The TLA$^+$ Proof Track

10 About Proofs and Proving

10.1 About Proofs

When writing proofs of properties of systems or algorithms, we should not look at mathematicians' proofs for models. There are three reasons why the way (almost all) mathematicians write proofs do not work for our proofs:

- The theorems mathematicians prove are very different from what we are proving. Mathematical theorems are usually deep, being based on knowledge that has been developed over centuries. Our theorems are shallow but wide. They generally use only simple mathematics, but require checking many details.

- Mathematicians don't care if their theorems are not quite correct. Omitting a simple hypothesis such as that a certain set is nonempty would not even be considered an error. For the proof of an algorithm, such an omission means a bug in a "corner case".

- The proofs mathematicians write are unreliable. Anecdotal evidence suggests that a significant fraction of published, refereed mathematical papers contain incorrect theorems—not ones ignoring corner cases, but results that mathematicians would consider wrong. Based on a tiny amount of data, I would guess that fraction to be more than a tenth. (This includes results that have a correct proof but are wrong because the proof relies on theorems that are wrong.)

  Mathematicians depend on the social process to weed out incorrect results. Most theorems are ignored and soon forgotten. The few important results are scrutinized by many mathematicians, and errors in them are eventually discovered. Two false 19^{th} century proofs of the four-color theorem were believed for 11 years. However, the greater number of mathematicians and improved communications make it very likely that an incorrect proof of such a major result would today be quickly discovered.

When a mathematician's style of proof is used to prove properties of systems, the result is often disastrous. There is no social process to find errors an engineer makes in a proof of a system she is designing. Even for published algorithms, the social process doesn't work very well. One dramatic example is provided by:

> Pamela Zave. *Lightweight verification of network protocols: The case of Chord.* AT&T Technical Report, January 2010.

The Chord algorithm was first published in:

I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. *Chord: A scalable peer-to-peer lookup service for Internet applications.* Proceedings of SIGCOMM. ACM, August 2001.

This paper states:

> Three features that distinguish Chord from many other peer-to-peer lookup protocols are its simplicity, provable correctness, and provable performance.

? ← → C I S

However, Zave reports:

> [T]he Chord routing protocol is neither proven nor correct. The only published proof of correctness excludes failures from consideration. Even within its scope the proof does not compel belief, due to ill-defined terms and missing or unjustified steps. The full protocol is clearly incorrect, even after bugs with straightforward fixes have been eliminated. Not one of the six properties claimed invariant for the full protocol ... is invariantly true.

Chord is quite well known. A web search on its title yields many thousands of references. The paper won a SIGCOMM Test of Time Award in 2011. Yet, the errors in the algorithm went unnoticed for almost 10 years. Zave found them by writing a formal specification of the algorithm (in the Alloy language) and applying a form of model checking. What distinguishes Chord from most published algorithms is that someone applied formal methods to check if it really was correct.

One way to make hand proofs more reliable is to structure them hierarchically. The *Principles* track of this hyperbook uses such structured proofs. Structuring is equally important in handling the complexity of a formal proof, and formal proofs written in TLA$^+$ are hierarchically structured.

10.2 About TLAPS

This track is about writing formal TLA$^+$ proofs that can be checked by TLAPS, the TLA$^+$ Proof System. TLAPS is a separate system that is called by the TLA$^+$ Toolbox. It is not distributed with the Toolbox; you can download it from here.

TLA$^+$ proofs are written in a declarative style that has the goal of making proofs independent of the proof engines used to check them. A proof specifies what facts are needed to justify a proof step; it does not say what to do with those facts. The writer of a TLA$^+$ proof should not have to know how the actual proof checking works. TLAPS has been designed to come as close as we could to achieving this goal.

In TLAPS, a proof manager translates the proof into a set of separate proof obligations, checks the ones that are trivially true, and sends the others to one

or more backend provers. Its default behavior is to try these three backend provers: *SMT*, *Zenon*, and *Isabelle*—in that order.

The TLAPS distribution includes a library of TLA modules that contain useful theorems that assert properties of mathematical objects like sets, functions, and sequences, as well as theorems that assert proof rules such as ones for mathematical induction. How to access library modules in your specifications is explained on the help page for the TLA$^+$ preferences page.

The proof manager remembers if it has already proved an obligation. Its default behavior is not to try to prove an obligation that has already been proved. TLAPS stores a record of what obligations have been proved in a *fingerprint* file. For a module named M, the fingerprint file is the file named *fingerprints* inside the folder (directory) *M.tlaps*. TLAPS may try to prove an obligation that it has already proved for any of the following reasons: you chose to override the default (using the Launch Prover command), the fingerprint file is not present (for example, if you are using a different computer), or some change to TLAPS has made the information in the fingerprint file obsolete.

Because the correctness of a proof obligation can depend on many things, some not at all obvious, fingerprinting is subtle. There have been bugs that cause the Toolbox to report that an obligation has been proved despite a change to the specification making the obligation no longer provable. If any part of the specification has changed since you proved something, you should reprove it without using fingerprints to be sure that it is still proved. In particular, it's a good idea to do this after you have finished your complete proof.

Even if nothing has changed, a proof that succeeded may fail if tried again, or vice-versa. There are two possible reasons for this.

- Whether or not a backend prover succeeds before it times out can depend on what computer you are using and on what other programs are running.
- If you are using a different version of TLAPS, then a TLAPS library file or a backend prover may have changed. We keep trying to make the libraries and the backend provers better. However, automatic proving involves tradeoffs, and making a library or a prover work better on some proofs may make it worse on others. We hope that any change we make helps more often than it hurts, but we cannot guarantee that it won't make one of your proofs fail.

11 Correctness of Euclid's Algorithm

We introduce the prover using Euclid's algorithm of Section 4 as an example. If you have not already done so, you should read the informal proof of Euclid's algorithm's correctness in Section 4.9. You should also read about stuttering in Section 6.7.2.

Here is the specification we wrote in Section 4, including the definitions of the invariants used in the proof, and here is the ASCII version. It calls module *GCD*; here is the ASCII version of that module. Open the spec in the Toolbox. You will have to run the PlusCal translator to make the parsing errors go away.

