A Tutorial Introduction to TLAPS

Jael Kriener [1] Tom Rodeheffer [2] Tomer Libal [1]

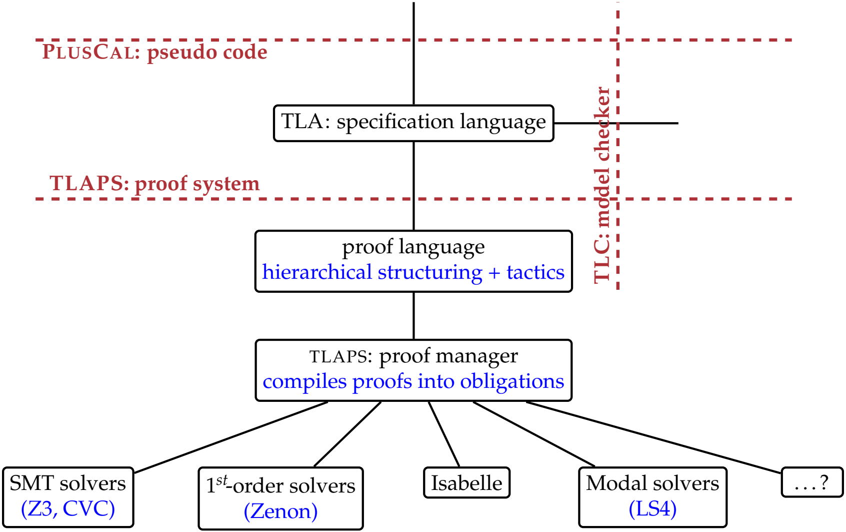
1 Microsoft Research - Inria JOINT CENTRE

2 Microsoft Research

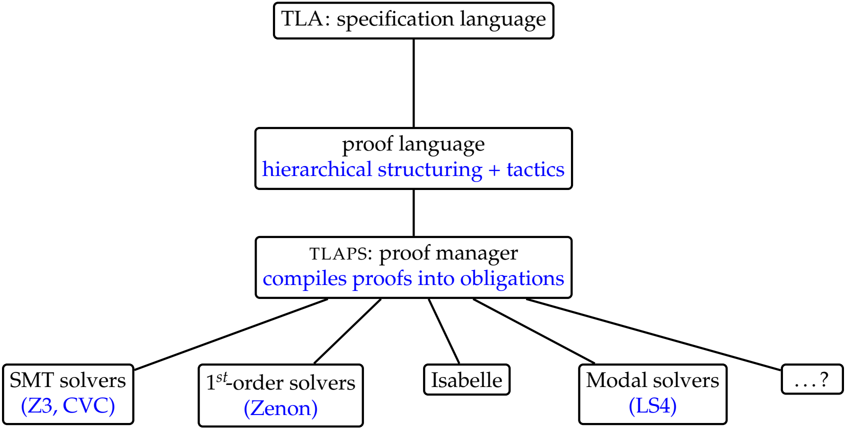
TLA+ Community Event, ABZ 2014
Toulouse, June 3, 2014

Outline

TLAPS in Context



TLAPS in Context

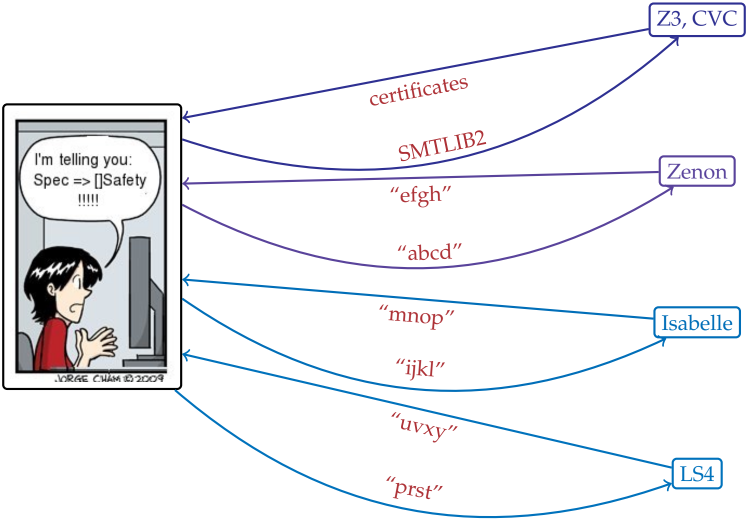


TLAPS in Context

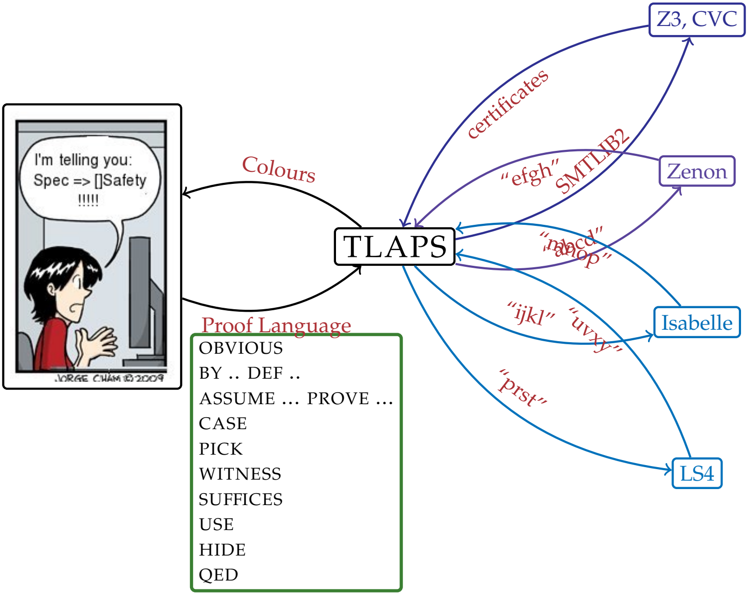
TLAPS essentially does two things:

- translate between TLA and the languages that the backend provers understand;
- help the user break up a theorem P into obligations $O_1 \ldots O_n$, while maintaining the fact that $O_1 \land \ldots \land O_n \Rightarrow P$.

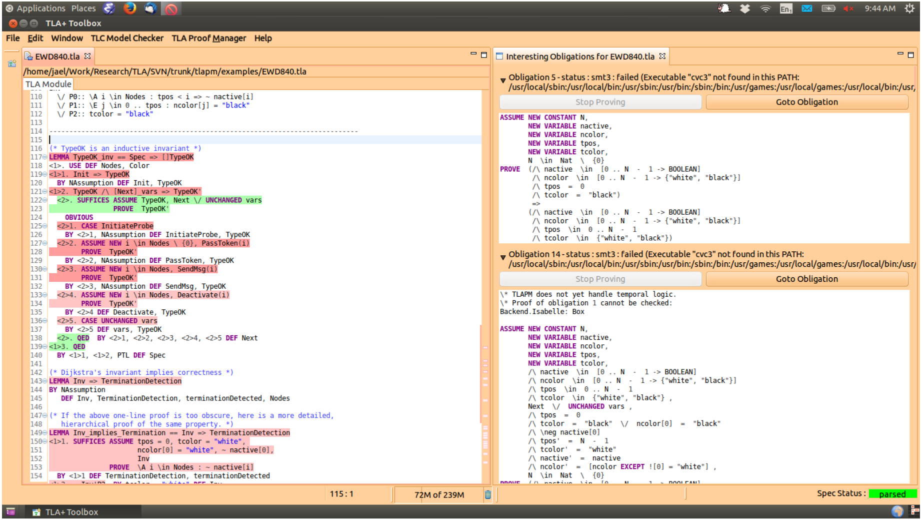
Talking to ATPs about TLA Specs



Talking to ATPs about TLA Specs



Hence the way the interface looks:



conversation

user $\Longleftrightarrow$ TLAPS

snipplets of conversations

TLAPS $\Longleftrightarrow$ backend provers

There are two kinds of TLAPS proofs:

one-liners

- OBVIOUS
- BY ... [DEF ...]

Each one-line proof generates one obligation.

(Or there abouts...)

hierarchical

```
⟨1⟩1 X
  ⟨2⟩1 Y
    OBVIOUS
    ⋮
  ⟨2⟩q QED
    BY ... DEF ...
⋮
⟨1⟩q QED
  BY ... DEF ...
```

Obligations

An *obligation* is a claim of the form $\Gamma \vdash P$,
which is translated and handed on to the backend provers.

Obligations

An *obligation* is a claim of the form ASSUME Γ PROVE P,
which is translated and handed on to the backend provers.

To prove an obligation, by default, TLAPS will ask:

1. CVC
2. Zenon
3. Isabelle

But one can change that...

