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].
1.2.2 Properties of Sequence.last and laststate .......... 7
1.3 Stuttering Invariance ........................................... 7
  1.3.1 Properties of nonstutseq ................................. 8
  1.3.2 Properties of nextnat .................................... 8
  1.3.3 Properties of nextsuffix ................................. 11
  1.3.4 Properties of next ....................................... 11
  1.3.5 Properties of ................................. 12
1.4 Similarity of Sequences ...................................... 13
  1.4.1 Properties of op ≈ ................................. 13

2 Representing Intensional Logic 19
  2.1 Abstract Syntax ............................................ 20
  2.2 Concrete Syntax ............................................ 21
  2.3 Definitions ................................................. 24
  2.4 Lemmas and Tactics ........................................ 24

3 Semantics 26
  3.1 Types of Formulas ........................................... 27
  3.2 Semantics of TLA* ........................................... 27
    3.2.1 Concrete Syntax .................................... 27
  3.3 Abbreviations .............................................. 28
    3.3.1 Concrete Syntax .................................... 28
  3.4 Properties of Operators .................................... 29
  3.5 Invariance Under Stuttering ................................ 29
    3.5.1 Properties of -stutinv ................................. 31
    3.5.2 Properties of -nstutinv ................................. 34
    3.5.3 Abbreviations ........................................ 35

4 Reasoning about PreFormulas 36
  4.1 Lemmas about Unchanged ...................................... 37
  4.2 Lemmas about after ........................................ 38
  4.3 Lemmas about before ..................................... 38
  4.4 Some general properties .................................... 39
  4.5 Unlifting attributes and methods .......................... 39

5 A Proof System for TLA* 40
  5.1 The Basic Axioms ........................................... 41
  5.2 Derived Theorems ........................................... 43
  5.3 Some other useful derived theorems ....................... 46
  5.4 Theorems about the eventually operator .................. 53
  5.5 Theorems about the leadsto operator ..................... 56
  5.6 Lemmas about the next operator ........................... 63
  5.7 Higher Level Derived Rules ............................... 65
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 \( \| \) 60)
where \( s \| i \equiv \lambda n. s (n+i) \)

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

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

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

1.1.1 Properties of \textit{first} and \textit{second}

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

1.1.2 Properties of \( s \| \)

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

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

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

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

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

**lemma** suffix-zero[simp]: \( s \| 0 = s \)
by (simp add: suffix-def)

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

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

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: $n > 0 \Rightarrow (s \#\# \sigma) \ n = \sigma (n - 1)$
  by (simp add: app-def)

1.2 Finite and Empty Sequences

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

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

abbreviation inf :: $(\alpha \ seq) \Rightarrow bool$
where inf $s \equiv \neg (\text{fin } s)$

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

definition laststate :: $(\alpha \ seq) \Rightarrow \alpha$
where laststate $s \equiv s (\text{last } s)$

definition emptyseq :: $(\alpha \ seq) \Rightarrow bool$
where emptyseq $s \equiv \lambda \ i. s \ i = s \ 0$

abbreviation notemptyseq :: $(\alpha \ seq) \Rightarrow bool$
where notemptyseq $s \equiv \neg (\text{emptyseq } s)$

Predicate fin holds if there is an element in the sequence such that all sub-
sequent 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 re-
turned. 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 \textit{Sequence}.last and laststate

lemma \textit{fin-stut-after-last}: assumes $H$: $\text{fin } s$ 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: \text{fin-def})
hence $\exists P \ (\text{last } s) \ \text{unfolding last-def by (rule LeastI)}$
with $j$ show $s j = s \text{ (last } s)$ by blast
qed

1.3 Stuttering Invariance

This subsection provides functions for removing stuttering steps of sequences, i.e. we formalise Lamport’s $\sharp$ operator. Our formal definition is close to that of Wahab in the PVS prover.
The key novelty with the \textit{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 \cite{Lamport1998} method is used to mechanise the equality of sequences up to stuttering: he defines the $\sharp$ operator, which collapses a sequence by removing all stuttering steps, except possibly infinite stuttering at the end of the sequence. These are left unchanged.

\textbf{definition} \textit{nonstutseq} :: $(\alpha \ \text{seq}) \Rightarrow \text{bool}$
where \textit{nonstutseq} $s \equiv \forall \ i. \ s i = s \ (\text{Suc } i) \longrightarrow (\forall \ j > i. \ s i = s \ j)$

\textbf{definition} \textit{stutstep} :: $(\alpha \ \text{seq}) \Rightarrow \text{nat} \Rightarrow \text{bool}$
where \textit{stutstep} $s \ n \equiv (s \ n = s \ (\text{Suc } n))$

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

\textbf{definition} \textit{nextsuffix} :: $(\alpha \ \text{seq}) \Rightarrow (\alpha \ \text{seq})$
where \textit{nextsuffix} $s \equiv s |_n \ (\text{nextnat } s)$

\textbf{fun} \textit{next} :: $\text{nat} \Rightarrow (\alpha \ \text{seq}) \Rightarrow (\alpha \ \text{seq})$
where $\text{next } 0 = \text{id}$
$\mid$ next $(\text{Suc } n) = \text{nextsuffix } o \ (\text{next } n)$

\textbf{definition} \textit{collapse} :: $(\alpha \ \text{seq}) \Rightarrow (\alpha \ \text{seq})$ ($\sharp$)
where $\exists s \equiv \lambda n. \ (\text{next } n \ s) \ 0$

Predicate \textit{nonstutseq} identifies sequences without any stuttering steps – except possibly for infinite stuttering at the end. Further, \textit{stutstep} $s \ n$ is a predicate which holds if the element after $s \ n$ is equal to $s \ n$, i.e. $\text{Suc } n$ is a stuttering step. $\sharp \ s$ formalises Lamport’s $\sharp$ operator. It returns the first state of the result of $\text{next } n \ s$. $\text{next } n \ s$ finds suffix of the $n^{th}$ change. Hence
the first element, which 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 \textit{nonstutseq}

\\textbf{lemma } \textit{seq-empty-is-nonstut}:
\begin{itemize}
\item \textbf{assumes } \(H\): \(\text{emptyseq } s\) \textbf{shows } \(\text{nonstutseq } s\)
\item \textbf{using } \(H\) \textbf{by } \textit{(auto simp: nonstutseq-def seq-empty-all)}
\end{itemize}

\\textbf{lemma } \textit{notempty-exist-nonstut}:
\begin{itemize}
\item \textbf{assumes } \(H\): \(\neg \text{emptyseq } (s \mid_s m)\) \textbf{shows } \(\exists i. \ s i \neq s m \land i > m\)
\item \textbf{using } \(H\) \textbf{proof } \textit{(auto simp: emptyseq-def suffix-def)}
\item \textit{fix } \(i\)
\item \textit{assume } \(i: \ s (i + m) \neq s m\)
\item \textit{hence } \(i \neq 0\) \textbf{by } \textit{(intro notI, simp)}
\item \textit{with } \(i\) \textbf{show } \(?\text{thesis}\) \textbf{by } \textit{auto}
\end{itemize}
\textbf{qed}

1.3.2 Properties of \textit{nextnat}

\\textbf{lemma } \textit{nextnat-le-unch}:
\begin{itemize}
\item \textbf{assumes } \(H\): \(n < \text{nextnat } s\) \textbf{shows } \(s n = s \emptyset\)
\item \textbf{proof } \textit{(cases emptyseq } s)\textbf{)}
\item \textbf{assume } \(\text{emptyseq} \ s\)
\item \textit{hence } \(\text{nextnat } s = \emptyset\) \textbf{by } \textit{(simp add: nextnat-def)}
\item \textit{with } \(H\) \textbf{show } \(?\text{thesis}\) \textbf{by } \textit{auto}
\end{itemize}
\textbf{next}
\begin{itemize}
\item \textbf{assume } \(\neg \text{emptyseq} \ s\)
\item \textit{hence } \(a1: \ \text{nextnat } s = (\text{LEAST } i. \ s i \neq s \emptyset)\) \textbf{by } \textit{(simp add: nextnat-def)}
\item \textbf{show } \(?\text{thesis}\)
\item \textbf{proof } \textit{(rule ccontr)}
\item \textbf{assume } \(a2: \ s n \neq s \emptyset\) \textbf{(is } \(?P n)\text{)}
\item \textit{hence } \((\text{LEAST } i. \ s i \neq s \emptyset) \leq n\) \textbf{by } \textit{(rule Least-le)}
\item \textit{hence } \(\neg(n < (\text{LEAST } i. \ s i \neq s \emptyset))\) \textbf{by } \textit{auto}
\item \textit{also from } \(H\) \textit{ a1 have } \(n < (\text{LEAST } i. \ s i \neq s \emptyset)\) \textbf{by } \textit{simp}
\item \textit{ultimately show } \(\text{False}\) \textbf{by } \textit{auto}
\end{itemize}
\textbf{qed}

\\textbf{lemma } \textit{stutnempty}:
\begin{itemize}
\item \textbf{assumes } \(H\): \(\neg \text{stutstep } s n\) \textbf{shows } \(\neg \text{emptyseq } (s \mid_s n)\)
\item \textbf{proof } \textit{(unfold emptyseq-def suffix-def)}
\end{itemize}

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 $H$: $\neg\text{stutstep} \ s \ n$ shows $\text{nexitnat} \ (s \ |_s \ n) = 1$
proof -
  from $H$ have $h'$: $\text{nexitnat} \ (s \ |_s \ n) = (\text{LEAST} \ i. \ (s \ |_s \ n) \ i \neq (s \ |_s \ n) \ 0)$
    by (auto simp add: nexitnat-def stutnempty)
  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 $g1'$: $\text{nexitnat} \ (s \ |_s \ n) \ = \ 0 \lor \text{nexitnat} \ (s \ |_s \ n) \ = \ 1$ by auto
  also have $\text{nexitnat} \ (s \ |_s \ n) \neq 0$
proof -
  from $H$ have $\neg\text{emptyseq} \ (s \ |_s \ n)$ 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 LeastI)
  with $h'$ have $g2$: $(s \ |_s \ n) \ (\text{nexitnat} \ (s \ |_s \ n)) \neq (s \ |_s \ n) \ 0$ by auto
  show $(\text{nexitnat} \ (s \ |_s \ n)) \neq 0$
proof
  assume $(\text{nexitnat} \ (s \ |_s \ n)) = 0$
  with $g2$ show False by simp
qed
qed
ultimately show $\text{nexitnat} \ (s \ |_s \ n) = 1$ by auto
qed

lemma stutstep-notempty-notempty:
  assumes $h1$: $\text{emptyseq} \ (s \ |_s \ \text{Suc} \ n)$ (is $\text{emptyseq} \ ?sn$)
    and $h2$: $\text{stutstep} \ s \ n$
  shows $\text{emptyseq} \ (s \ |_s \ n)$ (is $\text{emptyseq} \ ?s$)
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 = $?sn \ m by (simp add: suffix-def)
  also from $h1$ have $\ldots = $?sn \ 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 stmtstep-empty-suc:
  assumes stmtstep s n
  shows emptyseq (s |s n) = emptyseq (s |s n)
using assms by (auto elim: stmtstep-notempty-notempty suc-empty)

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

lemma nextnat-empty-neq: assumes H: ¬ emptyseq s shows s (nextnat s) ≠ s 0
proof
  from H have a1: nextnat s = (LEAST i. s i ≠ s 0) by (simp add: nextnat-def)
  from H obtain i where s i ≠ s 0 by (auto simp: emptyseq-def)
  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 (nextnat s) ≠ s 0 by (rule nextnat-empty-neq)
  have nextnat s ≠ 0
    proof
      assume nextnat s = 0
      with a1 show False by simp
    qed
qed
thus \( \text{nextnat } s > 0 \) by simp

\textbf{1.3.3 Properties of nextsuffix}

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

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

\textbf{lemma notstutstep-nextsuffix1:}
\begin{itemize}
  \item assumes \( H: \neg \text{stutstep } s \ n \) shows \( \text{nextsuffix } (s \ | s \ n) = s \ | s \ (\text{Suc } n) \)
  \item proof (unfold nextsuffix-def)
  \item show \( (s \ | s \ n \ | s \ (\text{nextnat } (s \ | s \ n))) = s \ | s \ (\text{Suc } n) \)
  \item proof –
  \item from \( H \) have \( \text{nextnat } (s \ | s \ n) = 1 \) by (rule notstutstep-nexnat1)
  \item hence \( (s \ | s \ n \ | s \ (\text{nextnat } (s \ | s \ n))) = s \ | s \ n \ | s \ 1 \) by auto
  \item thus \( \text{thesis } \) by (simp add: suffix-def)
\end{itemize}

\textbf{1.3.4 Properties of next}

\textbf{lemma next-suc-suffix: \( \text{next } (\text{Suc } n) \ s = \text{nextsuffix } (\text{next } n \ s) \)}
\begin{itemize}
  \item by simp
\end{itemize}

\textbf{lemma next-suffix-com: \( \text{nextsuffix } (\text{next } n \ s) = (\text{next } n \ (\text{nextsuffix } s)) \)}
\begin{itemize}
  \item by (induct \( n \), auto)
\end{itemize}

\textbf{lemma next-plus: \( \text{next } (m+n) \ s = \text{next } m \ (\text{next } n \ s) \)}
\begin{itemize}
  \item by (induct \( m \), auto)
\end{itemize}

\textbf{lemma next-empty; assumes \( H: \text{emptyseq } s \) shows \( \text{next } n \ s = s \)}
\begin{itemize}
  \item proof (induct \( n \))
  \item from \( H \) show \( \text{next } 0 \ s = s \) by auto
  \item next
  \item fix \( n \)
  \item assume \( a1: \text{next } n \ s = s \)
  \item have \( \text{next } (\text{Suc } n) \ s = \text{nextsuffix } (\text{next } n \ s) \) by auto
  \item with \( a1 \) have \( \text{next } (\text{Suc } n) \ s = \text{nextsuffix } s \) by simp
  \item with \( H \) show \( \text{next } (\text{Suc } n) \ s = s \)
  \item by (simp add: nextsuffix-def nextnat-def)
\end{itemize}

\textbf{lemma notempty-nextnotzero:}
\begin{itemize}
  \item assumes \( H: \neg \text{emptyseq } s \) shows \( \text{next } (\text{Suc } 0) \ s \neq s \ 0 \)
  \item proof –
\end{itemize}
from $H$ have \( g1: \) \( s \) (nextnat $s$) \( \neq \) \( s \) \( 0 \) by (rule nextnat-empty-neq)

have \( \text{next} \) (Suc \( 0 \)) \( s \) = nextsuffix \( s \) by auto

hence \( \text{next} \) (Suc \( 0 \)) \( s \) = \( s \) (nextnat \( s \)) by (simp add: nextsuffix-def suffix-def)

with \( g1 \) show ?thesis by simp

qed

lemma next-ex-id: \( \exists \) \( i \). \( s \) \( i \) = (next \( m \) \( s \)) \( 0 \)

proof

have \( \exists \) \( i \). (\( s \) \( |_s \) \( i \)) = (next \( m \) \( s \))

proof (induct \( m \))

have \( s \) \( |_s \) \( 0 \) = next \( 0 \) \( s \) by simp

thus \( \exists \) \( i \). (\( s \) \( |_s \) \( i \)) = (next \( 0 \) \( s \)) ..

next

fix \( m \)

assume \( a1: \) \( \exists \) \( i \). (\( s \) \( |_s \) \( i \)) = (next \( m \) \( s \))

then obtain \( i \) where \( a1' \): (\( s \) \( |_s \) \( i \)) = (next \( m \) \( s \)) ..

have \( \text{next} \) (Suc \( m \)) \( s \) = nextsuffix (next \( m \) \( s \)) by auto

hence \( \text{next} \) (Suc \( m \)) \( s \) = (next \( m \) \( s \)) \| _s (nextnat (next \( m \) \( s \))) by (simp add: nextsuffix-def)

hence \( \exists \) \( i \). \( \text{next} \) (Suc \( m \)) \( s \) = (next \( m \) \( s \)) \| _s \( i \) ..

then obtain \( j \) where \( \text{next} \) (Suc \( m \)) \( s \) = (next \( m \) \( s \)) \| _s \( j \) ..

with \( a1' \) have \( \text{next} \) (Suc \( m \)) \( s \) = (\( s \) \( |_s \) \( i \)) \| _s \( j \) by simp

hence \( \text{next} \) (Suc \( m \)) \( s \) = (\( s \) \( |_s \) (\( j+i \))) by (simp add: suffix-plus)

hence (\( s \) \( |_s \) (\( j+i \))) = next (Suc \( m \)) \( s \) by simp

thus \( \exists \) \( i \). (\( s \) \( |_s \) \( i \)) = (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 $\natural$

lemma emptyseq-collapse-eq: assumes $A1$: emptyseq \( s \) shows $\natural$ \( s \) = \( s \)

proof (unfold collapse-def, rule ext)

fix \( n \)

from $A1$ have \( \text{next} \) \( n \) \( s \) = \( s \) 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 ($\natural$ \( s \))

using $H$ by (simp add: emptyseq-collapse-eq)

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

lemma collapse-empty-iff-empty [simp]: emptyseq ($\varepsilon$ 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 $\sigma \approx \tau \equiv (\varepsilon \sigma) = (\varepsilon \tau)$

1.4.1 Properties of op $\approx$

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 = first t
proof −
  from $H$ have ($\varepsilon$ s) $0 = (\varepsilon$ t) $\vDash 0$ 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 $\approx$ tail t shows second s = second t
proof −
  from $H$ have first (tail s) = first (tail t) by (simp add: sim-first)
  thus second s = 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 $(s \ |_s (\text{Suc } n))$ $\approx$ nexsuffix $(s \ |_s n)$
using $H$ by (simp add: notstutstep-nexsuffix1)

lemma stat-nexsuffix-suc:
assumes $H$: statstep $s \ n$ shows nexsuffix $(s \ |_s n) =$ nexsuffix $(s \ |_s (\text{Suc } n))$
proof (cases emptyseq $(s \ |_s n)$)
case True

hence g1: nexsuffix $(s \ |_s n) = (s \ |_s n)$ by (simp add: nexsuffix-def nextnat-def)

from True have g2: nexsuffix $(s \ |_s \text{Suc } n) = (s \ |_s \text{Suc } n)$
by (simp add: suc-empty nexsuffix-def nextnat-def)

have $(s \ |_s n) = (s \ |_s \text{Suc } n)$
proof

fix $x$
from True have $(s (x + n)) = (s (0 + n))$ $(\text{Suc } x + n) = (s (0 + n))$
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 ?thesis by auto

next

case False

with $H$ have (nextnat $(s \ |_s n)) = \text{Suc } (\text{nextnat } (s \ |_s \text{Suc } n))$

by (simp add: statstep-notempty-sucnextnat)

thus ?thesis
by (simp add: nexsuffix-def suffix-plus)

qed

lemma seqsim-suffix-seqsim:
assumes $H$: $s \approx t$ shows nexsuffix $s \approx$ nexsuffix $t$
unfolding seqsimilar-def collapse-def
proof

fix $n$
from $H$ have (next $(\text{Suc } n) s) 0 = (\text{next } (\text{Suc } n) t) 0$
unfolding seqsimilar-def collapse-def by (rule fun-cong)

thus next $n$ (nexsuffix $s) 0 = next n$ (nexsuffix $t) 0$
by (simp add: next-suffix-com)

qed

lemma seqsim-statstep:
assumes $H$: statstep $s \ n$ shows $(s \ |_s (\text{Suc } n)) \approx (s \ |_s n)$ (is $?)n \approx (?)s$
unfolding seqsimilar-def collapse-def
proof

fix $m$
show next $m$ $(s \ |_s \text{Suc } n) 0 = next m$ $(s \ |_s n) 0$
proof (cases $m$
assume $m=0$
with $H$ show ?thesis by (simp add: suffix-def statstep-def)

next

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

15
with $H$ have $next\ m\ (s\ |_s\ Suc\ n) = next\ k\ (nextsuffix\ (s\ |_s\ n))$
  by (simp add: stut-success 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 $(\textit{first}\ t) ## (\textit{t}) \theta$
  by (simp add: first-def stutstep-def app-def suffix-def)

lemma addfeqsim: $(\textit{first}\ t) ## (\textit{t}) \approx t$
proof
  have stutstep $(\textit{first}\ t) ## (\textit{t}) \theta$ by (rule addfeqstut)
  hence $((\textit{first}\ t) ## (\textit{t})) |_{s} (Suc\ 0) \approx (((\textit{first}\ t) ## (\textit{t})) |_{s} 0)\ |_{\theta}$
    by (rule seqsim-stutstep)
  hence $tail\ ((\textit{first}\ t) ## (\textit{t}) |_{s}) \approx ((\textit{first}\ t) ## (\textit{t}) |_{s} 0)$
    by (simp add: suffix-def tail-def)
  hence $t \approx ((\textit{first}\ t) ## (\textit{t}) |_{\theta})$ by (simp add: tail-def app-def suffix-def)
  thus ?thesis by (rule seqsim-sym)
qed

lemma addfirststat:
assumes $H$: first $s = second\ s\ shows\ s \approx tail\ s$
proof
  have $g1\ (\textit{first}\ s) ## (\textit{tail}\ s) = s$ by (rule seq-app-first-tail)
  from $H$ have $(\textit{first}\ s) = first\ (\textit{tail}\ s)$
    by (simp add: first-def second-def tail-def)
  hence $(\textit{first}\ s) ## (\textit{tail}\ s) \approx (\textit{tail}\ s)$ by (simp add: addfeqsim)
  with $g1$ show ?thesis by simp
qed

lemma app-seqsimilar:
assumes $h1$: $s \approx t\ shows\ (x ## s) \approx (x ## t)$
proof (cases stutstep $(x ## s) 0$)
case True
  from $h1$ have first $s = (\textit{first}\ t)$ by (rule sim-first)
  with True have a2: stutstep $(x ## t) 0$
    by (simp add: stutstep-def first-def app-def)
  from True have $((x ## s) |_{s} (Suc\ 0)) \approx (x ## s) |_{0}$
    by (rule seqsim-stutstep)
  hence $tail\ (x ## s) \approx (x ## s)$ by (simp add: tail-def suffix-def)
  hence $g1: s \approx (x ## s)$ by (simp add: app-def tail-def suffix-def)
  from a2 have $((x ## t) |_{s} (Suc\ 0)) \approx (x ## t) |_{0}$ by (rule seqsim-stutstep)
  hence $tail\ (x ## t) \approx (x ## t)$ by (simp add: tail-def suffix-def)
  hence $g2: t \approx (x ## t)$ by (simp add: app-def tail-def suffix-def)
  from $h1\ g2$ have $s \approx (x ## t)$ by (rule seqsim-trans)
  from this[THEN seqsim-sym] $g1$ show $(x ## s) \approx (x ## t)$
    by (rule seqsim-sym[OF seqsim-trans])
next
case False
  from $h1$ have first $s = (\textit{first}\ t)$ by (rule sim-first)
with False have a2: \( \neg \text{statstep} (x \# \# t) \ 0 \)
  by (simp add: statstep-def first-def app-def)
from False have \((x \# \# s) \ |_s (\text{Suc} \ 0)\) \(\approx\) neixtsuffix \((x \# \# s) \ |_s 0\)
  by (rule seqsim-notstutstep)
hence (tail \((x \# \# s)\) \(\approx\) neixtsuffix \((x \# \# s)\)
  by (simp add: tail-def)
hence \(g1\): \(s \approx\) neixtsuffix \((x \# \# s)\)
by (simp add: seq-app-tail)
from \(a2\) have \((x \# \# t) \ |_s (\text{Suc} \ 0)\) \(\approx\) neixtsuffix \((x \# \# t) \ |_s 0\)
  by (rule seqsim-notstutstep)
hence (tail \((x \# \# t)\) \(\approx\) neixtsuffix \((x \# \# t)\)
by (simp add: tail-def)
hence \(g2\): \(t \approx\) neixtsuffix \((x \# \# t)\)
by (simp add: seq-app-tail)
with \(h1\) have \(s \approx\) neixtsuffix \((x \# \# t)\)
by (rule seqsim-trans)
from this[THEN seqsim-sym] \(g1\) have \(g3\): neixtsuffix \((x \# \# s)\) \(\approx\) neixtsuffix \((x \# \# t)\)
  by (rule seqsim-sym[OF seqsim-trans])
have first \((x \# \# s) = first \((x \# \# t)\)
by (simp add: first-def app-def)
from this \(g3\) 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 \(H\): \(s \approx t\) shows \(\exists \ m. \ ((s \ |_s n) \approx (t \ |_s m))\)
\textbf{proof} (induct \(n\))
  from \(H\) have \((s \ |_s 0) \approx (t \ |_s 0)\)
  by auto
  thus \(\exists \ m. \ ((s \ |_s 0) \approx (t \ |_s m))\)
next
  fix \(n\)
  assume \(\exists \ m. \ ((s \ |_s n) \approx (t \ |_s m))\)
  then obtain \(m\) where \(a1': \ (s \ |_s n) \approx (t \ |_s m)\)
  show \(\exists \ m. \ ((s \ |_s (\text{Suc} \ n)) \approx (t \ |_s m))\)
  proof (cases statstep \(s\) \(n\))
    case True
    hence \((s \ |_s (\text{Suc} \ n)) \approx (s \ |_s n)\)
    by (rule seqsim-statstep)
    from this \(a1'\) have \((s \ |_s (\text{Suc} \ n)) \approx (t \ |_s m)\)
    by (rule seqsim-trans)
    thus \(?thesis\)
next
  case False
  hence \((s \ |_s (\text{Suc} \ n)) \approx \text{nextsuffix} (s \ |_s n)\)
  by (rule seqsim-notstutstep)
  moreover
  from \(a1'\) have \text{nextsuffix} \((s \ |_s n) \approx \text{nextsuffix} (t \ |_s m)\)
  by (simp add: seqsim-suffix-seqsim)
  ultimately have \((s \ |_s (\text{Suc} \ n)) \approx \text{nextsuffix} (t \ |_s m)\)
  by (rule seqsim-trans)
  hence \((s \ |_s (\text{Suc} \ n)) \approx t \ |_s (m + (\text{nextnat} (t \ |_s m)))\)
  by (simp add: nextsuffix-def suffix-plus-com)
  thus \(\exists \ m. \ (s \ |_s (\text{Suc} \ n)) \approx t \ |_s m\)
  qed
  qed

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

lemma seqsim-prev-nextnat: $s \approx s \mid s \ ((\text{nextnat } s) - 1)$
proof (cases emptyseq $s$)
case True
hence $s \mid s \Suc n \approx s \mid s \ n$ by (rule seqsim-stutstep)
from this $H$ have $s \mid s \Suc n \approx t \mid s \ Suc m$ by (rule seqsim-trans)
with $H$ show $\psi$thesis by blast
next
case False
hence $\neg \text{emptyseq } (s \mid s \ n)$ by (rule stutnempty)
with $H$ have $a2$: $\neg \text{emptyseq } (t \mid s \ m)$ by (simp add: seqsim-empty-iff-empty)

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

lemma seqsim-suffix-suc:
assumes $H$: $s \mid s \ n \approx t \mid s \ m$
shows $\exists m', \ s \mid s \ n \approx t \mid s \ m' \land ((s \mid s \ Suc n \approx t \mid s \ Suc m') \lor (s \mid s \ Suc n \approx t \mid s \ m'))$
proof (cases stutstep $s \ n$)
case True
hence $s \mid s \ Suc n \approx s \mid s \ n$ by (rule seqsim-stutstep)
from this $H$ have $s \mid s \ Suc n \approx t \mid s \ m$ by (rule seqsim-trans)
with $H$ show $\psi$thesis by blast
next
case False
hence $\neg \text{emptyseq } (s \mid s \ n)$ by (rule stutnempty)
with $H$ have $a2$: $\neg \text{emptyseq } (t \mid s \ m)$ by (simp add: seqsim-empty-iff-empty)
hence \( g_4 : \text{nextsuffix} (t \mid_s m) = (t \mid_s m) \mid_s \text{nextnat} (t \mid_s m) - 1 \)
by (simp add: nextnat-empty-gzero nextsuffix-def)

have \( g_3 : (t \mid_s m) \approx (t \mid_s m) \mid_s (\text{nextnat} (t \mid_s m) - 1) \)
by (rule seqsim-prev-nextnat)

with \( H \) have \( G_1 : s \mid_s n \approx (t \mid_s m) \mid_s (\text{nextnat} (t \mid_s m) - 1) \)
by (rule seqsim-trans)

from False have \( G_1' : (s \mid_s n) \approx (t \mid_s m) \mid_s (\text{nextnat} (t \mid_s m) - 1) \)
by (rule seqsim-suffix-seqsim)

thus \(?thesis by blast\)

qed

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

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

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

**Theorem sim-step:**

**Assumes** \( H : s \approx t \)

**Shows** \( \exists m. s \mid_s n \approx t \mid_s m \land ((s \mid_s \text{Suc} n \approx t \mid_s \text{Suc} m) \lor (s \mid_s \text{Suc} n \approx t \mid_s m)) \)

**Is** \( \exists m. ?\text{Sim} n m \)

**Proof** (induct \( n \))

from \( H \) have \( s \mid_s 0 \approx t \mid_s 0 \) by simp
thus \( \exists m. ?\text{Sim} 0 m \) by (rule seqsim-suffix-suc)

next
fix \( n \)
assume \( \exists m. ?\text{Sim} n m \)

hence \( \exists k. s \mid_s \text{Suc} n \approx t \mid_s k \) by blast
thus \( \exists m. ?\text{Sim} (\text{Suc} n) m \) by (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 \( \text{impl} \): \( ?P \Rightarrow ?Q \) \( \Rightarrow \) ?P \( \rightarrow \) ?Q. However, the deduction theorem does not hold for most modal and temporal logics [6, page 95][7]. For example \( \vdash A \) 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.

class world

world is a type class of possible worlds. It is a subclass of all HOL types type. No axioms are provided, since its only purpose is to avoid silly use of the Intensional syntax.