11.1 Proving Safety

The correctness property we will prove is that *PartialCorrectness* is an invariant of the algorithm. This is expressed by the formula $Spec \Rightarrow \Box Correctness$. As we saw in Section 4, to do this we have to prove conditions $I1$, $I2$, and $I3$—except in light of stuttering we must replace $I2$ by

$$Inv \wedge [Next]_{vars} \Rightarrow Inv'$$

Conditions $I1$–$I3$ are the first three steps of the following proof, which you should add to the end of module *Euclid*.

THEOREM $Spec \Rightarrow \Box PartialCorrectness$ ASCII version
$\langle 1 \rangle 1.\ Init \Rightarrow Inv$
$\langle 1 \rangle 2.\ Inv \wedge [Next]_{vars} \Rightarrow Inv'$
$\langle 1 \rangle 3.\ Inv \Rightarrow PartialCorrectness$
$\langle 1 \rangle 4.$ QED

The proof of a theorem is a sequence of steps, the last step being a QED step. Each step may be followed by its proof. Of course, the theorem hasn't been proved until all of these four steps have been proved. We can prove the steps in any order, or work on several of them at once. However, it's usually a good idea to prove the QED step first.

QED means *that which was to be proved*, which in this proof is $Spec \Rightarrow \Box PartialCorrectness$. The proof of the QED step must show that the truth of this formula follows from the truth of what is already known, which are the laws of mathematics and steps $\langle 1 \rangle 1$–$\langle 1 \rangle 3$. If we think that the proof of a statement should be obvious, we can use the proof OBVIOUS. So, write the word `OBVIOUS` after `QED`. I like to write a step's proof indented, beginning on the line after the step; so I would type this as

```
<1>4. QED
  OBVIOUS
```

Indentation and line breaks are ignored in a proof—except as they affect the meaning of a list of conjuncts or disjuncts within an individual expression. However, you should begin each step on a new line, otherwise some Toolbox commands may not behave properly.

Any proof can begin with the optional word PROOF, so we can also write

```
<1>4. QED
  PROOF OBVIOUS
```

and

```
PROOF
<1>1. Init => Inv
  ...
```

Have TLAPS check the proof of the QED step by putting the cursor in the step or its proof and executing the Prove Step or Module command either by right-clicking and choosing it on the menu, or by typing control+g control+g.

The proof fails, shading the step (approximately) red and raising a window showing the obligation that the backend provers failed to prove—which in this case is

```
ASSUME NEW CONSTANT M,
       NEW CONSTANT N,
       NEW VARIABLE x,
       NEW VARIABLE y,
       NEW VARIABLE pc
PROVE  Spec  =>  []PartialCorrectness
```

This obligation is exactly what the backend provers are trying to prove. They are trying to show that the goal, which is the formula in the PROVE section, follows from the list of assumptions in the ASSUME section together with the ordinary laws of mathematics. That's all the backend provers know. They know nothing about the spec or about anything else in the proof.

Imagine printing out this obligation, giving it to some mathematician you have never seen before, and expecting her to verify that the goal follows from the assumptions. Obviously, she couldn't. How could she possibly determine the truth of a formula containing the symbols *Spec* and *PartialCorrectness* knowing only that M and N are constants and x, y, and pc are variables?

The truth of the goal follows from the truth of $\langle 1\rangle 1$–$\langle 1\rangle 3$. So, we have to tell the provers to use those facts. To do that, we replace the proof OBVIOUS with

```
BY <1>1, <1>2, <1>3
```

The proof again fails because the backend provers can't prove the obligation:

```
ASSUME NEW CONSTANT M,
        ...
       NEW VARIABLE pc,
       Init  =>  Inv,
       Inv  /\  [Next]_vars  =>  Inv',
       Inv  =>  PartialCorrectness
PROVE  Spec  =>  []PartialCorrectness
```

Can you see what the mathematician would say if you gave here this obligation? She'd say, "How can I prove a formula containing the symbol *Spec* if none of the assumptions even mention that symbol?" To prove the goal, we have to replace the symbol *Spec* with its definition. We tell TLAPS to do that by adding a DEF clause to the proof:

```
BY <1>1, <1>2, <1>3 DEF Spec
```

The proof still fails, this time with the obligation:

```
ASSUME NEW CONSTANT M,
        ...
       NEW VARIABLE pc,
       Init  =>  Inv,
       Inv  /\  [Next]_vars  =>  Inv',
       Inv  =>  PartialCorrectness
PROVE  Init  /\  [][Next]_vars  =>  []PartialCorrectness
```

From everything you've read so far, it should be clear that the goal follows from the three formulas in the ASSUME list. Why did the proof fail? Suppose we asked the mathematician if the goal follows from the assumptions? If she were a randomly chosen mathematician, she would probably know nothing about temporal logic or TLA$^+$, and she would say, "I don't know if that's true because I don't know what $\Box$ or $[Next]_{vars}$ mean." Sometimes the backend provers also try to tell you why they can't prove it in the window showing the obligation, on the line that begins *Obligation 1*. But this time, they say nothing. Most of the time, the only reason reported by a backend that makes any sense to you or me is *timeout*, indicating that it timed out. In that case, the prover might be able to prove the obligation if given more time, but it seldom can.

The default provers handle action reasoning—ordinary formulas that may contain primed and unprimed variables. They know what $[Next]_{vars}$ means because that's an action formula, but not what $\Box$ means. To prove our QED step, we use the *PTL* backend. (*PTL* stands for Propositional Temporal Logic.) To do this, we add the "fact" *PTL*. The symbol *PTL* is defined in the *TLAPS* library module. The actual definition in that module is $PTL \triangleq$ TRUE , but a bit of magic that has nothing to do with TLA$^+$ makes putting *PTL* in a BY clause

cause TLAPS to send the obligation to *PTL* rather than the default backends. The proof

```
BY <1>1, <1>2, <1>3, PTL DEF Spec
```

works, and TLAPS colors the QED step green.

