
qed

theorem LT12: \( \vdash (F \rightarrow H) \rightarrow (F \rightarrow (G \vee H)) \)
Proof -
  have \( \vdash H \rightarrow (G \vee H) \) by (rule LT3) auto
  with LT13[of F H TEMP (G \vee H)] show \(?thesis\) by force
qed

theorem LT14: \( \vdash ((F \vee G) \rightarrow H) \rightarrow (F \rightarrow H) \)
Theorem LT14 has been unfolded leadsto-def by (rule STL4) auto

theorem LT15: \( \vdash ((F \vee G) \rightarrow H) \rightarrow (G \rightarrow H) \)
Theorem LT15 has been unfolded leadsto-def by (rule STL4) auto

theorem LT16: \( \vdash (F \rightarrow H) \rightarrow (G \rightarrow H) \rightarrow ((F \vee G) \rightarrow H) \)
Proof -
  have \( \vdash \Box (F \rightarrow \Diamond H) \rightarrow \Box ((G \rightarrow \Diamond H) \rightarrow (F \vee G \rightarrow \Diamond H)) \) by (rule STL4) auto
  from lift-imp-trans[OF this MM0] show \(?thesis\) by (unfold leadsto-def)
qed

theorem LT17: \( \vdash ((F \vee G) \rightarrow H) = ((F \rightarrow H) \land (G \rightarrow H)) \)
  by (auto elim: LT14[unlift-rule] LT15[unlift-rule]
  LT16[unlift-rule])

theorem LT10:
  assumes \( h: \vdash (F \land \neg G) \rightarrow G \)
  shows \( \vdash F \rightarrow G \)
Proof -
  from \( h \) have \( ((F \land \neg G) \vee G) \rightarrow G \)
    by (auto simp: LT17[int-rewrite] LT1[int-rewrite])
  moreover
  have \( \vdash F \rightarrow ((F \land \neg G) \vee G) \) by (rule LT3, auto)
ultimately
show ?thesis by (force elim: LT13[unlift-rule])
qed

theorem LT18: \( \vdash (A \rightarrow (B \lor C)) \rightarrow (B \rightarrow D) \rightarrow (C \rightarrow D) \rightarrow (A \rightarrow D) \)
proof –
  have \( \vdash (B \rightarrow D) \rightarrow (C \rightarrow D) \rightarrow ((B \lor C) \rightarrow D) \) by (rule LT16)
  thus ?thesis by (force elim: LT13[unlift-rule])
qed

theorem LT19: \( \vdash (A \rightarrow (D \lor B)) \rightarrow (B \rightarrow D) \rightarrow (A \rightarrow D) \)
using LT18[of A D B D] LT1[of D] by force

theorem LT20: \( \vdash (A \rightarrow (B \lor D)) \rightarrow (B \rightarrow D) \rightarrow (A \rightarrow D) \)
using LT18[of A B D D] LT1[of D] by force

lemma LT23: \( \neg ((\exists x. F x) \rightarrow G) = (\forall x. (F x \rightarrow \neg G)) \)
by (auto dest: E23[of Q])

theorem LT24: \( \vdash \Box I \rightarrow ((P \land I) \rightarrow Q) \rightarrow P \rightarrow Q \)
proof –
  have \( \vdash \Box (P \land I \rightarrow Q) \rightarrow (P \rightarrow Q) \) by (rule STL4) auto
  from lift-imp-trans[OF this MM0] show ?thesis by (force elim: LT20[unlift-rule])
qed

lemma LT25: \( \vdash (F \rightarrow \# \false) = \neg \neg F \)
unfolding leadsto-def proof (rule MM1)
  show \( \vdash (F \rightarrow \# \false) = \neg F \) by simp
qed

lemma LT28:
  assumes h: \( \neg (P \rightarrow \circ Q) \)
  shows \( \neg (P \rightarrow \circ Q) \)
  using h E23[of Q] by force

lemma LT29:
  assumes h1: \( \neg P \rightarrow \circ P \lor \circ Q \) and h2: \( \neg P \land \text{Unchanged } v \rightarrow \circ P \)
shows $\vdash P \rightarrow \Box P \lor \Diamond Q$

proof
- from $h_1[\text{THEN LT28}]$ have $\neg \Box \neg Q \rightarrow (P \rightarrow \Box P)$ unfolding eventually-def
  by auto
  hence $\vdash \Box \Box \neg Q \rightarrow \Box P \rightarrow \Box P \rightarrow \Diamond Q$ by (rule M2)
  moreover
  have $\vdash \neg \Diamond Q \rightarrow \Box \Box \neg Q \rightarrow \Box P \rightarrow \Box P \rightarrow \Diamond Q$ unfolding dualization-rew by (rule ax2)
  moreover
  note $\text{ax3}[\text{OF } h_2]$
  ultimately
  show $\neg \Diamond Q \rightarrow \Box \Box \neg Q \rightarrow \Box \Box \neg Q \rightarrow \Box P \rightarrow \Box P \rightarrow \Diamond Q$ by (rule sq)
  hence $\vdash \Box \Box \neg Q \rightarrow \Box P \rightarrow \Box P \rightarrow \Diamond Q$ by (rule lift-imp-trans)
  moreover
  note $\text{ax3}[\text{OF } h_2]$
  ultimately
  show $\Diamond Q \rightarrow \Box \Box \neg Q \rightarrow \Box \Box \neg Q \rightarrow \Box P \rightarrow \Box P \rightarrow \Diamond Q$ by (rule sq)
  hence $\vdash \Diamond Q \rightarrow \Box \Box \neg Q \rightarrow \Box \Box \neg Q \rightarrow \Box P \rightarrow \Box P \rightarrow \Diamond Q$ by (rule lift-imp-trans)
  moreover
  note $\text{ax3}[\text{OF } h_2]$
  ultimately
  show $\Diamond Q \rightarrow \Box \Box \neg Q \rightarrow \Box \Box \neg Q \rightarrow \Box P \rightarrow \Box P \rightarrow \Diamond Q$ by (rule sq)
 qed

lemma $\text{LT30}$:
- assumes $h_1: \neg P \land N \rightarrow \Box P \lor \Box Q$
- shows $\neg N \rightarrow (P \rightarrow \Box P) \lor \Box Q$
- using $h \ E23$ by force

lemma $\text{LT31}$:
- assumes $h_1: \neg P \land N \rightarrow \Box P \lor \Box Q$ and $h_2: \neg P \land \text{Unchanged } v \rightarrow \Box P$
- shows $\Box N \rightarrow P \rightarrow \Box P \lor \Box Q$
- proof
  - from $h_1[\text{THEN LT30}]$ have $\neg N \rightarrow \Box \neg Q \rightarrow P \rightarrow \Box P \lor \Box Q$ unfolding eventually-def
    by auto
    hence $\vdash \Box N \rightarrow \Box \neg Q \rightarrow P \rightarrow \Box P \lor \Box Q$ by (rule sq)
    hence $\vdash \Box N \rightarrow \Box \Box \neg Q \rightarrow \Box P \rightarrow \Box P \lor \Box Q$
      by (force intro: $\text{ax4}[	ext{unlift-rule}]$)
    with $P_4$ have $\vdash \Box N \rightarrow \Box \Box \neg Q \rightarrow \Box P \rightarrow \Box P \lor \Box Q$ by (rule lift-imp-trans)
    moreover
    have $\vdash \neg \Diamond Q \rightarrow \Box \Box \neg Q \rightarrow \Box P \rightarrow \Box P \lor \Box Q$ unfolding dualization-rew by (rule ax2)
    moreover
    note $\text{ax3}[\text{OF } h_2]$
    ultimately
    show $\neg \Diamond Q \rightarrow \Box \Box \neg Q \rightarrow \Box \Box \neg Q \rightarrow \Box P \rightarrow \Box P \rightarrow \Diamond Q$ by (rule sq)
 qed

lemma $\text{LT33}$: $\vdash ((\# P \land F) \Rightarrow G) = (\# P \rightarrow (F \Rightarrow G))$
  by (cases $P$, auto simp: leadsto-def)

lemma $\text{AA1}$: $\vdash \Box [\# \text{False}] \rightarrow \neg \Diamond (Q) \rightarrow$
  unfolding dualization-rew by (rule M2) auto

lemma $\text{AA2}$: $\vdash \Box [P] \land \Diamond (Q) \rightarrow \Diamond (P \land Q) \rightarrow$
  proof
  - have $\vdash \Box [P] \rightarrow (P \land Q) \rightarrow \neg Q \rightarrow \Diamond (P \land Q) \rightarrow \Diamond (P \land Q) \rightarrow$
    unfolding dualization-rew by (rule sq) (auto simp: actrans-def)
    hence $\vdash \Box [P] \rightarrow \Box [(P \land Q)] \rightarrow \Box \Box [\neg Q] \rightarrow$
      by (force intro: $\text{ax4}[	ext{unlift-rule}]$)
    thus $\neg \Diamond Q \rightarrow \Box \Box \neg Q \rightarrow \Box \Box \neg Q \rightarrow \Box P \rightarrow \Box P \rightarrow \Diamond Q$ by (rule sq)
 qed
lemma $AA3$: $\vdash \square P \land \square[P \rightarrow Q] - v \land \Diamond(A) - v \rightarrow \Diamond Q$

proof

- have $\vdash \square P \land \square[P \rightarrow Q] - v \rightarrow \square[P \land (P \rightarrow Q)] - v$
  
  by (auto dest: $P4$ [unlift-rule] simp: $M8$ [int-rewrite])

moreover

- have $\vdash \square[P \land (P \rightarrow Q)] - v \rightarrow \square[Q] - v$ by (rule $M2$) auto

ultimately have $\vdash \square[P \land (P \rightarrow Q)] - v \rightarrow \square[Q] - v$ by (rule lift-imp-trans)

moreover

- have $\vdash \Diamond(Q \land A) \rightarrow \Diamond Q$ by (rule STL4-eve) auto

hence $\vdash \Diamond(Q \land A) - v \rightarrow \Diamond Q$ by (force dest: $E25$ [unlift-rule])

with $AA2$ have $\vdash \square[Q] - v \land \Diamond(A) - v \rightarrow \Diamond Q$ by (rule lift-imp-trans)

ultimately show $?thesis$ by force

qed

lemma $AA4$: $\vdash \Diamond\langle A\rangle - w \rightarrow \Diamond\langle A\rangle - v$

unfolding angle-action-def angle-actrans-def using $T5$ by force

lemma $AA7$: assumes $h: \sim F \rightarrow G$ shows $\vdash \Diamond(F) - v \rightarrow \Diamond(G) - v$

proof

- from $h$ have $\vdash \square[\neg G] - v \rightarrow \square[\neg F] - v$ by (intro $M2$) auto

  thus $?thesis$ unfolding angle-action-def by force

qed

lemma $AA6$: $\vdash \square[P] - v \land \Diamond\langle P\rangle - v \rightarrow \Diamond\langle Q\rangle - v$

proof

- have $\vdash \Diamond\langle P \rightarrow Q \land P\rangle - v \rightarrow \Diamond\langle Q\rangle - v$ by (rule $AA7$) auto

  with $AA2$ show $?thesis$ by (rule lift-imp-trans)

qed

lemma $AA8$: $\vdash \square[P] - v \land \Diamond(A) - v \rightarrow \Diamond\langle\square[P] - v \land A\rangle - v$

proof

- have $\vdash \square[\square[P] - v] - v \land \Diamond(A) - v \rightarrow \Diamond\langle\square[P] - v \land A\rangle - v$ by (rule $AA2$)

  with $P5$ show $?thesis$ by force

qed

lemma $AA9$: $\vdash \square[P] - v \land \Diamond(A) - v \rightarrow \Diamond\langle[P] - v \land A\rangle - v$

proof

- have $\vdash \square[P] - v \land \Diamond(A) - v \rightarrow \Diamond\langle[P] - v \land A\rangle - v$ by (rule $AA2$)

  thus $?thesis$ by simp

qed

lemma $AA10$: $\vdash \neg\langle\square[P] - v \land \Diamond(\neg P) - v\rangle$

unfolding angle-action-def by auto

lemma $AA11$: $\vdash \neg\Diamond(v \$ = \$ v) - v$

unfolding dualization-rew by (rule $ax5$)

lemma $AA15$: $\vdash \Diamond(P \land Q) - v \rightarrow \Diamond(P) - v$
by (rule AA7) auto

lemma AA16: \(\vdash \Diamond (P \land Q)_v \rightarrow \Diamond (Q)_v\)
  by (rule AA7) auto

lemma AA13: \(\vdash \Diamond (P)_v \rightarrow \Diamond (v \neq v)_v\)
proof -
  have \(\vdash \Box v \neq v_v \land \Diamond (P)_v \rightarrow \Diamond (v \neq v \land P)_v\) by (rule AA2)
  hence \(\vdash \Diamond (P)_v \rightarrow \Diamond (v \neq v \land P)_v\) by (simp add: ax5[int-rewrite])
from this AA15 show ?thesis by (rule lift-imp-trans)
qed

lemma AA14: \(\vdash \Diamond (P \lor Q)_v = (\Diamond (P)_v \lor \Diamond (Q)_v)\)
proof -
  have \(\vdash \Box \neg (P \lor Q)_v = \Box \neg P \land \neg Q_v\) by (rule MM10) auto
  hence \(\vdash \Box \neg (P \lor Q)_v = (\Box \neg P_v \land \Box \neg Q_v)\) by (unfold M8[int-rewrite])
thus ?thesis unfolding angle-action-def by auto
qed

lemma AA17: \(\vdash \Diamond ([P]_v \land A)_v \rightarrow \Diamond (P \land A)_v\)
proof -
  have \(\vdash \Box \neg ([P]_v \land A)_v \rightarrow \Box \neg ([P]_v \land A)_v\)
    by (rule M2) (auto simp: actrans-def unch-def)
  with ax5[of v] show ?thesis
    unfolding angle-action-def M8[int-rewrite] by force
qed

lemma AA19: \(\vdash \Box P \land \Diamond \langle A \rangle_v \rightarrow \Diamond (P \land \langle A \rangle_v)\)
  using P4 by (force intro: AA2[unlift-rule])

lemma AA20:
  assumes h1: \(\langle P \rightarrow \Box P \lor Q \rangle\)
    and h2: \(\langle P \land A \rightarrow \Box Q \rangle\)
    and h3: \(\langle P \land \text{Unchanged } w \rightarrow \Box P \rangle\)
  shows \(\vdash \Box \langle P \land \langle A \rangle_v \rightarrow \langle P \land \langle A \rangle_v \rightarrow \langle P \rangle \rightarrow \Box Q \rangle\)
proof -
  from h2 E23 have \(\langle P \land A \rightarrow \Box Q \rangle\) by force
  hence \(\vdash \Diamond (P \land A)_v \rightarrow \Diamond (Q)_v\) by (rule AA7)
  with E25[of TEMP \(\Diamond Q_v\)] have \(\vdash \Diamond (P \land A)_v \rightarrow \Diamond Q_v\) by force
  with AA19 have \(\vdash \Box (P \land \langle A \rangle_v) \rightarrow \Diamond Q_v\) by (rule lift-imp-trans)
  with LT29[OF h1 h3] have \(\vdash \Box P \rightarrow \Diamond \langle A \rangle_v \rightarrow \Diamond (P \rightarrow \Box Q)\) by force
  thus ?thesis unfolding leadsto-def by (rule STL4)
qed

lemma AA21: \(\langle P \rightarrow \Box F \rangle_v \rightarrow \Box \Box F\)
  using pa2x5[of TEMP \(\rightarrow F_v\)] unfolding angle-action-def eventually-def by auto

