A Definitional Encoding of TLA in Isabelle/HOL

Gudmund Grov & Stephan Merz

August 16, 2018

Abstract

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

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

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

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

Contents

1 (Infinite) Sequences .............................................. 3
1.1 Some operators on sequences ................................. 3
1.1.1 Properties of first and second ......................... 4
1.1.2 Properties of (\|s) ...................................... 4
1.1.3 Properties of (##) ...................................... 4
1.2 Finite and Empty Sequences ................................. 5
1.2.1 Properties of emptyseq ................................. 5
Liveness
6.1 Properties of -Enabled ........................................ 46
6.2 Fairness Properties ............................................ 47
6.3 Stuttering Invariance ........................................... 49

Representing state in TLA* ...................................... 49
7.1 Temporal Quantifiers ........................................... 51

A simple illustrative example .................................. 52

Lamport’s Inc example ............................................. 53

Reﬁning a Buffer Speciﬁcation ................................ 56
10.1 Buffer speciﬁcation ........................................... 56
10.2 Properties of the buffer ...................................... 57
10.3 Two FIFO buffers in a row implement a buffer .... 58

(Infinit) 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 par-
ticular 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 |ₙ 60)
where s |ₙ i ≡ λ n. s (n+i)

definition tail :: 'a seq ⇒ 'a seq
where tail s ≡ s |₁

definition app :: 'a ⇒ ('a seq ⇒ ('a seq)) ⇒ ('a seq) (infixl ## 60)
where s ## σ ≡ λ n. if n=0 then s else σ (n - 1)

s |ₙ i returns the suffix of sequence s from index i. first returns the first element of a sequence while second returns the second element. tail returns the sequence starting at the second element. s ## σ prefixes the sequence σ by element s.

1.1.1 Properties of first and second

lemma first-tail-second: first(tail s) = second s
⟨proof⟩

1.1.2 Properties of (|ₙ)

lemma suffix-first: first (s |ₙ n) = s n
⟨proof⟩

lemma suffix-second: second (s |ₙ n) = s (Suc n)
⟨proof⟩

lemma suffix-plus: s |ₙ n |ₙ m = s |ₙ (m + n)
⟨proof⟩

lemma suffix-commute: ((s |ₙ n) |ₙ m) = ((s |ₙ m) |ₙ n)
⟨proof⟩

lemma suffix-plus-com: s |ₙ m |ₙ n = s |ₙ (m + n)
⟨proof⟩

lemma suffix-zero[simp]: s |ₙ 0 = s
⟨proof⟩

lemma suffix-tail: s |ₙ 1 = tail s
⟨proof⟩

lemma tail-suffix-suc: s |ₙ (Suc n) = tail (s |ₙ n)
⟨proof⟩

1.1.3 Properties of (##)

lemma seq-app-second: (s ## σ) 1 = σ 0
lemma seq-app-first: \((s \#\# \sigma) \ 0 = s\)

lemma seq-app-first-tail: \((\text{first} \ s) \#\# (\text{tail} \ s) = s\)

lemma seq-app-tail: \(\text{tail} \ (x \#\# s) = s\)

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

\(\langle proof \rangle\)

1.2 Finite and Empty Sequences

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

definition fin :: ('a seq) ⇒ bool
where fin s ≡ ∃ i. ∀ j ≥ i. s j = s i

abbreviation inf :: ('a seq) ⇒ bool
where inf s ≡ ¬(fin s)

definition last :: ('a seq) ⇒ nat
where last s ≡ LEAST i. (∀ j ≥ i. s j = s i)

definition laststate :: ('a seq) ⇒ 'a
where laststate s ≡ s (last s)

definition emptyseq :: ('a seq) ⇒ bool
where emptyseq ≡ λ s. ∀ i. s i = s 0

abbreviation notemptyseq :: ('a seq) ⇒ bool
where notemptyseq s ≡ ¬(emptyseq s)

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

1.2.1 Properties of \(\text{emptyseq}\)

lemma empty-is-finite: assumes emptyseq s shows fin s
lemma empty-suffix-is-empty: assumes H: emptyseq s shows emptyseq (s |ₙ n)

lemma suc-empty: assumes H1: emptyseq (s |ₙ m) shows emptyseq (s |ₙ (Suc m))

lemma empty-suffix-exteq: assumes H: emptyseq s shows (s |ₙ s n) m = s m

lemma empty-suffix-eq: assumes H: emptyseq s shows (s |ₙ s n) = s

lemma seq-empty-all: assumes H: emptyseq s shows s i = s j

1.2.2 Properties of Sequence.last and laststate

lemma fin-stut-after-last: assumes H: fin s shows ∀ j ≥ last s. s j = s (last s)

1.3 Stuttering Invariance

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

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

definition nonstutseq :: ('a seq) ⇒ bool
where nonstutseq s ≡ ∀ i. s i = s (Suc i) ⟹ (∀ j > i. s i = s j)

definition stutstep :: ('a seq) ⇒ nat ⇒ bool
where stutstep s n ≡ (s n = s (Suc n))

definition nextnat :: ('a seq) ⇒ nat
where nextnat s ≡ if emptyseq s then 0 else LEAST i. s i ≠ s 0

definition nextsuffix :: ('a seq) ⇒ ('a seq)
where nextsuffix s ≡ s |ₙ (nextnat s)

fun next :: nat ⇒ ('a seq) ⇒ ('a seq) where
\( \text{next } 0 = \text{id} \)
\( \text{next } (\text{Suc } n) = \text{nextsuffix } \circ (\text{next } n) \)

**Definition**
\[
\text{collapse } :: \ (\text{’a seq}) \Rightarrow (\text{’a seq}) (\text{’z})
\]
\[
\text{where } \xi s \equiv \lambda n. (\text{next } n s) 0
\]

Predicate \(\text{nonstutseq}\) identifies sequences without any stuttering steps – except possibly for infinite stuttering at the end. Further, \(\text{stutstep } s n\) 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. \(\xi s\) formalises Lamports \(\xi\) 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 \(\xi s\) returns, is the state after the \(n\)th change. \(\text{next } n s\) is defined by primitive recursion on \(n\) using function composition of function \(\text{nextsuffix}\). E.g. \(\text{next } 3 s\) equals \(\text{nextsuffix } (\text{nextsuffix } (\text{nextsuffix } s))\). \(\text{nextsuffix } s\) returns the suffix of the sequence starting at the next changing state. All the real computation is done in this function. Firstly, an empty sequence will obviously not contain any changes, and \(0\) is therefore returned. In this case \(\text{nextsuffix}\) behaves like the identify function. If the sequence is not empty then the smallest number \(i\) such that \(s i\) is different from the initial state is returned. This is achieved by \(\text{Least}\).

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

**Lemma** \(\text{seq-empty-is-nonstut}\):
\[
\text{assumes } H: \text{emptyseq } s \text{ shows } \text{nonstutseq } s
\]
\[
\text{⟨proof}⟩
\]

**Lemma** \(\text{notempty-exist-nonstut}\):
\[
\text{assumes } H: \neg \text{emptyseq } (s \mid s m) \text{ shows } \exists i. s i \neq s m \land i > m
\]
\[
\text{⟨proof}⟩
\]

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

**Lemma** \(\text{nextnat-le-unch}\):
\[
\text{assumes } H: n < \text{nextnat } s \text{ shows } s n = s 0
\]
\[
\text{⟨proof}⟩
\]

**Lemma** \(\text{stutnempty}\):
\[
\text{assumes } H: \neg \text{stutstep } s n \text{ shows } \neg \text{emptyseq } (s \mid s n
\]
\[
\text{⟨proof}⟩
\]

**Lemma** \(\text{notstutstep-nextnat1}\):
\[
\text{assumes } H: \neg \text{stutstep } s n \text{ shows } \text{nextnat } (s \mid s n) = 1
\]
\[
\text{⟨proof}⟩
\]

**Lemma** \(\text{stutstep-notempty-notempty}\):
\[
\text{assumes } h1: \text{emptyseq } (s \mid s \text{Suc } n) (\text{is emptyseq } ?sn)
\]
\[
\text{and } h2: \text{stutstep } s n
\]
\[
\text{shows emptyseq } (s \mid s n) (\text{is emptyseq } ?s)
\]
proof

lemma stutter-empty-suc:
  assumes stutter s n
  shows emptyseq (s |\_ s Suc n) = emptyseq (s |\_ s n)
  ⟨proof⟩

lemma stutter-notempty-sucnextnat:
  assumes h1: ¬ emptyseq (s |\_ s n) and h2: stutter s n
  shows (nextnat (s |\_ s n)) = Suc (nextnat (s |\_ s (Suc n)))
  ⟨proof⟩

lemma nextnat-empty-neq: assumes H: ¬ emptyseq s shows s (nextnat s) ≠ s 0
  ⟨proof⟩

lemma nextnat-empty-gzero: assumes H: ¬ emptyseq s shows nextnat s > 0
  ⟨proof⟩

1.3.3 Properties of nextsuffix

lemma empty-nextsuffix:
  assumes H: emptyseq s shows nextsuffix s = s
  ⟨proof⟩

lemma empty-nextsuffix-id:
  assumes H: emptyseq s shows nextsuffix s = id s
  ⟨proof⟩

lemma notstutstep-nextsuffix1:
  assumes H: ¬ stutter s n shows nextsuffix (s |\_ s n) = s |\_ s (Suc n)
  ⟨proof⟩

1.3.4 Properties of next

lemma next-suc-suffix: next (Suc n) s = nextsuffix (next n s)
  ⟨proof⟩

lemma next-suffix-com: nextsuffix (next n s) = (next n (nextsuffix s))
  ⟨proof⟩

lemma next-plus: next (m+n) s = next m (next n s)
  ⟨proof⟩

lemma next-empty: assumes H: emptyseq s shows next n s = s
  ⟨proof⟩

lemma notempty-nextnotzero:
  assumes H: ¬ emptyseq s shows (next (Suc 0) s) 0 ≠ s 0
  ⟨proof⟩

8
lemma next-ex-id: \( \exists \ i. \ s \ i = (\text{next } m \ s) \ 0 \)  
\( \langle \text{proof} \rangle \)

1.3.5 Properties of \( \natural \)

lemma emptyseq-collapse-eq: assumes \( A1: \text{emptyseq } s \) shows \( \natural s = s \) 
\( \langle \text{proof} \rangle \)

lemma empty-collapse-empty:
  assumes \( H: \text{emptyseq } s \) shows \( \text{emptyseq } (\natural s) \) 
\( \langle \text{proof} \rangle \)

lemma collapse-empty-empty:
  assumes \( H: \text{emptyseq } (\natural s) \) shows \( \text{emptyseq } s \) 
\( \langle \text{proof} \rangle \)

lemma collapse-empty-iff-empty [simp]: \( \text{emptyseq } (\natural s) = \text{emptyseq } s \) 
\( \langle \text{proof} \rangle \)

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) \( \Rightarrow \) ('a seq) \( \Rightarrow \) bool (infixl \( \approx \) 50) 
where \( \sigma \approx \tau \equiv (\natural \sigma) = (\natural \tau) \)

1.4.1 Properties of (\( \approx \))

lemma seqsim-refl [iff]: \( s \approx s \) 
\( \langle \text{proof} \rangle \)

lemma seqsim-sym: assumes \( H: s \approx t \) shows \( t \approx s \) 
\( \langle \text{proof} \rangle \)

lemma seqeq-imp-sim: assumes \( H: s = t \) shows \( s \approx t \) 
\( \langle \text{proof} \rangle \)

lemma seqsim-trans [trans]: assumes \( h1: s \approx t \) and \( h2: t \approx z \) shows \( s \approx z \) 
\( \langle \text{proof} \rangle \)

theorem sim-first: assumes \( H: s \approx t \) shows \( \text{first } s = \text{first } t \) 
\( \langle \text{proof} \rangle \)

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

lemma tail-sim-second: assumes \( H: \text{tail } s \approx \text{tail } t \) shows \( \text{second } s = \text{second } t \)
lemma seqsimilarI:
  assumes 1: first s = first t and 2: nextsuffix s ≈ nextsuffix t
  shows s ≈ t
⟨proof⟩

lemma seqsim-empty-empty:
  assumes H1: s ≈ t and H2: emptyseq s shows emptyseq t
⟨proof⟩

lemma seqsim-empty-iff-empty:
  assumes H: s ≈ t shows emptyseq s = emptyseq t
⟨proof⟩

lemma seq-empty-eq:
  assumes H1: s 0 = t 0 and H2: emptyseq s and H3: emptyseq t
  shows s = t
⟨proof⟩

lemma seqsim-natstutstep:
  assumes H: ¬ (stutstep s n) shows (s ∣ₙ (Suc n)) ≈ nextsuffix (s ∣ₙ n)
⟨proof⟩

lemma stat-nextsf-suc:
  assumes H: statstep s n shows nextsuffix (s ∣ₙ n) = nextsuffix (s ∣ₙ (Suc n))
⟨proof⟩

lemma seqsim-suffix-seqsim:
  assumes H: s ≈ t shows nextsuffix s ≈ nextsuffix t
⟨proof⟩

lemma seqsim-stutstep:
  assumes H: statstep s n shows (s ∣ₙ (Suc n)) ≈ (s ∣ₙ n) (is ?sn ≈ ?s)
⟨proof⟩

lemma addfeqstut: statstep ((first t) ## t) 0
⟨proof⟩

lemma addfeqsim: ((first t) ## t) ≈ t
⟨proof⟩

lemma addfirststat:
  assumes H: first s = second s shows s ≈ tail s
⟨proof⟩

lemma app-seqsimilar:
  assumes h1: s ≈ t shows (x ## s) ≈ (x ## t)
⟨proof⟩
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.