Let's now prove the first step. We will have to tell the TLAPS to expand the definitions of *Init* and *Inv*, and expanding the definition of *Inv* requires expanding the definition of *TypeOK* as well. So, we can try the proof

```
BY DEF Init, Inv, TypeOK
```

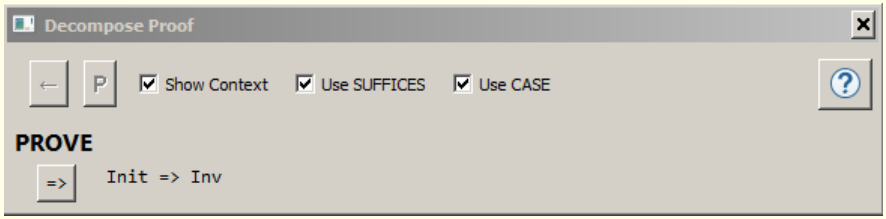
We can write DEFS instead of DEF.

(The order of items in a DEF clause makes no difference.) TLAPS fails, reporting this obligation.

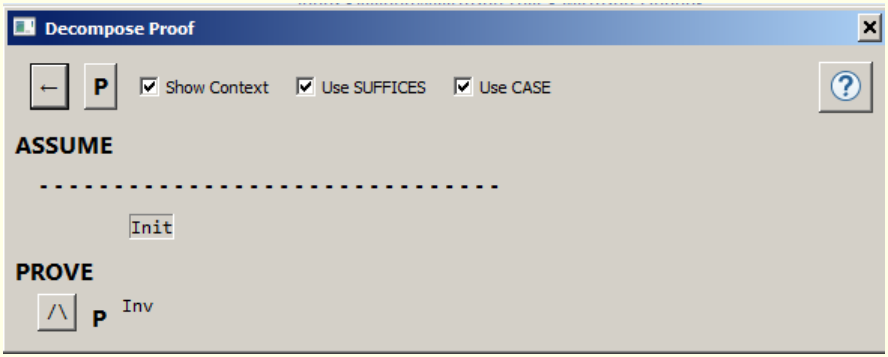
```
ASSUME NEW CONSTANT M,
        ...
       NEW VARIABLE pc
PROVE  (/\ x  =  M
        /\ y  =  N
        /\ pc  =  "Lbl_1")
        =>
       (/\ /\ x  \in  Nat  \  {0}
           /\ y  \in  Nat  \  {0}
        /\ GCD(x, y)  =  GCD(M, N)
        /\ pc  =  "Done"  =>  x  =  y)
```

If TLAPS fails to prove an obligation this simple, it's almost always because you've forgotten to tell it to use some fact or definition that it needs. We haven't told it to expand the definition of *GCD*, but this isn't necessary because $GCD(x, y) = GCD(M, N)$ follows from $x = M$ and $y = N$. Can you see what the problem is?

If you can't see the problem quickly, the quickest way to find it is usually to break the proof into steps. The logical structure of what we're trying to prove tells us how to do that. We're trying to prove that *Init* implies a conjunction, and this is done by proving separately that *Init* implies each of the conjuncts. The Toolbox's Decompose Proof (control+g control+r) command allows you to do that with a few mouse clicks. Put the cursor anywhere in that step or its proof and execute the command. The Toolbox raises a window like this:



The only thing you can do next is click on the button labeled =>, so do it. The window changes to:



This tells you that the decomposition so far is to change what is to be proved from $Init \Rightarrow Inv$ to

ASSUME $Init$
PROVE Inv

By itself, this accomplishes nothing. The implication and the ASSUME/PROVE are logically equivalent. However, the button labeled /\ next to Inv tells you that Inv is a conjunction and you can use that to decompose the proof. The P next to the button means that clicking on it generates the decomposition. Uncheck the Use SUFFICES and Use CASE options first and then click on that button. This replaces the proof with:

$\langle 2 \rangle 1$. ASSUME $Init$
PROVE $TypeOK$
BY $\langle 2 \rangle 1$ DEF $Init$, Inv, $TypeOK$
$\langle 2 \rangle 2$. ASSUME $Init$
PROVE $GCD(x, y) = GCD(M, N)$
BY $\langle 2 \rangle 2$ DEF $Init$, Inv, $TypeOK$
$\langle 2 \rangle 3$. ASSUME $Init$
PROVE $(pc = \text{“Done”}) \Rightarrow (x = y)$
BY $\langle 2 \rangle 3$ DEF $Init$, Inv, $TypeOK$
$\langle 2 \rangle 4$. QED
BY $\langle 2 \rangle 1$, $\langle 2 \rangle 2$, $\langle 2 \rangle 3$ DEF Inv

Note that the original proof has been put into the proofs of steps $\langle 2 \rangle 1$–$\langle 2 \rangle 3$. Rather surprisingly, each of those steps seems to be using itself in its own proof. The $\langle 2 \rangle 1$ in the proof of step $\langle 2 \rangle 1$ actually refers to the assumption $Init$ of the ASSUME clause. Like any other fact, we must explicitly tell TLAPS when to use the assumptions in an ASSUME clause. Within the proof of an ASSUME/PROVE step, the name of the step refers to the ASSUME clause's assumptions.

Save the module and run TLAPS on this proof by putting the cursor in step $\langle 1\rangle 1$ and executing Prove Step or Module (control+g control+g). Everything succeeds except the proof of *TypeOK*, which fails to prove the obligation:

```
ASSUME NEW CONSTANT M,
       NEW CONSTANT N,
       ...
       /\ x  =  M
       /\ y  =  N
       /\ pc  =  "Lbl_1" (now)
PROVE  /\ x  \in  Nat  \  {0}
       /\ y  \in  Nat  \  {0}
```

The problem should now be clear. This obligation is true only because the module asserts the assumptions that M and N are in $Nat \setminus \{0\}$, but this assumption doesn't appear in the obligation. We have to tell TLAPS to use it.