theorem AA24 [simp-unl]: \(\vdash \Diamond \langle P \rangle_v \rightarrow \Diamond (P)_v\)
  unfolding angle-action-def angle-actrans-def by simp
lemma AA22:
assumes h1: \( \neg P \land N \longrightarrow \O P \lor \O Q \)
and h2: \( \neg P \land N \land \langle A \rangle \cdot v \longrightarrow \O Q \)
and h3: \( \neg P \land \text{Unchanged } w \longrightarrow \O P \)
shows \( \Box N \land \Box \O P \longrightarrow \O \langle A \rangle \cdot v \longrightarrow (P \rightsquigarrow Q) \)
proof –
from h2 have \( \neg ((N \land P) \land A) \cdot v \longrightarrow \O Q \) by (auto simp: angle-actrans-sem[int-rewrite])
from pref-imp-trans[OF this E23] have \( \Diamond ((N \land P) \land A) \cdot v \longrightarrow \Diamond \langle \Diamond Q \rangle \cdot v \)
by (rule AA7)
  hence \( \Diamond ((N \land P) \land A) \cdot v \longrightarrow \O Q \) by (force dest: E25[unlift-rule])
with AA19 have \( \Box (N \land P) \land \Diamond \langle A \rangle \cdot v \longrightarrow \O Q \) by (rule lift-imp-trans)
  hence \( \Box N \land \Box P \land \Box \langle A \rangle \cdot v \longrightarrow \O Q \) by (auto simp: STL5[int-rewrite])
with LT31[OF h1 h3] have \( \Box N \land (\Box P \longrightarrow \Diamond \langle A \rangle \cdot v) \longrightarrow (P \longrightarrow \O Q) \) by
force
  hence \( \Box (\Box N \land (\Box P \longrightarrow \Diamond \langle A \rangle \cdot v)) \longrightarrow \Box (P \longrightarrow \O Q) \) by (rule STL4)
thus \( \text{thesis by (simp add: leadsto-def STL5[int-rewrite])} \)
qed

lemma AA23:
assumes \( \neg P \land N \longrightarrow \O P \lor \O Q \)
and \( \neg P \land N \land \langle A \rangle \cdot v \longrightarrow \O Q \)
and \( \neg P \land \text{Unchanged } w \longrightarrow \O P \)
shows \( \Box N \land \Box \langle A \rangle \cdot v \longrightarrow (P \longrightarrow Q) \)
proof –
  have \( \Box \langle A \rangle \cdot v \longrightarrow \Box (\Box P \longrightarrow \Diamond \langle A \rangle \cdot v) \) by (rule STL4) auto
with AA22[OF assms] show \( \text{thesis by force} \)
qed

lemma AA25:
assumes \( \neg \langle P \rangle \cdot v \longrightarrow \langle Q \rangle \cdot w \)
shows \( \Diamond \langle P \rangle \cdot v \longrightarrow \Diamond \langle Q \rangle \cdot w \)
proof –
  from h have \( \Diamond \langle \langle P \rangle \cdot v \rangle \cdot v \longrightarrow \Diamond \langle \langle P \rangle \cdot w \rangle \cdot v \)
    by (intro AA7) (auto simp: angle-actrans-def actrans-def)
with AA4 have \( \Diamond \langle P \rangle \cdot v \longrightarrow \Diamond \langle \langle P \rangle \cdot w \rangle \cdot w \) by force
from this AA7[OF h] have \( \Diamond \langle P \rangle \cdot v \longrightarrow \Diamond \langle \langle Q \rangle \cdot w \rangle \cdot w \) by (rule lift-imp-trans)
thus \( \text{thesis by simp} \)
qed

lemma AA26:
assumes \( \neg \langle A \rangle \cdot v = \langle B \rangle \cdot w \)
shows \( \Diamond \langle A \rangle \cdot v = \Diamond \langle B \rangle \cdot w \)
proof –
  from h have \( \neg \langle A \rangle \cdot v \longrightarrow \langle B \rangle \cdot w \) by auto
  hence \( \Diamond \langle A \rangle \cdot v \longrightarrow \Diamond \langle B \rangle \cdot w \) by (rule AA25)
moreover
  from h have \( \neg \langle B \rangle \cdot w \longrightarrow \langle A \rangle \cdot v \) by auto
  hence \( \Diamond \langle B \rangle \cdot w \longrightarrow \Diamond \langle A \rangle \cdot v \) by (rule AA25)

62
ultimately
show ?thesis by force
qed

theorem AA28[simp-and]: ⊢ ◯(A) = (A)
  unfolding eventually-def angle-action-def by simp

theorem AA29: ⊢ □[N] ∨ □◊(A) −→ □◊(N ∧ A)
proof −
  have ⊢ □[□[N]] ∨ ◯(A) −→ □◊(N ∧ A) by (rule STL4[OF AA2])
thus ?thesis by (simp add: STL5[int-rewrite])
qed

theorem AA30[simp-and]: ⊢ ◯(◊(P) ∨ F) = ◯(P)
  unfolding angle-action-def by simp

theorem AA31: ⊢ ◯(¬ F) −→ ◯(¬ F)
  using E21[of TEMP □[A]] by simp

lemma AA32[simp-and]: ⊢ ◯(¬ F) = ◯(¬ F)
  using E27[of TEMP ◯(A)] by simp

5.6 Lemmas about the next operator

lemma N2: assumes h: ⊢ F = G shows | ⊨ ◯ F = ◯ G
  by (simp add: h[int-rewrite])

lemmas next-and = T8

lemma next-or: | ⊨ ◯ (F ∨ G) = (◊ F ∨ ◯ G)
  proof (rule pref-iffI)
    have | ⊨ ◯ ((F ∨ G) ∧ ◯ F) −→ ◯ G by (rule N1) auto
    thus | ⊨ ◯ (F ∨ G) −→ ◯ F ∨ ◯ G by (auto simp: T8[int-rewrite])
next
  have | ⊨ ◯ F −→ ◯ (F ∨ G) by (rule N1) auto
  moreover have | ⊨ ◯ G −→ ◯ (F ∨ G) by (rule N1) auto
ultimately show | ⊨ ◯ (F ∨ G) −→ ◯ (F ∨ G) by force
qed

lemma next-imp: | ⊨ ◯ (F −→ G) = (◊ F −→ ◯ G)
  proof (rule pref-iffI)
    have | ⊨ ◯ G −→ ◯ (F −→ G) by (rule N1) auto
    moreover have | ⊨ ◯ ¬ F −→ ◯ (F −→ G) by (rule N1) auto
ultimately show | ⊨ (◊ F −→ ◯ G) −→ ◯ (F −→ G) by force
qed (rule pax2)
lemmas next-not = pax1

lemma next-eq: \(\sim (F = G) = (\Diamond F = G)\)
proof
  have \(\sim (F = G) = \Diamond ((F \rightarrow G) \land (G \rightarrow F))\) by (rule N2) auto
  from this[int-rewrite] show ?thesis
  by (auto simp: next-and[int-rewrite] next-imp[int-rewrite])
qed

lemma next-noteq: \(\sim (F \neq G) = (\Diamond F \neq G)\)
by (simp add: next-eq[int-rewrite])

lemma next-const[simp-unl]: \(\sim \#P = \#P\)
proof (cases P)
  assume P
  hence 1: \(\vdash \#P\) by auto
  hence \(\sim \#P\) by (rule nex)
  with 1 show ?thesis by force
next
  assume \(\neg P\)
  hence 1: \(\vdash \neg \#P\) by auto
  hence \(\sim \neg \#P\) by (rule nex)
  with 1 show ?thesis by force
qed

The following are proved semantically because they are essentially first-order theorems.

lemma next-fun1: \(\sim f < x > = f < \circ x >\)
by (auto simp: nexts-def)

lemma next-fun2: \(\sim f < x,y > = f < \circ x,\circ y >\)
by (auto simp: nexts-def)

lemma next-fun3: \(\sim f < x,y,z > = f < \circ x,\circ y,\circ z >\)
by (auto simp: nexts-def)

lemma next-fun4: \(\sim f < x,y,z,zz > = f < \circ x,\circ y,\circ z,\circ zz >\)
by (auto simp: nexts-def)

lemma next-forall: \(\sim (\forall x. P x) = (\forall x. \circ P x)\)
by (auto simp: nexts-def)

lemma next-exists: \(\sim (\exists x. P x) = (\exists x. \circ P x)\)
by (auto simp: nexts-def)

lemma next-exists1: \(\sim (\exists! x. P x) = (\exists! x. \circ P x)\)
by (auto simp: nexts-def)

Rewrite rules to push the “next” operator inward over connectives. (Note
that axiom \textit{pax1} and theorem \textit{next-const} are anyway active as rewrite rules.)

\textbf{lemmas} \textit{next-commutes}[\textit{int-rewrite}] =
\begin{itemize}
  \item next-and
  \item next-or
  \item next-imp
  \item next-eq
  \item next-fun1
  \item next-fun2
  \item next-fun3
  \item next-fun4
  \item next-forall
  \item next-exists
  \item next-exists1
\end{itemize}

\textbf{lemmas} \textit{ifs-eq}[\textit{int-rewrite}] = after-fun3 next-fun3 before-fun3

\textbf{lemmas} \textit{next-always} = \textit{pax3}

\textbf{lemma} \textit{t1}:
\begin{align*}
& | \not\Box x = x \\
& \text{by} \quad \text{(simp add: before-def after-def nexts-def first-tail-second)}
\end{align*}

Theorem \textit{next-eventually} should not be used ”blindly”.

\textbf{lemma} \textit{next-eventually}:
\begin{itemize}
  \item assumes \textit{h}:
  \item shows \textit{\not\Box F} \rightarrow \neg F \rightarrow \not\Box F
\end{itemize}

\textbf{proof} –
\begin{itemize}
  \item from \textit{h} have \textit{1}:
  \item have \textit{\not\Box \not F} = (\neg F \land \not\Box \neg F) \text{ unfolding } T7[\textit{OF pre-id-unch[OF 1]}, \textit{int-rewrite}]
  \item by simp
  \item thus \textit{?thesis} by (auto simp: \textit{eventually-def})
\end{itemize}
\textbf{qed}

\textbf{lemma} \textit{next-action}:
\begin{align*}
& | \not\Box[P]-v \rightarrow \Box\Box[P]-v \\
& \text{using } \textit{pax4}[\textit{of } P \textit{ v}] \text{ by auto}
\end{align*}

5.7 Higher Level Derived Rules

In most verification tasks the low-level rules discussed above are not used directly. Here, we derive some higher-level rules more suitable for verification. In particular, variants of Lamport’s rules \textit{TLA1, TLA2, INV1} and \textit{INV2} are derived, where \textit{|} is used where appropriate.

\textbf{theorem} \textit{TLA1}:
\begin{itemize}
  \item assumes \textit{H}:
  \item shows \textit{\Box P} = (P \land \Box[P \rightarrow \Box P]-f)
\end{itemize}

\textbf{proof} (rule \textit{int-ifI})
\begin{itemize}
  \item from \textit{ax1}\{P\} \textit{M0}\{P f\} \textit{show} \vdash \Box P \rightarrow P \land \Box[P \rightarrow \Box P]-f \text{ by force}
\end{itemize}

\textbf{next}
\begin{itemize}
  \item from \textit{ax3}\{\textit{OF H}\} \textit{show} \vdash P \land \Box[P \rightarrow \Box P]-f \rightarrow \Box P \text{ by auto}
\end{itemize}
\textbf{qed}

\textbf{theorem} \textit{TLA2}:
\begin{itemize}
  \item assumes \textit{h1}:
  \item and \textit{h2}:
  \item shows \textit{\Box P} \land \Box[A]-f \rightarrow \Box[Q \land \Box[B]-g}
\end{itemize}

\textbf{proof} –
\begin{itemize}
  \item from \textit{h2} have \vdash \Box[P \land \Box P \land [A]-f]-g \rightarrow \Box[B]-g \text{ by (rule } M2\text{)}
\end{itemize}
\textbf{qed}
hence $\vdash \Box P \land \Box P \land \Box [A]-f \rightarrow \Box [B]-g$ by (auto simp add: M8[int-rewrite])

with $M1$ of $P$ $g$ $T4$ of $A f g$ have $\vdash \Box P \land \Box [A]-f \rightarrow \Box [B]-g$ by force

with $h1$[THEN STL4] show $\text{thesis by force}$

qed

theorem INV1:
  assumes $H$: $\sim I \land [N]-f \rightarrow \circ I$
  shows $\vdash I \land \Box [N]-f \rightarrow \Box I$
proof -
  from $H$ have $\sim [N]-f \rightarrow I \rightarrow \circ I$ by auto
  hence $\vdash \Box [N]-f \rightarrow \Box I \rightarrow \circ I$ by (rule M2)
moreover
  from $H$ have $\sim I \land \text{Unchanged } f \rightarrow \circ I$ by (auto simp: actrans-def)
  hence $\vdash \Box [I \rightarrow \circ I]-f \rightarrow I \rightarrow \Box I$ by (rule ax3)
ultimately show $\text{thesis by force}$
qed

theorem INV2: $\vdash \Box I \rightarrow \Box [N]-f = \Box [N \land I \land \circ I]-f$
proof -
  from $M1$ of $I f$ have $\vdash \Box I \rightarrow (\Box [N]-f = \Box [N]-f \land \Box [I \land \circ I]-f)$ by auto
  thus $\text{thesis by } (auto \ simp: \ M8[int-rewrite])$
qed

lemma R1:
  assumes $H$: $\sim \text{Unchanged } w \rightarrow \text{Unchanged } v$
  shows $\vdash \Box [F]-w \rightarrow \Box [F]-v$
proof -
  from $H$ have $\sim [F]-w \rightarrow [F]-v$ by (auto simp: actrans-def)
  thus $\text{thesis by } (rule \ M11)$
qed

theorem invmono:
  assumes $h1$: $\vdash I \rightarrow P$
  and $h2$: $\sim P \land [N]-f \rightarrow \circ P$
  shows $\vdash I \land \Box [N]-f \rightarrow \Box P$
using $h1$ INV1[OF $h2$] by force

theorem preimpsplit:
  assumes $\sim I \land N \rightarrow Q$
  and $\sim I \land \text{Unchanged } v \rightarrow Q$
  shows $\sim I \land [N]-v \rightarrow Q$
using assms[unlift-rule] by (auto simp: actrans-def)

theorem refinement1:
  assumes $h1$: $\vdash P \rightarrow Q$
  and $h2$: $\sim I \land \circ I \land [A]-f \rightarrow [B]-g$
  shows $\vdash P \land \Box [I \land [A]-f \rightarrow Q \land \Box [B]-g$
proof -
  have $\vdash I \rightarrow \# \text{True}$ by simp
from this h2 have ⊨ □I ∧ □[A]-f → □# True ∧ □[B]-g by (rule TLA2)
with h1 show ?thesis by force
qed

theorem inv-join:
assumes ⊨ P → □Q and ⊨ P → □R
shows ⊨ P → □(Q ∧ R)
using assms[unlift-rule] unfolding STL5[int-rewrite] by force

lemma inv-cases: ⊨ □(A → B) ∧ □(¬A → B) → □B
proof –
  have ⊨ □((A → B) ∧ (¬A → B)) → □B by (rule STL4) auto
  thus ?thesis by (simp add: STL5[int-rewrite])
qed
end

6 Liveness

theory Liveness
imports Rules
begin
This theory derives proof rules for liveness properties.
definition enabled :: 'a formula ⇒ 'a formula
where enabled F ≡ λ s. ∃ t. ((first s) ## t) ⊨ F
syntax -Enabled :: lift ⇒ lift ((Enabled -) [90] 90)
translations -Enabled ⇌ CONST enabled
definition WeakF :: ('a::world) formula ⇒ ('a,'b) stfun ⇒ 'a formula
where WeakF F v ≡ TEMP ♦□Enabled ⟨F⟩-v → □♦⟨F⟩-v

definition StrongF :: ('a::world) formula ⇒ ('a,'b) stfun ⇒ 'a formula
where StrongF F v ≡ TEMP □♦Enabled ⟨F⟩-v → □♦⟨F⟩-v

Lamport’s TLA defines the above notions for actions. In TLA∗, (pre-
)formulas generalise TLA’s actions and the above definition is the natural
generalisation of enabledness to pre-formulas. In particular, we have chosen
to define enabled such that it yields itself a temporal formula, although its
value really just depends on the first state of the sequence it is evaluated
over. Then, the definitions of weak and strong fairness are exactly as in
TLA.
syntax
-WF :: [lift,lift] ⇒ lift ((WF'(-)'-(-)) [20,1000] 90)
-SF :: [lift,lift] ⇒ lift ((SF'(-)'-(-)) [20,1000] 90)
-WFsp :: [lift,lift] ⇒ lift ((WF'(-)'-(-)) [20,1000] 90)
-SFsp :: [lift,lift] ⇒ lift ((SF '(-)') (-)) [20,1000] 90

translations
-WF ⇔ CONST WeakF
-SF ⇔ CONST StrongF
-WFsp ⇔ CONST WeakF
-SFsp ⇔ CONST StrongF