**Lemma simstep-disj1**: Assumes \( s \approx t \) shows \( \exists m. ((s |_s n) \approx (t |_s m)) \)

\[ \text{⟨proof} \]

**Lemma nextnat-le-seqsim**: Assumes \( n: n < \text{nextnat } s \) shows \( s \approx (s |_s n) \)

\[ \text{⟨proof} \]

**Lemma seqsim-prev-nextnat**: \( s \approx s |_s (\text{nextnat } s - 1) \)

\[ \text{⟨proof} \]

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

\[ \begin{align*}
\bullet & \ s |_s n \text{ and } t |_s m \text{ are similar, and} \\
\bullet & \ s |_s (n+1) \text{ is similar to either } t |_s (m+1) \text{ or to } t |_s m.
\end{align*} \]

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

**Theorem sim-step**: Assumes \( s \approx t \) shows \( \exists m. (s |_s n \approx t |_s m) \wedge ((s |_s \text{Suc } n \approx t |_s \text{Suc } m) \vee (s |_s \text{Suc } n \approx t |_s m)) \)

\[ \text{⟨proof} \]

\end{proof}

\[ \text{end} \]

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 **impI**: \( (?P \rightarrow ?Q) \rightarrow ?P \rightarrow ?Q \). However, the deduction theorem does not hold for most modal and temporal logics [6, page 95][7]. For example \( A \vdash \Box A \) holds, meaning that if \( A \) holds in any world, then it always holds. However, \( \vdash A \rightarrow \Box A \), stating that \( A \) always holds if it initially holds, is not valid.

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

```
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 and Definitions

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

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

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

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

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

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

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

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

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