In a BY clause, instead of referring to a fact by its name, you can give the fact itself. For example, you can replace $\langle 2\rangle 2$ in the proof of $\langle 2\rangle 2$ with *Init*. However, the fact has to be stated in the syntactically identical form as its assertion in the module. TLAPS proves step $\langle 2\rangle 1$ with the proof

$$\begin{array}{l}\text{BY } \langle 2\rangle 1, \quad \land\, M \in Nat \setminus \{0\} \\ \qquad\qquad\ \land\, N \in Nat \setminus \{0\} \\ \quad \text{DEF } Init,\ Inv,\ TypeOK\end{array}$$

but not with the proof

$$\text{BY } \langle 2\rangle 1,\ M \in Nat \setminus \{0\} \land N \in Nat \setminus \{0\} \text{ DEF } Init,\ Inv,\ TypeOK$$

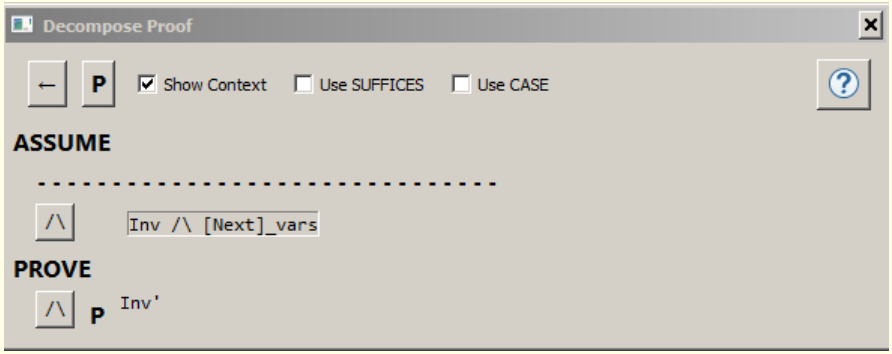
Rather than doing that, let's give the assumption a name. But first, there's no need to decompose the proof; the provers should have no problem proving $\langle 1\rangle 1$ with this additional fact. So, let's undo the decomposition of step $\langle 1\rangle 1$'s proof with the Toolbox's Undo (control+z) command. Now let's give the assumption the name *MNPosInt* by changing it to:

$$\begin{array}{l}\text{ASSUME } MNPosInt \triangleq \ \land\, M \in Nat \setminus \{0\} \\ \qquad\qquad\qquad\qquad\ \land\, N \in Nat \setminus \{0\}\end{array}$$

TLAPS will now verify this proof of step $\langle 1\rangle 1$:

$$\text{BY } MNPosInt \text{ DEF } Init, Inv, TypeOK$$

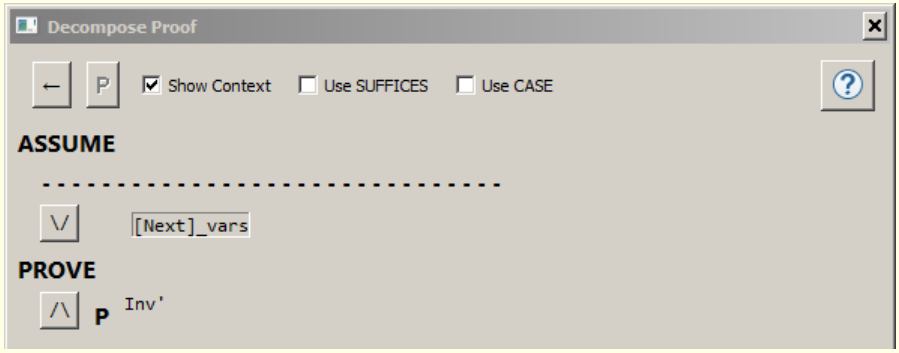
Let's now prove step $\langle 1\rangle 2$, the step that proves that *Inv* is an inductive invariant. This spec is simple enough that this step can be proved with a BY proof. But in most real-life examples, you're going to have to decompose this step, so let's start right away by executing the Decompose Proof command on the step. This provides the only option of a => decomposition, so click on it, producing this window.



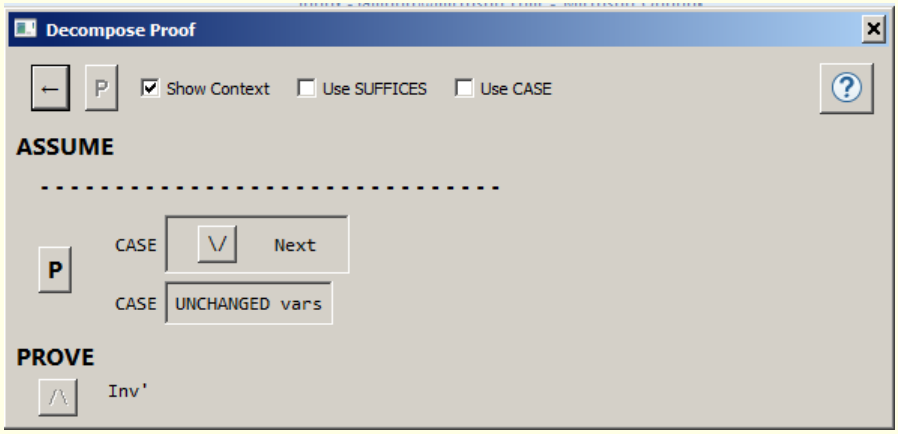
There are two basic ways to decompose this proof.

- We can prove each conjunction of Inv' separately.
- The next-state action *Next* is a disjunction, and we can prove $A_1 \lor \ldots \lor A_n \Rightarrow B$ by proving $A_i \Rightarrow B$ for each i.

