A Definitional Encoding of TLA in Isabelle/HOL

Gudmund Grov & Stephan Merz

March 17, 2025

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

theory *Sequence*
imports *Main*
begin

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

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

type-synonym *'a seq* = *nat* ⇒ *'a*

1.1 Some operators on sequences

Some general functions on sequences are provided

definition *first* :: *'a seq* ⇒ *'a*
where *first s* ≡ *s 0*

definition *second* :: (*'a seq*) ⇒ *'a*
where *second s* ≡ *s 1*

definition *suffix* :: *'a seq* ⇒ *nat* ⇒ *'a seq* (**infixl** ‹$|_s$› *60*)
where $s \;|_s\; i \equiv \lambda\ n.\ s\ (n{+}i)$

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

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

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

lemma *suffix-first*: $first\ (s\ |_s\ n) = s\ n$
⟨*proof*⟩

lemma *suffix-second*: $second\ (s\ |_s\ n) = s\ (Suc\ n)$
⟨*proof*⟩

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

lemma *suffix-commute*: $((s\ |_s\ n)\ |_s\ m) = ((s\ |_s\ m)\ |_s\ n)$
⟨*proof*⟩

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

lemma *suffix-zero*[*simp*]: $s\ |_s\ 0 = s$
⟨*proof*⟩

lemma *suffix-tail*: $s\ |_s\ 1 = tail\ s$
⟨*proof*⟩

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

1.1.3 Properties of (##)

lemma *seq-app-second*: $(s\ \#\#\ \sigma)\ 1 = \sigma\ 0$

⟨*proof*⟩

lemma *seq-app-first*: $(s \ \#\# \ \sigma) \ 0 = s$
⟨*proof*⟩

lemma *seq-app-first-tail*: $(\mathit{first}\ s) \ \#\# \ (\mathit{tail}\ s) = s$
⟨*proof*⟩

lemma *seq-app-tail*: $\mathit{tail}\ (x \ \#\# \ s) = s$
⟨*proof*⟩

lemma *seq-app-greater-than-zero*: $n > 0 \Longrightarrow (s \ \#\# \ \sigma) \ n = \sigma \ (n - 1)$
⟨*proof*⟩

1.2 Finite and Empty Sequences

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

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

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

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

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

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

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

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

1.2.1 Properties of *emptyseq*

lemma *empty-is-finite*: **assumes** *emptyseq s* **shows** *fin s*

⟨*proof*⟩

lemma *empty-suffix-is-empty*: **assumes** *H*: *emptyseq s* **shows** *emptyseq* $(s \mid_s n)$
⟨*proof*⟩

lemma *suc-empty*: **assumes** *H1*: *emptyseq* $(s \mid_s m)$ **shows** *emptyseq* $(s \mid_s (Suc\ m))$
⟨*proof*⟩

lemma *empty-suffix-exteq*: **assumes** *H*:*emptyseq s* **shows** $(s \mid_s n)\ m = s\ m$
⟨*proof*⟩

lemma *empty-suffix-eq*: **assumes** *H*: *emptyseq s* **shows** $(s \mid_s n) = s$
⟨*proof*⟩

lemma *seq-empty-all*: **assumes** *H*: *emptyseq s* **shows** $s\ i = s\ j$
⟨*proof*⟩

1.2.2 Properties of *Sequence.last* and *laststate*

lemma *fin-stut-after-last*: **assumes** *H*: *fin s* **shows** $\forall j \geq last\ s.\ s\ j = s\ (last\ s)$
⟨*proof*⟩

1.3 Stuttering Invariance

This subsection provides functions for removing stuttering steps of sequences, i.e. we formalise Lamports ♮ 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 ♮ 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* :: $('a\ seq) \Rightarrow bool$
where *nonstutseq* $s \equiv \forall\ i.\ s\ i = s\ (Suc\ i) \longrightarrow (\forall\ j > i.\ s\ i = s\ j)$

definition *stutstep* :: $('a\ seq) \Rightarrow nat \Rightarrow bool$
where *stutstep* $s\ n \equiv (s\ n = s\ (Suc\ n))$

definition *nextnat* :: $('a\ seq) \Rightarrow nat$
where *nextnat* $s \equiv$ *if emptyseq s then 0 else LEAST i.* $s\ i \neq s\ 0$

definition *nextsuffix* :: $('a\ seq) \Rightarrow ('a\ seq)$
where *nextsuffix* $s \equiv s \mid_s (nextnat\ s)$

fun *next* :: $nat \Rightarrow ('a\ seq) \Rightarrow ('a\ seq)$ **where**

next 0 = *id*
| *next* (*Suc n*) = *nextsuffix o* (*next n*)

definition *collapse* :: (*'a seq*) ⇒ (*'a seq*) (‹♮›)
where ♮ *s* ≡ λ *n*. (*next n s*) *0*

Predicate *nonstutseq* identifies sequences without any stuttering steps – except possibly for infinite stuttering at the end. Further, *stutstep s n* is a predicate which holds if the element after *s n* is equal to *s n*, i.e. *Suc n* is a stuttering step. ♮ *s* formalises Lamports ♮ operator. It returns the first state of the result of *next n s*. *next n s* finds suffix of the n^{th} change. Hence the first element, which ♮ *s* returns, is the state after the n^{th} change. *next n s* is defined by primitive recursion on *n* using function composition of function *nextsuffix*. E.g. *next 3 s* equals *nextsuffix* (*nextsuffix* (*nextsuffix s*)). *nextsuffix s* returns the suffix of the sequence starting at the next changing state. It uses *nextnat* to obtain this. 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 *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 *Least*.

1.3.1 Properties of *nonstutseq*

lemma *seq-empty-is-nonstut*:
  assumes *H*: *emptyseq s* **shows** *nonstutseq s*
  ⟨*proof*⟩

lemma *notempty-exist-nonstut*:
  assumes *H*: ¬ *emptyseq* ($s\ |_s\ m$) **shows** ∃ *i*. $s\ i \neq s\ m \wedge i > m$
⟨*proof*⟩

1.3.2 Properties of *nextnat*

lemma *nextnat-le-unch*: **assumes** *H*: *n* < *nextnat s* **shows** *s n* = *s 0*
⟨*proof*⟩

lemma *stutnempty*:
  assumes *H*: ¬ *stutstep s n* **shows** ¬ *emptyseq* ($s\ |_s\ n$)
⟨*proof*⟩

lemma *notstutstep-nexnat1*:
  assumes *H*: ¬ *stutstep s n* **shows** *nextnat* ($s\ |_s\ n$) = *1*
⟨*proof*⟩

lemma *stutstep-notempty-notempty*:
  assumes *h1*: *emptyseq* ($s\ |_s$ *Suc n*) (**is** *emptyseq ?sn*)
    and *h2*: *stutstep s n*
  shows *emptyseq* ($s\ |_s\ n$) (**is** *emptyseq ?s*)

⟨*proof*⟩

lemma *stutstep-empty-suc*:
assumes *stutstep s n*
shows *emptyseq* $(s \mid_s \mathit{Suc}\ n) = \mathit{emptyseq}\ (s \mid_s n)$
⟨*proof*⟩

lemma *stutstep-notempty-sucnextnat*:
assumes *h1*: ¬ *emptyseq* $(s \mid_s n)$ **and** *h2*: *stutstep s n*
shows $(\mathit{nextnat}\ (s \mid_s n)) = \mathit{Suc}\ (\mathit{nextnat}\ (s \mid_s (\mathit{Suc}\ n)))$
⟨*proof*⟩

lemma *nextnat-empty-neq*: **assumes** *H*: ¬ *emptyseq s* **shows** $s\ (\mathit{nextnat}\ s) \neq s\ 0$
⟨*proof*⟩

lemma *nextnat-empty-gzero*: **assumes** *H*: ¬ *emptyseq s* **shows** $\mathit{nextnat}\ s > 0$
⟨*proof*⟩

1.3.3 Properties of *nextsuffix*

lemma *empty-nextsuffix*:
assumes *H*: *emptyseq s* **shows** $\mathit{nextsuffix}\ s = s$
⟨*proof*⟩

lemma *empty-nextsuffix-id*:
assumes *H*: *emptyseq s* **shows** $\mathit{nextsuffix}\ s = \mathit{id}\ s$
⟨*proof*⟩

lemma *notstutstep-nextsuffix1*:
assumes *H*: ¬ *stutstep s n* **shows** $\mathit{nextsuffix}\ (s \mid_s n) = s \mid_s (\mathit{Suc}\ n)$
⟨*proof*⟩

1.3.4 Properties of *next*

lemma *next-suc-suffix*: $\mathit{next}\ (\mathit{Suc}\ n)\ s = \mathit{nextsuffix}\ (\mathit{next}\ n\ s)$
⟨*proof*⟩

lemma *next-suffix-com*: $\mathit{nextsuffix}\ (\mathit{next}\ n\ s) = (\mathit{next}\ n\ (\mathit{nextsuffix}\ s))$
⟨*proof*⟩

lemma *next-plus*: $\mathit{next}\ (m+n)\ s = \mathit{next}\ m\ (\mathit{next}\ n\ s)$
⟨*proof*⟩

lemma *next-empty*: **assumes** *H*: *emptyseq s* **shows** $\mathit{next}\ n\ s = s$
⟨*proof*⟩

lemma *notempty-nextnotzero*:
assumes *H*: ¬*emptyseq s* **shows** $(\mathit{next}\ (\mathit{Suc}\ 0)\ s)\ 0 \neq s\ 0$
⟨*proof*⟩

lemma *next-ex-id*: ∃ *i*. *s i* = (*next m s*) *0*
⟨*proof*⟩

1.3.5 Properties of ♮

lemma *emptyseq-collapse-eq*: **assumes** *A1*: *emptyseq s* **shows** ♮ *s* = *s*
⟨*proof*⟩

lemma *empty-collapse-empty*:
assumes *H*: *emptyseq s* **shows** *emptyseq* (♮ *s*)
⟨*proof*⟩

lemma *collapse-empty-empty*:
assumes *H*: *emptyseq* (♮ *s*) **shows** *emptyseq s*
⟨*proof*⟩

lemma *collapse-empty-iff-empty* [*simp*]: *emptyseq* (♮ *s*) = *emptyseq s*
⟨*proof*⟩

1.4 Similarity of Sequences

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

definition *seqsimilar* :: (′*a seq*) ⇒ (′*a seq*) ⇒ *bool* (**infixl** ‹≈› *50*)
where $\sigma \approx \tau \equiv$ (♮ σ) = (♮ τ)

1.4.1 Properties of (≈)

lemma *seqsim-refl* [*iff*]: *s* ≈ *s*
⟨*proof*⟩

lemma *seqsim-sym*: **assumes** *H*: *s* ≈ *t* **shows** *t* ≈ *s*
⟨*proof*⟩

lemma *seqeq-imp-sim*: **assumes** *H*: *s* = *t* **shows** *s* ≈ *t*
⟨*proof*⟩

lemma *seqsim-trans* [*trans*]: **assumes** *h1*: *s* ≈ *t* **and** *h2*: *t* ≈ *z* **shows** *s* ≈ *z*
⟨*proof*⟩

theorem *sim-first*: **assumes** *H*: *s* ≈ *t* **shows** *first s* = *first t*
⟨*proof*⟩

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

lemma *tail-sim-second*: **assumes** *H*: *tail s* ≈ *tail t* **shows** *second s* = *second t*

⟨*proof*⟩

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-notstutstep*:
assumes *H*: ¬ (*stutstep s n*) **shows** ($s\ |_s$ (*Suc n*)) ≈ *nextsuffix* ($s\ |_s\ n$)
⟨*proof*⟩

lemma *stut-nextsuf-suc*:
assumes *H*: *stutstep s n* **shows** *nextsuffix* ($s\ |_s\ n$) = *nextsuffix* ($s\ |_s$ (*Suc n*))
⟨*proof*⟩

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

lemma *seqsim-stutstep*:
assumes *H*: *stutstep s n* **shows** ($s\ |_s$ (*Suc n*)) ≈ ($s\ |_s\ n$) (**is** *?sn* ≈ *?s*)
⟨*proof*⟩

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

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

lemma *addfirststut*:
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** *H*: $s \approx t$ **shows** $\exists\ m.\ ((s\ |_s\ n) \approx (t\ |_s\ m))$
⟨*proof*⟩

lemma *nextnat-le-seqsim*:
assumes *n*: $n < nextnat\ s$ **shows** $s \approx (s\ |_s\ n)$
⟨*proof*⟩

lemma *seqsim-prev-nextnat*: $s \approx s\ |_s\ ((nextnat\ s) - 1)$
⟨*proof*⟩

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

lemma *seqsim-suffix-suc*:
assumes *H*: $s\ |_s\ n \approx t\ |_s\ m$
shows $\exists m'.\ s\ |_s\ n \approx t\ |_s\ m' \wedge ((s\ |_s\ Suc\ n \approx t\ |_s\ Suc\ m') \vee (s\ |_s\ Suc\ n \approx t\ |_s\ m'))$
⟨*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

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

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

theorem *sim-step*:
assumes *H*: $s \approx t$
shows $\exists\ m.\ s\ |_s\ n \approx t\ |_s\ m \wedge$
$((s\ |_s\ Suc\ n \approx t\ |_s\ Suc\ m) \vee (s\ |_s\ Suc\ n \approx t\ |_s\ m))$
(**is** $\exists m.\ ?Sim\ n\ m$)
⟨*proof*⟩

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 ($\longrightarrow$) and meta-level entailment ($\Longrightarrow$) commute, viz. the proof rule *impI*: $(?P \Longrightarrow ?Q) \Longrightarrow ?P \longrightarrow ?Q$. 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, $\vdash A \longrightarrow \Box A$, 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* $= 'w \Rightarrow 'a$
type-synonym $'w$ *form* $= ('w,$ *bool*$)$ *expr*

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

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

definition *const* :: $'a \Rightarrow ('w::world,\ 'a)$ *expr*
  where *unl-con*: *const* $c\ w \equiv c$

definition *lift* :: $['a \Rightarrow 'b,\ ('w::world,\ 'a)\ expr] \Rightarrow ('w,'b)\ expr$
  where *unl-lift*: *lift* $f\ x\ w \equiv f\ (x\ w)$

definition *lift2* :: $['a \Rightarrow 'b \Rightarrow 'c,\ ('w::world,'a)\ expr,\ ('w,'b)\ expr] \Rightarrow ('w,'c)\ expr$
  where *unl-lift2*: *lift2* $f\ x\ y\ w \equiv f\ (x\ w)\ (y\ w)$

definition *lift3* :: $['a \Rightarrow 'b => 'c \Rightarrow 'd,\ ('w::world,'a)\ expr,\ ('w,'b)\ expr,\ ('w,'c)\ expr] \Rightarrow ('w,'d)\ expr$
  where *unl-lift3*: *lift3* $f\ x\ y\ z\ w \equiv f\ (x\ w)\ (y\ w)\ (z\ w)$

definition *lift4* :: $['a \Rightarrow 'b => 'c \Rightarrow 'd \Rightarrow 'e,\ ('w::world,'a)\ expr,\ ('w,'b)\ expr,\ ('w,'c)\ expr,('w,'d)\ expr] \Rightarrow ('w,'e)\ expr$
  where *unl-lift4*: *lift4* $f\ x\ y\ z\ zz\ w \equiv f\ (x\ w)\ (y\ w)\ (z\ w)\ (zz\ w)$

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* :: $('a \Rightarrow ('w::world)\ form) \Rightarrow 'w\ form$ (**binder** ‹*Rall* › *10*)

where *unl-Rall*: (*Rall x. A x*) *w* ≡ ∀ *x. A x w*

definition *REx* :: (*'a* ⇒ (*'w::world*) *form*) ⇒ *'w form* (**binder** ‹*Rex* › *10*)
where *unl-Rex*: (*Rex x. A x*) *w* ≡ ∃ *x. A x w*

definition *REx1* :: (*'a* ⇒ (*'w::world*) *form*) ⇒ *'w form* (**binder** ‹*Rex*! › *10*)
where *unl-Rex1*: (*Rex*! *x. A x*) *w* ≡ ∃!*x. A x w*

RAll, *REx* and *REx1* introduces "rigid" quantification over values (of non-world types) within "intensional" formulas. *RAll* is universal quantification, *REx* is existential quantifcation. *REx1* requires unique existence.