2.1 Abstract Syntax

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).

cons

Valid :: (’w::world) form ⇒ bool
const :: ’a ⇒ (’w::world, ’a) expr
lift :: [’a ⇒ b, (’w::world, ’a) expr] ⇒ (’w,b) expr
lift2 :: [’a ⇒ b ⇒ c, (’w::world,’a) expr, (’w,b) expr] ⇒ (’w,’c) expr
lift3 :: [’a ⇒ b ⇒ c ⇒ d, (’w::world,’a) expr, (’w,b) expr, (’w,’c) expr] ⇒ (’w,’d) expr
lift4 :: [’a ⇒ b ⇒ c ⇒ d ⇒ e, (’w::world,’a) expr, (’w,b) expr, (’w,’c) expr, (’w,’d) expr] ⇒ (’w,’e) expr

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.)

cons

RAAll :: (’a ⇒ (’w::world) form) ⇒ ’w form (binder RAll 10)
REx :: (’a ⇒ (’w::world) form) ⇒ ’w form (binder REx 10)
REx1 :: (’a ⇒ (’w::world) form) ⇒ ’w form (binder REx! 10)

RAAll, REx and REx1 introduces “rigid” quantification over values (of non-world types) within “intensional” formulas. RAAll is universal quantification, REx is existential quantification. REx1 requires unique existence.
2.2 Concrete Syntax

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

```
lift :: lift ⇒ 'a
```

The syntax ```lift``` represents lifted expressions.

```
LIFT :: lift ⇒ 'a
```

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

```
liftargs :: liftargs ⇒ lift
```

The non-terminal liftargs represents lifted arguments.

```
liftC :: liftargs ⇒ lift
```

The non-terminal liftC represents lifted continuations.

```
liftargs :: liftargs ⇒ lift
```

The non-terminal liftargs represents lifted arguments.

```
lift2 :: lift args ⇒ lift
```

The non-terminal lift2 represents lifted arguments.

```
lift3 :: lift args ⇒ lift
```

The non-terminal lift3 represents lifted arguments.

```
lift4 :: lift args ⇒ lift
```

The non-terminal lift4 represents lifted arguments.

```
liftImp :: lift ⇒ bool
```

The non-terminal liftImp represents lifted implications.

```
liftPlus :: lift ⇒ lift
```

The non-terminal liftPlus represents lifted sums.

```
liftMinus :: lift ⇒ lift
```

The non-terminal liftMinus represents lifted differences.

```
liftTimes :: lift ⇒ lift
```

The non-terminal liftTimes represents lifted products.

```
liftDiv :: lift ⇒ lift
```

The non-terminal liftDiv represents lifted divisions.

```
liftMod :: lift ⇒ lift
```

The non-terminal liftMod represents lifted moduli.

```
liftLess :: lift ⇒ lift
```

The non-terminal liftLess represents lifted less than.

```
liftEq :: lift ⇒ lift
```

The non-terminal liftEq represents lifted equality.

```
liftFinset :: liftargs ⇒ lift
```

The non-terminal liftFinset represents lifted finite sets.

```
liftFunset :: liftargs ⇒ lift
```

The non-terminal liftFunset represents lifted function sets.
-liftPair :: [lift, liftargs] ⇒ lift
    \((1'\,/-')\)

-liftCons :: [lift, lift] ⇒ lift
    \((-\,\#\,\,65,66\,65)\)

-liftApp :: [lift, lift] ⇒ lift
    \((-\,\@/\,\,65,66\,65)\)

-liftList :: liftargs ⇒ lift
    \([([\,])])\)

-ARAll :: [idts, lift] ⇒ lift
    \(((3!\,\,/-\,\,0,10\,10)\)

-AREx :: [idts, lift] ⇒ lift
    \(((3?\,\,/-\,\,0,10\,10)\)

-AREx1 :: [idts, lift] ⇒ lift
    \(((3EX\,\,/-\,\,0,10\,10)\)

-RAll :: [idts, lift] ⇒ lift
    \(((3ALL\,\,/-\,\,0,10\,10)\)

-REx :: [idts, lift] ⇒ lift
    \(((3EX\,\,/-\,\,0,10\,10)\)

-REx1 :: [idts, lift] ⇒ lift
    \(((3EX!\,\,/-\,\,0,10\,10)\)

translations
-const \(\equiv\) CONST const

translations
-lift \(\equiv\) CONST lift
-lift2 \(\equiv\) CONST lift2
-lift3 \(\equiv\) CONST lift3
-lift4 \(\equiv\) CONST lift4

translations
-Valid \(\equiv\) CONST Valid

translations
-RAll x A \(\equiv\) Rall x. A
-REx x A \(\equiv\) Rex x. A
-REx1 x A \(\equiv\) Rex! x. A

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

\(w\,\models\,A\) \(\rightarrow\) A w
LIFT A \(\rightarrow\) A:::-:-

translations
-liftEqu u v \(\equiv\) -lift2 (op =)
-liftNeg u v \(\equiv\) -liftNot (-liftEqu u v)
-liftNot \(\equiv\) -lift (CONST Not)
-liftAnd u v \(\equiv\) -lift2 (op &)
-liftOr u v \(\equiv\) -lift2 (op |)
-liftImp u v \(\equiv\) -lift2 (op −−>)
-liftIf u v \(\equiv\) -lift3 (CONST If)
-liftPlus u v \(\equiv\) -lift2 (op +)
-liftMinus u v \(\equiv\) -lift2 (op −)
-liftTimes u v \(\equiv\) -lift2 (op *)
-\textit{liftDiv} \iff \textit{lift2} (op \textit{div})
-\textit{liftMod} \iff \textit{lift2} (op \textit{mod})
-\textit{liftLess} \iff \textit{lift2} (op \textit{<})
-\textit{liftLeq} \iff \textit{lift2} (op \leq)
-\textit{liftMem} \iff \textit{lift2} (op \in)
-\textit{liftNotMem} x \textit{xs} \iff \textit{liftNot} (\textit{liftMem} x \textit{xs})

\textbf{translations}

-\textit{liftFinset} (\textit{liftargs} \textit{xs} \textit{x}) \iff \textit{lift2} \text{(CONST insert)} \textit{x} (\textit{liftFinset} \textit{xs})
-\textit{liftFinset} \textit{x} \iff \textit{lift2} \text{(CONST insert)} \textit{x} (\text{-const} (\text{CONST Set.empty}))
-\textit{liftPair} \textit{x} (\textit{liftargs} \textit{y} \textit{z}) \iff \textit{liftPair} \textit{x} (\textit{liftPair} \textit{y} \textit{z})
-\textit{liftApp} \iff \textit{lift2} \text{(-const \@)}
-\textit{liftList} (\textit{liftargs} \textit{xs} \textit{x}) \iff \textit{liftCons} (\textit{liftList} \textit{xs} \textit{x})
-\textit{liftList} \textit{x} \iff \textit{liftCons} \textit{x} (\textit{-const \[]})

\textit{w} \iff \sim A \iff \textit{liftNot} A \textit{w}
\textit{w} \iff A \land B \iff \textit{liftAnd} A B \textit{w}
\textit{w} \iff A \lor B \iff \textit{liftOr} A B \textit{w}
\textit{w} \iff A \rightarrow B \iff \textit{liftImp} A B \textit{w}
\textit{w} \iff u = v \iff \textit{liftEq} u v \textit{w}
\textit{w} \iff \text{ALL x.} A \iff \textit{RA} \textit{x} A \textit{w}
\textit{w} \iff \text{EX x.} A \iff \textit{RE} \textit{x} A \textit{w}
\textit{w} \iff \text{EX! x.} A \iff \textit{RE} \textit{EX} \textit{x} A \textit{w}

\textbf{syntax (ax symbols)}

-\textit{Valid} \iff \textit{lift} \Rightarrow \textit{bool} \iff ((\sim \cdot) 5)
-\textit{holdsAt} \iff \text{[\textit{a}, \textit{lift}]} \Rightarrow \textit{bool} \iff ((\cdot \iff \cdot) [100,10] 10)
-\textit{liftNeq} \iff \textit{lift} \Rightarrow \textit{lift} \iff (\text{infixl} \neq 50)
-\textit{liftNot} \iff \textit{lift} \Rightarrow \textit{lift} \iff (\text{infixr} \sim 35)
-\textit{liftAnd} \iff \text{[\textit{lift}, \textit{lift}]} \Rightarrow \textit{lift} \iff (\text{infixr} \land 30)
-\textit{liftOr} \iff \text{[\textit{lift}, \textit{lift}]} \Rightarrow \textit{lift} \iff (\text{infixr} \lor 30)
-\textit{liftImp} \iff \text{[\textit{lift}, \textit{lift}]} \Rightarrow \textit{lift} \iff (\text{infixr} \rightarrow 25)
-\textit{RA} \iff \text{[\textit{ids}, \textit{lift}]} \Rightarrow \textit{lift} \iff ((\text{\textit{\exists}} \cdot \cdot \cdot) [0, 10] 10)
-\textit{RE} \iff \text{[\textit{ids}, \textit{lift}]} \Rightarrow \textit{lift} \iff ((\text{\textit{\exists}} \cdot \cdot \cdot) [0, 10] 10)
-\textit{RE} \iff \text{[\textit{ids}, \textit{lift}]} \Rightarrow \textit{lift} \iff ((\text{\textit{\exists}}! \cdot \cdot \cdot) [0, 10] 10)
-\textit{liftLeq} \iff \text{[\textit{lift}, \textit{lift}]} \Rightarrow \textit{lift} \iff ((\text{\textit{\leq}} \cdot \cdot \cdot) [50, 51] 50)
-\textit{liftNotMem} \iff \text{[\textit{lift}, \textit{lift}]} \Rightarrow \textit{lift} \iff ((\text{\textit{\notin}} \cdot \cdot \cdot) [50, 51] 50)

\textbf{syntax (HTML output)}

-\textit{liftNeq} \iff \text{[\textit{lift}, \textit{lift}]} \Rightarrow \textit{lift} \iff (\text{infixl} \neq 50)
-\textit{liftNot} \iff \text{[\textit{lift}, \textit{lift}]} \Rightarrow \textit{lift} \iff (\text{infixr} \sim 35)
-\textit{liftAnd} \iff \text{[\textit{lift}, \textit{lift}]} \Rightarrow \textit{lift} \iff (\text{infixr} \land 30)
-\textit{liftOr} \iff \text{[\textit{lift}, \textit{lift}]} \Rightarrow \textit{lift} \iff (\text{infixr} \lor 30)
-\textit{RA} \iff \text{[\textit{ids}, \textit{lift}]} \Rightarrow \textit{lift} \iff ((\text{\textit{\exists}} \cdot \cdot \cdot) [0, 10] 10)
-\textit{RE} \iff \text{[\textit{ids}, \textit{lift}]} \Rightarrow \textit{lift} \iff ((\text{\textit{\exists}} \cdot \cdot \cdot) [0, 10] 10)
-\textit{RE} \iff \text{[\textit{ids}, \textit{lift}]} \Rightarrow \textit{lift} \iff ((\text{\textit{\exists}}! \cdot \cdot \cdot) [0, 10] 10)
2.3 Definitions

**defs**

- **Valid-def**: \( \vdash A \equiv \forall w. w \models A \)**
- **unl-con**: \( \text{LIFT } c \ w \equiv c \)**
- **unl-lift**: \( (LIFT f < x >) w \equiv f (x w) \)**
- **unl-lift2**: \( \text{LIFT } f \langle x, y \rangle w \equiv f (x w) (y w) \)**
- **unl-lift3**: \( \text{LIFT } f \langle x, y, z \rangle w \equiv f (x w) (y w) (z w) \)**
- **unl-lift4**: \( \text{LIFT } f \langle x, y, z, z \rangle w \equiv f (x w) (y w) (z w) (zz w) \)**

**defs**

- **unl-Rall**: \( w \models \forall x. A x \equiv \forall x. (w \models A x) \)**
- **unl-Rex**: \( w \models \exists x. A x \equiv \exists x. (w \models A x) \)**
- **unl-Rex1**: \( w \models \exists! x. A x \equiv \exists! x. (w \models A x) \)**

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

**lemmas** intensional-rews[simp] =

- **unl-con**
- **unl-lift**
- **unl-lift2**
- **unl-lift3**
- **unl-lift4**
- **unl-Rall**
- **unl-Rex**
- **unl-Rex1**

2.4 Lemmas and Tactics

**lemma** intD[dest]: \( \vdash A \implies w \models A \)**

**proof** –

- **assume** \( a : A \)
- **from** \( a \) **have** \( \forall w. w \models A \) **by** (auto simp add: Valid-def)
- **thus** \(?thesis \) ..

**qed**

**lemma** intI [intro!]: **assumes** \( P1 : (\forall w. w \models A) \) **shows** \( \vdash A \)**

**using** **assms** **by** (auto simp: Valid-def)

Basic unlifting introduces a parameter \( w \) and applies basic rewrites, e.g \( \vdash F = G \) becomes \( F w = G w \) and \( \vdash F \rightarrow G \) becomes \( F w \rightarrow G w \).

**method-setup** int-unft = \( \langle\langle \text{Scan.succeed } \langle fn \text{ ctxt } => \text{SIMPLE-METHOD'}
\quad \text{rtac } @\{ \text{thm intI} \} \ \text{THEN'} \text{ rewrite-goal-tac ctxt } @\{ \text{thms intensional-rews} \} \rangle \rangle \) **method** to unlift and followed by intensional rewrites

**lemma** inteq-reflection: **assumes** \( P1 : \vdash x=y \) **shows** \( (x \equiv y) \)**

**proof** –

- **from** \( P1 \) **have** \( P2 : \forall \ w. x w = y w \) **by** (unfold Valid-def unl-lift2)
- **hence** \( P3: x=y \) **by** blast
- **thus** \( x \equiv y \) **by** (rule eq-reflection)
lemma int-simps:
\[ \vdash (x = x) = # \text{True} \]
\[ \vdash (\neg # \text{True}) = # \text{False} \]
\[ \vdash (\neg # \text{False}) = # \text{True} \]
\[ \vdash (\neg P) = P \]
\[ \vdash ((\neg P) = P) = # \text{False} \]
\[ \vdash (P = (\neg P)) = # \text{False} \]
\[ \vdash (P \neq Q) = (P = (\neg Q)) \]
\[ \vdash (# \text{True} P) = P \]
\[ \vdash (P = # \text{True}) = P \]
\[ \vdash (# \text{True} \rightarrow P) = P \]
\[ \vdash (\neg # \text{True} \rightarrow P) = # \text{True} \]
\[ \vdash (P \rightarrow # \text{True}) = # \text{True} \]
\[ \vdash (P \rightarrow P) = # \text{True} \]
\[ \vdash (P \rightarrow \neg P) = (\neg P) \]
\[ \vdash (P \rightarrow \neg P) = (\neg P) \]
\[ \vdash (P \land # \text{True}) = P \]
\[ \vdash (# \text{True} \land P) = P \]
\[ \vdash (P \land \neg P) = # \text{False} \]
\[ \vdash (\neg P \land P) = # \text{False} \]
\[ \vdash (P \lor # \text{True}) = P \]
\[ \vdash (P \lor \neg P) = P \]
\[ \vdash (P \lor \neg P) = # \text{True} \]
\[ \vdash (\neg P \lor P) = # \text{True} \]
\[ \vdash (\forall x. P) = P \]
\[ \vdash (\exists x. P) = P \]

by auto

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

method-setup int-rewrite = \(\langle\langle \text{Scan.succeed (fn ctxt \rightarrow SIMPLE-METHOD' (rewrite-goal-tac ctxt @\{thms intensional-simps\}))} \rangle\rangle\)

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

qed
lemma TrueW [simp]: ⊢ # True
  by auto

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

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

lemma int-iffD1: assumes h: ⊢ F = G shows ⊢ F −→ G
  using h by auto

lemma int-iffD2: assumes h: ⊢ F = G shows ⊢ G −→ F
  using h by auto

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

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

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

end

3 Semantics

theory Semantics
imports Sequence Intensional
begin

This theory mechanises a shallow embedding of TLA\(^\ast\) using the Sequence and Intensional theories. A shallow embedding represents TLA\(^\ast\) using Isabelle/HOL predicates, while a deep embedding would represent TLA\(^\ast\) 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 verifi-

\(^1\)See e.g. [10] for a discussion about deep vs. shallow embeddings in Isabelle/HOL.
cation 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:

\[
\begin{align*}
\text{type-synonym} & \quad (\forall a, b) \text{formfun} = \forall a \text{ seq } \Rightarrow b \\
\text{type-synonym} & \quad a \text{ formula} = (a, \text{bool}) \text{ formfun} \\
\text{type-synonym} & \quad (\forall a, b) \text{ stfun} = \forall a \Rightarrow b \\
\text{type-synonym} & \quad a \text{ stpred} = (a, \text{bool}) \text{ stfun}
\end{align*}
\]

instance

fun :: (type, type) world ..

instance

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.

\[
\begin{align*}
\text{definition} \quad \text{always} :: (a::world) \text{ formula } \Rightarrow (a \text{ formula} \\
\text{where} \quad \text{always F} & \equiv \lambda s. \exists n. (s|s n) \models F \\
\text{definition} \quad \text{nexts} :: (a::world) \text{ formula } \Rightarrow (a \text{ formula} \\
\text{where} \quad \text{nexts F} & \equiv \lambda s. (\text{tail } s) \models F \\
\text{definition} \quad \text{before} :: (a::world, b) \text{ stfun } \Rightarrow (a, b) \text{ formfun} \\
\text{where} \quad \text{before f} & \equiv \lambda s. (\text{first } s) \models f \\
\text{definition} \quad \text{after} :: (a::world, b) \text{ stfun } \Rightarrow (a, b) \text{ formfun} \\
\text{where} \quad \text{after f} & \equiv \lambda s. (\text{second } s) \models f \\
\text{definition} \quad \text{unch} :: (a::world, b) \text{ stfun } \Rightarrow (a \text{ formula} \\
\text{where} \quad \text{unch v} & \equiv \lambda s. s \models (\text{after } v) = (\text{before } v) \\
\text{definition} \quad \text{action} :: (a::world) \text{ formula } \Rightarrow (a, b) \text{ stfun } \Rightarrow (a \text{ formula} \\
\text{where} \quad \text{action P v} & \equiv \lambda s. \forall i. ((s|s i) \models P) \lor ((s|s i) \models \text{unch } v)
\end{align*}
\]

3.2.1 Concrete Syntax

This is the concrete syntax for the (abstract) operators above.
translations
-always \(\Rightarrow\) CONST always
-nexts \(\Rightarrow\) CONST nexts
-action \(\Rightarrow\) CONST action
-before \(\Rightarrow\) CONST before
-after \(\Rightarrow\) CONST after
-prime \(\Rightarrow\) CONST after
-unch \(\Rightarrow\) CONST unch
TEMP F \(\Rightarrow\) \((F:: (\text{nat} \Rightarrow -)) \Rightarrow -)\)

syntax (xsymbols)
-always :: lift \(\Rightarrow\) lift ((\[\[\cdot\cdot\] -]) [90] 90)
-nexts :: lift \(\Rightarrow\) lift ((\langle\cdot\rangle) [90] 90)
-action :: lift, lift \(\Rightarrow\) lift ((\langle\langle\cdot\rangle\rangle) [20,1000] 90)
-before :: lift \(\Rightarrow\) lift ((\$) [100] 99)
-after :: lift \(\Rightarrow\) lift ((\$) [100] 99)
-prime :: lift \(\Rightarrow\) lift ((\cdot) [100] 99)
-unch :: lift \(\Rightarrow\) lift ((\text{Unchanged -}) [100] 99)
TEMP :: lift \(\Rightarrow\) 'b ((TEMP -))

3.3 Abbreviations

Some standard temporal abbreviations, with their concrete syntax.

definition actrans :: ('a::world) formula \(\Rightarrow\) ('a,'b) stfun \(\Rightarrow\) 'a formula
where actrans P v \(\equiv\) TEMP(P \(\lor\) unch v)

definition eventually :: ('a::world) formula \(\Rightarrow\) 'a formula
where eventually F \(\equiv\) LIFT(\![\neg (\neg F)]\!)

definition angle-action :: ('a::world) formula \(\Rightarrow\) ('a,'b) stfun \(\Rightarrow\) 'a formula
where angle-action P v \(\equiv\) LIFT(\![\neg (\neg P) \cdot v]\!)

definition angle-actrans :: ('a::world) formula \(\Rightarrow\) ('a,'b) stfun \(\Rightarrow\) 'a formula
where angle-actrans P v \(\equiv\) TEMP (~ actrans (LIFT(~ P)) v)

definition leadsto :: ('a::world) formula \(\Rightarrow\) 'a formula \(\Rightarrow\) 'a formula
where leadsto P Q \(\equiv\) LIFT \([P \rightarrow \text{eventually } Q]\!)

3.3.1 Concrete Syntax

syntax
-actrans :: lift, lift \(\Rightarrow\) lift ((\langle\langle\cdot\rangle\rangle) [20,1000] 90)
-eventually :: lift \(\Rightarrow\) lift ((\langle\cdot\rangle) [90] 90)
-angle-action :: lift, lift \(\Rightarrow\) lift ((\langle\langle\cdot\rangle\rangle) [20,1000] 90)
-angle-actrans :: lift, lift \(\Rightarrow\) lift ((\langle\langle\cdot\rangle\rangle) [20,1000] 90)
-leadsto :: [lift,lift] ⇒ lift ((−→ −) [26,25] 25)

translations
-actrans ⇒ CONST actrans
-eventually ⇒ CONST eventually
-angle-action ⇒ CONST angle-action
-angle-actrans ⇒ CONST angle-actrans
-leadsto ⇒ CONST leadsto

syntax (xsymbols)
-eventually :: lift ⇒ lift (((○) [90] 90)
-angle-action :: [lift,lift] ⇒ lift (((○)′(-)) [20,1000] 90)
-angle-actrans :: [lift,lift] ⇒ lift (((′)((-)) [20,1000] 90)
-leadsto :: [lift,lift] ⇒ lift ((−⇝ −) [26,25] 25)

3.4 Properties of Operators

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

lemma eventually-defs: (w |= ◇ F) = (∃ n. (w |s n) |= F)
  by (simp add: eventually-def always-def)

lemma angle-action-defs: (w |= ◇(P)·v) = (∃ i. ((w |s i) |= P) ∧ ((w |s i) |= v$ ≠ $v))
  by (simp add: angle-action-def action-def unch-def)

lemma unch-defs: (w |= Unchanged v) = (((second w) |= v) = ((first w) |= v))
  by (simp add: unch-def before-def nexts-def after-def tail-def suffix-def first-def second-def)

lemma linalw:
  assumes h1: a ≤ b and h2: (w |s a) |= □A
  shows (w |s b) |= □A
proof (clarsimp simp: always-def)
  fix n
  from h1 obtain k where g1: b = a + k by (auto simp: le-iff-add)
  with h2 show (w |s b |s n) |= A by (auto simp: always-def suffix-plus ac-simps)
qed

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**

\[
\text{stutinv} :: (\alpha, \beta) \xrightarrow{\text{formfun}} \text{bool}
\]

**Where**

\[
\text{stutinv} F \equiv \forall \sigma \tau. \sigma \approx \tau \rightarrow (\sigma \models F) = (\tau \models F)
\]

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: $\square[P]$-$\nu$. To show that $\square[P]$-$\nu$ is stuttering invariant, is must be shown that a slightly weaker predicate holds for $P$. For example, if $P$ contains a term of the form $\diamond Q$, then it is not a well-formed pre-formula, thus $\square[P]$-$\nu$ is not stuttering invariant. This weaker version of stuttering invariance has been named near stuttering invariance.

**Definition**

\[
\text{nstutinv} :: (\alpha, \beta) \xrightarrow{\text{formfun}} \text{bool}
\]

**Where**

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

**Syntax**

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

**Translations**

- $\text{-stutinv} \equiv \text{CONST stutinv}$
- $\text{-nstutinv} \equiv \text{CONST nstutinv}$

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** $\text{nstimv-def}$

**Proof** (clarify)

fix $s t :: \text{nat} \Rightarrow \alpha$
assume $a1$: $\text{first } s = \text{first } t$ and $a2$: $(\text{tail } s) \approx (\text{tail } t)$
have $s \approx t$
proof
t have $tg1$: $(\text{first } s) \#\# (\text{tail } s) = s$ by $\text{rule seq-app-first-tail}$
  have $tg2$: $(\text{first } t) \#\# (\text{tail } t) = t$ by $\text{rule seq-app-first-tail}$
with $a1$ have $tg2'$: $(\text{first } s) \#\# (\text{tail } t) = t$ by simp
from $a2$ have $(\text{first } s) \#\# (\text{tail } s) \approx (\text{first } s) \#\# (\text{tail } t)$ by $\text{rule app-seqsimilar}$
  with $tg1$ $tg2'$ show $\text{thesis}$ by simp
qed
with h show \((s \models F) = (t \models F)\) by (simp add: stutinv-def)
qed

3.5.1 Properties of \textit{stutinv}

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

\textbf{theorem} \textit{stut-before}: \textit{STUTINV} \$F$
\textbf{proof} (clarsimp simp: stutinv-def)
fix \(s \text{ and } t\) :: 'a seq
assume \(a1\): \(s \approx t\)
then \((\text{first } s) = (\text{first } t)\) by (rule sim-first)
thus \((s \mid= F) = (t \mid= F)\) by (simp add: before-def)
qed

\textbf{lemma} \textit{nstut-after}: \textit{NSTUTINV} \(F\$
\textbf{proof} (clarsimp simp: stutinv-def)
fix \(s \text{ and } t\) :: 'a seq
assume \(a1\): \(\text{tail } s \approx \text{tail } t\)
then \((s \mid= F\$) = (t \mid= F\$)\) by (simp add: after-def tail-sim-second)
qed

The always operator preserves stuttering invariance.