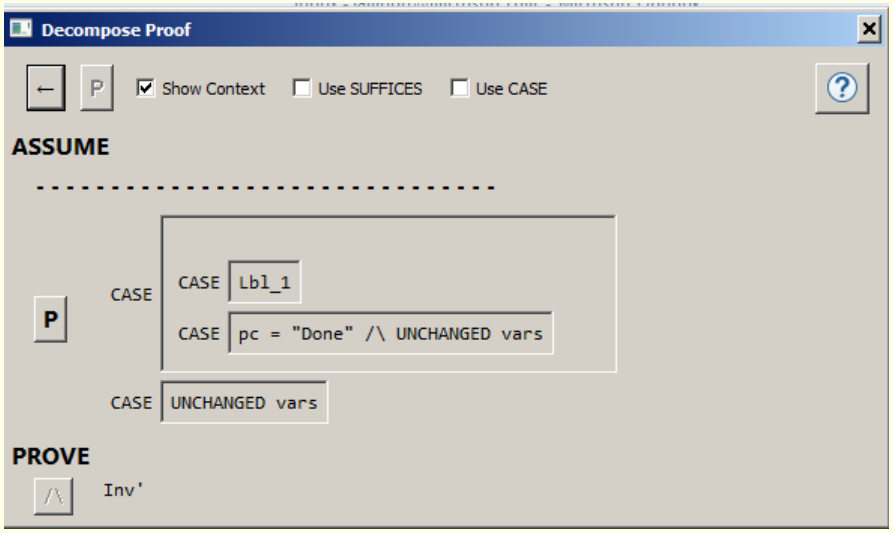
I usually find the second approach easier, though I'm not sure if it really is or if it's just what I'm used to doing. Anyway, let's do it that way. Click on the top `/\` button to turn the assumption $Inv \land [Next]_{vars}$ into the part of it that can be further decomposed:



Click on the `\/` button to get



This tells us that the assumption $[Next]_{vars}$ equals $Next \lor (\text{UNCHANGED } vars)$ and we can split the proof of Inv' into the two cases of assuming $Next$ and of assuming UNCHANGED $vars$. The \/ button next to $Next$ tells us that formula $Next$ is also a disjunction on which we can do a case split. Click on it to produce:



Look at the definition of $Next$ in the algorithm's translation to see that $Next$ is indeed the disjunction of the formulas in the two subcases.

Make sure that the Use SUFFICES and Use CASE options are unchecked and click on the P button, which generates the proof. The generated steps $\langle 2\rangle 1$–$\langle 2\rangle 3$ have no proofs, since there was originally no proof for step $\langle 1\rangle 2$. We're going to need to expand the definition of Inv and $TypeOK$ in all those proofs. So we should have saved ourselves some typing by having a

BY DEF *Inv*, *TypeOK*

proof on step ⟨1⟩1. Undo the decomposition, add that proof to ⟨1⟩1 and redo the decomposition. Read the proof this produced and make sure you understand why ⟨1⟩2 follows from ⟨2⟩1–⟨2⟩3 and the definition of *Next*, as the proof of the QED step asserts. Run the prover on this QED step to see that the backend provers can verify this simple deduction.

Before we prove this, let's use the Decompose Proof command to learn a little more about writing proofs. Undo the decomposition and rerun the decomposition, except this time checking the Use SUFFICES options. This has changed the resulting proof by adding the first step

⟨2⟩ SUFFICES ASSUME *Inv*,
$[Next]_{vars}$
PROVE *Inv*′
OBVIOUS

and removing the assumptions *Inv* and $[Next]_{vars}$ from steps ⟨2⟩1–⟨2⟩3. The SUFFICES step asserts that to prove the goal of ⟨1⟩2, which is $Inv \land [Next]_{vars} \Rightarrow Inv'$, it suffices to assume *Inv* and $[Next]_{vars}$ and prove *Inv*′. It also allows the use of the assumptions *Inv* and $[Next]_{vars}$ in the remaining steps of this proof (the proof of ⟨1⟩2).

Note that the SUFFICES step has a level number, but not a name. Facts asserted by an unnamed step are used by the backend provers without having to be mentioned in a BY step. Run the prover on step ⟨2⟩1. The proof fails because the definition of *Lbl*_1 isn't being used, but examine the obligation. Note that the assumption following the declaration of *pc*—an assumption that is ended by a comma—is *Inv* with its definition and the definition of *TypeOK* expanded. The next assumption is $[Next]_{vars}$. Now give the SUFFICES step a name—say ⟨2⟩*x*. Run the prover once again on step ⟨2⟩1. You'll see that the two assumptions no longer appear in the obligation. You can add those assumptions to the obligation by adding ⟨2⟩*x* to the BY proof.

About step names.

Unnecessary assumptions make it harder for the backend provers to prove an obligation, so unnamed steps should be used with care. An unnecessary assumption like *Inv* has little effect if the definition of *Inv* is not expanded. Similarly, the assumption $[Next]_{vars}$ is pretty harmless if the definitions of *Next* and *vars* are not expanded. So not numbering this SUFFICES step is not a problem.

Now undo the decomposition of the proof of ⟨1⟩2 and redo it, this time with both the Use SUFFICES and Use CASE options checked. Steps ⟨2⟩1–⟨2⟩3 have been changed from ASSUME/PROVE steps to CASE steps. For example,

⟨2⟩1. ASSUME *Lbl*_1
PROVE *Inv*′

has been changed to

⟨2⟩1. CASE *Lbl*_1

In general, the step CASE A is equivalent to the step ASSUME A PROVE G, where G is the current goal. In this case, the current goal for steps ⟨2⟩1–⟨2⟩3 is set to Inv' by the SUFFICES step.

Undo the proof decomposition and run the Decompose Proof command again, now with Use CASE checked and Use SUFFICES unchecked. This time, the proof has no CASE steps. The ASSUME/PROVE steps all have two assumptions, so they can't be turned into CASE steps. I think the version of the proof with SUFFICES and CASE is the nicest one.

Using whichever decomposition you prefer, let's now check the proof. We obviously need to add *Lbl*_1 to the DEF clause in the proof of ⟨2⟩1 and *vars* to the DEF clauses in the proofs of ⟨2⟩2 and ⟨2⟩3. Do that and run the prover on the proof. You can either run the Prove Step or Module command on step ⟨1⟩2 (by putting the cursor in that step) or on the entire theorem (by putting the cursor on the statement of the theorem) or on the entire module (by putting the cursor outside any theorem or proof). Because of its use of fingerprints, TLAPS doesn't spend any significant amount of time rechecking proofs it has already checked. (It does take time to compute the obligations, so you don't want to run TLAPS unnecessarily on more than a few dozen steps.) It will not check any steps that have no proof, coloring those steps yellow.