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

lemmas *intensional-rews*[*simp*] =
unl-con unl-lift unl-lift2 unl-lift3 unl-lift4
unl-Rall unl-Rex unl-Rex1

2.2 Concrete Syntax

nonterminal
lift **and** *liftargs*

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

syntax

```
             :: id ⇒ lift                    (‹-›)
             :: longid ⇒ lift                (‹-›)
             :: var ⇒ lift                   (‹-›)
-applC       :: [lift, cargs] ⇒ lift         (‹(1-/ -)› [1000, 1000] 999)
             :: lift ⇒ lift                  (‹'(-')›)
-lambda      :: [idts, 'a] ⇒ lift            (‹(3%-./ -)› [0, 3] 3)
-constrain   :: [lift, type] ⇒ lift          (‹(-::-)› [4, 0] 3)
             :: lift ⇒ liftargs              (‹-›)
-liftargs    :: [lift, liftargs] ⇒ liftargs  (‹-,/ -›)
-Valid       :: lift ⇒ bool                  (‹(⊢ -)› 5)
-holdsAt     :: ['a, lift] ⇒ bool            (‹(- ⊨ -)› [100,10] 10)


LIFT         :: lift ⇒ 'a                    (‹LIFT -›)


-const       :: 'a ⇒ lift                    (‹(#-)› [1000] 999)
-lift        :: ['a, lift] ⇒ lift            (‹(-<->)› [1000] 999)
-lift2       :: ['a, lift, lift] ⇒ lift      (‹(-<-,/ ->)› [1000] 999)
-lift3       :: ['a, lift, lift, lift] ⇒ lift   (‹(-<-,/ -,/ ->)› [1000] 999)
-lift4       :: ['a, lift, lift, lift,lift] ⇒ lift   (‹(-<-,/ -,/ -,/ ->)› [1000] 999)
```

```
-liftEqu    :: [lift, lift] ⇒ lift            (‹(- =/ -)› [50,51] 50)
-liftNeq    :: [lift, lift] ⇒ lift            (infixl ‹≠› 50)
-liftNot    :: lift ⇒ lift                    (‹¬ -› [90] 90)
-liftAnd    :: [lift, lift] ⇒ lift            (infixr ‹∧› 35)
-liftOr     :: [lift, lift] ⇒ lift            (infixr ‹∨› 30)
-liftImp    :: [lift, lift] ⇒ lift            (infixr ‹⟶› 25)
-liftIf     :: [lift, lift, lift] ⇒ lift      (‹(if (-)/ then (-)/ else (-))› 10)
-liftPlus   :: [lift, lift] ⇒ lift            (‹(- +/ -)› [66,65] 65)
-liftMinus  :: [lift, lift] ⇒ lift            (‹(- −/ -)› [66,65] 65)
-liftTimes  :: [lift, lift] ⇒ lift            (‹(- ∗/ -)› [71,70] 70)
-liftDiv    :: [lift, lift] ⇒ lift            (‹(- div -)› [71,70] 70)
-liftMod    :: [lift, lift] ⇒ lift            (‹(- mod -)› [71,70] 70)
-liftLess   :: [lift, lift] ⇒ lift            (‹(-/ < -)› [50, 51] 50)
-liftLeq    :: [lift, lift] ⇒ lift            (‹(-/ ≤ -)› [50, 51] 50)
-liftMem    :: [lift, lift] ⇒ lift            (‹(-/ ∈ -)› [50, 51] 50)
-liftNotMem :: [lift, lift] ⇒ lift            (‹(-/ ∉ -)› [50, 51] 50)
-liftFinset :: liftargs => lift               (‹{(-)}›)

-liftPair   :: [lift,liftargs] ⇒ lift         (‹(1′(-,/ -′))›)

-liftCons :: [lift, lift] ⇒ lift              (‹(- #/ -)› [65,66] 65)
-liftApp  :: [lift, lift] ⇒ lift              (‹(- @/ -)› [65,66] 65)
-liftList :: liftargs ⇒ lift                  (‹[(-)]›)


-ARAll  :: [idts, lift] ⇒ lift                (‹(3! -./ -)› [0, 10] 10)
-AREx   :: [idts, lift] ⇒ lift                (‹(3? -./ -)› [0, 10] 10)
-AREx1  :: [idts, lift] ⇒ lift                (‹(3?! -./ -)› [0, 10] 10)
-RAll     :: [idts, lift] ⇒ lift              (‹(3∀ -./ -)› [0, 10] 10)
-REx      :: [idts, lift] ⇒ lift              (‹(3∃ -./ -)› [0, 10] 10)
-REx1     :: [idts, lift] ⇒ lift              (‹(3∃!-./ -)› [0, 10] 10)
```

translations
-const ⇌ *CONST const*

translations
-lift ⇌ *CONST lift*
-lift2 ⇌ *CONST lift2*
-lift3 ⇌ *CONST lift3*
-lift4 ⇌ *CONST lift4*
-Valid ⇌ *CONST Valid*

translations
-RAll x A ⇌ *Rall x. A*
-REx x A ⇌ *Rex x. A*
-REx1 x A ⇌ *Rex! x. A*

translations

-ARAll ⇀ *-RAll*
-AREx ⇀ *-REx*
-AREx1 ⇀ *-REx1*

w ⊨ *A* ⇀ *A w*
LIFT A ⇀ *A*::-⇒-

translations
-liftEqu ⇌ *-lift2* (=)
-liftNeq u v ⇌ *-liftNot* (*-liftEqu u v*)
-liftNot ⇌ *-lift* (*CONST Not*)
-liftAnd ⇌ *-lift2* (&)
-liftOr ⇌ *-lift2* ((|))
-liftImp ⇌ *-lift2* (−−>)
-liftIf ⇌ *-lift3* (*CONST If*)
-liftPlus ⇌ *-lift2* (+)
-liftMinus ⇌ *-lift2* (−)
-liftTimes ⇌ *-lift2* (∗)
-liftDiv ⇌ *-lift2* (*div*)
-liftMod ⇌ *-lift2* (*mod*)
-liftLess ⇌ *-lift2* (<)
-liftLeq ⇌ *-lift2* (<=)
-liftMem ⇌ *-lift2* (:)
-liftNotMem x xs ⇌ *-liftNot* (*-liftMem x xs*)

translations
-liftFinset (*-liftargs x xs*) ⇌ *-lift2* (*CONST insert*) *x* (*-liftFinset xs*)
-liftFinset x ⇌ *-lift2* (*CONST insert*) *x* (*-const* (*CONST Set.empty*))
-liftPair x (*-liftargs y z*) ⇌ *-liftPair x* (*-liftPair y z*)
-liftPair ⇌ *-lift2* (*CONST Pair*)
-liftCons ⇌ *-lift2* (*CONST Cons*)
-liftApp ⇌ *-lift2* (@)
-liftList (*-liftargs x xs*) ⇌ *-liftCons x* (*-liftList xs*)
-liftList x ⇌ *-liftCons x* (*-const* [])

w ⊨ ¬ *A* ↽ *-liftNot A w*
w ⊨ *A* ∧ *B* ↽ *-liftAnd A B w*
w ⊨ *A* ∨ *B* ↽ *-liftOr A B w*
w ⊨ *A* ⟶ *B* ↽ *-liftImp A B w*
w ⊨ *u* = *v* ↽ *-liftEqu u v w*
w ⊨ ∀ *x*. *A* ↽ *-RAll x A w*
w ⊨ ∃ *x*. *A* ↽ *-REx x A w*
w ⊨ ∃!*x*. *A* ↽ *-REx1 x A w*

syntax (*ASCII*)
-Valid :: *lift* ⇒ *bool* (‹(|− -)› *5*)
-holdsAt :: [′*a*, *lift*] ⇒ *bool* (‹(- |= -)› [*100*,*10*] *10*)
-liftNeq :: [*lift*, *lift*] ⇒ *lift* (‹(- ~=/ -)› [*50*,*51*] *50*)
-liftNot :: *lift* ⇒ *lift* (‹(~ -)› [*90*] *90*)

-liftAnd :: [*lift*, *lift*] ⇒ *lift* (‹(- &/ -)› [*36*,*35*] *35*)
-liftOr :: [*lift*, *lift*] ⇒ *lift* (‹(- |/ -)› [*31*,*30*] *30*)
-liftImp :: [*lift*, *lift*] ⇒ *lift* (‹(- −−>/ -)› [*26*,*25*] *25*)
-liftLeq :: [*lift*, *lift*] ⇒ *lift* (‹(-/ <= -)› [*50*, *51*] *50*)
-liftMem :: [*lift*, *lift*] ⇒ *lift* (‹(-/ : -)› [*50*, *51*] *50*)
-liftNotMem :: [*lift*, *lift*] ⇒ *lift* (‹(-/ ~: -)› [*50*, *51*] *50*)
-RAll :: [*idts*, *lift*] ⇒ *lift* (‹(*3ALL* -./ -)› [*0*, *10*] *10*)
-REx :: [*idts*, *lift*] ⇒ *lift* (‹(*3EX* -./ -)› [*0*, *10*] *10*)
-REx1 :: [*idts*, *lift*] ⇒ *lift* (‹(*3EX*! -./ -)› [*0*, *10*] *10*)

2.3 Lemmas and Tactics

lemma *intD*[*dest*]: $\vdash A \Longrightarrow w \models A$
⟨*proof*⟩

lemma *intI* [*intro*!]: **assumes** *P1*:($\bigwedge$ *w*. $w \models A$) **shows** $\vdash A$
⟨*proof*⟩

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

⟨*ML*⟩

lemma *inteq-reflection*: **assumes** *P1*: $\vdash x{=}y$ **shows** $(x \equiv y)$
⟨*proof*⟩

lemma *int-simps*:
$\vdash (x{=}x) = \#True$
$\vdash (\neg \#True) = \#False$
$\vdash (\neg \#False) = \#True$
$\vdash (\neg\neg P) = P$
$\vdash ((\neg P) = P) = \#False$
$\vdash (P = (\neg P)) = \#False$
$\vdash (P \neq Q) = (P = (\neg Q))$
$\vdash (\#True{=}P) = P$
$\vdash (P{=}\#True) = P$
$\vdash (\#True \longrightarrow P) = P$
$\vdash (\#False \longrightarrow P) = \#True$
$\vdash (P \longrightarrow \#True) = \#True$
$\vdash (P \longrightarrow P) = \#True$
$\vdash (P \longrightarrow \#False) = (\neg P)$
$\vdash (P \longrightarrow {\sim}P) = (\neg P)$
$\vdash (P \wedge \#True) = P$
$\vdash (\#True \wedge P) = P$
$\vdash (P \wedge \#False) = \#False$
$\vdash (\#False \wedge P) = \#False$
$\vdash (P \wedge P) = P$
$\vdash (P \wedge {\sim}P) = \#False$
$\vdash (\neg P \wedge P) = \#False$
$\vdash (P \vee \#True) = \#True$

$\vdash (\#True \vee P) = \#True$
$\vdash (P \vee \#False) = P$
$\vdash (\#False \vee P) = P$
$\vdash (P \vee P) = P$
$\vdash (P \vee \neg P) = \#True$
$\vdash (\neg P \vee P) = \#True$
$\vdash (\forall\ x.\ P) = P$
$\vdash (\exists\ x.\ P) = P$
⟨*proof*⟩

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

⟨*ML*⟩

lemma *Not-Rall*: $\vdash (\neg(\forall\ x.\ F\ x)) = (\exists\ x.\ \neg F\ x)$
⟨*proof*⟩

lemma *Not-Rex*: $\vdash (\neg(\exists\ x.\ F\ x)) = (\forall\ x.\ \neg F\ x)$
⟨*proof*⟩

lemma *TrueW* [*simp*]: $\vdash \#True$
⟨*proof*⟩

lemma *int-eq*: $\vdash X = Y \Longrightarrow X = Y$
⟨*proof*⟩

lemma *int-iffI*:
assumes $\vdash F \longrightarrow G$ **and** $\vdash G \longrightarrow F$
shows $\vdash F = G$
⟨*proof*⟩

lemma *int-iffD1*: **assumes** *h*: $\vdash F = G$ **shows** $\vdash F \longrightarrow G$
⟨*proof*⟩

lemma *int-iffD2*: **assumes** *h*: $\vdash F = G$ **shows** $\vdash G \longrightarrow F$
⟨*proof*⟩

lemma *lift-imp-trans*:
assumes $\vdash A \longrightarrow B$ **and** $\vdash B \longrightarrow C$
shows $\vdash A \longrightarrow C$
⟨*proof*⟩

lemma *lift-imp-neg*: **assumes** $\vdash A \longrightarrow B$ **shows** $\vdash \neg B \longrightarrow \neg A$
⟨*proof*⟩

lemma *lift-and-com*: $\vdash (A \wedge B) = (B \wedge A)$
⟨*proof*⟩

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:

type-synonym $('a,'b)$ $formfun = 'a\ seq \Rightarrow 'b$
type-synonym $'a\ formula = ('a,bool)\ formfun$
type-synonym $('a,'b)$ $stfun = 'a \Rightarrow 'b$
type-synonym $'a\ stpred = ('a,bool)\ stfun$

instance
 $fun :: (type,type)\ world\ \langle proof \rangle$

instance
 $prod :: (type,type)\ world\ \langle proof \rangle$

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.