\textbf{theorem} \textit{stut-always}: \textbf{assumes} \(H\):\textit{STUTINV} \(F\) \textbf{shows} \textit{STUTINV} \(\Box F\)
\textbf{proof} (clarsimp simp: stutinv-def)
fix \(s \text{ and } t\) :: 'a seq
assume \(a2\): \(s \approx t\)
show \((s \models (\Box F)) = (t \models (\Box F))\)
  proof
    assume \(a1\): \(t \models \Box F\)
    show \(s \models \Box F\)
      proof (clarsimp simp: always-def)
        fix \(n\)
        from \(a2[THEN \text{ sim-step}]\) obtain \(m\) where \(s \mid s \mid n \approx t \mid s \mid m\) by blast
        from \(a1\) have \((t \mid s \mid m) \models F\) by (simp add: always-def)
        with \(H\) \(m\) show \((s \mid s \mid n) \models F\) by (simp add: stutinv-def)
      qed
  next
  assume \(a1\): \(s \models (\Box F)\)
  show \(t \models (\Box F)\)
    proof (clarsimp simp: always-def)
      fix \(n\)
      from \(a2[THEN \text{ seqsim-sym}, \text{ THEN sim-step}]\) obtain \(m\) where \(m: t \mid s \mid n \approx s \mid s \mid m\) by blast
      from \(a1\) have \((s \mid s \mid m) \models F\) by (simp add: always-def)
      with \(H\) \(m\) show \((t \mid s \mid n) \models F\) by (simp add: stutinv-def)
    qed

31
proof
definition

proof

lemma stut-action-lemma:
assumes H: NSTUTINV P and st: s \approx t and P: t \models \Box[P] - v
shows s \models \Box[P] - v

proof (clarsimp simp: action-def)

fix n
assume \neg((s \mid_s n) \models Unchanged v)

by (simp add: unch-defs first-def second-def suffix-def)

from \mathbf{st}(\mathbf{THEN} \mathbf{sim-step}) obtain m where
\begin{align*}
a2'': s \mid_s n \approx t \mid_s m \\
\land (s \mid_s Suc n \approx t \mid_s Suc m \lor s \mid_s Suc n \approx t \mid_s m) \
\end{align*}

hence g1': (s \mid_s n \approx t \mid_s m) by simp

hence g1'': first (s \mid_s n) = first (t \mid_s m) by (simp add: sim-first)

hence g1': s n = t m by (simp add: suffix-def first-def)

from a2' have g2': s \mid_s Suc n \approx t \mid_s Suc m \lor s \mid_s Suc n \approx t \mid_s m by simp

from P have a1': ((t \mid_s m) \models P) \lor ((t \mid_s m) \models Unchanged v) by (simp add: action-def)

from g2 show (s \mid_s n) \models P

proof

assume s \mid_s Suc n \approx t \mid_s m

hence first (s \mid_s Suc n) = first (t \mid_s m) by (simp add: sim-first)

hence s (Suc n) = t m by (simp add: suffix-def first-def)

with g1' v show \?thesis by simp — by contradiction

next

assume a3: s \mid_s Suc n \approx t \mid_s Suc m

hence first (s \mid_s Suc n) = first (t \mid_s Suc m) by (simp add: sim-first)

hence a3': s (Suc n) = t (Suc m) by (simp add: suffix-def first-def)

from a1' show \?thesis

proof

assume (t \mid_s m) \models Unchanged v

hence v (t (Suc m)) = v (t m)

by (simp add: unch-defs first-def second-def suffix-def)

with g1' a3' v show \?thesis by simp — again, by contradiction

next

assume a4': (t \mid_s m) \models P

from a3 have tail (s \mid_s n) \approx tail (t \mid_s m) by (simp add: tail-def suffix-plus)

with H g1'' a4 show \?thesis by (auto simp: nstutinv-def)

qed

qed

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

proof (clarsimp simp: stutinv-def)

fix s t :: 'a seq

32
assume \( st: s \approx t \)
\[ \text{show } (s \models \square[P]v) = (t \models \square[P]v) \]
proof
assume \( t \models \square[P]v \)
with \( H \) \( st \) show \( s \models \square[P]v \) by (rule stat-action-lemma)
next
assume \( s \models \square[P]v \)
with \( H \) seqsim-sym show \( t \models \square[P]v \) by (rule stat-action-lemma)
qed

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

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

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

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

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

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

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

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

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

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

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

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

\textbf{lemma} \( \text{stut-fun3: } [\text{STUTINV } X;\text{STUTINV } Y;\text{STUTINV } Z] \implies \text{STUTINV } (f <X,Y,Z>) \)
by (simp add: stutinv-def)
lemma `stut-fun4`: 
\[ [\text{STUTINV } X; \text{STUTINV } Y; \text{STUTINV } Z; \text{STUTINV } W] \implies \text{STUTINV} (f < X, Y, Z, W>) \]
by \((\text{simp add: stutinv-def})\)

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

### 3.5.2 Properties of \text{-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 \((\text{simp add: stutinv-def nstutinv-def nexts-def})\)

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

lemma `nstut-and`: 
\[ [\text{NSTUTINV } F; \text{NSTUTINV } G] \implies \text{NSTUTINV} (F \land G) \]
by \((\text{auto simp: nstutinv-def})\)

lemma `nstut-or`: 
\[ [\text{NSTUTINV } F; \text{NSTUTINV } G] \implies \text{NSTUTINV} (F \lor G) \]
by \((\text{auto simp: nstutinv-def})\)

lemma `nstut-imp`: 
\[ [\text{NSTUTINV } F; \text{NSTUTINV } G] \implies \text{NSTUTINV} (F \rightarrow G) \]
by \((\text{auto simp: nstutinv-def})\)

lemma `nstut-eq`: 
\[ [\text{NSTUTINV } F; \text{NSTUTINV } G] \implies \text{NSTUTINV} (F = G) \]
by \((\text{force simp: nstutinv-def})\)

lemma `nstut-not`: \(\text{NSTUTINV } F \implies \text{NSTUTINV} (~ F)\)
by \((\text{auto simp: nstutinv-def})\)

lemma `nstut-eq2`: 
\[ [\text{NSTUTINV } F; \text{NSTUTINV } G] \implies \text{NSTUTINV} (F \neq G) \]
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 \((\text{auto simp: nstutinv-def})\)

lemma `nstut-ex`: \((\exists x. \text{NSTUTINV} (F x)) \implies \text{NSTUTINV} (\exists x. F x)\)
by \((\text{auto simp: nstutinv-def})\)

lemma `nstut-const`: \(\text{NSTUTINV} \# c\)
by \((\text{auto simp: nstutinv-def})\)

lemma `nstut-fun1`: \(\text{NSTUTINV } X \implies \text{NSTUTINV} (f < X>)\)
by \((\text{force simp: nstutinv-def})\)

lemma `nstut-fun2`: 
\[ [\text{NSTUTINV } X; \text{NSTUTINV } Y] \implies \text{NSTUTINV} (f < X,Y>) \]
by (force simp: nstutinv-def)

lemma nstut-fun3: \([\text{NSTUTINV } X; \text{NSTUTINV } Y; \text{NSTUTINV } Z] \implies \text{NSTUTINV} (f <X,Y,Z>)\)
  by (force simp: nstutinv-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 (force simp: nstutinv-def)

lemma nstut-plus: \([\text{NSTUTINV } x; \text{NSTUTINV } y] \implies \text{NSTUTINV} (x+y)\)
  by (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: \(\text{NSTUTINV} (\text{Unchanged } v)\)
  proof (unfold unch-def)
    have g1: \(\text{NSTUTINV } v\$\) by (rule nstut-after)
    have \(\text{NSTUTINV } \$v\) by (rule stat-before[THEN statinv-strictly-stronger])
    with g1 show \(\text{NSTUTINV} (v\$ = \$v)\) by (rule nstut-eq)
  qed

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

lemma nstut-actrans: \(\text{NSTUTINV } P \implies \text{NSTUTINV} [P]-v\)
  by (simp add: actrans-def nstut-unch nstut-or)

lemma stat-eventually: \(\text{STUTINV } F \implies \text{STUTINV} ♦ F\)
  by (simp add: eventually-def stat-not stat-always)

lemma stat-leadsto: \([\text{STUTINV } F; \text{STUTINV } G] \implies \text{STUTINV} (F \leadsto G)\)
  by (simp add: leadsto-def stat-always stat-eventually stat-imp)

lemma stat-angle-action: \(\text{NSTUTINV } P \implies \text{NSTUTINV} ♦(P)-v\)
  by (simp add: angle-action-def nstut-not nstut-action nstut-not)

lemma nstut-angle-acttrans: \(\text{NSTUTINV } P \implies \text{NSTUTINV} ⟨P⟩-v\)
  by (simp add: angle-actrans-def nstut-not nstut-actrans)

lemmas stutinvs = stat-before stat-always stat-action
  stat-and stat-or stat-imp stat-eq stat-noteq stat-not
  stat-all stat-ex stat-eventually stat-leadsto stat-angle-action stat-const
lemmas nstutinvs = nstut-after nstut-nexts nstut-actrans
nstut-unch nstut-and nstut-or nstut-imp nstut-eq nstut-noteq nstut-not
nstut-actrans nstut-all nstut-ex nstut-angle-acttrans nstutinvs
nstut-strictly-stronger
nstut-fun1 nstut-fun2 nstut-fun3 nstut-fun4 nstutinvs

lemmas bothnstutinvs = nstutinvs 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 ≡ ∀ w. w |\sim A

syntax
-PreValid :: lift \Rightarrow bool  ((|\sim -) 5)

translations
-PreValid \equiv CONST PreValid

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

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

method-setup pref-unlift = ⟨
Scan.succeed (fn ctxt => SIMPLE-METHOD'
  (rtac @{thm 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-iff1: assumes \( \neg F \implies G \) and \( \neg G \implies F \) shows \( \neg F = G \) using assms by force

lemma pref-iffD1: assumes \( \neg F = G \) shows \( \neg F \implies G \) using assms by auto

lemma pref-iffD2: assumes \( \neg F = G \) shows \( \neg G \implies F \) using assms by auto

lemma unl-pref-imp: assumes \( \neg F \implies G \) shows \( \forall w. \, w \models F \implies G \) using assms by auto

lemma pref-imp-trans: assumes \( \neg F \implies G \) and \( \neg G \implies H \) shows \( \neg F \implies H \) 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 \( \neg F \land \text{Unchanged } id \implies oF \)
proof (pref-unlift, clarify)
fix \( s \)
assume \( a1: \models F \land \text{Unchanged } id \)
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 \( h \) have \( (\text{tail } s) \models F \) by (simp add: stutinv-def)
thus \( s \models oF \) by (unfold nexts-def)
qed

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

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

lemmas unch-eq1 = unch-pair[THEN pref-eq]
lemmas unch-eq2 = unch-pair[THEN pref-eq-reflection]
lemma \text{angle-actrans-sem}: \sim \langle F \rangle - v = (F \land v \neq \$ v)
by (auto simp: \text{angle-actrans-def} \text{actrans-def} \text{unch-def})

lemmas \text{angle-actrans-sem-eq} = \text{angle-actrans-sem} \THEN \text{pref-eq}

4.2 Lemmas about \textit{after}

lemma \text{after-const}: \sim (\# c)' = \# c
by (auto simp: \text{nexts-def} \text{before-def} \text{after-def})

lemma \text{after-fun1}: \sim \langle f, x' \rangle = f <x'>
by (auto simp: \text{nexts-def} \text{before-def} \text{after-def})

lemma \text{after-fun2}: \sim \langle f, x, y \rangle = f <x', y'>
by (auto simp: \text{nexts-def} \text{before-def} \text{after-def})

lemma \text{after-fun3}: \sim \langle f, x, y, z \rangle = f <x', y', z'>
by (auto simp: \text{nexts-def} \text{before-def} \text{after-def})

lemma \text{after-fun4}: \sim \langle f, x, y, z, \_ \rangle = f <x', y', z', \_'>
by (auto simp: \text{nexts-def} \text{before-def} \text{after-def})

lemma \text{after-forall}: \sim (\forall x. P x)' = (\forall x. (P x)')
by (auto simp: \text{nexts-def} \text{before-def} \text{after-def})

lemma \text{after-exists}: \sim (\exists x. P x)' = (\exists x. (P x)')
by (auto simp: \text{nexts-def} \text{before-def} \text{after-def})

lemma \text{after-exists1}: \sim (\exists! x. P x)' = (\exists! x. (P x)')
by (auto simp: \text{nexts-def} \text{before-def} \text{after-def})

lemmas \text{all-after} = \text{after-const} \text{after-fun1} \text{after-fun2} \text{after-fun3} \text{after-fun4}
\text{after-forall} \text{after-exists} \text{after-exists1}

lemmas \text{all-after-unl} = \text{all-after} \THEN \text{prefD}

lemmas \text{all-after-eq} = \text{all-after} \THEN \text{pref-eq-reflection}

4.3 Lemmas about \textit{before}

lemma \text{before-const}: \vdash (\# c) = \# c
by (auto simp: \text{before-def})

lemma \text{before-fun1}: \vdash f <x> = f <x>
by (auto simp: \text{before-def})

lemma \text{before-fun2}: \vdash f <x, y> = f <x, y>
by (auto simp: \text{before-def})

lemma \text{before-fun3}: \vdash f <x, y, z> = f <x, y, z>
by (auto simp: \text{before-def})
lemma before-fun4: \( \vdash (f \prec x, y, z, \_zz) = f \prec \_x, \_y, \_z, \_zz \) 
by (auto simp: before-def)

lemma before-forall: \( \vdash (\forall \_x. P \_x) = (\forall \_x. P \_x) \) 
by (auto simp: before-def)

lemma before-exists: \( \vdash (\exists \_x. P \_x) = (\exists \_x. P \_x) \) 
by (auto simp: before-def)

lemma before-exists1: \( \vdash (\exists! \_x. P \_x) = (\exists! \_x. P \_x) \) 
by (auto simp: before-def)

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: \( \sim (\langle F \land G \rangle - v) = (\langle F \rangle - v \land \langle G \rangle - v) \) 
by (auto simp: angle-actrans-def actrans-def unch-def)

lemma angle-actrans-disj: \( \sim (\langle F \lor G \rangle - v) = (\langle F \rangle - v \lor \langle G \rangle - v) \) 
by (auto simp: angle-actrans-def actrans-def unch-def)

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

lemma pref-eq-true: \( \vdash P = \vdash P = \# \text{True} \) 
by auto

4.5 Unlifting attributes and methods

Attribute which unlifts an intensional formula or preformula

\[
\text{ML:} \quad \text{fun unl-rewr ctxt thm = let} \quad \begin{array}{l}
\quad \text{val unl = (thm RS \{thm intD\}) handle THM - => (thm RS \{thm prefD\}) handle THM - => thm} \\
\quad \text{val rewr = rewrite-rule ctxt @\{thms intensional-rews\}} \\
\quad \text{in unl |> rewr end;}
\end{array}
\]

\[
\text{attribute-setup unlifted = \langle Scan.succeed (Thm.rule-attribute (unl-rewr o Context.proof-of))\rangle}
\]
unlift intensional formulas

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

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

ML ⟨⟨
  fun int-rewr thm =
    (thm RS @{thm inteq-reflection})
    handle THM - => (thm RS @{thm preeq-reflection})
    handle THM - => ((thm RS @{thm int-eq-true}) RS @{thm inteq-reflection})
    handle THM - => ((thm RS @{thm pref-eq-true}) RS @{thm preeq-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 definitions 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 \textit{pmp} are changed to accommodate this. It is argued that while the hetero-
geneous 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 \(\square[P]\)-v, which is not-
tionally 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

\begin{itemize}
  \item \texttt{fmp}: assumes \(\vdash F \text{ and } \vdash F \rightarrow G\) shows \(\vdash G\)
    \begin{itemize}
      \item using \texttt{assms[unlifted]} by \texttt{auto}
    \end{itemize}
  \item \texttt{pmp}: assumes \(\vdash \neg F \text{ and } \vdash \neg F \rightarrow G\) shows \(\vdash \neg G\)
    \begin{itemize}
      \item using \texttt{assms[unlifted]} by \texttt{auto}
    \end{itemize}
  \item \texttt{sq}: assumes \(\vdash \neg \neg P\) shows \(\vdash \square[P]\)-v
    \begin{itemize}
      \item using \texttt{assms[unlifted]} by \texttt{(auto simp: action-def)}
    \end{itemize}
  \item \texttt{pre}: assumes \(\vdash F\) shows \(\vdash \neg F\)
    \begin{itemize}
      \item using \texttt{assms} by \texttt{auto}
    \end{itemize}
  \item \texttt{nex}: assumes \texttt{h1:} \(\vdash F\) shows \(\vdash \o F\)
    \begin{itemize}
      \item using \texttt{assms} by \texttt{(auto simp: nexts-def)}
    \end{itemize}
  \item \texttt{ax0}: \(\vdash \# \text{ True}\)
    \begin{itemize}
      \item by \texttt{auto}
    \end{itemize}
  \item \texttt{ax1}: \(\vdash \square F \rightarrow F\)
    \begin{itemize}
      \item proof (\texttt{clarsimp simp: always-def})
      \item fix \(w\)
      \item assume \(\forall n. (w \mid s n) \models F\)
      \item hence \((w \mid s 0) \models F\), ..
      \item thus \(w \models F\) by \texttt{simp}
    \end{itemize}
    \texttt{qed}
  \item \texttt{ax2}: \(\vdash \square F \rightarrow \square \square F\)-v
    \begin{itemize}
      \item by \texttt{(auto simp: always-def action-def suffix-plus)}
    \end{itemize}
  \item \texttt{ax3}:
    \begin{itemize}
      \item assumes \(H: \vdash \neg F \wedge Unchanged v \rightarrow \o F\)
      \item shows \(\vdash \square[F \rightarrow \o F]\)-v \(\rightarrow (F \rightarrow \square F)\)
    \end{itemize}
    \begin{itemize}
      \item proof (\texttt{clarsimp simp: always-def})
      \item fix \(w\) \(n\)
      \item assume \(a1: w \models \square[F \rightarrow \o F]\)-v and \(a2: w \models F\)
      \item show \((w \mid s n) \models F\)
      \item proof (\texttt{induct n})
        \begin{itemize}
          \item from \(a2\) show \((w \mid s 0) \models F\) by \texttt{simp}
        \end{itemize}
    \end{itemize}
\end{itemize}
next
  fix m
  assume a3: (w |s m) |= F
  with a1 H[unlifted] show (w |s (Suc m)) |= F
    by (auto simp: nexts-def action-def tail-suffix-suc)
qed
qed

theorem ax4: ⊢ □[P → Q]-v → (□[P]-v → □[Q]-v)
  by (force simp: action-def)

theorem ax5: ⊢ □[v' ≠ $v]-v
  by (auto simp: action-def unch-def)

theorem pax0: "¬ # True"
  by auto

theorem pax1 [simp-unl]: "¬ (o¬F) = (¬oF)
  by (auto simp: nexts-def)

theorem pax2: "¬ o(F → G) → (oF → oG)
  by (auto simp: nexts-def)

theorem pax3: "¬ □F → o□F
  by (auto simp: always-def nexts-def tail-def suffix-plus)

theorem pax4: "¬ □[P]-v = ([P]-v ∧ o□[P]-v)
  proof (auto)
    fix w
    assume w |= □[P]-v
    from this[unfolded action-def] have ((w |s 0) |= P) ∨ ((w |s 0) |= Unchanged v) ..
    thus w |= [P]-v by (simp add: actrans-def)
  next
    fix w
    assume w |= □[P]-v
    thus w |= o□[P]-v by (auto simp: nexts-def action-def tail-def suffix-plus)
  next
    fix w
    assume 1: w |= [P]-v and 2: w |= o□[P]-v
    show w |= □[P]-v
      proof (auto simp: action-def)
        fix i
        assume 3: ¬ ((w |s i) |= Unchanged v)
        show (w |s i) |= P
          proof (cases i)
            assume i = 0
            with 1 3 show ?thesis by (simp add: actrans-def)
          next
fix
assume 
with 2 3 show thesis by (auto simp: nexts-def action-def tail-def suffix-plus)
qed
qed

theorem pax5:
assume h1: F 
with stutinv F 
shows thesis by auto
qed

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 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 alw2 using an additional hypothesis \( \neg F \land \text{ Unchanged } v \to \neg \).

theorem alw2:
assumes h1: F and h2: \( \neg F \land \text{ Unchanged } v \to \neg F \)
shows \( \Box F \)
proof
  from h1 have g2: \( \neg F \) by (rule nex)
  hence g3: \( \neg F \to \Box F \) by auto
  hence g4: \( \Box F \to \Box F \) by (rule sq)
  from h2 have g5: \( \Box F \to \Box F \) by (rule ax3)
  with g4 [unlifted] have g6: \( F \to \Box F \) by auto
  with h1 [unlifted] show thesis by auto
qed

Similar theorem, assuming that F is stuttering invariant.

theorem alw3:
assumes h1: F and h2: stutinv F
shows \( \Box F \)
proof –
from h2 have \( \sim F \land \text{Unchanged id} \rightarrow \Diamond F \) by (rule pre-id-unch)
with h1 show \( \Diamond \text{thesis} \) by (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 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 (auto simp: always-def)

**Theorem alw-valid-iff-valid**: \( (\vdash \Box F) = (\vdash F) \)
proof
assume \( \vdash \Box F \)
from this ax1 show \( \vdash F \) by (rule fmp)
qed (rule alw)

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

**Theorem T1[simp-ung]:** \( \vdash \Box [P] \rightarrow \Box [P] \)
proof (auto simp: always-def suffix-plus)
fix \( w n \)
assume \( \forall m k. (w \upharpoonright n (k+m)) \vdash F \)
hence \( (w \upharpoonright n (n+0)) \vdash F \) by blast
thus \( (w \upharpoonright n) \vdash F \) by simp
qed

**Theorem T2[simp-ung]:** \( \vdash \Box [P] \rightarrow \Box [P] \)
proof
have 1: \( \sim \Box [P] \rightarrow \Diamond [P] \) by force
hence \( \vdash \Box [P] \rightarrow \Diamond [P] \) by (rule sq)
moreover
have \( \vdash \Box [P] \rightarrow \Diamond [P] \rightarrow \Box [P] \rightarrow \Box [P] \)
by (rule ax3) (auto elim: 1[unlift-rule])
moreover
have \( \vdash \Box [P] \rightarrow \Box [P] \) by (rule ax1)
ultimately show \( \Diamond [P] \) by force
qed

**Theorem T3[simp-ung]:** \( \vdash \Box [P] \rightarrow \Box [P] \)
proof
have \( \sim P \rightarrow [P] \) by (auto simp: actrans-def)
hence \( \vdash \Box [P] \rightarrow [P] \) by (rule sq)
with ax4 have \( \vdash [P] \rightarrow [P] \) by force
moreover
have \( \sim [P] \rightarrow v \neq v \rightarrow P \) by (auto simp: unch-def actrans-def)
hence \( \vdash \Box [P] \rightarrow v \neq v \rightarrow P \) by (rule sq)
with \textit{ax}5 \textbf{have} \vdash \Box [F] \rightarrow \Box [G] \textbf{by} \ (\textit{force intro: ax}4\texttt{[unlift-rule]})
ultimately \textbf{show} \ ?thesis \textbf{by} \textit{force}

\textbf{qed}

\textbf{theorem} \ M2:
\textbf{assumes} \ h: \sim F \rightarrow G
\textbf{shows} \ sq[\Box F] \rightarrow sq[\Box G]
\textbf{using} \ \textit{force} \ \textbf{by} \ \textit{force}

\textbf{theorem} \ N1:
\textbf{assumes} \ h: \vdash F \rightarrow G
\textbf{shows} \ \sim oF \rightarrow oG
\textbf{by} \ \ (\textit{rule pmp}[OF \ nextrx[OF \ h \ ax2])}

\textbf{theorem} \ T4: \vdash \Box [F] \rightarrow \Box [G]
\textbf{proof} –
\textbf{have} \ \vdash \Box [\Box F] \rightarrow \Box [\Box G]
\textbf{by} \ (\textit{rule ax}2)
\textbf{moreover}
\textbf{from} \ \textit{paxx} \ \textbf{have} \ \sim \Box [\Box F] \rightarrow \Box [\Box G] \textbf{by} \ \textit{force}
\textbf{hence} \ \vdash \Box [\Box [\Box F] \rightarrow \Box [\Box [\Box G]] \textbf{by} \ (\textit{rule M2})
ultimately \textbf{show} \ ?thesis \textbf{unfolding} \ \textit{T}2[\texttt{int-rewrite}] \textbf{by} \ (\textit{rule lift-imp-trans})
\textbf{qed}

\textbf{theorem} \ T5: \vdash \Box [\Box F] \rightarrow \Box [\Box G]
\textbf{proof} –
\textbf{have} \ \vdash [\Box F] \rightarrow [\Box G]
\textbf{by} \ (\texttt{auto simp: actrans-def})
\textbf{hence} \ \vdash \Box [\Box F] \rightarrow \Box [\Box G]
\textbf{by} \ (\textit{rule M2})
\textbf{with} \ T4 \ \textbf{show} \ ?thesis \textbf{unfolding} \ \textit{T}3[\texttt{int-rewrite}] \textbf{by} \ (\textit{rule lift-imp-trans})
\textbf{qed}

\textbf{theorem} \ T6: \vdash \Box F \rightarrow \Box [oF]
\textbf{proof} –
\textbf{from} \ \textit{ax}1 \ \textbf{have} \ \sim (\Box F \rightarrow F) \textbf{by} \ (\textit{rule nex})
\textbf{with} \ \textit{paxx} \ \textbf{have} \ \sim oF \rightarrow oF \textbf{by} \ \textit{force}
\textbf{with} \ \textit{paxx} \ \textbf{have} \ \sim F \rightarrow oF \textbf{by} \ (\textit{rule pref-imp-trans})
\textbf{hence} \ \vdash \Box [\Box F] \rightarrow \Box [\Box oF] \textbf{by} \ (\textit{rule M2})
\textbf{with} \ T2 \ \textbf{show} \ ?thesis \textbf{by} \ (\textit{rule lift-imp-trans})
\textbf{qed}

\textbf{theorem} \ T7:
\textbf{assumes} \ h: \sim F \land \texttt{Unchanged} v \rightarrow oF
\textbf{shows} \ ?thesis \textbf{by} \ (\textit{force intro: ax}4\texttt{[unlift-rule]})