6.1 Properties of -Enabled

theorem enabledI: ⊢ F → Enabled F
proof (clarsimp)
fix w
assume w ⊨ F
with seq-app-first-tail[w] have ((first w) # tail w) ⊨ F by simp
thus w ⊨ Enabled F by (auto simp: enabled-def)
qed

theorem enabledE:
assumes s ⊨ Enabled F and ⋀ u. (first s # tail u) ⊨ F =⇒ Q
shows Q
using assms unfolding enabled-def by blast

lemma enabled-mono:
assumes w ⊨ Enabled F and ⊢ F → G
shows w ⊨ Enabled G
using assms[unlifted] unfolding enabled-def by blast

lemma Enabled-disj1: ⊢ Enabled F → Enabled (F ∨ G)
by (auto simp: enabled-def)

lemma Enabled-disj2: ⊢ Enabled F → Enabled (G ∨ F)
by (auto simp: enabled-def)

lemma Enabled-conj1: ⊢ Enabled (F ∧ G) → Enabled F
by (auto simp: enabled-def)

lemma Enabled-conj2: ⊢ Enabled (G ∧ F) → Enabled F
by (auto simp: enabled-def)

lemma Enabled-disjD: ⊢ Enabled (F ∨ G) → Enabled F ∨ Enabled G
by (auto simp: enabled-def)

lemma Enabled-disj: ⊢ Enabled (F ∨ G) = (Enabled F ∨ Enabled G)
by (auto simp: enabled-def)

lemmas enabled-disj-rew = Enabled-disj[int-rewrite]

lemma Enabled-ex: ⊢ Enabled (∃ x. F x) = (∃ x. Enabled (F x))
6.2 Fairness Properties

**lemma** \( WF\text{-}alt: \vdash WF(A) - v = (\Box \Diamond \neg Enabled (A) - v \lor \Box \Diamond (A) - v) \)**

**proof**
- \( \vdash WF(A) - v = (\neg \Diamond \neg Enabled (A) - v \lor \Box \Diamond (A) - v) \) by (auto simp: WeakF-def)
  - **thus** ?thesis by (simp add: dualization-rew)

**qed**

**lemma** \( SF\text{-}alt: \vdash SF(A) - v = (\Diamond \Box \neg Enabled (A) - v \lor \Box \Diamond (A) - v) \)**

**proof**
- \( \vdash SF(A) - v = (\neg \Box \Diamond Enabled (A) - v \lor \Box \Diamond (A) - v) \) by (auto simp: StrongF-def)
  - **thus** ?thesis by (simp add: dualization-rew)

**qed**

**lemma** \( alwaysWFI: \vdash WF(A) - v \rightarrow \Box WF(A) - v \)

**proof**
- unfolding \( WF\text{-}alt \)[int-rewrite] by (rule MM6)

**theorem** \( WF\text{-}always[simp-unl]: \vdash \Box WF(A) - v = WF(A) - v \)

**proof**
- (rule int-iffI[OF ax1 alwaysWFI])

**theorem** \( WF\text{-}eventually[simp-unl]: \vdash \Diamond WF(A) - v = WF(A) - v \)

**proof**
- (rule int-iffI[OF ax1 alwaysWFI])

**lemma** \( alwaysSFI: \vdash SF(A) - v \rightarrow \Box SF(A) - v \)

**proof**
- unfolding \( SF\text{-}alt\)[int-rewrite] by simp

**qed**

**theorem** \( SF\text{-}always[simp-unl]: \vdash \Box SF(A) - v = SF(A) - v \)

**proof**
- (rule int-iffI[OF ax1 alwaysSFI])

**theorem** \( SF\text{-}eventually[simp-unl]: \vdash \Diamond SF(A) - v = SF(A) - v \)

**proof**
- (rule int-iffI[OF ax1 alwaysSFI])

69
by (auto simp: eventually-def)

qed

theorem SF-imp-WF: \( \vdash SF \, (A) \, \rightarrow \, WF \, (A) \, \rightarrow \)

unfolding WeakF-def StrongF-def by (auto dest: E20[unlift-rule])

lemma enabled-WFSF: \( \vdash \Box \, \text{Enabled} \, (F) \, \rightarrow \, WF \,(F) \, \rightarrow \, SF \,(F) \, \rightarrow \)

proof –

have \( \vdash \Box \, \text{Enabled} \, (F) \, \rightarrow \, \Diamond \, \Box \, \text{Enabled} \, (F) \, \rightarrow \) by (rule E3)

hence \( \vdash \Box \, \text{Enabled} \, (F) \, \rightarrow \, WF \,(F) \, \rightarrow \, SF \,(F) \, \rightarrow \) by (auto simp: WeakF-def StrongF-def)

moreover

have \( \vdash \Box \, \text{Enabled} \, (F) \, \rightarrow \, \Box \, \Diamond \, \text{Enabled} \, (F) \, \rightarrow \) by (rule STL4[OF E3])

hence \( \vdash \Box \, \text{Enabled} \, (F) \, \rightarrow \, SF \,(F) \, \rightarrow \, WF \,(F) \, \rightarrow \) by (auto simp: WeakF-def StrongF-def)

ultimately show \( \text{thesis} \) by force

qed

theorem WF1-general:

assumes h1: \( \neg \, P \, \land \, N \, \rightarrow \, \Diamond \, P \, \lor \, \Diamond \, Q \)

and h2: \( \neg \, P \, \land \, N \, \land \, \{A\} \, \rightarrow \, \Diamond \, Q \)

and h3: \( \vdash P \, \land \, N \, \rightarrow \, \text{Enabled} \, \{A\} \, \rightarrow \)

and h4: \( \neg \, P \, \land \, \text{Unchanged} \, w \, \rightarrow \, \Diamond \, P \)

shows \( \vdash \Box \, N \, \land \, WF \,(A) \, \rightarrow \, (P \, \rightarrow \, Q) \)

proof –

have \( \vdash \Box \, \Box \, N \, \land \, WF \,(A) \, \rightarrow \, \Box \, \Box \, P \, \rightarrow \, \Diamond \, \{A\} \, \rightarrow \)

proof (rule STL4)

have \( \vdash \Box \, \Box \, P \, \land \, N \, \rightarrow \, \Diamond \, \Diamond \, \text{Enabled} \, \{A\} \, \rightarrow \) by (rule lift-imp-trans[OF h3[THEN STL4[OF E3]])

hence \( \vdash \Box \, P \, \land \, \Box \, N \, \land \, WF \,(A) \, \rightarrow \, \Box \, \Diamond \, \{A\} \, \rightarrow \) by (auto simp: WeakF-def STL5[int-reWRITE])

with \( ax1 \) of TEMP \( \Diamond \, \{A\} \, \rightarrow \) show \( \vdash \Box \, N \, \land \, WF \,(A) \, \rightarrow \, \Box \, P \, \rightarrow \, \Diamond \, \{A\} \, \rightarrow \) by force

qed

Lamport’s version of the rule is derived as a special case.

theorem WF1:

assumes h1: \( \neg \, P \, \land \, \{N\} \, \rightarrow \, \Diamond \, P \, \lor \, \Diamond \, Q \)

and h2: \( \neg \, P \, \land \, \{N \, \land \, A\} \, \rightarrow \, \Diamond \, Q \)

and h3: \( \vdash P \, \rightarrow \, \text{Enabled} \, \{A\} \, \rightarrow \)

and h4: \( \neg \, P \, \land \, \text{Unchanged} \, v \, \rightarrow \, \Diamond \, P \)

shows \( \vdash \Box \, \{N\} \, \land \, WF \,(A) \, \rightarrow \, (P \, \rightarrow \, Q) \)

proof –

have \( \vdash \Box \, \Box \, \{N\} \, \land \, WF \,(A) \, \rightarrow \, (P \, \rightarrow \, Q) \)

proof (rule WF1-general)

70
from h1 T9[of N v] show \( \neg P \land \Box[N]-v \rightarrow \Diamond P \lor \Diamond Q \) by force

next

from T9[of N v] have \( \neg P \land [N]-v \land (A)-v \rightarrow P \land (N \land A)-v \)

by (auto simp: actrans-def angle-actrans-def)

from this h2 show \( \neg P \land [N]-v \land (A)-v \rightarrow \Diamond Q \) by (rule pref-imp-trans)

next

from h3 T9[of N v] show \( P \land [N]-v \rightarrow \text{Enabled}(A)-v \) by force

qed (rule h4)

thus \( \forall \text{thesis} \) by simp

qed

The corresponding rule for strong fairness has an additional hypothesis \( \Box F \), which is typically a conjunction of other fairness properties used to prove that the helpful action eventually becomes enabled.

**Theorem SF1-general:**

- **assumes** h1: \( \neg P \land N \rightarrow \Diamond P \lor \Diamond Q \)
- **and** h2: \( \neg P \land N \land (A)-v \rightarrow \Diamond Q \)
- **and** h3: \( \sqsubseteq P \land \Box N \land \Box F \rightarrow \Diamond \text{Enabled}(A)-v \)
- **and** h4: \( \neg P \land \text{Unchanged} v \rightarrow \Diamond P \)

**shows** \( \sqsubseteq N \land SF(A)-v \land \Box F \rightarrow (P \rightarrow Q) \)

**proof** –

have \( \sqsubseteq [N] \land SF(A)-v \land \Box F \rightarrow \Diamond (P \rightarrow \Diamond(A)-v) \)

**proof** (rule STL4)

have \( \sqsubseteq (P \land N \land [N]-v) \rightarrow \Diamond \Diamond \text{Enabled}(A)-v \) by (rule STL4[OF h3])

hence \( \sqsubseteq P \land [N] \land [N]-v \land SF(A)-v \rightarrow \Diamond \Diamond(A)-v \) by (auto simp: StrongF-def STL5[int-rewrite])

with ax1[of TEMP] show \( \sqsubseteq N \land SF(A)-v \land \Box F \rightarrow \Diamond P \rightarrow \Diamond(A)-v \)

by force

qed

hence \( \sqsubseteq N \land SF(A)-v \land \Box F \rightarrow \Diamond (P \rightarrow \Diamond(A)-v) \)

by (simp add: STL5[int-rewrite])

with AA22[OF h1 h2 h4] show \( \forall \text{thesis} \) by force

qed

**Theorem SF1:**

- **assumes** h1: \( \neg P \land [N]-v \rightarrow \Diamond P \lor \Diamond Q \)
- **and** h2: \( \neg P \land (N \land A)-v \rightarrow \Diamond Q \)
- **and** h3: \( \sqsubseteq P \land [N]-v \land [N]-v \rightarrow \Diamond \text{Enabled}(A)-v \)
- **and** h4: \( \neg P \land \text{Unchanged} v \rightarrow \Diamond P \)

**shows** \( \sqsubseteq [N]-v \land SF(A)-v \land \Box F \rightarrow (P \rightarrow Q) \)

**proof** –

have \( \sqsubseteq [N]-v \land SF(A)-v \land \Box F \rightarrow (P \rightarrow Q) \)

**proof** (rule SF1-general)

from h1 T9[of N v] show \( \neg P \land [N]-v \rightarrow \Diamond P \lor \Diamond Q \) by force

next

from T9[of N v] have \( \neg P \land [N]-v \land (A)-v \rightarrow P \land (N \land A)-v \)

by (auto simp: actrans-def angle-actrans-def)

from this h2 show \( \neg P \land [N]-v \land (A)-v \rightarrow \Diamond Q \) by (rule pref-imp-trans)

next
Lamport proposes the following rule as an introduction rule for $WF$ formulas.

**Theorem WF2:**

**Assumes** $h_1$: $\lnot (N \land B).f \rightarrow (M)\cdot g$

**and** $h_2$: $\lnot P \land \lnot P \land (N \land A).f \rightarrow B$

**and** $h_3$: $\vdash P \land Enabled (M)\cdot g \rightarrow Enabled (A)\cdot f$

**and** $h_4$: $\vdash \Box[N \land \lnot B].f \land WF(A)\cdot f \land \Box F \land \Box \Box Enabled (M)\cdot g \rightarrow \Box \Box P$

**Shows**

$\vdash \Box[N]\cdot f \land WF(A)\cdot f \land \Box F \land \Box \Box Enabled (M)\cdot g \land \lnot \Box \Box (M)\cdot g \rightarrow \Box \Box (M)\cdot g$

**Proof**

**Have**

$\vdash \Box[N]\cdot f \land WF(A)\cdot f \land \Box F \land \Box \Box Enabled (M)\cdot g \land \lnot \Box \Box (M)\cdot g \rightarrow \Box \Box (M)\cdot g$

**Proof**

**Have** $A$: $\vdash \Box[N]\cdot f \land WF(A)\cdot f \land \Box F \land \Box \Box Enabled (M)\cdot g \land \lnot \Box \Box (M)\cdot g \rightarrow \Box \Box \Box (N)\cdot f \land WF(A)\cdot f \land \Box F \land \Box \Box Enabled (M)\cdot g \land \Box \Box (\lnot M)\cdot g$

**Unfolding** $STL6[int-rewrite]$

**By** (auto simp: $STL5[int-rewrite]$ dualization-rew)

**Have** $B$: $\vdash (\Box(N)\cdot f \land WF(A)\cdot f \land \Box F) \land \Box \Box (\Box \Box Enabled (M)\cdot g \land \Box \Box (\lnot M)\cdot g)$

$$\vdash \Box ((\Box[N]\cdot f \land WF(A)\cdot f \land \Box F) \land \Box \Box (\Box \Box Enabled (M)\cdot g \land \Box \Box (\lnot M)\cdot g))$$

**By** (rule $SE2$)

**From** $lift-imp-trans[OF $A$ $B$]

**Have** $\vdash (\Box[N]\cdot f \land WF(A)\cdot f \land \Box F) \land \Box \Box Enabled (M)\cdot g \land \lnot \Box \Box (M)\cdot g \rightarrow \Box ((\Box(N)\cdot f \land WF(A)\cdot f \land \Box F) \land (\Box \Box Enabled (M)\cdot g \land \Box \Box (\lnot M)\cdot g))$

**By** (simp add: $STL5[int-rewrite]$)

**Moreover**

**From** $h_1$ **Have** $\lnot [N]\cdot f \rightarrow [\lnot M]\cdot g \rightarrow [N \land \lnot B].f$ **By** (auto simp: actrans-def angle-actrans-def)

**Hence** $\vdash \Box[[M]\cdot f] \rightarrow \Box[[\lnot M]\cdot g \rightarrow [N \land \lnot B].f$ **By** (rule $M2$)

**From** $lift-imp-trans[OF this $ax4$] **Have** $\vdash \Box[N]\cdot f \land \Box[[\lnot M]\cdot g \rightarrow \Box[N \land \lnot B].f$

**By** (force intro: $T4[unlift-rule]$)

**With** $h_4$ **Have** $\vdash (\Box[N]\cdot f \land WF(A)\cdot f \land \Box F) \land (\Box \Box Enabled (M)\cdot g \land \Box \Box (\lnot M)\cdot g)$

$$\vdash \Box \Box P$$

**By** force

**From** $STL4-nee[OF this]$ **Have** $\vdash \Box ((\Box[N]\cdot f \land WF(A)\cdot f \land \Box F) \land (\Box \Box Enabled (M)\cdot g \land \Box \Box (\lnot M)\cdot g))$

$$\vdash \Box \Box P$$ **By** simp

**Ultimately**

show thesis by (rule $lift-imp-trans$)

**Qed**

**Have** $2$: $\vdash \Box[N]\cdot f \land WF(A)\cdot f \land \Box \Box Enabled (M)\cdot g \land \Box \Box P \rightarrow \Box \Box (M)\cdot g$

**Proof**
Lamport proposes an analogous theorem for introducing strong fairness, and its proof is very similar, in fact, it was obtained by copy and paste, with minimal modifications.