definition $always :: ('a::world)\ formula \Rightarrow 'a\ formula$
where $always\ F \equiv \lambda\ s.\ \forall\ n.\ (s\ |_s\ n) \models F$

definition $nexts :: ('a::world)\ formula \Rightarrow 'a\ formula$

[1]See e.g. [10] for a discussion about deep vs. shallow embeddings in Isabelle/HOL.

where *nexts F* ≡ λ *s*. (*tail s*) ⊨ *F*

definition *before* :: ('*a*::*world*,'*b*) *stfun* ⇒ ('*a*,'*b*) *formfun*
where *before f* ≡ λ *s*. (*first s*) ⊨ *f*

definition *after* :: ('*a*::*world*,'*b*) *stfun* ⇒ ('*a*,'*b*) *formfun*
where *after f* ≡ λ *s*. (*second s*) ⊨ *f*

definition *unch* :: ('*a*::*world*,'*b*) *stfun* ⇒ '*a formula*
where *unch v* ≡ λ *s*. *s* ⊨ (*after v*) = (*before v*)

definition *action* :: ('*a*::*world*) *formula* ⇒ ('*a*,'*b*) *stfun* ⇒ '*a formula*
where *action P v* ≡ λ *s*. ∀ *i*. (($s \mid_s i$) ⊨ *P*) ∨ (($s \mid_s i$) ⊨ *unch v*)

3.2.1 Concrete Syntax

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

syntax
-always :: *lift* ⇒ *lift* (‹(□-)› [*90*] *90*)
-nexts :: *lift* ⇒ *lift* (‹(○-)› [*90*] *90*)
-action :: [*lift*,*lift*] ⇒ *lift* (‹(□[-]'-(-))› [*20*,*1000*] *90*)
-before :: *lift* ⇒ *lift* (‹($-)› [*100*] *99*)
-after :: *lift* ⇒ *lift* (‹(-$)› [*100*] *99*)
-prime :: *lift* ⇒ *lift* (‹(-‘)› [*100*] *99*)
-unch :: *lift* ⇒ *lift* (‹(*Unchanged* -)› [*100*] *99*)
TEMP :: *lift* ⇒ '*b* (‹(*TEMP* -)›)

syntax (*ASCII*)
-always :: *lift* ⇒ *lift* (‹([]-)› [*90*] *90*)
-nexts :: *lift* ⇒ *lift* (‹(*Next* -)› [*90*] *90*)
-action :: [*lift*,*lift*] ⇒ *lift* (‹([][-]'-(-))› [*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*
TEMP F ⇀ (*F*:: (*nat* ⇒ -) ⇒ -)

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* ≡ *TEMP*(*P* ∨ *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*(¬□[¬*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* (‹([-]'-(-))› [*20*,*1000*] *90*)
  -eventually :: *lift* ⇒ *lift* (‹(<>-)› [*90*] *90*)
  -angle-action :: [*lift*,*lift*] ⇒ *lift* (‹(<><->'-(-))› [*20*,*1000*] *90*)
  -angle-actrans :: [*lift*,*lift*] ⇒ *lift* (‹(<->'-(-))› [*20*,*1000*] *90*)
  -leadsto :: [*lift*,*lift*] ⇒ *lift* (‹(- ~> -)› [*26*,*25*] *25*)

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* (‹(- ↝ -)› [*26*,*25*] *25*)

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*: ($w \models \Diamond\ F$) = ($\exists\ n.\ (w\ |_s\ n) \models F$)
  ⟨*proof*⟩

lemma *angle-action-defs*: ($w \models \Diamond\langle P\rangle$-$v$) = ($\exists\ i.\ ((w\ |_s\ i) \models P) \wedge ((w\ |_s\ i) \models v\$ \neq \$v)$)
  ⟨*proof*⟩

lemma *unch-defs*: ($w \models$ *Unchanged* v) = (((*second* w) $\models v$) = ((*first* w) $\models v$))
  ⟨*proof*⟩

lemma *linalw*:

assumes *h1*: $a \leq b$ **and** *h2*: $(w\ |_s\ a) \models \Box A$
shows $(w\ |_s\ b) \models \Box A$
$\langle proof \rangle$

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.

definition *stutinv* :: $('a,'b)$ *formfun* $\Rightarrow$ *bool*
where *stutinv* $F \equiv \forall\ \sigma\ \tau.\ \sigma \approx \tau \longrightarrow (\sigma \models F) = (\tau \models F)$

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

definition *nstutinv* :: $('a,'b)$ *formfun* $\Rightarrow$ *bool*
where *nstutinv* $P \equiv \forall\ \sigma\ \tau.\ (\mathit{first}\ \sigma = \mathit{first}\ \tau) \land (\mathit{tail}\ \sigma) \approx (\mathit{tail}\ \tau) \longrightarrow (\sigma \models P) = (\tau \models P)$

syntax
-stutinv :: *lift* $\Rightarrow$ *bool* (‹(*STUTINV* -)› [*40*] *40*)
-nstutinv :: *lift* $\Rightarrow$ *bool* (‹(*NSTUTINV* -)› [*40*] *40*)

translations
-stutinv $\rightleftharpoons$ *CONST stutinv*
-nstutinv $\rightleftharpoons$ *CONST nstutinv*

Predicate *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 *NSTUTINV P* should be read as *nearly stuttering invariant* – and is required for some stuttering invariance proofs.

lemma *stutinv-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* $\Box F$
⟨*proof*⟩

Assuming that formula P is nearly suttering invariant then $\Box[P]\text{-}v$ will be stuttering invariant.

lemma *stut-action-lemma*:
assumes *H*: *NSTUTINV P* **and** *st*: $s \approx t$ **and** *P*: $t \models \Box[P]\text{-}v$
shows $s \models \Box[P]\text{-}v$
⟨*proof*⟩

theorem *stut-action*: **assumes** *H*: *NSTUTINV P* **shows** *STUTINV* $\Box[P]\text{-}v$
⟨*proof*⟩

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

lemma *stut-and*: $[\![STUTINV\ F; STUTINV\ G]\!] \Longrightarrow STUTINV\ (F \wedge G)$
⟨*proof*⟩

lemma *stut-or*: $[\![STUTINV\ F; STUTINV\ G]\!] \Longrightarrow STUTINV\ (F \vee G)$
⟨*proof*⟩

lemma *stut-imp*: $[\![STUTINV\ F; STUTINV\ G]\!] \Longrightarrow STUTINV\ (F \longrightarrow G)$
⟨*proof*⟩

lemma *stut-eq*: $[\![STUTINV\ F; STUTINV\ G]\!] \Longrightarrow STUTINV\ (F = G)$
⟨*proof*⟩

lemma *stut-noteq*: $[\![STUTINV\ F; STUTINV\ G]\!] \Longrightarrow STUTINV\ (F \neq G)$
⟨*proof*⟩

lemma *stut-not*: $STUTINV\ F \Longrightarrow STUTINV\ (\neg\ F)$

⟨*proof*⟩

lemma *stut-all*: $(\bigwedge x.\ STUTINV\ (F\ x)) \Longrightarrow STUTINV\ (\forall\ x.\ F\ x)$
⟨*proof*⟩

lemma *stut-ex*: $(\bigwedge x.\ STUTINV\ (F\ x)) \Longrightarrow STUTINV\ (\exists\ x.\ F\ x)$
⟨*proof*⟩

lemma *stut-const*: $STUTINV\ \#c$
⟨*proof*⟩

lemma *stut-fun1*: $STUTINV\ X \Longrightarrow STUTINV\ (f\ <X>)$
⟨*proof*⟩

lemma *stut-fun2*: $[\![STUTINV\ X;STUTINV\ Y]\!] \Longrightarrow STUTINV\ (f\ <X,Y>)$
⟨*proof*⟩

lemma *stut-fun3*: $[\![STUTINV\ X;STUTINV\ Y;STUTINV\ Z]\!] \Longrightarrow STUTINV\ (f\ <X,Y,Z>)$
⟨*proof*⟩

lemma *stut-fun4*: $[\![STUTINV\ X;STUTINV\ Y;STUTINV\ Z;\ STUTINV\ W]\!] \Longrightarrow STUTINV\ (f\ <X,Y,Z,W>)$
⟨*proof*⟩

lemma *stut-plus*: $[\![STUTINV\ x;STUTINV\ y]\!] \Longrightarrow STUTINV\ (x+y)$
⟨*proof*⟩

3.5.2 Properties of *-nstutinv*

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

lemma *nstut-nexts*: **assumes** H: $STUTINV\ F$ **shows** $NSTUTINV\ \bigcirc F$
⟨*proof*⟩

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

lemma *nstut-and*: $[\![NSTUTINV\ F;NSTUTINV\ G]\!] \Longrightarrow NSTUTINV\ (F \wedge G)$
⟨*proof*⟩

lemma *nstut-or*: $[\![NSTUTINV\ F;NSTUTINV\ G]\!] \Longrightarrow NSTUTINV\ (F \vee G)$
⟨*proof*⟩

lemma *nstut-imp*: $[\![NSTUTINV\ F;NSTUTINV\ G]\!] \Longrightarrow NSTUTINV\ (F \longrightarrow G)$
⟨*proof*⟩

lemma *nstut-eq*: $[\![NSTUTINV\ F;\ NSTUTINV\ G]\!] \Longrightarrow NSTUTINV\ (F = G)$
⟨*proof*⟩

lemma *nstut-not*: $NSTUTINV\ F \Longrightarrow NSTUTINV\ (\neg\ F)$
⟨*proof*⟩

lemma *nstut-noteq*: $[\![NSTUTINV\ F;\ NSTUTINV\ G]\!] \Longrightarrow NSTUTINV\ (F \neq G)$
⟨*proof*⟩

lemma *nstut-all*: $(\bigwedge x.\ NSTUTINV\ (F\ x)) \Longrightarrow NSTUTINV\ (\forall\ x.\ F\ x)$
⟨*proof*⟩

lemma *nstut-ex*: $(\bigwedge x.\ NSTUTINV\ (F\ x)) \Longrightarrow NSTUTINV\ (\exists\ x.\ F\ x)$
⟨*proof*⟩

lemma *nstut-const*: $NSTUTINV\ \#c$
⟨*proof*⟩

lemma *nstut-fun1*: $NSTUTINV\ X \Longrightarrow NSTUTINV\ (f\ <X>)$
⟨*proof*⟩

lemma *nstut-fun2*: $[\![NSTUTINV\ X;\ NSTUTINV\ Y]\!] \Longrightarrow NSTUTINV\ (f\ <X,Y>)$
⟨*proof*⟩

lemma *nstut-fun3*: $[\![NSTUTINV\ X;\ NSTUTINV\ Y;\ NSTUTINV\ Z]\!] \Longrightarrow NSTUTINV\ (f\ <X,Y,Z>)$
⟨*proof*⟩

lemma *nstut-fun4*: $[\![NSTUTINV\ X;\ NSTUTINV\ Y;\ NSTUTINV\ Z;\ NSTUTINV\ W]\!] \Longrightarrow NSTUTINV\ (f\ <X,Y,Z,W>)$
⟨*proof*⟩

lemma *nstut-plus*: $[\![NSTUTINV\ x;NSTUTINV\ y]\!] \Longrightarrow NSTUTINV\ (x+y)$
⟨*proof*⟩

3.5.3 Abbreviations

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

lemmas *nstut-before* = *stut-before*[*THEN stutinv-strictly-stronger*]

lemma *nstut-unch*: $NSTUTINV\ (Unchanged\ v)$
⟨*proof*⟩

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

lemma *nstut-actrans*: $NSTUTINV\ P \Longrightarrow NSTUTINV\ [P]$-$v$
⟨*proof*⟩

lemma *stut-eventually*: $STUTINV\ F \Longrightarrow STUTINV\ \Diamond F$
⟨*proof*⟩

lemma *stut-leadsto*: $[\![STUTINV\ F;\ STUTINV\ G]\!] \Longrightarrow STUTINV\ (F \leadsto G)$
⟨*proof*⟩

lemma *stut-angle-action*: $NSTUTINV\ P \Longrightarrow STUTINV\ \Diamond\langle P\rangle\text{-}v$
⟨*proof*⟩

lemma *nstut-angle-acttrans*: $NSTUTINV\ P \Longrightarrow NSTUTINV\ \langle P\rangle\text{-}v$
⟨*proof*⟩

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-actrans*
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 $|^\sim A$, for pre-formulas. Although it is semantically identical to $\vdash A$, it helps users distinguish pre-formulas from formulas in TLA* proofs.