\textbf{proof} –
\textbf{have} \ \vdash oF \rightarrow \Box [oF] \textbf{by} \ \ (\textit{force sq}) \texttt{auto}
\textbf{with} \ \textit{ax}4 \ \textbf{have} \ \vdash [oF] \rightarrow [F \rightarrow oF] \textbf{by} \ \textit{force}
\textbf{with} \ \textit{ax}3[\Box F, \texttt{unlift}] \ \textbf{have} \ \vdash [\Box F] \rightarrow (F \rightarrow \Box F) \textbf{by} \ \textit{force}
\textbf{with} \ \textit{ax}5 \ \textbf{have} \ \sim F \land oF \rightarrow \Box F \textbf{by} \ \textit{force}
\textbf{with} \ \textit{ax}1[\texttt{of TEMP} F, \texttt{unlift}] \ \textit{paxx}[\texttt{of TEMP} F, \texttt{unlift}] \ \textbf{show} \ ?thesis \textbf{by} \ \textit{force}
\textbf{qed}
**Theorem T8:** \( \sim \circ (F \land G) = (\circ F \land \circ G) \)

**Proof:**

have \( \sim \circ (F \land G) \rightarrow \circ F \) by (rule N1) auto

moreover

have \( \sim \circ (F \land G) \rightarrow \circ G \) by (rule N1) auto

moreover

have \( \vdash F \rightarrow G \) by auto

from \( \text{nex[OF this]} \) have \( \sim \circ F \rightarrow \circ G \rightarrow \circ (F \land G) \) by (force intro: pax2[unlift-rule])

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

qed

**Lemma T9:** \( \sim \Box [A] \rightarrow [A] \rightarrow \)

using pax4 by force

**Theorem H1:**

assumes \( h1 : \vdash \Box P \rightarrow Q \) and \( h2 : \vdash \Box P \rightarrow Q \)

shows \( \vdash \Box P \rightarrow Q \)

using assms ax4[unlifted] by force

**Theorem H2:** assumes \( h1 : \vdash \Box P \rightarrow Q \)

shows \( \vdash \Box P \rightarrow Q \)

using \( h1 \) by (blast dest: pre sq)

**Theorem H3:**

assumes \( h1 : \vdash \Box P \rightarrow Q \) and \( h2 : \vdash \Box Q \rightarrow R \)

shows \( \vdash \Box P \rightarrow R \)

**Proof:**

have \( \sim (P \rightarrow Q) \rightarrow (Q \rightarrow R) \rightarrow (P \rightarrow R) \) 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 \( \text{thesis by (rule H1)} \)

qed

**Theorem H4:** \( \vdash \Box[P] \rightarrow P \)

**Proof:**

have \( \sim v' \neq \emptyset \rightarrow ([P] \rightarrow P) \) by (auto simp: unch-def actrans-def)

hence \( \vdash \Box[v' \neq \emptyset \rightarrow ([P] \rightarrow P)] \) by (rule sq)

with ax5 show \( \text{thesis by (rule H1)} \)

qed

**Theorem H5:** \( \vdash \Box \Box F \rightarrow \circ \Box F \)

by (rule pax3[THEN sq])

5.3 Some other useful derived theorems

**Theorem P1:** \( \sim \Box F \rightarrow \circ F \)

**Proof:**

have \( \sim \circ F \rightarrow \circ F \) by (rule N1[OF ax1])
with \texttt{pax3} show \texttt{thesis by (rule \texttt{pref-imp-trans})}
\texttt{qed}

\textbf{theorem P2:} \ \lnot \ \Box F \rightarrow F \land \Diamond F
\begin{itemize}
\item \textit{using \texttt{ax1[of F] P1[of F] by force}}
\end{itemize}

\textbf{theorem P4:} \ \neg \Box F \rightarrow \Box F [v]
\begin{itemize}
\item \textit{proof –}
\item \textit{have \neg \Box F [v] \rightarrow \Box F [v] by (rule \texttt{M2[OF pre[OF ax1]])}}
\item \textit{with \texttt{ax2} show \texttt{thesis by (rule \texttt{lift-imp-trans})}}
\end{itemize}
\texttt{qed}

\textbf{theorem P5:} \ \Box [P] -v \rightarrow \Box [\Diamond P] -w
\begin{itemize}
\item \textit{proof –}
\item \textit{from \texttt{P1} have \lnot \Box F \rightarrow F \rightarrow \Diamond F by force}
\item \textit{hence \Box F [v] \rightarrow \Box F [v] by (rule \texttt{M2})}
\item \textit{with \texttt{ax2} show \texttt{thesis by (unfold \texttt{T2[int-rewrite])}}}
\end{itemize}
\texttt{qed}

\textbf{theorem M0:} \ \Box F \rightarrow \Box [F \rightarrow \Diamond F [v]
\begin{itemize}
\item \textit{proof –}
\item \textit{have \lnot \Box F \rightarrow F \rightarrow \Diamond F by (rule \texttt{P2})}
\item \textit{hence \Box F [v] \rightarrow \Box F [v] by (rule \texttt{M2})}
\item \textit{with \texttt{ax2} show \texttt{thesis by (rule lift-imp-trans)}}
\end{itemize}
\texttt{qed}

\textbf{theorem M1:} \ \Box F \rightarrow \Box [F \land \Diamond F [v]
\begin{itemize}
\item \textit{proof –}
\item \textit{have \lnot \Box F \rightarrow F \rightarrow \Diamond F by (rule \texttt{P2})}
\item \textit{hence \Box F [v] \rightarrow \Box F [v] by (rule \texttt{M2})}
\item \textit{with \texttt{ax2} show \texttt{thesis by (rule lift-imp-trans)}}
\end{itemize}
\texttt{qed}

\textbf{theorem M3:} \ \textit{assumes} \ h: \ \Box F \ \textit{shows} \ \Box [\Diamond F] -v
\begin{itemize}
\item \textit{using \texttt{alw[OF h] T6 by (rule fmp)}}
\end{itemize}

\textbf{lemma M4:} \ \Box [\Diamond (F \land G)] -v \rightarrow (\Diamond F \land \Diamond G) [v]
\begin{itemize}
\item \textit{by (rule \texttt{sq[OF T8])}}
\end{itemize}

\textbf{theorem M5:} \ \Box [\Box P] -v \rightarrow \Box [\Diamond P] -v
\begin{itemize}
\item \textit{proof (rule \texttt{sq})}
\item \textit{show \lnot \Box P [v] \rightarrow \Box \Diamond P [v] by (auto simp: \texttt{pax4[unlifted]})}
\end{itemize}
\texttt{qed}

\textbf{theorem M6:} \ \Box [F \land G] -v \rightarrow \Box [F] -v \land \Box [G] -v
\begin{itemize}
\item \textit{proof –}
\item \textit{have \Box F \land G -v \rightarrow \Box F -v by (rule \texttt{M2}) auto}
\item \textit{moreover have \Box F \land G -v \rightarrow \Box G -v by (rule \texttt{M2}) auto}
\item \textit{ultimately show \texttt{thesis by force}}
\end{itemize}
We now derive Lamport’s 6 simple temporal logic rules (STL1)-(STL6) [5].

Firstly, STL1 is the same as \( \vdash \neg \neg F \iff \vdash \Box \neg \neg F \) derived above.

**theorems** \( STL1 = \text{alw} \)

STL2 and STL3 have also already been derived.
theorems \textit{STL2} = \textit{ax1}

theorems \textit{STL3} = \textit{T1}

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

\begin{proof}
\textbf{theorem} \textit{STL4-2}:
\textit{assumes} h1: \( \vdash F \implies G \) and h2: \( \neg G \land \text{Unchanged} \mathcal{v} \implies \mathcal{G} \)
\textit{shows} \( \vdash \Box F \implies \Box G \)
\textbf{proof} –
from \textit{ax1} \[ \text{of } F \] h1 have \( \vdash \Box F \implies \Box G \) by (rule \textit{lift-imp-trans})
moreover
from h1 have \( \neg \mathcal{F} \implies \mathcal{G} \) by (rule \textit{N1})
hence \( \neg \mathcal{F} \implies G \) by auto
hence \( \Box \neg \mathcal{F} \implies \Box G \) by (rule \textit{M2})
with \textit{T6} have \( \vdash \Box F \implies \Box G \) by (rule \textit{lift-imp-trans})
moreover
from h2 have \( \vdash \Box \neg \mathcal{G} \implies \mathcal{G} \) by auto
hence \( \vdash \Box \neg \mathcal{F} \implies \Box \neg \mathcal{G} \implies \mathcal{G} \) by (rule \textit{ax3})
ultimately
show \( \neg \text{thesis} \) by \textit{force}
\end{proof}

\textbf{lemma} \textit{STL4-3}:
\textit{assumes} h1: \( \vdash F \implies G \) and h2: \( \text{STUTINV } G \)
\textit{shows} \( \vdash \Box (F \land G) = (\Box F \land \Box G) \)
\textbf{using} h1 h2 [\textit{THEN pre-id-unch}] by (rule \textit{STL4-2})

Of course, the original rule can be derived semantically

\textbf{lemma} \textit{STL4}: \textit{assumes} h: \( \vdash F \implies G \) \textit{shows} \( \vdash \Box F \implies \Box G \)
\textbf{using} h by (\textit{force simp: always-def})

\textbf{Dual rule for} \( \Diamond \)

\textbf{lemma} \textit{STL4-eve}: \textit{assumes} h: \( \vdash F \implies G \) \textit{shows} \( \vdash \Diamond F \implies \Diamond G \)
\textbf{using} h by (\textit{force simp: eventually-defs})

Similarly, a purely syntactic derivation of \textit{(STL5)} requires extra hypotheses.

\begin{proof}
\textbf{theorem} \textit{STL5-2}:
\textit{assumes} h1: \( \neg F \land \text{Unchanged } \mathcal{f} \implies \mathcal{F} \)
\textit{and} h2: \( \neg G \land \text{Unchanged } \mathcal{g} \implies \mathcal{G} \)
\textit{shows} \( \vdash \Box (F \land G) = (\Box F \land \Box G) \)
\textbf{proof} (rule \textit{int-iffI})
\textbf{have} \( \vdash F \land G \implies \Box F \) by \textit{auto}
from this \textbf{h1} have \( \vdash \Box (F \land G) \implies \Box F \) by (rule \textit{STL4-2})
moreover
\textbf{have} \( \vdash F \land G \implies \Box G \) by \textit{auto}
from this \textbf{h2} have \( \vdash \Box (F \land G) \implies \Box G \) by (rule \textit{STL4-2})
ultimately show \( \vdash \Box (F \land G) \implies \Box F \land \Box G \) by \textit{force}
\end{proof}
Finally, we derive STL6 (only semantically)

\[
\text{lemma STL6: } \vdash \Diamond \Diamond (F \land G) = (\Diamond \Diamond F \land \Diamond \Diamond G)
\]

\[\text{proof auto} \]

\[
\text{fix } w
\]

\[
\text{assume } a1: w \models \Diamond \Diamond F \text{ and } a2: w \models \Diamond \Diamond G
\]

\[\text{from } a1 \text{ obtain } nf \text{ where } nf: (w \mid s \cdot nf) \models \Diamond F \text{ by (auto simp: eventually-defs)} \]

\[\text{from } a2 \text{ obtain } ng \text{ where } ng: (w \mid s \cdot ng) \models \Diamond G \text{ by (auto simp: eventually-defs)} \]

\[\text{let } ?n = \text{max } nf \text{ ng} \]

\[\text{have } nf' \leq ?n \text{ by simp} \]

\[\text{from } this \text{ nf have } (w \mid s \cdot ?n) \models \Diamond F \text{ by (rule linalw)} \]
moreover have \( ng \leq ?n \) by simp from this \( ng \) have \( (w \mid_s ?n) \models \square G \) by (rule linalw) ultimately have \( (w \mid_s ?n) \models \square(F \land G) \) by (rule box-conjE) thus \( w \models \Diamond \square(F \land G) \) by (auto simp: eventually-defs)

next fix \( w \) assume \( h: w \models \Diamond \square(F \land G) \) have \( \vdash F \land G \rightarrow F \) by auto hence \( \vdash \Diamond \square(F \land G) \rightarrow \Diamond \square F \) by (rule STL4-eve[OF STL4]) with \( h \) show \( w \models \Diamond \square F \) by auto

next

fix \( w \) assume \( h: w \models \Diamond \square(F \land G) \) have \( \vdash F \land G \rightarrow G \) by auto hence \( \vdash \Diamond \square(F \land G) \rightarrow \Diamond \square G \) by (rule STL4-eve[OF STL4]) with \( h \) show \( w \models \Diamond \square G \) by auto

qed

lemma MM0: \( \vdash \square(F \rightarrow G) \rightarrow \square F \rightarrow \square G \)
proof
have \( \vdash \square(F \rightarrow G) \rightarrow \square G \) by (rule STL4) auto thus \( \vdash \square F \rightarrow \square G \) by (auto simp: STL5[int-rewrite])
qed

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

theorem MM2: \( \vdash \square(A \land \square(B \rightarrow C)) \rightarrow \square(A \land B \rightarrow C) \)
proof
have \( \vdash \square(A \land (B \rightarrow C)) \rightarrow \square(A \land B \rightarrow C) \) by (rule STL4) auto thus \( \vdash \square F \rightarrow \square G \) by (auto simp: STL5[int-rewrite])
qed

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

theorem MM4[simp-und]: \( \vdash \square #F = #F \)
proof (cases \( F \))
assume \( F \)
hence \( I: \vdash #F \) by auto
hence \( \vdash \square #F \) by (rule alw)
with \( I \) show \( ?thesis \) by force
next
assume \( \neg F \)
hence \( I: \vdash \neg #F \) by auto
from ax1 have \( \vdash \neg #F \rightarrow \neg \square #F \) by (rule lift-imp-neg)
with \( I \) show \( ?thesis \) by force

51
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 \( \Box \)thesis by auto
qed

theorem MM5: \( \vdash \Box F \lor \Box G \rightarrow \Box (F \lor G) \)
proof –
  have \( \vdash \Box F \rightarrow \Box (F \lor G) \) by (rule STL4) auto
  moreover
  have \( \vdash \Box G \rightarrow \Box (F \lor G) \) by (rule STL4) auto
  ultimately show \( \Box \)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) auto
  thus \( \Box \)thesis by simp
qed

lemma MM10:
  assumes h: \( \sim F = G \) shows \( \vdash \Box[F]-v = \Box[G]-v \)
  by (auto simp: h[int-rewrite])

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

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

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

52
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

**Theorems dualization-rew = dualization[int-rewrite]**
**Theorems dualization-unl = dualization[unlifted]**

**Theorem E1:** \[\vdash \Diamond (F \lor G) = (\Diamond F \lor \Diamond G)\]
**Proof** —
\[\begin{align*}
\text{have } & \vdash \Box \neg (F \lor G) = \Box (\neg F \land \neg G) \text{ by (rule MM1) auto} \\
\text{thus } & \text{?thesis unfolding eventually-def STL5[int-rewrite] by force}
\end{align*}\]
**Qed**

**Theorem E3:** \[\vdash F \longrightarrow \Diamond F\]
unfolding eventually-def by (force dest: ax1[unlift-rule])

**Theorem E4:** \[\vdash \Box F \longrightarrow \Diamond F\]
by (rule lift-imp-trans[OF ax1 E3])

**Theorem E5:** \[\vdash \Box F \longrightarrow \Box \Diamond F\]
**Proof** —
\[\begin{align*}
\text{have } & \vdash \Box \Box F \longrightarrow \Box \Diamond F \text{ by (rule STL4[OF E4])} \\
\text{thus } & \text{?thesis by simp}
\end{align*}\]
**Qed**

**Theorem E6:** \[\vdash \Box F \longrightarrow \Diamond \Box F\]
using E4[of TEMP \Box F] by simp

**Theorem E7:**
\begin{align*}
\text{assumes } & h: \neg F \land Unchanged v \longrightarrow \circ \neg F \\
\text{shows } & \neg \Diamond F \longrightarrow F \lor \circ \Diamond F
\end{align*}
**Proof** —
\[\begin{align*}
\text{from } & h \text{ have } \neg F \land \circ \Box \neg F \longrightarrow \Box \neg F \text{ by (rule M10)} \\
\text{thus } & \text{?thesis by (auto simp: eventually-def)}
\end{align*}\]
**Qed**

**Theorem E8:** \[\vdash \Diamond (F \longrightarrow G) \longrightarrow \Box F \longrightarrow \Diamond G\]
**Proof** —
\[\begin{align*}
\text{have } & \vdash \Box (F \land \neg G) \longrightarrow \Box \neg (F \longrightarrow G) \text{ by (rule STL4) auto} \\
\text{thus } & \text{?thesis unfolding eventually-def STL5[int-rewrite] by auto}
\end{align*}\]
**Qed**

**Theorem E9:** \[\vdash \Box (F \longrightarrow G) \longrightarrow \Diamond F \longrightarrow \Diamond G\]
**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]: ⊢ ♦¬#F = ¬#F 
by (simp add: eventually-def) 

theorem E15b[simp-unl]: ⊢ ♦¬#F = ¬#F 
by (simp add: eventually-def) 

theorem E16: ⊢ ♦□F → ♦F 
by (rule STL4-eve[OF ax1]) 

An action version of STL6 
lemma STL6-act: ⊢ ♦(□[F]-v ∧ □[G]-w) = (♦□[F]-v ∧ ♦□[G]-w) 
proof – 
  have ⊢ (♦□[F]-v ∧ □[G]-w)) = ♦(♦□[F]-v ∧ □□[G]-w) by (rule E22[OF STL5]) 
  thus ?thesis by (auto simp: STL6[int-rewrite]) 
qed 

lemma SE1: ⊢ □F ∧ ♦G → ♦(□F ∧ G) 
proof – 
  have ⊢ □¬(□F ∧ G) → □(□F → ¬G) by (rule STL4) auto 
  with MM0 show ?thesis by (force simp: eventually-def) 
qed 

lemma SE2: ⊢ □F ∧ ♦G → ♦(F ∧ G) 
proof – 
  have ⊢ □F ∧ G → F ∧ G by (auto elim: ax1[unlift-rule]) 
  hence ⊢ ♦(□F ∧ G) → ♦(F ∧ G) by (rule STL4-eve) 
  with SE1 show ?thesis by (rule lift-imp-trans) 
qed 

lemma SE3: ⊢ □F ∧ ♦G → ♦(G ∧ F) 
proof – 
  have ⊢ ♦(F ∧ G) → ♦(G ∧ F) by (rule STL4-eve) auto 
  with SE2 show ?thesis by (rule lift-imp-trans) 
qed
lemma SE4:
assumes h1: s |= □F and h2: s |= ◊G and h3: □F ∧ G → H
shows s |= ◊H
using h1 h2 h3 [THEN STL4-eve] SE1 by force

theorem E17: ⊢ □◊□F → □◊F
by (rule STL4 [OF STL4-eve [OF ax1]])

theorem E18: ⊢ □◊□F → □◊□F
by (rule ax1)

theorem E19: ⊢ □◊□F → □◊□F
by (rule STL4-eve [OF MM1])

proof -
have ⊢ (□F ∧ ¬□F) = #False by auto
hence ⊢ ◊□□F = ◊□□□F by (rule E22 [OF MM1])
thus ?thesis unfolding STL6 [int-rewrite] by (auto simp: eventually-def)
qed

theorem E20: ⊢ □◊F → □◊F
by (rule lift-imp-trans [OF E19 E17])

theorem E21: □◊□F = □◊F
by (rule int-iffI [OF E18 E19])

lemma E23: (s |= ◊F) → □◊F
using P1 by (force simp: eventually-def)

lemma E24: ⊢ ◊Q → □◊Q
by (rule lift-imp-trans [OF E20 P4])

lemma E25: ⊢ ◊(A) → □◊A
by (rule lift-imp-trans [OF E20 P4])

lemma E26: □◊(A) → □◊A
by (rule SE4 [OF E25])

lemma allBox: (s |= □(∀x. F x)) = (∀x. s |= □(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 $\vdash \Box F$
using $h2$ h1[THEN STL4-eve] by (rule fmp)

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

proof
have $\vdash \Box(F \rightarrow \Box F) \land \Box F \rightarrow \Box(\Box(F \rightarrow \Box F) \land F)$ by (rule SE1)
moreover
have $\vdash \Box(F \rightarrow \Box F) \land F \rightarrow \Box F$ using ax1[of TEMP F \rightarrow \Box F] by auto
hence $\vdash \Box(F \rightarrow \Box F) \land F \rightarrow \Box \Box F$ by (rule STL4-eve)
ultimately show ?thesis by (rule lift-imp-trans)
qed

lemma allActBox: $(s \models (\forall x. F x) \rightarrow v) = (\forall x. (s \models (F x) \rightarrow v))$

unfolding allActT[unlifted] ..

theorem exEE: $\vdash (\exists x. \diamond(F(x))) = \diamond(\exists x. F(x))$

proof
have $\vdash \neg(\exists x. \diamond(F(x))) = \neg(\diamond(\exists x. F(x)))$
by (auto simp: eventually-def Not-Rex[int-rewrite] allBox)
thus ?thesis by force
qed

theorem exActE: $\vdash (\exists x. \diamond(F(x)) \rightarrow v) = \diamond(\exists x. (F(x) \rightarrow v))$

proof
have $\vdash \neg(\exists x. \diamond(F(x)) \rightarrow v) = \neg(\diamond(\exists x. (F(x) \rightarrow v)))$
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: $\vdash F \leadsto F$
unfolding leadsto-def by (rule alw[OF E3])

theorem LT2: assumes $h: \vdash F \rightarrow G$ shows $\vdash F \rightarrow \diamond G$
bys (rule lift-imp-trans[OF h E3])

theorem LT3: assumes $h: \vdash F \rightarrow G$ shows $\vdash F \rightarrow G$
unfolding leadsto-def by (rule alw[OF LT2[OF h]])

theorem LT4: $\vdash (F \leadsto G) \rightarrow \diamond G$
unfolding leadsto-def using ax1[of TEMP F \rightarrow \diamond G] by auto

theorem LT5: $\vdash (\square(F \rightarrow \diamond G) \rightarrow \diamond F \rightarrow \diamond G$
using $E9[\text{of } F \text{ TEMP } \downarrow G]$ by simp

theorem LT6: $\vdash \Diamond F \rightarrow (F \rightsquivalence G) \rightarrow \Diamond G$
  unfolding leadsto-def using $LT5[\text{of } F \text{ G}]$ by auto

theorem LT9 [simp-unl]: $\vdash (F \rightsquivalence G) = (F \rightsquivalence G)$
  by (auto simp: leadsto-def)

proof -
  have $\vdash \Box \Diamond F \rightarrow \Box((F \rightsquivalence G) \rightarrow \Diamond G)$ by (rule STL4 [OF LT6])
  from lift-imp-trans [OF this MM0] show ?thesis by simp
qed

theorem LT7: $\vdash \Box \Diamond F \rightarrow (F \rightsquivalence G) \rightarrow \Box \Diamond G$
  unfolding leadsto-def by (rule STL4) auto

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

theorem LT11: $\vdash (F \rightsquivalence G) \rightarrow (F \rightsquivalence (G \lor H))$
  proof -
    have $\vdash G \rightsquivalence (G \lor H)$ by (rule LT3) auto
    with $LT13[\text{of } F \text{ G TEMP } (G \lor H)]$ show ?thesis by force
qed

theorem LT12: $\vdash (F \rightsquivalence H) \rightarrow (F \rightsquivalence (G \lor H))$
  proof -
    have $\vdash H \rightsquivalence (G \lor H)$ by (rule LT3) auto
    with $LT13[\text{of } F \text{ H TEMP } (G \lor H)]$ show ?thesis by force
qed

theorem LT14: $\vdash ((F \lor G) \rightsquivalence H) \rightarrow (F \rightsquivalence H)$
  unfolding leadsto-def by (rule STL4) auto

theorem LT15: $\vdash ((F \lor G) \rightsquivalence H) \rightarrow (G \rightsquivalence H)$
  unfolding leadsto-def by (rule STL4) auto

theorem LT16: $\vdash (F \rightsquivalence H) \rightarrow (G \rightsquivalence H) \rightarrow ((F \lor G) \rightsquivalence H)$
  proof -
    have $\vdash \Box(F \rightarrow \Diamond H) \rightarrow \Box((G \rightarrow \Diamond H) \rightarrow (F \lor G \rightarrow \Diamond H))$ by (rule STL4)
    auto
    from lift-imp-trans [OF this MM0] show ?thesis by (unfold leadsto-def)
qed
\textbf{theorem} \textit{LT17}: \( \vdash ((F \lor G) \rightsquigarrow H) = ((F \rightsquigarrow H) \land (G \rightsquigarrow H)) \)
\begin{itemize}
  \item by \textit{(auto elim: LT14[unlift-rule] LT15[unlift-rule] LT16[unlift-rule])}
\end{itemize}

\textbf{theorem} \textit{LT10}:
\begin{itemize}
  \item assumes \( \vdash (F \land \neg G) \rightsquigarrow G \)
  \item shows \( \vdash F \rightsquigarrow G \)
\end{itemize}
\begin{itemize}
  \item proof –
  \item from \( h \) have \( \vdash ((F \land \neg G) \lor G) \rightsquigarrow G \)
    \begin{itemize}
      \item by \textit{(auto simp: LT17[int-rewrite] LT1[int-rewrite])}
    \end{itemize}
  \item moreover have \( \vdash F \rightsquigarrow ((F \land \neg G) \lor G) \)
    \begin{itemize}
      \item by \textit{(rule LT3, auto)}
    \end{itemize}
  \item ultimately show \( ?\text{thesis} \)
    \begin{itemize}
      \item by \textit{(force elim: LT13[unlift-rule])}
    \end{itemize}
\end{itemize}
\textbf{qed}

\textbf{theorem} \textit{LT18}:
\begin{itemize}
  \item \( \vdash (A \rightsquigarrow (B \lor C)) \rightarrow (B \rightsquigarrow D) \rightarrow (C \rightsquigarrow D) \rightarrow (A \rightsquigarrow D) \)
\end{itemize}
\begin{itemize}
  \item proof –
  \item have \( \vdash (B \rightsquigarrow D) \rightarrow (C \rightsquigarrow D) \rightarrow ((B \lor C) \rightsquigarrow D) \)
    \begin{itemize}
      \item by \textit{(rule LT16)}
    \end{itemize}
  \item thus \( ?\text{thesis} \)
    \begin{itemize}
      \item by \textit{(force elim: LT13[unlift-rule])}
    \end{itemize}
\end{itemize}
\textbf{qed}

\textbf{theorem} \textit{LT19}:
\begin{itemize}
  \item \( \vdash (A \rightsquigarrow (D \lor B)) \rightarrow (B \rightsquigarrow D) \rightarrow (A \rightsquigarrow D) \)
\end{itemize}
\begin{itemize}
  \item using \textit{LT18[of A D B D] LT1[of D] by force}
\end{itemize}

\textbf{theorem} \textit{LT20}:
\begin{itemize}
  \item \( \vdash (A \rightsquigarrow (B \lor D)) \rightarrow (B \rightsquigarrow D) \rightarrow (A \rightsquigarrow D) \)
\end{itemize}
\begin{itemize}
  \item using \textit{LT18[of A B D D] LT1[of D] by force}
\end{itemize}