An *obligation* is a claim of the form ASSUME Γ PROVE P,
which is translated and handed on to the backend provers.

The TLAPS game is mainly to construct obligations so that:

1. they are true, i.e.:
   1. Γ contains all relevant facts), and
   2. all relevant definitions are unfolded;
2. they are not too big for the backend provers to handle.

Obligations: Controlling Γ

Once one has one's logic right, the game is to control Γ.

By default:

- all constant-/variable-declarations, with domain-assumptions, are in Γ;
- no definitions are unfolded.

Once one has one's logic right, the game is to control Γ.

named & un-named steps

$\langle 1 \rangle 1 \;\; X$
$\quad \ldots$

$\langle 1 \rangle \;\; Y$
$\quad \ldots$

$\langle 1 \rangle 3 \;\; Z$
(* here Y is in Γ, but X is not *)
BY $\langle 1 \rangle 1$ (* here Y and X are in Γ *)

USE & HIDE

The keywords USE resp. HIDE include in resp. remove from Γ steps, theorems or assumptions;

USE [DEF] resp. HIDE [DEF] fold resp. unfold definitions in Γ.

Whether a USE- and HIDE-step is named or un-named does not matter.

Writing a simple Hierarchical Proof

quick recap: EWD 840

Writing a simple Hierarchical Proof

The safety-proof has the following structure:

LEMMA $Spec \Rightarrow \Box TerminationDetection$

(* Dijkstra's invariant implies correctness *)
$\langle 1 \rangle 1\ \ Inv \Rightarrow TerminationDetection$

(* Dijkstra's invariant is (trivially) established by the initial condition *)
$\langle 1 \rangle 2\ \ Init \Rightarrow Inv$

(* Dijkstra's invariant is inductive relative to the type invariant *)
$\langle 1 \rangle 3\ \ TypeOK \wedge Inv \wedge [Next]_{vars} \Rightarrow Inv'$

$\langle 1 \rangle q\ \ QED$
BY $\langle 1 \rangle 1$, $\langle 1 \rangle 2$, $\langle 1 \rangle 3$, $TypeOK_{inv}$, PTL DEF $Spec$

writing a simple hierarchical proof

Some more Proof Constructs: WITNESS

When proving a goal of the form:

$$\exists x \in S : \; P(x)$$

To prove it we can write:

$$\langle 1 \rangle 6 \;\; \textsc{witness} \;\; a \in S$$

for some a already in Γ.

The effect is:

1. step $\langle 1 \rangle 6$ needs a proof that $a \in S$;
2. the goal from now on is $P(a)$.

Some more Proof Constructs: PICK

When Γ contains a statement of the form:

$$\exists x \in S : \; P(x)$$

To use it we can write:

$$\langle 1 \rangle 6 \;\; \text{PICK} \;\; a \in S : P(a)$$

for some fresh a.

The effect is:

1. we have a new $a \in S$ in Γ;
2. using $\langle 1 \rangle 6$ will put $P(a)$ into Γ.

SUFFICES is useful to avoid deeply nested hierarchical proofs:

$\langle 6 \rangle 4$ X
  $\langle 7 \rangle$ proof Π

$\langle 6 \rangle q$ QED
  BY $\langle 6 \rangle 4$, proof Σ

$\langle 6 \rangle 4$ SUFFICES X
  proof Σ

$\langle 6 \rangle 5$ proof Π

$\langle 6 \rangle q$ QED
  BY $\langle 6 \rangle 4$, $\langle 6 \rangle 5$

Outline

A simple theorem about sequences

- *Concat left cancellation:* Given three sequences A, B, C where $C \circ A = C \circ B$, it follows that $A = B$.
  - Simple, but not trivial. Multiplication, for example, does not have left cancellation, because you can multiply by zero.

A simple theorem about sequences

- *Concat left cancellation:* Given three sequences A, B, C where $C \circ A = C \circ B$, it follows that $A = B$.
  - Simple, but not trivial. Multiplication, for example, does not have left cancellation, because you can multiply by zero.
- Write the theorem in TLA

A simple theorem about sequences

- *Concat left cancellation:* Given three sequences A, B, C where $C \circ A = C \circ B$, it follows that $A = B$.
  - Simple, but not trivial. Multiplication, for example, does not have left cancellation, because you can multiply by zero.
- Write the theorem in TLA

  As a quantified formula:

  $$\forall S : \forall A, B, C \in Seq(S) :$$
  $$C \circ A = C \circ B \Rightarrow A = B$$

A simple theorem about sequences

- *Concat left cancellation:* Given three sequences A, B, C where $C \circ A = C \circ B$, it follows that $A = B$.
  - Simple, but not trivial. Multiplication, for example, does not have left cancellation, because you can multiply by zero.
- Write the theorem in TLA

As a quantified formula:

$$\forall\, S : \forall\, A, B, C \in Seq(S) : \\ C \circ A = C \circ B \Rightarrow A = B$$

As an ASSUME-PROVE:

$$\begin{array}{l} \text{ASSUME} \\ \quad \text{NEW } S, \\ \quad \text{NEW } A \in Seq(S), \\ \quad \text{NEW } B \in Seq(S), \\ \quad \text{NEW } C \in Seq(S), \\ \quad C \circ A = C \circ B \\ \text{PROVE } A = B \end{array}$$

Proof attempt 1 - is it obvious - fail

THEOREM $ConcatLeftCancel \triangleq$
  ASSUME
    NEW S,
    NEW $A \in Seq(S)$,
    NEW $B \in Seq(S)$,
    NEW $C \in Seq(S)$,
    $C \circ A = C \circ B$
  PROVE
  $A = B$
PROOF
  $\langle 1 \rangle$ QED OBVIOUS   unable to prove it

Proof attempt 2 - add some facts - still fail

THEOREM $ConcatLeftCancel \triangleq$
  ASSUME
    NEW S,
    NEW $A \in Seq(S)$,
    NEW $B \in Seq(S)$,
    NEW $C \in Seq(S)$,
    $C \circ A = C \circ B$
  PROVE
  $A = B$
PROOF
  $\langle 1 \rangle 1.\ Len(A) = Len(B)$ OBVIOUS $C \circ A = C \circ B$
  $\langle 1 \rangle 2.\ A \in [1 \,..\, Len(A) \rightarrow S]$ OBVIOUS $A \in Seq(S)$
  $\langle 1 \rangle 3.\ B \in [1 \,..\, Len(A) \rightarrow S]$ BY $\langle 1 \rangle 1$
  $\langle 1 \rangle 4.\ \forall\, i \in 1 \,..\, Len(A) : A[i] = (C \circ A)[i + Len(C)]$ OBVIOUS
  $\langle 1 \rangle 5.\ \forall\, i \in 1 \,..\, Len(A) : B[i] = (C \circ B)[i + Len(C)]$ BY $\langle 1 \rangle 1$
  $\langle 1 \rangle$ QED BY $\langle 1 \rangle 2, \langle 1 \rangle 3, \langle 1 \rangle 4, \langle 1 \rangle 5$ unable to prove it

Proof attempt 3 - another fact - success

THEOREM $ConcatLeftCancel \triangleq$
  ASSUME
    NEW S,
    NEW $A \in Seq(S)$,
    NEW $B \in Seq(S)$,
    NEW $C \in Seq(S)$,
    $C \circ A = C \circ B$
  PROVE
  $A = B$
PROOF
  $\langle 1 \rangle 1.\ Len(A) = Len(B)$ OBVIOUS $C \circ A = C \circ B$
  $\langle 1 \rangle 2.\ A \in [1 \,..\, Len(A) \rightarrow S]$ OBVIOUS $A \in Seq(S)$
  $\langle 1 \rangle 3.\ B \in [1 \,..\, Len(A) \rightarrow S]$ BY $\langle 1 \rangle 1$
  $\langle 1 \rangle 4.\ \forall i \in 1 \,..\, Len(A) : A[i] = (C \circ A)[i + Len(C)]$ OBVIOUS
  $\langle 1 \rangle 5.\ \forall i \in 1 \,..\, Len(A) : B[i] = (C \circ B)[i + Len(C)]$ BY $\langle 1 \rangle 1$
  $\langle 1 \rangle 6.\ \forall i \in 1 \,..\, Len(A) : A[i] = B[i]$ BY $\langle 1 \rangle 4, \langle 1 \rangle 5$
  $\langle 1 \rangle$ QED BY $\langle 1 \rangle 2, \langle 1 \rangle 3, \langle 1 \rangle 6$