theorem SF2: assumes h1: "\[ \neg (N \land B) \rightarrow (M)\] -g 
and h2: "\[ \neg P \land O \land N \land A \rightarrow B \] 
and h3: "\[ P \land Enabled (M) -g \rightarrow Enabled (A)\] -f 
and h4: "\[ \neg[N \land \neg B] -f \land SF(A) -f \land \neg \neg F \land \neg \neg Enabled (M) -g \rightarrow \neg \neg F \] 
shows "\[ \neg[N] -f \land SF(A) -f \land \neg \neg F \rightarrow SF(M) -g \] 
proof – have "\[ \neg[N] -f \land SF(A) -f \land \neg \neg F \land \neg \neg Enabled (M) -g \rightarrow \neg \neg F \] -g 
proof – have 1: "\[ \neg[N] -f \land SF(A) -f \land \neg \neg F \land \neg \neg Enabled (M) -g \rightarrow \neg \neg F \] 
proof – have A: "\[ \neg[N] -f \land SF(A) -f \land \neg \neg F \land \neg \neg Enabled (M) -g \rightarrow \neg \neg F \] 

unfolding \( \text{STL6[int-rewrite]} \)

by (auto simp: STL5[int-rewrite] dualization-rws)

have \( \vdash □[\sim N].f \land SF(A).f \land □ F \) and \( □ □ \hi((□[\sim N].f \land SF(A).f \land □ F) \land □ □ \hi(Enabled \( (M).g \land □[\sim M].g)\)

by (rule SE2)

from lift-imp-trans[OF A B]

have \( \vdash □[\sim N].f \land SF(A).f \land □ F \land □ □ \hi(Enabled \( (M).g \land \sim □ □ \hi(M).g \)) \)

by (simp add: STL5[int-rewrite])

moreover

from \( h1 \) have \( \sim N].f \rightarrow [\sim M].g \rightarrow [N \land \sim B].f \) by (auto simp: actrans-def angle-actrans-def)

hence \( \vdash □[\sim N].f \rightarrow □[\sim M].g \rightarrow [N \land \sim B].f \) by (rule M2)

from lift-imp-trans[OF this ax4] have \( \vdash □[\sim N].f \land □[\sim M].g \rightarrow □[N \land \sim B].f \)

by (force intro: T4[unlift-rule])

with \( h4 \) have \( \vdash □[\sim N].f \land SF(A).f \land □ F \land □ □ \hi(Enabled \( (M).g \land □[\sim M].g)\)

by force

from STL4-eve[OF this]

have \( \vdash □ □ \hi((□[\sim N].f \land SF(A).f \land □ F) \land □ □ \hi(Enabled \( (M).g \land □[\sim M].g)\)) \rightarrow □ □ P \) by simp

ultimately

show \( \square \thinspace \text{thesis} \) by (rule lift-imp-trans)

qed

have 2: \( \vdash □[\sim N].f \land SF(A).f \land □ □ \hi(Enabled \( (M).g \land □ □ P \) → □ □ (M).g\)

proof

–

have \( \vdash □ □ \hi(P \land Enabled \( (M).g \land SF(A).f \rightarrow □ □ \hi(A).f\)

using \( h3[T\text{HE}N \text{STL4-eve, THEN STL4}] \) by (auto simp: StrongF-def)

with E28 have \( A: \vdash □ □ P \land □ □ \hi(Enabled \( (M).g \land SF(A).f \rightarrow □ □ \hi(A).f\)

by force

have \( B: \vdash □[\sim N].f \land □ □ P \land □ □ \hi(A).f \rightarrow □ □ (M).g \)

proof

–

from M1[of P f] have \( \vdash □ □ P \land □ □ \hi(N \land A).f \rightarrow □ □ ((P \land □ P) \land (N \land A)).f\)

by (force intro: AA29[unlift-rule])

hence \( \vdash □ □ (P \land □ □ \hi((P \land □ P) \land (N \land A)).f \)

by (rule STL4-eve[OF STL4])

hence \( \vdash □ □ (P \land □ □ \hi(N \land A).f \rightarrow □ □ ((P \land □ P) \land (N \land A)).f \)

by (simp add: STL6[int-rewrite])

with AA29[of N f A]

have \( B1: \vdash □[\sim N].f \land □ □ P \land □ □ \hi(A).f \rightarrow □ □ ((P \land □ P) \land (N \land A)).f \)

by force

–

from \( h2 \) have \( \sim ((P \land □ P) \land (N \land A)).f \rightarrow (N \land B).f \)

by (auto simp: angle-actrans-sem[unlifted])

from \( B1 \) this[THEN AA25, THEN STL4] have \( \vdash □[\sim N].f \land □ □ P \land □ □ \hi(A).f \rightarrow □ □ (N \land B).f \)

by (rule lift-imp-trans)

moreover

74
have $\vdash \Box \Diamond (N \land B)$-f $\rightarrow \Box \Diamond (M)$-g by (rule $h1[THEN\ AA25,\ THEN\ STL4]$)
ultimately show thesis by (rule lift-imp-trans)
qed
from $A\ B$ show thesis by force
qed
from $1\ 2$ show thesis by force
qed
thus thesis by (auto simp: StrongF-def)
qed

This is the lattice rule from TLA

theorem wf-leadsto:
assumes $h1$: $\text{wf } r$
and $h2$: $\forall x. \vdash F x \leadsto (G \lor (\exists y. \#(((y,x) \in r) \land F y)))$
shows $\vdash F x \leadsto G$
using $h1$
proof (rule wf-induct)
fix $x$
assume $ih$: $\forall y. (y, x) \in r \rightarrow (\vdash F y \leadsto G)$
show $\vdash F x \leadsto G$
proof -
from $ih$ have $\vdash (\exists y. \#(((y,x) \in r) \land F y) \leadsto G$
by (force simp: LT21[int-rewrite] LT33[int-rewrite])
with $h2$ show thesis by (force intro: LT19[unlift-rule])
qed

6.3 Stuttering Invariance

theorem stut-Enabled: $\text{STUTINV Enabled } (F)$-v
by (auto simp: enabled-def stutinv-def dest!: sim-first)

theorem stut-WF: $\text{NSTUTINV } F \Rightarrow \text{STUTINV } WF(F)$-v
by (auto simp: WeakF-def stut-Enabled bothstutinvs)

theorem stut-SF: $\text{NSTUTINV } F \Rightarrow \text{STUTINV } SF(F)$-v
by (auto simp: StrongF-def stut-Enabled bothstutinvs)

lemmas livestutinv = stat-WF stat-SF stat-Enabled

end

7 Representing state in TLA*

theory State
imports Liveness
begin
We adopt the hidden state approach, as used in the existing Isabelle/HOL TLA embedding [7]. This approach is also used in [3]. Here, 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.

typedec! 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

Formalizing type state would require formulas to be tagged with their underlying state space and would result in a system that is much harder to use. (Unlike Hoare logic or Unity, TLA has quantification over state variables, and therefore one usually works with different state spaces within a single specification.) Instead, state is just an anonymous type whose only purpose is to provide Skolem constants. Moreover, 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, mainly to define the enabledness of actions. The user identifies (tuples of) “base” state variables in a specification via the “meta predicate” basevars, which is defined here.

definition stvars :: 'a statefun ⇒ bool

where basevars-def: stvars ≡ surj

syntax

PRED :: lift ⇒ 'a

-stvars :: lift ⇒ bool

(PRED -)
(basevars -)

translations

PRED P → (P::state => -)
-stvars = CONST stvars

Base variables may be assigned arbitrary (type-correct) values. In the following lemma, note that vs may be a tuple of variables. The correct identification of base variables is up to the user who must take care not to introduce an inconsistency. For example, basevars (x, x) would definitely be inconsistent.

lemma basevars: basevars vs ⇒ ∃u. vs u = c

proof (unfold basevars-def surj-def)
assume \( \forall y \exists x. y = \text{vs} x \)
then obtain \( x \) where \( c = \text{vs} x \) by \text{blast}
thus \( \exists u. \text{vs} u = c \) by \text{blast}

\text{qed}

\text{lemma baseE:}
\begin{align*}
\text{assumes } & H1: \text{basevars } v \\
\text{and } & H2: \forall x. v x = c \implies Q \\
\text{shows } & Q \\
\text{using } & H1[\text{THEN } \text{basevars}] \ H2 \text{ by } auto
\end{align*}

A variant written for sequences rather than single states.

\text{lemma first-baseE:}
\begin{align*}
\text{assumes } & H1: \text{basevars } v \\
\text{and } & H2: \forall x. v (\text{first } x) = c \implies Q \\
\text{shows } & Q \\
\text{using } & H1[\text{THEN } \text{basevars}] \ H2 \text{ by } (\text{force simp: first-def})
\end{align*}

\text{lemma base-pair1:}
\begin{align*}
\text{assumes } & h: \text{basevars } (x,y) \\
\text{shows } & \text{basevars } x \\
\text{proof } & (\text{auto simp: basevars-def}) \\
\text{fix } & c \\
\text{from } & h[\text{THEN } \text{basevars}] \ \text{obtain } s \ \text{where } (\text{LIFT } (x,y)) \ s = (c, \text{arbitrary}) \text{ by auto} \\
\text{thus } & c \in \text{range } x \text{ by auto}
\end{align*}
\text{qed}

\text{lemma base-pair2:}
\begin{align*}
\text{assumes } & h: \text{basevars } (x,y) \\
\text{shows } & \text{basevars } y \\
\text{proof } & (\text{auto simp: basevars-def}) \\
\text{fix } & d \\
\text{from } & h[\text{THEN } \text{basevars}] \ \text{obtain } s \ \text{where } (\text{LIFT } (x,y)) \ s = (\text{arbitrary}, d) \text{ by auto} \\
\text{thus } & d \in \text{range } y \text{ by auto}
\end{align*}
\text{qed}

\text{lemma base-pair: } \text{basevars } (x,y) \implies \text{basevars } x \land \text{basevars } y
\text{ by } (\text{auto elim: base-pair1 base-pair2})

Since the \textit{unit} type has just one value, any state function of unit type satisfies the predicate \textit{basevars}. The following theorem can sometimes be useful because it gives a trivial solution for \textit{basevars} premises.

\text{lemma unit-base: } \text{basevars } (v::\text{state} \Rightarrow \text{unit})
\text{ by } (\text{auto simp: basevars-def})

A pair of the form \((x,x)\) will generally not satisfy the predicate \textit{basevars} – except for pathological cases such as \(x::\text{unit}\).

\text{lemma}
fixes $x :: state \Rightarrow bool$
assumes $h1 :: basevars (x,x)$
shows $False$
proof
from $h1$ have $\exists u. (LIFT (x,x)) u = (False, True)$ by (rule basevars)
thus $False$ by auto
qed

lemma
fixes $x :: state \Rightarrow nat$
assumes $h1 :: basevars (x,x)$
shows $False$
proof
from $h1$ have $\exists u. (LIFT (x,x)) u = (0, 1)$ by (rule basevars)
thus $False$ by auto
qed

The following theorem reduces the reasoning about the existence of a state sequence satisfying an enabledness predicate to finding a suitable value $c$ at the successor state for the base variables of the specification. This rule is intended for reasoning about standard TLA specifications, where $Enabled$ is applied to actions, not arbitrary pre-formulas.

lemma base-enabled:
  assumes $h1 :: basevars vs$
  and $h2 :: \forall u. vs (first u) = c \Longrightarrow ((first s) \#\# u) \models F$
  shows $s \models F$
using $h1$ proof (rule first-baseE)
  fix $t$
  assume $vs (first t) = c$
  hence $((first s) \#\# t) \models F$ by (rule $h2$)
  thus $s \models F$ unfolding enabled-def by blast
qed

7.1 Temporal Quantifiers

In [5], Lamport gives a stuttering invariant definition of quantification over (flexible) variables. It relies on similarity of two sequences (as supported in our TLA.Sequence theory), and equivalence of two sequences up to a variable (the bound variable). However, sequence equivalence up to a variable, requires state equivalence up to a variable. Our state representation above does not support this, hence we cannot encode Lamport’s definition in our theory. Thus, we need to axiomatise quantification over (flexible) variables. Note that with a state representation supporting this, our theory should allow such an encoding.

consts
  $EEx :: ('a statefun \Rightarrow temporal) \Rightarrow temporal$ (binder $Eex 10$)
  $AAll :: ('a statefun \Rightarrow temporal) \Rightarrow temporal$ (binder $Aall 10$)
syntax

-EEx :: [idts, lift] => lift ((∃∃ -./ -) [0,10] 10)
-AAll :: [idts, lift] => lift ((∀∀ -./ -) [0,10] 10)

translations

-EEx v A == Eex v . A
-AAll v A == Aall v . A

axiomatization where

eexI: ⊢ F x ⇒ (∃ x. F x)
and eexE: [ s | (∃ x. F x) ; basevars vs; (! x. [ basevars (x,vs); s |= F x ] ⇒ s |= G)]
⇒ (s |= G)
and all-def: ⊢ (∀ x. F x) = (¬ (∃ x. ¬(F x)))
and eexSTUT: STUTINV F x ⇒ STUTINV (∃ x. F x)
and history: ⊢ (I ∧ □[A]¬v) = (∃ h. ($h = ha) ∧ I ∧ □[A ∧ h$=hb]¬(h,v))

lemmas eexI-unl = eex[unlift-rule] — w |= F x ⇒ w |= (∃ x. F x)


8 A simple illustrative example

theory Even
imports State
begin

A trivial example illustrating invariant proofs in the logic, and how Isabelle/HOL can help with specification. It proves that \( x \) is always even in a program where \( x \) is initialized as 0 and always incremented by 2.

inductive-set
Even :: nat set
where

| even-zero: 0 ∈ Even
| even-step: n ∈ Even ⇒ Suc(Suc n) ∈ Even

locale Program =

| fixes x :: state ⇒ nat
| init :: temporal
| and act :: temporal
| and phi :: temporal

end
defines \( \text{init} \equiv \text{TEMP } \#x = 0 \)
and \( \text{act} \equiv \text{TEMP } x' = \text{Suc}<\text{Suc}<\#x>> \)
and \( \text{phi} \equiv \text{TEMP } \text{init} \land \Box[\text{act}]\neg x \)

lemma (in Program) \text{STUTINV } \text{phi}
by (auto simp: \text{phi-def init-def act-def stutinvs nstutinvs})

lemma (in Program) \text{inveven}: \vdash \text{phi} \rightarrow \Box(\#x \in \#\text{Even})
unfolding \text{phi-def}
proof (rule invmono)
show \( \vdash \text{init} \rightarrow \#x \in \#\text{Even} \)
by (auto simp: \text{init-def even-zero})

next
show \( \neg \#x \in \#\text{Even} \land [\text{act}]\neg x \rightarrow \Box(\#x \in \#\text{Even}) \)
by (auto simp: \text{act-def even-step tla-defs})
qed

end

9 Lamport’s Inc example

theory Inc
imports State
begin

This example illustrates use of the embedding by mechanising the running example of Lamport’s original TLA paper [5].