\textbf{theorem} \textit{LT21}:
\begin{itemize}
  \item \( \vdash ((\exists x. F x) \rightsquigarrow G) = (\forall x. (F x \rightarrow \Diamond G)) \)
\end{itemize}
\begin{itemize}
  \item proof –
  \item have \( \Box((\exists x. F x) \rightarrow \Diamond G) = (\forall x. (F x \rightarrow \Diamond G)) \)
    \begin{itemize}
      \item by \textit{(rule MM1 auto)}
    \end{itemize}
  \item thus \( ?\text{thesis} \)
    \begin{itemize}
      \item by \textit{(unfold leadsto-def allT[int-rewrite])}
    \end{itemize}
\end{itemize}
\textbf{qed}

\textbf{theorem} \textit{LT22}:
\begin{itemize}
  \item \( \vdash (F \rightsquigarrow (G \lor H)) \rightarrow \Box \neg G \rightarrow (F \rightsquigarrow H) \)
\end{itemize}
\begin{itemize}
  \item proof –
  \item have \( \Box \neg G \rightarrow (G \rightsquigarrow H) \)
    \begin{itemize}
      \item \textbf{unfolding leadsto-def by (rule STL4 auto)}
    \end{itemize}
  \item thus \( ?\text{thesis} \)
    \begin{itemize}
      \item by \textit{(force elim: LT20[unlift-rule])}
    \end{itemize}
\end{itemize}
\textbf{qed}

\textbf{lemma} \textit{LT23}:
\begin{itemize}
  \item \( \neg (P \rightarrow \Diamond Q) \rightarrow (P \rightarrow \Box Q) \)
\end{itemize}
\begin{itemize}
  \item by \textit{(auto dest: E23[unlift-rule])}
\end{itemize}

\textbf{theorem} \textit{LT24}:
\begin{itemize}
  \item \( \vdash \Box I \rightarrow ((P \land I) \rightsquigarrow Q) \rightarrow P \rightsquigarrow Q \)
\end{itemize}
\begin{itemize}
  \item proof –
  \item have \( \Box I \rightarrow \Box((P \land I) \rightarrow \Diamond Q) \rightarrow (P \rightarrow \Diamond Q) \)
    \begin{itemize}
      \item by \textit{(rule STL4 auto)}
    \end{itemize}
  \item from \textit{lift-imp-trans[of this MM0]} show \( ?\text{thesis} \)
    \begin{itemize}
      \item by \textit{(unfold leadsto-def)}
    \end{itemize}
\end{itemize}
\textbf{qed}

58
theorem LT25[simp-unl]: \( F \rightsquigarrow \#\text{False} = \square \neg F \)

unfolding leadsto-def proof (rule MM1)

show \( \vdash (F \implies \Diamond \#\text{False}) = \neg F \) by simp

qed

lemma LT28:

assumes \( h: \sim P \implies \circ P \vee \circ Q \)

shows \( \sim (P \implies \circ P) \vee \Diamond Q \)

using \( h \ E23[of \ Q] \) by force

lemma LT29:

assumes \( h1: \sim P \implies \circ P \vee \circ Q \) and \( h2: \sim P \wedge \text{Unchanged} v \implies \circ P \)

shows \( \Box N \implies P \implies \square P \vee \Diamond Q \)

proof –

from \( h1[\text{THEN LT28}] \) have \( \sim \Box \neg Q \implies (P \implies \circ P) \) unfolding eventually-def

by auto

hence \( \vdash \Box[\Box \neg Q]-v \implies \Box[P \implies \circ P]-v \) by (rule M2)

moreover

have \( \vdash \Diamond Q \implies \Box[\Box \neg Q]-v \) unfolding dualization-rev by (rule ax2)

moreover

note \( ax3[OF \ h2] \)

ultimately

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

qed

lemma LT30:

assumes \( h: \sim P \wedge N \implies \circ P \vee \circ Q \)

shows \( \sim N \implies (P \implies \circ P) \vee \Diamond Q \)

using \( h \ E23 \) by force

lemma LT31:

assumes \( h1: \sim P \wedge N \implies \circ P \vee \circ Q \) and \( h2: \sim P \wedge \text{Unchanged} v \implies \circ P \)

shows \( \Box N \implies P \implies \square P \vee \Diamond Q \)

proof –

from \( h1[\text{THEN LT30}] \) have \( \sim N \implies \Box \neg Q \implies P \implies \circ P \) unfolding eventually-def

by auto

hence \( \vdash \Box[N]-v \implies \Box[\Box \neg Q]-v \implies \Box[P \implies \circ P]-v \)

by \( \text{force intro: ax4[unlift-rule]} \)

with \( P4 \) have \( \vdash \Box N \implies \Box[\Box \neg Q]-v \implies \Box[P \implies \circ P]-v \) by (rule lift-imp-trans)

moreover

have \( \vdash \Diamond Q \implies \Box[\Box \neg Q]-v \implies \Box[P \implies \circ P]-v \) unfolding dualization-rev by (rule ax2)

moreover

note \( ax3[OF \ h2] \)

ultimately

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

qed

59
lemma LT33: \( \vdash (\#P \land F) \rightarrow G = (\#P \rightarrow (F \sim \rightarrow G)) \)
by (cases P, auto simp: leadsto-def)

lemma AA1: \( \vdash [\#\text{False}] \rightarrow \neg \langle Q \rangle \rightarrow \)
unfolding dualization-rev by (rule M2) auto

proof
have \( \vdash [P] \rightarrow \neg \langle P \land Q \rangle \rightarrow \neg \langle Q \rangle \rightarrow \)
by (rule sq) (auto simp: actrans-def)

hence \( \vdash [P] \rightarrow \neg \langle P \land Q \rangle \rightarrow \neg \langle Q \rangle \rightarrow \)
by (force intro: ax4[unlift-rule])
thus \( \vdash \) thesis by (auto simp: angle-action-def)

qed

lemma AA2: \( \vdash [P] \rightarrow [P \land Q] \rightarrow [\langle Q \rangle \rightarrow \]
proof
have \( \vdash \)
by (auto dest: P4[unlift-rule] simp: M8[int-rewrite])

moreover
have \( \vdash \)
by (rule M2) auto

ultimately have \( \vdash \)
by (rule lift-imp-trans)

moreover
have \( \vdash \)
by (rule STL4-eve) auto

hence \( \vdash \)
by (force dest: E25[unlift-rule])

with AA2 have \( \vdash \)
by (rule lift-imp-trans)

ultimately show \( \vdash \) thesis by force

qed

lemma AA3: \( \vdash \langle A \rangle \rightarrow [P \land \langle A \rangle \rightarrow \]
unfolding angle-action-def angle-actrans-def using T5 by force

lemma AA4: \( \vdash \langle A \rangle \rightarrow [P \land \langle A \rangle \rightarrow \]

proof
have \( \vdash \)
by (rule AA7) auto

with AA2 show \( \vdash \) thesis by (rule lift-imp-trans)

qed

lemma AA5: \( \vdash \langle A \rangle \rightarrow [P \land \langle A \rangle \rightarrow \]

proof
have \( \vdash \)
by (rule AA2) auto

with P5 show \( \vdash \) thesis by force

qed
lemma AA9: \( \vdash \Box[P]-v \land \Diamond(A)-v \rightarrow \Diamond([P]-v \land A)-v \)
proof
  have \( \vdash \Box[P]-v \land \Diamond(A)-v \rightarrow \Diamond([P]-v \land A)-v \) by (rule AA2)
thus ?thesis by simp
qed

lemma AA10: \( \vdash \neg(\Box[P] \land \Diamond(\neg P)-v) \)
unfolding angle-action-def by auto

lemma AA11: \( \vdash \neg(\langle v$ = $v \rangle)-v \)
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 \( \vdash \Box[v$ \neq $v \land \neg(P \land A)]-v \rightarrow \Box[\neg([P]-v \land A)]-v \)
  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 AA18: \( \vdash \Box P \land \Diamond(A)-v \rightarrow \Diamond(P \land A)-v \)
using P4 by (force intro: AA2[unlift-rule])

lemma AA20:
  assumes h1: \( \neg P \rightarrow oP \lor oQ \)
  and h2: \( \neg P \land A \rightarrow oQ \)
  and h3: \( \neg P \land Unchanged w \rightarrow oP \)
  shows \( \vdash \Box[oP \lor oQ \rightarrow oP \Rightarrow Q] \)
proof
  --
lemma AA21: \(\neg \langle\neg F\rangle \rightarrow \neg \neg \neg F\)

using pax5[of TEMP \neg F v] unfolding angle-action-def eventually-def by auto

lemma AA22:

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

and h2: \(\neg P \land N \land \langle A\rangle \rightarrow \neg Q\)

and h3: \(\neg P \land \text{Unchanged} w \rightarrow \neg P\)

shows \(\neg \Box P \land \neg \Box (\neg P \land \langle A\rangle \rightarrow \neg Q)\)

proof –

from h2 have \(\neg (\neg P \land A \rightarrow Q)\) by (auto simp: angle-action-def sem[int-rewrite])

from pref-imp-trans[of this E23] have \(\neg \langle\neg (N \land P) \land A\rangle \rightarrow \neg Q\)

by (rule AA7)

hence \(\neg \langle\neg (N \land P) \land A\rangle \rightarrow \neg Q\) by (force dest: E25[unlift-rule])

with AA19 have \(\neg \Box (N \land P) \land \langle A\rangle \rightarrow \neg Q\)

by (rule lift-imp-trans)

hence \(\neg \Box (N \land P \land \langle A\rangle \rightarrow \neg Q)\)

by (auto simp: STL5[int-rewrite])

with LT31[of h1 h3] have \(\neg \Box N \land (\neg P \rightarrow \neg \langle A\rangle \rightarrow \neg Q)\)

by force

hence \(\neg \Box (\neg P \rightarrow \neg \langle A\rangle \rightarrow \neg Q)\)

by (rule STL4)

thus \(?thesis\) by (simp add: leadsto-def STL5[int-rewrite])

qed

lemma AA23:

assumes \(\neg P \land N \rightarrow \neg P \lor \neg Q\)

and \(\neg P \land N \land \langle A\rangle \rightarrow \neg Q\)

and \(\neg P \land \text{Unchanged} w \rightarrow \neg P\)

shows \(\neg \Box N \land \neg \Box \langle A\rangle \rightarrow \neg (P \rightarrow Q)\)

proof –

have \(\neg \Box \langle A\rangle \rightarrow \neg \Box (\neg P \land \langle A\rangle \rightarrow \neg Q)\) by (rule STL4) auto

with AA22[of assms] show \(?thesis\) by force

qed

lemma AA25:

assumes h: \(\neg \langle P\rangle \rightarrow \langle Q\rangle\)

shows \(\neg \langle \neg \langle P\rangle \rightarrow \langle Q\rangle\) by (intro AA7) (auto simp: angle-action-def actrans-def)

with AA4 have \(\neg \langle P\rangle \rightarrow \neg \langle \neg \langle P\rangle \rightarrow \langle Q\rangle\)

by force

62
from this AA7[OF h] have ⊢ ◊(P)-v → ◊(Q)-w by (rule lift-imp-trans)
thus ?thesis by simp
qed

lemma AA26:
assumes h: \sim ⟨A⟩-v = ⟨B⟩-w
shows ⊢ ◊⟨A⟩-v = ◊⟨B⟩-w
proof –
from h have \sim ⟨A⟩-v --> ⟨B⟩-w by auto
hence ⊢ ⟨A⟩-v --> ⟨B⟩-w by (rule AA25)
moreover from h have \sim ⟨B⟩-w --> ⟨A⟩-v by auto
hence ⊢ ⟨B⟩-w --> ⟨A⟩-v by (rule AA25)
ultimately show ?thesis by force
qed

theorem AA28[simp-unl]: ⊢ □♦[N]-v ∧ □♦⟨A⟩-v --> □♦[N ∧ A]-v
proof –
have ⊢ □[N]-v ∧ ◊⟨A⟩-v --> □◊⟨N ∧ A⟩-v by (rule STL4[OF AA2])
thus ?thesis by (simp add: STL5[int-rewrite])
qed

lemma AA29[simp-unl]: ⊢ ◊⟨♦[P]-f⟩-f = ◊⟨P⟩-f
unfolding eventually-def angle-action-def by simp

theorem AA30[simp-unl]: ⊢ ◊⟨◦F⟩-v = ◊⟨F⟩-v
using E27[of TEMP ◊⟨A⟩-v] by simp

lemmas next-and = T8

lemma next-or: \sim (F \lor G) = (◦F \lor ◦G)
by (simp add: h[int-rewrite])

lemmas next-or = T8
next

have \( \sim \circ F \rightarrow \circ(F \lor G) \) by (rule N1) auto
moreover have \( \sim \circ G \rightarrow \circ(F \lor G) \) by (rule N1) auto
ultimately show \( \sim \circ F \lor \circ G \rightarrow \circ(F \lor G) \) by force
qed

lemma next-imp: \( \sim \circ(F \rightarrow G) = (\circ F \rightarrow \circ G) \)
proof (rule pref-iffI)
have \( \sim \circ G \rightarrow \circ(F \rightarrow G) \) by (rule N1) auto
moreover have \( \sim \circ F \rightarrow \circ(F \rightarrow G) \) by (rule N1) auto
ultimately show \( \sim (\circ F \rightarrow \circ G) \rightarrow \circ(F \rightarrow G) \) by force
qed (rule pax2)

lemmas next-not = pax1

lemma next-eq: \( \sim \circ(F = G) = (\circ F = \circ G) \)
proof
have \( \sim \circ(F = G) = \circ((F \rightarrow G) \land (G \rightarrow F)) \) 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: \( \sim \circ(F \neq G) = (\circ F \neq \circ G) \)
  by (simp add: next-eq[int-rewrite])

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

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

lemma next-fun3: \( \sim \circ<\times,\y,\z> = \circ<\circ\times,\circ\y,\circ\z> \)
  by (auto simp: nexts-def)
**Lemma** `next-fun4`: \( \sim o<x,y,z,,x> = f<x,o,y,o,,x> \)
by (auto simp: nexts-def)

**Lemma** `next-forall`: \( \sim o(\forall x. P x) = (\forall x. o P x) \)
by (auto simp: nexts-def)

**Lemma** `next-exists`: \( \sim o(\exists x. P x) = (\exists x. o P x) \)
by (auto simp: nexts-def)

**Lemma** `next-exists1`: \( \sim o(\exists! x. P x) = (\exists! x. o P x) \)
by (auto simp: nexts-def)

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

lemmas `next-commutes`[int-rewrite] =
  `next-and` `next-or` `next-imp` `next-eq`
  `next-fun1` `next-fun2` `next-fun3` `next-fun4`
  `next-forall` `next-exists` `next-exists1`

lemmas `ifs-eq`[int-rewrite] =
  `after-fun3` `next-fun3` `before-fun3`

lemmas `next-always` = `pax3`

**Lemma** `t1`:
\( \sim o$x = x$ \)
by (simp add: before-def after-def nexts-def first-tail-second)

Theorem `next-eventually` should not be used "blindly".

**Lemma** `next-eventually`:
assumes `h`: `stutinv F`
shows `\sim □ F \longrightarrow \neg F \longrightarrow o□ F`
proof –
  from `h` have `I`: `stutinv (TEMP \neg F)` by (rule stat-not)
  have `\sim □ \neg F = (\neg F \land o □ \neg F)` unfolding `TT[OF pre-id-unch[OF 1], int-rewrite]`
  by simp
  thus ?thesis by (auto simp: eventually-def)
qed

**Lemma** `next-action`:
\( \sim □ [P] \longrightarrow o □ [P] \)
using `pax4[of P v]` by auto

### 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 `TLA1`, `TLA2`, `INV1` and `INV2` are derived, where `\sim` is used where appropriate.

**Theorem** `TLA1`:
assumes `H`: \( \sim P \land Unchanged f \longrightarrow oP \)
shows $\vdash \Box P = (P \land \Box P \to \circ P)$

proof (rule int-ifI)

from ax1[of P] M0[of P f] show $\vdash \Box P \to P \land \Box [P \to \circ P]$ by force

next

from ax3[OF H] show $\vdash P \land \Box [P \land \circ P] \to \Box P$ by auto

qed

theorem TLA2:

assumes h1: $\vdash P \to Q$

and h2: $\vdash P \land \circ P \land [A] \to [B] g$

shows $\vdash \Box P \land \Box [A] \land [B] g$

proof

from h2 have $\vdash \Box [P \land \circ P \land [A] \to [B] g]$ by (rule M2)

hence $\vdash \Box P \land \Box [A] \land [B] g$ by (rule int-rewrite)

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

with h1[THEN STL4] show $?thesis$ by force

qed

theorem INV1:

assumes H: $\vdash \sim I \land [N] \to \circ I$

shows $\vdash I \land \Box [N] \to \Box I$

proof

from H have $\vdash \sim [N] \to I \to \circ I$ by auto

hence $\vdash \Box [N] \to \Box I \to \circ I$ by (rule M2)

moreover

from H have $\vdash \sim I \land \text{Unchanged} f \to \circ I$ by (auto simp: actrans-def)

hence $\vdash \Box I \to \Box I \to \Box I$ by (rule ax3)

ultimately show $?thesis$ by force

qed

theorem INV2: $\vdash \Box I \to \Box [N] \to \Box [N \land I \land \circ I]$

proof

from M1[of I f] have $\vdash \Box I \to (\Box [N] \to \Box [N] \land I \land \circ I)$ by auto

thus $?thesis$ by (auto simp: M8[int-rewrite])

qed

lemma R1:

assumes H: $\vdash \sim \text{Unchanged} w \to \text{Unchanged} v$

shows $\vdash \Box [F] \to \Box [F] \to v$

proof

from H have $\vdash \sim [F] \to [F] \to \text{Unchanged} w$ by (auto simp: actrans-def)

thus $?thesis$ by (rule M11)

qed

theorem invmono:

assumes h1: $\vdash I \to P$

and h2: $\vdash P \land [N] \to \circ P$

shows $\vdash I \land \Box [N] \to \Box P$

using h1 INV1[OF h2] by force
theorem preimpsplit:
assumes \( \sim I \land N \rightarrow Q \)
and \( \sim I \land Unchanged\ v \rightarrow Q \)
shows \( \sim I \land [N]\cdot v \rightarrow Q \)
using assms[unlift-rule] by (auto simp: actrans-def)

theorem refinement1:
assumes \( h1: \vdash P \rightarrow Q \)
and \( h2: \vdash \sim I \land \circ I \land [A]\cdot f \rightarrow [B]\cdot g \)
shows \( \vdash P \land \Box I \land \Box [A]\cdot f \rightarrow Q \land \Box [B]\cdot g \)
proof –
  have this \( h2 \) have \( \vdash \Box I \land \Box [A]\cdot f \rightarrow \Box \# True \land \Box [B]\cdot g \) by (rule TLA2)
  with \( h1 \) show \(?thesis\) by force
qed

theorem inv-join:
assumes \( \vdash P \rightarrow \Box Q \) and \( \vdash P \rightarrow \Box R \)
shows \( \vdash P \rightarrow \Box (Q \land R) \)
using assms[unlift-rule] unfolding STL5[int-rewrite] by force

lemma inv-cases: \( \vdash \Box (A \rightarrow B) \land \Box (\sim A \rightarrow B) \rightarrow \Box B \)
proof –
  have \( \vdash \Box ((A \rightarrow B) \land (\sim A \rightarrow B)) \rightarrow \Box B \) 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 :: \( \cdot a \) formula \( \Rightarrow \cdot a \) formula
where enabled \( F \equiv \lambda s. \exists t. ((first s) \#\# t) \vdash F \)
syntax -Enabled :: lift \Rightarrow lift ((Enabled -)[90] 90)
translations -Enabled \( \Rightarrow \) CONST enabled

definition WeakF :: \( \cdot a::world \) formula \( \Rightarrow \cdot (a,\ b) \) stfun \( \Rightarrow \cdot a \) formula
where WeakF \( F \ v \equiv TEMP \odot \Box \text{Enabled} \ (F)\cdot v \rightarrow \Box \odot (F)\cdot v \)
definition StrongF :: \( \cdot a::world \) formula \( \Rightarrow \cdot (a,\ b) \) stfun \( \Rightarrow \cdot a \) formula
where StrongF \( F \ v \equiv TEMP \Box \odot \text{Enabled} \ (F)\cdot v \rightarrow \Box \Box (F)\cdot v \)
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** :: \([\text{lift}, \text{lift}] \Rightarrow \text{lift } ((WF'(-')(-)) [20,1000] 90)\)
- **SF** :: \([\text{lift}, \text{lift}] \Rightarrow \text{lift } ((SF'(-')(-)) [20,1000] 90)\)
- **WFsp** :: \([\text{lift}, \text{lift}] \Rightarrow \text{lift } ((WF'(-')(-)) [20,1000] 90)\)
- **SFsp** :: \([\text{lift}, \text{lift}] \Rightarrow \text{lift } ((SF'(-')(-)) [20,1000] 90)\)

**translations**

- **WF** \(\equiv\) CONST WeakF
- **SF** \(\equiv\) CONST StrongF
- **WFsp** \(\rightarrow\) CONST WeakF
- **SFsp** \(\rightarrow\) CONST StrongF

### 6.1 Properties of -Enabled

**theorem** enabledI: \(\vdash F \rightarrow \text{Enabled } F\)

**proof** (clarsimp)

fix \(w\)

assume \(w \models F\)

with seq-app-first-tail[of \(w\)] have ((first \(w\) ## tail \(w\)) \(\models F\)) by simp

thus \(w \models \text{Enabled } F\) by simp (auto simp: enabled-def)

qed

**theorem** enabledE:

assumes \(s \models \text{Enabled } F\) and \(\bigwedge u. (\text{first } s \# \# u) \models F \implies Q\)

shows \(Q\)

using assms unfolding enabled-def by blast

**lemma** enabled-mono:

assumes \(w \models \text{Enabled } F\) and \(\vdash F \rightarrow G\)

shows \(w \models \text{Enabled } G\)

using assms[unlifted] unfolding enabled-def by blast

**lemma** Enabled-disj1: \(\vdash \text{Enabled } F \rightarrow \text{Enabled } (F \lor G)\)

by (auto simp: enabled-def)

**lemma** Enabled-disj2: \(\vdash \text{Enabled } F \rightarrow \text{Enabled } (G \lor F)\)

by (auto simp: enabled-def)

**lemma** Enabled-conj1: \(\vdash \text{Enabled } (F \land G) \rightarrow \text{Enabled } F\)

by (auto simp: enabled-def)
lemma Enabled-conj2: \( \vdash \text{Enabled} (G \land F) \rightarrow \text{Enabled} F \)
by (auto simp: enabled-def)

lemma Enabled-disjD: \( \vdash \text{Enabled} (F \lor G) \rightarrow \text{Enabled} F \lor \text{Enabled} G \)
by (auto simp: enabled-def)

lemma Enabled-disj: \( \vdash \text{Enabled} (F \lor G) = (\text{Enabled} F \lor \text{Enabled} G) \)
by (auto simp: enabled-def)

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

lemma Enabled-ex: \( \vdash \text{Enabled} (\exists x. F x) = (\exists x. \text{Enabled} (F x)) \)
by (force simp: enabled-def)

6.2 Fairness Properties

lemma WF-alt: \( \vdash \text{WF}(A)-v = (\square \Diamond \neg \text{Enabled} (A)-v \lor \square \Diamond (A)-v) \)
proof
- have \( \vdash \text{WF}(A)-v = (\neg \Diamond \square \neg \text{Enabled} (A)-v \lor \square \Diamond (A)-v) \)
  by (auto simp: WeakF-def)
thus \( \text{thesis} \)
  by (simp add: dualization-rew)
qed

lemma SF-alt: \( \vdash \text{SF}(A)-v = (\Diamond \square \neg \text{Enabled} (A)-v \lor \square \Diamond (A)-v) \)
proof
- have \( \vdash \text{SF}(A)-v = (\neg \square \Diamond \neg \text{Enabled} (A)-v \lor \square \Diamond (A)-v) \)
  by (auto simp: StrongF-def)
thus \( \text{thesis} \)
  by (simp add: dualization-rew)
qed

lemma alwaysWFI: \( \vdash \text{WF}(A)-v \rightarrow \square \text{WF}(A)-v \)
unfolding WF-alt[int-rewrite] by (rule MM6)

theorem WF-always[simp-unl]: \( \vdash \square \text{WF}(A)-v = \text{WF}(A)-v \)
by (rule int-iffI[OF ax1 alwaysWFI])

theorem WF-eventually[simp-unl]: \( \vdash \Diamond \text{WF}(A)-v = \text{WF}(A)-v \)
proof
- have \( 1: \vdash \neg \text{WF}(A)-v = (\Diamond \square \neg \text{Enabled} (A)-v \land \neg \square \Diamond (A)-v) \)
  by (auto simp: WeakF-def)
  have \( \vdash \square \neg \text{WF}(A)-v = \neg \text{WF}(A)-v \)
  by (simp add: 1[int-rewrite] STL5[int-rewrite] dualization-rew)
thus \( \text{thesis} \)
  by (auto simp: eventually-def)
qed

lemma alwaysSFI: \( \vdash \text{SF}(A)-v \rightarrow \square \text{SF}(A)-v \)
proof
- have \( \vdash \square \Diamond \neg \text{Enabled} (A)-v \lor \square \Diamond (A)-v \rightarrow \square (\square \Diamond \neg \text{Enabled} (A)-v \lor \square \Diamond (A)-v) \)
  by (rule MM6)
thus \( \text{thesis} \)
  unfolding SF-alt[int-rewrite] by simp
qed

**Theorem**: $SF$-always $[\text{simp-unl}]: \vdash \Box SF(A) \dashv \vdash SF(A) \dashv$

by (rule int-iffI[OF ax1 alwaysSF])

**Theorem**: $SF$-eventually $[\text{simp-unl}]: \vdash \Diamond SF(A) \dashv \vdash SF(A) \dashv$

proof –
  have 1: $\vdash \neg SF(A) \dashv \vdash (\Diamond \Diamond \text{Enabled} \langle A \rangle \dashv \vdash \neg \Box \Diamond \langle A \rangle \dashv)$
    by (auto simp: StrongF-def)
  have $\vdash \Box \neg SF(A) \dashv \vdash \neg SF(A) \dashv$
    by (simp add: I[int-rewrite] STL5[int-rewrite] dualization-rew)
  thus $\neg$thesis
    by (auto simp: eventually-def)
qed

**Lemma**: enabled-$WFSF$: $\vdash \Box \text{Enabled} \langle F \rangle \dashv \vdash (WF(F) \dashv \vdash SF(F) \dashv)$

proof –
  have $\vdash \Box \text{Enabled} \langle F \rangle \dashv \vdash \Diamond \Box \text{Enabled} \langle F \rangle \dashv$ by (rule E3)
  hence $\vdash \Box \text{Enabled} \langle F \rangle \dashv \vdash WF(F) \dashv \vdash SF(F) \dashv$ by (auto simp: WeakF-def StrongF-def)
  moreover
  have $\vdash \Box \text{Enabled} \langle F \rangle \dashv \vdash \Diamond \Box \text{Enabled} \langle F \rangle \dashv$ by (rule STL4[OF E3])
  hence $\vdash \Box \text{Enabled} \langle F \rangle \dashv \vdash WF(F) \dashv \vdash SF(F) \dashv$ by (auto simp: WeakF-def StrongF-def)

ultimately show $\neg$thesis by force
qed

**Theorem**: $WF$-general:

assumes $h1$: $\neg P \times N \dashv \vdash \circ P \times \circ Q$
  and $h2$: $\neg P \times N \times \langle A \rangle \dashv \vdash \circ Q$
  and $h3$: $\vdash P \times N \dashv \vdash \text{Enabled} \langle A \rangle \dashv$
  and $h4$: $\neg P \times \text{Unchanged} \times w \dashv \vdash \circ P$

shows $\vdash \Box N \times WF(A) \dashv \vdash (P \dashv \vdash Q)$

proof –
  have $\vdash \Box \Box N \times WF(A) \dashv \vdash (\Box \Box P \dashv \vdash (\Box \langle A \rangle \dashv)$
  proof (rule STL4)
    have $\vdash \Box (P \times N) \dashv \vdash \Diamond \Box \text{Enabled} \langle A \rangle \dashv$ by (rule lift-imp-trans[OF h3[OF THEN STL4] E3])
    hence $\vdash \Box P \times \Box N \times WF(A) \dashv \vdash (\Box \Diamond \langle A \rangle \dashv$ by (auto simp: WeakF-def STL5[int-rewrite])
      with ax1[OF TEMP $\Diamond \langle A \rangle \dashv$]
    show $\vdash \Box N \times WF(A) \dashv \vdash (\Box P \dashv \vdash \langle A \rangle \dashv$
      by force
  qed
  hence $\vdash \Box N \times WF(A) \dashv \vdash (\Box \Box P \dashv \vdash (\Box \langle A \rangle \dashv$
    by (simp add: STL5[int-rewrite])
  with $AA22[OF h1 h2 h4]$ show $\neg$thesis by force
qed