Proof with better structure

THEOREM $ConcatLeftCancel \triangleq$
  ASSUME
    NEW S,
    NEW $A \in Seq(S)$,
    NEW $B \in Seq(S)$,
    NEW $C \in Seq(S)$,
    $C \circ A = C \circ B$
  PROVE
  $A = B$
PROOF
  $\langle 1 \rangle 1.$ $Len(A) = Len(B)$ OBVIOUS $C \circ A = C \circ B$
  $\langle 1 \rangle 2.$ $A \in [1 \,..\, Len(A) \rightarrow S]$ OBVIOUS $A \in Seq(S)$
  $\langle 1 \rangle 3.$ $B \in [1 \,..\, Len(A) \rightarrow S]$ BY $\langle 1 \rangle 1$
  $\langle 1 \rangle 4.$ ASSUME NEW $i \in 1 \,..\, Len(A)$ PROVE $A[i] = B[i]$
    $\langle 2 \rangle 1.$ $A[i] = (C \circ A)[i + Len(C)]$ OBVIOUS   defn of $C \circ A$
    $\langle 2 \rangle 2.$ $B[i] = (C \circ B)[i + Len(C)]$ BY $\langle 1 \rangle 1$ defn of $C \circ B$
    $\langle 2 \rangle$ QED BY $\langle 2 \rangle 1$, $\langle 2 \rangle 2$
  $\langle 1 \rangle$ QED BY $\langle 1 \rangle 2$, $\langle 1 \rangle 3$, $\langle 1 \rangle 4$

Lessons from proving *ConcatLeftCancel*

Lessons from proving *ConcatLeftCancel*

- The proof centers on showing $A = B$ where A, B are functions
  - For two functions to be equal, you must show
    - they have the same domain
    - they have the same value at each point in the domain
  - It seems this is relatively difficult for TLAPS to conclude

Lessons from proving *ConcatLeftCancel*

- The proof centers on showing $A = B$ where A, B are functions
  - For two functions to be equal, you must show
    - they have the same domain
    - they have the same value at each point in the domain
  - It seems this is relatively difficult for TLAPS to conclude
- Before writing a subproof, check if TLAPS thinks a fact is obvious

Lessons from proving *ConcatLeftCancel*

- The proof centers on showing $A = B$ where A, B are functions
  - For two functions to be equal, you must show
    - they have the same domain
    - they have the same value at each point in the domain
  - It seems this is relatively difficult for TLAPS to conclude
- Before writing a subproof, check if TLAPS thinks a fact is obvious
- When TLAPS fails, try to figure out what specific fact you could provide that it is failing to consider

Lessons from proving *ConcatLeftCancel*

- The proof centers on showing $A = B$ where A, B are functions
  - For two functions to be equal, you must show
    - they have the same domain
    - they have the same value at each point in the domain
  - It seems this is relatively difficult for TLAPS to conclude
- Before writing a subproof, check if TLAPS thinks a fact is obvious
- When TLAPS fails, try to figure out what specific fact you could provide that it is failing to consider
- When introducing a new symbol x, generally it is a good idea to use a domain formula $x \in S$

Using the theorem *ConcatLeftCancel*

Often, what a theorem considers as constant parameters are messy formulas at the point where we wish to apply the theorem. In this example, we conjure up formulas that happen to be sequences, and ask TLAPS to apply *ConcatLeftCancel*.

Use attempt 1 - is it obvious - fail

THEOREM $UseConcatLeftCancel \triangleq$
  ASSUME
    NEW S,
    NEW $u \in Seq(S)$,
    NEW $v \in Seq(S)$,
    NEW $w \in Seq(S)$,
    NEW $x \in Seq(S)$,
    NEW $m \in S$,
    NEW $n \in S$,
    $u \circ \langle m, n \rangle \circ v \circ x = u \circ \langle m, n \rangle \circ w \circ x$
  PROVE
  $v \circ x = w \circ x$
PROOF
  $\langle 1 \rangle$ QED BY $ConcatLeftCancel$ unable to prove it

Use attempt 2 - add a closure fact - still fail

THEOREM $UseConcatLeftCancel \triangleq$
  ASSUME
    NEW S,
    NEW $u \in Seq(S)$,
    NEW $v \in Seq(S)$,
    NEW $w \in Seq(S)$,
    NEW $x \in Seq(S)$,
    NEW $m \in S$,
    NEW $n \in S$,
    $u \circ \langle m, n \rangle \circ v \circ x = u \circ \langle m, n \rangle \circ w \circ x$
  PROVE
  $v \circ x = w \circ x$
PROOF
  $\langle 1 \rangle 1.\ u \circ \langle m, n \rangle \in Seq(S)$ OBVIOUS ◦ closed
  $\langle 1 \rangle$ QED BY $\langle 1 \rangle 1$, $ConcatLeftCancel$ unable to prove it

Use attempt 3 - add more closure facts - still fail

THEOREM $UseConcatLeftCancel \triangleq$
  ASSUME
    NEW S,
    NEW $u \in Seq(S)$,
    NEW $v \in Seq(S)$,
    NEW $w \in Seq(S)$,
    NEW $x \in Seq(S)$,
    NEW $m \in S$,
    NEW $n \in S$,
    $u \circ \langle m, n \rangle \circ v \circ x = u \circ \langle m, n \rangle \circ w \circ x$
  PROVE
    $v \circ x = w \circ x$
PROOF
  $\langle 1 \rangle 1.\ u \circ \langle m, n \rangle \in Seq(S)$ OBVIOUS ◦ closed
  $\langle 1 \rangle 2.\ v \circ x \in Seq(S)$ OBVIOUS ◦ closed
  $\langle 1 \rangle 3.\ w \circ x \in Seq(S)$ OBVIOUS ◦ closed
  $\langle 1 \rangle$ QED BY $\langle 1 \rangle 1$, $\langle 1 \rangle 2$, $\langle 1 \rangle 3$, $ConcatLeftCancel$ unable to prove it

Use attempt 4 - add an associativity fact - success

THEOREM $UseConcatLeftCancel \triangleq$
  ASSUME
    NEW S,
    NEW $u \in Seq(S)$,
    NEW $v \in Seq(S)$,
    NEW $w \in Seq(S)$,
    NEW $x \in Seq(S)$,
    NEW $m \in S$,
    NEW $n \in S$,
    $u \circ \langle m, n \rangle \circ v \circ x = u \circ \langle m, n \rangle \circ w \circ x$
  PROVE
  $v \circ x = w \circ x$
PROOF
  $\langle 1 \rangle 1.\ u \circ \langle m, n \rangle \in Seq(S)$ OBVIOUS ○ closed
  $\langle 1 \rangle 2.\ v \circ x \in Seq(S)$ OBVIOUS ○ closed
  $\langle 1 \rangle 3.\ w \circ x \in Seq(S)$ OBVIOUS ○ closed
  $\langle 1 \rangle 4.\ u \circ \langle m, n \rangle \circ (v \circ x) = u \circ \langle m, n \rangle \circ (w \circ x)$ OBVIOUS
  $\langle 1 \rangle$ QED BY $\langle 1 \rangle 1, \langle 1 \rangle 2, \langle 1 \rangle 3, \langle 1 \rangle 4, ConcatLeftCancel$

Lessons from applying *ConcatLeftCancel*

Lessons from applying *ConcatLeftCancel*

- Common mathematical properties of closure and associativity can be important

Lessons from applying *ConcatLeftCancel*

- Common mathematical properties of closure and associativity can be important
  - Humans are really good at utilizing these properties

Lessons from applying *ConcatLeftCancel*

- Common mathematical properties of closure and associativity can be important
  - Humans are really good at utilizing these properties
  - Even though TLAPS considered the properties obvious, it was unable to supply them automatically when trying to prove a deduction that required them

Lessons from applying *ConcatLeftCancel*

- Common mathematical properties of closure and associativity can be important
  - Humans are really good at utilizing these properties
  - Even though TLAPS considered the properties obvious, it was unable to supply them automatically when trying to prove a deduction that required them
  - In my experience, TLAPS has a really difficult time applying associativity

Lessons from applying *ConcatLeftCancel*

- Common mathematical properties of closure and associativity can be important
  - Humans are really good at utilizing these properties
  - Even though TLAPS considered the properties obvious, it was unable to supply them automatically when trying to prove a deduction that required them
  - In my experience, TLAPS has a really difficult time applying associativity
- When TLAPS fails, try to figure out what specific fact you could provide that it is failing to consider

Finite induction over naturals

Finite induction over naturals

- The ordinary form of induction is simple induction over the naturals, in which a predicate $P(i)$ is proved to hold for all $i \in Nat$.