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

syntax
-PreValid :: *lift* ⇒ *bool* (‹(|~ -)› *5*)

translations
-PreValid ⇌ *CONST PreValid*

lemma *prefD*[*dest*]: $|^\sim A \Longrightarrow w \models A$
⟨*proof*⟩

lemma *prefI*[*intro*!]: $(\bigwedge w.\ w \models A) \Longrightarrow |{\sim}\ A$
⟨*proof*⟩

⟨*ML*⟩

lemma *prefeq-reflection*: **assumes** *P1*: $|{\sim}\ x{=}y$ **shows** $(x \equiv y)$
⟨*proof*⟩

lemma *pref-True*[*simp*]: $|{\sim}\ \#True$
⟨*proof*⟩

lemma *pref-eq*: $|{\sim}\ X = Y \Longrightarrow X = Y$
⟨*proof*⟩

lemma *pref-iffI*:
assumes $|{\sim}\ F \longrightarrow G$ **and** $|{\sim}\ G \longrightarrow F$
shows $|{\sim}\ F = G$
⟨*proof*⟩

lemma *pref-iffD1*: **assumes** $|{\sim}\ F = G$ **shows** $|{\sim}\ F \longrightarrow G$
⟨*proof*⟩

lemma *pref-iffD2*: **assumes** $|{\sim}\ F = G$ **shows** $|{\sim}\ G \longrightarrow F$
⟨*proof*⟩

lemma *unl-pref-imp*:
assumes $|{\sim}\ F \longrightarrow G$ **shows** $\bigwedge w.\ w \models F \Longrightarrow w \models G$
⟨*proof*⟩

lemma *pref-imp-trans*:
assumes $|{\sim}\ F \longrightarrow G$ **and** $|{\sim}\ G \longrightarrow H$
shows $|{\sim}\ F \longrightarrow H$
⟨*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 $|{\sim}\ F \wedge$ *Unchanged id* $\longrightarrow \bigcirc F$
⟨*proof*⟩

lemma *pre-ex-unch*:
assumes *h*: *stutinv F*
shows $\exists (v{::}'a{::}world \Rightarrow 'a).\ (\ |{\sim}\ F \wedge$ *Unchanged v* $\longrightarrow \bigcirc F)$
⟨*proof*⟩

lemma *unch-pair*: |~ *Unchanged* (*x*,*y*) = (*Unchanged x* ∧ *Unchanged y*)
⟨*proof*⟩

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

lemma *angle-actrans-sem*: |~ ⟨*F*⟩-*v* = (*F* ∧ *v*$ ≠ $*v*)
⟨*proof*⟩

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

4.2 Lemmas about *after*

lemma *after-const*: |~ (#*c*)‘ = #*c*
⟨*proof*⟩

lemma *after-fun1*: |~ *f*<*x*>‘ = *f*<*x*‘>
⟨*proof*⟩

lemma *after-fun2*: |~ *f*<*x*,*y*>‘ = *f* <*x*‘,*y*‘>
⟨*proof*⟩

lemma *after-fun3*: |~ *f*<*x*,*y*,*z*>‘ = *f* <*x*‘,*y*‘,*z*‘>
⟨*proof*⟩

lemma *after-fun4*: |~ *f*<*x*,*y*,*z*,*zz*>‘ = *f* <*x*‘,*y*‘,*z*‘,*zz*‘>
⟨*proof*⟩

lemma *after-forall*: |~ (∀ *x*. *P x*)‘ = (∀ *x*. (*P x*)‘)
⟨*proof*⟩

lemma *after-exists*: |~ (∃ *x*. *P x*)‘ = (∃ *x*. (*P x*)‘)
⟨*proof*⟩

lemma *after-exists1*: |~ (∃! *x*. *P x*)‘ = (∃! *x*. (*P x*)‘)
⟨*proof*⟩

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

lemmas *all-after-unl* = *all-after*[*THEN prefD*]
lemmas *all-after-eq* = *all-after*[*THEN prefeq-reflection*]

4.3 Lemmas about *before*

lemma *before-const*: ⊢ $(#*c*) = #*c*
⟨*proof*⟩

lemma *before-fun1*: ⊢ $(*f*<*x*>) = *f* <$*x*>
⟨*proof*⟩

lemma *before-fun2*: $\vdash \$(f<x,y>) = f <\$x,\$y>$
⟨*proof*⟩

lemma *before-fun3*: $\vdash \$(f<x,y,z>) = f <\$x,\$y,\$z>$
⟨*proof*⟩

lemma *before-fun4*: $\vdash \$(f<x,y,z,zz>) = f <\$x,\$y,\$z,\$zz>$
⟨*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*: $|^\sim ((\langle F \wedge G\rangle\text{-}v) = (\langle F\rangle\text{-}v \wedge \langle G\rangle\text{-}v)$
⟨*proof*⟩

lemma *angle-actrans-disj*: $|^\sim ((\langle F \vee G\rangle\text{-}v) = (\langle F\rangle\text{-}v \vee \langle G\rangle\text{-}v)$
⟨*proof*⟩

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

lemma *pref-eq-true*: $|^\sim P \Longrightarrow |^\sim P = \#True$
⟨*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*⟩

end

5 A Proof System for TLA*

theory *Rules*
imports *PreFormulas*
begin

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

The TLA* paper [8] suggest both a *hetereogeneous* 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 hetereogenous 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 $\Box[P]$-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** $\vdash F$ **and** $\vdash F \longrightarrow G$ **shows** $\vdash G$
$\langle proof \rangle$

theorem *pmp*: **assumes** $|^\sim F$ **and** $|^\sim F \longrightarrow G$ **shows** $|^\sim G$
$\langle proof \rangle$

theorem *sq*: **assumes** $|^\sim P$ **shows** $\vdash \Box[P]$-v
$\langle proof \rangle$

theorem *pre*: **assumes** $\vdash F$ **shows** $|^\sim F$
$\langle proof \rangle$

theorem *nex*: **assumes** *h1*: $\vdash F$ **shows** $|^\sim \bigcirc F$
$\langle proof \rangle$

theorem *ax0*: $\vdash \#$ *True*
$\langle proof \rangle$

theorem *ax1*: $\vdash \Box F \longrightarrow F$
⟨*proof*⟩

theorem *ax2*: $\vdash \Box F \longrightarrow \Box[\Box F]\text{-}v$
⟨*proof*⟩

theorem *ax3*:
assumes *H*: $|^{\sim}\ F \wedge \mathit{Unchanged}\ v \longrightarrow \bigcirc F$
shows $\vdash \Box[F \longrightarrow \bigcirc F]\text{-}v \longrightarrow (F \longrightarrow \Box F)$
⟨*proof*⟩

theorem *ax4*: $\vdash \Box[P \longrightarrow Q]\text{-}v \longrightarrow (\Box[P]\text{-}v \longrightarrow \Box[Q]\text{-}v)$
⟨*proof*⟩

theorem *ax5*: $\vdash \Box[v` \neq \$v]\text{-}v$
⟨*proof*⟩

theorem *pax0*: $|^{\sim}\ \#\ \mathit{True}$
⟨*proof*⟩

theorem *pax1* [*simp-unl*]: $|^{\sim}\ (\bigcirc\neg F) = (\neg\bigcirc F)$
⟨*proof*⟩

theorem *pax2*: $|^{\sim}\ \bigcirc(F \longrightarrow G) \longrightarrow (\bigcirc F \longrightarrow \bigcirc G)$
⟨*proof*⟩

theorem *pax3*: $|^{\sim}\ \Box F \longrightarrow \bigcirc\Box F$
⟨*proof*⟩

theorem *pax4*: $|^{\sim}\ \Box[P]\text{-}v = ([P]\text{-}v \wedge \bigcirc\Box[P]\text{-}v)$
⟨*proof*⟩

theorem *pax5*: $|^{\sim}\ \bigcirc\Box F \longrightarrow \Box[\bigcirc F]\text{-}v$
⟨*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*: $\vdash (\forall x.\ \Box[F\ x]\text{-}v) = (\Box[(\forall x.\ F\ x)]\text{-}v)$
⟨*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. $F \vDash \Box F$. 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 $|^\sim F \wedge \mathit{Unchanged}\ v \longrightarrow \bigcirc F$.

theorem *alw2*:
  assumes *h1*: $\vdash F$ **and** *h2*: $|^\sim F \wedge \mathit{Unchanged}\ v \longrightarrow \bigcirc F$
  shows $\vdash \Box F$
⟨*proof*⟩

Similar theorem, assuming that F is stuttering invariant.

theorem *alw3*:
  assumes *h1*: $\vdash F$ **and** *h2*: *stutinv F*
  shows $\vdash \Box F$
⟨*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*: $\vdash F$ **shows** $\vdash \Box F$
  ⟨*proof*⟩

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

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

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

theorem *T2*[*simp-unl*]: $\vdash \Box\Box[P]\text{-}v = \Box[P]\text{-}v$
⟨*proof*⟩

theorem *T3*[*simp-unl*]: $\vdash \Box[[P]\text{-}v]\text{-}v = \Box[P]\text{-}v$
⟨*proof*⟩

theorem *M2*:
  assumes *h*: $|^\sim F \longrightarrow G$
  shows $\vdash \Box[F]\text{-}v \longrightarrow \Box[G]\text{-}v$
  ⟨*proof*⟩

theorem *N1*:
  assumes *h*: $\vdash F \longrightarrow G$
  shows $|^\sim \bigcirc F \longrightarrow \bigcirc G$

⟨*proof*⟩

theorem *T4*: $\vdash \Box[P]\text{-}v \longrightarrow \Box[[P]\text{-}v]\text{-}w$
⟨*proof*⟩

theorem *T5*: $\vdash \Box[[P]\text{-}w]\text{-}v \longrightarrow \Box[[P]\text{-}v]\text{-}w$
⟨*proof*⟩

theorem *T6*: $\vdash \Box F \longrightarrow \Box[\bigcirc F]\text{-}v$
⟨*proof*⟩

theorem *T7*:
assumes *h*: $|^{\sim}\ F \wedge \mathit{Unchanged}\ v \longrightarrow \bigcirc F$
shows $|^{\sim}\ (F \wedge \bigcirc\Box F) = \Box F$
⟨*proof*⟩

theorem *T8*: $|^{\sim}\ \bigcirc(F \wedge G) = (\bigcirc F \wedge \bigcirc G)$
⟨*proof*⟩

lemma *T9*: $|^{\sim}\ \Box[A]\text{-}v \longrightarrow [A]\text{-}v$
⟨*proof*⟩

theorem *H1*:
assumes *h1*: $\vdash \Box[P]\text{-}v$ **and** *h2*: $\vdash \Box[P \longrightarrow Q]\text{-}v$
shows $\vdash \Box[Q]\text{-}v$
⟨*proof*⟩

theorem *H2*: **assumes** *h1*: $\vdash F$ **shows** $\vdash \Box[F]\text{-}v$
⟨*proof*⟩

theorem *H3*:
assumes *h1*: $\vdash \Box[P \longrightarrow Q]\text{-}v$ **and** *h2*: $\vdash \Box[Q \longrightarrow R]\text{-}v$
shows $\vdash \Box[P \longrightarrow R]\text{-}v$
⟨*proof*⟩

theorem *H4*: $\vdash \Box[[P]\text{-}v \longrightarrow P]\text{-}v$
⟨*proof*⟩

theorem *H5*: $\vdash \Box[\Box F \longrightarrow \bigcirc\Box F]\text{-}v$
⟨*proof*⟩

5.3 Some other useful derived theorems

theorem *P1*: $|^{\sim}\ \Box F \longrightarrow \bigcirc F$
⟨*proof*⟩

theorem *P2*: $|^{\sim}\ \Box F \longrightarrow F \wedge \bigcirc F$
⟨*proof*⟩

theorem $P4$: $\vdash \Box F \longrightarrow \Box[F]\text{-}v$
⟨*proof*⟩

theorem $P5$: $\vdash \Box[P]\text{-}v \longrightarrow \Box[\Box[P]\text{-}v]\text{-}w$
⟨*proof*⟩

theorem $M0$: $\vdash \Box F \longrightarrow \Box[F \longrightarrow \bigcirc F]\text{-}v$
⟨*proof*⟩

theorem $M1$: $\vdash \Box F \longrightarrow \Box[F \wedge \bigcirc F]\text{-}v$
⟨*proof*⟩

theorem $M3$: **assumes** h: $\vdash F$ **shows** $\vdash \Box[\bigcirc F]\text{-}v$
⟨*proof*⟩

lemma $M4$: $\vdash \Box[\bigcirc(F \wedge G) = (\bigcirc F \wedge \bigcirc G)]\text{-}v$
⟨*proof*⟩

theorem $M5$: $\vdash \Box[\ \Box[P]\text{-}v \longrightarrow \bigcirc\Box[P]\text{-}v\ ]\text{-}w$
⟨*proof*⟩

theorem $M6$: $\vdash \Box[F \wedge G]\text{-}v \longrightarrow \Box[F]\text{-}v \wedge \Box[G]\text{-}v$
⟨*proof*⟩

theorem $M7$: $\vdash \Box[F]\text{-}v \wedge \Box[G]\text{-}v \longrightarrow \Box[F \wedge G]\text{-}v$
⟨*proof*⟩

theorem $M8$: $\vdash \Box[F \wedge G]\text{-}v = (\Box[F]\text{-}v \wedge \Box[G]\text{-}v)$
⟨*proof*⟩

theorem $M9$: $|^\sim \Box F \longrightarrow F \wedge \bigcirc\Box F$
⟨*proof*⟩

theorem $M10$:
assumes h: $|^\sim F \wedge \textit{Unchanged } v \longrightarrow \bigcirc F$
shows $|^\sim F \wedge \bigcirc\Box F \longrightarrow \Box F$
⟨*proof*⟩

theorem $M11$:
assumes h: $|^\sim [A]\text{-}f \longrightarrow [B]\text{-}g$
shows $\vdash \Box[A]\text{-}f \longrightarrow \Box[B]\text{-}g$
⟨*proof*⟩

theorem $M12$: $\vdash (\Box[A]\text{-}f \wedge \Box[B]\text{-}g) = \Box[[A]\text{-}f \wedge [B]\text{-}g]\text{-}(f,g)$
⟨*proof*⟩

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

lemmas $STL1 = alw$

STL2 and STL3 have also already been derived.

lemmas *STL2* = *ax1*

lemmas *STL3* = *T1*

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

theorem *STL4-2*:
  assumes *h1*: $\vdash F \longrightarrow G$ **and** *h2*: $|^{\sim}\ G \wedge$ *Unchanged v* $\longrightarrow \bigcirc G$
  shows $\vdash \Box F \longrightarrow \Box G$
⟨*proof*⟩

lemma *STL4-3*:
  assumes *h1*: $\vdash F \longrightarrow G$ **and** *h2*: *STUTINV G*
  shows $\vdash \Box F \longrightarrow \Box G$
⟨*proof*⟩

Of course, the original rule can be derived semantically

lemma *STL4*: **assumes** *h*: $\vdash F \longrightarrow G$ **shows** $\vdash \Box F \longrightarrow \Box G$
  ⟨*proof*⟩

Dual rule for $\Diamond$

lemma *STL4-eve*: **assumes** *h*: $\vdash F \longrightarrow G$ **shows** $\vdash \Diamond F \longrightarrow \Diamond G$
  ⟨*proof*⟩

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

theorem *STL5-2*:
  assumes *h1*: $|^{\sim}\ F \wedge$ *Unchanged f* $\longrightarrow \bigcirc F$
    and *h2*: $|^{\sim}\ G \wedge$ *Unchanged g* $\longrightarrow \bigcirc G$
  shows $\vdash \Box(F \wedge G) = (\Box F \wedge \Box G)$
⟨*proof*⟩

theorem *STL5-21*:
  assumes *h1*: *stutinv F* **and** *h2*: *stutinv G*
  shows $\vdash \Box(F \wedge G) = (\Box F \wedge \Box G)$
  ⟨*proof*⟩

We also derive STL5 semantically.

lemma *STL5*: $\vdash \Box(F \wedge G) = (\Box F \wedge \Box G)$
  ⟨*proof*⟩

Elimination rule corresponding to *STL5* in unlifted form.

lemma *box-conjE*:
  assumes $s \models \Box F$ **and** $s \models \Box G$ **and** $s \models \Box(F \wedge G) \Longrightarrow P$
  shows *P*
  ⟨*proof*⟩

lemma *box-thin*:
assumes *h1*: $s \models \Box F$ **and** *h2*: *PROP W*
shows *PROP W*
⟨*proof*⟩