datatype \text{pcount} = a | b | g

locale Firstprogram =
fixes \( x :: \text{state } \Rightarrow \text{nat} \)
and \( y :: \text{state } \Rightarrow \text{nat} \)
and \( \text{init} :: \text{temporal} \)
and \( \text{m1} :: \text{temporal} \)
and \( \text{m2} :: \text{temporal} \)
and \( \text{phi} :: \text{temporal} \)
and \( \text{Live} :: \text{temporal} \)
defines \( \text{init} \equiv \text{TEMP } \#x = 0 \land \#y = 0 \)
and \( \text{m1} \equiv \text{TEMP } x' = \text{Suc}<\text{Suc}<\#x>> \land y' = \#y \)
and \( \text{m2} \equiv \text{TEMP } y' = \text{Suc}<\#y>> \land x' = \#x \)
and \( \text{Live} \equiv \text{TEMP } \text{WF}(\text{m1})(x,y) \land \text{WF}(\text{m2})(x,y) \)
and \( \text{phi} \equiv \text{TEMP } (\text{init} \land \Box[\text{m1} \lor \text{m2}]-x,y) \land \text{Live} \)
assumes \text{bvar: basevars } (x,y)

lemma (in Firstprogram) \text{STUTINV } \text{phi}
by (auto simp: \text{phi-def init-def m1-def m2-def stutinvs nstutinvs live-stutinv})
lemma (in Firstprogram) enabled-m1: \( \vdash \) Enabled \( \langle m1 \rangle \cdot (x,y) \)
proof (clarify)
  fix \( s \)
  show \( s \models \) Enabled \( \langle m1 \rangle \cdot (x,y) \)
    by (rule base-enabled[OF bear]) (auto simp: m1-def tla-defs)
qed

lemma (in Firstprogram) enabled-m2: \( \vdash \) Enabled \( \langle m2 \rangle \cdot (x,y) \)
proof (clarify)
  fix \( s \)
  show \( s \models \) Enabled \( \langle m2 \rangle \cdot (x,y) \)
    by (rule base-enabled[OF bear]) (auto simp: m2-def tla-defs)
qed

locale Secondprogram = Firstprogram +
  fixes \( \text{sem} :: \text{state} \Rightarrow \text{nat} \)
  and \( \text{pc1} :: \text{state} \Rightarrow \text{pcount} \)
  and \( \text{pc2} :: \text{state} \Rightarrow \text{pcount} \)
  and \( \text{vars} \)
  and \( \text{initPsi} :: \text{temporal} \)
  and \( \text{alpha1} :: \text{temporal} \)
  and \( \text{alpha2} :: \text{temporal} \)
  and \( \text{beta1} :: \text{temporal} \)
  and \( \text{beta2} :: \text{temporal} \)
  and \( \text{gamma1} :: \text{temporal} \)
  and \( \text{gamma2} :: \text{temporal} \)
  and \( \text{n1} :: \text{temporal} \)
  and \( \text{n2} :: \text{temporal} \)
  and \( \text{Live2} :: \text{temporal} \)
  and \( \text{psi} :: \text{temporal} \)
  and \( \text{I} :: \text{temporal} \)
  defines \( \text{vars} \equiv \text{LIFT} (x,y,\text{sem},\text{pc1},\text{pc2}) \)
  and \( \text{initPsi} \equiv \text{TEMP} \$\text{pc1} = # a \land \$\text{pc2} = # a \land \$x = # 0 \land \$y = # 0 \land \$\text{sem} = # 1 \)
  and \( \text{alpha1} \equiv \text{TEMP} \$\text{pc1} = # a \land \$\text{pc2} = # b \land \$\text{sem} = $\text{sem} - # 1 \land \text{Unchanged} (x,y,\text{pc2}) \)
  and \( \text{beta1} \equiv \text{TEMP} \$\text{pc1} = # a \land \$\text{pc2} = # b \land \$\text{sem} = $\text{sem} - # 1 \land \text{Unchanged} (x,y,\text{pc1}) \)
  and \( \text{gamma1} \equiv \text{TEMP} \$\text{pc1} = # g \land \text{pc1}' = \text{Suc}<\$x> \land \text{Unchanged} (x,\text{sem},\text{pc2}) \)
  and \( \text{n1} \equiv \text{TEMP} \$\text{pc2} = # b \land \text{pc2}' = # g \land \$y' = \text{Suc}<\$y> \land \text{Unchanged} (x,y,\text{pc1}) \)
  and \( \text{alpha2} \equiv \text{TEMP} \$\text{pc1} = # g \land \text{pc1}' = # a \land \text{sem}' = \text{Suc}<\$\text{sem}> \land \text{Unchanged} (x,y,\text{pc2}) \)
  and \( \text{beta2} \equiv \text{TEMP} \$\text{pc2} = # b \land \text{pc2}' = # g \land \text{y}' = \text{Suc}<\$y> \land \text{Unchanged} (x,y,\text{pc2}) \)
  and \( \text{gamma2} \equiv \text{TEMP} \$\text{pc1} = # g \land \text{pc1}' = # a \land \text{sem}' = \text{Suc}<\$\text{sem}> \land \text{Unchanged} (x,y,\text{pc2}) \)
  and \( \text{Live2} \equiv \text{TEMP} \text{SF(n1)} \cdot \text{vars} \land \text{SF(n2)} \cdot \text{vars} \)
and $\psi \equiv \text{TEMP} \ (\text{init} \psi \land \square [n_1 \lor n_2] \cdot \text{vars} \land \text{Live}_2)$
and $I \equiv \text{TEMP} \ (\# \text{sem} = \# 1 \land \# \text{pc}_1 = \# a \land \# \text{pc}_2 = \# a \lor \# \text{sem} = \# 0 \land ((\# \text{pc}_1 = \# a \land \# \text{pc}_2 \in \{\# b, \# g\})$ 
$\lor (\# \text{pc}_2 = \# a \land \# \text{pc}_1 \in \{\# b, \# g\}))$
assumes $\text{bvar}_2$: basevars vars

lemmas (in Secondprogram) $\text{Sact}_2\text{-defs} = \text{n}_1\text{-def} \text{ n}_2\text{-def} \text{ alpha}_1\text{-def} \text{ beta}_1\text{-def} \text{ gamma}_1\text{-def} \text{ alpha}_2\text{-def} \text{ beta}_2\text{-def} \text{ gamma}_2\text{-def}$

Proving invariants is the basis of every effort of system verification. We show that $I$ is an inductive invariant of specification $\psi$.

lemma (in Secondprogram) $\vdash \psi \rightarrow \square I$
proof
  have init: $\vdash \text{init} \psi \rightarrow I$ by (auto simp: init Psi-def I-def)
  have $\neg I \land \text{Unchanged vars} \rightarrow \square I$ by (auto simp: I-def vars-def tla-defs)
  moreover
  have $\neg I \land n_1 \rightarrow \square I$ by (auto simp: I-def vars-def tla-defs)
  moreover
  have $\neg I \land n_2 \rightarrow \square I$ by (auto simp: I-def vars-def tla-defs)
  ultimately have step: $\neg I \land [n_1 \lor n_2] \cdot \text{vars} \rightarrow \square I$ by (force simp: act-trans-def)
  from init step have goal: $\vdash \text{init} \psi \land \square [n_1 \lor n_2] \cdot \text{vars} \rightarrow \square I$ by (rule invmono)
  have $\vdash \text{init} \psi \land \square [n_1 \lor n_2] \cdot \text{vars} \land \text{Live}_2 \Longrightarrow \vdash \text{init} \psi \land \square [n_1 \lor n_2] \cdot \text{vars}$ by auto
with goal show ?thesis unfolding psi-def by auto
qed

Using this invariant we now prove step simulation, i.e. the safety part of the refinement proof.

theorem (in Secondprogram) step-simulation: $\vdash \psi \rightarrow \text{init} \land \square [m_1 \lor m_2] \cdot (x,y)$
proof
  have $\vdash \text{init} \psi \land \square I \land \square [n_1 \lor n_2] \cdot \text{vars} \rightarrow \text{init} \land \square [m_1 \lor m_2] \cdot (x,y)$
proof (rule refinement1)
  show $\vdash \text{init} \psi \rightarrow \text{init}$ by (auto simp: init Psi-def init-def)
next
  show $\neg I \land \square I \land [n_1 \lor n_2] \cdot \text{vars} \rightarrow [m_1 \lor m_2] \cdot (x,y)$ by (auto simp: I-def vars-def tla-defs)
qed
with psi I show ?thesis unfolding psi-def by force
qed

Liveness proofs require computing the enabledness conditions of actions. The first lemma below shows that all steps are visible, i.e. they change at least one variable.

lemma (in Secondprogram) n1-ch: $\neg (n_1) \cdot \text{vars} = n_1$
proof
  have $\neg n_1 \rightarrow (n_1) \cdot \text{vars}$ by (auto simp: Sact2-defs tla-defs vars-def)
  thus ?thesis by (auto simp: angle-act-trans-sem[int rewrite])
qed
lemma (in Secondprogram) enab-alpha1: ⊢ $pc1 = #a \rightarrow \# 0 < $sem → Enabled alpha1
proof (clarsimp simp: tla-defs)
  fix s :: state seq
  assume pc1 (s 0) = a and 0 < sem (s 0)
  thus s |= Enabled alpha1
    by (intro base-enabled[OF bear2]) (auto simp: Sact2-defs tla-defs vars-def)
qed

lemma (in Secondprogram) enab-beta1: ⊢ $pc1 = #b \rightarrow Enabled beta1
proof (clarsimp simp: tla-defs)
  fix s :: state seq
  assume pc1 (s 0) = b
  thus s |= Enabled beta1
    by (intro base-enabled[OF bear2]) (auto simp: Sact2-defs tla-defs vars-def)
qed

lemma (in Secondprogram) enab-gamma1: ⊢ $pc1 = #g \rightarrow Enabled gamma1
proof (clarsimp simp: tla-defs)
  fix s :: state seq
  assume pc1 (s 0) = g
  thus s |= Enabled gamma1
    by (intro base-enabled[OF bear2]) (auto simp: Sact2-defs tla-defs vars-def)
qed

lemma (in Secondprogram) enab-n1:
  ⟨n1⟩-vars = ($pc1 = #a \rightarrow \# 0 < $sem)
unfolding n1-ch[int-rewrite] proof (rule int-iffI)
  show ⟨n1⟩-vars = ($pc1 = #a \rightarrow \# 0 < $sem
    by (auto elim!: enabledE simp: Sact2-defs tla-defs)
next
  show ($pc1 = #a \rightarrow \# 0 < $sem) → Enabled n1
    proof (clarsimp simp: tla-defs)
      fix s :: state seq
      assume pc1 (s 0) = a \rightarrow 0 < sem (s 0)
      thus s |= Enabled n1
        using enab-alpha1[unlift-rule]
        enab-beta1[unlift-rule]
        enab-gamma1[unlift-rule]
        by (cases pc1 (s 0)) (force simp: n1-def Enabled-disj[int-rewrite] tla-defs)
    qed
qed

The analogous properties for the second process are obtained by copy and paste.

lemma (in Secondprogram) n2-ch: ¬ (n2)-vars = n2
proof
  have ¬ n2 → (n2)-vars by (auto simp: Sact2-defs tla-defs vars-def)
thus \( \text{thesis by (auto simp: angle-actrans-sem[int-rewrite])} \)

qed

lemma (in Secondprogram) enab-alpha2: \( \vdash \text{pc2 = \#a \rightarrow \# 0 < \$sem \rightarrow \text{Enabled alpha2}} \)

proof (clarsimp simp: tla-defs)

fix \( s :: \text{state seq} \)

assume pc2 (s 0) = a and 0 < sem (s 0)

thus \( s \models \text{Enabled alpha2} \)

by (intro base-enabled[OF bvar2]) (auto simp: Sact2-defs tla-defs vars-def)

qed

lemma (in Secondprogram) enab-beta2: \( \vdash \text{pc2 = \#b \rightarrow \text{Enabled beta2}} \)

proof (clarsimp simp: tla-defs)

fix \( s :: \text{state seq} \)

assume pc2 (s 0) = b

thus \( s \models \text{Enabled beta2} \)

by (intro base-enabled[OF bvar2]) (auto simp: Sact2-defs tla-defs vars-def)

qed

lemma (in Secondprogram) enab-gamma2: \( \vdash \text{pc2 = \#g \rightarrow \text{Enabled gamma2}} \)

proof (clarsimp simp: tla-defs)

fix \( s :: \text{state seq} \)

assume pc2 (s 0) = g

thus \( s \models \text{Enabled gamma2} \)

by (intro base-enabled[OF bvar2]) (auto simp: Sact2-defs tla-defs vars-def)

qed

lemma (in Secondprogram) enab-n2: \( \vdash \text{Enabled (n2)-vars = (pc2 = \#a \rightarrow \# 0 < sem)} \)

unfolding n2-ch[int-rewrite] proof (rule int-iffI)

show \( \vdash \text{Enabled n2 \rightarrow pc2 = \#a \rightarrow \# 0 < sem} \)

by (auto elim!: enabledE simp: Sact2-defs tla-defs)

next

show \( \vdash (pc2 = \#a \rightarrow \# 0 < \$sem) \rightarrow \text{Enabled n2} \)

proof (clarsimp simp: tla-defs)

fix \( s :: \text{state seq} \)

assume pc2 (s 0) = a \rightarrow 0 < \$sem (s 0)

thus \( s \models \text{Enabled n2} \)

using enab-alpha2[unlift-rule]

enab-beta2[unlift-rule]

enab-gamma2[unlift-rule]

by (cases pc2 (s 0)) (force simp: n2-def Enabled-disj[int-rewrite] tla-defs)+

qed

We use rule SF2 to prove that \( \psi \) implements strong fairness for the abstract action \( m1 \). Since strong fairness implies weak fairness, it follows that \( \psi \) refines the liveness condition of \( \phi \).
Showing that \( \text{pc1} \) \( SF \) (combining leadsto properties. reaches \( \text{pc1} \) \( \text{pc1} \) The plan of the proof is to show that from any state where the first process completely controls this transition.

\[
\begin{align*}
\text{lemma (in Secondprogram) psi-fair-m1: } & \vdash \psi \rightarrow SF(m1)-(x,y) \\
\text{proof –} \\
\text{have } & \vdash \Box[(n1 \lor n2)-\text{vars} \land SF(n1)-\text{vars} \land \Box(I \land SF(n2)-\text{vars}) \rightarrow SF(m1)-(x,y)] \\
\text{proof (rule SF2)} & \end{align*}
\]

Rule SF2 requires us to choose a helpful action (whose effect implies \( (m1)-(x,y) \)) and a persistent condition, which will eventually remain true if the helpful action is never executed. In our case, the helpful action is \( \text{beta1} \) and the persistent condition is \( \text{pc1} = b \).

\[
\begin{align*}
\text{show } & \vdash \Box[(n1 \lor n2) \land \neg \text{beta1}]-\text{vars} \\
& \land SF(n1)-\text{vars} \land \Box(I \land SF(n2)-\text{vars}) \land \Box\Diamond \text{Enabled (m1)-(x, y)} \\
& \rightarrow \Box\Diamond (\$\text{pc1} = \#b) \\
\text{proof –} \\
\text{have } & \vdash \Box[(n1 \lor n2) \land \neg \text{beta1}]-\text{vars} \rightarrow \Box(\$\text{pc1} = \#b \rightarrow \Box(\$\text{pc1} = \#b)) \\
\text{proof (rule STL4)} & \end{align*}
\]

\[
\begin{align*}
\text{have } & \vdash \Box[(n1 \lor n2) \land \neg \text{beta1}]-\text{vars} \rightarrow \Box(\$\text{pc1} = \#b) \\
\text{by (auto simp: Sact2-defs vars-def tla-defs)} & \\
\text{from this[THEN INV1]} & \\
\text{show } & \vdash \Box[(n1 \lor n2) \land \neg \text{beta1}]-\text{vars} \rightarrow \$\text{pc1} = \#b \rightarrow \Box(\$\text{pc1} = \#b) \\
\text{by auto} & \\
\text{qed} & \\
\text{hence } & I: \vdash \Box[(n1 \lor n2) \land \neg \text{beta1}]-\text{vars} \\
& \rightarrow \Diamond(\$\text{pc1} = \#b) \\
& \rightarrow \Diamond\Box(\$\text{pc1} = \#b) \\
\text{by (force intro: E31[unlift-rule])} & \\
\text{have } & \vdash \Box[(n1 \lor n2) \land \neg \text{beta1}]-\text{vars} \land SF(n1)-\text{vars} \land \Box(I \land SF(n2)-\text{vars}) \\
& \rightarrow \Diamond(\$\text{pc1} = \#b) \\
\text{proof –} & \\
\end{align*}
\]

The plan of the proof is to show that from any state where \( \text{pc1} = g \) one eventually reaches \( \text{pc1} = a \), from where one eventually reaches \( \text{pc1} = b \). The result follows by combining leadsto properties.

\[
\begin{align*}
\text{let } & \text{SF} = \text{LIFT (Box[(n1 \lor n2) \land \neg beta1]-vars} \\
& \land SF(n1)-\text{vars} \land \Box(I \land SF(n2)-\text{vars})) \\
\text{Showing that } & \text{pc1} = g \text{ leads to } \text{pc1} = a \text{ is a simple application of rule SF1 because} \\
\text{the first process completely controls this transition.} & \\
\end{align*}
\]

85
have ga: ⊢ ?F →→ ($pc1 = #g → $pc1 = #a)
proof (rule SF1)
  show ¬ $pc1 = #g ∧ [(n1 ∨ n2) ∧ ¬ beta1]-vars → ○($pc1 = #g) ∨ ○($pc1 = #a)
    by (auto simp: Sact2defs vars-def tla-defs)
next
  show ¬ $pc1 = #g ∧ ⟨((n1 ∨ n2) ∧ ¬ beta1) ∧ n1⟩-vars → ○($pc1 = #a)
    by (auto simp: Sact2defs vars-def tla-defs)
next
  have ⊢ $pc1 = #g → Enabled ⟨n1⟩-vars
    unfolding enab-n1 [int-rewrite] by (auto simp: tla-defs)
  hence ⊢ □($pc1 = #g) → Enabled ⟨n1⟩-vars
    by (rule lift-imp-trans [OF ax1])
  hence ⊢ □($pc1 = #g) → ○Enabled ⟨n1⟩-vars
    by (rule lift-imp-trans [OF - E3])
thus ⊢ □($pc1 = #g) ∧ □[((n1 ∨ n2) ∧ ¬ beta1) ∧ n1]-vars ∧ □(I ∧ SF(n2)-vars) → ○Enabled ⟨n1⟩-vars
    by auto
qed

The proof that pc1 = a leads to pc1 = b follows the same basic schema. However, showing that n1 is eventually enabled requires reasoning about the second process, which must liberate the critical section.