```
definition RAll :: ('a ⇒ ('w::world) form) ⇒ 'w form (binder Rall 10)
```
where \(\text{unl-Rall}: (\text{Rall} \ x. \ A \ x) \ w \equiv \forall x. \ A \ x \ w\)

definition \(\text{REx} :: (\lambda a \Rightarrow (\text{name}::\text{world}) \text{ form}) \Rightarrow \text{form} \ ) \text{ (binder Rex} \ 10)\)
\[\text{where } \text{unl-REx}: (\text{REx} \ x. \ A \ x) \ w \equiv \exists x. \ A \ x \ w\)

definition \(\text{REx1} :: (\lambda a \Rightarrow (\text{name}::\text{world}) \text{ form}) \Rightarrow \text{form} \ ) \text{ (binder Rex}! \ 10)\)
\[\text{where } \text{unl-REx1}: (\text{REx!} \ x. \ A \ x) \ w \equiv \exists !x. \ A \ x \ w\)

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

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

lemmas intensional-rews\([\text{simp}] = \]
\[\text{unl-con unl-lift unl-lift2 unl-lift3 unl-lift4}
\[\text{unl-Rall unl-REx unl-REx1}\]

2.2 Concrete Syntax

nonterminal
\(\text{lift} \text{ and liftargs}\)

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

syntax
\[
\text{:: id } \Rightarrow \text{lift} \ 
\text{:: longid } \Rightarrow \text{lift} \ 
\text{:: var } \Rightarrow \text{lift} \ 
\text{:: applC } \text{[lift, cargs]} \Rightarrow \text{lift} \ (1/1 \ [1000, 1000] 999) \ 
\text{:: lift } \Rightarrow \text{lift} \ (\lambda) \ 
\text{:: lambda } \text{[idts, ’a]} \Rightarrow \text{lift} \ ((3\%/- -) \ [0, 3] 3) \ 
\text{:: constr } \text{[lift, type]} \Rightarrow \text{lift} \ ((=\text{::}) \ [4, 0] 3) \ 
\text{:: lift } \Rightarrow \text{liftargs} \ (-) \ 
\text{:: liftargs } \text{[lift, liftargs]} \Rightarrow \text{liftargs} \ (-/ -) \ 
\text{:: Valid } \text{[lift]} \Rightarrow \text{bool} \ ((\text{=} \ -) 5) \ 
\text{:: holdsAt } \text{[’a, lift]} \Rightarrow \text{bool} \ ((\text{=} \ -) \ [100, 10] 10) \]

\(\text{LIFT } \text{:: lift } \Rightarrow \text{’a} \ (\text{LIFT } -)\)

\(\text{-const } \text{:: ’a } \Rightarrow \text{lift} \ ((\#- ) \ [1000] 999) \ 
\text{-lift } \text{:: [’a, lift]} \Rightarrow \text{lift} \ ((\text{-} \ -) \ [1000] 999) \ 
\text{-lift2 } \text{:: [’a, lift, lift]} \Rightarrow \text{lift} \ ((\text{-} \ -) \ [1000] 999) \ 
\text{-lift3 } \text{:: [’a, lift, lift, lift]} \Rightarrow \text{lift} \ ((\text{-} \ -) \ [1000] 999) \ 
\text{-lift4 } \text{:: [’a, lift, lift, lift, lift]} \Rightarrow \text{lift} \ ((\text{-} \ -) \ [1000] 999)\)
-\text{liftequiv} :: \text{[lift, lift]} \Rightarrow \text{lift}
\text{((- =/) [50, 51] 50)}

-\text{liftnot} :: \text{[lift, lift]} \Rightarrow \text{lift}
\text{(infixl) \neq 50)}

-\text{lifland} :: \text{lift} \Rightarrow \text{lift}
\text{(- \& [90]) 90)}

-\text{liftor} :: \text{[lift, lift]} \Rightarrow \text{lift}
\text{(infixr) \lor 30)}

-\text{liflimp} :: \text{[lift, lift]} \Rightarrow \text{lift}
\text{(infixr) \rightarrow 25)}

-\text{lifflif} :: \text{[lift, lift, lift]} \Rightarrow \text{lift}
\text{((if (-)/ then (-)/ else (-)) 10)}

-\text{lifpluseq} :: \text{[lift, lift]} \Rightarrow \text{lift}
\text{((- +/) [66, 65] 65)}

-\text{lifminuseq} :: \text{[lift, lift]} \Rightarrow \text{lift}
\text{((- -/) [66, 65] 65)}

-\text{lifmultiseq} :: \text{[lift, lift]} \Rightarrow \text{lift}
\text{((- *./) [71, 70] 70)}

-\text{lifdivide} :: \text{[lift, lift]} \Rightarrow \text{lift}
\text{((- \div/) [71, 70] 70)}

-\text{lifmod} :: \text{[lift, lift]} \Rightarrow \text{lift}
\text{((- \mod/) [71, 70] 70)}

-\text{lifless} :: \text{[lift, lift]} \Rightarrow \text{lift}
\text{((- /</) [50, 51] 50)}

-\text{lifleq} :: \text{[lift, lift]} \Rightarrow \text{lift}
\text{((- /\le/) [50, 51] 50)}

-\text{lifmem} :: \text{[lift, lift]} \Rightarrow \text{lift}
\text{((- /\in/) [50, 51] 50)}

-\text{lifnotmem} :: \text{[lift, lift]} \Rightarrow \text{lift}
\text{((- /\notin/) [50, 51] 50)}

-\text{lifinset} :: \text{liftargs} \Rightarrow \text{lift}
\text{((\{\}) \})}

-\text{lifpair} :: \text{[lift, liftargs]} \Rightarrow \text{lift}
\text{((1/(\',/ -\')))}

-\text{lifcons} :: \text{[lift, lift]} \Rightarrow \text{lift}
\text{((- /\#/ -/) [65, 66] 65)}

-\text{lifcons} :: \text{[lift, lift]} \Rightarrow \text{lift}
\text{((- /\&/ -/) [65, 66] 65)}

\text{translations}
-\text{const} \equiv \text{CONST const}

\text{translations}
-\text{lift} \equiv \text{CONST lift}
-\text{lift2} \equiv \text{CONST lift2}
-\text{lift3} \equiv \text{CONST lift3}
-\text{lift4} \equiv \text{CONST lift4}
-\text{Valid} \equiv \text{CONST Valid}

\text{translations}
-\text{RAAll} \equiv \text{Rall} \ x \ A
-\text{REx} \equiv \text{Rex} \ x \ A
-\text{REx1} \equiv \text{Rex1} \ x \ A
\[w \models A \quad \rightarrow \quad A \ w\]

**translations**

\[-\text{liftEqu} \quad \rightleftharpoons \quad \text{lift2} (\text{=})\]
\[-\text{liftNeq} \ u \ v \quad \rightleftharpoons \quad \text{liftNot} (\text{liftEqu} \ u \ v)\]
\[-\text{liftNot} \quad \rightleftharpoons \quad \text{lift} (\text{CONST Not})\]
\[-\text{liftAnd} \quad \rightleftharpoons \quad \text{lift2} (\&\&)\]
\[-\text{liftOr} \quad \rightleftharpoons \quad \text{lift2} ((\mid))\]
\[-\text{liftImp} \quad \rightleftharpoons \quad \text{lift2} (\rightarrow)\]
\[-\text{liftIf} \quad \rightleftharpoons \quad \text{lift3} (\text{CONST If})\]
\[-\text{liftPlus} \quad \rightleftharpoons \quad \text{lift2} (+)\]
\[-\text{liftMinus} \quad \rightleftharpoons \quad \text{lift2} (-)\]
\[-\text{liftTimes} \quad \rightleftharpoons \quad \text{lift2} ((\ast))\]
\[-\text{liftDiv} \quad \rightleftharpoons \quad \text{lift2} (\div)\]
\[-\text{liftMod} \quad \rightleftharpoons \quad \text{lift2} (\text{mod})\]
\[-\text{liftLess} \quad \rightleftharpoons \quad \text{lift2} (<)\]
\[-\text{liftLeq} \quad \rightleftharpoons \quad \text{lift2} (\leq)\]
\[-\text{liftMem} \quad \rightleftharpoons \quad \text{lift2} (:)\]
\[-\text{liftNotMem} \quad \rightleftharpoons \quad \text{liftNot} (\text{liftMem} \ x \ xs)\]

**translations**

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

\[w \models \neg A \quad \leftarrow \quad \text{liftNot} \ A \ w\]
\[w \models A \wedge B \quad \leftarrow \quad \text{liftAnd} \ A \ B \ w\]
\[w \models A \lor B \quad \leftarrow \quad \text{liftOr} \ A \ B \ w\]
\[w \models u = v \quad \leftarrow \quad \text{liftEqu} \ u \ v \ w\]
\[w \models \forall x. \ A \quad \leftarrow \quad \text{RAll} \ x \ A \ w\]
\[w \models \exists x. \ A \quad \leftarrow \quad \text{REx} \ x \ A \ w\]
\[w \models \exists \! x. \ A \quad \leftarrow \quad \text{REx1} \ x \ A \ w\]

**syntax (ASCII)**

\[-\text{Valid} \quad :: \quad \text{lift} \quad \Rightarrow \quad \text{bool} \quad \leftarrow \quad (\text{\texttt{[-]} \ 5})\]
\[-\text{holdsAt} \quad :: \quad [a, \text{lift}] \quad \Rightarrow \quad \text{bool} \quad \leftarrow \quad (\text{\texttt{[-]} \ = \ 100,10] \ 10})\]
\[-\text{liftNeq} \quad :: \quad [\text{lift}, \text{lift}] \quad \Rightarrow \quad \text{lift} \quad \leftarrow \quad (\text{\texttt{[\sim]} \ / \ 50,51] \ 50})\]
\[-\text{liftNot} \quad :: \quad \text{lift} \quad \Rightarrow \quad \text{lift} \quad \leftarrow \quad (\text{\texttt{[-]} \ [90] \ 90})\]
-liftAnd :: [lift, lift] ⇒ lift
((\& / \&) [36, 35] 35)

-liftOr :: [lift, lift] ⇒ lift
((\| / \|) [31, 30] 30)

-liftImp :: [lift, lift] ⇒ lift
((\> / >) [50, 51] 50)

-liftLeq :: [lift, lift] ⇒ lift
((/- <= -) [50, 51] 50)

-liftNotMem :: [lift, lift] ⇒ lift
((/- /:) [50, 51] 50)

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

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

2.3 Lemmas and Tactics

lemma intD[dest]: ⊢ A ⇒ w |= A
⟨proof⟩

lemma intI [intro!]: assumes P1:(\∧ w. w |= A) shows ⊢ A
⟨proof⟩

Basic unlifting introduces a parameter w and applies basic rewrites, e.g ⊢ F = G becomes F w = G w and ⊢ F → G becomes F w → G w.
⟨ML⟩

lemma inteq-reflection: assumes P1: x=y shows (x ≡ y)
⟨proof⟩

lemma int-simps:
⊢ (x=x) = #True
⊢ (\¬ #True) = #False
⊢ (\¬ #False) = #True
⊢ (\¬¬ P) = P
⊢ ((\¬ P = P) = #False
⊢ (P = (\¬ P)) = #False
⊢ (P ≠ Q) = (P = (\¬ Q))
⊢ (#True=P) = P
⊢ (P=#True) = P
⊢ (#True → P) = P
⊢ (#False → P) = #True
⊢ (P → #True) = #True
⊢ (P → P) = #True
⊢ (P → \¬#False) = (\¬P)
⊢ (P → \¬#True) = (\¬P)
⊢ (P ∧ #True) = P
⊢ (#True ∧ P) = P
⊢ (P ∧ #False) = #False
⊢ (#False ∧ P) = #False
⊢ (P ∧ P) = P
⊢ (P ∧ \¬P) = #False
⊢ (\¬P ∧ P) = #False
⊢ (P ∨ #True) = #True
\[ \vdash (\# \text{True} \lor P) = \# \text{True} \]
\[ \vdash (P \lor \# \text{False}) = P \]
\[ \vdash (\# \text{False} \lor P) = P \]
\[ \vdash (P \lor P) = P \]
\[ \vdash (\neg P \lor P) = \# \text{True} \]
\[ \vdash (\forall x. P) = P \]
\[ \vdash (\exists x. P) = P \]

\section*{lemmas} intensional-simps[simp] = int-simps[THEN inteq-reflection]

\section*{(ML)}

\begin{enumerate}
\item \textbf{lemma} Not-Rall: \( \vdash (\neg(\forall x. F x)) = (\exists x. \neg F x) \)
\item \textbf{lemma} Not-Rex: \( \vdash (\neg(\exists x. F x)) = (\forall x. \neg F x) \)
\item \textbf{lemma} TrueW [simp]: \( \vdash \# \text{True} \)
\item \textbf{lemma} int-eq: \( \vdash X = Y \implies X = Y \)
\item \textbf{lemma} int-iffI:
  \begin{enumerate}
  \item \textbf{assumes} \( \vdash F \rightarrow G \) and \( \vdash G \rightarrow F \)
  \item \textbf{shows} \( \vdash F = G \)
  \end{enumerate}
\item \textbf{lemma} int-iffD1: \( \assumes h: \vdash F = G \) \( \shows \vdash F \rightarrow G \)
\item \textbf{lemma} int-iffD2: \( \assumes h: \vdash F = G \) \( \shows \vdash G \rightarrow F \)
\item \textbf{lemma} lift-imp-trans:
  \begin{enumerate}
  \item \textbf{assumes} \( \vdash A \rightarrow B \) and \( \vdash B \rightarrow C \)
  \item \textbf{shows} \( \vdash A \rightarrow C \)
  \end{enumerate}
\item \textbf{lemma} lift-imp-neg: \( \assumes A \rightarrow B \) \( \shows \vdash \neg B \rightarrow \neg A \)
\item \textbf{lemma} lift-and-com: \( \vdash (A \land B) = (B \land A) \)
\end{enumerate}
\section*{end}
3 Semantics

theory Semantics
imports Sequence Intensional
begin

This theory mechanises a shallow embedding of TLA* using the Sequence and Intensional theories. A shallow embedding represents TLA* using Isabelle/HOL predicates, while a deep embedding would represent TLA* formulas and pre-formulas as mutually inductive datatypes\(^1\). The choice of a shallow over a deep embedding is motivated by the following factors: a shallow embedding is usually less involved, and existing Isabelle theories and tools can be applied more directly to enhance automation; due to the lifting in the Intensional theory, a shallow embedding can reuse standard logical operators, whilst a deep embedding requires a different set of operators for both formulas and pre-formulas. Finally, since our target is system verification rather than proving meta-properties of TLA*, which requires a deep embedding, a shallow embedding is more fit for purpose.

3.1 Types of Formulas

To mechanise the TLA* semantics, the following type abbreviations are used:

\[ \text{type-synonym} (\alpha,\beta) \rightarrow = \alpha \rightarrow \beta \]
\[ \text{type-synonym} \ \alpha \ \text{formfun} = (\alpha,\beta) \rightarrow \beta \]
\[ \text{type-synonym} (\alpha,\beta) \rightarrow = \alpha \rightarrow \beta \]

\[ \text{type-synonym} (\alpha,\beta) \rightarrow stpred = (\alpha,\beta) \rightarrow \beta \]

\[ \text{instance} \]
\[ \text{fun} :: (\alpha,\beta) \text{ world} \langle \text{proof} \rangle \]

\[ \text{instance} \]
\[ \text{prod} :: (\alpha,\beta) \text{ world} \langle \text{proof} \rangle \]

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.

\[ \text{definition} \quad \text{always} :: (\alpha::world) \ \text{formula} \Rightarrow (\alpha) \ \text{formula} \]
\[ \text{where} \quad \text{always} \ F \equiv \lambda \ s. \ \forall \ n. \ (s \mid s \ n) \models F \]

\[ \text{definition} \quad \text{nexts} :: (\alpha::world) \ \text{formula} \Rightarrow (\alpha) \ \text{formula} \]

\(^1\)See e.g. [10] for a discussion about deep vs. shallow embeddings in Isabelle/HOL.
where \( \text{nexts} \ F \equiv \lambda \ s. \ (\text{tail} \ s) \models F \)

**definition** before :: (′a::world,′b) stfun ⇒ (′a,′b) formfun
where before \( f \equiv \lambda \ s. \ (\text{first} \ s) \models f \)

**definition** after :: (′a::world,′b) stfun ⇒ (′a,′b) formfun
where after \( f \equiv \lambda \ s. \ (\text{second} \ s) \models f \)