Finite induction over naturals

- The ordinary form of induction is simple induction over the naturals, in which a predicate $P(i)$ is proved to hold for all $i \in Nat$.
- TLAPS has a libary theorem *NatInduction*, in the library module *NaturalsInduction*, that encapsulates the simple inductive argument. For any $P(\_)$, given the base case

$$P(0)$$

and the inductive step

$$\forall i \in Nat : P(i) \Rightarrow P(i+1)$$

NatInduction concludes

$$\forall i \in Nat : P(i)$$

Finite induction over naturals - 2

- Sometimes we do not want or need to prove that $P(i)$ holds for all $i \in Nat$, but rather only for a finite range $i \in m..n$. This often occurs when proving things about sequences.

Finite induction over naturals - 2

- Sometimes we do not want or need to prove that $P(i)$ holds for all $i \in Nat$, but rather only for a finite range $i \in m..n$. This often occurs when proving things about sequences.
- In such cases, we could, of course, define a more general predicate

$$Q(i) \triangleq i \in m..n \Rightarrow P(i)$$

use *NatInduction* to prove that $Q(i)$ holds for all $i \in Nat$ and then deduce what we want about $P(\_)$. But the proof would be cluttered with the transitions of i into and out of $m..n$.

Finite induction over naturals - 2

- Sometimes we do not want or need to prove that $P(i)$ holds for all $i \in Nat$, but rather only for a finite range $i \in m..n$. This often occurs when proving things about sequences.
- In such cases, we could, of course, define a more general predicate

$$Q(i) \triangleq i \in m..n \Rightarrow P(i)$$

  use *NatInduction* to prove that $Q(i)$ holds for all $i \in Nat$ and then deduce what we want about $P(\_)$. But the proof would be cluttered with the transitions of i into and out of $m..n$.
- A better approach is to define a prove and prove a theorem *FiniteNatInduction* that explicitly deals with finite induction over the naturals.

Setting up the inductive argument

THEOREM $FiniteNatInduction \triangleq$
  ASSUME
    NEW $P(\_)$, predicate
    NEW $m \in Nat$, start
    NEW $n \in Nat$, limit
    $P(m)$, base case
    $\forall\, i \in m \,..\, (n-1) : P(i) \Rightarrow P(i+1)$ finite ind hyp
  PROVE $\forall\, i \in m \,..\, n : P(i)$
PROOF
  $\langle 1 \rangle$ DEFINE $Q(i) \triangleq i \in m \,..\, n \Rightarrow P(i)$
  $\langle 1 \rangle$ SUFFICES $\forall\, i \in Nat : Q(i)$ OBVIOUS
  base case
  $\langle 1 \rangle 1.\ Q(0)$ OBVIOUS
  inductive step
  $\langle 1 \rangle 2.\ \forall\, i \in Nat : Q(i) \Rightarrow Q(i+1)$
  $\langle 1 \rangle$ HIDE DEF Q hide defn of induction predicate
  $\langle 1 \rangle$ QED BY $\langle 1 \rangle 1$, $\langle 1 \rangle 2$, $NatInduction$

- Define the more general predicate $Q(\_)$
- Use a SUFFICES to change the goal to $\forall\, i \in Nat : Q(i)$
- State the base case and inductive step as facts
- Hide the definition of the inductive predicate $Q(\_)$
- Appeal to $NatInduction$

Completing the subproof of the inductive step

```
THEOREM FiniteNatInduction ≜
  ASSUME
    NEW P(_),              predicate
    NEW m ∈ Nat,           start
    NEW n ∈ Nat,           limit
    P(m),                  base case
    ∀ i ∈ m..(n − 1) : P(i) ⇒ P(i + 1)   finite ind hyp
  PROVE ∀ i ∈ m..n : P(i)
PROOF
  ⟨1⟩ DEFINE Q(i) ≜ i ∈ m..n ⇒ P(i)
  ⟨1⟩ SUFFICES ∀ i ∈ Nat : Q(i) OBVIOUS
  base case
  ⟨1⟩1. Q(0) OBVIOUS
  inductive step
  ⟨1⟩2. ∀ i ∈ Nat : Q(i) ⇒ Q(i + 1)
    ⟨2⟩1. SUFFICES ASSUME NEW i ∈ Nat, Q(i)
             PROVE Q(i + 1) OBVIOUS
    ⟨2⟩2.CASE i + 1 ∈ (m + 1)..n BY ⟨2⟩1, ⟨2⟩2
    ⟨2⟩3.CASE i + 1 = m BY ⟨2⟩3
    ⟨2⟩4.CASE i + 1 ∉ m..n BY ⟨2⟩4
    ⟨2⟩ QED BY ⟨2⟩2, ⟨2⟩3, ⟨2⟩4
  ⟨1⟩ HIDE DEF Q        hide defn of induction predicate
  ⟨1⟩ QED BY ⟨1⟩1, ⟨1⟩2, NatInduction
```

- Use SUFFICES ASSUME PROVE to disassemble the universal quantifier and the implication
- Use CASE to perform a case analysis
- The cases must cover all possibilities

Simplified proof of *FiniteNatInduction*

```
THEOREM FiniteNatInduction ≜
  ASSUME
    NEW P(_),                         predicate
    NEW m ∈ Nat,                      start
    NEW n ∈ Nat,                      limit
    P(m),                             base case
    ∀ i ∈ m .. (n − 1) : P(i) ⇒ P(i + 1)   finite ind hyp
  PROVE ∀ i ∈ m .. n : P(i)
PROOF
  ⟨1⟩ DEFINE Q(i) ≜ i ∈ m .. n ⇒ P(i)
  ⟨1⟩ SUFFICES ∀ i ∈ Nat : Q(i) OBVIOUS
  base case
  ⟨1⟩1. Q(0) OBVIOUS
  inductive step
  ⟨1⟩2. ∀ i ∈ Nat : Q(i) ⇒ Q(i + 1) OBVIOUS
  ⟨1⟩ HIDE DEF Q          hide defn of induction predicate
  ⟨1⟩ QED BY ⟨1⟩1, ⟨1⟩2, NatInduction
```

- It turns out that TLAPS thinks that the inductive step is obvious. We neglected to check this before plunging into the case analysis. Hence the proof can be simplified considerably.

Lessons from proving *FiniteNatInduction*

Lessons from proving *FiniteNatInduction*

- Hide the definition of the induction predicate before appealing to the induction theorem

Lessons from proving *FiniteNatInduction*

- Hide the definition of the induction predicate before appealing to the induction theorem
  - More generally, when applying a proof rule containing a NEW $Q(\_)$ that must be instantiated with some operator *Op*, you should hide the definition of *Op*

Lessons from proving *FiniteNatInduction*

- Hide the definition of the induction predicate before appealing to the induction theorem
  - More generally, when applying a proof rule containing a NEW $Q(\_)$ that must be instantiated with some operator *Op*, you should hide the definition of *Op*
- Use SUFFICES to change the goal

Lessons from proving *FiniteNatInduction*

- Hide the definition of the induction predicate before appealing to the induction theorem
  - More generally, when applying a proof rule containing a NEW $Q(\_)$ that must be instantiated with some operator *Op*, you should hide the definition of *Op*
- Use SUFFICES to change the goal
- Use SUFFICES ASSUME PROVE to disassemble universal quantifiers and implications

Lessons from proving *FiniteNatInduction*

- Hide the definition of the induction predicate before appealing to the induction theorem
  - More generally, when applying a proof rule containing a NEW $Q(\_)$ that must be instantiated with some operator *Op*, you should hide the definition of *Op*
- Use SUFFICES to change the goal
- Use SUFFICES ASSUME PROVE to disassemble universal quantifiers and implications
- Use CASE statements to disassemble the current goal into cases
  - TLAPS will have to be convinced that all cases are covered
  - Often it can figure this out on its own, but sometimes you need to present the fact explicitly

Lessons from proving *FiniteNatInduction*

- Hide the definition of the induction predicate before appealing to the induction theorem
  - More generally, when applying a proof rule containing a NEW $Q(\_)$ that must be instantiated with some operator *Op*, you should hide the definition of *Op*
- Use SUFFICES to change the goal
- Use SUFFICES ASSUME PROVE to disassemble universal quantifiers and implications
- Use CASE statements to disassemble the current goal into cases
  - TLAPS will have to be convinced that all cases are covered
  - Often it can figure this out on its own, but sometimes you need to present the fact explicitly
- Always check to see if TLAPS can prove a fact (given the necessary predicate facts) before plunging into a subproof