have ab: ⊢ ?F →→ ($pc1 = #a → $pc1 = #b)
proof (rule SF1)
  show ¬ $pc1 = #a ∧ [(n1 ∨ n2) ∧ ¬ beta1]-vars → ○($pc1 = #a) ∨ ○($pc1 = #b)
    by (auto simp: Sact2defs vars-def tla-defs)
next
  show ¬ $pc1 = #a ∧ ⟨((n1 ∨ n2) ∧ ¬ beta1) ∧ n1⟩-vars → ○($pc1 = #b)
    by (auto simp: Sact2defs vars-def tla-defs)
next
  show ¬ $pc1 = #a ∧ Unchanged vars → ○($pc1 = #a)
    by (auto simp: vars-def tla-defs)
next

We establish a suitable leadsto-chain.

let ?G = LIFT □[(n1 ∨ n2) ∧ ¬ beta1]-vars ∧ SF(n2)-vars ∧ □($pc1 = #a ∧ I)
have ⊢ ?G → ○($pc2 = #a ∧ $pc1 = #a ∧ I)
proof -

Rule SF1 takes us from pc2 = b to pc2 = g.

have bg2: ⊢ ?G →→ ($pc2 = #b → $pc2 = #g)
proof (rule SF1)  
show |¬ $pc2 = \#b \land [(n1 \lor n2) \land \neg beta1]$-vars $\longrightarrow \Box (pc2 = \#b)$ 
\lor \Box (pc2 = \#g)  
  by (auto simp: Sact2-defs vars-def tla-defs)  
next  
show |¬ $pc2 = \#b \land ((n1 \lor n2) \land \neg beta1) \land n2$-vars $\longrightarrow \Box (pc2 = \#b)$  
  by (auto simp: Sact2-defs vars-def tla-defs)  

Thus, \( PC2 = b \) leads to \( PC2 = g \).

have \( \vdash \forall i. (PC2 = \#b \rightarrow \text{Enabled \ (n2)-vars}) \)  
  unfolding \( \text{enab-n2[\text{int-rewrite}] \ by (auto simp: tla-defs)} \)  
  hence \( \Box (PC2 = \#b) \rightarrow \text{Enabled \ (n2)-vars} \)  
  by (rule lift-imp-trans \[OF ax1\])  
  hence \( \Box (PC2 = \#b) \rightarrow \Diamond \text{Enabled \ (n2)-vars} \)  
  by (rule lift-imp-trans \[OF - E3\])  
  thus \( \Box (PC2 = \#b) \land \Box ((n1 \lor n2) \land \neg beta1)$-vars $\land \Box (PC1 = \#a)$  
  \( \land I \)  
  $\longrightarrow \Diamond \text{Enabled \ (n2)-vars} \)  
  by auto  
qed
qed
with bg2 have \( \vdash ?G \rightarrow (\$pc2 = \#b \leadsto \$pc2 = \#a) \)
  by (force elim: LT13[unlift-rule])
with ga2 have \( \vdash ?G \rightarrow (\$pc2 = \#a \lor \$pc2 = \#b \lor \$pc2 = \#g) \leadsto \)
  \( (\$pc2 = \#a) \)
  unfolding LT17[int-rewrite] LT1[int-rewrite] by force
moreover
have \( \vdash \$pc2 = \#a \lor \$pc2 = \#b \lor \$pc2 = \#g \)
proof (clarsimp simp: tla-defs)
  fix s :: state seq
  assume pc2 \((s 0)\) \(\neq a\) and pc2 \((s 0)\) \(\neq g\)
  thus \(pc2 \((s 0)\) = b\) by (cases pc2 \((s 0)\), auto)
qed
hence \( \vdash ((\$pc2 = \#a \lor \$pc2 = \#b \lor \$pc2 = \#g) \leadsto \$pc2 = \#a) \rightarrow \)
  \( \Box(\$pc2 = \#a) \)
  by (rule frp[OF - LT4])
ultimately
have \( \vdash \vdash ?thesis \) by (auto intro!: SE3[unlift-rule])
qed
moreover
have \( \vdash \vdash \Box(\$pc2 = \#a \land \$pc1 = \#a \land I) \rightarrow \Box Enabled \langle n1 \rangle\)-vars
  unfolding enab-n1[int-rewrite] by (rule STL4-eve) (auto simp: I-def
  tla-defs)
ultimately
  show \( \vdash \Box(\$pc1 = \#a) \land \Box((n1 \lor n2) \land \neg beta1\)-vars \land \Box(I \land
  SF(n2)-vars)
  \rightarrow \Box Enabled \langle n1 \rangle\)-vars
  by (force simp: STL5[int-rewrite])
qed
from ga ab have \( \vdash ?F \rightarrow (\$pc1 = \#g \leadsto \$pc1 = \#b) \)
  by (force elim: LT13[unlift-rule])
with ab have \( \vdash ?F \rightarrow ((\$pc1 = \#a \lor \$pc1 = \#b \lor \$pc1 = \#g) \leadsto \$pc1 = \#b) \rightarrow \)
  \( (\$pc1 = \#a \lor \$pc1 = \#b \lor \$pc1 = \#g) \leadsto \$pc1 = \#b) \rightarrow \)
  \( \Box(\$pc1 = \#b) \)
  by (rule frp[OF - LT4])
ultimately show \( ?thesis \) by (rule lift-imp-trans)
qed
with 1 show \( ?thesis \) by force
qed

88
In the same way we prove that \( psi \) implements strong fairness for the abstract action \( m1 \). The proof is obtained by copy and paste from the previous one.

**Lemma (in Secondprogram)** \( psi\text{-}fair\text{-}m2: \vdash psi \rightarrow SF(m2)-(x,y) \)

**Proof**

- have \( \square[n1 \lor n2]-vars \land SF(n2)-vars \land \square(I \land SF(n1)-vars) \rightarrow SF(m2)-(x,y) \)

   **Rule \( SF2 \)**

Rule \( SF2 \) requires us to choose a helpful action (whose effect implies \( (m2)-(x,y) \)) and a persistent condition, which will eventually remain true if the helpful action is never executed. In our case, the helpful action is \( beta2 \) and the persistent condition is \( pc2 = b \).

```
show \( \angle (\{n1 \lor n2\} \land beta2)-vars \rightarrow \langle m2 \rangle -(x,y) \)
  by (auto simp: beta2-def m2-def vars-def tla-defs)
next
show \( \angle \langle pc2 = #b \rangle \land \langle (\{n1 \lor n2\} \land \langle n1 \lor n2\rangle)-vars \rightarrow beta2 \)
  by (auto simp: n2-def alpha2-def beta2-def gamma2-def tla-defs)
next
show \( \angle \langle pc2 = #b \rangle \land Enabled \langle m2 \rangle -(x,y) \rightarrow Enabled \langle n2 \rangle -vars \)
  unfolding enab-n2[\int\text{-}rewrite] by auto
next
```

The difficult part of the proof is showing that the persistent condition will eventually always be true if the helpful action is never executed. We show that (1) whenever the condition becomes true it remains so and (2) eventually the condition must be true.

```
show \( \square[(n1 \lor n2) \land \neg beta2]-vars \land SF(n2)-vars \land \square(I \land SF(n1)-vars) \land \square \langle pc2 = #b \rangle \rightarrow \square \langle pc2 = #b \rangle \)
proof
  have \( \square[(n1 \lor n2) \land \neg beta2]-vars \rightarrow \square \langle pc2 = #b \rangle \rightarrow \square \langle pc2 = #b \rangle \)
  proof (\text{rule \( STL4 \)})
  have \( \angle \langle pc2 = #b \rangle \land \langle n1 \lor n2 \rangle \land \neg beta2]-vars \rightarrow \angle \langle pc2 = #b \rangle \)
    by (auto simp: Sact2-defs vars-def tla-defs)
  from this[\text{THEN INV1}]
  show \( \square[(n1 \lor n2) \land \neg beta2]-vars \rightarrow \langle pc2 = #b \rangle \rightarrow \square \langle pc2 = #b \rangle \)
  by auto
qed

hence \( 1: \vdash \square[(n1 \lor n2) \land \neg beta2]-vars \rightarrow \Diamond \langle pc2 = #b \rangle \rightarrow \Diamond \langle pc2 = #b \rangle \)
  by (force intro: EL2[\text{unlift-rule}])
  have \( \square[(n1 \lor n2) \land \neg beta2]-vars \land SF(n2)-vars \land \square(I \land SF(n1)-vars) \rightarrow \Diamond \langle pc2 = #b \rangle \)
  proof
```

The plan of the proof is to show that from any state where \( pc2 = g \) one eventually
reaches \( pc2 = a \), from where one eventually reaches \( pc2 = b \). The result follows by combining leadsto properties.

\[
\text{let } \varphi = \text{LIFT} (\Box (n1 \lor n2) \land \neg \beta2 \land SF(n2) \land \Box (I \land SF(n1)))
\]

Showing that \( pc2 = g \) leads to \( pc2 = a \) is a simple application of rule \( SF1 \) because the second process completely controls this transition.

\[
\begin{align*}
\text{have } & \vdash \varphi \rightarrow (\text{pc2} = \#g \rightarrow \text{pc2} = \#a) \\
\text{proof } & (\text{rule SF1}) \\
\text{show } & \neg \text{pc2} = \#g \land [(n1 \lor n2) \land \neg \beta2] \land SF(n2) \land \Box (I \land SF(n1)) \rightarrow \Box (\text{pc2} = \#g) \lor \Box (\text{pc2} = \#a) \\
& \quad \text{by (auto simp: Sact2-defs vars-def tla-defs)} \\
\text{next } & \text{show } \neg \text{pc2} = \#g \land \Box (n1 \lor n2) \land \neg \beta2 \land SF(n2) \land \Box (I \land SF(n1)) \rightarrow \Box (\text{pc2} = \#a) \\
& \quad \text{by (auto simp: vars-def tla-defs)} \\
\text{next } & \text{have } \vdash \text{pc2} = \#g \rightarrow \text{Enabled } (n2) \\
& \quad \text{unfolding enab-n2[int-rewrite] by (auto simp: tla-defs)} \\
& \quad \text{hence } \vdash \Box (\text{pc2} = \#g) \rightarrow \text{Enabled } (n2) \\
& \quad \text{by (rule lift-imp-trans[OF ax1])} \\
& \quad \text{hence } \vdash \Box (\text{pc2} = \#g) \rightarrow \Box \text{Enabled } (n2) \\
& \quad \text{by (rule lift-imp-trans[OF - E3])} \\
& \quad \text{thus } \vdash \Box (\text{pc2} = \#g) \land \Box (n1 \lor n2) \land \neg \beta2 \land SF(n2) \land \Box (I \land SF(n1)) \rightarrow \Box \text{Enabled } (n2) \\
& \quad \text{by auto} \\
\end{align*}
\]

The proof that \( pc2 = a \) leads to \( pc2 = b \) follows the same basic schema. However, showing that \( n2 \) is eventually enabled requires reasoning about the second process, which must liberate the critical section.

\[
\begin{align*}
\text{have } & \vdash \varphi \rightarrow (\text{pc2} = \#a \rightarrow \text{pc2} = \#b) \\
\text{proof } & (\text{rule SF1}) \\
\text{show } & \neg \text{pc2} = \#a \land [(n1 \lor n2) \land \neg \beta2] \land SF(n2) \land \Box (I \land SF(n1)) \rightarrow \Box (\text{pc2} = \#a) \lor \Box (\text{pc2} = \#b) \\
& \quad \text{by (auto simp: Sact2-defs vars-def tla-defs)} \\
\text{next } & \text{show } \neg \text{pc2} = \#a \land \Box (n1 \lor n2) \land \neg \beta2 \land SF(n2) \land \Box (I \land SF(n1)) \rightarrow \Box (\text{pc2} = \#b) \\
& \quad \text{by (auto simp: n2-def alpha2-def beta2-def gamma2-def vars-def tla-defs)} \\
\text{next } & \text{show } \neg \text{pc2} = \#a \land \Box \text{Unchanged vars} \rightarrow \Box (\text{pc2} = \#a) \\
& \quad \text{by (auto simp: vars-def tla-defs)} \\
\text{next } & \text{have } \vdash \text{pc2} = \#a \rightarrow \text{Enabled } n2 \\
& \quad \text{unfolding enab-n2[int-rewrite] by (int-rewrite simp: tla-defs)} \\
& \quad \text{hence } \vdash \Box (\text{pc2} = \#a) \rightarrow \Box \text{Enabled } n2 \\
& \quad \text{by (rule lift-imp-trans[OF ax1])} \\
& \quad \text{hence } \vdash \Box (\text{pc2} = \#a) \rightarrow \Box \text{Enabled } n2 \\
& \quad \text{by (rule lift-imp-trans[OF - E3])} \\
& \quad \text{thus } \vdash \Box (\text{pc2} = \#a) \land \Box (n1 \lor n2) \land \neg \beta2 \land SF(n2) \land \Box (I \land SF(n1)) \rightarrow \Box \text{Enabled } n2 \\
& \quad \text{by auto} \\
\end{align*}
\]

We establish a suitable leadsto-chain.
let \( ?G = LIFT \Box[(n1 \lor n2) \land \lnot \beta2]-vars \land SF(n1)-vars \land \Box(\text{pc2} = \#a \land I) \)

have \( ?G \rightarrow \Diamond(\text{pc1} = \#a \land \text{pc2} = \#a \land I) \)

proof -

Rule SF1 takes us from \( \text{pc1} = b \) to \( \text{pc1} = g \).

have bg1: \( ?G \rightarrow (\text{pc1} = \#b \rightarrow \text{pc1} = \#g) \)

proof (rule SF1)

show \( \lnot \Diamond(\text{pc1} = \#b \land [\langle (n1 \lor n2) \land \lnot \beta2 \rangle -vars \rightarrow \Diamond(\text{pc1} = \#b) \lor \Diamond(\text{pc1} = \#a) \)\)

by (auto simp: Sact2-defs vars-def tla-defs)

next

show \( \lnot \Diamond(\text{pc1} = \#b \land \langle (n1 \lor n2) \land \lnot \beta2 \rangle \land n1\)-vars \rightarrow \Diamond(\text{pc1} = \#b) \)

by (auto simp: n1-def alpha1-def beta1-def gamma1-def vars-def tla-defs)

next

show \( \lnot \Diamond(\text{pc1} = \#b \land \lnot \langle (n1 \lor n2) \land \lnot \beta2 \rangle \land \text{pc2} = \#a \land I) \)

by auto

qed

Similarly, \( \text{pc1} = b \) leads to \( \text{pc1} = g \).