**definition** unch :: (′a::world,′b) stfun ⇒ ′a formula
where unch \( v \equiv \lambda \ s. \ s \models ((\text{after} \ v) \lor (\text{before} \ v)) \)

**3.2.1 Concrete Syntax**

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

**syntax**
- **always** :: \( \text{lift} \Rightarrow \text{lift} ((\square \cdot) \ [90] \ 90) \)
- **nexts** :: \( \text{lift} \Rightarrow \text{lift} ((\# \cdot) \ [90] \ 90) \)
- **action** :: \( [\text{lift},\text{lift}] \Rightarrow \text{lift} ((\square \cdot) \cdot (\cdot \cdot)) \ [20,1000] \ 90 \)
- **before** :: \( \text{lift} \Rightarrow \text{lift} ((\$ \cdot) \ [100] \ 99) \)
- **after** :: \( \text{lift} \Rightarrow \text{lift} ((\$ \cdot) \ [100] \ 99) \)
- **prime** :: \( \text{lift} \Rightarrow \text{lift} ((\cdot \cdot) \ [100] \ 99) \)
- **unch** :: \( \text{lift} \Rightarrow \text{lift} ((\text{Unchanged} \cdot) \ [100] \ 99) \)

**syntax (ASCII)**
- **always** :: \( \text{lift} \Rightarrow \text{lift} ((\square \cdot) \ [90] \ 90) \)
- **nexts** :: \( \text{lift} \Rightarrow \text{lift} ((\# \cdot) \ [90] \ 90) \)
- **action** :: \( [\text{lift},\text{lift}] \Rightarrow \text{lift} ((\square \cdot) \cdot (\cdot \cdot)) \ [20,1000] \ 90 \)

**translations**
- **always** ⇒ CONST always
- **nexts** ⇒ CONST nexts
- **action** ⇒ CONST action
- **before** ⇒ CONST before
- **after** ⇒ CONST after
- **prime** ⇒ CONST after
- **unch** ⇒ CONST unch

**3.3 Abbreviations**

Some standard temporal abbreviations, with their concrete syntax.

**definition** actrans :: (′a::world) formula ⇒ (′a,′b) stfun ⇒ ′a formula
where actrans \( P \ v \equiv \text{TEMP}(P \lor \text{unch} \ v) \)
**Definition** eventually :: (‘a::world) formula ⇒ ’a formula
where eventually F ≡ LIFT(¬□(¬F))

**Definition** angle-action :: (‘a::world) formula ⇒ (’a,’b) stfun ⇒ ’a formula
where angle-action P v ≡ LIFT(¬actrans (LIFT(¬P)) v)

**Definition** angle-actrans :: (‘a::world) formula ⇒ (’a,’b) stfun ⇒ ’a formula
where angle-actrans P v ≡ TEMP(¬actrans (LIFT(¬P))) v

**Definition** leadsto :: (‘a::world) formula ⇒ ’a formula ⇒ ’a formula
where leadsto P Q ≡ LIFT□(P −→ eventually Q)

### 3.3.1 Concrete Syntax

**Syntax (ASCII)**
- actrans :: [lift,lift] ⇒ lift ((([.2]-[.1])) [20,1000] 90)
- eventually :: lift ⇒ lift ((([>]-[<]) [90] 90)
- angle-action :: [lift,lift] ⇒ lift ((((<<<>[.1]-[.2])) [20,1000] 90)
- angle-actrans :: [lift,lift] ⇒ lift ((([>][<]) [20,1000] 90)
- leadsto :: [lift,lift] ⇒ lift ((([>][<]) [20,1000] 90)

**Syntax**
- 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 ((([>][<]) [20,1000] 90)

**Translations**
- actrans ⇒ CONST actrans
- eventually ⇒ CONST eventually
- angle-action ⇒ CONST angle-action
- angle-actrans ⇒ CONST angle-actrans
- leadsto ⇒ CONST leadsto

### 3.4 Properties of Operators

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

**Lemma** eventually-defs: (w |= □ F) = (∃ n. (w |, n) |= F)
(proof)

**Lemma** angle-action-defs: (w |= ∘(P).v) = (∃ i. ((w |, i) |= P) ∧ ((w |, i) |= v$v ≠$ v))
(proof)

**Lemma** unch-defs: (w |= Unchanged v) = (((second w) |= v) = ((first w) |= v))
(proof)

**Lemma** linalw:
assumes h1: \( a \leq b \) and h2: \( (w \mid_a a) \models \Box A \)
shows \( (w \mid_b b) \models \Box A \)

\( \langle \text{proof} \rangle \)

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.

\[ \text{definition } \text{stutinv } :: (\text{formfun} \Rightarrow \text{bool}) \]
\[ \text{where } \text{stutinv } F \equiv \forall \sigma \tau. \sigma \approx \tau \rightarrow (\sigma \models F) = (\tau \models F) \]

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: \( \Box[P]\cdot v \). To show that \( \Box[P]\cdot v \) 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 \( \bigcirc Q \), then it is not a well-formed pre-formula, thus \( \Box[P]\cdot v \) is not stuttering invariant. This weaker version of stuttering invariance has been named near stuttering invariance.

\[ \text{definition } \text{nstutinv } :: (\text{formfun} \Rightarrow \text{bool}) \]
\[ \text{where } \text{nstutinv } P \equiv \forall \sigma \tau. (\text{first } \sigma \approx \text{first } \tau) \land (\text{tail } \sigma \approx \text{tail } \tau) \rightarrow (\sigma \models P) = (\tau \models P) \]

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

\[ \text{translations } \]
\[ \text{-stutinv } = \text{CONST stutinv} \]
\[ \text{-nstutinv } = \text{CONST nstutinv} \]

Predicate \( \text{STUTINV } F \) formalises stuttering invariance for formula \( F \). That is if two sequences are similar \( s \approx t \) (equal up to stuttering) then the validity of \( F \) under both \( s \) and \( t \) are equivalent. Predicate \( \text{NSTUTINV } P \) should be read as nearly stuttering invariant – and is required for some stuttering invariance proofs.
lemma stut-inv-strictly-stronger:
assumes h: STUTINV F shows NSTUTINV F
⟨proof⟩

3.5.1 Properties of stutinv

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

theorem stut-before: STUTINV §F
⟨proof⟩

lemma nstut-after: NSTUTINV F§
⟨proof⟩
The always operator preserves stuttering invariance.

theorem stut-always: assumes H:STUTINV F shows STUTINV □F
⟨proof⟩

Assuming that formula P is nearly stuttering invariant then □[P]-v will be stuttering invariant.

lemma stut-action-lemma:
assumes H: NSTUTINV P and st: s ≈ t and P: t |= □[P]-v
shows s |= □[P]-v
⟨proof⟩

theorem stut-action: assumes H: NSTUTINV P shows STUTINV □[P]-v
⟨proof⟩
The lemmas below shows that propositional and predicate operators preserve stuttering invariance.

lemma stut-and: [STUTINV F;STUTINV G] ⇒ STUTINV (F ∧ G)
⟨proof⟩

lemma stut-or: [STUTINV F;STUTINV G] ⇒ STUTINV (F ∨ G)
⟨proof⟩

lemma stut-imp: [STUTINV F;STUTINV G] ⇒ STUTINV (F → G)
⟨proof⟩

lemma stut-eq: [STUTINV F;STUTINV G] ⇒ STUTINV (F = G)
⟨proof⟩

lemma stut-noteq: [STUTINV F;STUTINV G] ⇒ STUTINV (F ≠ G)
⟨proof⟩

lemma stut-not: STUTINV F ⇒ STUTINV (¬ F)
lemma stut-all: $(\forall x. \text{STUTINV} \ (F \ x)) \implies \text{STUTINV} \ (\forall x. F \ x)$

lemma stut-ex: $(\exists x. \text{STUTINV} \ (F \ x)) \implies \text{STUTINV} \ (\exists x. F \ x)$

lemma stut-const: $\text{STUTINV} \ # c$

lemma stut-fun1: $\text{STUTINV} \ X \implies \text{STUTINV} \ (f < X>)$

lemma stut-fun2: $[(\text{STUTINV} \ X; \text{STUTINV} \ Y)] \implies \text{STUTINV} \ (f < X, Y>)$

lemma stut-fun3: $[(\text{STUTINV} \ X; \text{STUTINV} \ Y; \text{STUTINV} \ Z)] \implies \text{STUTINV} \ (f < X, Y, Z>)$

lemma stut-fun4: $[(\text{STUTINV} \ X; \text{STUTINV} \ Y; \text{STUTINV} \ Z; \text{STUTINV} \ W)] \implies \text{STUTINV} \ (f < X, Y, Z, W>)$

lemma stut-plus: $[(\text{STUTINV} \ x; \text{STUTINV} \ y)] \implies \text{STUTINV} \ (x + y)$

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 $\text{NSTUTINV} \ (\circ F)$

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

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

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

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

lemma nstut-eq: $[(\text{NSTUTINV} \ F; \text{NSTUTINV} \ G)] \implies \text{NSTUTINV} \ (F \equiv G)$
lemma nstut-not: $\text{NSTUTINV } F \implies \text{NSTUTINV } (\neg F)$
  ⟨proof⟩

lemma nstut-noteq: $[\text{NSTUTINV } F; \text{NSTUTINV } G] \implies \text{NSTUTINV } (F \neq G)$
  ⟨proof⟩

lemma nstut-all: $(\forall x. \text{NSTUTINV } (F x)) \implies \text{NSTUTINV } (\forall x. F x)$
  ⟨proof⟩

lemma nstut-ex: $(\exists x. \text{NSTUTINV } (F x)) \implies \text{NSTUTINV } (\exists x. F x)$
  ⟨proof⟩

lemma nstut-const: $\text{NSTUTINV } \# c$
  ⟨proof⟩

lemma nstut-fun1: $\text{NSTUTINV } X \implies \text{NSTUTINV } (f <X>)$
  ⟨proof⟩

lemma nstut-fun2: $[\text{NSTUTINV } X; \text{NSTUTINV } Y] \implies \text{NSTUTINV } (f <X,Y>)$
  ⟨proof⟩

lemma nstut-fun3: $[\text{NSTUTINV } X; \text{NSTUTINV } Y; \text{NSTUTINV } Z] \implies \text{NSTUTINV } (f <X,Y,Z>)$
  ⟨proof⟩

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

lemma nstut-plus: $[\text{NSTUTINV } x; \text{NSTUTINV } y] \implies \text{NSTUTINV } (x+y)$
  ⟨proof⟩

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⟩

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

lemma nstut-actrans: $\text{NSTUTINV } P \implies \text{NSTUTINV } [P] \cdot v$
  ⟨proof⟩
lemma stut-eventually: \[ \text{STUTINV } F \implies \text{STUTINV } \Diamond F \]
\[\langle \text{proof} \rangle\]

lemma stut-leadsto: \[ [\text{STUTINV } F; \text{STUTINV } G] \implies \text{STUTINV } (F \Rightarrow G) \]
\[\langle \text{proof} \rangle\]

lemma stut-angle-action: \[ \text{NSTUTINV } P \implies \text{STUTINV } \Diamond (P \vdash-v) \]
\[\langle \text{proof} \rangle\]

lemma nstut-angle-acttrans: \[ \text{NSTUTINV } P \implies \text{NSTUTINV } \langle P \rangle \vdash-v \]
\[\langle \text{proof} \rangle\]

lemmas stutinvs = stut-before stut-always stut-action
stut-and stut-or stut-imp stut-eq stut-noteq stut-not
stut-all stut-ex stut-eventually stut-leadsto stut-angle-action stut-const
stut-fun1 stut-fun2 stut-fun3 stut-fun4

lemmas nstutinvs = nstut-after nstut-nexts nstut-acttrans
nstut-unch nstut-and nstut-or nstut-imp nstut-eq nstut-noteq nstut-not
nstut-all nstut-ex nstut-angle-acttrans stutinv-strictly-stronger
nstut-fun1 nstut-fun2 nstut-fun3 nstut-fun4 stutinvs[THEN stutinv-strictly-stronger]

lemmas bothstutinvs = stutinvs nstutinvs

end

4 Reasoning about PreFormulas

theory PreFormulas
imports Semantics
begin

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

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

syntax
-PreValid :: lift \Rightarrow bool \quad (\langle \top \rangle)

translations
-PreValid = CONST PreValid

lemma prefD[dest]: \( \models A \Rightarrow w \models A \)
\[\langle \text{proof} \rangle\]
lemma pref[I.intro]: (\wedge w. w \models A) \Rightarrow \neg A
(proof)

(ML)

lemma prefeq-reflection: assumes P1: \neg x=y shows (x \equiv y)
(proof)

lemma pref-True[simp]: \neg \# True
(proof)

lemma pref-eq: \neg X = Y \Rightarrow X = Y
(proof)

lemma pref-iffI:
  assumes \neg F \rightarrow G and \neg G \rightarrow F
  shows \neg F = G
(proof)

lemma pref-iffD1: assumes \neg F = G shows \neg F \rightarrow G
(proof)

lemma pref-iffD2: assumes \neg F = G shows \neg G \rightarrow F
(proof)

lemma unl-pref-imp:
  assumes \neg F \rightarrow G shows \wedge w. w \models F \Rightarrow w \models G
(proof)

lemma pref-imp-trans:
  assumes \neg F \rightarrow G and \neg G \rightarrow H
  shows \neg F \rightarrow H
(proof)

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: stutinv F
  shows \neg F \wedge Unchanged id \rightarrow \Box F
(proof)

lemma pre-ex-unch:
  assumes h: stutinv F
  shows \exists(v::'a::world \Rightarrow 'a). (\neg F \wedge Unchanged v \rightarrow \Box F)
(proof)
Lemma unch-pair: \(\sim \text{Unchanged}(x,y) = (\text{Unchanged} x \land \text{Unchanged} y)\)

Lemma angle-actrans-sem: \(\sim (F) v = (F \land v S \neq S)\)

4.2 Lemmas about after

Lemma after-const: \(\sim (\# c) = \# c\)

Lemma after-fun1: \(\sim f \langle x \rangle = f \langle x' \rangle\)

Lemma after-fun2: \(\sim f \langle x,y \rangle = f \langle x',y' \rangle\)

Lemma after-fun3: \(\sim f \langle x,y,z \rangle = f \langle x',y',z' \rangle\)

Lemma after-fun4: \(\sim f \langle x,y,z,zz \rangle = f \langle x',y',z',zz' \rangle\)

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

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

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

4.3 Lemmas about before

Lemma before-const: \(\vdash S(\# c) = \# c\)

Lemma before-fun1: \(\vdash S(f \langle x \rangle) = f \langle S x \rangle\)
lemma before-fun2: \[ \vdash (f\langle x,y\rangle) = f\langle x,y\rangle \]
⟨proof⟩

lemma before-fun3: \[ \vdash (f\langle x,y,z\rangle) = f\langle x,y,z\rangle \]
⟨proof⟩

lemma before-fun4: \[ \vdash (f\langle x,y,z,zz\rangle) = f\langle x,y,z,zz\rangle \]
⟨proof⟩

lemma before-forall: \[ \vdash (\forall x. P x) = (\forall x. P x) \]
⟨proof⟩

lemma before-exists: \[ \vdash (\exists x. P x) = (\exists x. P x) \]
⟨proof⟩

lemma before-exists1: \[ \vdash (\exists! x. P x) = (\exists! x. P x) \]
⟨proof⟩

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: \[ \vdash (\langle F \land G \rangle -v) = (\langle F \rangle -v \land \langle G \rangle -v) \]
⟨proof⟩