Finally, we derive STL6 (only semantically)

lemma *STL6*: $\vdash \Diamond\Box(F \wedge G) = (\Diamond\Box F \wedge \Diamond\Box G)$
⟨*proof*⟩

lemma *MM0*: $\vdash \Box(F \longrightarrow G) \longrightarrow \Box F \longrightarrow \Box G$
⟨*proof*⟩

lemma *MM1*: **assumes** *h*: $\vdash F = G$ **shows** $\vdash \Box F = \Box G$
⟨*proof*⟩

theorem *MM2*: $\vdash \Box A \wedge \Box(B \longrightarrow C) \longrightarrow \Box(A \wedge B \longrightarrow C)$
⟨*proof*⟩

theorem *MM3*: $\vdash \Box\neg A \longrightarrow \Box(A \wedge B \longrightarrow C)$
⟨*proof*⟩

theorem *MM4*[*simp-unl*]: $\vdash \Box\#F = \#F$
⟨*proof*⟩

theorem *MM4b*[*simp-unl*]: $\vdash \Box\neg\#F = \neg\#F$
⟨*proof*⟩

theorem *MM5*: $\vdash \Box F \vee \Box G \longrightarrow \Box(F \vee G)$
⟨*proof*⟩

theorem *MM6*: $\vdash \Box F \vee \Box G \longrightarrow \Box(\Box F \vee \Box G)$
⟨*proof*⟩

lemma *MM10*:
assumes *h*: $|^{\sim}\ F = G$ **shows** $\vdash \Box[F]\text{-}v = \Box[G]\text{-}v$
⟨*proof*⟩

lemma *MM9*:
assumes *h*: $\vdash F = G$ **shows** $\vdash \Box[F]\text{-}v = \Box[G]\text{-}v$
⟨*proof*⟩

theorem *MM11*: $\vdash \Box[\neg(P \wedge Q)]\text{-}v \longrightarrow \Box[P]\text{-}v \longrightarrow \Box[P \wedge \neg Q]\text{-}v$
⟨*proof*⟩

theorem *MM12*[*simp-unl*]: $\vdash \Box[\Box[P]\text{-}v]\text{-}v = \Box[P]\text{-}v$
⟨*proof*⟩

5.4 Theorems about the eventually operator

theorem *dualization*:
$\vdash \neg\Box F = \Diamond\neg F$
$\vdash \neg\Diamond F = \Box\neg F$
$\vdash \neg\Box[A]\text{-}v = \Diamond\langle\neg A\rangle\text{-}v$
$\vdash \neg\Diamond\langle A\rangle\text{-}v = \Box[\neg A]\text{-}v$
⟨*proof*⟩

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

theorem *E1*: $\vdash \Diamond(F \vee G) = (\Diamond F \vee \Diamond G)$
⟨*proof*⟩

theorem *E3*: $\vdash F \longrightarrow \Diamond F$
⟨*proof*⟩

theorem *E4*: $\vdash \Box F \longrightarrow \Diamond F$
⟨*proof*⟩

theorem *E5*: $\vdash \Box F \longrightarrow \Box\Diamond F$
⟨*proof*⟩

theorem *E6*: $\vdash \Box F \longrightarrow \Diamond\Box F$
⟨*proof*⟩

theorem *E7*:
assumes *h*: $|^\sim \neg F \wedge Unchanged\ v \longrightarrow \bigcirc\neg F$
shows $|^\sim \Diamond F \longrightarrow F \vee \bigcirc\Diamond F$
⟨*proof*⟩

theorem *E8*: $\vdash \Diamond(F \longrightarrow G) \longrightarrow \Box F \longrightarrow \Diamond G$
⟨*proof*⟩

theorem *E9*: $\vdash \Box(F \longrightarrow G) \longrightarrow \Diamond F \longrightarrow \Diamond G$
⟨*proof*⟩

theorem *E10*[*simp-unl*]: $\vdash \Diamond\Diamond F = \Diamond F$
⟨*proof*⟩

theorem *E22*:
assumes *h*: $\vdash F = G$
shows $\vdash \Diamond F = \Diamond G$
⟨*proof*⟩

theorem *E15*[*simp-unl*]: $\vdash \Diamond\#F = \#F$
⟨*proof*⟩

theorem *E15b*[*simp-unl*]: $\vdash \Diamond\neg\#F = \neg\#F$

$\langle proof \rangle$

theorem *E16*: $\vdash \Diamond\Box F \longrightarrow \Diamond F$
$\langle proof \rangle$

An action version of STL6

lemma *STL6-act*: $\vdash \Diamond(\Box[F]\text{-}v \wedge \Box[G]\text{-}w) = (\Diamond\Box[F]\text{-}v \wedge \Diamond\Box[G]\text{-}w)$
$\langle proof \rangle$

lemma *SE1*: $\vdash \Box F \wedge \Diamond G \longrightarrow \Diamond(\Box F \wedge G)$
$\langle proof \rangle$

lemma *SE2*: $\vdash \Box F \wedge \Diamond G \longrightarrow \Diamond(F \wedge G)$
$\langle proof \rangle$

lemma *SE3*: $\vdash \Box F \wedge \Diamond G \longrightarrow \Diamond(G \wedge F)$
$\langle proof \rangle$

lemma *SE4*:
assumes *h1*: $s \models \Box F$ **and** *h2*: $s \models \Diamond G$ **and** *h3*: $\vdash \Box F \wedge G \longrightarrow H$
shows $s \models \Diamond H$
$\langle proof \rangle$

theorem *E17*: $\vdash \Box\Diamond\Box F \longrightarrow \Box\Diamond F$
$\langle proof \rangle$

theorem *E18*: $\vdash \Box\Diamond\Box F \longrightarrow \Diamond\Box F$
$\langle proof \rangle$

theorem *E19*: $\vdash \Diamond\Box F \longrightarrow \Box\Diamond\Box F$
$\langle proof \rangle$

theorem *E20*: $\vdash \Diamond\Box F \longrightarrow \Box\Diamond F$
$\langle proof \rangle$

theorem *E21*[*simp-unl*]: $\vdash \Box\Diamond\Box F = \Diamond\Box F$
$\langle proof \rangle$

theorem *E27*[*simp-unl*]: $\vdash \Diamond\Box\Diamond F = \Box\Diamond F$
$\langle proof \rangle$

lemma *E28*: $\vdash \Diamond\Box F \wedge \Box\Diamond G \longrightarrow \Box\Diamond(F \wedge G)$
$\langle proof \rangle$

lemma *E23*: $|^{\sim} \bigcirc F \longrightarrow \Diamond F$
$\langle proof \rangle$

lemma *E24*: $\vdash \Diamond\Box Q \longrightarrow \Box[\Diamond Q]\text{-}v$
$\langle proof \rangle$

lemma $E25$: $\vdash \Diamond\langle A\rangle\text{-}v \longrightarrow \Diamond A$
$\langle proof \rangle$

lemma $E26$: $\vdash \Box\Diamond\langle A\rangle\text{-}v \longrightarrow \Box\Diamond A$
$\langle proof \rangle$

lemma $allBox$: $(s \models \Box(\forall x.\ F\ x)) = (\forall x.\ s \models \Box(F\ x))$
$\langle proof \rangle$

lemma $E29$: $|^\sim \bigcirc\Diamond F \longrightarrow \Diamond F$
$\langle proof \rangle$

lemma $E30$:
assumes $h1$: $\vdash F \longrightarrow \Box F$ **and** $h2$: $\vdash \Diamond F$
shows $\vdash \Diamond\Box F$
$\langle proof \rangle$

lemma $E31$: $\vdash \Box(F \longrightarrow \Box F) \land \Diamond F \longrightarrow \Diamond\Box F$
$\langle proof \rangle$

lemma $allActBox$: $(s \models \Box[(\forall x.\ F\ x)]\text{-}v) = (\forall x.\ s \models \Box[(F\ x)]\text{-}v)$
$\langle proof \rangle$

theorem $exEE$: $\vdash (\exists x.\ \Diamond(F\ x)) = \Diamond(\exists x.\ F\ x)$
$\langle proof \rangle$

theorem $exActE$: $\vdash (\exists x.\ \Diamond\langle F\ x\rangle\text{-}v) = \Diamond\langle(\exists x.\ F\ x)\rangle\text{-}v$
$\langle proof \rangle$

5.5 Theorems about the leadsto operator

theorem $LT1$: $\vdash F \leadsto F$
$\langle proof \rangle$

theorem $LT2$: **assumes** h: $\vdash F \longrightarrow G$ **shows** $\vdash F \longrightarrow \Diamond G$
$\langle proof \rangle$

theorem $LT3$: **assumes** h: $\vdash F \longrightarrow G$ **shows** $\vdash F \leadsto G$
$\langle proof \rangle$

theorem $LT4$: $\vdash F \longrightarrow (F \leadsto G) \longrightarrow \Diamond G$
$\langle proof \rangle$

theorem $LT5$: $\vdash \Box(F \longrightarrow \Diamond G) \longrightarrow \Diamond F \longrightarrow \Diamond G$
$\langle proof \rangle$

theorem $LT6$: $\vdash \Diamond F \longrightarrow (F \leadsto G) \longrightarrow \Diamond G$
$\langle proof \rangle$

theorem *LT9*[*simp-unl*]: $\vdash \Box(F \leadsto G) = (F \leadsto G)$
⟨*proof*⟩

theorem *LT7*: $\vdash \Box\Diamond F \longrightarrow (F \leadsto G) \longrightarrow \Box\Diamond G$
⟨*proof*⟩

theorem *LT8*: $\vdash \Box\Diamond G \longrightarrow (F \leadsto G)$
⟨*proof*⟩

theorem *LT13*: $\vdash (F \leadsto G) \longrightarrow (G \leadsto H) \longrightarrow (F \leadsto H)$
⟨*proof*⟩

theorem *LT11*: $\vdash (F \leadsto G) \longrightarrow (F \leadsto (G \lor H))$
⟨*proof*⟩

theorem *LT12*: $\vdash (F \leadsto H) \longrightarrow (F \leadsto (G \lor H))$
⟨*proof*⟩

theorem *LT14*: $\vdash ((F \lor G) \leadsto H) \longrightarrow (F \leadsto H)$
⟨*proof*⟩

theorem *LT15*: $\vdash ((F \lor G) \leadsto H) \longrightarrow (G \leadsto H)$
⟨*proof*⟩

theorem *LT16*: $\vdash (F \leadsto H) \longrightarrow (G \leadsto H) \longrightarrow ((F \lor G) \leadsto H)$
⟨*proof*⟩

theorem *LT17*: $\vdash ((F \lor G) \leadsto H) = ((F \leadsto H) \land (G \leadsto H))$
⟨*proof*⟩

theorem *LT10*:
assumes *h*: $\vdash (F \land \neg G) \leadsto G$
shows $\vdash F \leadsto G$
⟨*proof*⟩

theorem *LT18*: $\vdash (A \leadsto (B \lor C)) \longrightarrow (B \leadsto D) \longrightarrow (C \leadsto D) \longrightarrow (A \leadsto D)$
⟨*proof*⟩

theorem *LT19*: $\vdash (A \leadsto (D \lor B)) \longrightarrow (B \leadsto D) \longrightarrow (A \leadsto D)$
⟨*proof*⟩

theorem *LT20*: $\vdash (A \leadsto (B \lor D)) \longrightarrow (B \leadsto D) \longrightarrow (A \leadsto D)$
⟨*proof*⟩

theorem *LT21*: $\vdash ((\exists x.\ F\ x) \leadsto G) = (\forall x.\ (F\ x \leadsto G))$
⟨*proof*⟩

theorem *LT22*: $\vdash (F \leadsto (G \lor H)) \longrightarrow \Box\neg G \longrightarrow (F \leadsto H)$

⟨*proof*⟩

lemma *LT23*: $|^{\sim}$ $(P \longrightarrow \bigcirc Q) \longrightarrow (P \longrightarrow \Diamond Q)$
⟨*proof*⟩

theorem *LT24*: $\vdash \Box I \longrightarrow ((P \wedge I) \leadsto Q) \longrightarrow P \leadsto Q$
⟨*proof*⟩

theorem *LT25*[*simp-unl*]: $\vdash (F \leadsto \#False) = \Box\neg F$
⟨*proof*⟩

lemma *LT28*:
assumes *h*: $|^{\sim}$ $P \longrightarrow \bigcirc P \vee \bigcirc Q$
shows $|^{\sim}$ $(P \longrightarrow \bigcirc P) \vee \Diamond Q$
⟨*proof*⟩

lemma *LT29*:
assumes *h1*: $|^{\sim}$ $P \longrightarrow \bigcirc P \vee \bigcirc Q$ **and** *h2*: $|^{\sim}$ $P \wedge Unchanged\ v \longrightarrow \bigcirc P$
shows $\vdash P \longrightarrow \Box P \vee \Diamond Q$
⟨*proof*⟩

lemma *LT30*:
assumes *h*: $|^{\sim}$ $P \wedge N \longrightarrow \bigcirc P \vee \bigcirc Q$
shows $|^{\sim}$ $N \longrightarrow (P \longrightarrow \bigcirc P) \vee \Diamond Q$
⟨*proof*⟩

lemma *LT31*:
assumes *h1*: $|^{\sim}$ $P \wedge N \longrightarrow \bigcirc P \vee \bigcirc Q$ **and** *h2*: $|^{\sim}$ $P \wedge Unchanged\ v \longrightarrow \bigcirc P$
shows$\vdash \Box N \longrightarrow P \longrightarrow \Box P \vee \Diamond Q$
⟨*proof*⟩

lemma *LT33*: $\vdash ((\#P \wedge F) \leadsto G) = (\#P \longrightarrow (F \leadsto G))$
⟨*proof*⟩

lemma *AA1*: $\vdash \Box[\#False]\text{-}v \longrightarrow \neg\Diamond\langle Q\rangle\text{-}v$
⟨*proof*⟩

lemma *AA2*: $\vdash \Box[P]\text{-}v \wedge \Diamond\langle Q\rangle\text{-}v \longrightarrow \Diamond\langle P \wedge Q\rangle\text{-}v$
⟨*proof*⟩

lemma *AA3*: $\vdash \Box P \wedge \Box[P \longrightarrow Q]\text{-}v \wedge \Diamond\langle A\rangle\text{-}v \longrightarrow \Diamond Q$
⟨*proof*⟩

lemma *AA4*: $\vdash \Diamond\langle\langle A\rangle\text{-}v\rangle\text{-}w \longrightarrow \Diamond\langle\langle A\rangle\text{-}w\rangle\text{-}v$
⟨*proof*⟩

lemma *AA7*: **assumes** *h*: $|^{\sim}$ $F \longrightarrow G$ **shows** $\vdash \Diamond\langle F\rangle\text{-}v \longrightarrow \Diamond\langle G\rangle\text{-}v$
⟨*proof*⟩

lemma $AA6$: $\vdash \Box[P \longrightarrow Q]\text{-}v \wedge \Diamond\langle P\rangle\text{-}v \longrightarrow \Diamond\langle Q\rangle\text{-}v$
⟨*proof*⟩