have g1: \( ?G \rightarrow (\text{pc1} = \#g \rightarrow \text{pc1} = \#a) \)

proof (rule SF1)

show \( \lnot \Diamond(\text{pc1} = \#g \land [\langle (n1 \lor n2) \land \lnot \beta2 \rangle -vars \rightarrow \Diamond(\text{pc1} = \#g) \lor \Diamond(\text{pc1} = \#a) \)\)

by (auto simp: Sact2-defs vars-def tla-defs)

next

show \( \lnot \Diamond(\text{pc1} = \#g \land \lnot \langle (n1 \lor n2) \land \lnot \beta2 \rangle \land \text{pc2} = \#a \land I) \)

by (auto simp: n1-def alpha1-def beta1-def gamma1-def vars-def tla-defs)

next

show \( \lnot \Diamond(\text{pc1} = \#g \land Unchanged vars \rightarrow \Diamond(\text{pc1} = \#g) \)\)

by (auto simp: vars-def tla-defs)

next

have \( \text{pc1} = \#g \rightarrow Enabled \langle n1 \rangle\)-vars

unfolding enab-n1[int-rewrite] by (auto simp: tla-defs)

thus \( \Diamond Enabled \langle n1 \rangle\)-vars

by auto

qed
hence ⊢ □(pc1 = #g) → Enabled ⟨n1⟩-vars
by (rule lift-imp-trans[OF ax1])
hence ⊢ □(pc1 = #g) → Enabled ⟨n1⟩-vars
by (rule lift-imp-trans[OF - E3])
thus ⊢ □(pc1 = #g) ∧ □[(n1 ∨ n2) ∧ ¬ beta2]-vars ∧ □(pc2 = #a ∧ I)
    → ∨ Enabled ⟨n1⟩-vars
by auto
qed
with by1 have ⊢ ?G → ($pc1 = #a ∨ $pc1 = #b ∨ $pc1 = #g) →
  unfolding LT17[int-rewrite] LT1[int-rewrite] by force
moreover
have ⊢ $pc1 = #a ∨ $pc1 = #b ∨ $pc1 = #g
proof (clarsimp simp: tla-defs)
  fix s :: state seq
  assume pc1 (s 0) \neq a and pc1 (s 0) \neq g
  thus pc1 (s 0) = b by (cases pc1 (s 0)) auto
qed
hence ⊢ (($pc1 = #a ∨ $pc1 = #b ∨ $pc1 = #g) → $pc1 = #a) →
  ⊢ ?thesis by (auto intro!: SE3[unlift-rule])
qed
moreover
have ⊢ ?G → □ Enabled ⟨n2⟩-vars
  unfolding enab-n2[int-rewrite] by (rule STL4-eve) (auto simp: I-def tla-defs)
ultimately
show ⊢ □($pc2 = #a) ∧ □[SF(n1) ∧ SF(n1)]-vars → □ Enabled ⟨n2⟩-vars
    → △ Enabled ⟨n2⟩-vars
by (force simp: STL5[int-rewrite])
qed
from ga ab have ⊢ ?F → ($pc2 = #g → $pc2 = #b)
by (force elim: LT13[unlift-rule])
with ab have ⊢ ?F → (($pc2 = #a ∨ $pc2 = #b ∨ $pc2 = #g) → $pc2 = #b)
    unfolding LT17[int-rewrite] LT1[int-rewrite] by force
moreover
have ⊢ $pc2 = #a ∨ $pc2 = #b ∨ $pc2 = #g
proof (clarsimp simp: tla-defs)
  fix s :: state seq
  assume pc2 (s 0) \neq a and pc2 (s 0) \neq g
  thus pc2 (s 0) = b by (cases pc2 (s 0)) auto
qed

hence ⊢ (($pc2 = #a ∨ $pc2 = #b ∨ $pc2 = #g) ⇒ $pc2 = #b) → $pc2 = #b) →

by (rule fmp[OF - LT4])

ultimately show ?thesis by (rule lift-imp-trans)

qed

with 1 show ?thesis by force

qed

with psiI show ?thesis unfolding psi-def Live2-def STL5[int-rewrite] by force

qed

We can now prove the main theorem, which states that psi implements phi.

theorem (in Secondprogram) impl: ⊢ psi → phi

unfolding phi-def Live-def

by (auto dest: step-simulation[unlift-rule]
    lift-imp-trans[OF psi-fair-m1 SF-imp-WF, unlift-rule]
    lift-imp-trans[OF psi-fair-m2 SF-imp-WF, unlift-rule])

end

10 Refining a Buffer Specification

theory Buffer

imports State

begin

We specify a simple FIFO buffer and prove that two FIFO buffers in a row implement a FIFO buffer.

10.1 Buffer specification

The following definitions all take three parameters: a state function representing the input channel of the FIFO buffer, another representing the internal queue, and a third one representing the output channel. These parameters will be instantiated later in the definition of the double FIFO.

definition BInit :: 'a statefun ⇒ 'a list statefun ⇒ 'a statefun ⇒ temporal

where BInit ic q oc ≡ TEMP $q = #[]
    ∧ $ic = $oc — initial condition of buffer

definition Enq :: 'a statefun ⇒ 'a list statefun ⇒ 'a statefun ⇒ temporal

where Enq ic q oc ≡ TEMP ic$ ≠ $ic
    ∧ q$ = $q @ [ ic$ ]
    ∧ oc$ = $oc — enqueue a new value

definition Deq :: 'a statefun ⇒ 'a list statefun ⇒ 'a statefun ⇒ temporal

where Deq ic q oc ≡ TEMP # 0 < length$q>
\[ oc = hd<q> \]
\[ q = tl<q> \]
\[ ic = ic \quad \text{— dequeue value at front} \]

**definition** \textit{Nxt} :: \textquote{\texttt{a statefun}} \Rightarrow \textquote{\texttt{a list statefun}} \Rightarrow \textquote{\texttt{a statefun}} \Rightarrow \textquote{\texttt{temporal}}

**where** \textit{Nxt ic q oc} \equiv TEMP (\textit{Enq ic q oc} \lor \textit{Deq ic q oc})

— internal specification with buffer visible

**definition** \textit{ISpec} :: \textquote{\texttt{a statefun}} \Rightarrow \textquote{\texttt{a list statefun}} \Rightarrow \textquote{\texttt{a statefun}} \Rightarrow \textquote{\texttt{temporal}}

**where** \textit{ISpec ic q oc} \equiv TEMP BInit ic q oc
\[ \land \Box[Nxt ic q oc]-\{ic,q,oc\} \]
\[ \land WF(Deq ic q oc)-(ic,q,oc) \]

— external specification: buffer hidden

**definition** \textit{Spec} :: \textquote{\texttt{a statefun}} \Rightarrow \textquote{\texttt{a statefun}} \Rightarrow \textquote{\texttt{temporal}}

**where** \textit{Spec ic oc} \equiv TEMP (\exists \exists \ q. \ ISpec ic q oc)

10.2 Properties of the buffer

The buffer never enqueues the same element twice. We therefore have the following invariant:

- any two subsequent elements in the queue are different, and the last element in the queue is different from the value of the output channel,
- if the queue is non-empty then the last element in the queue is the value that appears on the input channel,
- if the queue is empty then the values on the output and input channels are equal.

The following auxiliary predicate \textit{noreps} is true if no two subsequent elements in a list are identical.

**definition** \textit{noreps} :: \textquote{\texttt{a list}} \Rightarrow \textquote{\texttt{bool}}

**where** \textit{noreps xs} \equiv \forall i < length xs - 1. xs!i \neq xs!(Suc i)

**definition** \textit{BInv} :: \textquote{\texttt{a statefun}} \Rightarrow \textquote{\texttt{a list statefun}} \Rightarrow \textquote{\texttt{a statefun}} \Rightarrow \textquote{\texttt{temporal}}

**where** \textit{BInv ic q oc} \equiv TEMP List.last\<oc> = ic \land \textit{noreps}<\oc> \land \textit{WF}(\textit{Deq ic q oc})-(ic,q,oc)

**lemmas** buffer-defs = BInit-def Enq-def Deq-def Nxt-def
Spec-def Spec-def BInv-def

**lemma** \textit{ISpec-stutinv}: \textquote{\texttt{STUTINV}} (ISpec ic q oc)

**unfolding** buffer-defs by (simp add: bothstutinvs livestutinv)

**lemma** \textit{Spec-stutinv}: \textquote{\texttt{STUTINV}} Spec ic oc

**unfolding** buffer-defs by (simp add: bothstutinvs livestutinv eczSTUT)

A lemma about lists that is useful in the following
lemma tl-self-iff-empty[simp]: \((tl \; xs = xs) = (xs = [])\)
proof
  assume 1: \(tl \; xs = xs\)
  show \(xs = []\)
  proof (rule ccontr)
    assume \(xs \neq []\) with 1 show False
    by (auto simp: neq-Nil-conv)
  qed
qed (auto)

lemma tl-self-iff-empty'[simp]: \((xs = tl \; xs) = (xs = [])\)
proof
  assume 1: \(xs = tl \; xs\)
  show \(xs = []\)
  proof (rule ccontr)
    assume \(xs \neq []\) with 1 show False
    by (auto simp: neq-Nil-conv)
  qed
qed (auto)

lemma Deq-visible:
  assumes \(\vdash Unchanged \; v \rightarrow Unchanged \; q\)
  shows \(\sim <Deq \; ic \; q \; oc>-v = Deq \; ic \; q \; oc\)
proof (auto simp: tla-defs)
  fix \(w\)
  assume \(deq: w \models Deq \; ic \; q \; oc\) and \(unch: v \; (w \; (Suc \; 0)) = v \; (w \; 0)\)
  from \(unch v[\text{unlifted}]\) have \(q \; (w \; (Suc \; 0)) = q \; (w \; 0)\)
    by (auto simp: tla-defs)
  with \(deq\) show False
    by (auto simp: Deq-def tla-defs)
qed

lemma Deq-enabledE: \(\vdash Enabled <Deq \; ic \; q \; oc>-\langle ic, q, oc \rangle \rightarrow q \sim = #[]\)
by (auto elim!: enabledE simp: Deq-def tla-defs)

We now prove that \(BInv\) is an invariant of the Buffer specification.
We need several lemmas about \(noreps\) that are used in the invariant proof.

lemma noreps-empty [simp]: \(noreps []\)
  by (auto simp: noreps-def)

lemma noreps-singleton: \(noreps [x]\) — special case of following lemma
  by (auto simp: noreps-def)

lemma noreps-cons [simp]:
  \(noreps \; (x \# xs) = (noreps \; xs \land (xs = [] \lor x \neq \text{hd} \; xs))\)
proof (auto simp: noreps-singleton)
  assume cons: \(noreps \; (x \# xs)\)
  show \(noreps \; xs\)
  proof (auto simp: noreps-def)
    fix \(i\)
  qed
assume \( i \): \( i < \text{length} \; xs - \text{Suc} \; 0 \) and eq: \( \text{xs}!i = \text{xs}!\text{Suc} \; i \)
from \( i \) have \( \text{Suc} \; i < \text{length} \; (\text{xs} \# \text{xs}) - 1 \) by auto
moreover
from eq have \( (\text{xs} \# \text{xs})!\text{Suc} \; i = (\text{xs} \# \text{xs})!\text{Suc} \; (\text{Suc} \; i) \) by auto
moreover
note cons
ultimately show False by (auto simp: noreps-def)
qed
next
assume 1: noreps \( \text{hd} \; \text{xs} \# \text{xs} \) and 2: \( \text{xs} \neq [] \)
from 2 obtain \( x \; \text{xxs} \) where \( \text{xs} = x \# \text{xxs} \) by (cases \( \text{xs} \), auto)
with 1 show False by (auto simp: noreps-def)
next
assume 1: noreps \( \text{xs} \) and 2: \( x \neq \text{hd} \; \text{xs} \)
show noreps \( x \# \text{xs} \)
proof (auto simp: noreps-def)
fix \( i \)
assume \( i \): \( i < \text{length} \; \text{xs} \) and eq: \( (\text{xs} \# \text{xs})!i = \text{xs}!i \)
from \( i \) obtain \( y \; \text{ys} \) where \( \text{xs} = y \# \text{ys} \) by (cases \( \text{xs} \), auto)
show False
proof (cases \( i \))
assume \( i \): \( i = 0 \)
with eq 2 \( \text{xs} \) show False by auto
next
fix \( k \)
assume \( k \): \( i = \text{Suc} \; k \)
with \( i \) eq \( \text{xs} \) 1 show False by (auto simp: noreps-def)
qed
qed

lemma noreps-append [simp]:
noreps \( \text{xs} @ \text{ys} \) =
(\text{noreps} \; \text{xs} \land \text{noreps} \; \text{ys} \land (\text{xs} = [] \lor \text{ys} = [] \lor \text{List.last} \; \text{xs} \neq \text{hd} \; \text{ys}))
proof auto
assume 1: noreps \( \text{xs} @ \text{ys} \)
show noreps \( \text{xs} \)
proof (auto simp: noreps-def)
fix \( i \)
assume \( i \): \( i < \text{length} \; \text{xs} - \text{Suc} \; 0 \) and eq: \( \text{xs}!i = \text{xs}!\text{Suc} \; i \)
from \( i \) have \( i < \text{length} \; (\text{xs} @ \text{ys}) - \text{Suc} \; 0 \) by auto
moreover
from \( i \) eq have \( (\text{xs} @ \text{ys})!i = (\text{xs} @ \text{ys})!\text{Suc} \; i \) by (auto simp: nth-append)
moreover note 1
ultimately show False by (auto simp: noreps-def)
qed
next
assume 1: noreps \( \text{xs} @ \text{ys} \)
show noreps \( \text{ys} \)
proof (auto simp: noreps-def)
  fix i
  assume i: i < length ys - Suc 0 and eq: ys!i = ys!(Suc i)
  from i have i + length xs < length (xs @ ys) = Suc 0 by auto
  moreover
  from i eq have (xs @ ys)!(i+length xs) = (xs@ys)!(Suc (i + length xs))
    by (auto simp: nth-append)
  moreover note 1
  ultimately show False by (auto simp: noreps-def)
qed

next
  assume 1: noreps (xs @ ys) and 2: xs ≠ [] and 3: ys ≠ []
  and 4: List.last xs = hd ys
  from 2 obtain x xxs where xs: xs = x # xxs by (cases xs, auto)
  from 3 obtain y yys where ys: ys = y # yys by (cases ys, auto)
  from xs ys have 5: length xxs < length (xs @ ys) - 1 by auto
  from 4 xs ys have (xs @ ys) ! (length xxs) = (xs @ ys) !(Suc (length xxs))
    by (auto simp: nth-append last-conv-nth)
  with 5 1 show False by (auto simp: noreps-def)

next
  assume 1: noreps xs and 2: noreps ys and 3: List.last xs ≠ hd ys
  show noreps (xs @ ys)
    proof (cases xs = [] ∨ ys = [])
      case True
      with 1 2 show ?thesis by auto
    next
      case False
      then obtain x xxs where xs: xs = x # xxs by (cases xs, auto)
      from False obtain y yys where ys: ys = y # yys by (cases ys, auto)
      show ?thesis
        proof (auto simp: noreps-def)
          fix i
          assume i: i < length xs + length ys - Suc 0
          and eq: (xs @ ys)!i = (xs @ ys)!(Suc i)
          show False
            proof (cases i < length xxs)
              case True
              hence i < length (x # xxs) by simp
              hence xsi: ((x # xxs) @ ys)!i = (x # xxs)!i
                unfolding nth-append by simp
              from True have (xs @ ys)!i = xxs!i by (auto simp: nth-append)
              with True xsi eq 1 xs show False by (auto simp: noreps-def)
            next
              assume i2: ¬(i < length xxs)
              show False
                proof (cases i = length xxs)
                  case True
                  with xs have xsi: (xs @ ys)!i = List.last xs
                    by (auto simp: nth-append last-conv-nth)
              qed
            qed
          qed
        qed
      qed
    qed
  qed
from True xs ys have (xs @ ys)! (Suc i) = y by (auto simp: nth-append)
with 3 ys eq xsi show False by simp
next
case False
with i2 xs have xsi: ¬(i < length xs) by auto
hence (xs @ ys)!i = ys!(i - length xs)
  by (simp add: nth-append)
moreover
from xsi have Suc i - length xs = Suc (i - length xs) by auto
with xsi have (xs @ ys)! (Suc i) = ysi!(Suc (i - length xs))
  by (simp add: nth-append)
moreover
from i xsi have i - length xs < length ys - 1 by auto
with 2 have ysi!(i - length xs) ≠ ysi!(Suc (i - length xs))
  by (auto simp: noreps-def)
moreover
note eq
ultimately show False by simp
qed
qed
qed
qed
qed

lemma ISpec-BInv-lemma:
  ⊢ BInit ic q oc ∧ □[Nxt ic q oc]- (ic,q,oc) → □(BInv ic q oc)
proof (rule invmono)
show ⊢ BInit ic q oc → BInv ic q oc
  by (auto simp: BInit-def BInv-def)
next
have enq: [¬ Enq ic q oc → BInv ic q oc → □(BInv ic q oc)]
  by (auto simp: Enq-def BInv-def tla-defs)
have deq: [¬ Deq ic q oc → BInv ic q oc → □(BInv ic q oc)]
  by (auto simp: Deq-def BInv-def tla-defs neq-Nil-conv)
have unch: [¬ Unchanged (ic,q,oc) → BInv ic q oc → □(BInv ic q oc)]
  by (auto simp: BInv-def tla-defs)
show [¬ BInv ic q oc ∧ [Nxt ic q oc]- (ic, q, oc) → □(BInv ic q oc)]
  by (auto simp: Nxt-def actrans-def
         elim: enq[unlift-rule] deq[unlift-rule] unch[unlift-rule])
qed

theorem ISpec-BInv: ⊢ ISpec ic q oc → □(BInv ic q oc)
proof (auto simp: ISpec-def intro: ISpec-BInv-lemma[unlift-rule])

10.3 Two FIFO buffers in a row implement a buffer

locale DBuffer =
  fixes inp :: 'a statefun --- input channel for double FIFO
We need to specify the behavior of two FIFO buffers in a row. Intuitively, that specification is just the conjunction of two buffer specifications, where the first buffer has input channel \( \text{inp} \) and output channel \( \text{mid} \) whereas the second one receives from \( \text{mid} \) and outputs on \( \text{out} \). However, this conjunction allows a simultaneous enqueue action of the first buffer and dequeue of the second one. It would not implement the previous buffer specification, which excludes such simultaneous enqueuing and dequeuing (it is written in “interleaving style”). We could relax the specification of the FIFO buffer above, which is esthetically pleasant, but non-interleaving specifications are usually hard to get right and to understand. We therefore impose an interleaving constraint on the specification of the double buffer, which requires that enqueuing and dequeuing do not happen simultaneously.