lemma angle-actrans-disj: \[ \vdash (\langle F \lor G \rangle -v) = (\langle F \rangle -v \lor \langle G \rangle -v) \]
⟨proof⟩

lemma int-eq-true: \[ \vdash P \Longrightarrow \vdash P = \# True \]
⟨proof⟩

lemma pref-eq-true: \[ \vdash P \Longrightarrow \vdash P = \# True \]
⟨proof⟩

4.5 Unlifting attributes and methods

Attribute which unlifts an intensional formula or preformula
⟨ML⟩

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⟩
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 pmp are changed to accommodate this. It is argued that while the heterogeneous version is easier to understand, the homogenous system avoids the introduction of an auxiliary provability relation. However, the price to pay is that reasoning about pre-formulas (in particular, actions) has to be performed in the scope of temporal operators such as □[P]-v, which is notationally quite heavy. We prefer here the heterogeneous approach, which exposes the pre-formulas and lets us use standard HOL rules more directly.

5.1 The Basic Axioms

**theorem fmp**: assumes ⊢ F and ⊢ F → G shows ⊢ G
 ⟨proof⟩

**theorem pmp**: assumes `F and `F → G shows `G
 ⟨proof⟩

**theorem sq**: assumes `P shows `□[P]-v
 ⟨proof⟩

**theorem pre**: assumes ⊢ F shows `F
 ⟨proof⟩

**theorem nex**: assumes h1: ⊢ F shows `○F
 ⟨proof⟩

**theorem ax0**: ⊢ # True
 ⟨proof⟩
theorem ax1:
\[ \vdash \Box F \rightarrow F \]
⟨proof⟩

theorem ax2:
\[ \vdash \Box F \rightarrow \Box[\Box F] \rightarrow \Box F \]
⟨proof⟩

theorem ax3:
assumes H: \[ \Lnot F \land \text{Unchanged } v \rightarrow \Diamond F \]
sows \[ \vdash \Box[F \rightarrow \Diamond F] \rightarrow (F \rightarrow \Box F) \]
⟨proof⟩

theorem ax4:
\[ \vdash \Box[P \rightarrow Q] \rightarrow (\Box[P] \rightarrow \Box[Q]) \]
⟨proof⟩

theorem ax5:
\[ \vdash \Box[v' \neq \$v] \rightarrow \Box v \]
⟨proof⟩

theorem pax0: \[ \Lnot \# True \]
⟨proof⟩

theorem pax1 [simp-unl]: \[ \Lnot (\Diamond \Lnot F) = (\Lnot \Diamond F) \]
⟨proof⟩

theorem pax2: \[ \Lnot (\Diamond F \rightarrow G) \rightarrow (\Diamond F \rightarrow \Diamond G) \]
⟨proof⟩

theorem pax3: \[ \Lnot \Box F \rightarrow \Diamond \Box F \]
⟨proof⟩

theorem pax4: \[ \Lnot \Box[P] \rightarrow ([P] \rightarrow \Diamond \Box[P]) \]
⟨proof⟩

theorem pax5: \[ \Lnot \Diamond \Box F \rightarrow \Box[\Diamond F] \rightarrow \Box \]
⟨proof⟩

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

theorem allT: \[ \vdash (\forall x. \Box F x) = (\Box(\forall x. F x)) \]
⟨proof⟩

theorem allActT: \[ \vdash (\forall x. \Box[F x] \rightarrow (\Box[(\forall x. F x)] \rightarrow \Box \]
⟨proof⟩

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 \( \neg F \land Unchanged v \rightarrow \Diamond F \).

**theorem alw2:**
- **assumes** \( h1: F \models \Box F \) and \( h2: \neg F \land Unchanged v \rightarrow \Diamond F \)
- **shows** \( \Box F \)
  ⟨proof⟩

Similar theorem, assuming that F is stuttering invariant.

**theorem alw3:**
- **assumes** \( h1: F \models \Box F \) and \( h2: stutinv F \)
- **shows** \( \Box F \)
  ⟨proof⟩

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 \( H1: F \models \Box F \) shows \( \Box F \)
  ⟨proof⟩

**theorem alw-valid-iff-valid:** \( (\models \Box F) = (\models F) \)
  ⟨proof⟩

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

**theorem T1[simp-unl]:** \( \Box [\Box F] = [\Box F] \)
  ⟨proof⟩

**theorem T2[simp-unl]:** \( \Box [\Box [P] - v] = [\Box [P] - v] \)
  ⟨proof⟩

**theorem T3[simp-unl]:** \( \Box [\Box [P] - v] = [\Box [P] - v] \)
  ⟨proof⟩

**theorem M2:**
- **assumes** \( h: \neg F \rightarrow G \)
- **shows** \( \Box [F] - v \rightarrow [\Box [G] - v] \)
  ⟨proof⟩

**theorem N1:**
- **assumes** \( h: F \rightarrow G \)
- **shows** \( \neg \Diamond F \rightarrow \Diamond G \)
theorem T4: \( \vdash \Box [P] \dashv \rightarrow \Box [P] \dashv \rightarrow \)

theorem T5: \( \vdash \Box [P] \dashv \rightarrow \Box [P] \dashv \rightarrow \)

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

theorem T7:  
assumes h1: \( \vdash \Box [P] \dashv \rightarrow \) and h2: \( \vdash [P] \rightarrow [Q] \dashv \rightarrow \)  
shows \( \vdash \Box [Q] \dashv \rightarrow \)

theorem T8: \( \vdash \Box [\Box F] \dashv \rightarrow \Box [\Box F] \dashv \rightarrow \)

lemma T9: \( \vdash \Box [\Box F] \dashv \rightarrow \Box [\Box F] \dashv \rightarrow \)

theorem H1:  
assumes h1: \( \vdash \Box [P] \dashv \rightarrow \) and h2: \( \vdash \Box [P] \rightarrow Q] \dashv \rightarrow \)  
shows \( \vdash \Box [Q] \dashv \rightarrow \)

theorem H2:  
assumes h1: \( \vdash \Box [P] \dashv \rightarrow \) and h2: \( \vdash \Box [Q] \rightarrow R] \dashv \rightarrow \)  
shows \( \vdash \Box [P] \rightarrow R] \dashv \rightarrow \)

theorem H3: \( \vdash \Box [P] \dashv \rightarrow \Box [P] \dashv \rightarrow \)

theorem H4: \( \vdash \Box [P] \dashv \rightarrow \Box [P] \dashv \rightarrow \)

5.3 Some other useful derived theorems

theorem P1: \( \vdash \Box F \rightarrow \Box F \)

theorem P2: \( \vdash \Box F \rightarrow \Box F \wedge \Box F \)
We now derive Lamport’s 6 simple temporal logic rules (STL1)-(STL6) [5].

Firstly, STL1 is the same as \( \vdash ?F \implies \Box ?F \) derived above.
STL2 and STL3 have also already been derived.

lemmas $STL2 = ax1$

lemmas $STL3 = T1$

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

**Theorem STL4-2:**

**Assumes** $h1: \vdash F \rightarrow G$ and $h2: \neg G \land Unchanged \rightarrow \Box G$

**Shows** $\vdash \Box F \rightarrow \Box G$

**Proof**

**Lemma STL4-3:**

**Assumes** $h1: \vdash F \rightarrow G$ and $h2: STUTINV G$

**Shows** $\vdash \Box F \rightarrow \Box G$

**Proof**

Of course, the original rule can be derived semantically

**Lemma STL4:**

**Assumes** $h: \vdash F \rightarrow G$

**Shows** $\vdash \forall \Box F \rightarrow \forall \Box G$

**Proof**

Dual rule for $\forall$

**Lemma STL4-eve:**

**Assumes** $h: \vdash F \rightarrow G$

**Shows** $\vdash \forall F \rightarrow \forall G$

**Proof**

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

**Theorem STL5-2:**

**Assumes** $h1: \neg F \land Unchanged \rightarrow \Box F$

**And** $h2: \neg G \land Unchanged \rightarrow \Box G$

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

**Proof**

**Theorem STL5-21:**

**Assumes** $h1: stuvinv F$ and $h2: stuinv G$

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

**Proof**

We also derive STL5 semantically.

**Lemma STL5:**

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

**Proof**

Elimination rule corresponding to STL5 in unlifted form.

**Lemma box-conjE:**

**Assumes** $s \models \Box F$ and $s \models \Box G$ and $s \models \Box (F \land G) \implies P$

**Shows** $P$

**Proof**

34
lemma box-thin:
assumes h1: s |= □F and h2: PROP W
shows PROP W
⟨proof⟩

Finally, we derive STL6 (only semantically)

lemma STL6: ⊢ ◇□(F ∧ G) = (◇□F ∧ ◇□G)
⟨proof⟩

lemma MM0: ⊢ □(F → G) → □F → □G
⟨proof⟩

lemma MM1: assumes h: ⊢ F = G shows ⊢ □F = □G
⟨proof⟩

theorem MM2: ⊢ □A ∧ □(B → C) → □(A ∧ B → C)
⟨proof⟩

theorem MM3: ⊢ □¬A → □(A ∧ B → C)
⟨proof⟩

theorem MM4[simp-unl]: ⊢ □#F = #F
⟨proof⟩

theorem MM4b[simp-unl]: ⊢ □¬#F = ¬#F
⟨proof⟩

theorem MM5: ⊢ □F ∨ □G → □(F ∨ G)
⟨proof⟩

theorem MM6: ⊢ □F ∨ □G → □(□F ∨ □G)
⟨proof⟩

lemma MM10:
assumes h: ∼ F = G shows ⊢ □[F]-v = □[G]-v
⟨proof⟩

lemma MM9:
assumes h: ⊢ F = G shows ⊢ □[F]-v = □[G]-v
⟨proof⟩

theorem MM11: ⊢ □[¬(P ∧ Q)]-v → □[P]-v → □[P ∧ ¬Q]-v
⟨proof⟩

theorem MM12[simp-unl]: ⊢ □[□[P]-v]-v = □[P]-v
⟨proof⟩
5.4 Theorems about the eventually operator
— rules to push negation inside modal operators, sometimes useful for rewriting

**Theorem dualization:**
\[ \vdash \neg \square F = \Diamond \neg F \]
\[ \vdash \neg \Diamond F = \square \neg F \]
\[ \vdash \neg \Diamond [A]v = \Diamond \langle \neg A \rangle v \]
\[ \vdash \neg \Diamond \langle A \rangle v = \square [\neg A]v \]

(\textit{proof})

**Lemmas**
\[ \text{dualization-rew = dualization[int-rewrite]} \]
\[ \text{dualization-unl = dualization[unlifted]} \]

**Theorem E1:**
\[ \vdash \Diamond (F \lor G) = (\Diamond F \lor \Diamond G) \]
(\textit{proof})

**Theorem E3:**
\[ \vdash F \rightarrow \Diamond F \]
(\textit{proof})

**Theorem E4:**
\[ \vdash \square F \rightarrow \Diamond F \]
(\textit{proof})

**Theorem E5:**
\[ \vdash \square F \rightarrow \square \Diamond F \]
(\textit{proof})

**Theorem E6:**
\[ \vdash \square F \rightarrow \Diamond \square F \]
(\textit{proof})

**Theorem E7:**
\[ \text{assumes } h : \neg F \land \text{Unchanged } v \rightarrow \Diamond \neg F \]
\[ \text{shows } \neg \Diamond F \rightarrow F \lor \Diamond \Diamond F \]
(\textit{proof})

**Theorem E8:**
\[ \vdash \Diamond (F \rightarrow G) \rightarrow \square F \rightarrow \Diamond G \]
(\textit{proof})

**Theorem E9:**
\[ \vdash \square (F \rightarrow G) \rightarrow \Diamond F \rightarrow \Diamond G \]
(\textit{proof})

**Theorem E10**\[\text{[simp-unl]}\]:
\[ \vdash \Diamond \Diamond F = \Diamond F \]
(\textit{proof})

**Theorem E22:**
\[ \text{assumes } h : F = G \]
\[ \text{shows } \vdash \Diamond F = \Diamond G \]
(\textit{proof})

**Theorem E15**\[\text{[simp-unl]}\]:
\[ \vdash \Diamond \# F = \# F \]
(\textit{proof})
theorem E15b[simp-unl]: \( \vdash \Diamond \neg \# F = \neg \# F \)  
(proof)

theorem E16: \( \vdash \Diamond \square F \rightarrow \Diamond F \)  
(proof)

An action version of STL6

lemma STL6-act: \( \vdash (\square[F]-v \land \square[G]-w) = (\Diamond[F]-v \land \Diamond[G]-w) \)  
(proof)

lemma SE1: \( \vdash \square F \land \Diamond G \rightarrow \Diamond(F \land G) \)  
(proof)

lemma SE2: \( \vdash \square F \land \Diamond G \rightarrow \Diamond(F \land G) \)  
(proof)

lemma SE3: \( \vdash \square F \land \Diamond G \rightarrow \Diamond(G \land F) \)  
(proof)

lemma SE4:  
assumes \( h1: s \models \square F \) \text{ and } \( h2: s \models \Diamond G \) \text{ and } \( h3: s \models \square F \land G \rightarrow H \)  
shows \( s \models \Diamond H \)  
(proof)

theorem E17: \( \vdash \square \Diamond \square F \rightarrow \square \Diamond F \)  
(proof)

theorem E18: \( \vdash \square \Diamond \square F \rightarrow \Diamond \square F \)  
(proof)

theorem E19: \( \vdash \Diamond \square F \rightarrow \square \Diamond \square F \)  
(proof)

theorem E20: \( \vdash \Diamond \square F \rightarrow \square \Diamond F \)  
(proof)

theorem E21[simp-unl]: \( \vdash \square \Diamond F = \Diamond \square F \)  
(proof)

theorem E27[simp-unl]: \( \vdash \Diamond \square F = \square \Diamond F \)  
(proof)

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

lemma E23: \( \vdash \sim \Diamond F \rightarrow \Diamond F \)  
(proof)

lemma E24: \( \vdash \Diamond Q \rightarrow \square[\Diamond Q]-v \)  
(proof)
lemma E25: \( \vdash \lozenge (A \cdot v) \rightarrow \lozenge A \)

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

lemma allBox: \( s \models \Box (\forall x. F x) \) = \( (\forall x. s \models \Box (F x)) \)

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

lemma E30: assumes h1: \( \vdash F \rightarrow \Box F \) and h2: \( \vdash \lozenge F \) shows \( \vdash \lozenge \Box F \)

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

lemma allActBox: \( s \models \Box (\forall x. F x) \cdot v \) = \( (\forall x. s \models \Box (F x) \cdot v) \)

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

theorem exActE: \( \vdash (\exists x. \lozenge (F x) \cdot v) = \lozenge (\exists x. (F x) \cdot v) \)

5.5 Theorems about the leadsto operator

theorem LT1: \( \vdash F \leadsto F \)

theorem LT2: assumes h: \( \vdash F \rightarrow G \) shows \( \vdash F \rightarrow \lozenge G \)

theorem LT3: assumes h: \( \vdash F \rightarrow G \) shows \( \vdash F \leadsto G \)

theorem LT4: \( \vdash F \rightarrow (F \leadsto G) \rightarrow \lozenge G \)

theorem LT5: \( \vdash \Box (F \rightarrow \lozenge G) \rightarrow \lozenge F \rightarrow \lozenge G \)

theorem LT6: \( \vdash \lozenge F \rightarrow (F \leadsto G) \rightarrow \lozenge G \)
\(\{\text{proof}\}\)

**Theorem LT9 [simp-unl]:** \(\vdash \Box (F \sim G) = (F \sim G)\)

\(\{\text{proof}\}\)

**Theorem LT7:** \(\vdash \Box \lozenge F \rightarrow (F \sim G) \rightarrow \Box \lozenge G\)

\(\{\text{proof}\}\)

**Theorem LT8:** \(\vdash \Box \lozenge G \rightarrow (F \sim G)\)

\(\{\text{proof}\}\)