lemma $AA8$: $\vdash \Box[P]\text{-}v \wedge \Diamond\langle A\rangle\text{-}v \longrightarrow \Diamond\langle\Box[P]\text{-}v \wedge A\rangle\text{-}v$
⟨*proof*⟩

lemma $AA9$: $\vdash \Box[P]\text{-}v \wedge \Diamond\langle A\rangle\text{-}v \longrightarrow \Diamond\langle[P]\text{-}v \wedge A\rangle\text{-}v$
⟨*proof*⟩

lemma $AA10$: $\vdash \neg(\Box[P]\text{-}v \wedge \Diamond\langle\neg P\rangle\text{-}v)$
⟨*proof*⟩

lemma $AA11$: $\vdash \neg\Diamond\langle v\$ = \$v\rangle\text{-}v$
⟨*proof*⟩

lemma $AA15$: $\vdash \Diamond\langle P \wedge Q\rangle\text{-}v \longrightarrow \Diamond\langle P\rangle\text{-}v$
⟨*proof*⟩

lemma $AA16$: $\vdash \Diamond\langle P \wedge Q\rangle\text{-}v \longrightarrow \Diamond\langle Q\rangle\text{-}v$
⟨*proof*⟩

lemma $AA13$: $\vdash \Diamond\langle P\rangle\text{-}v \longrightarrow \Diamond\langle v\$ \neq \$v\rangle\text{-}v$
⟨*proof*⟩

lemma $AA14$: $\vdash \Diamond\langle P \vee Q\rangle\text{-}v = (\Diamond\langle P\rangle\text{-}v \vee \Diamond\langle Q\rangle\text{-}v)$
⟨*proof*⟩

lemma $AA17$: $\vdash \Diamond\langle[P]\text{-}v \wedge A\rangle\text{-}v \longrightarrow \Diamond\langle P \wedge A\rangle\text{-}v$
⟨*proof*⟩

lemma $AA19$: $\vdash \Box P \wedge \Diamond\langle A\rangle\text{-}v \longrightarrow \Diamond\langle P \wedge A\rangle\text{-}v$
⟨*proof*⟩

lemma $AA20$:
assumes $h1$: $|{\sim}\ P \longrightarrow \bigcirc P \vee \bigcirc Q$
and $h2$: $|{\sim}\ P \wedge A \longrightarrow \bigcirc Q$
and $h3$: $|{\sim}\ P \wedge \mathit{Unchanged}\ w \longrightarrow \bigcirc P$
shows $\vdash \Box(\Box P \longrightarrow \Diamond\langle A\rangle\text{-}v) \longrightarrow (P \leadsto Q)$
⟨*proof*⟩

lemma $AA21$: $|{\sim}\ \Diamond\langle\bigcirc F\rangle\text{-}v \longrightarrow \bigcirc\Diamond F$
⟨*proof*⟩

theorem $AA24$[*simp-unl*]: $\vdash \Diamond\langle\langle P\rangle\text{-}f\rangle\text{-}f = \Diamond\langle P\rangle\text{-}f$
⟨*proof*⟩

lemma $AA22$:
assumes $h1$: $|{\sim}\ P \wedge N \longrightarrow \bigcirc P \vee \bigcirc Q$
and $h2$: $|{\sim}\ P \wedge N \wedge \langle A\rangle\text{-}v \longrightarrow \bigcirc Q$

and *h3*: $|^{\sim}$ $P \wedge Unchanged\ w \longrightarrow \bigcirc P$
shows $\vdash \Box N \wedge \Box(\Box P \longrightarrow \Diamond\langle A\rangle\text{-}v) \longrightarrow (P \leadsto Q)$
⟨*proof*⟩

lemma *AA23*:
assumes $|^{\sim}$ $P \wedge N \longrightarrow \bigcirc P \vee \bigcirc Q$
and $|^{\sim}$ $P \wedge N \wedge \langle A\rangle\text{-}v \longrightarrow \bigcirc Q$
and $|^{\sim}$ $P \wedge Unchanged\ w \longrightarrow \bigcirc P$
shows $\vdash \Box N \wedge \Box\Diamond\langle A\rangle\text{-}v \longrightarrow (P \leadsto Q)$
⟨*proof*⟩

lemma *AA25*:
assumes *h*: $|^{\sim}$ $\langle P\rangle\text{-}v \longrightarrow \langle Q\rangle\text{-}w$
shows $\vdash \Diamond\langle P\rangle\text{-}v \longrightarrow \Diamond\langle Q\rangle\text{-}w$
⟨*proof*⟩

lemma *AA26*:
assumes *h*: $|^{\sim}$ $\langle A\rangle\text{-}v = \langle B\rangle\text{-}w$
shows $\vdash \Diamond\langle A\rangle\text{-}v = \Diamond\langle B\rangle\text{-}w$
⟨*proof*⟩

theorem *AA28*[*simp-unl*]: $\vdash \Diamond\Diamond\langle A\rangle\text{-}v = \Diamond\langle A\rangle\text{-}v$
⟨*proof*⟩

theorem *AA29*: $\vdash \Box[N]\text{-}v \wedge \Box\Diamond\langle A\rangle\text{-}v \longrightarrow \Box\Diamond\langle N \wedge A\rangle\text{-}v$
⟨*proof*⟩

theorem *AA30*[*simp-unl*]: $\vdash \Diamond\langle\Diamond\langle P\rangle\text{-}f\rangle\text{-}f = \Diamond\langle P\rangle\text{-}f$
⟨*proof*⟩

theorem *AA31*: $\vdash \Diamond\langle\bigcirc F\rangle\text{-}v \longrightarrow \Diamond F$
⟨*proof*⟩

lemma *AA32*[*simp-unl*]: $\vdash \Box\Diamond\Box[A]\text{-}v = \Diamond\Box[A]\text{-}v$
⟨*proof*⟩

lemma *AA33*[*simp-unl*]: $\vdash \Diamond\Box\Diamond\langle A\rangle\text{-}v = \Box\Diamond\langle A\rangle\text{-}v$
⟨*proof*⟩

5.6 Lemmas about the next operator

lemma *N2*: **assumes** *h*: $\vdash F = G$ **shows** $|^{\sim}$ $\bigcirc F = \bigcirc G$
⟨*proof*⟩

lemmas *next-and* = *T8*

lemma *next-or*: $|^{\sim}$ $\bigcirc(F \vee G) = (\bigcirc F \vee \bigcirc G)$
⟨*proof*⟩

lemma *next-imp*: $|^\sim \bigcirc(F \longrightarrow G) = (\bigcirc F \longrightarrow \bigcirc G)$
⟨*proof*⟩

lemmas *next-not* = *pax1*

lemma *next-eq*: $|^\sim \bigcirc(F = G) = (\bigcirc F = \bigcirc G)$
⟨*proof*⟩

lemma *next-noteq*: $|^\sim \bigcirc(F \neq G) = (\bigcirc F \neq \bigcirc G)$
⟨*proof*⟩

lemma *next-const*[*simp-unl*]: $|^\sim \bigcirc\#P = \#P$
⟨*proof*⟩

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

lemma *next-fun1*: $|^\sim \bigcirc f{<}x{>} = f{<}\bigcirc x{>}$
⟨*proof*⟩

lemma *next-fun2*: $|^\sim \bigcirc f{<}x,y{>} = f{<}\bigcirc x,\bigcirc y{>}$
⟨*proof*⟩

lemma *next-fun3*: $|^\sim \bigcirc f{<}x,y,z{>} = f{<}\bigcirc x,\bigcirc y,\bigcirc z{>}$
⟨*proof*⟩

lemma *next-fun4*: $|^\sim \bigcirc f{<}x,y,z,zz{>} = f{<}\bigcirc x,\bigcirc y,\bigcirc z,\bigcirc zz{>}$
⟨*proof*⟩

lemma *next-forall*: $|^\sim \bigcirc(\forall\ x.\ P\ x) = (\forall\ x.\ \bigcirc\ P\ x)$
⟨*proof*⟩

lemma *next-exists*: $|^\sim \bigcirc(\exists\ x.\ P\ x) = (\exists\ x.\ \bigcirc\ P\ x)$
⟨*proof*⟩

lemma *next-exists1*: $|^\sim \bigcirc(\exists!\ x.\ P\ x) = (\exists!\ x.\ \bigcirc\ P\ x)$
⟨*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*

lemma *t1*: $|^\sim \bigcirc\$x = x\$$

$\langle proof \rangle$

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

lemma *next-eventually*:
  assumes *h*: *stutinv F*
  shows $|^\sim \Diamond F \longrightarrow \neg F \longrightarrow \bigcirc\Diamond F$
$\langle proof \rangle$

lemma *next-action*: $|^\sim \Box[P]\text{-}v \longrightarrow \bigcirc\Box[P]\text{-}v$
  $\langle proof \rangle$

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 $|^\sim$ is used where appropriate.

theorem *TLA1*:
  assumes *H*: $|^\sim P \wedge \mathit{Unchanged}\ f \longrightarrow \bigcirc P$
  shows $\vdash \Box P = (P \wedge \Box[P \longrightarrow \bigcirc P]\text{-}f)$
$\langle proof \rangle$

theorem *TLA2*:
  assumes *h1*: $\vdash P \longrightarrow Q$
    and *h2*: $|^\sim P \wedge \bigcirc P \wedge [A]\text{-}f \longrightarrow [B]\text{-}g$
  shows $\vdash \Box P \wedge \Box[A]\text{-}f \longrightarrow \Box Q \wedge \Box[B]\text{-}g$
$\langle proof \rangle$

theorem *INV1*:
  assumes *H*: $|^\sim I \wedge [N]\text{-}f \longrightarrow \bigcirc I$
  shows $\vdash I \wedge \Box[N]\text{-}f \longrightarrow \Box I$
$\langle proof \rangle$

theorem *INV2*: $\vdash \Box I \longrightarrow \Box[N]\text{-}f = \Box[N \wedge I \wedge \bigcirc I]\text{-}f$
$\langle proof \rangle$

lemma *R1*:
  assumes *H*: $|^\sim \mathit{Unchanged}\ w \longrightarrow \mathit{Unchanged}\ v$
  shows $\vdash \Box[F]\text{-}w \longrightarrow \Box[F]\text{-}v$
$\langle proof \rangle$

theorem *invmono*:
  assumes *h1*: $\vdash I \longrightarrow P$
    and *h2*: $|^\sim P \wedge [N]\text{-}f \longrightarrow \bigcirc P$
  shows $\vdash I \wedge \Box[N]\text{-}f \longrightarrow \Box P$
  $\langle proof \rangle$

theorem *preimpsplit*:

assumes $|\!\sim I \wedge N \longrightarrow Q$
and $|\!\sim I \wedge \mathit{Unchanged}\ v \longrightarrow Q$
shows $|\!\sim I \wedge [N]\text{-}v \longrightarrow Q$
⟨*proof*⟩

theorem *refinement1*:
assumes *h1*: $\vdash P \longrightarrow Q$
and *h2*: $|\!\sim I \wedge \bigcirc I \wedge [A]\text{-}f \longrightarrow [B]\text{-}g$
shows $\vdash P \wedge \Box I \wedge \Box[A]\text{-}f \longrightarrow Q \wedge \Box[B]\text{-}g$
⟨*proof*⟩

theorem *inv-join*:
assumes $\vdash P \longrightarrow \Box Q$ **and** $\vdash P \longrightarrow \Box R$
shows $\vdash P \longrightarrow \Box(Q \wedge R)$
⟨*proof*⟩

lemma *inv-cases*: $\vdash \Box(A \longrightarrow B) \wedge \Box(\neg A \longrightarrow B) \longrightarrow \Box B$
⟨*proof*⟩

end

6 Liveness

theory *Liveness*
imports *Rules*
begin

This theory derives proof rules for liveness properties.

definition *enabled* :: *'a formula* ⇒ *'a formula*
where *enabled F* ≡ λ *s*. ∃ *t*. ((*first s*) ## *t*) ⊨ *F*

syntax *-Enabled* :: *lift* ⇒ *lift* (‹(*Enabled* -)› [*90*] *90*)

translations *-Enabled* ⇌ *CONST enabled*

definition *WeakF* :: (*'a*::*world*) *formula* ⇒ (*'a*,*'b*) *stfun* ⇒ *'a formula*
where *WeakF F v* ≡ *TEMP* ◇□*Enabled* ⟨*F*⟩-*v* ⟶ □◇⟨*F*⟩-*v*

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

Lamport's TLA defines the above notions for actions. In TLA*, (pre-)formulas generalise TLA's actions and the above definition is the natural generalisation of enabledness to pre-formulas. In particular, we have chosen to define *enabled* such that it yields itself a temporal formula, although its value really just depends on the first state of the sequence it is evaluated over. Then, the definitions of weak and strong fairness are exactly as in TLA.

syntax
-*WF* :: [*lift*,*lift*] ⇒ *lift* (‹(*WF*'(-')'-(-))› [*20*,*1000*] *90*)
-*SF* :: [*lift*,*lift*] ⇒ *lift* (‹(*SF*'(-')'-(-))› [*20*,*1000*] *90*)
-*WFsp* :: [*lift*,*lift*] ⇒ *lift* (‹(*WF* '(-')'-(-))› [*20*,*1000*] *90*)
-*SFsp* :: [*lift*,*lift*] ⇒ *lift* (‹(*SF* '(-')'-(-))› [*20*,*1000*] *90*)

translations
-*WF* ⇌ *CONST WeakF*
-*SF* ⇌ *CONST StrongF*
-*WFsp* ⇀ *CONST WeakF*
-*SFsp* ⇀ *CONST StrongF*

6.1 Properties of -*Enabled*

theorem *enabledI*: ⊢ *F* ⟶ *Enabled F*
⟨*proof*⟩

theorem *enabledE*:
assumes *s* ⊨ *Enabled F* **and** ⋀*u*. (*first s* ## *u*) ⊨ *F* ⟹ *Q*
shows *Q*
⟨*proof*⟩

lemma *enabled-mono*:
assumes *w* ⊨ *Enabled F* **and** ⊢ *F* ⟶ *G*
shows *w* ⊨ *Enabled G*
⟨*proof*⟩

lemma *Enabled-disj1*: ⊢ *Enabled F* ⟶ *Enabled* (*F* ∨ *G*)
⟨*proof*⟩

lemma *Enabled-disj2*: ⊢ *Enabled F* ⟶ *Enabled* (*G* ∨ *F*)
⟨*proof*⟩

lemma *Enabled-conj1*: ⊢ *Enabled* (*F* ∧ *G*) ⟶ *Enabled F*
⟨*proof*⟩

lemma *Enabled-conj2*: ⊢ *Enabled* (*G* ∧ *F*) ⟶ *Enabled F*
⟨*proof*⟩

lemma *Enabled-disjD*: ⊢ *Enabled* (*F* ∨ *G*) ⟶ *Enabled F* ∨ *Enabled G*
⟨*proof*⟩

lemma *Enabled-disj*: ⊢ *Enabled* (*F* ∨ *G*) = (*Enabled F* ∨ *Enabled G*)
⟨*proof*⟩