TLAPS proves everything except step ⟨2⟩1, which is the only non-trivial one. Examining the obligation, we see two problems. First, we've forgotten our assumption *MNPosInt* telling us that M and N are positive integers. Second, the obligation can't be proved without knowing something about *GCD*. We can try adding *MNPosInt* to the BY clause and expanding the definition of *GCD*, which requires also expanding the definitions of *SetMax*, *DivisorsOf*, and *Divides*. Do that and try the proof again. It fails. The obligation is complicated enough that we'd have a hard time trying to understand it. However, the backend provers can handle obligations that large—especially the SMT solvers. The SMT solvers are especially good at dealing with arithmetic formulas that involves inequalities, addition, and subtraction. However, this obligation also involves multiplication. (The other backends are pretty bad at any kind of arithmetic.) Furthermore, the obligation contains a number of CHOOSE expressions and existential quantifiers, both of which can trip up the provers.

We could further decompose the proof of ⟨2⟩1 by trying to prove each of the conjuncts of Inv' separately. We would find that the conjunct whose proof fails is $(GCD(x, y) = GCD(M, N))'$, which is equivalent to $(GCD(x', y') = GCD(M, N))$. We have reached the point where we can't get any further by blindly decomposing the proof. We have to think about why $GCD(x', y')$ equals $GCD(M, N)$. Going back to the informal proof, we remember that correctness of Euclid's algorithm rests on properties *GCD*1—*GCD*3, which are defined in the module *GCD*. In fact, the invariance of *Inv* is proved using *GCD*2 and *GCD*3. The proof of ⟨2⟩1 succeeds if we add *GCD*2 and *GCD*3 to the BY proof. In fact,

there's no need to decompose step ⟨1⟩2. TLAPS verifies it with the proof

BY *MNPosInt*, *GCD*2, *GCD*3 DEF *Inv*, *TypeOK*, *Next*, *Lbl_1*, *vars*

You should be able to prove step ⟨1⟩3, using *GCD*1.

11.2 Proving Properties of the GCD

Checking the correctness of algorithms and systems is hard enough. You should not have to check the correctness of mathematical theorems. That should be the job of mathematicians. So, normally you would not bother to get TLAPS to check the correctness of theorems *GCD*1–*GCD*3 of the *GCD* module. If a mathematical result is not very simple, then you should be able to find it in a math book. You just have to translate the result from the informal math used by mathematicians to a TLA$^+$ theorem. To avoid errors in formalizing the result, you should check the theorem with TLC.

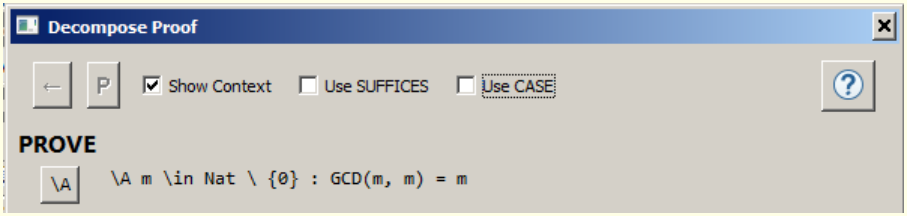
However, our topic here is how to write proofs, and the proofs of *GCD*1–*GCD*3 are instructive. So, let's write them. Open module *GCD* in the Toolbox. You don't need to make it a separate specification; you can open it within the *Euclid* spec with the Open Module command in the Toolbox's File menu. (Another way to go to the *GCD* module is to apply the Goto Declaration (F3) command to the identifier *GCD*, which will go to its definition within that module. The Return from Goto Declaration (F4) command returns to the original cursor position.)

Let's start trying to prove each of the theorems *GCD*1–*GCD*3 with a BY proof. We expand the definition of *GCD* down to its TLA$^+$ primitive operators, which is done by the following proof:

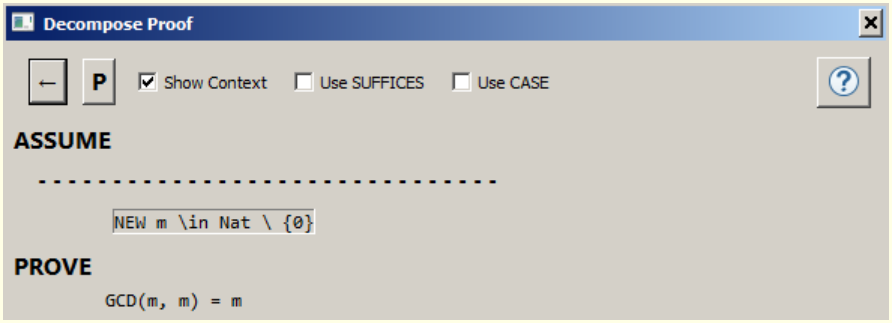
BY DEF *GCD*, *SetMax*, *DivisorsOf*, *Divides*

Add that proof after each of the three theorems and run the prover on all of them (by executing the Prove Step or Module command with the cursor outside any theorem or proof). This succeeds in proving only *GCD*2. In fact, *GCD*2 follows from the top-level definition of *GCD* and commutativity of set union; there is no need to expand the definitions of *SetMax* or *DivisorsOf*. Remove the unnecessary definition expansions from the proof of *GCD*2 and check that TLAPS still proves it.

Let's prove *GCD*1 next. To decompose the proof, let's use the Decompose Proof command. It raises this window



which gives us only one option: clicking on the `\A` button. This changes the dialog to



which tells us that we can prove the goal by assuming that m is an element of $Nat \setminus \{0\}$ and proving $GCD(m, m) = m$. Check the Use SUFFICES option and click on the P button to produce this proof.