Outline

Temporal proofs in TLAPS

```
LEMMA TypeOK_inv == Spec => []TypeOK
<1>1. Init => TypeOK
  BY NAssumption DEF Init, TypeOK, Nodes, Color
<1>2. TypeOK /\ [Next]_vars => TypeOK'
  <2>. USE DEF Nodes, Color
  <2>. SUFFICES ASSUME TypeOK, Next \/ UNCHANGED vars
                PROVE  TypeOK'
    OBVIOUS
  <2>1. CASE InitiateProbe
    BY <2>1, NAssumption DEF InitiateProbe, TypeOK
  <2>2. ASSUME NEW i \in Nodes \ {0}, PassToken(i)
        PROVE  TypeOK'
    BY <2>2, NAssumption DEF PassToken, TypeOK
  <2>3. ASSUME NEW i \in Nodes, SendMsg(i)
        PROVE  TypeOK'
    BY <2>3, NAssumption DEF SendMsg, TypeOK
  <2>4. ASSUME NEW i \in Nodes, Deactivate(i)
        PROVE  TypeOK'
    BY <2>4 DEF Deactivate, TypeOK
  <2>5. CASE UNCHANGED vars
    BY <2>5 DEF vars, TypeOK
  <2>. QED  BY <2>1, <2>2, <2>3, <2>4, <2>5 DEF Next
<1>3. QED
  BY <1>1, <1>2, PTL DEF Spec
```

- A standard safety proof
  - validation of a temporal formula
  - mostly action reasoning
  - temporal reasoning for validating QED step

Temporal proofs in TLAPS

```
LEMMA TypeOK_inv == Spec => []TypeOK
<1>1. Init => TypeOK
  BY NAssumption DEF Init, TypeOK, Nodes, Color
<1>2. TypeOK /\ [Next]_vars => TypeOK'
  <2>. USE DEF Nodes, Color
  <2>. SUFFICES ASSUME TypeOK, Next \/ UNCHANGED vars
                PROVE  TypeOK'
    OBVIOUS
  <2>1. CASE InitiateProbe
    BY <2>1, NAssumption DEF InitiateProbe, TypeOK
  <2>2. ASSUME NEW i \in Nodes \ {0}, PassToken(i)
        PROVE  TypeOK'
    BY <2>2, NAssumption DEF PassToken, TypeOK
  <2>3. ASSUME NEW i \in Nodes, SendMsg(i)
        PROVE  TypeOK'
    BY <2>3, NAssumption DEF SendMsg, TypeOK
  <2>4. ASSUME NEW i \in Nodes, Deactivate(i)
        PROVE  TypeOK'
    BY <2>4 DEF Deactivate, TypeOK
  <2>5. CASE UNCHANGED vars
    BY <2>5 DEF vars, TypeOK
  <2>. QED  BY <2>1, <2>2, <2>3, <2>4, <2>5 DEF Next
<1>3. QED
  BY <1>1, <1>2, PTL DEF Spec
```

- A standard safety proof
  - validation of a temporal formula
  - mostly action reasoning
  - temporal reasoning for validating QED step
- In this talk:
  - Why

    quantified temporal formulas can be proved using

    first-order and propositional temporal backends

Temporal proofs in TLAPS

```
LEMMA TypeOK_inv == Spec => []TypeOK
<1>1. Init => TypeOK
  BY NAssumption DEF Init, TypeOK, Nodes, Color
<1>2. TypeOK /\ [Next]_vars => TypeOK'
  <2>. USE DEF Nodes, Color
  <2>. SUFFICES ASSUME TypeOK, Next \/ UNCHANGED vars
                PROVE  TypeOK'
    OBVIOUS
  <2>1. CASE InitiateProbe
    BY <2>1, NAssumption DEF InitiateProbe, TypeOK
  <2>2. ASSUME NEW i \in Nodes \ {0}, PassToken(i)
        PROVE  TypeOK'
    BY <2>2, NAssumption DEF PassToken, TypeOK
  <2>3. ASSUME NEW i \in Nodes, SendMsg(i)
        PROVE  TypeOK'
    BY <2>3, NAssumption DEF SendMsg, TypeOK
  <2>4. ASSUME NEW i \in Nodes, Deactivate(i)
        PROVE  TypeOK'
    BY <2>4 DEF Deactivate, TypeOK
  <2>5. CASE UNCHANGED vars
    BY <2>5 DEF vars, TypeOK
  <2>. QED  BY <2>1, <2>2, <2>3, <2>4, <2>5 DEF Next
<1>3. QED
  BY <1>1, <1>2, PTL DEF Spec
```

- A standard safety proof
  - validation of a temporal formula
  - mostly action reasoning
  - temporal reasoning for validating QED step
- In this talk:
  - Why

    quantified temporal formulas can be proved using first-order and propositional temporal backends
  - How

    to write the proofs correctly

Temporal proofs in TLAPS

```
LEMMA TypeOK_inv == Spec => []TypeOK
<1>1. Init => TypeOK
  BY NAssumption DEF Init, TypeOK, Nodes, Color
<1>2. TypeOK /\ [Next]_vars => TypeOK'
  <2>. USE DEF Nodes, Color
  <2>. SUFFICES ASSUME TypeOK, Next \/ UNCHANGED vars
                PROVE  TypeOK'
    OBVIOUS
  <2>1. CASE InitiateProbe
    BY <2>1, NAssumption DEF InitiateProbe, TypeOK
  <2>2. ASSUME NEW i \in Nodes \ {0}, PassToken(i)
        PROVE  TypeOK'
    BY <2>2, NAssumption DEF PassToken, TypeOK
  <2>3. ASSUME NEW i \in Nodes, SendMsg(i)
        PROVE  TypeOK'
    BY <2>3, NAssumption DEF SendMsg, TypeOK
  <2>4. ASSUME NEW i \in Nodes, Deactivate(i)
        PROVE  TypeOK'
    BY <2>4 DEF Deactivate, TypeOK
  <2>5. CASE UNCHANGED vars
    BY <2>5 DEF vars, TypeOK
  <2>. QED  BY <2>1, <2>2, <2>3, <2>4, <2>5 DEF Next
<1>3. QED
  BY <1>1, <1>2, PTL DEF Spec
```

- A standard safety proof
  - validation of a temporal formula
  - mostly action reasoning
  - temporal reasoning for validating QED step
- In this talk:
  - Why

    quantified temporal formulas can be proved using

    first-order and propositional temporal backends
  - How

    to write the proofs correctly
  - Which

    formulas can be proved using that
    - Note: TLA^+ is not complete for quantified temporal logic.

Temporal concepts in TLA$^+$

```
(*********
--algorithm Simple {
   variables x = 0; {
   while (TRUE) {
    x := x + 3;
   }}
}
********)
\* BEGIN TRANSLATION
VARIABLE x
vars == << x >>
Init == x = 0
Next == x' = x + 3
Spec == Init /\ [][Next]_vars
\* END TRANSLATION
------------------------
P == x >= 0
THEOREM Spec => []P
```

Temporal concepts in TLA$^+$

```
(*********
--algorithm Simple {
   variables x = 0; {
   while (TRUE) {
    x := x + 3;
   }}
}
********)
\* BEGIN TRANSLATION
VARIABLE x
vars == << x >>
Init == x = 0
Next == x' = x + 3
Spec == Init /\ [][Next]_vars
\* END TRANSLATION
-----------------------
P == x >= 0
THEOREM Spec => []P
```

- Semantics
  - Program Executions

Temporal concepts in TLA$^+$

```
(*********
--algorithm Simple {
  variables x = 0; {
  while (TRUE) {
   x := x + 3;
  }}
}
*********)
\* BEGIN TRANSLATION
VARIABLE x
vars == << x >>
Init == x = 0
Next == x' = x + 3
Spec == Init /\ [][Next]_vars
\* END TRANSLATION
------------------------
P == x >= 0
THEOREM Spec => []P
```

- Semantics
  - Program Executions
  - States

Temporal concepts in TLA$^+$

```
(********
--algorithm Simple {
   variables x = 0; {
   while (TRUE) {
    x := x + 3;
   }}
}
********)
\* BEGIN TRANSLATION
VARIABLE x
vars == << x >>
Init == x = 0
Next == x' = x + 3
Spec == Init /\ [][Next]_vars
\* END TRANSLATION
------------------------
P == x >= 0
THEOREM Spec => []P
```

- Semantics
  - Program Executions
  - States
  - Behaviors and suffixes

Temporal concepts in TLA$^+$

```
(*********
--algorithm Simple {
   variables x = 0; {
   while (TRUE) {
    x := x + 3;
   }}
}
*********)
\* BEGIN TRANSLATION
VARIABLE x
vars == << x >>
Init == x = 0
Next == x' = x + 3
Spec == Init /\ [][Next]_vars
\* END TRANSLATION
------------------------
P == x >= 0
THEOREM Spec => []P
```

- Semantics
  - Program Executions
  - States
  - Behaviors and suffixes
- Syntax
  - Constant expressions