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

lemma *Enabled-ex*: ⊢ *Enabled* (∃ *x*. *F x*) = (∃ *x*. *Enabled* (*F x*))
⟨*proof*⟩

6.2 Fairness Properties

lemma *WF-alt*: $\vdash WF(A)\text{-}v = (\Box\Diamond\neg Enabled\ \langle A\rangle\text{-}v \vee \Box\Diamond\langle A\rangle\text{-}v)$
⟨*proof*⟩

lemma *SF-alt*: $\vdash SF(A)\text{-}v = (\Diamond\Box\neg Enabled\ \langle A\rangle\text{-}v \vee \Box\Diamond\langle A\rangle\text{-}v)$
⟨*proof*⟩

lemma *alwaysWFI*: $\vdash WF(A)\text{-}v \longrightarrow \Box WF(A)\text{-}v$
⟨*proof*⟩

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

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

lemma *alwaysSFI*: $\vdash SF(A)\text{-}v \longrightarrow \Box SF(A)\text{-}v$
⟨*proof*⟩

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

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

theorem *SF-imp-WF*: $\vdash SF\ (A)\text{-}v \longrightarrow WF\ (A)\text{-}v$
⟨*proof*⟩

lemma *enabled-WFSF*: $\vdash \Box Enabled\ \langle F\rangle\text{-}v \longrightarrow (WF(F)\text{-}v = SF(F)\text{-}v)$
⟨*proof*⟩

theorem *WF1-general*:
assumes *h1*: $|\sim\ P \wedge N \longrightarrow \bigcirc P \vee \bigcirc Q$
and *h2*: $|\sim\ P \wedge N \wedge \langle A\rangle\text{-}v \longrightarrow \bigcirc Q$
and *h3*: $\vdash P \wedge N \longrightarrow Enabled\ \langle A\rangle\text{-}v$
and *h4*: $|\sim\ P \wedge Unchanged\ w \longrightarrow \bigcirc P$
shows $\vdash \Box N \wedge WF(A)\text{-}v \longrightarrow (P \leadsto Q)$
⟨*proof*⟩

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

theorem *WF1*:
assumes *h1*: $|\sim\ P \wedge [N]\text{-}v \longrightarrow \bigcirc P \vee \bigcirc Q$
and *h2*: $|\sim\ P \wedge \langle N \wedge A\rangle\text{-}v \longrightarrow \bigcirc Q$
and *h3*: $\vdash P \longrightarrow Enabled\ \langle A\rangle\text{-}v$
and *h4*: $|\sim\ P \wedge Unchanged\ v \longrightarrow \bigcirc P$
shows $\vdash \Box[N]\text{-}v \wedge WF(A)\text{-}v \longrightarrow (P \leadsto Q)$
⟨*proof*⟩

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

theorem *SF1-general*:
  assumes *h1*: $|^\sim$ $P \wedge N \longrightarrow \bigcirc P \vee \bigcirc Q$
    and *h2*: $|^\sim$ $P \wedge N \wedge \langle A \rangle\text{-}v \longrightarrow \bigcirc Q$
    and *h3*: $\vdash \Box P \wedge \Box N \wedge \Box F \longrightarrow \Diamond Enabled\ \langle A \rangle\text{-}v$
    and *h4*: $|^\sim$ $P \wedge Unchanged\ w \longrightarrow \bigcirc P$
  shows $\vdash \Box N \wedge SF(A)\text{-}v \wedge \Box F \longrightarrow (P \leadsto Q)$
⟨*proof*⟩

theorem *SF1*:
  assumes *h1*: $|^\sim$ $P \wedge [N]\text{-}v \longrightarrow \bigcirc P \vee \bigcirc Q$
    and *h2*: $|^\sim$ $P \wedge \langle N \wedge A \rangle\text{-}v \longrightarrow \bigcirc Q$
    and *h3*: $\vdash \Box P \wedge \Box [N]\text{-}v \wedge \Box F \longrightarrow \Diamond Enabled\ \langle A \rangle\text{-}v$
    and *h4*: $|^\sim$ $P \wedge Unchanged\ v \longrightarrow \bigcirc P$
  shows $\vdash \Box [N]\text{-}v \wedge SF(A)\text{-}v \wedge \Box F \longrightarrow (P \leadsto Q)$
⟨*proof*⟩

Lamport proposes the following rule as an introduction rule for *WF* formulas.

theorem *WF2*:
  assumes *h1*: $|^\sim$ $\langle N \wedge B \rangle\text{-}f \longrightarrow \langle M \rangle\text{-}g$
    and *h2*: $|^\sim$ $P \wedge \bigcirc P \wedge \langle N \wedge A \rangle\text{-}f \longrightarrow B$
    and *h3*: $\vdash P \wedge Enabled\ \langle M \rangle\text{-}g \longrightarrow Enabled\ \langle A \rangle\text{-}f$
    and *h4*: $\vdash \Box [N \wedge \neg B]\text{-}f \wedge WF(A)\text{-}f \wedge \Box F \wedge \Diamond \Box Enabled\ \langle M \rangle\text{-}g \longrightarrow \Diamond \Box P$
  shows $\vdash \Box [N]\text{-}f \wedge WF(A)\text{-}f \wedge \Box F \longrightarrow WF(M)\text{-}g$
⟨*proof*⟩

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

theorem *SF2*:
  assumes *h1*: $|^\sim$ $\langle N \wedge B \rangle\text{-}f \longrightarrow \langle M \rangle\text{-}g$
    and *h2*: $|^\sim$ $P \wedge \bigcirc P \wedge \langle N \wedge A \rangle\text{-}f \longrightarrow B$
    and *h3*: $\vdash P \wedge Enabled\ \langle M \rangle\text{-}g \longrightarrow Enabled\ \langle A \rangle\text{-}f$
    and *h4*: $\vdash \Box [N \wedge \neg B]\text{-}f \wedge SF(A)\text{-}f \wedge \Box F \wedge \Box \Diamond Enabled\ \langle M \rangle\text{-}g \longrightarrow \Diamond \Box P$
  shows $\vdash \Box [N]\text{-}f \wedge SF(A)\text{-}f \wedge \Box F \longrightarrow SF(M)\text{-}g$
⟨*proof*⟩

This is the lattice rule from TLA

theorem *wf-leadsto*:
  assumes *h1*: *wf r*
    and *h2*: $\bigwedge x.\ \vdash F\ x \leadsto (G \vee (\exists\, y.\ \#((y,x) \in r) \wedge F\ y))$
  shows $\vdash F\ x \leadsto G$
⟨*proof*⟩

6.3 Stuttering Invariance

theorem *stut-Enabled*: *STUTINV Enabled* $\langle F \rangle\text{-}v$

⟨*proof*⟩

theorem *stut-WF*: *NSTUTINV F* $\Longrightarrow$ *STUTINV WF*(F)*-v*
⟨*proof*⟩

theorem *stut-SF*: *NSTUTINV F* $\Longrightarrow$ *STUTINV SF*(F)*-v*
⟨*proof*⟩

lemmas *livestutinv* = *stut-WF stut-SF stut-Enabled*

end

7 Representing state in TLA*

theory *State*
imports *Liveness*
begin

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

typedecl *state*

instance *state* :: *world* ⟨*proof*⟩

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

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

definition *stvars* :: *'a statefun* ⇒ *bool*
where *basevars-def*: *stvars* ≡ *surj*

syntax
PRED :: *lift* ⇒ *'a* (‹*PRED* -›)
-stvars :: *lift* ⇒ *bool* (‹*basevars* -›)

translations
PRED P ⇀ (*P*::*state* => -)
-stvars ⇌ *CONST stvars*

Base variables may be assigned arbitrary (type-correct) values. In the following lemma, note that *vs* may be a tuple of variables. The correct identification of base variables is up to the user who must take care not to introduce an inconsistency. For example, *basevars* (x, x) would definitely be inconsistent.

lemma *basevars*: *basevars vs* $\Longrightarrow \exists\, u.\ vs\ u = c$
⟨*proof*⟩

lemma *baseE*:
assumes *H1*: *basevars v* **and** *H2*: $\bigwedge x.\ v\ x = c \Longrightarrow Q$
shows *Q*
⟨*proof*⟩

A variant written for sequences rather than single states.

lemma *first-baseE*:
assumes *H1*: *basevars v* **and** *H2*: $\bigwedge x.\ v\ (\mathit{first}\ x) = c \Longrightarrow Q$
shows *Q*
⟨*proof*⟩

lemma *base-pair1*:
assumes *h*: *basevars* (x,y)
shows *basevars x*
⟨*proof*⟩

lemma *base-pair2*:
assumes *h*: *basevars* (x,y)
shows *basevars y*
⟨*proof*⟩

lemma *base-pair*: *basevars* $(x,y) \Longrightarrow$ *basevars x* ∧ *basevars y*
⟨*proof*⟩

Since the *unit* type has just one value, any state function of unit type satisfies the predicate *basevars*. The following theorem can sometimes be useful because it gives a trivial solution for *basevars* premises.

lemma *unit-base*: *basevars* (*v*::*state* ⇒ *unit*)
⟨*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 satisfiyng 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*: $\bigwedge u.\ vs\ (first\ u) = c \Longrightarrow ((first\ s)\ \#\#\ u) \models F$
  shows $s \models Enabled\ F$
⟨*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 equaivalence up to a variable, requires state equaivalence up to a variable. Our state representation above does not support this, hence we cannot encode Lamport's definition in our theory. Thus, we need to axiomatise quantification over (flexible) variables. Note that with a state representation supporting this, our theory should allow such an encoding.

consts
  EEx :: (′*a statefun* ⇒ *temporal*) ⇒ *temporal* (**binder** ‹*Eex* › *10*)
  AAll :: (′*a statefun* ⇒ *temporal*) ⇒ *temporal* (**binder** ‹*Aall* › *10*)

syntax
  -EEx :: [*idts*, *lift*] => *lift* (‹(*3*∃∃ -./ -)› [*0*,*10*] *10*)
  -AAll :: [*idts*, *lift*] => *lift* (‹(*3*∀∀ -./ -)› [*0*,*10*] *10*)
translations
  -EEx v A == *Eex v. A*
  -AAll v A == *Aall v. A*

axiomatization where
$eexI$: $\vdash F\ x \longrightarrow (\exists\exists\ x.\ F\ x)$
and $eexE$: $[\![s \models (\exists\exists\ x.\ F\ x)\ ;\ basevars\ vs;\ (!!\ x.\ [\![\ basevars\ (x,vs);\ s \models F\ x\ ]\!] \Longrightarrow s \models G)]\!]$
$\Longrightarrow (s \models G)$
and $all\text{-}def$: $\vdash (\forall\forall\ x.\ F\ x) = (\neg(\exists\exists\ x.\ \neg(F\ x)))$
and $eexSTUT$: $STUTINV\ F\ x \Longrightarrow STUTINV\ (\exists\exists\ x.\ F\ x)$
and $history$: $\vdash (I \land \Box[A]\text{-}v) = (\exists\exists\ h.\ (\$h = ha) \land I \land \Box[A \land h\${=}hb]\text{-}(h,v))$

lemmas $eexI\text{-}unl = eexI[unlift\text{-}rule]$ — $w \models F\ x \Longrightarrow w \models (\exists\exists\ x.\ F\ x)$

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

lemmas *tla-defs = unch-def before-def after-def first-def second-def suffix-def 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
$even\text{-}zero$: $0 \in Even$
| $even\text{-}step$: $n \in Even \Longrightarrow Suc\ (Suc\ n) \in Even$

locale *Program* =
fixes $x :: state \Rightarrow nat$
and $init :: temporal$
and $act :: temporal$
and $phi :: temporal$
defines $init \equiv TEMP\ \$x = \#\ 0$
and $act \equiv TEMP\ x` = Suc<Suc<\$x>>$
and $phi \equiv TEMP\ init \land \Box[act]\text{-}x$

lemma (**in** *Program*) *stutinvprog*: *STUTINV phi*
$\langle proof \rangle$

lemma (**in** *Program*) *inveven*: ⊢ *phi* ⟶ □($*x* ∈ # *Even*)
⟨*proof*⟩

end

9 Lamport's Inc example

theory *Inc*
imports *State*
begin

This example illustrates use of the embedding by mechanising the running example of Lamports original TLA paper [5].

datatype *pcount* = *a* | *b* | *g*

locale *Firstprogram* =
fixes *x* :: *state* ⇒ *nat*
and *y* :: *state* ⇒ *nat*
and *init* :: *temporal*
and *m1* :: *temporal*
and *m2* :: *temporal*
and *phi* :: *temporal*
and *Live* :: *temporal*
defines *init* ≡ *TEMP* $*x* = # *0* ∧ $*y* = # *0*
and *m1* ≡ *TEMP* *x*‘ = *Suc*<$*x*> ∧ *y*‘ = $*y*
and *m2* ≡ *TEMP* *y*‘ = *Suc*<$*y*> ∧ *x*‘ = $*x*
and *Live* ≡ *TEMP* *WF*(*m1*)-(*x*,*y*) ∧ *WF*(*m2*)-(*x*,*y*)
and *phi* ≡ *TEMP* (*init* ∧ □[*m1* ∨ *m2*]-(*x*,*y*) ∧ *Live*)
assumes *bvar*: *basevars* (*x*,*y*)

lemma (**in** *Firstprogram*) *STUTINV phi*
⟨*proof*⟩

lemma (**in** *Firstprogram*) *enabled-m1*: ⊢ *Enabled* ⟨*m1*⟩-(*x*,*y*)
⟨*proof*⟩

lemma (**in** *Firstprogram*) *enabled-m2*: ⊢ *Enabled* ⟨*m2*⟩-(*x*,*y*)
⟨*proof*⟩

locale *Secondprogram* = *Firstprogram* +
fixes *sem* :: *state* ⇒ *nat*
and *pc1* :: *state* ⇒ *pcount*
and *pc2* :: *state* ⇒ *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*,*pc2*)
and *gamma2* ≡ *TEMP* $pc2 =#g ∧ pc2‘ = #a ∧ sem‘ = Suc<$sem> ∧ *Unchanged* (*x*,*y*,*pc1*)
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-ch*: |~ ⟨*n1*⟩-*vars* = *n1*