**Theorem LT13:** \(\vdash (F \sim G) \rightarrow (G \sim H) \rightarrow (F \sim H)\)

\(\{\text{proof}\}\)

**Theorem LT11:** \(\vdash (F \sim G) \rightarrow (F \sim (G \lor H))\)

\(\{\text{proof}\}\)

**Theorem LT12:** \(\vdash (F \sim H) \rightarrow (F \sim (G \lor H))\)

\(\{\text{proof}\}\)

**Theorem LT14:** \(\vdash ((F \lor G) \sim H) \rightarrow (F \sim H)\)

\(\{\text{proof}\}\)

**Theorem LT15:** \(\vdash ((F \lor G) \sim H) \rightarrow (G \sim H)\)

\(\{\text{proof}\}\)

**Theorem LT16:** \(\vdash (F \sim H) \rightarrow (G \sim H) \rightarrow ((F \lor G) \sim H)\)

\(\{\text{proof}\}\)

**Theorem LT17:** \(\vdash ((F \lor G) \sim H) = ((F \sim H) \land (G \sim H))\)

\(\{\text{proof}\}\)

**Theorem LT10:**

**Assumes** \(h: \vdash (F \land \neg G) \sim G\)

**Shows** \(\vdash F \sim G\)

\(\{\text{proof}\}\)

**Theorem LT18:** \(\vdash (A \sim (B \lor C)) \rightarrow (B \sim D) \rightarrow (C \sim D) \rightarrow (A \sim D)\)

\(\{\text{proof}\}\)

**Theorem LT19:** \(\vdash (A \sim (D \lor B)) \rightarrow (B \sim D) \rightarrow (A \sim D)\)

\(\{\text{proof}\}\)

**Theorem LT20:** \(\vdash (A \sim (B \lor D)) \rightarrow (B \sim D) \rightarrow (A \sim D)\)

\(\{\text{proof}\}\)

**Theorem LT21:** \(\vdash ((\exists x. F x) \sim G) = (\forall x. (F x \sim G))\)

\(\{\text{proof}\}\)
\textbf{theorem} LT22: $\vdash (F \leadsto (G \lor H)) \rightarrow \Box \neg G \rightarrow (F \leadsto H)$

\textit{(proof)}

\textbf{lemma} LT23: $\vdash (P \rightarrow \Diamond Q) \rightarrow (P \rightarrow \Diamond Q)$

\textit{(proof)}

\textbf{theorem} LT24: $\vdash \Box I \rightarrow ((P \land I) \leadsto Q) \rightarrow P \leadsto Q$

\textit{(proof)}

\textbf{theorem} LT25\[simp-unl]: $\vdash (F \leadsto \# \text{False}) = \Box \neg F$

\textit{(proof)}

\textbf{lemma} LT28:

\begin{itemize}
  \item assumes $h$: $\vdash \neg P \rightarrow \Diamond P \lor \Diamond Q$
  \item shows $\vdash \neg (P \rightarrow \Diamond P) \lor \Diamond Q$
\end{itemize}

\textit{(proof)}

\textbf{lemma} LT29:

\begin{itemize}
  \item assumes $h1$: $\vdash \neg P \rightarrow \Diamond P \lor \Diamond Q$ and $h2$: $\vdash \neg P \land \text{Unchanged } v \rightarrow \Diamond P$
  \item shows $\vdash P \rightarrow \Box P \lor \Diamond Q$
\end{itemize}

\textit{(proof)}

\textbf{lemma} LT30:

\begin{itemize}
  \item assumes $h$: $\vdash \neg P \land \neg N \rightarrow \Diamond P \lor \Diamond Q$
  \item shows $\vdash \neg N \rightarrow \Box P \lor \Diamond Q$
\end{itemize}

\textit{(proof)}

\textbf{lemma} LT31:

\begin{itemize}
  \item assumes $h1$: $\vdash \neg P \land \neg N \rightarrow \Diamond P \lor \Diamond Q$ and $h2$: $\vdash \neg P \land \text{Unchanged } v \rightarrow \Diamond P$
  \item shows $\vdash \Box N \rightarrow P \rightarrow \Diamond P \lor \Diamond Q$
\end{itemize}

\textit{(proof)}

\textbf{lemma} LT33: $\vdash ((\#P \land F) \leadsto G) = (\#P \leadsto (F \leadsto G))$

\textit{(proof)}

\textbf{lemma} AA1: $\vdash \Box[\#\text{False}]\rightarrow v \rightarrow \neg \Diamond (Q)\rightarrow v$

\textit{(proof)}

\textbf{lemma} AA2: $\vdash \Box[P] \rightarrow v \land \Diamond (Q) \rightarrow v \rightarrow \Diamond (P \land Q) \rightarrow v$

\textit{(proof)}

\textbf{lemma} AA3: $\vdash \Box P \land \Box[P \rightarrow Q] \rightarrow v \land \Diamond (A) \rightarrow v \rightarrow \Diamond Q$

\textit{(proof)}

\textbf{lemma} AA4: $\vdash \Diamond (A) \rightarrow w \rightarrow \Diamond (A) \rightarrow w$

\textit{(proof)}

\textbf{lemma} AA7: assumes $h$: $\vdash \neg F \rightarrow G$ shows $\vdash \Diamond (F) \rightarrow \Diamond (G) \rightarrow v$

\textit{(proof)}

40
lemma AA6: \(\vdash \Box P \rightarrow Q\) -v \(\land \Diamond (P)\) -v \(\rightarrow \Diamond (Q)\) -v
(proof)

lemma AA8: \(\vdash \Box P \land \Diamond (A)\) -v \(\rightarrow \Diamond (\Box P \land A)\) -v
(proof)

lemma AA9: \(\vdash \Box P \land \Diamond (A)\) -v \(\rightarrow \Diamond (\Box P \land A)\) -v
(proof)

lemma AA10: \(\vdash \neg (\Box P \land \Diamond (\neg P)\) -v
(proof)

lemma AA11: \(\vdash \neg (v \neq v)\) -v
(proof)

lemma AA15: \(\vdash \Diamond (P \land Q)\) -v \(\rightarrow \Diamond (P)\) -v
(proof)

lemma AA16: \(\vdash \Diamond (P \lor Q)\) -v = \((\Diamond P\) -v \(\lor \Diamond (Q)\) -v)
(proof)

lemma AA17: \(\vdash \Diamond (P \land A)\) -v \(\rightarrow \Diamond (P \land A)\) -v
(proof)

lemma AA19: \(\vdash \Box P \land \Diamond (A)\) -v \(\rightarrow \Diamond (P \land A)\) -v
(proof)

lemma AA20:
assumes h1: \(\neg P \rightarrow \Box P \land A\)
and h2: \(\neg P \land A \rightarrow \Box P\)
and h3: \(\neg P \land Unchanged \) -v \(\rightarrow \Box P\)
shows \(\vdash \Box (\Box P \rightarrow \Diamond (A)\) -v \(\rightarrow (P \rightarrow Q)\)
(proof)

lemma AA21: \(\neg (\Diamond (\neg F)\) -v \(\rightarrow \Box (\neg F)\)
(proof)

theorem AA24[simp-unl]: \(\vdash \Diamond (\neg F)\) -f = \(\Diamond (P)\) -f
(proof)

lemma AA22:
assumes h1: \(\neg P \land N \rightarrow \Box P \land Q\)

41
5.6 Lemmas about the next operator

lemma N2: assumes F = G shows ∼#F = #G

(proof)

lemmas next-and = T8

(proof)

lemma next-or:

(⟨proof⟩)

and h2: P ∧ N → (Q ∧ A)−v−→ #Q

and h3: ∼P ∧ Unchanged w −→ #P

shows ⊢ □N ∧ □♦⟨A⟩−v−→ (P ⇝ Q)

⟨proof⟩

lemma AA23:

assumes h2: ∼P ∧ N −→ #P ∨ #Q

and h3: ∼P ∧ Unchanged w −→ #P

shows ⊢ □N ∧ □♦⟨A⟩−v−→ (P ⇝ Q)

⟨proof⟩

lemma AA25:

assumes h: ∼⟨P⟩−v−→⟨Q⟩−w

shows ⊢ ♦⟨P⟩−v−→ ♦⟨Q⟩−w

⟨proof⟩

lemma AA26:

assumes h: ∼⟨A⟩−v= ⟨B⟩−w

shows ⊢ ♦⟨A⟩−v= ♦⟨B⟩−w

⟨proof⟩

theorem AA28 [simp-unl]: ⊢ ♦♦⟨A⟩−v= ♦⟨A⟩−v

⟨proof⟩

theorem AA29:

⊢ □N −v ∧ □♦⟨A⟩−v−→ □♦⟨N ∧ A⟩−v

⟨proof⟩

theorem AA30 [simp-unl]: ⊢ ♦⟨♦⟨P⟩−f⟩−f= ♦⟨P⟩−f

⟨proof⟩

theorem AA31:

⊢ ♦⟨#F⟩−v−→ ♦F

⟨proof⟩

lemma AA32 [simp-unl]

⊢ □♦□A−v= ♦□A−v

⟨proof⟩

lemma AA33 [simp-unl]

⊢ ♦□♦A−v= □♦A−v

⟨proof⟩

42
lemma next-imp: |¬ ◦ (F → G) = (◦ F → ◦ G)
⟨proof⟩

lemmas next-not = pax1

lemma next-eq: |¬ ◦ (F = G) = (◦ F = ◦ G)
⟨proof⟩

lemma next-noteq: |¬ ◦ (F ≠ G) = (◦ F ≠ ◦ G)
⟨proof⟩

lemma next-const[simp-unl]: |¬ ◦ #P = #P
⟨proof⟩

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

lemma next-fun1: |¬ ◦ f <x> = f <◦x>
⟨proof⟩

lemma next-fun2: |¬ ◦ f <x,y> = f <◦x,◦y>
⟨proof⟩

lemma next-fun3: |¬ ◦ f <x,y,z> = f <◦x,◦y,◦z>
⟨proof⟩

lemma next-fun4: |¬ ◦ f <x,y,z,zz> = f <◦x,◦y,◦z,◦zz>
⟨proof⟩

lemma next-forall: |¬ ◦ (∀ x. P x) = (∀ x. ◦ P x)
⟨proof⟩

lemma next-exists: |¬ ◦ (∃ x. P x) = (∃ x. ◦ P x)
⟨proof⟩

lemma next-exists1: |¬ ◦ (∃! x. P x) = (∃! x. ◦ P x)
⟨proof⟩

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

43
lemma t1: \( \neg \Diamond x = x \)
\( \langle \text{proof} \rangle \)

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

lemma next-eventually:
  assumes h: stutinv F
  shows \( \neg \Diamond F \longrightarrow \neg F \longrightarrow \Diamond \Diamond F \)
\( \langle \text{proof} \rangle \)

lemma next-action: \( \neg \Box [P] \cdot v \longrightarrow \Diamond \Diamond [P] \cdot v \)
\( \langle \text{proof} \rangle \)

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 \( \neg \) is used where appropriate.

theorem TLA1:
  assumes H: \( \neg P \land \text{Unchanged } f \longrightarrow \Diamond P \)
  shows \( \Diamond P = (P \land \Box [P \longrightarrow \Diamond P] \cdot f) \)
\( \langle \text{proof} \rangle \)

theorem TLA2:
  assumes h1: \( \vdash P \longrightarrow Q \)
  and h2: \( \neg P \land \Diamond P \land [A] \cdot f \longrightarrow [B] \cdot g \)
  shows \( \vdash \Box P \land \Box [A] \cdot f \longrightarrow \Box Q \land \Box [B] \cdot g \)
\( \langle \text{proof} \rangle \)

theorem INV1:
  assumes H: \( \neg I \land [N] \cdot f \longrightarrow \Diamond I \)
  shows \( \vdash I \land \Box [N] \cdot f \longrightarrow \Box I \)
\( \langle \text{proof} \rangle \)

theorem INV2: \( \vdash \Box I \longrightarrow \Box [N] \cdot f = \Box [N \land I \land \Diamond I] \cdot f \)
\( \langle \text{proof} \rangle \)

lemma R1:
  assumes H: \( \neg \text{Unchanged } w \longrightarrow \text{Unchanged } v \)
  shows \( \vdash \Box [F] \cdot w \longrightarrow \Box [F] \cdot v \)
\( \langle \text{proof} \rangle \)

theorem invmono:
  assumes h1: \( \vdash I \longrightarrow P \)
  and h2: \( \neg P \land [N] \cdot f \longrightarrow \Diamond P \)
  shows \( \vdash I \land \Box [N] \cdot f \longrightarrow \Box P \)
\( \langle \text{proof} \rangle \)
theorem preimpsplit:
assumes $\neg I \land N \rightarrow Q$
and $\neg I \land \text{Unchanged } v \rightarrow Q$
shows $\neg I \land [N]v \rightarrow Q$
(proof)

theorem refinement1:
assumes $h1: \vdash P \rightarrow Q$
and $h2: \vdash \neg I \land \#I \land [A]f \rightarrow [B]g$
shows $\vdash P \land \Box I \land \Box [A]f \rightarrow Q \land \Box [B]g$
(proof)

theorem inv-join:
assumes $\vdash P \rightarrow \Box Q$ and $\vdash P \rightarrow \Box R$
shows $\vdash P \rightarrow \Box (Q \land R)$
(proof)

lemma inv-cases: $\vdash \Box (A \rightarrow B) \land \Box (\neg A \rightarrow B) \rightarrow \Box B$
(proof)

end

6 Liveness

theory Liveness
imports Rules
begin

This theory derives proof rules for liveness properties.

definition enabled :: 'a formula $\Rightarrow$ 'a formula
where enabled $F \equiv \lambda s. \exists t. ((\text{first } s) \#\# t) \mid F$

syntax -Enabled :: lift $\Rightarrow$ lift ((Enabled -) [90] 90)

translations -Enabled $\Rightarrow$ CONST enabled

definition WeakF :: ('a::world) formula $\Rightarrow$ ('a,'b) stfun $\Rightarrow$ 'a formula
where WeakF $F v \equiv \text{TEMP } \Box\Box\text{Enabled } (F)\cdot v \rightarrow \Box\Box(F)\cdot v$

definition StrongF :: ('a::world) formula $\Rightarrow$ ('a,'b) stfun $\Rightarrow$ 'a formula
where StrongF $F v \equiv \text{TEMP } \Box\Box\text{Enabled } (F)\cdot v \rightarrow \Box\Box(F)\cdot v$

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
defined 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
- \(WF::[\text{lift, lift}] \Rightarrow \text{lift} ((WF)'(-)'(-)) [20,1000] 90\)
- \(SF::[\text{lift, lift}] \Rightarrow \text{lift} ((SF)'(-)'(-)) [20,1000] 90\)
- \(WFsp::[\text{lift, lift}] \Rightarrow \text{lift} ((WF)'(-)'(-)) [20,1000] 90\)
- \(SFsp::[\text{lift, lift}] \Rightarrow \text{lift} ((SF)'(-)'(-)) [20,1000] 90\)

translations
- \(WF \equiv \text{CONST WeakF}\)
- \(SF \equiv \text{CONST StrongF}\)
- \(WFsp \rightarrow \text{CONST WeakF}\)
- \(SFsp \rightarrow \text{CONST StrongF}\)

6.1 Properties of -Enabled

theorem enabledI: \(\vdash F \longrightarrow \text{Enabled} F\)
\(\langle \text{proof} \rangle\)

theorem enabledE:
\(\text{assumes } s \models \text{Enabled} F \text{ and } \land u. (\text{first} s \#\# u) \models F \implies Q\)
\(\text{shows } Q\)
\(\langle \text{proof} \rangle\)

lemma enabled-mono:
\(\text{assumes } w \models \text{Enabled} F \text{ and } \vdash F \rightarrow G\)
\(\text{shows } w \models \text{Enabled} G\)
\(\langle \text{proof} \rangle\)

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

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

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

lemma Enabled-conj2: \(\vdash \text{Enabled} (G \land F) \rightarrow \text{Enabled} F\)
\(\langle \text{proof} \rangle\)

lemma Enabled-disjD: \(\vdash \text{Enabled} (F \lor G) \rightarrow \text{Enabled} F \lor \text{Enabled} G\)
\(\langle \text{proof} \rangle\)

lemma Enabled-disj: \(\vdash \text{Enabled} (F \lor G) = (\text{Enabled} F \lor \text{Enabled} G)\)
\(\langle \text{proof} \rangle\)

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

lemma Enabled-ex: \(\vdash \text{Enabled} (\exists x. F x) = (\exists x. \text{Enabled} (F x))\)
6.2 Fairness Properties

lemma WF-alt: \( \vdash \text{WF} (A)-v = (\square \Diamond \neg \text{Enabled} \langle A \rangle-v \lor \square \Diamond (A)-v) \)

(\text{proof})

lemma SF-alt: \( \vdash \text{SF} (A)-v = (\Diamond \square \neg \text{Enabled} \langle A \rangle-v \lor \Diamond \square (A)-v) \)

(\text{proof})

lemma alwaysWFI: \( \vdash \text{WF} (A)-v \rightarrow \square \text{WF} (A)-v \)

(\text{proof})

theorem WF-always[simp-und]: \( \vdash \square \text{WF} (A)-v = \text{WF} (A)-v \)

(\text{proof})

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

(\text{proof})

lemma alwaysSFI: \( \vdash \text{SF} (A)-v \rightarrow \square \text{SF} (A)-v \)