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

**theorem WF1:**

assumes h1: \( \sim P \land [N]\cdot v \rightarrow \circ P \lor \circ Q \)
and h2: \( \sim P \land (N \land A)\cdot v \rightarrow \circ Q \)
and h3: \( \vdash P \rightarrow \mathit{Enabled \ (A)}\cdot v \)
and h4: \( \sim P \land \mathit{Unchanged} v \rightarrow \circ P \)

shows \( \vdash [N]\cdot v \land WF(A)\cdot v \rightarrow (P \leadsto Q) \)

**proof**

have \( \vdash [N]\cdot v \land WF(A)\cdot v \rightarrow (P \leadsto Q) \)

**proof** (rule WF1-general)

from h1 T9[of N v] show \( \sim P \land [N]\cdot v \rightarrow \circ P \lor \circ Q \) by force

next

from T9[of N v] have \( \sim P \land [N]\cdot v \land (A)\cdot v \rightarrow P \land (N \land A)\cdot v \)

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

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

next

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

qed (rule h4)

thus \(?thesis by simp\)

qed

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

**theorem SF1-general:**

assumes h1: \( \sim P \land N \rightarrow \circ P \lor \circ Q \)
and h2: \( \sim P \land N \land (A)\cdot v \rightarrow \circ Q \)
and h3: \( \vdash \mathit{Box} P \land \mathit{Box} N \land \mathit{Box} F \rightarrow \mathit{Enabled \ (A)}\cdot v \)
and h4: \( \sim P \land \mathit{Unchanged} v \rightarrow \circ P \)

shows \( \vdash \mathit{Box} N \land SF(A)\cdot v \land \mathit{Box} F \rightarrow (P \leadsto Q) \)

**proof**

have \( \vdash \mathit{Box} N \land SF(A)\cdot v \land \mathit{Box} F \rightarrow \mathit{Box} P \rightarrow \mathit{Enabled \ (A)}\cdot v \)

by force

qed

hence \( \vdash \mathit{Box} N \land SF(A)\cdot v \land \mathit{Box} F \rightarrow \mathit{Box} P \rightarrow \mathit{Enabled \ (A)}\cdot v \)

by (simp add: STL5[int-rewrite])

with ax1[of TEMP \( \mathit{Enabled \ (A)}\cdot v \)] show \( \vdash \mathit{Box} N \land SF(A)\cdot v \land \mathit{Box} F \rightarrow \mathit{Box} P \rightarrow \mathit{Enabled \ (A)}\cdot v \)

by force

qed

**theorem SF1:**

assumes h1: \( \sim P \land [N]\cdot v \rightarrow \circ P \lor \circ Q \)
and h2: \( \sim P \land (N \land A)\cdot v \rightarrow \circ Q \)

71
and \( h3: \vdash \Box P \land \Box[N]-v \land \Box F \rightarrow \Diamond Enabled \langle A \rangle-v \)
and \( h4: \vdash \sim P \land Unchanged \sim v \rightarrow oP \)
shows \( \vdash \Box[N]-v \land SF(A)-v \land \Box F \rightarrow (P \rightarrow Q) \)
proof
| have \( \vdash \Box[N]-v \land SF(A)-v \land \Box F \rightarrow (P \rightarrow Q) \)
| proof (rule SF1-general)
| from \( h1 \) \( T9[\text{of } N \sim v] \) show \( \vdash P \land \Box[N]-v \rightarrow oP \lor oQ \) by force
next
| from \( T9[\text{of } N \sim v] \) have \( \vdash P \land \Box[N]-v \land (\langle A \rangle-v \rightarrow P \land (N \land A)-v) \)
| by (auto simp: actrans-def angle-actrans-def)
| from this \( h2 \) show \( \vdash P \land \Box[N]-v \land (\langle A \rangle-v \rightarrow oQ \) by (rule pref-imp-trans)
next
| from \( h3 \) show \( \vdash \Box P \land \Box[N]-v \land \Box F \rightarrow \Diamond Enabled \langle A \rangle-v \) by simp
| qed (rule \( h4 \))
| thus \( \sim \)thesis by simp
| qed

Lamport proposes the following rule as an introduction rule for \textit{WF} formulas.

\textbf{theorem} \textit{WF2}:

\textbf{assumes} \( h1: \vdash \sim (N \land B).f \rightarrow (M).g \)
\textbf{and} \( h2: \vdash P \land \\ oP \land (N \land A).f \rightarrow B \)
\textbf{and} \( h3: \vdash P \land Enabled \langle M \rangle.g \rightarrow Enabled \langle A \rangle.f \)
\textbf{and} \( h4: \vdash \Box[N \land \sim B].f \land WF(A).f \land \Box F \land \Diamond \Diamond Enabled \langle M \rangle.g \rightarrow \Diamond \Box P \)
\textbf{proof} –
| have \( \vdash \Box[N].f \landWF(A).f \land \Box F \land \Diamond \Diamond Enabled \langle M \rangle.g \land \sim \Diamond \Diamond \langle M \rangle.g \rightarrow \)
| \( \Diamond \Diamond \langle M \rangle.g \)
| proof –
| have \( 1: \vdash \Box[N].f \landWF(A).f \land \Box F \land \Diamond \Diamond Enabled \langle M \rangle.g \land \sim \Diamond \Diamond \langle M \rangle.g \rightarrow \)
| \( \Diamond \Box P \)
| proof –
| have \( A: \vdash \Box[N]-f \landWF(A).f \land \Box F \land \Diamond \Diamond Enabled \langle M \rangle.g \land \sim \Diamond \Diamond \langle M \rangle.g \rightarrow \)
| \( \Diamond \Box[N].f \landWF(A).f \land \Box F \land \Diamond \Diamond Enabled \langle M \rangle.g \land \Box \langle \sim M \rangle.g \)
| unfolding \textit{STL6[\text{int-rewrite]}}
| by (auto simp: \textit{STL5[\text{int-rewrite]}} \textit{dualization-rew})
| have \( B: \vdash \Box[\Box[N].f \landWF(A).f \land \Box F] \land \Diamond \Box \Diamond \Diamond Enabled \langle M \rangle.g \land \Box \langle \sim M \rangle.g \)
| \( \rightarrow \)
| \( \Diamond \Box (\Box[N].f \landWF(A).f \land \Box F) \land \Diamond \Diamond \Diamond Enabled \langle M \rangle.g \land \Box \langle \sim M \rangle.g \)
| by (rule \textit{SE2})
| from \textit{lift-imp-trans[\textit{OF A B]}}
| have \( \vdash \Box[N].f \landWF(A).f \land \Box F \land \Diamond \Diamond Enabled \langle M \rangle.g \land \sim \Diamond \Diamond \langle M \rangle.g \rightarrow \)
| \( \Diamond \Box (\Box[N].f \landWF(A).f \land \Box F) \land \Diamond \Diamond \Diamond Enabled \langle M \rangle.g \land \Box \langle \sim M \rangle.g \)
| by (simp add: \textit{STL5[\text{int-rewrite]}])
| moreover
| from \( h1 \) have \( \vdash \sim \langle N \rangle.f \rightarrow \sim \langle M \rangle.g \rightarrow [N \land \sim B].f \) by (auto simp: actrans-def angle-actrans-def)
| hence \( \vdash \Box[N].f \landWF(A).f \land \Box F \land \Diamond \Diamond Enabled \langle M \rangle.g \land \sim \Diamond \Diamond \langle M \rangle.g \rightarrow \)
| \( \Diamond \Box [\sim M].f \landWF(A).f \land \Box F] \land \Diamond \Diamond \Diamond Enabled \langle M \rangle.g \land \Box \langle \sim M \rangle.g \)
| by (rule \textit{M2})
| from \textit{lift-imp-trans[\textit{OF this ax4}]} have \( \vdash \Box[N].f \land \Box \langle \sim M \rangle.g \rightarrow \Box [N \land \sim B].f \)
by (force intro: T4[unlift-rule])

with h4 have \( \vdash (\Box[N] \cdot f \land WF(A) \cdot f \land \Box F) \land (\Diamond \Box \text{Enabled} \langle M \rangle \cdot g \land \Box \neg M \cdot g) \)

\( \rightarrow \Box \Box P \)

by force

from STL4-eve[OF this]

have \( \vdash (\Box[N] \cdot f \land WF(A) \cdot f \land \Box F) \land (\Diamond \Box \text{Enabled} \langle M \rangle \cdot g \land \Box \neg M \cdot g) \)

\( \rightarrow \Box \Box P \) by simp

ultimately

show \(?thesis by (rule lift-imp-trans)

qed

have 2: \( \vdash \Box[N] \cdot f \land WF(A) \cdot f \land \Diamond \Box \text{Enabled} \langle M \rangle \cdot g \land \Box P \rightarrow \Box \Box \langle M \rangle \cdot g \)

proof –

have A: \( \vdash \Diamond \Box P \land \Diamond \Box \text{Enabled} \langle M \rangle \cdot g \land WF(A) \cdot f \rightarrow \Box \Box \langle A \rangle \cdot f \)

using h3[THEN STL4 , THEN STL4-eve] by (auto simp: STL6[int-rewrite]

WeakF-def)

have B: \( \vdash \Box[N] \cdot f \land \Box \Box P \land \Box \Box \langle A \rangle \cdot f \rightarrow \Box \Box \langle (P \land \circ P) \land (N \land A) \rangle \cdot f \)

by (force intro: AA29[unlift-rule])

hence \( \vdash \Diamond \Box \Box \langle (P \land \circ P) \land (N \land A) \rangle \cdot f \)

by (rule STL4-eve[OF STL4])

hence \( \vdash \Diamond \Box P \land \Diamond \Box \langle N \land A \rangle \cdot f \rightarrow \Box \Box \langle (P \land \circ P) \land (N \land A) \rangle \cdot f \)

by (simp add: STL6[int-rewrite])

with AA29[of N f A]

have B1: \( \vdash \Box[N] \cdot f \land \Box \Box P \land \Box \Box \langle A \rangle \cdot f \rightarrow \Box \Box \langle (P \land \circ P) \land (N \land A) \rangle \cdot f \)

by force

from h2 have \( \neg \langle (P \land \circ P) \land (N \land A) \rangle \cdot f \rightarrow \langle N \land B \rangle \cdot f \)

by (auto simp: angle-actrans-sem[unlifted])

from B1 this[THEN AA25, THEN STL4] have \( \vdash \Box[N] \cdot f \land \Box \Box P \land \Box \Box \langle A \rangle \cdot f \rightarrow \Box \Box \langle N \land B \rangle \cdot f \)

by (rule lift-imp-trans)

moreover

have \( \vdash \Box \Box \langle N \land B \rangle \cdot f \rightarrow \Box \Box \langle M \rangle \cdot g \) 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: WeakF-def)

qed

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: \( \neg \langle N \land B \rangle \cdot f \rightarrow \langle M \rangle \cdot g \)
and \( h2: \sim P \land oP \land (N \land A) \rightarrow B \)

and \( h3: P \land \text{Enabled} (M) \rightarrow \text{Enabled} (A) \)

and \( h4: \\
\sim (N \land B) \rightarrow SF(A) \land \square F \land \square \text{Enabled} (M) \rightarrow \square \square P \)

shows \( \vdash \square [N] \rightarrow SF(A) \land \square F \rightarrow SF(M) \rightarrow \square P \)

proof –

have \( \vdash \square [N] \rightarrow SF(A) \land \square F \land \square \text{Enabled} (M) \rightarrow \square \square P \)

proof –

have \( 1: \vdash [N] \rightarrow SF(A) \land \square F \land \square \text{Enabled} (M) \rightarrow \square \square P \)

proof –

have \( A: \vdash [N] \rightarrow SF(A) \land \square F \land \square \text{Enabled} (M) \rightarrow \square \square P \)

by (rule SE2)

from lift-imp-trans[OF A B]

have \( \vdash (N) \rightarrow SF(A) \land \square F \land \square \text{Enabled} (M) \rightarrow \square \square P \)

by (rule M2)

by (force intro: T4[unlift-rule])

with \( h4 \) have \( \vdash (\square [N] \land SF(A) \land \square F) \land (\square \text{Enabled} (M) \land \square \square P) \)

by simp

ultimately

show \( \square \square P \) by (rule lift-imp-trans)

qed

have \( 2: \vdash [N] \land SF(A) \land \square \text{Enabled} (M) \land \square \square P \rightarrow \square \square P \)

proof –

have \( \vdash (P \land \text{Enabled} (M) \land SF(A) \land \square \text{Enabled} (A) \land \square P \land oP \land (N \land A)) \rightarrow \square \square P \)

by (force intro: AA29[unlift-rule])

by (auto simp: actrans-def)

by (force intro: actrans-def)

by (rule M2)

from lift-imp-trans[OF this ax4] have \( \vdash \square [N] \rightarrow \square \square P \)

by (force intro: T4[unlift-rule])

with \( h4 \) have \( \vdash (\square [N] \rightarrow SF(A) \land \square F) \land (\square \text{Enabled} (M) \land \square \square P) \)

by simp

ultimately

show ?thesis by (rule lift-imp-trans)

qed
This is the lattice rule from TLA

**Theorem**: `wf-leadsto`

**Assumes**: $h1$: $\forall x. E x \Rightarrow (G \lor E x)$

**Shows**: $\forall x. E x \Rightarrow G$

**Using**: $h1$

**Proof** (rule `wf-induct`)

1. **Fix** $x$
2. **Assume** $ih$: $\forall y. (y, x) \in r \Rightarrow (\forall y. E y \Rightarrow G)$
3. **Show**: $\forall y. E y \Rightarrow G$

**Proof**

- **From** $ih$ have $\forall y. ((y, x) \in r) \land E y \Rightarrow G$
- **By** (force `simp` `LT21` `LT33`)  
- **With** $h2$ show $\forall y. E y \Rightarrow G$

**QED**

### 6.3 Stuttering Invariance

**Theorem**: `stut-Enabled`

**By** (auto `simp`: `StrongF-def`)

**Theorem**: `stut-WF`

**By** (auto `simp`: `WeakF-def stut-invs bothstutinvs`)

**Theorem**: `stut-SF`

**By** (auto `simp`: `ennf-Enabled bothstutinvs`)

---

75
by (auto simp: StrongF-def stut-Enabled bothstatinv)

lemmas livestatinv = 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 un-
defined type, which is an instance of the world class to enable use with the
Intensional theory.

typedecl state

instance state :: world ..

type-synonym 'a statefun = (state,'a) stfun
type-synonym statepred = bool statefun
type-synonym 'a tempfun = (state,'a) formfun
type-synonym temporal = state formula

Formalizing type state would require formulas to be tagged with their un-
derlying 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 ac-
tions. 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"               (PRED -)
  "-stvars :: lift ⇒ bool"          (basevars -)

76
translations

\[ PRED\ P \rightarrow (P ::state \Rightarrow \cdot) \]

\[-stvars \Leftrightarrow CONST stvars \]

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: \( basevars vs \Rightarrow \exists u. vs u = c \)

**proof** (unfold basevars-def surj-def)

\[ assume \ \forall y. \exists x. y = vs x \]

\[ then\ obtain\ x\ where\ c = vs x\ by\ blast \]

\[ thus\ \exists u. vs u = c\ by\ blast \]

**qed**

**lemma** first-baseE:

\[ assumes H1: basevars v\ and\ H2: \bigwedge x. v x = c \Rightarrow Q \]

\[ shows Q \]

**using** \( H1[TEN basevars] \ H2\ by\ auto \)

A variant written for sequences rather than single states.

**lemma** first-baseE:

\[ assumes H1: basevars v\ and\ H2: (x y) = \bigwedge c (first x) = c \Rightarrow Q \]

\[ shows Q \]

**using** \( H1[TEN basevars] \ H2\ by\ (force simp: first-def) \)

**lemma** base-pair1:

\[ assumes h: basevars (x y) \]

\[ shows basevars x \]

**proof** (auto simp: basevars-def)

\[ fix\ c \]

\[ from\ h[TEN basevars]\ obtain\ s\ where\ (LIFT (x y)) s = (c, arbitrary)\ by\ auto \]

\[ thus\ c \in range x\ by\ auto \]

**qed**

**lemma** base-pair2:

\[ assumes h: basevars (x y) \]

\[ shows basevars y \]

**proof** (auto simp: basevars-def)

\[ fix\ d \]

\[ from\ h[TEN basevars]\ obtain\ s\ where\ (LIFT (x y)) s = (arbitrary, d)\ by\ auto \]

\[ thus\ d \in range y\ by\ auto \]

**qed**

**lemma** base-pair:

\[ basevars (x y) \Rightarrow basevars x \land basevars y \]
Since the unit type has just one value, any state function of unit type satisfies the predicate basevars. The following theorem can sometimes be useful because it gives a trivial solution for basevars premises.

**lemma** unit-base: basevars (v::state ⇒ unit)
  by (auto simp: basevars-def)

A pair of the form (x,x) will generally not satisfy the predicate basevars – except for pathological cases such as x::unit.

**lemma**
  fixes x :: state ⇒ bool
  assumes h1: basevars (x,x)
  shows False
  proof –
  from h1 have ∃ u. (LIFT (x,x)) u = (False, True) by (rule basevars)
  thus False by auto
  qed

**lemma**
  fixes x :: state ⇒ nat
  assumes h1: basevars (x,x)
  shows False
  proof –
  from h1 have ∃ 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: ∀ u, vs (first u) = c ⇒ ((first s) ## u) |= F
  shows s |= Enabled F
  using h1 proof (rule first-baseE)
  fix t
  assume vs (first t) = c
  hence ((first s) ## t) |= F by (rule h2)
  thus s |= Enabled 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 *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 ⇒ temporal) ⇒ temporal` (binder `Eex 10`)
- `AAll :: (a statefun ⇒ temporal) ⇒ temporal` (binder `Aall 10`)

**translations**

- `EEx v A == Eex v . A`
- `AAll v A == Aall v . A`

**axiomatization where**

- `eexI: ⊢ F x → (∃ x. F x)`
- `eexE: [s | (∃ x. F x) ; basevars vs; (! x. [ basevars (x,vs); s | F x ]] → s | G)]` → `(s | G)`
- `all-def: ⊢ (∀ x. F x) = (¬(∃ x. ¬(F x)))`
- `eeexSTUT: STUTINV F x ⇒ STUTINV (∃ x. F x)`
- `history: ⊢ (I ∧ □[A]-v) = (∃ h. ($h = ha) ∧ I ∧ □[A ∧ h$=hb]-h,v)`

**lemmas**

- `eexI-unl = eexI[unlift-rule] — w | F x ⇒ w | (∃ x. F x)`

**tladefs** can be used to unfold TLA definitions into lowest predicate level. This is particularly useful for reasoning about enabledness of formulas.

- `theory Even`
- `imports State`
- `begin`

---

8 A simple illustrative example
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.

\[
\text{inductive-set}
\]

\[
\text{Even :: nat set}
\]

\[
\text{where}
\]

\[
\text{even-zero: } 0 \in \text{Even}
\]

\[
\text{even-step: } n \in \text{Even} \implies \text{Suc(Suc } n) \in \text{Even}
\]

locale Program =

fixes \( x :: \text{state} \Rightarrow \text{nat} \)

and \( \text{init} :: \text{temporal} \)

and \( \text{act} :: \text{temporal} \)

and \( \text{phi} :: \text{temporal} \)

defines \( \text{init} \equiv \text{TEMP } x = \# \ 0 \)

and \( \text{act} \equiv \text{TEMP } x' = \text{Suc}<\text{Suc}\<$x$)>\)

and \( \text{phi} \equiv \text{TEMP } \text{init} \land \Box [\text{act}] - x \)

lemma (in Program) stutinvprog: \text{STUTINV } \text{phi}

by (auto simp: phi-def init-def act-def stutinv nsutinv)

lemma (in Program) inweven: \( \vdash \text{phi} \implies \Box (x \in \# \text{Even}) \)

unfolding phi-def

proof (rule invmono)

show \( \vdash \text{init} \implies x \in \# \text{Even} \)

by (auto simp: init-def even-zero)

next

show \( \vdash \lnot x \in \# \text{Even} \land [\text{act}] - x \implies o(x \in \# \text{Even}) \)

by (auto simp: 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 \( \text{pcount} = a \mid b \mid g \)

locale Firstprogram =

fixes \( x :: \text{state} \Rightarrow \text{nat} \)

and \( y :: \text{state} \Rightarrow \text{nat} \)

and \( \text{init} :: \text{temporal} \)
and \( m_1 :: \text{temporal} \)
and \( m_2 :: \text{temporal} \)
and \( \phi :: \text{temporal} \)
and \( \text{Live} :: \text{temporal} \)
defines init \( \equiv \text{TEMP } x = \# 0 \land y = \# 0 \)
and \( m_1 \equiv \text{TEMP } x' = \text{Suc } x > y' = y \)
and \( m_2 \equiv \text{TEMP } y' = \text{Suc } y > x' = x \)
and \( \text{Live} \equiv \text{TEMP } \text{WF}(m_1)-(x,y) \land \text{WF}(m_2)-(x,y) \)
and \( \phi \equiv \text{TEMP } (\text{init} \land \Box[m_1 \lor m_2]-(x,y) \land \text{Live}) \)
assumes bvar :: basevars (x,y)

lemma (in Firstprogram) \( \text{STUTINV } \phi \)
by (auto simp: phi-def init-def m1-def m2-def Live-def stutinvstutinv)

lemma (in Firstprogram) enabled-m1: \( \vdash \text{Enabled } (m_1)-(x,y) \)
proof (clarify)
fix s
show \( s \models \text{Enabled } (m_1)-(x,y) \)
by (rule base-enabled[OF bvar]) (auto simp: m1-def tla-defs)
qed

lemma (in Firstprogram) enabled-m2: \( \vdash \text{Enabled } (m_2)-(x,y) \)
proof (clarify)
fix s
show \( s \models \text{Enabled } (m_2)-(x,y) \)
by (rule base-enabled[OF bvar]) (auto simp: m2-def tla-defs)
qed

locale Secondprogram = Firstprogram +
fixes sem :: state \( \Rightarrow \) nat
and pc1 :: state \( \Rightarrow \) pcount
and pc2 :: state \( \Rightarrow \) pcount
and vars
and initPsi :: temporal
and alpha1 :: temporal
and alpha2 :: temporal
and beta1 :: temporal
and beta2 :: temporal
and gamma1 :: temporal
and gamma2 :: temporal
and n1 :: temporal
and n2 :: temporal
and Live2 :: temporal
and psi :: temporal
and I :: temporal
defines vars \( \equiv \text{LIFT } (x,y,sem,pc1,pc2) \)
and initPsi \( \equiv \text{TEMP } \text{pc1} = \# a \land \text{pc2} = \# a \land x = \# 0 \land y = \# 0 \land \text{sem} = \# 1 \)
\[ \text{and } \alpha_1 \equiv \text{TEMP } \{ p_1 = \#a \land \# 0 < \text{sem} \land p_1S = \#b \land \text{sem}S = \text{sem} \land 1 \land \text{Unchanged } (x,y,pc) \]  
\[ \text{and } \alpha_2 \equiv \text{TEMP } p_{c2} = \#a \land \# 0 < \text{sem} \land p_{c2} = \#b \land \text{sem}S = \text{sem} \land 1 \land \text{Unchanged } (x,y,pc) \]  
\[ \text{and } \beta_1 \equiv \text{TEMP } p_{c1} = \#b \land \text{pc} = \#a \land \text{pc} = \#g \land \text{pc} = \text{Suc}<x> \land \text{Unchanged } (y,\text{sem},pc_{c1}) \]  
\[ \text{and } \beta_2 \equiv \text{TEMP } p_{c2} = \#b \land p_{c2} = \#g \land y' = \text{Suc}<y> \land \text{Unchanged } (x,\text{sem},pc_{c2}) \]  
\[ \text{and } \gamma_1 \equiv \text{TEMP } p_{c1} = \#a \land \text{pc} = \#g \land \text{pc} = \text{Suc}<\text{sem}> \land \text{Unchanged } (x,y,pc) \]  
\[ \text{and } \gamma_2 \equiv \text{TEMP } p_{c2} = \#a \land p_{c2} = \#g \land \text{pc} = \text{Suc}<\text{sem}> \land \text{Unchanged } (x,y,pc) \]  
\[ \text{and } \delta_1 \equiv \text{TEMP } (\alpha_1 \lor \beta_1 \lor \gamma_1) \]  
\[ \text{and } \delta_2 \equiv \text{TEMP } (\alpha_2 \lor \beta_2 \lor \gamma_2) \]  
\[ \text{and } \text{Live}_2 \equiv \text{TEMP } SF(n_1)-\text{vars} \land SF(n_2)-\text{vars} \]  
\[ \text{and } \psi \equiv \text{TEMP } (\text{initPsi} \land \text{พ}[n_1 \lor n_2]-\text{vars} \land \text{Live}_2) \]  
\[ \text{and } I \equiv \text{TEMP } (\text{sem} = \# 1 \land \text{pc} = \#a \land \text{pc} = \#g) \] 
\[ \land (\text{sem} = \# 0 \land ((\text{pc} = \#a \land \text{pc} = \#g)) \] 
\[ \lor (\text{pc} = \#a \land \text{pc} \in \{\#b, \#g\}) \] 
\[ \lor (\text{pc} = \#a \land \text{pc} \in \{\#b, \#g\})) \] 
\[ \text{assumes } \text{vvar}_2 : \text{basevars} \text{ vars} \]

**lemmas** \((\text{in Secondprogram}) \) \(\text{Sact}_2-\text{defs} = n_1-\text{def} n_2-\text{def} \alpha_1-\text{def} \beta_1-\text{def} \gamma_1-\text{def} \delta_1-\text{def} \gamma_2-\text{def} \delta_2-\text{def} \)

Proving invariants is the basis of every effort of system verification. We show that \(I\) is an inductive invariant of specification \(\psi\).