$\langle proof \rangle$

lemma (**in** *Secondprogram*) *enab-alpha1*: $\vdash \$pc1 = \#a \longrightarrow \# \ 0 < \$sem \longrightarrow$ *Enabled alpha1*
$\langle proof \rangle$

lemma (**in** *Secondprogram*) *enab-beta1*: $\vdash \$pc1 = \#b \longrightarrow$ *Enabled beta1*
$\langle proof \rangle$

lemma (**in** *Secondprogram*) *enab-gamma1*: $\vdash \$pc1 = \#g \longrightarrow$ *Enabled gamma1*
$\langle proof \rangle$

lemma (**in** *Secondprogram*) *enab-n1*:
$\vdash$ *Enabled* $\langle n1 \rangle$*-vars* $= (\$pc1 = \#a \longrightarrow \# \ 0 < \$sem)$
$\langle proof \rangle$

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

lemma (**in** *Secondprogram*) *n2-ch*: $|^{\sim} \langle n2 \rangle$*-vars* $=$ *n2*
$\langle proof \rangle$

lemma (**in** *Secondprogram*) *enab-alpha2*: $\vdash \$pc2 = \#a \longrightarrow \# \ 0 < \$sem \longrightarrow$ *Enabled alpha2*
$\langle proof \rangle$

lemma (**in** *Secondprogram*) *enab-beta2*: $\vdash \$pc2 = \#b \longrightarrow$ *Enabled beta2*
$\langle proof \rangle$

lemma (**in** *Secondprogram*) *enab-gamma2*: $\vdash \$pc2 = \#g \longrightarrow$ *Enabled gamma2*
$\langle proof \rangle$

lemma (**in** *Secondprogram*) *enab-n2*:
$\vdash$ *Enabled* $\langle n2 \rangle$*-vars* $= (\$pc2 = \#a \longrightarrow \# \ 0 < \$sem)$
$\langle proof \rangle$

We use rule *SF2* to prove that *psi* implements strong fairness for the abstract action *m1*. Since strong fairness implies weak fairness, it follows that *psi* refines the liveness condition of *phi*.

lemma (**in** *Secondprogram*) *psi-fair-m1*: $\vdash$ *psi* $\longrightarrow$ *SF*(*m1*)-(*x*,*y*)
$\langle proof \rangle$

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-fair-m2*: $\vdash$ *psi* $\longrightarrow$ *SF*(*m2*)-(*x*,*y*)
$\langle proof \rangle$

We can now prove the main theorem, which states that *psi* implements *phi*.

theorem (**in** *Secondprogram*) *impl*: $\vdash$ *psi* $\longrightarrow$ *phi*

⟨*proof*⟩

end

10 Refining a Buffer Specification

theory *Buffer*
imports *State*
begin

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

10.1 Buffer specification

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

```
definition BInit :: 'a statefun ⇒ 'a list statefun ⇒ 'a statefun ⇒ temporal
where BInit ic q oc ≡ TEMP $q = #[]
                        ∧ $ic = $oc   — initial condition of buffer
```

```
definition Enq :: 'a statefun ⇒ 'a list statefun ⇒ 'a statefun ⇒ temporal
where Enq ic q oc ≡ TEMP ic$ ≠ $ic
                     ∧ q$ = $q @ [ ic$ ]
                     ∧ oc$ = $oc      — enqueue a new value
```

```
definition Deq :: 'a statefun ⇒ 'a list statefun ⇒ 'a statefun ⇒ temporal
where Deq ic q oc ≡ TEMP # 0 < length<$q>
                     ∧ oc$ = hd<$q>
                     ∧ q$ = tl<$q>
                     ∧ ic$ = $ic      — dequeue value at front
```

```
definition Nxt :: 'a statefun ⇒ 'a list statefun ⇒ 'a statefun ⇒ temporal
where Nxt ic q oc ≡ TEMP (Enq ic q oc ∨ Deq ic q oc)
```

— internal specification with buffer visible

```
definition ISpec :: 'a statefun ⇒ 'a list statefun ⇒ 'a statefun ⇒ temporal
where ISpec ic q oc ≡ TEMP BInit ic q oc
                        ∧ □[Nxt ic q oc]-(ic,q,oc)
                        ∧ WF(Deq ic q oc)-(ic,q,oc)
```

— external specification: buffer hidden

```
definition Spec :: 'a statefun ⇒ 'a statefun ⇒ temporal
where Spec ic oc == TEMP (∃∃ q. ISpec ic q oc)
```

10.2 Properties of the buffer

The buffer never enqueues the same element twice. We therefore have the following invariant:

- any two subsequent elements in the queue are different, and the last element in the queue is different from the value of the output channel,
- if the queue is non-empty then the last element in the queue is the value that appears on the input channel,
- if the queue is empty then the values on the output and input channels are equal.

The following auxiliary predicate *noreps* is true if no two subsequent elements in a list are identical.

definition *noreps* :: *'a list* ⇒ *bool*
where *noreps xs* ≡ ∀ *i* < *length xs* − *1*. *xs*!*i* ≠ *xs*!(*Suc i*)

definition *BInv* :: *'a statefun* ⇒ *'a list statefun* ⇒ *'a statefun* ⇒ *temporal*
where *BInv ic q oc* ≡ *TEMP List.last*<$*oc* # $*q*> = $*ic* ∧ *noreps*<$*oc* # $*q*>

lemmas *buffer-defs* = *BInit-def Enq-def Deq-def Nxt-def*
ISpec-def Spec-def BInv-def

lemma *ISpec-stutinv*: *STUTINV* (*ISpec ic q oc*)
⟨*proof*⟩

lemma *Spec-stutinv*: *STUTINV Spec ic oc*
⟨*proof*⟩

A lemma about lists that is useful in the following

lemma *tl-self-iff-empty*[*simp*]: (*tl xs* = *xs*) = (*xs* = [])
⟨*proof*⟩

lemma *tl-self-iff-empty'*[*simp*]: (*xs* = *tl xs*) = (*xs* = [])
⟨*proof*⟩

lemma *Deq-visible*:
assumes *v*: ⊢ *Unchanged v* ⟶ *Unchanged q*
shows |~ <*Deq ic q oc*>-*v* = *Deq ic q oc*
⟨*proof*⟩

lemma *Deq-enabledE*: ⊢ *Enabled* <*Deq ic q oc*>-(*ic*,*q*,*oc*) ⟶ $*q* ~= #[]
⟨*proof*⟩

We now prove that *BInv* is an invariant of the Buffer specification.

We need several lemmas about *noreps* that are used in the invariant proof.

lemma *noreps-empty* [*simp*]: *noreps* []
⟨*proof*⟩

lemma *noreps-singleton*: *noreps* [*x*] — special case of following lemma
⟨*proof*⟩

lemma *noreps-cons* [*simp*]:
noreps (*x* # *xs*) = (*noreps xs* ∧ (*xs* = [] ∨ *x* ≠ *hd xs*))
⟨*proof*⟩

lemma *noreps-append* [*simp*]:
noreps (*xs* @ *ys*) =
(*noreps xs* ∧ *noreps ys* ∧ (*xs* = [] ∨ *ys* = [] ∨ *List.last xs* ≠ *hd ys*))
⟨*proof*⟩

lemma *ISpec-BInv-lemma*:
⊢ *BInit ic q oc* ∧ □[*Nxt ic q oc*]-(*ic*,*q*,*oc*) ⟶ □(*BInv ic q oc*)
⟨*proof*⟩

theorem *ISpec-BInv*: ⊢ *ISpec ic q oc* ⟶ □(*BInv ic q oc*)
⟨*proof*⟩

10.3 Two FIFO buffers in a row implement a buffer

locale *DBuffer* =
fixes *inp* :: *'a statefun* — input channel for double FIFO
and *mid* :: *'a statefun* — channel linking the two buffers
and *out* :: *'a statefun* — output channel for double FIFO
and *q1* :: *'a list statefun* — inner queue of first FIFO
and *q2* :: *'a list statefun* — inner queue of second FIFO
and *vars*
defines *vars* ≡ *LIFT* (*inp*,*mid*,*out*,*q1*,*q2*)
assumes *DB-base*: *basevars vars*
begin

We need to specify the behavior of two FIFO buffers in a row. Intuitively, that specification is just the conjunction of two buffer specifications, where the first buffer has input channel *inp* and output channel *mid* whereas the second one receives from *mid* and outputs on *out*. However, this conjunction allows a simultaneous enqueue action of the first buffer and dequeue of the second one. It would not implement the previous buffer specification, which excludes such simultaneous enqueueing and dequeueing (it is written in "interleaving style"). We could relax the specification of the FIFO buffer above, which is esthetically pleasant, but non-interleaving specifications are usually hard to get right and to understand. We therefore impose an interleaving constraint on the specification of the double buffer, which requires that enqueueing and dequeueing do not happen simultaneously.

definition *DBSpec*

where *DBSpec* ≡ *TEMP ISpec inp q1 mid*
$\wedge$ *ISpec mid q2 out*
$\wedge$ $\Box[\neg(\textit{Enq inp q1 mid} \wedge \textit{Deq mid q2 out})]\text{-}\textit{vars}$

The proof rules of TLA are geared towards specifications of the form *Init* $\wedge$ $\Box[\textit{Next}]\text{-}\textit{vars} \wedge L$, and we prove that *DBSpec* corresponds to a specification in this form, which we now define.

definition *FullInit*
where *FullInit* ≡ *TEMP* (*BInit inp q1 mid* $\wedge$ *BInit mid q2 out*)

definition *FullNxt*
where *FullNxt* ≡ *TEMP* (*Enq inp q1 mid* $\wedge$ *Unchanged* (*q2*,*out*)
$\vee$ *Deq inp q1 mid* $\wedge$ *Enq mid q2 out*
$\vee$ *Deq mid q2 out* $\wedge$ *Unchanged* (*inp*,*q1*))

definition *FullSpec*
where *FullSpec* ≡ *TEMP FullInit*
$\wedge$ $\Box[\textit{FullNxt}]\text{-}\textit{vars}$
$\wedge$ *WF*(*Deq inp q1 mid*)-*vars*
$\wedge$ *WF*(*Deq mid q2 out*)-*vars*

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

definition *qc* :: *'a list statefun*
where *qc* ≡ *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*:
$\vdash \Box\textit{BInv inp q1 mid}$
$\wedge \Box[\textit{Nxt inp q1 mid}]\text{-}(\textit{inp},\textit{q1},\textit{mid})$
$\wedge \Box[\textit{Nxt mid q2 out}]\text{-}(\textit{mid},\textit{q2},\textit{out})$
$\wedge \Box[\neg(\textit{Enq inp q1 mid} \wedge \textit{Deq mid q2 out})]\text{-}\textit{vars}$
$\longrightarrow \Box[\textit{FullNxt}]\text{-}\textit{vars}$

(**is** $\vdash \Box ?inv \wedge ?nxts \longrightarrow \Box[FullNxt]\text{-}vars$)
⟨*proof*⟩

It is now easy to show that *DBSpec* refines *FullSpec*.

theorem *DBSpec-impl-FullSpec*: $\vdash DBSpec \longrightarrow FullSpec$
⟨*proof*⟩

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*: $\vdash FullInit \longrightarrow BInit\ inp\ qc\ out$
⟨*proof*⟩

lemma *Full-step-simulation*:
$|^\sim [FullNxt]\text{-}vars \longrightarrow [Nxt\ inp\ qc\ out]\text{-}(inp,qc,out)$
⟨*proof*⟩

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*: $\vdash Enabled\ \langle Deq\ inp\ q1\ mid\rangle\text{-}vars = (\$q1 \neq \#[])$
⟨*proof*⟩

lemma *Deq2-enabled*: $\vdash Enabled\ \langle Deq\ mid\ q2\ out\rangle\text{-}vars = (\$q2 \neq \#[])$
⟨*proof*⟩

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*:
$\vdash \Box[FullNxt]\text{-}vars \wedge WF(Deq\ mid\ q2\ out)\text{-}vars \wedge \Box WF(Deq\ inp\ q1\ mid)\text{-}vars$
$\longrightarrow WF(Deq\ inp\ qc\ out)\text{-}(inp,qc,out)$
⟨*proof*⟩

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

theorem *FullSpec-impl-ISpec*: $\vdash FullSpec \longrightarrow ISpec\ inp\ qc\ out$
⟨*proof*⟩

theorem *FullSpec-impl-Spec*: $\vdash FullSpec \longrightarrow Spec\ inp\ out$
⟨*proof*⟩

By transitivity, two buffers in a row also implement a single buffer.

theorem *DBSpec-impl-Spec*: $\vdash$ *DBSpec* $\longrightarrow$ *Spec inp out*
⟨*proof*⟩

end — locale DBuffer

end

References

[1] K. Chaudhuri, D. Doligez, L. Lamport, and S. Merz. Verifying safety properties with the tla$^+$ proof system. In J. Giesl and R. Hähnle, editors, *5th Intl. Joint Conf. Automated Reasoning (IJCAR 2010)*, volume 6173 of *Lecture Notes in Computer Science*, pages 142–148, Edinburgh, UK, 2010. Springer.

[2] M. Devillers, D. Griffioen, and O. Müller. Possibly Infinite Sequences in Theorem Provers: A comparative study. In E. L. Gunter and A. P. Felty, editors, *10th International Conference on Theorem Proving in Higher Order Logics*, volume 1275 of *Lecture Notes in Computer Science*, pages 89–104. Springer, August 1997.

[3] S. O. Ehmety and L. C. Paulson. Representing Component States in Higher-Order Logic. In R. J. Boulton and P. B. Jackson, editors, *Theorem Proving in Higher Order Logics*, pages 151–158, 2001.

[4] G. Grov. *Reasoning about Correctness Properties of a Coordination Programming Language.* PhD thesis, Heriot-Watt University, March 2009.

[5] L. Lamport. The Temporal Logic of Actions. *ACM Transactions on Programming Languages and Systems*, 16(3):872–923, May 1994.

[6] L. Lamport. *Specifying Systems — The TLA+ Language and Tools for Hardware and Software Engineers.* Addison-Wesley, Reading, Massachusetts, 2002.

[7] S. Merz. An Encoding of TLA in Isabelle. http://www.pst.informatik.uni-muenchen.de/˜merz/isabelle/. Part of the Isabelle distribution., 1998.

[8] S. Merz. A More Complete TLA. In J. Wing, J. Woodcock, and J. Davies, editors, *FM'99: World Congress on Formal Methods*, volume 1709 of *Lecture Notes in Computer Science*, pages 1226–1244, Toulouse, France, Sept. 1999. Springer-Verlag.

[9] M. Wenzel. Using Axiomatic Type Classes in Isabelle, May 2000.

[10] M. Wildmoser and T. Nipkow. Certifying Machine Code Safety: Shallow versus Deep Embedding. In K. Slind, A. Bunker, and G. Gopalakrishnan, editors, *Theorem Proving in Higher Order Logics (TPHOLs 2004)*, volume 3223 of *LNCS*, pages 305–320. Springer, 2004.