(\text{proof})

theorem SF-always[simp-und]: \( \vdash \square \text{SF} (A)-v = \text{SF} (A)-v \)

(\text{proof})

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

(\text{proof})

theorem SF-imp-WF: \( \vdash \text{SF} (A)-v \rightarrow \text{WF} (A)-v \)

(\text{proof})

lemma enabled-WFSF: \( \vdash \square \text{Enabled} \langle F \rangle-v \rightarrow (\text{WF} (F)-v = \text{SF} (F)-v) \)

(\text{proof})

theorem WF1-general:
\text{assumes} h1: \( \text{\sim} P \land [N].v \rightarrow \circ P \lor \circ Q \)
\text{and} h2: \( \text{\sim} P \land [A].v \rightarrow \circ Q \)
\text{and} h3: \( \vdash P \land [N].v \rightarrow \text{Enabled} \langle A \rangle-v \)
\text{and} h4: \( \text{\sim} P \land \text{Unchanged} v \rightarrow \circ P \)
\text{shows} \( \vdash N \land [N] \land \text{WF} (A)-v \rightarrow (P \sim Q) \)

(\text{proof})

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

theorem WF1:
\text{assumes} h1: \( \text{\sim} P \land [N].v \rightarrow \circ P \lor \circ Q \)
\text{and} h2: \( \text{\sim} P \land (N \land [A].v \rightarrow \circ Q \)
\text{and} h3: \( \vdash P \land (N \land [A].v \rightarrow \text{Enabled} \langle A \rangle-v \)
\text{and} h4: \( \text{\sim} P \land \text{Unchanged} v \rightarrow \circ P \)
\text{shows} \( \vdash [N] \land \text{WF} (A)-v \rightarrow (P \sim Q) \)
The corresponding rule for strong fairness has an additional hypothesis □F, which is typically a conjunction of other fairness properties used to prove that the helpful action eventually becomes enabled.

**theorem SF1-general:**
assumes

\[ h_1: \neg P \land N \rightarrow \circ P \lor \circ Q \]

and

\[ h_2: \neg P \land (N \land A) \rightarrow \circ Q \]

and

\[ h_3: \vdash \square P \land \square N \land \square F \rightarrow \diamond \text{Enabled} \langle A \rangle \]

and

\[ h_4: \neg P \land \text{Unchanged} \rightarrow \circ P \]

shows

\[ \vdash \square N \land SF(A) \land \square F \rightarrow (P \Rightarrow Q) \]

**⟨proof⟩**

Lamport proposes the following rule as an introduction rule for \(WF\) formulas.

**theorem WF2:**
assumes

\[ h_1: \neg (N \land B) \rightarrow \langle M \rangle \]

and

\[ h_2: \neg P \land \circ P \land (N \land A) \rightarrow \langle B \rangle \]

and

\[ h_3: \vdash P \land \text{Enabled} \langle M \rangle \rightarrow \text{Enabled} \langle A \rangle \]

and

\[ h_4: \vdash \square (N \land \neg B) \land WF(A) \land \square F \land \diamond \text{Enabled} \langle M \rangle \rightarrow \diamond \square P \]

shows

\[ \vdash \square (N \land \neg B) \land WF(A) \land \square F \rightarrow WF(M) \]

**⟨proof⟩**

Lamport proposes an analogous theorem for introducing strong fairness, and its proof is very similar, in fact, it was obtained by copy and paste, with minimal modifications.

**theorem SF2:**
assumes

\[ h_1: \neg (N \land B) \rightarrow \langle M \rangle \]

and

\[ h_2: \neg P \land \circ P \land (N \land A) \rightarrow \langle B \rangle \]

and

\[ h_3: \vdash P \land \text{Enabled} \langle M \rangle \rightarrow \text{Enabled} \langle A \rangle \]

and

\[ h_4: \vdash \square (N \land \neg B) \land SF(A) \land \square F \land \diamond \text{Enabled} \langle M \rangle \rightarrow \diamond \square P \]

shows

\[ \vdash \square (N \land \neg B) \land SF(A) \land \square F \rightarrow SF(M) \]

**⟨proof⟩**

This is the lattice rule from TLA

**theorem wf-leadsto:**
assumes

\[ h_1: \text{wf r} \]

and

\[ h_2: \forall x. \vdash F x \rightarrow (G \lor (\exists y. \#((y, x) \in r) \land F y)) \]

shows

\[ \vdash F x \rightarrow G \]

**⟨proof⟩**
6.3 Stuttering Invariance

\textbf{theorem} \textit{stut-Enabled: \text{STUTINV Enabled (F)}-v}
(\text{proof})

\textbf{theorem} \textit{stut-WF: \text{NSTUTINV F} \implies \text{STUTINV WF(F)}-v}
(\text{proof})

\textbf{theorem} \textit{stut-SF: \text{NSTUTINV F} \implies \text{STUTINV SF(F)}-v}
(\text{proof})

\textbf{lemmas} \textit{livestutinv = stat-WF stat-SF stat-Enabled}

end

7 Representing state in TLA*

t\textit{heory State}
\textbf{imports Liveness}
\textbf{begin}

We adopt the hidden state approach, as used in the existing Isabelle/HOL TLA embedding [7]. This approach is also used in [3]. Here, a state space is defined by its projections, and everything else is unknown. Thus, a variable is a projection of the state space, and has the same type as a state function. Moreover, strong typing is achieved, since the projection function may have any result type. To achieve this, the state space is represented by an undefined type, which is an instance of the \textit{world} class to enable use with the \textit{Intensional} theory.

\textit{typedec state}

\textbf{instance} \textit{state :: world (proof)}

\textbf{type-synonym} \textit{\"a statefun = (state,\"a) stfun}
\textbf{type-synonym} \textit{statepred = bool statefun}
\textbf{type-synonym} \textit{\"a tempfun = (state,\"a) formfun}
\textbf{type-synonym} \textit{temporal = state formula}

Formalizing type state would require formulas to be tagged with their underlying state space and would result in a system that is much harder to use. (Unlike Hoare logic or Unity, TLA has quantification over state variables, and therefore one usually works with different state spaces within a single specification.) Instead, state is just an anonymous type whose only purpose is to provide Skolem constants. Moreover, we do not define a type of state variables separate from that of arbitrary state functions, again in order to simplify the definition of flexible quantification later on. Nevertheless, we need to distinguish state variables, mainly to define the enabledness of ac-
tions. The user identifies (tuples of) “base” state variables in a specification via the “meta predicate” \textit{basevars}, which is defined here.

**Definition**\[ \textit{stvars} :: \text{'}a \text{ statefun} \Rightarrow \text{bool} \]

**Where** \text{\textit{basevars-def}}: \textit{stvars} \equiv \text{surj}

**Syntax**
\[
\begin{align*}
\textit{PRED} &:: \text{lift} \Rightarrow \text{'}a \\
\textit{-stvars} &:: \text{lift} \Rightarrow \text{bool}
\end{align*}
\]

**Translations**
\[
\begin{align*}
\textit{PRED} P &\rightarrow (\textit{P}:\text{state} => -) \\
\textit{-stvars} &\leftarrow \text{CONST stvars}
\end{align*}
\]

Base variables may be assigned arbitrary (type-correct) values. In the following lemma, note that \textit{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, \textit{basevars} \((x, x)\) would definitely be inconsistent.

**Lemma** \textit{basevars}: \textit{basevars} \textit{vs} \Rightarrow \exists u. \textit{vs} u = c

**Proof**

**Lemma** \textit{baseE}:
\[
\begin{align*}
\text{assumes } H1: \text{basevars} \textit{v} \text{ and } H2: \forall x. \textit{v} x = c \Rightarrow Q \\
\text{shows } Q
\end{align*}
\]

**Proof**

A variant written for sequences rather than single states.

**Lemma** \textit{first-baseE}:
\[
\begin{align*}
\text{assumes } H1: \text{basevars} \textit{v} \text{ and } H2: \forall x. \textit{v} (\text{first} x) = c \Rightarrow Q \\
\text{shows } Q
\end{align*}
\]

**Proof**

**Lemma** \textit{base-pair1}:
\[
\begin{align*}
\text{assumes } h: \text{basevars} \textit{(x,y)} \\
\text{shows } \text{basevars} \textit{x}
\end{align*}
\]

**Proof**

**Lemma** \textit{base-pair2}:
\[
\begin{align*}
\text{assumes } h: \text{basevars} \textit{(x,y)} \\
\text{shows } \text{basevars} \textit{y}
\end{align*}
\]

**Proof**

**Lemma** \textit{base-pair}: \textit{basevars} \textit{(x,y)} \Rightarrow \text{basevars} \textit{x} \land \text{basevars} \textit{y}

**Proof**

Since the \textit{unit} type has just one value, any state function of unit type satisfies the predicate \textit{basevars}. The following theorem can sometimes be useful because it gives a trivial solution for \textit{basevars} premises.
lemma unit-base: basevars (v::state ⇒ unit)
⟨proof⟩
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⟩

lemma
fixes x :: state ⇒ nat
assumes h1: basevars (x,x)
shows False
⟨proof⟩

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
⟨proof⟩

7.1 Temporal Quantifiers

In [5], Lamport gives a stuttering invariant definition of quantification over (flexible) variables. It relies on similarity of two sequences (as supported in our TLA.Sequence theory), and equivalence of two sequences up to a variable (the bound variable). However, sequence equivalence up to a variable, requires state equivalence up to a variable. Our state representation above does not support this, hence we cannot encode Lamport’s definition in our theory. Thus, we need to axiomatise quantification over (flexible) variables. Note that with a state representation supporting this, our theory should allow such an encoding.

consts
EEx :: (∀ statefun ⇒ temporal) ⇒ temporal (binder Eex 10)
AAll :: (∀ statefun ⇒ temporal) ⇒ temporal (binder Aall 10)
syntax
- EEx :: [idts, lift] ⇒ lift ((∃∃ -./ -) [0,10] 10)
- AAll :: [idts, lift] ⇒ lift ((∀∀ -./ -) [0,10] 10)
translations
-EEz v A == Eez v. A
-AAll v A == Aall v. A

axiomatization where

\[ \text{exI}: \vdash F x \rightarrow (\exists x. F x) \]
and \[ \text{exE}: [s \models (\exists x. F x) ; \text{basevars } vs; (!! x. [\text{basevars } (x, vs); s \models F x] \rightarrow s \models G)] \rightarrow (s \models G) \]
and \[ \text{all-def}: \vdash (\forall x. F x) = (\neg (\exists x. \neg(F x))) \]
and \[ \text{exSTUT}: \text{STUTINV } F x \Rightarrow \text{STUTINV } (\exists x. F x) \]
and \[ \text{history}: \vdash (I \land \Box [A] - v) = (\exists h. (h = h_a) \land I \land \Box [A \land h$=h_b]-(h,v)) \]

lemmas \[ \text{exI-unl} = \text{exI}[\text{unlift-rule}] \rightarrow w \models F x \Rightarrow w \models (\exists x. F x) \]

\text{tla-defs} can be used to unfold TLA definitions into lowest predicate level. This is particularly useful for reasoning about enabledness of formulas.

lemmas \[ \text{tla-defs} = \text{unch-def before-def after-def first-def second-def suffix-def} \]
\[ \text{tail-def nexts-def app-def angle-actrans-def actrans-def} \]

end

8 A simple illustrative example

theory Even
imports State
begin

A trivial example illustrating invariant proofs in the logic, and how Isabelle/HOL can help with specification. It proves that \( x \) is always even in a program where \( x \) is initialized as 0 and always incremented by 2.

inductive-set

Even :: nat set
where

| even-zero: 0 ∈ Even |
| even-step: n ∈ Even ⇒ Suc (Suc n) ∈ Even

locale Program =

fixes x :: state ⇒ nat
and init :: temporal
and act :: temporal
and phi :: temporal

defines init ≡ TEMP $x = # 0
and act ≡ TEMP x' = Suc<Suc<$x>>
and phi ≡ TEMP init ∧ □[act]-x

lemma (in Program) stutinvprog: STUTINV phi
lemma (in Program) ineven: \( \phi \rightarrow \Box(\$x \in \# \text{Even}) \)

\[ \]

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 pcount = a | b | g

locale Firstprogram =

fixes x :: state \Rightarrow nat

and y :: state \Rightarrow nat

and init :: temporal

and m1 :: temporal

and m2 :: temporal

and phi :: temporal

and Live :: temporal

defines init \equiv TEMP \$x = \#0 \land \$y = \#0

and m1 \equiv TEMP x' = Suc\$x\rangle \land y' = \$y

and m2 \equiv TEMP y' = Suc\$y\rangle \land x' = \$x

and Live \equiv TEMP WF(m1)-(x,y) \land WF(m2)-(x,y)

and phi \equiv TEMP (init \land \Box[m1 \lor m2]-(x,y) \land Live)

assumes bvar: basevars (x,y)

lemma (in Firstprogram) STUTINV phi

\[ \]

lemma (in Firstprogram) enabled-m1: \( \vdash \text{Enabled } (m1)-(x,y) \)

\[ \]

lemma (in Firstprogram) enabled-m2: \( \vdash \text{Enabled } (m2)-(x,y) \)

\[ \]

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 ≡ LIFT (x,y,sem,pc1,pc2)
and initPsi ≡ TEMP $pc1 = # a ∧ $pc2 = # a ∧ $x = # 0 ∧ $y = # 0 ∧ $sem = # 1
and alpha1 ≡ TEMP $pc1 =# a ∧ # 0 < $sem ∧ pc1$ = #b ∧ sem$ = $sem
- # 1 ∧ Unchanged (x,y,pc2)
and alpha2 ≡ TEMP $pc2 =# a ∧ # 0 < $sem ∧ pc2' = #b ∧ sem$ = $sem
- # 1 ∧ Unchanged (x,y,pc1)
and beta1 ≡ TEMP $pc1 =# b ∧ pc1' = #g ∧ x' = Suc<$x$> ∧ Unchanged (y,sem,pc2)
and beta2 ≡ TEMP $pc2 =# b ∧ pc2' = #g ∧ y' = Suc<$y$> ∧ Unchanged (x,sem,pc1)
and gamma1 ≡ TEMP $pc1 =# g ∧ pc1' = #a ∧ sem' = Suc<$sem$> ∧ Unchanged (x,y,pc1)
and gamma2 ≡ TEMP $pc2 =# g ∧ pc2' = #a ∧ sem' = Suc<$sem$> ∧ Unchanged (x,y,pc2)