Temporal concepts in TLA$^+$

```
(*********
--algorithm Simple {
   variables x = 0; {
   while (TRUE) {
    x := x + 3;
   }}
}
********)
\* BEGIN TRANSLATION
VARIABLE x
vars == << x >>
Init == x = 0
Next == x' = x + 3
Spec == Init /\ [][Next]_vars
\* END TRANSLATION
-----------------------
P == x >= 0
THEOREM Spec => []P
```

- Semantics
  - Program Executions
  - States
  - Behaviors and suffixes
- Syntax
  - Constant expressions
  - State expressions

Temporal concepts in TLA$^+$

```
(*********
--algorithm Simple {
   variables x = 0; {
   while (TRUE) {
    x := x + 3;
   }}
}
*********)
\* BEGIN TRANSLATION
VARIABLE x
vars == << x >>
Init == x = 0
Next == x' = x + 3
Spec == Init /\ [][Next]_vars
\* END TRANSLATION
------------------------
P == x >= 0
THEOREM Spec => []P
```

- Semantics
  - Program Executions
  - States
  - Behaviors and suffixes
- Syntax
  - Constant expressions
  - State expressions
  - Action expressions

Temporal concepts in TLA$^+$

```
(*********
--algorithm Simple {
   variables x = 0; {
   while (TRUE) {
    x := x + 3;
   }}
}
*********)
\* BEGIN TRANSLATION
VARIABLE x
vars == << x >>
Init == x = 0
Next == x' = x + 3
Spec == Init /\ [][Next]_vars
\* END TRANSLATION
-----------------------
P == x >= 0
THEOREM Spec => []P
```

- Semantics
  - Program Executions
  - States
  - Behaviors and suffixes
- Syntax
  - Constant expressions
  - State expressions
  - Action expressions
  - Temporal expressions

Temporal concepts in TLA$^+$

```
(********
--algorithm Simple {
  variables x = 0; {
  while (TRUE) {
   x := x + 3;
  }}
}
********)
\* BEGIN TRANSLATION
VARIABLE x
vars == << x >>
Init == x = 0
Next == x' = x + 3
Spec == Init /\ [][Next]_vars
\* END TRANSLATION
------------------------
P == x >= 0
THEOREM Spec => []P
```

- Semantics
  - Program Executions
  - States
  - Behaviors and suffixes
- Syntax
  - Constant expressions
  - State expressions
  - Action expressions
  - Temporal expressions
- Logic
  - First-order [1]

[1] Coalescing: Syntactic Abstraction for Reasoning in First-Order Modal Logics

Temporal concepts in TLA$^+$

```
(********
--algorithm Simple {
   variables x = 0; {
   while (TRUE) {
    x := x + 3;
   }}
}
********)
\* BEGIN TRANSLATION
VARIABLE x
vars == << x >>
Init == x = 0
Next == x' = x + 3
Spec == Init /\ [][Next]_vars
\* END TRANSLATION
------------------------
P == x >= 0
THEOREM Spec => []P
```

- Semantics
  - Program Executions
  - States
  - Behaviors and suffixes
- Syntax
  - Constant expressions
  - State expressions
  - Action expressions
  - Temporal expressions
- Logic
  - First-order [1]
  - Temporal

[1] Coalescing: Syntactic Abstraction for Reasoning in First-Order Modal Logics

Temporal concepts in TLA+

```
(*********
--algorithm Simple {
   variables x = 0; {
   while (TRUE) {
    x := x + 3;
   }}
}
*********)
\* BEGIN TRANSLATION
VARIABLE x
vars == << x >>
Init == x = 0
Next == x' = x + 3
Spec == Init /\ [][Next]_vars
\* END TRANSLATION
-----------------------
P == x >= 0
THEOREM Spec => []P
```

- Semantics
  - Program Executions
  - States
  - Behaviors and suffixes
- Syntax
  - Constant expressions
  - State expressions
  - Action expressions
  - Temporal expressions
- Logic
  - First-order [1]
  - Temporal
    - PTL

[1] Coalescing: Syntactic Abstraction for Reasoning in First-Order Modal Logics

Breaking temporal formulas into action formulas

- Proving quantified temporal formulas from action formulas and propositional temporal rules.

Breaking temporal formulas into action formulas

- Proving quantified temporal formulas from action formulas and propositional temporal rules.
  - find a temporal rule

Breaking temporal formulas into action formulas

- Proving quantified temporal formulas from action formulas and propositional temporal rules.
  - find a temporal rule
  - verify the rule

Breaking temporal formulas into action formulas

- Proving quantified temporal formulas from action formulas and propositional temporal rules.
  - find a temporal rule
  - verify the rule
  - understand failures

How to find the rules

- Safety properties - based on variations of the inductive invariant rule:

```
THEOREM Inductive_Invariant ==
    ASSUME STATE I, STATE V,
           ACTION N, STATE P,
           I => P,
           P /\ [N]_V => P'
    PROVE I /\ [][N]_V => []P
```

How to find the rules

- Safety properties - based on variations of the inductive invariant rule:

```
THEOREM Inductive_Invariant ==
    ASSUME STATE I, STATE V,
           ACTION N, STATE P,
           I => P,
           P /\ [N]_V => P'
    PROVE I /\ [][N]_V => []P
```

```
LEMMA TypeOK_inv == Spec => []TypeOK
<1>1. Init => TypeOK
<1>2. TypeOK /\ [Next]_vars => TypeOK'
<1>3. QED
  BY <1>1, <1>2, PTL DEF Spec
```

How to find the rules

- Safety properties - based on variations of the inductive invariant rule:

```
THEOREM Inductive_Invariant ==
    ASSUME STATE I, STATE V,
           ACTION N, STATE P,
           I => P,
           P /\ [N]_V => P'
    PROVE I /\ [][N]_V => []P
```

```
LEMMA TypeOK_inv == Spec => []TypeOK
<1>1. Init => TypeOK
<1>2. TypeOK /\ [Next]_vars => TypeOK'
<1>3. QED
  BY <1>1, <1>2, PTL DEF Spec
```

```
THEOREM Spec => []MutualExclusion
<1>1. Init => Inv
<1>2. Inv /\ [Next]_vars => Inv'
<1>3. Inv => MutualExclusion
<1>4. QED
  BY <1>1, <1>2, <1>3, PTL DEF Spec
```

How to find the rules

- Safety properties - based on variations of the inductive invariant rule:

```
THEOREM Inductive_Invariant ==
    ASSUME STATE I, STATE V,
           ACTION N, STATE P,
           I => P,
           P /\ [N]_V => P'
    PROVE I /\ [][N]_V => []P
```

```
LEMMA TypeOK_inv == Spec => []TypeOK
<1>1. Init => TypeOK
<1>2. TypeOK /\ [Next]_vars => TypeOK'
<1>3. QED
  BY <1>1, <1>2, PTL DEF Spec
```

```
THEOREM Spec => []MutualExclusion
<1>1. Init => Inv
<1>2. Inv /\ [Next]_vars => Inv'
<1>3. Inv => MutualExclusion
<1>4. QED
  BY <1>1, <1>2, <1>3, PTL DEF Spec
```

```
THEOREM Spec => []StructOK1
<1>. USE DEFS Ballot, TypeOK, StructOK1
<1>1. Init => StructOK1
<1>2. TypeOK /\ StructOK1 /\ [Next]_vars => StructOK1'
<1>q. QED
  BY ONLY <1>1, <1>2, typing, PTL DEF Spec
```

How to find the rules

- Safety properties - based on variations of the inductive invariant rule:

```
THEOREM Inductive_Invariant ==
    ASSUME STATE I, STATE V,
           ACTION N, STATE P,
           I => P,
           P /\ [N]_V => P'
    PROVE I /\ [][N]_V => []P
```

- Other properties - other rules

```
LEMMA TypeOK_inv == Spec => []TypeOK
<1>1. Init => TypeOK
<1>2. TypeOK /\ [Next]_vars => TypeOK'
<1>3. QED
  BY <1>1, <1>2, PTL DEF Spec
```

```
THEOREM Spec => []MutualExclusion
<1>1. Init => Inv
<1>2. Inv /\ [Next]_vars => Inv'
<1>3. Inv => MutualExclusion
<1>4. QED
  BY <1>1, <1>2, <1>3, PTL DEF Spec
```

```
THEOREM Spec => []StructOK1
<1>. USE DEFS Ballot, TypeOK, StructOK1
<1>1. Init => StructOK1
<1>2. TypeOK /\ StructOK1 /\ [Next]_vars => StructOK1'
<1>q. QED
  BY ONLY <1>1, <1>2, typing, PTL DEF Spec
```