**lemma** \((\text{in Secondprogram}) \) \(\text{psiI} : \vdash \psi \rightarrow \square I\)

**proof**
- have \(\vdash \text{initPsi} \rightarrow I\) by \((\text{auto simp: initPsi-def I-def})\)
- have \(\vdash \neg I \land \text{Unchanged vars} \rightarrow \neg I\) by \((\text{auto simp: I-def vars-def tla-defs})\)
- moreover
  - have \(\vdash \neg I \land \text{n1} \rightarrow \neg I\) by \((\text{auto simp: I-def Sact2-defs tla-defs})\)
- moreover
  - have \(\vdash \neg I \land \text{n2} \rightarrow \neg I\) by \((\text{auto simp: I-def Sact2-defs tla-defs})\)
- **ultimately have** step: \(\vdash \neg I \land \text{n1} \lor \text{n2}-\text{vars} \rightarrow \square I\) by \((\text{force simp: actrans-def})\)
- **from** init step have goal: \(\vdash \text{initPsi} \land \square[n_1 \lor n_2]-\text{vars} \rightarrow \square I\) by \((\text{rule inmono})\)
- have \(\vdash \text{initPsi} \land \square[n_1 \lor n_2]-\text{vars} \land \text{Live}_2 \rightarrow \vdash \text{initPsi} \land \square[n_1 \lor n_2]-\text{vars} \land \text{Live}_2 \land I\) by \((\text{auto})\)
- 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** \((\text{in Secondprogram}) \) \(\text{step-simulation} : \vdash \psi \rightarrow \text{init} \land \square[m_1 \lor m_2]-\{x,y\}\)

**proof**
- have \(\vdash \text{initPsi} \land \square \land \square[n_1 \lor n_2]-\text{vars} \rightarrow \text{init} \land \square[m_1 \lor m_2]-\{x,y\}\)
- **proof** \((\text{rule refinement1})\)
  - show \(\vdash \text{initPsi} \rightarrow \text{init}\) by \((\text{auto simp: initPsi-def init-def})\)
  - next

82
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: $\mathord{\langle \lnot n1 \rangle}$-vars = $n1$

**Proof**
- Have $\mathord{\lnot n1} \rightarrow (n1)$-vars
  - By (auto simp: $Sact2$-defs $tlas$-defs $vars$-def)

**QED**

**Lemma (in Secondprogram)** $enab$-alpha1: $\vdash \#pc1 = \#a \rightarrow \#0 < \#sem \rightarrow Enabled\ alpha1$

**Proof**
- Fix $s :: state\ seq$
- Assume $pc1\ (s\ 0) = a$ and $0 < sem\ (s\ 0)$
- Thus $s \models Enabled\ alpha1$
  - By (intro base-enabled[OF bvar2]) (auto simp: $Sact2$-defs $tlas$-defs $vars$-def)

**QED**

**Lemma (in Secondprogram)** $enab$-beta1: $\vdash \#pc1 = \#b \rightarrow Enabled\ beta1$

**Proof**
- Fix $s :: state\ seq$
- Assume $pc1\ (s\ 0) = b$
- Thus $s \models Enabled\ beta1$
  - By (intro base-enabled[OF bvar2]) (auto simp: $Sact2$-defs $tlas$-defs $vars$-def)

**QED**

**Lemma (in Secondprogram)** $enab$-gamma1: $\vdash \#pc1 = \#\ g \rightarrow Enabled\ gamma1$

**Proof**
- Fix $s :: state\ seq$
- Assume $pc1\ (s\ 0) = g$
- Thus $s \models Enabled\ gamma1$
  - By (intro base-enabled[OF bvar2]) (auto simp: $Sact2$-defs $tlas$-defs $vars$-def)

**QED**

**Lemma (in Secondprogram)** $enab$-$n1$:
- $\vdash Enabled\ \langle n1 \rangle$-vars = $(\#pc1 = \#a \rightarrow \#0 < \#sem)$
  - Unfolding $n1$-ch[revert rewrite] *Proof* (rule int-iff)
    - Show $\vdash Enabled\ n1 \rightarrow \#pc1 = \#a \rightarrow \#0 < \#sem$
      - By (auto elim!: enabledE simp: $Sact2$-defs $tlas$-defs)
  - Next
    - Show $\vdash (\#pc1 = \#a \rightarrow \#0 < \#sem) \rightarrow Enabled\ n1$
      - *Proof* (clar simp: $tlas$-defs)
        - Fix $s :: state\ seq$
assume \( pc_1 \ (s \ 0) = a \rightarrow 0 < \text{sem} \ (s \ 0) \)

thus \( s \models \text{Enabled} \ n_1 \)

using \( \text{enab-alpha1[unlift-rule]} \)

\( \text{enab-beta1[unlift-rule]} \)

\( \text{enab-gamma1[unlift-rule]} \)

by (cases \( pc_1 \ (s \ 0) \)) \( \text{force simp: n1-def Enabled-disj[int-rewrite] tla-defs} \)

qed

qed

The analogous properties for the second process are obtained by copy and paste.

lemma (in Secondprogram) \( n_2\text{-ch: } \lnot \ \langle n_2 \rangle\text{-vars} = n_2 \)

proof

have \( \lnot \ n_2 \rightarrow \langle n_2 \rangle\text{-vars} \)

by (auto simp: Sact2-defs tla-defs vars-def)

thus ?thesis by (auto simp: angle-actrans-sem)

qed

lemma (in Secondprogram) \( \text{enab-alpha2: } \vdash \# pc_2 \rightarrow \# 0 < \$ \text{sem} \rightarrow \text{Enabled alpha2} \)

proof (clarsimp simp: tla-defs)

fix \( s :: \text{state seq} \)

assume \( pc_2 \ (s \ 0) = a \ \text{and} \ 0 < \text{sem} \ (s \ 0) \)

thus \( s \models \text{Enabled alpha2} \)

by (intro base-enabled[OF bvar2]) (auto simp: Sact2-defs tla-defs vars-def)

qed

lemma (in Secondprogram) \( \text{enab-beta2: } \vdash \# pc_2 \rightarrow \text{Enabled beta2} \)

proof (clarsimp simp: tla-defs)

fix \( s :: \text{state seq} \)

assume \( pc_2 \ (s \ 0) = b \)

thus \( s \models \text{Enabled beta2} \)

by (intro base-enabled[OF bvar2]) (auto simp: Sact2-defs tla-defs vars-def)

qed

lemma (in Secondprogram) \( \text{enab-gamma2: } \vdash \# pc_2 \rightarrow \text{Enabled gamma2} \)

proof (clarsimp simp: tla-defs)

fix \( s :: \text{state seq} \)

assume \( pc_2 \ (s \ 0) = g \)

thus \( s \models \text{Enabled gamma2} \)

by (intro base-enabled[OF bvar2]) (auto simp: Sact2-defs tla-defs vars-def)

qed

lemma (in Secondprogram) \( \text{enab-n2: } \vdash \text{Enabled} \ \langle n_2 \rangle\text{-vars} = (\$ pc_2 = \# a \rightarrow \# 0 < \$ \text{sem}) \)

unfolding \( n_2\text{-ch[int-rewrite]} \)

proof (rule int-iffI)

show \( \vdash \text{Enabled} \ n_2 \rightarrow \$ pc_2 = \# a \rightarrow \# 0 < \$ \text{sem} \)

by (auto elim!: enabledE simp: Sact2-defs tla-defs)

next

show \( \vdash (\$ pc_2 = \# a \rightarrow \# 0 < \$ \text{sem}) \rightarrow \text{Enabled} \ n_2 \)

84
proof (clarsimp simp: tla-defs)
fix s :: state seq
assume pc2 (s 0) = a →→ 0 < sem (s 0)
thus s \models 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
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.

lemma (in Secondprogram) psi-fair-m1: \(\vdash\) \(\psi \rightarrow SF(m1)-(x,y)\)

proof
have \(\vdash \square[n1 \lor n2]-vars \land SF(n1)-vars \land \square(I \land SF(n2)-vars) \rightarrow SF(m1)-(x,y)\)
proof (rule SF2)
Rule SF2 requires us to choose a helpful action (whose effect implies \(\langle m1 \rangle-(x,y))\) and a persistent condition, which will eventually remain true if the helpful action is never executed. In our case, the helpful action is beta1 and the persistent condition is \(pc1 = b\).

show \(\sim (\langle n1 \lor n2 \rangle \land \beta1 )\)-vars \(\rightarrow \langle m1 \rangle-(x,y)\)
by (auto simp: beta1-def m1-def vars-def tla-defs)
next
show \(\sim \#pc1 = \#b \land \circ(\#pc1 = \#b) \land (\langle n1 \lor n2 \rangle \land \n1 ) - vars \rightarrow \beta1\)
by (auto simp: n1-def alpha1-def beta1-def gamma1-def tla-defs)
next
show \(\vdash \#pc1 = \#b \land \text{Enabled }\langle m1 \rangle-(x,y) \rightarrow \text{Enabled }\langle n1 \rangle - vars\)
unfolding enab-n1[int-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 \(\vdash \square\square(n1 \lor n2) \land \neg beta1\)-vars
\land SF(n1)-vars \land \square(I \land SF(n2)-vars) \land \square\Diamond\text{Enabled }\langle m1 \rangle-(x,y)
\rightarrow \square\square(\#pc1 = \#b)\)
proof
have \(\square\square[n1 \lor n2) \land \neg beta1\]-vars \(\rightarrow \square(\#pc1 = \#b \rightarrow \square(\#pc1 = \#b))\)
proof (rule STL4)
have \(\sim \#pc1 = \#b \land (\langle n1 \lor n2 \rangle \land \neg beta1\)-vars \(\rightarrow \circ(\#pc1 = \#b)\)
by (auto simp: Sact2-defs vars-def tla-defs)
from this[THEN INV1]
show \(\vdash \square(n1 \lor n2) \land \neg beta1\)-vars \(\rightarrow \#pc1 = \#b \rightarrow \square(\#pc1 = \#b)\)
by auto
qed
hence $1 \vdash \Box[(n1 \lor n2) \land \neg beta1]-\text{vars} \longrightarrow \Diamond(\neg pc1 = \#b) \longrightarrow \Diamond(\neg pc1 = \#b)$

by (force intro: E31[unlift-rule])

have $1 \vdash \Box[(n1 \lor n2) \land \neg beta1]-\text{vars} \land SF(n1)-\text{vars} \land \Box(I \land SF(n2)-\text{vars})

\longrightarrow \Diamond(\neg pc1 = \#b)$

proof –

The plan of the proof is to show that from any state where $pc1 = g$ one eventually reaches $pc1 = a$, from where one eventually reaches $pc1 = b$. The result follows by combining leads to properties.

let $?F = \text{LIFT} \ (\Box[(n1 \lor n2) \land \neg beta1]-\text{vars} \land SF(n1)-\text{vars} \land \Box(I \land SF(n2)-\text{vars}))$

Showing that $pc1 = g$ leads to $pc1 = a$ is a simple application of rule SF1 because the first process completely controls this transition.

have $ga \vdash \neg F \longrightarrow (\neg pc1 = \#g \iff \neg pc1 = \#a)$

proof (rule SF1)

show $\neg \neg pc1 = \#g \land [(n1 \lor n2) \land \neg beta1]-\text{vars} \longrightarrow \circ(\neg pc1 = \#g) \lor
\circ(\neg pc1 = \#a)$

by (auto simp: Sact2-defs vars-def tla-defs)

next

show $\neg \neg pc1 = \#g \land \langle((n1 \lor n2) \land \neg beta1) \land n1\rangle-\text{vars} \longrightarrow \circ(\neg pc1 = \#a)$

by (auto simp: Sact2-defs vars-def tla-defs)

next

have $\neg \neg pc1 = \#g \longrightarrow \text{Enabled } \langle n1\rangle-\text{vars}$

unfolding enab-n1[int-rewrite] by (auto simp: tla-defs)

hence $\neg \neg (\neg pc1 = \#g) \longrightarrow \text{Enabled } \langle n1\rangle-\text{vars}$

by (rule lift-imp-trans[OF az1])

hence $\neg \neg (\neg pc1 = \#g) \longrightarrow \Diamond \text{Enabled } \langle n1\rangle-\text{vars}$

by (rule lift-imp-trans[OF E31])

thus $\neg \neg (\neg pc1 = \#g) \land \Box[(n1 \lor n2) \land \neg beta1]-\text{vars} \land \Box(I \land SF(n2)-\text{vars})

\longrightarrow \Diamond \text{Enabled } \langle n1\rangle-\text{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 $ab \vdash \neg F \longrightarrow (\neg pc1 = \#a \iff \neg pc1 = \#b)$

proof (rule SF1)

show $\neg \neg pc1 = \#a \land [(n1 \lor n2) \land \neg beta1]-\text{vars} \longrightarrow \circ(\neg pc1 = \#a) \lor
\circ(\neg pc1 = \#b)$

by (auto simp: Sact2-defs vars-def tla-defs)

next

show $\neg \neg pc1 = \#a \land \langle((n1 \lor n2) \land \neg beta1) \land n1\rangle-\text{vars} \longrightarrow \circ(\neg pc1 = \#b)$

86
by (auto simp: Sact2-defs vars-def tla-defs)
next
show \(\neg \#pc1 = \#a \land \text{Unchanged vars} \rightarrow o(\#pc1 = \#a)\)
by (auto simp: vars-def tla-defs)

We establish a suitable leadsto-chain.

let \(?G = \text{LIFT} \Box[(n1 \lor n2) \land \neg beta1]\text{-vars} \land SF(n2)\text{-vars} \land \Box(\#pc1 = \#a \land I)\)

have \(\vdash ?G \rightarrow o(\#pc2 = \#a \land \#pc1 = \#a \land I)\)
proof

Rule SF1 takes us from \(pc2 = b\) to \(pc2 = g\).

have bg2: \(\vdash ?G \rightarrow (\#pc2 = \#b \leadsto \#pc2 = \#g)\)
proof (rule SF1)

show \(\neg \#pc2 = \#b \land [(n1 \lor n2) \land \neg beta1]\text{-vars} \rightarrow o(\#pc2 = \#b)\)
by (auto simp: Sact2-defs vars-def tla-defs)
next

show \(\neg \#pc2 = \#b \land \text{Unchanged vars} \rightarrow o(\#pc2 = \#b)\)
by (auto simp: vars-def tla-defs)

have \(\vdash \#pc2 = \#b \leadsto \Diamond\text{Enabled } (n2)\text{-vars}\)
next

unfolding enab-n2[int-rewrite] by (auto simp: tla-defs)

hence \(\vdash \Box(\#pc2 = \#b) \leadsto \Diamond\text{Enabled } (n2)\text{-vars}\)
by (rule lift-imp-trans[OF ax1])

hence \(\vdash \Box(\#pc2 = \#b) \leadsto \Diamond\text{Enabled } (n2)\text{-vars}\)
by (rule lift-imp-trans[OF - E3])

thus \(\vdash \Box(\#pc2 = \#b) \land \Box[(n1 \lor n2) \land \neg beta1]\text{-vars} \land \Box(\#pc1 = \#a \land I)\)

\rightarrow \Diamond\text{Enabled } (n2)\text{-vars}\)
by auto
qed

Similarly, \(pc2 = b\) leads to \(pc2 = g\).

have ga2: \(\vdash ?G \rightarrow (\#pc2 = \#g \leadsto \#pc2 = \#a)\)
proof (rule SF1)

show \(\neg \#pc2 = \#g \land [(n1 \lor n2) \land \neg beta1]\text{-vars} \rightarrow o(\#pc2 = \#g)\)
by (auto simp: Sact2-defs vars-def tla-defs)
next

show \(\neg \#pc2 = \#g \land \langle(n1 \lor n2) \land \neg beta1 \rangle \land n2\text{-vars} \rightarrow o(\#pc2 = \#a)\)
by (auto simp: a2-def alpha2-def beta2-def gamma2-def vars-def tla-defs)
next
  show \( \neg \) $pc2 = \# g \land \text{Unchanged vars} \rightarrow \circ (\neg \text{vars-def tla-defs})
  by (auto simp: vars-def tla-defs)
next
  have \( \vdash \) $pc2 = \# g \rightarrow \text{Enabled } \langle n2 \rangle \text{-vars}
    unfolding enab-n2[int-rewrite] by (auto simp: tla-defs)
  hence \( \Box \)($pc2 = \# g) \rightarrow \Diamond \text{Enabled } \langle n2 \rangle \text{-vars}
    by (rule lift-imp-trans[OF ax1])
  thus \( \vdash \) $pc2 = \# g \rightarrow \Diamond \text{Enabled } \langle n2 \rangle \text{-vars}
    by (rule lift-imp-trans[OF E3])
thus \( \vdash \) $pc2 = \# g \land \Box \[(n1 \lor n2) \land \neg beta1]\text{-vars} \land \Box (\neg \text{vars-def tla-defs})
  by (force elim: LT13[unlift-rule])
moreover
  have \( \vdash \Diamond (\neg \text{vars-def tla-defs}) \land \Box \[(n1 \lor n2) \land \neg beta1]\text{-vars} \land \Box (\neg \text{vars-def tla-defs})
    by (force simp: STL5[int-rewrite])
ultimately
  show \( \vdash \Diamond \text{Enabled } \langle n1 \rangle \text{-vars}
    by (force simp: STL5[int-rewrite])
qed

next
  have \( \vdash \) $pc2 = \# g \rightarrow \text{Enabled } \langle n2 \rangle \text{-vars}
    unfolding enab-n2[int-rewrite] by (auto simp: tla-defs)
  hence \( \Box \)($pc2 = \# g) \rightarrow \Diamond \text{Enabled } \langle n2 \rangle \text{-vars}
    by (rule lift-imp-trans[OF ax1])
  thus \( \vdash \) $pc2 = \# g \rightarrow \Diamond \text{Enabled } \langle n2 \rangle \text{-vars}
    by (rule lift-imp-trans[OF E3])
thus \( \vdash \) $pc2 = \# g \land \Box \[(n1 \lor n2) \land \neg beta1]\text{-vars} \land \Box (\neg \text{vars-def tla-defs})
  by (force elim: LT13[unlift-rule])
moreover
  have \( \vdash \Diamond (\neg \text{vars-def tla-defs}) \land \Box \[(n1 \lor n2) \land \neg beta1]\text{-vars} \land \Box (\neg \text{vars-def tla-defs})
    by (force simp: STL5[int-rewrite])
ultimately
  show \( \vdash \Diamond \text{Enabled } \langle n1 \rangle \text{-vars}
    by (force simp: STL5[int-rewrite])
qed

next
  show \( \neg \) $pc2 = \# g \land \text{Unchanged vars} \rightarrow \circ (\neg \text{vars-def tla-defs})
  by (auto simp: vars-def tla-defs)
moreover

have \( \vdash \$pc1 = \#a \lor \$pc1 = \#b \lor \$pc1 = \#g \)

proof (clarsimp simp: tla-defs)

fix \( s :: \mathtt{state} \ \mathtt{seq} \)

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 \( \vdash ((\$pc1 = \#a \lor \$pc1 = \#b \lor \$pc1 = \#g) \rightsquigarrow \$pc1 = \#b) \)

\( \leadsto \)

by (rule \( \text{fmp} \)[OF - LT4])

ultimately show \( \text{thesis} \) by (rule lift-imp-trans)

qed

with 1 show \( \text{thesis} \) by force

qed

qed

with \( \text{psil} \) show \( \text{thesis} \) unfolding \( \psi\text{-def} \ \text{Live2-def} \ \text{STL5}[\text{int-rewrite}] \) by force

qed

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) \( \psi\text{-fair-m2}: \vdash \psi \rightarrow SF(m2)-(x,y) \)

proof -

have \( \vdash (\Box[n1 \lor n2]\text{-vars} \land SF(n2)\text{-vars} \land \Box(I \land SF(n1)\text{-vars}) \rightarrow SF(m2)-(x,y)) \)

proof (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 \( \text{beta2} \) and the persistent condition is \( pc2 = b \).

show \( \neg ((n1 \lor n2) \land \text{beta2}\text{-vars}) \rightarrow (m2)-(x,y) \)

by (auto simp: beta2-def m2-def vars-def tla-defs)

next

show \( \neg \$pc2 = \#b \land \circ(\$pc2 = \#b) \land ((n1 \lor n2) \land n2)\text{-vars} \rightarrow \text{beta2} \)

by (auto simp: n2-def alpha2-def beta2-def gamma2-def tla-defs)

next

show \( \vdash \$pc2 = \#b \land \text{Enabled} (m2)-(x, y) \rightarrow \text{Enabled} (n2)\text{-vars} \)

unfolding \( \text{enab-n2}[\text{int-rewrite}] \) 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 \( \vdash \Box[(n1 \lor n2) \land \neg \text{beta2}\text{-vars} \land SF(n2)\text{-vars} \land \Box(I \land SF(n1)\text{-vars}) \land \Box\neg \text{Enabled} (m2)-(x, y) \rightarrow \Box\neg (\$pc2 = \#b) \)

proof -

have \( \vdash \Box[(n1 \lor n2) \land \neg \text{beta2}\text{-vars} \rightarrow \Box(\$pc2 = \#b \rightarrow \Box(\$pc2 = \#b)) \)

proof (rule STL4)
showing that combining leads to properties.

The plan of the proof is to show that from any state where the second process completely controls this transition, which must liberate the critical section.

The proof that $pc2 = g$ one eventually reaches $pc2 = a$, from where one eventually reaches $pc2 = b$. The result follows by combining leadsto properties.

let $F = LIFT \ ((n1 \lor n2) \land \lnot beta2) \land SF(n2) \land \Box(I \land SF(n1))$

showing that $pc2 = g$ leads to $pc2 = a$ is a simple application of rule $SF1$ because the second process completely controls this transition.

have $g a: \vdash F \rightarrow (pc2 = g \rightarrow pc2 = a)$

proof (rule $SF1$)

| show $\vdash (pc2 = g \land (n1 \lor n2) \land \lnot beta2) \rightarrow \Box\Box(pc2 = g) \lor \Box(pc2 = a)$ |

by (auto simp: Sact2-defs vars-def tla-defs)

next

| show $\vdash \Box(pc2 = g \land \langle(n1 \lor n2) \land \lnot beta2\rangle \land \lnot n2) \rightarrow \Box\Box(pc2 = g)$ |

by (auto simp: n2-def alpha2-def beta2-def gamma2-def vars-def tla-defs)

next

| show $\vdash (pc2 = g \land Unchanged \rightarrow \Box\Box(pc2 = g)$ |

by (auto simp: vars-def tla-defs)

next

| have $\vdash \Box(pc2 = g) \rightarrow Enabled\langle n2\rangle$ |

| unfolding enab-n2[int-rewrite] by (auto simp: tla-defs)

hence $\vdash \Box\Box(pc2 = g) \rightarrow Enabled\langle n2\rangle$ |

by (rule lift-imp-trans[OF az1])

hence $\vdash \Box\Box(pc2 = g) \rightarrow \Box\Box pc2$ |

by (rule lift-imp-trans[OF - E3])

thus $\vdash \Box(pc2 = g) \land \Box\Box\Box((n1 \lor n2) \land \lnot beta2) \land \Box(I \land SF(n1)\land SF(n1)\land SF(n1))$ |

| $\rightarrow \Box\Box pc2$ |

by auto

qed

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.

have $ab: \vdash F \rightarrow (pc2 = a \rightarrow pc2 = b)$
proof (rule SF1)
   show \( \neg \ p_{c2} = \#a \land [(n1 \lor n2) \land \neg beta2]\)-vars \( \rightarrow \ o(p_{c2} = \#a) \lor o(p_{c2} = \#b) \)
   by (auto simp: Sact2-defs vars-def tla-defs)
next
   show \( \neg \ p_{c2} = \#a \land ((n1 \lor n2) \land \neg beta2) \land n2\)-vars \( \rightarrow \ o(p_{c2} = \#b) \)
   by (auto simp: vars-def tla-defs)
next
   have \( \neg \ p_{c2} = \#a \land Unchanged\) vars \( \rightarrow o(p_{c2} = \#a) \)
   by (auto simp: vars-def tla-defs)
next

We establish a suitable leadsto-chain.

let \(?G = LIFT \Box [(n1 \lor n2) \land \neg beta2]\)-vars \& SF(n1)-vars \& \Box(p_{c2} = \#a \land I)\)
have \( \vdash ?G \rightarrow \Diamond(p_{c1} = \#a \land p_{c2} = \#a \land I) \)
proof –

Rule SF1 takes us from \( p_{c1} = b \) to \( p_{c1} = g \).

have bg1: \( \vdash ?G \rightarrow (p_{c1} = \#b \rightarrow p_{c1} = \#g) \)
proof (rule SF1)
   show \( \neg p_{c1} = \#b \land [(n1 \lor n2) \land \neg beta2]\)-vars \( \rightarrow o(p_{c1} = \#b) \lor o(p_{c1} = \#g) \)
   by (auto simp: Sact2-defs vars-def tla-defs)
next
   show \( \neg p_{c1} = \#b \land ((n1 \lor n2) \land \neg beta2) \land n1\)-vars \( \rightarrow o(p_{c1} = \#g) \)
   by (auto simp: n1-def alpha1-def beta1-def gamma1-def vars-def tla-defs)
next
   have \( \vdash p_{c1} = \#b \rightarrow Enabled\) \( \langle n1\rangle\)-vars
   unfolding enab-n1[int-rewrite] by (auto simp: tla-defs)
   hence \( \vdash \Box(p_{c1} = \#b) \rightarrow Enabled\) \( \langle n1\rangle\)-vars
   by (rule lift-imp-trans[OF ax1])
   hence \( \vdash \Box(p_{c1} = \#b) \rightarrow \Diamond Enabled\) \( \langle n1\rangle\)-vars
   by (rule lift-imp-trans[OF - E3])
   thus \( \vdash \Box(p_{c1} = \#b) \land \Box[(n1 \lor n2) \land \neg beta2]\)-vars \& \Box(p_{c2} = \#a \land I) \)
   \( \rightarrow \Diamond Enabled\) \( \langle n1\rangle\)-vars
   by auto
   qed

Similarly, \( p_{c1} = b \) leads to \( p_{c1} = g \).

have ga1: \( \vdash ?G \rightarrow (p_{c1} = \#g \rightarrow p_{c1} = \#a) \)
proof (rule SF1)
\[
\begin{align*}
\text{show } \neg \circ \#c_1 & = \#a \\
\circ (\#c_1 & = \#a) \\
\text{by (auto simp: Sact2-defs vars-def tla-defs)} \\
\text{next} \\
\text{show } \neg \circ (\#c_1 & = \#a) \\
\circ (\#c_1 & = \#a) \\
\text{by (auto simp: n1-def alpha1-def beta1-def gamma1-def vars-def tla-defs)} \\
\text{next} \\
\text{show } \neg \circ (\#c_1 & = \#a) \\
\circ (\#c_1 & = \#a) \\
\text{by (auto simp: vars-def tla-defs)} \\
\text{next} \\
\text{have } \vdash \circ (\#c_1 & = \#a) \\
\text{by auto} \\
\text{qed} \\
\end{align*}
\]
We can now prove the main theorem, which states that \( \psi \) implements \( \phi \).

**Theorem (in Secondprogram)** impl: \( \vdash \psi \rightarrow \phi \)

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 output channel, and a function representing the buffer state.

We specify a simple FIFO buffer and prove that two FIFO buffers in a row implement a FIFO buffer.

**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.
internal queue, and a third one representing the output channel. These parameters will be instantiated later in the definition of the double FIFO.