**definition DBSpec**

\[
DBSpec \equiv \text{TEMP ISpec inp q1 mid} \\
\quad \land \text{ISpec mid q2 out} \\
\quad \land \Box [\neg (\text{Enq inp q1 mid} \land \text{Deq mid q2 out})]-\vars
\]

The proof rules of TLA are geared towards specifications of the form \( \text{Init} \land \Box [\text{Next}]-\vars \land L \), and we prove that \( DBSpec \) corresponds to a specification in this form, which we now define.

**definition FullInit**

\[
\text{FullInit} \equiv \text{TEMP (BInit inp q1 mid} \land \text{BInit mid q2 out)}
\]

**definition FullNxt**

\[
\text{FullNxt} \equiv \text{TEMP (Enq inp q1 mid} \land \text{Unchanged (q2,out)} \\
\quad \lor \text{Deq inp q1 mid} \land \text{Enq mid q2 out} \\
\quad \lor \text{Deq mid q2 out} \land \text{Unchanged (inp,q1)}
\]

**definition FullSpec**

\[
\text{FullSpec} \equiv \text{TEMP FullInit} \\
\quad \land \Box [\text{FullNxt}]-\vars \\
\quad \land \text{WF(Deq inp q1 mid)-vars} \\
\quad \land \text{WF(Deq mid q2 out)-vars}
\]

The concatenation of the two queues will serve as the refinement mapping.

**definition qc**

\[
qc \equiv \text{LIFT (q2 @ q1)}
\]
lemmas db-defs = buffer-defs DBSpec-def FullInit-def FullNxt-def FullSpec-def
cq-def vars-def

lemma DBSpec-statinv: STUTINV DBSpec  
unfolding db-defs by (simp add: bothstatinvs livestatinv)

lemma FullSpec-statinv: STUTINV FullSpec  
unfolding db-defs by (simp add: bothstatinvs livestatinv)

We prove that DBSpec implies FullSpec. (The converse implication also holds but is not needed for our implementation proof.)

The following lemma is somewhat more bureaucratic than we’d like it to be. It shows that the conjunction of the next-state relations, together with the invariant for the first queue, implies the full next-state relation of the combined queues.

lemma DBNxt-then-FullNxt:  
\( \vdash \Box \text{BInv inp q1 mid} \)
\( \land \Box [\text{Nxt inp q1 mid}]-\{(\text{inp,q1,mid})\} \)
\( \land \Box [\text{Nxt mid q2 out}]-\{(\text{mid,q2,out})\} \)
\( \land \Box [\neg (\text{Enq inp q1 mid} \land \text{Deq mid q2 out})]-\text{vars} \)
\( \longrightarrow \Box [\text{FullNxt}]-\text{vars} \)

(is \( \vdash \Box \text{inv} \land \Box \text{nxts} \longrightarrow \Box [\text{FullNxt}]-\text{vars} \))

proof –

have \( \vdash \Box [\text{Nxt inp q1 mid}]-\{(\text{inp,q1,mid})\} \)
\( \land \Box [\text{Nxt mid q2 out}]-\{(\text{mid,q2,out})\} \)
\( \longrightarrow \Box [\text{Nxt inp q1 mid}]-\{(\text{inp,q1,mid}),(\text{mid,q2,out})\} \)

(is \( \vdash \text{tmp} \longrightarrow \Box [\text{?b1b2}]-\text{?vs} \))

by (auto simp: M12[int-rewrite])

moreover

have \( \vdash \Box [\text{?b1b2}]-\text{?vs} \longrightarrow \Box [\text{?b1b2}]-\text{?vs} \)

by (rule R1, auto simp: vars-def tla-defs)

ultimately

have 1: \( \vdash \Box [\text{Nxt inp q1 mid}]-\{(\text{inp,q1,mid})\} \)
\( \land \Box [\text{Nxt mid q2 out}]-\{(\text{mid,q2,out})\} \)
\( \longrightarrow \Box [\text{Nxt inp q1 mid}]-\{(\text{inp,q1,mid}),(\text{mid,q2,out})\} \)

by force

have 2: \( \vdash \Box [\text{?b1b2}]-\text{vars} \land \Box [\neg (\text{Enq inp q1 mid} \land \text{Deq mid q2 out})]-\text{vars} \)
\( \longrightarrow \Box [\text{?b1b2} \land \neg (\text{Enq inp q1 mid} \land \text{Deq mid q2 out})]-\text{vars} \)

(is \( \vdash \text{tmp2} \longrightarrow \Box [\text{?mid}]-\text{vars} \))

by (simp add: M8[int-rewrite])

have \( \vdash \Box \text{inv} \longrightarrow \# \text{True} \) by auto

moreover

have \( \Box \neg \text{inv} \longrightarrow [\text{FullNxt}]-\text{vars} \)

proof –

have \( \Box \neg \text{inv} \land \Box \text{mid} \longrightarrow [\text{FullNxt}]-\text{vars} \)

100
proof –

have A: \[ Nxt \text{ inp } q_1 \text{ mid} \]
\[ \quad \rightarrow [Nxt \text{ mid } q_2 \text{ out}]-(mid,q_2,\text{out}) \]
\[ \quad \rightarrow \neg(\text{Enq } \text{ inp } q_1 \text{ mid } \land \text{Deq } \text{ mid } q_2 \text{ out}) \]
\[ \quad \rightarrow \neg(\text{Enq } \text{ inp } q_1 \text{ mid } \land \text{Deq } \text{ mid } q_2 \text{ out}) \]
\[ \quad \rightarrow \text{FullNxt} \]

proof –

have enq: \[ \neg \text{ Enq } \text{ inp } q_1 \text{ mid} \land \neg(\text{Deq } \text{ mid } q_2 \text{ out}) \rightarrow \text{Unchanged} \]
by (auto simp: db-defs tla-defs)

have deq1: \[ \neg \text{ Deq } \text{ inp } q_1 \text{ mid} \rightarrow \text{Unchanged} \text{ q2 out} \]
by (auto simp: Deq-def BInv-def)

have deq2: \[ \neg \text{ Deq } \text{ mid } q_2 \text{ out} \rightarrow \text{mid} \neq \text{mid} \]
by (auto simp: Deq-def)

have deq: \[ \neg \text{ Deq } \text{ inp } q_1 \text{ mid } \land \neg(\text{Deq } \text{ mid } q_2 \text{ out}) \rightarrow \text{Unchanged} \text{ vars} \]
by (auto simp: vars-def tla-defs)

with enq show ?thesis by (force simp: Nxt-def FullNxt-def)

qed

have B: \[ \neg \text{ Nxt } \text{ mid } q_2 \text{ out} \]
\[ \quad \rightarrow \text{Unchanged} \text{ (inp,q1,mid)} \]
\[ \quad \rightarrow \text{FullNxt} \]
by (auto simp: db-defs tla-defs)

have C: \[ \vdash \text{Unchanged} \text{ (inp,q1,mid)} \]
\[ \quad \rightarrow \text{Unchanged} \text{ (mid,q2,out)} \]
\[ \quad \rightarrow \text{Unchanged} \text{ vars} \]
by (auto simp: vars-def tla-defs)

show ?thesis
by (force simp: actrans-def

qed

thus ?thesis by (auto simp: tla-defs)

qed

ultimately

have \[ \vdash \square ?inv \land \square[?mid]-\text{vars} \rightarrow \square \# \text{True} \land \square[\text{FullNxt}]-\text{vars} \]
by (rule TLA2)

with 1 2 show ?thesis by force

qed

It is now easy to show that DBSpec refines FullSpec.

theorem DBSpec-impl-FullSpec: \[ DBSpec \rightarrow FullSpec \]
proof –

have 1: \[ DBSpec \rightarrow FullInit \]
by (auto simp: DBSpec-def FullInit-def ISpec-def)
We now prove that two FIFO buffers in a row (as specified by formula Full-Spec) implement a FIFO buffer whose internal queue is the concatenation of the two buffers. We start by proving step simulation.

**Lemma FullInit:** \( \text{FullInit} \vdash \text{BInit} \text{inp q1 out} \)

**Proof** by (auto simp: db-defs tla-defs)

**Lemma Full-step-simulation:**

\[ \vdash [\text{FullNxt}]\text{-vars} \rightarrow [\text{Nxt} \text{inp q1 out}]\text{-}(\text{inp,qc, out}) \]
The liveness condition requires that the combined buffer eventually performs a `Deq` action on the output channel if it contains some element. The idea is to use the fairness hypothesis for the first buffer to prove that in that case, eventually the queue of the second buffer will be non-empty, and that it must therefore eventually dequeue some element.

The first step is to establish the enabledness conditions for the two `Deq` actions of the implementation.

```latex
lemma Deq1-enabled: \vdash \text{Enabled } \langle \text{Deq inp q1 mid} \rangle \text{-vars } = (\$q1 \neq #[]) 

proof
  have 1: \sim\langle \text{Deq inp q1 mid} \rangle \text{-vars } = \text{Deq inp q1 mid}
    by (rule Deq-visible, auto simp: vars-def tla-defs)

  have \vdash \text{Enabled } (\text{Deq inp q1 mid}) = (\$q1 \neq #[])
    by (force simp: Deq-def tla-defs vars-def intro: base-enabled[OF DB-base] elim!: enabledE)

  thus \thesis by (simp add: 1[int-rewrite])
qed
```

```latex
lemma Deq2-enabled: \vdash \text{Enabled } \langle \text{Deq mid q2 out} \rangle \text{-vars } = (\$q2 \neq #[]) 

proof
  have 1: \sim \langle \text{Deq mid q2 out} \rangle \text{-vars } = \text{Deq mid q2 out}
    by (rule Deq-visible, auto simp: vars-def tla-defs)

  have \vdash \text{Enabled } (\text{Deq mid q2 out}) = (\$q2 \neq #[])
    by (force simp: Deq-def tla-defs vars-def intro: base-enabled[OF DB-base] elim!: enabledE)

  thus \thesis by (simp add: 1[int-rewrite])
qed
```

We now use rule `WF2` to prove that the combined buffer (behaving according to specification `FullSpec`) implements the fairness condition of the single buffer under the refinement mapping.

```latex
lemma Full-fairness:
  \vdash \Box \langle \text{FullNxt} \rangle \land WF(\text{Deq mid q2 out}) \land WF(\text{Deq inp q1 mid}) \rightarrow WF(\text{Deq inp qc out})-(\text{inp,qc,out})

proof (rule WF2)
  -- the helpful action is the `Deq` action of the second queue
  show \sim(\langle \text{FullNxt} \land \text{Deq mid q2 out} \rangle \text{-vars } 
    \rightarrow (\text{Deq inp qc out})-(\text{inp,qc,out}))
    by (auto simp: db-defs tla-defs)

next
  -- the helpful condition is the second queue being non-empty
  show \sim(\$q2 \neq #[]) \land \Box(\$q2 \neq #[]) \land (\langle \text{FullNxt} \land \text{Deq mid q2 out} \rangle \text{-vars } 
    \rightarrow \text{Deq mid q2 out})
    by (auto simp: tla-defs)

next
  show \vdash \$q2 \neq #[] \land \text{Enabled } (\text{Deq inp qc out})-(\text{inp, qc, out})
    \rightarrow \text{Enabled } (\text{Deq mid q2 out})\text{-vars }

unfolding Deq2-enabled[int-rewrite] by auto
```

103
The difficult part of the proof is to show that the helpful condition will eventually always be true provided that the combined dequeue action is eventually always enabled and that the helpful action is never executed. We prove that (1) the helpful condition persists and (2) that it must eventually become true.

have \( \vdash \Box \Box [\text{FullNxt} \land \neg (\text{Deq mid q2 out})] \)-vars
\( \rightarrow \Box (\text{q2} \neq \#[]) \rightarrow \Box (\text{q2} \neq \#[]) \)

proof (rule STL4)

have \( \neg \text{q2} \neq \#[] \land [\text{FullNxt} \land \neg (\text{Deq mid q2 out})] \)-vars
\( \rightarrow \Box (\text{q2} \neq \#[]) \)
by (auto simp: db-defs tla-defs)

from this [THEN INV1]

show \( \vdash \Box [\text{FullNxt} \land \neg \text{Deq mid q2 out}] \)-vars
\( \rightarrow (\text{q2} \neq \#[]) \rightarrow \Box (\text{q2} \neq \#[]) \)
by auto

qed

hence \( \vdash \Box [\text{FullNxt} \land \neg (\text{Deq mid q2 out})] \)-vars
\( \rightarrow (\text{q1} \neq \#[]) \rightarrow \Box (\text{q2} \neq \#[]) \)
by (force intro: E31 [unlift-rule])

have \( \vdash \Box [\text{FullNxt} \land \neg (\text{Deq mid q2 out})] \)-vars
\( \land \text{WF} (\text{Deq inp q1 mid}) \)-vars
\( \rightarrow (\text{Enabled} (\text{Deq inp qc out}) \land (\text{inp, qc, out}) \rightarrow \text{q2} \neq \#[]) \)

proof (rule WF1)

have \( \vdash \Box (\text{q1} \neq \#[]) \rightarrow (\text{q2} \neq \#[]) \)
by (auto simp: db-defs tla-defs)

show \( \vdash \neg \text{q1} \neq \#[] \land \{\text{FullNxt} \land \neg \text{Deq mid q2 out} \} \land \text{Deq inp q1 mid}\)-vars
\( \rightarrow \Box (\text{q2} \neq \#[]) \)
by (auto simp: db-defs tla-defs)

next

show \( \vdash \neg \text{q1} \neq \#[] \land \text{Unchanged vars} \rightarrow \Box (\text{q1} \neq \#[]) \)
by (auto simp: vars-def tla-defs)

next

show \( \vdash \neg \text{q1} \neq \#[] \land \text{ Enabled} (\text{Deq inp q1 mid}) \)-vars
by (simp add: Deq1-enabled[int-rewrite])

next

show \( \neg \text{q1} \neq \#[] \land \text{Unchanged vars} \rightarrow \Box (\text{q1} \neq \#[]) \)
by (auto simp: vars-def tla-defs)

qed

hence \( \vdash \Box [\text{FullNxt} \land \neg (\text{Deq mid q2 out})] \)-vars
\( \land \text{WF} (\text{Deq inp q1 mid}) \)-vars
\( \rightarrow (\text{qc} \neq \#[] \rightarrow \text{q2} \neq \#[]) \)
by (auto simp: qc[int-rewrite] LT17[int-rewrite] LT1 [int-rewrite])

moreover
have ⊢ Enabled ⟨Deq inp qc out⟩-(inp, qc, out) ↦ $qc ≠ [] by (rule Deq-enabledE[THEN LT3])
ultimately show thesis by (force elim: LT13[unlift-rule])
qed

with LT6
have ⊢ □[FullNxt ∧ ¬(Deq mid q2 out)]-vars ∧ WF(Deq inp q1 mid)-vars ∧ □ Enabled (Deq inp qc out)-(inp, qc, out) → □($q2 ≠ [])
by force with 1 E16
show ⊢ □[FullNxt ∧ ¬(Deq mid q2 out)]-vars ∧ WF(Deq mid q2 out)-vars ∧ □WF(Deq mid q1 mid)-vars ∧ □□ Enabled (Deq inp qc out)-(inp, qc, out) → □□($q2 ≠ [])
by force
qed

Putting everything together, we obtain that FullSpec refines the Buffer specification under the refinement mapping.

theorem FullSpec-impl-ISpec: ⊢ FullSpec → ISpec inp qc out
unfolding FullSpec-def ISpec-def using FullInit Full-step-simulation[THEN M11] Full-fairness
by force

theorem FullSpec-impl-Spec: ⊢ FullSpec → Spec inp out
unfolding Spec-def using FullSpec-impl-ISpec
by (force intro: eexI[unlift-rule])

By transitivity, two buffers in a row also implement a single buffer.

theorem DBSpec-impl-Spec: ⊢ DBSpec → Spec inp out
by (rule lift-imp-trans[OF DBSpec-impl-FullSpec FullSpec-impl-Spec])

end — locale DBuffer

References