Are the rules sound?

```
LEMMA TypeOK_inv == Spec => []TypeOK
<1>1. Init => TypeOK
<1>2. TypeOK /\ [Next]_vars => TypeOK'
<1>3. QED
  BY <1>1, <1>2, PTL DEF Spec
```

Are the rules sound?

```
LEMMA TypeOK_inv == Spec => []TypeOK
<1>1. Init => TypeOK
<1>2. TypeOK /\ [Next]_vars => TypeOK'
<1>3. QED
  BY <1>1, <1>2, PTL DEF Spec
```

- Rule is an instance of the PTL rule:

```
THEOREM Inductive_Invariant ==
    ASSUME STATE I, STATE V,
           ACTION N, STATE P,
           I => P,
           P /\ [N]_V => P'
    PROVE I /\ [][N]_V => []P
```

Are the rules sound?

```
LEMMA TypeOK_inv == Spec => []TypeOK
<1>1. Init => TypeOK
<1>2. TypeOK /\ [Next]_vars => TypeOK'
<1>3. QED
  BY <1>1, <1>2, PTL DEF Spec
```

- Rule is an instance of the PTL rule:

```
THEOREM Inductive_Invariant ==
   ASSUME STATE I, STATE V,
          ACTION N, STATE P,
          I => P,
          P /\ [N]_V => P'
   PROVE I /\ [][N]_V => []P
```

- Success of PTL backend verifies this

Understanding failures

```
LEMMA TypeOK_inv == ASSUME Spec PROVE []TypeOK
<1>1. Init => TypeOK
<1>2. TypeOK /\ [Next]_vars => TypeOK'
<1>3. QED
  BY <1>1, <1>2, PTL DEF Spec
```

- Consider this valid lemma

Understanding failures

```
LEMMA TypeOK_inv == ASSUME Spec PROVE []TypeOK
<1>1. Init => TypeOK
<1>2. TypeOK /\ [Next]_vars => TypeOK'
<1>3. QED
  BY <1>1, <1>2, PTL DEF Spec
```

- Consider this valid lemma
- which seems to be an instance of the PTL rule:

```
THEOREM Inductive_Invariant ==
   ASSUME STATE I, STATE V,
          ACTION N, STATE P,
          I /\ [][N]_V,
          I => P,
          P /\ [N]_V => P'
   PROVE  []P
```

Understanding failures

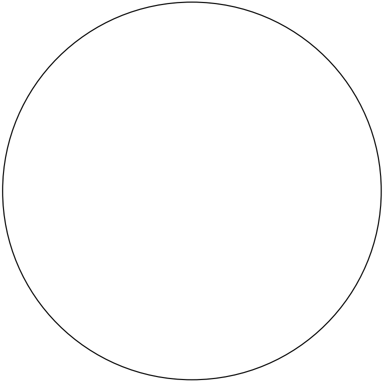
```
LEMMA TypeOK_inv == ASSUME Spec PROVE []TypeOK
<1>1. Init => TypeOK
<1>2. TypeOK /\ [Next]_vars => TypeOK'
<1>3. QED
  BY <1>1, <1>2, PTL DEF Spec
```

- Consider this valid lemma
- which seems to be an instance of the PTL rule:

```
THEOREM Inductive_Invariant ==
  ASSUME STATE I, STATE V,
         ACTION N, STATE P,
         I /\ [][N]_V,
         I => P,
         P /\ [N]_V => P'
  PROVE  []P
```

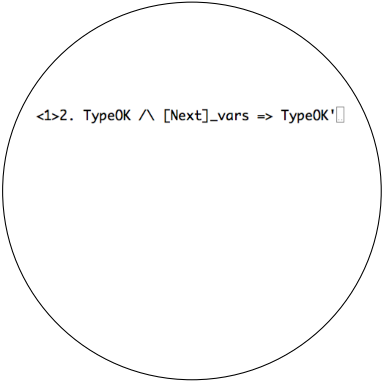
- But it is not, why?

Necessitation



```
LEMMA TypeOK_inv == Spec => []TypeOK
<1>1. Init => TypeOK
<1>2. TypeOK /\ [Next]_vars => TypeOK'
<1>3. QED
  BY <1>1, <1>2, PTL DEF Spec
```

Necessitation




```
LEMMA TypeOK_inv == Spec => []TypeOK
<1>1. Init => TypeOK
<1>2. TypeOK /\ [Next]_vars => TypeOK'
<1>3. QED
  BY <1>1, <1>2, PTL DEF Spec
```

Necessitation

```
<1>2. TypeOK /\ [Next]_vars => TypeOK'

<1>2. [](TypeOK /\ [Next]_vars => TypeOK')
```

```
LEMMA TypeOK_inv == Spec => []TypeOK
<1>1. Init => TypeOK
<1>2. TypeOK /\ [Next]_vars => TypeOK'
<1>3. QED
  BY <1>1, <1>2, PTL DEF Spec
```

- Since $\langle 1 \rangle 2$ holds in all behaviours, it can be boxed
- This is called necessitation

Necessitation

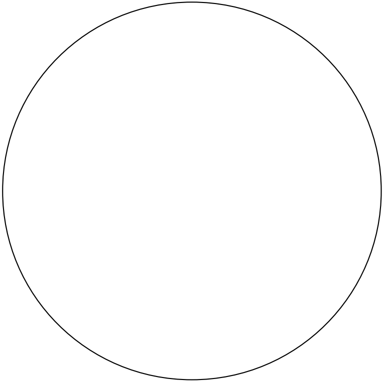
```
<1>2. TypeOK /\ [Next]_vars => TypeOK'

<1>2. [](TypeOK /\ [Next]_vars => TypeOK')
```

```
LEMMA TypeOK_inv == Spec => []TypeOK
<1>1. Init => TypeOK
<1>2. TypeOK /\ [Next]_vars => TypeOK'
<1>3. QED
  BY <1>1, <1>2, PTL DEF Spec
```

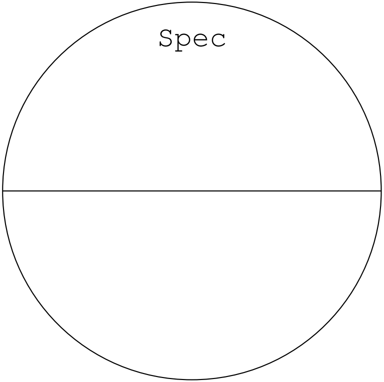
- Since $\langle 1 \rangle 2$ holds in all behaviours, it can be boxed
- This is called necessitation
- The PTL rules normally requires the application of necessitation on the action steps

Necessitation



```
LEMMA TypeOK_inv == ASSUME Spec PROVE []TypeOK
<1>1. Init => TypeOK
<1>2. TypeOK /\ [Next]_vars => TypeOK'
<1>3. QED
  BY <1>1, <1>2, PTL DEF Spec
```

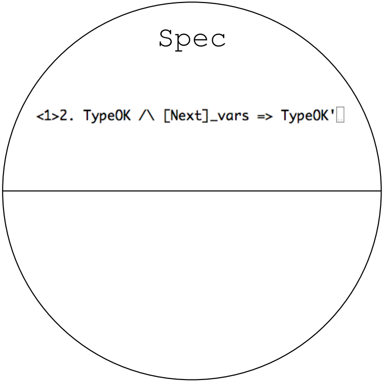
Necessitation




```
LEMMA TypeOK_inv == ASSUME Spec PROVE []TypeOK
<1>1. Init => TypeOK
<1>2. TypeOK /\ [Next]_vars => TypeOK'
<1>3. QED
  BY <1>1, <1>2, PTL DEF Spec
```

- Spec is assumed when proving the proof steps

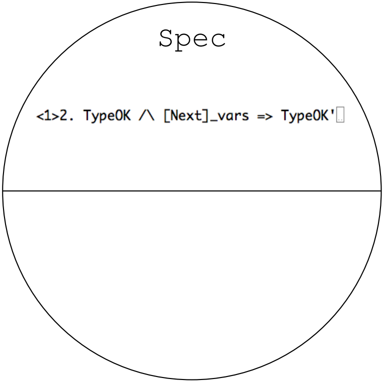
Necessitation




```
LEMMA TypeOK_inv == ASSUME Spec PROVE []TypeOK
<1>1. Init => TypeOK
<1>2. TypeOK /\ [Next]_vars => TypeOK'
<1>3. QED
  BY <1>1, <1>2, PTL DEF Spec
```

- Spec is assumed when proving the proof steps
- $\langle 1 \rangle 2$ doesn't hold in all behaviours