**definition** $\text{BInit}$ :: 'a statefun $\Rightarrow$ 'a list statefun $\Rightarrow$ 'a statefun $\Rightarrow$ temporal

where $\text{BInit } ic\ q\ oc \equiv \text{TEMP}\ \$q = \#[]$  
$\land\ \$ic = \$oc$  — initial condition of buffer

**definition** $\text{Enq}$ :: 'a statefun $\Rightarrow$ 'a list statefun $\Rightarrow$ 'a statefun $\Rightarrow$ temporal

where $\text{Enq } ic\ q\ oc \equiv \text{TEMP}\ \$ic \neq \$ic$  
$\land\ q\$ = $\$q @ [\ \$ic\ ]$  
$\land\ \$oc = \$oc$  — enqueue a new value

**definition** $\text{Deq}$ :: 'a statefun $\Rightarrow$ 'a list statefun $\Rightarrow$ 'a statefun $\Rightarrow$ temporal

where $\text{Deq } ic\ q\ oc \equiv \text{TEMP}\ \#0 < \text{length}\ <\$q\>$  
$\land\ \$oc = \text{hd}\ <\$q\>$  
$\land\ \$ic = \$ic$  — dequeue value at front

**definition** $\text{Nxt}$ :: 'a statefun $\Rightarrow$ 'a list statefun $\Rightarrow$ 'a statefun $\Rightarrow$ temporal

where $\text{Nxt } ic\ q\ oc \equiv \text{TEMP}\ \text{(Enq } ic\ q\ oc \lor \text{Deq } ic\ q\ oc\ )$

— internal specification with buffer visible

**definition** $\text{ISpec}$ :: 'a statefun $\Rightarrow$ 'a list statefun $\Rightarrow$ 'a statefun $\Rightarrow$ temporal

where $\text{ISpec } ic\ q\ oc \equiv \text{TEMP}\ \text{BInit } ic\ q\ oc$  
$\land\ \square\ [\text{Nxt } ic\ q\ oc]-\{(ic,q,oc)\}$  
$\land\ \text{WF}(\text{Deq } ic\ q\ oc)-\{(ic,q,oc)\}$

— external specification: buffer hidden

**definition** $\text{Spec}$ :: 'a statefun $\Rightarrow$ 'a statefun $\Rightarrow$ temporal

where $\text{Spec } ic\ oc \equiv\ \text{TEMP}\ (\text{EEX } q.\ \text{ISpec } ic\ q\ oc)$

### 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 $\text{noreps}$ is true if no two subsequent elements in a list are identical.

**definition** $\text{noreps}$ :: 'a list $\Rightarrow$ bool

where $\text{noreps } xs \equiv \forall\ i < \text{length } xs - 1.\ xs!i \neq xs!(\text{Suc } i)$
**Definition**: \( B\text{Inv} :: \text{a statefun} \Rightarrow \text{a list statefun} \Rightarrow \text{a statefun} \Rightarrow \text{temporal} \)

Where \( B\text{Inv} \ ic \ q \ oc \equiv \text{TEMP List}.\text{last} <\$oc \neq \$q> = \$ic \land \text{noreps} <\$oc \neq \$q> \)

**Lemmas**: \( \text{buffer-defs} = B\text{Init-def Enq-def Deq-def Nxt-def} \)

\( \text{ISpec-def Spec-def BInv-def} \)

**Lemma** ISpec-stutinv: \( \text{STUTINV} (\text{ISpec ic q oc}) \)

Unfolding buffer-defs by \( (\text{simp add: bothstutinvs livestutinv}) \)

**Lemma** Spec-stutinv: \( \text{STUTINV Spec ic oc} \)

Unfolding buffer-defs by \( (\text{simp add: bothstutinvs livestutinv exzSTUT}) \)

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 \( v : \alpha \vdash \text{Unchanged v} \longrightarrow \text{Unchanged q} \)

Shows \( \sim <\text{Deq ic q oc}> v = \text{Deq ic q oc} \)

Proof (auto simp: tla-defs)

Fix \( w \)

Assume \( \text{deq}: w \models \text{Deq ic q oc} \text{ and anch: v (w (Suc 0)) = v (w 0)} \)

From \( \text{unch v [unlifted]} \) have \( q (w (Suc 0)) = q (w 0) \)

By (auto simp: tla-defs)

With \( \text{deq} \) show False by (auto simp: Deq-def tla-defs)

Qed

**Lemma** Deq-enabledE: \( \vdash \text{Enabled <Deq ic q oc> (ic,q,oc)} \longrightarrow \$q \sim = \#[] \)

By (auto elim!: enabledE simp: Deq-def tla-defs)

We now prove that \( B\text{Inv} \) is an invariant of the Buffer specification.
We need several lemmas about `noreps` that are used in the invariant proof.

**Lemma noreps-empty** \(\text{simp} \): `noreps []`

by (auto simp: noreps-def)

**Lemma noreps-singleton**

`noreps [x]` — special case of following lemma

by (auto simp: noreps-def)

**Lemma noreps-cons** \(\text{simp} \):

`noreps (x # xs) = (noreps xs ∧ (xs = [] ∨ x ≠ hd xs))`

proof (auto simp: noreps-singleton)

assume cons: `noreps (x # xs)`

show `noreps xs`

proof (auto simp: noreps-def)

fix `i`

assume `i < length xs - Suc 0` and `eq: xs!i = xs!(Suc i)`

from `i` have `Suc i < length (x#xs) - 1` by auto

moreover

from `eq` have `(x#xs)!(Suc i) = (x#xs)!(Suc (Suc i))` by auto

moreover

note cons

ultimately show `False` by (auto simp: noreps-def)

qed

next

assume 1: `noreps (hd xs # xs)` and 2: `xs ≠ []`

from 2 obtain `x xxs` where `xs = x # xxs` by (cases xs, auto)

with 1 show `False` by (auto simp: noreps-def)

next

assume 1: `noreps xs` and 2: `x ≠ hd xs`

show `noreps (x # xs)`

proof (auto simp: noreps-def)

fix `i`

assume `i < length xs` and `eq: (x # xs)!i = xs!i`

from `i` obtain `y ys` where `xs = y # ys` by (cases xs, auto)

show `False`

proof (cases `i`)

assume `i = 0`

with `eq 2 xs` show `False` by auto

next

fix `k`

assume `k: i = Suc k`

with `i eq xs 1` show `False` by (auto simp: noreps-def)

qed

qed

**Lemma noreps-append** \(\text{simp} \):

`noreps (xs @ ys) = (noreps xs ∧ noreps ys ∧ (xs = [] ∨ ys = [] ∨ List.last xs ≠ hd ys))`

proof auto
assume 1: noreps (xs @ ys)

show noreps xs
proof (auto simp: noreps-def)
  fix i
  assume i: i < length xs - Suc 0 and eq: xs!i = xs!(Suc i)
  from i have i < length (xs @ ys) - Suc 0 by auto
  moreover
  from i eq have (xs @ ys)!i = (xs@ys)!(Suc i) by (auto simp: nth-append)
  moreover note 1
  ultimately show False by (auto simp: noreps-def)
qed

next

assume 1: noreps (xs @ ys)

show noreps 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 4 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 \(i < \text{length } \text{xs}\))

  case True
  hence \(i < \text{length } (x \# \text{xs})\) by simp

  hence \(\text{xs}i : ((x \# \text{xs}) @ \text{ys})i = (x \# \text{xs})i\)
  unfolding nth-append by simp

  from True have \((\text{xs} @ \text{ys})i = \text{xs}i\) by (auto simp: nth-append)
  with True \(\text{xs}i\) eq 1 \(\text{xs}\) show False by (auto simp: noreps-def)

next

  assume \(i2 : \neg(i < \text{length } \text{xs})\)
  show False
  proof (cases \(i = \text{length } \text{xs}\))

    case True
    with \(\text{xs}\) have \(\text{xs}i : (\text{xs} @ \text{ys})i = \text{List.last } \text{xs}\)
    by (auto simp: nth-append last-conv-nth)

    from True \(\text{xs}\) \(\text{ys}\) have \((\text{xs}@\text{ys})!\ Suc i = y\)
    by (auto simp: nth-append)

    with \(3\) \(\text{ys}\) eq \(\text{xs}i\) show False by simp

  next

    case False
    with \(i2\) \(\text{xs}\) have \(\neg(i < \text{length } \text{xs})\) by auto

    hence \((\text{xs}@\text{ys})!i = \text{ys}!(i - \text{length } \text{xs})\)
    by (simp add: nth-append)

    moreover

    from \(\text{xs}i\) have \(\text{Suc } i - \text{length } \text{xs} = \text{Suc } (i - \text{length } \text{xs})\) by auto

    with \(\text{xs}i\) have \((\text{xs}@\text{ys})!\ (\text{Suc } i) = \text{ys}!(\text{Suc } (i - \text{length } \text{xs}))\)
    by (simp add: nth-append)

    moreover

    from \(i\) \(\text{xs}i\) have \(i - \text{length } \text{xs} < \text{length } \text{ys} - 1\) by auto

    with \(\neg\) have \(\text{ys}!(i - \text{length } \text{xs}) \neq \text{ys}!(\text{Suc } (i - \text{length } \text{xs}))\)
    by (auto simp: noreps-def)

    moreover

    note eq
    ultimately show False by simp

  qed

qed

lemma ISpec-BInv-lemma:
\(\vdash \text{BInit } ic \ q \ oc \land \Box[\text{Nxt } ic \ q \ oc]-(ic,q,oc) \rightarrow \Box(\text{BInv } ic \ q \ oc)\)

proof (rule invmono)

  show \(\vdash \text{BInit } ic \ q \ oc \rightarrow \text{BInv } ic \ q \ oc\)
  by (auto simp: BInit-def BInv-def)

next

  have enq: \(" \text{Enq } ic \ q \ oc \rightarrow \text{BInv } ic \ q \ oc \rightarrow o(\text{BInv } ic \ q \ oc)\"
  by (auto simp: Enq-def BInv-def tla-defs)

  have deq: \(" \text{Deq } ic \ q \ oc \rightarrow \text{BInv } ic \ q \ oc \rightarrow o(\text{BInv } ic \ q \ oc)\"
  by (auto simp: Deq-def BInv-def tla-defs neq-Nil-conv)
have \[ \text{UNCH}: \sim \text{Unchanged}(ic,q,oc) \rightarrow BInv ic q oc \rightarrow \circ(BInv ic q oc) \] by (auto simp: BInv-def tla-defs)

show \[ \sim BInv ic q oc \land [\text{Nxt} ic q oc]-(ic, q, oc) \rightarrow \circ(BInv ic q oc) \] by (auto simp: Nxt-def actrans-def elim: enq[unlift-rule] deq[unlift-rule] unch[unlift-rule])

qed

theorem ISpec-BInv: \[ \vdash \text{ISpec } ic q oc \rightarrow \Box(BInv ic q oc) \] by (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
and mid :: 'a statefun — channel linking the two buffers
and out :: 'a statefun — output channel for double FIFO
and q1 :: 'a list statefun — inner queue of first FIFO
and q2 :: 'a list statefun — inner queue of second FIFO
and vars
defines vars \equiv \text{LIFT } (\text{inp, mid, out, } q1, q2)
assumes DB-base: basevars vars

begin

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 inp and output channel mid whereas the second one receives from mid and outputs on 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 enqueueing and dequeueing (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 enqueueing and dequeueing do not happen simultaneously.

definition DBSpec
where DBSpec \equiv TEMP ISpec inp q1 mid
\land ISpec mid q2 out
\land \Box[\neg(\text{Enq} \text{ inp } q1 \text{ mid } \land \text{Deq} mid q2 out)]-vars

The proof rules of TLA are geared towards specifications of the form \[ Init \land \Box[Next].vars \land L, \] and we prove that DBSpec corresponds to a specification in this form, which we now define.

definition FullInit
where FullInit \equiv TEMP (BInit inp q1 mid \land BInit mid q2 out)

definition FullNxt
where $\text{FullNxt} \equiv \text{TEMP} (\text{Enq inp } q1 \text{ mid} \land \text{Unchanged} (q2,\text{out})$
\begin{align*}
\lor & \text{Deq inp } q1 \text{ mid} \land \text{Enq mid } q2 \text{ out} \\
\lor & \text{Deq mid } q2 \text{ out} \land \text{Unchanged} (\text{inp},q1))
\end{align*}

\text{definition FullSpec}
where $\text{FullSpec} \equiv \text{TEMP FullInit}$
\begin{align*}
\land & \Box[\text{FullNxt}]-\text{vars} \\
\land & \text{WF}(\text{Deq inp } q1 \text{ mid})-\text{vars} \\
\land & \text{WF}(\text{Deq mid } q2 \text{ out})-\text{vars}
\end{align*}

The concatenation of the two queues will serve as the refinement mapping.

\text{definition $qc : \text{'}a Map statefun$}
where $qc \equiv \text{LIFT} (q2 @ q1)$

\text{lemmas $\text{db-defs} = \text{buffer-defs DBSpec-def FullInit-def FullNxt-def FullSpec-def}$}
\text{qc-def vars-def}$

\text{lemma $\text{DBSpec-stutinv}$: $\text{STUTINV DBSpec}$}
unfolding $\text{db-defs by simp add: bothstutinvs livestutinv}$

\text{lemma $\text{FullSpec-stutinv}$: $\text{STUTINV FullSpec}$}
unfolding $\text{db-defs by simp add: bothstutinvs livestutinve}$

We prove that $\text{DBSpec}$ implies $\text{FullSpec}$. (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.

\text{lemma $\text{DBNxt-then-FullNxt}$:}
$\vdash \Box[\Box[\text{Nxt inp } q1 \text{ mid}]}-(\text{inp},q1,\text{mid})$
\begin{align*}
\land & \Box[\text{Nxt mid } q2 \text{ out}]}-(\text{mid},q2,\text{out}) \\
\land & \Box[\neg(\text{Enq inp } q1 \text{ mid} \land \text{Deq mid } q2 \text{ out})]-\text{vars} \\
\rightarrow & \Box[\text{FullNxt}]-\text{vars}
\end{align*}
(is $\vdash \Box[?\text{inv} \land ?\text{nxts} \rightarrow \Box[\text{FullNxt}]-\text{vars}$)

\text{proof}

\text{have $\vdash \Box[\text{Nxt inp } q1 \text{ mid}]}-(\text{inp},q1,\text{mid})$
\begin{align*}
\land & \Box[\text{Nxt mid } q2 \text{ out}]}-(\text{mid},q2,\text{out}) \\
\rightarrow & \Box[\text{[Nxt inp } q1 \text{ mid}]}-(\text{inp},q1,\text{mid}) \\
\land & \Box[\text{Nxt mid } q2 \text{ out}]}-(\text{mid},q2,\text{out})]-((\text{inp},q1,\text{mid}),\text{(mid},q2,\text{out}))
\end{align*}
(is $\vdash ?\text{tmp} \rightarrow \Box[?b1b2].?-\text{vs}$)

\text{by (auto simp: M12[int-reverse])}

\text{moreover}
\text{have $\vdash \Box[?b1b2].?-\text{vs} \rightarrow \Box[?b1b2].?-\text{vs}$}

\text{by (rule R1, auto simp: vars-def lta-defs)}

\text{ultimately}
have 1: \(\vdash \Box [\text{Nxt inp } q_1 \text{ mid}]-(\text{inp}, q_1, \text{mid})\)
\(\wedge \Box [\text{Nxt mid } q_2 \text{ out}]-(\text{mid}, q_2, \text{out})\)
\(\longrightarrow \Box [\text{Nxt inp } q_1 \text{ mid}]-(\text{inp}, q_1, \text{mid})\)
\(\wedge [\text{Nxt mid } q_2 \text{ out}]-(\text{mid}, q_2, \text{out})]\)-vars
by force

have 2: \(\vdash \Box [\text{?b1b2}]-\text{vars} \wedge \Box [\neg(\text{Enq inp } q_1 \text{ mid} \wedge \text{Deq mid } q_2 \text{ out})]-\text{vars}\)
\(\longrightarrow \Box [\text{?b1b2} \wedge \neg(\text{Enq inp } q_1 \text{ mid} \wedge \text{Deq mid } q_2 \text{ out})]-\text{vars}\)
(is \(\vdash \text{tmp2} \longrightarrow \Box [\text{?mid}]-\text{vars}\))
by (simp add: M8[int-rewrite])

have \(\vdash \text{inv} \longrightarrow \# \text{True} \) by auto

moreover

have \(\neg \text{inv} \wedge \neg \text{mid} \longrightarrow [\text{FullNxt}]-\text{vars}\)
proof —

have A: \(\neg \text{Nxt inp } q_1 \text{ mid}\)
\(\longrightarrow [\text{Nxt mid } q_2 \text{ out}]-(\text{mid}, q_2, \text{out})\)
\(\longrightarrow \neg(\text{Enq inp } q_1 \text{ mid} \wedge \text{Deq mid } q_2 \text{ out})\)
\(\longrightarrow \text{inv}\)
\(\longrightarrow \text{FullNxt}\)

proof —

have enq: \(\neg \text{Enq inp } q_1 \text{ mid}\)
\(\wedge [\text{Nxt mid } q_2 \text{ out}]-(\text{mid}, q_2, \text{out})\)
\(\wedge \neg(\text{Deq mid } q_2 \text{ out})\)
\(\longrightarrow \text{Unchanged } (q_2, \text{out})\)
by (auto simp: db-defs tla-defs)

have deq1: \(\neg \text{Deq inp } q_1 \text{ mid} \longrightarrow \text{inv} \longrightarrow \text{mid}$\neq \text{mid}$\)
by (auto simp: Deq-def BInv-def)

have deq2: \(\neg \text{Deq mid } q_2 \text{ out} \longrightarrow \text{mid} = \text{mid}\$
by (auto simp: Deq-def)

have deq: \(\neg \text{Deq inp } q_1 \text{ mid}\)
\(\wedge [\text{Nxt mid } q_2 \text{ out}]-(\text{mid}, q_2, \text{out})\)
\(\wedge \text{inv}\)
\(\longrightarrow \text{Enq mid } q_2 \text{ out}\)
by (force simp: Nxt-def tla-defs

dest: deq1[unlift-rule] deq2[unlift-rule])

with enq show ?thesis by (force simp: Nxt-def FullNxt-def)

qed

have B: \(\neg \text{Nxt mid } q_2 \text{ out}\)
\(\longrightarrow \text{Unchanged } (\text{inp}, q_1, \text{mid})\)
\(\longrightarrow \text{FullNxt}\)
by (auto simp: db-defs tla-defs)

have C: \(\vdash \text{Unchanged } (\text{inp}, q_1, \text{mid})\)
\(\longrightarrow \text{Unchanged } (\text{mid}, q_2, \text{out})\)
\(\longrightarrow \text{Unchanged vars}\)
by (auto simp: vars-def tla-defs)

show ?thesis
by (force simp: actrans-def

It is now easy to show that $DBSpec$ refines $FullSpec$.

**Theorem DBSpec-impl-FullSpec:** $\vdash DBSpec \rightarrow FullSpec$

**Proof**

- **have 1:** $\vdash DBSpec \rightarrow FullInit$
  - by (auto simp: DBSpec-def FullInit-def ISpec-def)
- **have 2:** $\vdash DBSpec \rightarrow □[FullNxt]-vars$
  - by (auto simp: DBSpec-def intro: ISpec-BInv)
  - moreover have $\vdash DBSpec \land □(BInv inp q1 mid) \rightarrow □[FullNxt]-vars$
    - by (auto simp: DBSpec-def ISpec-def intro: DBNxt-then-FullNxt)
  - ultimately show $\vdash$ thesis by force

**qed**

- **have 3:** $\vdash DBSpec \rightarrowWF(Deq inp q1 mid)-vars$
  - by (auto simp: DBSpec-def ISpec-def WeakF-def)
  - deq[int-rewrite] deq[THEN AA26,int-rewrite]

**qed**

- **have 4:** $\vdash DBSpec \rightarrowWF(Deq mid q2 out)-vars$
  - by (auto simp: DBSpec-def ISpec-def WeakF-def)
  - deq[int-rewrite] deq[THEN AA26,int-rewrite]

**qed**
show thesis
by (auto simp: FullSpec-def
    elim: 1[unlift-rule] 2[unlift-rule] 3[unlift-rule]
     4[unlift-rule])

qed

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: ⊢ FullInit → BInit inp qc out
by (auto simp: db-defs tla-defs)

lemma Full-step-simulation:
|∼ [FullNxt]-vars → [Nxt inp qc out].(inp,qc,out)
by (auto simp: db-defs tla-defs)

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.

lemma Deq1-enabled: ⊢ Enabled (Deq inp q1 mid)-vars = ($q1 ≠ #[])
proof -
  have 1: |∼ (Deq inp q1 mid)-vars = Deq inp q1 mid
        by (rule Deq-visible, auto simp: vars-def tla-defs)
  have ⊢ Enabled (Deq inp q1 mid) = ($q1 ≠ #[])
        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

lemma Deq2-enabled: ⊢ Enabled (Deq mid q2 out)-vars = ($q2 ≠ #[])
proof -
  have 1: |∼ (Deq mid q2 out)-vars = Deq mid q2 out
        by (rule Deq-visible, auto simp: vars-def tla-defs)
  have ⊢ Enabled (Deq mid q2 out) = ($q2 ≠ #[])
        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.

lemma Full-fairness:
⊢ □[FullNxt]-vars ∧ WF(Deq mid q2 out)-vars ∧ □WF(Deq inp q1 mid)-vars
→ WF(Deq inp qc out)-(imp, qc, out)

proof (rule WF2)
— the helpful action is the Deq action of the second queue

show \[\sim (\text{FullNxt} \land \text{Deq mid q2 out})\text{-vars} \rightarrow (\text{Deq inp qc out})\text{-imp, qc, out}\]
by (auto simp: db-defs tla-defs)

next
— the helpful condition is the second queue being non-empty

show \[\sim (\{q2 \neq []\} \land o(q2 \neq [])) \land (\text{FullNxt} \land \text{Deq mid q2 out})\text{-vars} \rightarrow \text{Deq mid q2 out}\]
by (auto simp: tla-defs)

next
show \[\sim q2 \neq [] \land \text{Enabled} (\text{Deq inp qc out})\text{-imp, qc, out} \rightarrow \text{Enabled} (\text{Deq mid q2 out})\text{-vars}\]
unfolding Deq2-enabled[int-rewrite] by auto

next

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 \[\top\lnot\square(\text{FullNxt} \land \sim(\text{Deq mid q2 out}))\text{-vars}\]
proof (rule STL4)
have \[\sim q2 \neq [] \land (\text{FullNxt} \land \sim(\text{Deq mid q2 out}))\text{-vars} \rightarrow o(q2 \neq []))\]
by (auto simp: qc-def tla-defs)
from this[THEN INV1] show \[\top\lnot(\text{FullNxt} \land \sim \text{Deq mid q2 out})\text{-vars} \rightarrow (q2 \neq [] \rightarrow \square(q2 \neq []))\]
by auto

qed

hence 1: \[\top\lnot\square(\text{FullNxt} \land \sim(\text{Deq mid q2 out}))\text{-vars} \rightarrow (\square(q2 \neq [] \rightarrow \square(q2 \neq []))\]
by (force intro: E31[unlift-rule])

have 2: \[\top\lnot(\text{FullNxt} \land \sim(\text{Deq mid q2 out}))\text{-vars} \land \text{WF}(\text{Deq inp q1 mid})\text{-vars}\]
\rightarrow (\text{Enabled} (\text{Deq inp qc out})\text{-imp, qc, out} \rightarrow q2 \neq [])

proof

have q: \[\top(q \neq []) \rightarrow (q1 \neq [] \lor q2 \neq []))\]
by (auto simp: qc-def tla-defs)

have \[\top\lnot(\text{FullNxt} \land \sim(\text{Deq mid q2 out}))\text{-vars} \land \text{WF}(\text{Deq inp q1 mid})\text{-vars}\]
\rightarrow (q1 \neq [] \rightarrow q2 \neq []))
proof (rule WF1)

show \[\sim q1 \neq [] \land (\text{FullNxt} \land \sim \text{Deq mid q2 out})\text{-vars} \rightarrow o(q1 \neq []) \lor o(q2 \neq [])\]
by (auto simp: db-defs tla-defs)

next

show \[\sim q1 \neq []\]
\[ \begin{align*}
& \land \langle (\text{FullNxt} \land \neg \text{Deq mid q2 out}) \land \text{Deq inp q1 mid} \rangle \text{-vars} \rightarrow \\
& \circ(s2 \neq #[]) \\
& \text{by (auto simp: db-defs tla-defs)}
\end{align*} \]

next

\begin{align*}
\text{show} & \vdash s1 \neq #[] \rightarrow \text{Enabled} \langle \text{Deq inp q1 mid} \rangle \text{-vars} \\
& \text{by (simp add: Deq1-enabled[int-rewrite])}
\end{align*}

\begin{align*}
\text{next} & \\
\text{show} & \neg s1 \neq #[] \land \text{Unchanged vars} \rightarrow \circ(s1 \neq #[]) \\
& \text{by (auto simp: vars-def tla-defs)}
\end{align*}

\text{qed}

\text{hence} \vdash \square\langle \text{FullNxt} \land \neg (\text{Deq mid q2 out}) \rangle \text{-vars} \\
\land \text{WF(Deq inp q1 mid)-vars} \\
\rightarrow (sqc \neq #[] \rightarrow s2 \neq #[])

\text{by (auto simp: gc[int-rewrite] LT17[int-rewrite] LT11[int-rewrite])}


\text{moreover}

\begin{align*}
\text{have} & \vdash \text{Enabled} \langle \text{Deq inp qc out} \rangle -(\text{inp, qc, out}) \rightarrow sqc \neq #[] \\
& \text{by (rule Deq-enabledE[THEN LT3])}
\end{align*}

\text{ultimately show} \ ?\text{thesis by (force elim: LT13[unlift-rule])}

\text{qed}

\text{with LT6}

\begin{align*}
\text{have} & \vdash \square\langle \text{FullNxt} \land \neg (\text{Deq mid q2 out}) \rangle \text{-vars} \\
& \land \text{WF(Deq mid q2 out)-vars} \\
& \land \square\text{Enabled} \langle \text{Deq inp qc out} \rangle -(\text{inp, qc, out}) \\
& \rightarrow \circ(sq2 \neq #[]) \\
& \text{by force}
\end{align*}

\text{with I E16}

\begin{align*}
\text{show} & \vdash \square\langle \text{FullNxt} \land \neg (\text{Deq mid q2 out}) \rangle \text{-vars} \\
& \land \square\text{WF(Deq inp q1 mid)-vars} \\
& \land \square\square\text{Enabled} \langle \text{Deq inp qc out} \rangle -(\text{inp, qc, out}) \\
& \rightarrow \circ\square(sq2 \neq #[]) \\
& \text{by force}
\end{align*}

\text{qed}

Putting everything together, we obtain that \text{FullSpec} refines the Buffer specification under the refinement mapping.

\text{theorem FullSpec-impl-ISpec:} \vdash \text{FullSpec} \rightarrow \text{ISpec inp qc out}

\text{unfolding FullSpec-def ISpec-def}

\text{using FullInit Full-step-simulation[THEN M11] Full-fairness}

\text{by force}

\text{theorem FullSpec-impl-Spec:} \vdash \text{FullSpec} \rightarrow \text{Spec inp out}

\text{unfolding Spec-def using FullSpec-impl-ISpec}

\text{by (force intro: cexf[unfold-rule])}

By transitivity, two buffers in a row also implement a single buffer.

\text{theorem DBSpec-impl-Spec:} \vdash \text{DBSpec} \rightarrow \text{Spec inp out}

\text{by (rule lift-imp-trans[OF DBSpec-impl-FullSpec FullSpec-impl-Spec])}

\[ 105 \]
References