$\langle 1 \rangle$ SUFFICES ASSUME NEW $m \in Nat \setminus \{0\}$
PROVE $GCD(m, m) = m$
OBVIOUS
$\langle 1 \rangle$ QED
BY DEF GCD, $SetMax$, $DivisorsOf$, $Divides$

Run the Prove Step or Module command on the theorem. It proves the SUFFICES step, but not the QED step. Proving the SUFFICES step means proving that assuming its assumptions and proving its PROVE formula proves the current goal, which is the statement of the theorem. This is trivial because the ASSUME/PROVE is equivalent to the theorem. Naturally, transforming the theorem's goal to something obviously equivalent isn't going to enable the backend provers to prove it, so the proof of the QED step fails. What the SUFFICES step accomplishes is to remove the quantifier, allowing us to prove $GCD(m, n) = m$ for a particular choice of m.

The Decompose Proof command can take us no further; we have to start thinking. To show that m is the GCD of m and m, we have to show that (1) m divides m and (2) m is the largest number that divides m. This suggests that we add the following two steps after the SUFFICES step

$\langle 1 \rangle 1$. $Divides(m, m)$
$\langle 1 \rangle 2$. $\forall\, i \in Nat : Divides(i, m) \Rightarrow (i \leq m)$

Before trying to prove them, it's best to check that they are all we need. So, add the facts $\langle 1 \rangle 1$ and $\langle 1 \rangle 2$ to the BY clause of the QED step's proof and see if TLAPS will prove that step.

Why number these steps?

TLAPS does prove the QED step. If it hadn't, we would have had to do some more thinking to see why not. Our thinking about the proof should have led us

to realize that proving the theorem from $\langle 1\rangle 1$ and $\langle 1\rangle 2$ doesn't require expanding the definition of *Divides*. Expanding the definition made the formula that had to be proved more complicated than necessary. An unnecessary definition expansion can make the difference between a proof succeeding and failing. So, remove *Divides* from the DEF clause and check that TLAPS still proves the QED step.

Now we have to prove steps $\langle 1\rangle 1$ and $\langle 1\rangle 2$. We have no useful facts at our disposal; all we can do is tell the TLAPS to expand the definition of *Divides*. So, add the proof

```
BY DEF Divides
```

to both of them and try proving them both by telling TLAPS to prove the entire theorem. TLAPS proves $\langle 1\rangle 1$ and $\langle 1\rangle 2$, thus proving the theorem.

Finally, we must prove theorem *GCD*3. Run the Decompose Proof command, clicking on the `\A` button, then the `=>` button, and then the `P` button to produce this proof.

```
⟨1⟩ SUFFICES ASSUME NEW m ∈ Nat \ {0}, NEW n ∈ Nat \ {0},
                    n > m
             PROVE  GCD(m, n) = GCD(m, n − m)
  OBVIOUS
⟨1⟩ QED
    BY DEF GCD, DivisorsOf, SetMax, Divides
```

We should have TLAPS check the proof of the SUFFICES step to make sure that the Decompose Proof command didn't make an error. There's no point wasting time having TLAPS check the QED step because the decomposition so far has been logically trivial. We have to do some thinking to decompose the proof into simpler steps.

That result of that thinking appears in our informal proof of *GCD*3. The first step of that proof states that, to prove the goal $GCD(m, n) = GCD(m, n - m)$ it suffices to prove

$$\forall\, i \in Int : Divides(i,\, m) \land Divides(i,\, n) \equiv Divides(i,\, m) \land Divides(i,\, n - m)$$

Let's see if TLAPS believes that is enough to prove our goal. Insert that as an unnumbered step (labeled by $\langle 1\rangle$) before the QED step and now run the TLAPS on the QED step. It succeeds. (In fact, it's not necessary to expand the definition of *Divides*.)

All we have left to do is to prove the newly added step. You will find that TLAPS successfully checks the proof

```
BY DEF Divides
```

Run the Prove Step or Module command on the entire module to verify that we have indeed proved all three theorems.

12 The Proof Language

Now that you've seen the basics of using TLAPS, it's time to examine proofs and the proof language more closely. Before we do that, let's examine the theorems that we are trying to prove.

12.1 What a Theorem Asserts

A *theorem* consists of one of the equivalent keywords

THEOREM LEMMA COROLLARY PROPOSITION

followed optionally by an identifier and the symbol $\triangleq$, followed by an assertion. A theorem of the form

THEOREM $id \triangleq A$

defines id to equal assertion A. Naming theorems (and assumptions) is a good idea, because it makes them easier to use in a proof.

An *assertion* is either a formula or an assume/prove. An assume/prove has the form

ASSUME $A_1, \ldots, A_n$
PROVE B

where B is a formula and each of the assumptions A_i is either a formula or a declaration. A declaration is something like NEW v or NEW $v \in S$ where v is a variable and S is an expression. The keyword NEW can be replaced by CONSTANT or NEW CONSTANT. The declaration NEW $v \in S$ is almost equivalent to the declaration NEW v followed by the assumption $v \in S$. The only difference is that when it appears in NEW $v \in S$, the formula $v \in S$ is called a *domain assumption*. However, the formula $v \in S$ is not a domain assumption when it appears by itself in a separate ASSUME clause. The significance of domain assumptions is explained below. It's usually best to write NEW $v \in S$ instead of splitting the assumption in two.

TLA$^+$ allows other kinds of declarations, but you'll never write them.

To simplify the exposition, we consider an assertion that is a formula F to be an assume/prove with no assumptions, as if it were written ASSUME PROVE F. (TLA$^+$ does not allow you to write such an assume/prove, so I color it red.)

An assertion asserts (the truth of) a formula. The assertion

ASSUME PROVE F

asserts the formula F. The assertion

ASSUME NEW $x \in S$, $P(x)$
PROVE $Q \lor R(x)$

asserts the formula $\forall\, x \in S : P(x) \Rightarrow (Q \vee R(x))$. The assertion

ASSUME NEW $P(\_)$, NEW x, NEW y, $x = y$
PROVE $P(x) = P(y)$

asserts the formula

$$\forall\, P(\_) : \forall\, x \,:\, \forall\, y \,:\, (x = y) \Rightarrow (P(x) = P(y))$$

This isn't a legal TLA$^+$ formula, since TLA$^+$ doesn't allow quantifying over an operator that takes an argument. I will use such formulas for the purpose of explaining proofs, coloring the illegal parts red.