and n1 ≡ TEMP (alpha1 ∨ beta1 ∨ gamma1)
and n2 ≡ TEMP (alpha2 ∨ beta2 ∨ gamma2)
and Live2 ≡ TEMP SF(n1)-vars ∧ SF(n2)-vars
and psi ≡ TEMP (initPsi ∧ □[n1 ∨ n2]-vars ∧ Live2)
and I ≡ TEMP ($sem = # 1 ∧ $pc1 = # a ∧ $pc2 = # a)
∨ ($sem = # 0 ∧ (($pc1 = # a ∧ $pc2 ∈ {#b , #g})
∨ ($pc2 = # a ∧ $pc1 ∈ {#b , #g})))

assumes bvar2: basevars vars

lemmas (in Secondprogram) Sact2-defs = n1-def n2-def alpha1-def beta1-def gamma1-def alpha2-def beta2-def gamma2-def

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

lemma (in Secondprogram) psiI: ⊢ psi → □I
(proof)

Using this invariant we now prove step simulation, i.e. the safety part of the
refinement proof.

theorem (in Secondprogram) step-simulation: ⊢ psi → init ∧ □[m1 ∨ m2]-(x,y)
(proof)

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) \( n1\text{-ch} : \neg \langle n1\rangle\text{-vars} = n1 \)

(\textit{proof})

**Lemma** (in Secondprogram) \( \text{enab-alpha1} : \vdash \langle pc1 = \#a \rightarrow \# 0 < \text{sem} \rightarrow \rangle \text{Enabled alpha1} \)

(\textit{proof})

**Lemma** (in Secondprogram) \( \text{enab-beta1} : \vdash \langle pc1 = \#b \rightarrow \rangle \text{Enabled beta1} \)

(\textit{proof})

**Lemma** (in Secondprogram) \( \text{enab-gamma1} : \vdash \langle pc1 = \#g \rightarrow \rangle \text{Enabled gamma1} \)

(\textit{proof})

**Lemma** (in Secondprogram) \( \text{enab-n1} : \vdash \langle \text{Enabled} \langle n1\rangle\text{-vars} = (\langle pc1 = \#a \rightarrow \# 0 < \text{sem} \rangle \rangle \)

(\textit{proof})

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

**Lemma** (in Secondprogram) \( n2\text{-ch} : \neg \langle n2\rangle\text{-vars} = n2 \)

(\textit{proof})

**Lemma** (in Secondprogram) \( \text{enab-alpha2} : \vdash \langle pc2 = \#a \rightarrow \# 0 < \text{sem} \rightarrow \rangle \text{Enabled alpha2} \)

(\textit{proof})

**Lemma** (in Secondprogram) \( \text{enab-beta2} : \vdash \langle pc2 = \#b \rightarrow \rangle \text{Enabled beta2} \)

(\textit{proof})

**Lemma** (in Secondprogram) \( \text{enab-gamma2} : \vdash \langle pc2 = \#g \rightarrow \rangle \text{Enabled gamma2} \)

(\textit{proof})

**Lemma** (in Secondprogram) \( \text{enab-n2} : \vdash \langle \text{Enabled} \langle n2\rangle\text{-vars} = (\langle pc2 = \#a \rightarrow \# 0 < \text{sem} \rangle \rangle \)

(\textit{proof})

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) \( \psi\text{-fair-m1} : \vdash \psi \rightarrow SF(m1)-(x,y) \)

(\textit{proof})

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

(\textit{proof})
We can now prove the main theorem, which states that \( \psi \) implements \( \phi \).

**Theorem** (in Secondprogram) \( \vdash \psi \rightarrow \phi \)

(Proof)

\[\text{end}\]

## 10 Refining a Buffer Specification

**Theory** Buffer

**Imports** State

**Begin**

We specify a simple FIFO buffer and prove that two FIFO buffers in a row implement a FIFO buffer.

### 10.1 Buffer specification

The following definitions all take three parameters: a state function representing the input channel of the FIFO buffer, another representing the internal queue, and a third one representing the output channel. These parameters will be instantiated later in the definition of the double FIFO.

**Definition** BInit :: \( \text{a statefun} \Rightarrow \text{a list statefun} \Rightarrow \text{a statefun} \Rightarrow \text{temporal} \)

where \( \text{BInit } ic \ q \ oc \equiv \text{TEMP } \# q = \# [] \)

\( \land ic = oc \) — initial condition of buffer

**Definition** Enq :: \( \text{a statefun} \Rightarrow \text{a list statefun} \Rightarrow \text{a statefun} \Rightarrow \text{temporal} \)

where \( \text{Enq } ic \ q \ oc \equiv \text{TEMP } \# ic \neq \# oc \)

\( \land q = q @ [ ic ] \)

\( \land oc = oc \) — enqueue a new value

**Definition** Deq :: \( \text{a statefun} \Rightarrow \text{a list statefun} \Rightarrow \text{a statefun} \Rightarrow \text{temporal} \)

where \( \text{Deq } ic \ q \ oc \equiv \text{TEMP } \# 0 < \text{length } q \)

\( \land oc = \text{hd } q \)

\( \land q = \text{tl } q \)

\( \land ic = ic \) — dequeue value at front

**Definition** Nxt :: \( \text{a statefun} \Rightarrow \text{a list statefun} \Rightarrow \text{a statefun} \Rightarrow \text{temporal} \)

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** ISpec :: \( \text{a statefun} \Rightarrow \text{a list statefun} \Rightarrow \text{a statefun} \Rightarrow \text{temporal} \)

where \( \text{ISpec } ic \ q \ oc \equiv \text{TEMP } \text{BInit } ic \ q \ oc \)

\( \land \Box [\text{Nxt } ic \ q \ oc]-(ic,q,oc) \)

\( \land \text{WF(Deq } ic \ q \ oc)-(ic,q,oc) \)

— external specification: buffer hidden

**Definition** Spec :: \( \text{a statefun} \Rightarrow \text{a statefun} \Rightarrow \text{temporal} \)

56
where \( \text{Spec } ic \ oc \leftarrow \text{TEMP } (\exists q. \text{ISpec } ic \ q \ oc) \)

### 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** \( \text{noreps} :: \text{'a list } \Rightarrow \text{bool} \)

where

\[
\text{noreps } xs \equiv \forall i < \text{length } xs - 1. \ xs!i \neq xs!(\text{Suc } i)
\]

**Definition** \( \text{BInv} :: \text{'a statefun } \Rightarrow \text{'a list statefun } \Rightarrow \text{'a statefun } \Rightarrow \text{temporal} \)

where

\[
\text{BInv } ic \ q \ oc \equiv \text{TEMP List.last}<\$oc \ # \$q> = \$ic \wedge \text{noreps}<\$oc \ # \$q>
\]

**Lemmas** \( \text{buffer-defs } = \text{BInit-def Enq-def Deq-def Nxt-def} \)

\( \text{ISpec-def Spec-def BInv-def} \)

**Lemma** \( \text{ISpec-stutinv}: \text{STUTINV } (\text{ISpec } ic \ q \ oc) \)

\( \langle \text{proof} \rangle \)

**Lemma** \( \text{Spec-stutinv}: \text{STUTINV } \text{Spec } ic \ oc \)

\( \langle \text{proof} \rangle \)

A lemma about lists that is useful in the following

**Lemma** \( \text{tl-self-iff-empty}[\text{simp}]: (\text{tl } xs = xs) = (xs = []) \)

\( \langle \text{proof} \rangle \)

**Lemma** \( \text{tl-self-iff-empty}[\text{simp}]: (xs = \text{tl } xs) = (xs = []) \)

\( \langle \text{proof} \rangle \)

**Lemma** \( \text{Deq-visible}: \)

\( \text{assumes } v : \vdash \text{Unchanged } v \rightarrow \text{Unchanged } q \)

\( \text{shows } \sim <\text{Deq } ic \ q \ oc>-v = \text{Deq } ic \ q \ oc \)

\( \langle \text{proof} \rangle \)

**Lemma** \( \text{Deq-enabledE}: \vdash \text{Enabled } <\text{Deq } ic \ q \ oc>-\{ic,q,oc\} \rightarrow q \sim = \#[] \)

\( \langle \text{proof} \rangle \)

We now prove that \( \text{BInv} \) is an invariant of the Buffer specification.
We need several lemmas about \textit{noreps} that are used in the invariant proof.

\begin{itemize}
  \item \textbf{lemma} \texttt{noreps-empty [simp]: noreps []}
  \item \textbf{lemma} \texttt{noreps-singleton: noreps [x] \textemdash\ textit{special case of following lemma}}
  \item \textbf{lemma} \texttt{noreps-cons [simp]}
    \begin{equation}
      \texttt{noreps (x \# xs)} = (\texttt{noreps xs} \land (\texttt{xs} = [] \lor x \neq \texttt{hd xs}))
    \end{equation}
  \item \textbf{lemma} \texttt{noreps-append [simp]}
    \begin{equation}
      \texttt{noreps (xs @ ys)} =
      (\texttt{noreps xs} \land \texttt{noreps ys} \land (\texttt{xs} = [] \lor \texttt{ys} = [] \lor \text{List.last xs} \neq \texttt{hd ys}))
    \end{equation}
\end{itemize}

\begin{itemize}
  \item \textbf{lemma} \texttt{ISpec-BInv-lemma:}
    \begin{equation}
      \vdash \texttt{Binit ic q oc} \land \square [\texttt{Nxt ic q oc}]- (ic,q,oc) \rightarrow \square (\texttt{BInv ic q oc})
    \end{equation}
  \item \textbf{theorem} \texttt{ISpec-BInv: \vdash ISpec ic q oc \rightarrow \square (BInv ic q oc)}
\end{itemize}

\section{Two FIFO buffers in a row implement a buffer}

\begin{itemize}
  \item \textbf{locale} \texttt{DBuffer =}
    \begin{itemize}
      \item \texttt{fixes} \texttt{inp :: 'a statefun \textemdash\ textit{input channel for double FIFO}}
      \item \texttt{and} \texttt{mid :: 'a statefun \textemdash\ textit{channel linking the two buffers}}
      \item \texttt{and} \texttt{out :: 'a statefun \textemdash\ textit{output channel for double FIFO}}
      \item \texttt{and} \texttt{q1 :: 'a list statefun \textemdash\ textit{inner queue of first FIFO}}
      \item \texttt{and} \texttt{q2 :: 'a list statefun \textemdash\ textit{inner queue of second FIFO}}
      \item \texttt{and} \texttt{vars}
    \end{itemize}
    \begin{itemize}
      \item \texttt{defines} \texttt{vars \equiv LIFT (inp,mid,out,q1,q2)}
      \item \texttt{assumes} \texttt{DB-base: basevars vars}
    \end{itemize}
\end{itemize}

\begin{itemize}
  \item \texttt{begin}
  \item 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 \texttt{inp} and output channel \texttt{mid} whereas the second one receives from \texttt{mid} and outputs on \texttt{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.
\end{itemize}
The proof rules of TLA are geared towards specifications of the form $Init \land \Box[Next] \cdot vars \land L$, 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** $FullNxt \equiv TEMP \ (Enq \ inp \ q1 \ mid \land Unchanged \ (q2, out) \lor Deq \ inp \ q1 \ mid \land Enq \ mid \ q2 \ out \lor Deq \ mid \ q2 \ out \land Unchanged \ (inp, q1))$

**definition FullSpec**

**where** $FullSpec \equiv TEMP \ FullInit \land \Box[FullNxt] \cdot vars \land WF(Deq \ inp \ q1 \ mid) \cdot vars \land WF(Deq \ mid \ q2 \ out) \cdot vars$

The concatenation of the two queues will serve as the refinement mapping.

**definition** $qc :: \ 'a \ list \ statefun$

**where** $qc \equiv LIFT \ (q2 @ q1)$

**lemmas** $db-defs = buffer-defs \ DBSpec-def \ FullInit-def \ FullNxt-def \ FullSpec-def \ qc-def \ vars-def$

**lemma** $DBSpec-stutinv: \ STUTINV \ DBSpec$

(Proof)

**lemma** $FullSpec-stutinv: \ STUTINV \ FullSpec$

(Proof)

We prove that $DBSpec$ implies $FullSpec$. (The converse implication also holds but is not needed for our implementation proof.)

The following lemma is somewhat more bureaucratic than we’d like it to be. It shows that the conjunction of the next-state relations, together with the invariant for the first queue, implies the full next-state relation of the combined queues.

**lemma** $DBNxt-then-FullNxt:$$\vdash \Box[\Box[Nxt \ inp \ q1 \ mid] \cdot (inp, q1, mid) \land \Box[Nxt \ mid \ q2 \ out] \cdot (mid, q2, out) \land \Box[\neg(Enq \ inp \ q1 \ mid \land Deq \ mid \ q2 \ out)] \cdot vars$
It is now easy to show that $DBSpec$ refines $FullSpec$.

**Theorem DBSpec-impl-FullSpec:** $DBSpec \rightarrow FullSpec$

We now prove that two FIFO buffers in a row (as specified by formula $FullSpec$) implement a FIFO buffer whose internal queue is the concatenation of the two buffers. We start by proving step simulation.

**Lemma FullInit:** $FullInit \rightarrow BInit inp qc out$

**Lemma Full-step-simulation:**

\[ \neg \square[FullNxt]-vars \rightarrow [Nxt inp q c out](inp,qc,out) \]

The liveness condition requires that the combined buffer eventually performs a $Deq$ action on the output channel if it contains some element. The idea is to use the fairness hypothesis for the first buffer to prove that in that case, eventually the queue of the second buffer will be non-empty, and that it must therefore eventually dequeue some element.

The first step is to establish the enabledness conditions for the two $Deq$ actions of the implementation.

**Lemma Deq1-enabled:** $\neg \square[FullNxt]-vars \rightarrow Enabled\ (Deq\ inp\ q1\ mid)(q1 \neq \#)$

**Lemma Deq2-enabled:** $\neg \square[FullNxt]-vars \rightarrow Enabled\ (Deq\ mid\ q2\ out)(q2 \neq \#)$

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:**

\[ \neg \square[FullNxt]-vars \land WF(Deq\ mid\ q2\ out)-vars \land \square WF(Deq\ inp\ q1\ mid)-vars \rightarrow WF(Deq\ inp\ q c out)-(inp,qc,out) \]

Putting everything together, we obtain that $FullSpec$ refines the Buffer specification under the refinement mapping.

**Theorem FullSpec-impl-ISpec:** $FullSpec \rightarrow ISpec\ inp\ q c out$

**Theorem FullSpec-impl-Spec:** $FullSpec \rightarrow Spec\ inp\ out$
By transitivity, two buffers in a row also implement a single buffer.

\textbf{theorem} DBSpec-impl-Spec: \(\vdash\) DBSpec \(\longrightarrow\) Spec inp out

end — locale DBuffer

References