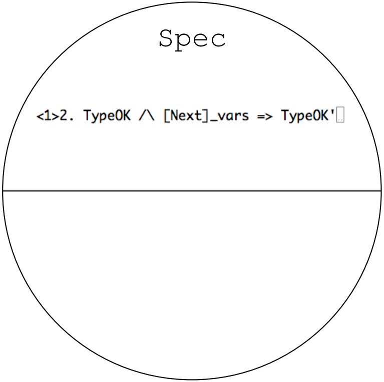
Necessitation




```
LEMMA TypeOK_inv == ASSUME Spec PROVE []TypeOK
<1>1. Init => TypeOK
<1>2. TypeOK /\ [Next]_vars => TypeOK'
<1>3. QED
  BY <1>1, <1>2, PTL DEF Spec
```

- Spec is assumed when proving the proof steps
- $\langle 1 \rangle 2$ doesn't hold in all behaviours
- Necessitation is not applied

Necessitation




```
LEMMA TypeOK_inv == ASSUME Spec PROVE []TypeOK
<1>1. Init => TypeOK
<1>2. TypeOK /\ [Next]_vars => TypeOK'
<1>3. QED
  BY <1>1, <1>2, PTL DEF Spec
```

- Spec is assumed when proving the proof steps
- $\langle 1 \rangle 2$ doesn't hold in all behaviours
- Necessitation is not applied
- Note: There is a workaround

Necessitation and assumptions

```
VARIABLE x

THEOREM ASSUME x=0 PROVE [][x'=x+1]_x => [](x \in {0,1})
<1>1. x=0 => x \in {0,1}
  OBVIOUS
<1>2. x\in {0,1} /\ x'=x+1 => x' \in {0,1}
  OBVIOUS
<1>3. QED BY <1>1,<1>2,PTL
```

- Consider the following clearly invalid claim

Necessitation and assumptions

```
VARIABLE x

THEOREM ASSUME x=0 PROVE [][x'=x+1]_x => [](x \in {0,1})
<1>1. x=0 => x \in {0,1}
  OBVIOUS
<1>2. x\in {0,1} /\ x'=x+1 => x' \in {0,1}
  OBVIOUS
<1>3. QED BY <1>1,<1>2,PTL
```

- Consider the following clearly invalid claim
- The rule is again an instance of the previous PTL rule

Necessitation and assumptions

```
VARIABLE x

THEOREM ASSUME x=0 PROVE [][x'=x+1]_x => [](x \in {0,1})
<1>1. x=0 => x \in {0,1}
  OBVIOUS
<1>2. x\in {0,1} /\ x'=x+1 => x' \in {0,1}
  OBVIOUS \* using the assumption x=0
<1>3. QED BY <1>1,<1>2,PTL
```

- Consider the following clearly invalid claim
- The rule is again an instance of the previous PTL rule
- The two hypothesis are valid but the rule is not sound
- Why? Necessitation fails for $\langle 1 \rangle 2$

Necessitation and assumptions

```
VARIABLE x

THEOREM ASSUME x=0 PROVE □[x'=x+1]_x => □(x \in {0,1})
<1>1. x=0 => x \in {0,1}
  OBVIOUS
<1>2. x\in {0,1} /\ x'=x+1 => x' \in {0,1}
  OBVIOUS \* using the assumption x=0
<1>3. QED BY <1>1,<1>2,PTL
```

Obligation 1 – status : ls4 : failed

Stop Proving | Goto Obligation

```
ASSUME NEW VARIABLE x,
       x = 0 (* non-□ *),
       x = 0 => x \in {0, 1} (* non-□ *),
       x \in {0, 1} /\ x' = x + 1 => x' \in {0, 1} (* non-□ *)
PROVE  □[x' = x + 1]_x => □(x \in {0, 1})
```

- Consider the following clearly invalid claim
- The rule is again an instance of the previous PTL rule
- The two hypothesis are valid but the rule is not sound
- Why? Necessitation fails for $\langle 1 \rangle 2$
- Confusing? Necessitation failures are reported in the obligation window

Necessitation and assumptions II

```
CONSTANT x

THEOREM ASSUME x=0 PROVE [][x'=x+1]_x => [](x \in {0,1})
<1>1. x=0 => x \in {0,1}
  OBVIOUS
<1>2. x\in {0,1} /\ x'=x+1 => x' \in {0,1}
  OBVIOUS
<1>3. QED BY <1>1,<1>2,PTL
```

- Now, the claim is valid, even if in a trivial way

Necessitation and assumptions II

```
CONSTANT x

THEOREM ASSUME x=0 PROVE [][x'=x+1]_x => [](x \in {0,1})
<1>1. x=0 => x \in {0,1}
  OBVIOUS
<1>2. x\in {0,1} /\ x'=x+1 => x' \in {0,1}
  OBVIOUS
<1>3. QED BY <1>1,<1>2,PTL
```

- Now, the claim is valid, even if in a trivial way
- The proof is idential to the previous one

Necessitation and assumptions II

```
CONSTANT x

THEOREM ASSUME x=0 PROVE [][x'=x+1]_x => [](x \in {0,1})
<1>1. x=0 => x \in {0,1}
  OBVIOUS
<1>2. x\in {0,1} /\ x'=x+1 => x' \in {0,1}
  OBVIOUS
<1>3. QED BY <1>1,<1>2,PTL
```

- Now, the claim is valid, even if in a trivial way
- The proof is idential to the previous one
- This time, necessitation is applied

Necessitation and assumptions II

```
CONSTANT x

THEOREM ASSUME x=0 PROVE [][x'=x+1]_x => [](x \in {0,1})
<1>1. x=0 => x \in {0,1}
  OBVIOUS
<1>2. x\in {0,1} /\ x'=x+1 => x' \in {0,1}
  OBVIOUS
<1>3. QED BY <1>1,<1>2,PTL
```

- Now, the claim is valid, even if in a trivial way
- The proof is idential to the previous one
- This time, necessitation is applied
- What is the difference?

Boxable assumptions

- Assumptions P, such that $P \Leftrightarrow \Box P$, allow for necessitation.

Boxable assumptions

- Assumptions P, such that $P \Leftrightarrow \Box P$, allow for necessitation.
- We determine this using the following `is_box` algorithm:

```
» is_?(A,B,... |- F) -> is_?(F)
» is_?(F) -> True if F is constant
» is_?(True | False) -> True
» is_box([]F) -> True
» is_diamond(<>F) -> True
» is_?([]F | <>F | F') -> is_?(F)
» is_box(~F) -> is_diamond(F)
» is_diamond(~F) -> is_box(F)
» is_?(F /\ G | F \/ G) -> is_?(F) /\ is_?(G) and similarly for lists
» is_box(F -> G) -> is_diamond(F) /\ is_box(G) and similarly for {{is_diamond(F -> G)} |
  is_?(F <-> G)}
» is_?(op(a,b,...)) -> is_?(a) /\ is_?(b) /\ ...
» else False
```

Boxable assumptions

- Assumptions P, such that $P \Leftrightarrow \Box P$, allow for necessitation.
- We determine this using the following `is_box` algorithm:

```
» is_?(A,B,... |- F) -> is_?(F)
» is_?(F) -> True if F is constant
» is_?(True | False) -> True
» is_box([]F) -> True
» is_diamond(<>F) -> True
» is_?([]F | <>F | F') -> is_?(F)
» is_box(~F) -> is_diamond(F)
» is_diamond(~F) -> is_box(F)
» is_?(F /\ G | F \/ G) -> is_?(F) /\ is_?(G) and similarly for lists
» is_box(F -> G) -> is_diamond(F) /\ is_box(G) and similarly for {{is_diamond(F -> G)} |
  is_?(F <-> G)}
» is_?(op(a,b,...)) -> is_?(a) /\ is_?(b) /\ ...
» else False
```

- An assumption proved in the scope of a non-boxed assumption is considered as non-boxed as well

Conclusion

- TLA$^+$ proofs for quantified temporal formulas - Why and How

Conclusion

- TLA$^+$ proofs for quantified temporal formulas - Why and How
- Which:

Conclusion

- TLA$^+$ proofs for quantified temporal formulas - Why and How
- Which:
  - for all safety properties

Conclusion

- TLA$^+$ proofs for quantified temporal formulas - Why and How
- Which:
  - for all safety properties
  - for liveness properties - still require:

Conclusion

- TLA$^+$ proofs for quantified temporal formulas - Why and How
- Which:
  - for all safety properties
  - for liveness properties - still require:
    - reasoning about ENABLED

Conclusion

- TLA$^+$ proofs for quantified temporal formulas - Why and How
- Which:
  - for all safety properties
  - for liveness properties - still require:
    - reasoning about ENABLED
    - some proofs require full quantified temporal reasoning - Ex: $\forall x.\Box P(x) \Leftrightarrow \Box\forall x.P(x)$