12.2 The Hierarchical Structure of a Proof

12.2.1 Writing Structured Proofs

A theorem may have a proof. A proof consists of the optional keyword PROOF followed by either:

- A *non-leaf proof* that is a sequence of steps, ending with a QED step, each of which may (but need not) have a proof.
- A *leaf proof*, which is either the keyword OBVIOUS, the keyword OMITTED, or a BY proof.

The leaf proof OMITTED is equivalent to having no proof; use it to indicate that you are deliberately assuming something and have not simply forgotten to prove it.

A step (of a non-leaf proof) begins with a preface token consisting of the following three parts, with no spaces between them:

- A *level specifier* of the form $\langle i \rangle$, where i is a non-negative integer called the *level number*. (It is typed <*i*>.) All steps in a single non-leaf proof must have the same level number. If the step has a non-leaf proof, the steps of that proof must have a level number greater than i.
- A string of digits, letters, and/or _ characters that may be empty. If it is non-empty, the step is said to be *named*, and its *name* is the level specifier followed by this string.
- An optional period (.).

For example, $\langle 3 \rangle 2a.\ 1 + 1 = 3$ is a named level-3 step with name $\langle 3 \rangle 2a$ and assertion ASSUME PROVE $1 + 1 = 3$. If, like me, you prefer to name most of the steps in a non-leaf proof $\langle i \rangle 1$, $\langle i \rangle 2$, ... , see the help page of the Toolbox's Renumber Proof command.

We describe the hierarchical structure of a proof in the usual way as an upside-down tree (with the root on top), where steps at a lower level (deeper) in the proof structure (the tree) have larger level numbers.

12.2.2 Reading Structured Proofs

You may have noticed a little ⊖ next to theorems and proof steps. (For brevity, I will write *step* to mean either a proof step or the statement of a theorem.) Clicking on the ⊖ hides the step's proof, replacing the ⊖ with ⊕. Clicking on the ⊕ undoes the effect of clicking on the ⊖.

There are a number of commands for viewing the proof as hypertext that provide finer control of what is shown than you can get by just clicking on ⊖ and ⊕. The following commands are executed on a step by putting the cursor on the step and either right-clicking and selecting the command or typing the indicated keystrokes.

? ← → C I S

Clicking on ⊖ or ⊕ sometimes doesn't work properly. These other commands should always do what they're supposed to.

Show Current Subtree (control+g control+s)
: Reveals the complete proof of the step.

Hide Current Subtree (control+g control+h)
: Hides the proof of the step.

Show Children Only (control+g control+c)
: Reveals the top level of the step's proof.

Focus on Step (control+g control+f)
: Hides everything except the top level of the step's proof and the siblings of (steps at the same level as) the step and of every ancestor of that step in the proof.

  This is useful when writing the proof because, after executing this command on a step, the steps before it that are shown are precisely the ones that can be referred to in the proof of the step.

The following two commands are performed with the cursor anywhere in the module.

Show All Proofs (control+g control+a)
: Reveals the complete proof of every theorem in the module.

Hide All Proofs (control+g control+n)
: Hides the proof of every theorem in the module. (The Focus on Step command performed outside a proof is equivalent to Hide All Proofs.)

As you have undoubtedly noticed, the Toolbox editor commands having to do with proofs are executed from the keyboard by typing control+g plus another control character. If you just type control+g and wait a second, you will see a list of all the commands you can execute with an additional keystroke.

12.3 The State of a Proof

At each step in a proof, and at each leaf proof, there is a state that consists of the following components:

G A formula that is the current goal of the proof.

$\mathcal{S}$ The sequence of current symbol declarations. Here are examples of symbol declarations:

VARIABLE x CONSTANT C CONSTANT $Op(\_, \_, \_)$
CONSTANT $\_ \oplus \_$

There is one additional kind of symbol declaration: CONSTANT $id \in S$, where id is an identifier and S is an arbitrary expression. We call the formula $id \in S$ the declaration's *domain assumption*.

$\mathcal{F}_K$ The known facts, which is the set of formulas currently asserted by the user to be true, and which can be used to prove new facts.

$\mathcal{F}_U$ The usable facts, which is the subset of $\mathcal{F}_K$ consisting of those facts that are used by default in proofs.

$\mathcal{D}_K$ The set of all user-defined symbols.

$\mathcal{D}_U$ The subset of $\mathcal{D}_K$ containing all user-defined symbols whose definitions are by default expanded in proofs.

$\mathcal{B}$ A sequence of backend provers.

The proof state determines the proof obligations that are sent to the backend provers, and what backend provers they are sent to, as described below in Section 12.4.

The proof-state components other than **G** are also defined at all high-level statements in a module. They are all empty at the beginning of the module, except that $\mathcal{B}$ equals the default sequence $\langle SMT, Zenon, Isa \rangle$ of backend provers. These components are changed by ordinary module statements in the following ways:

CONSTANTS C, $F(\_)$
: Appends the sequence $\langle$CONSTANT C, CONSTANT $F(\_)\rangle$ of declarations to $\mathcal{S}$. In other words, if $\mathcal{S}$ equals $\langle$VARIABLE x, CONSTANT $AB\rangle$ before the declaration, then it equals

$$\langle \text{VARIABLE } x, \text{ CONSTANT } AB, \text{ CONSTANT } C, \text{ CONSTANT } F(\_)\rangle$$

immediately after the declaration. The CONSTANTS statement leaves the other components of the proof state unchanged. A VARIABLES statement has a similar effect.

$f(a) \triangleq \{x, a\}$

adds the symbol f to $\mathcal{D}_K$, but leaves the other components, including $\mathcal{D}_U$, unchanged. Thus, a module's definitions are not usable by default. A function definition has essentially the same effect.

THEOREM $thm \triangleq$ ASSUME NEW $i \in S$, $P(i)$
PROVE $Q(i)$

Adds the formula $\forall\, i \in S : P(i) \Rightarrow Q(i)$ to $\mathcal{F}_K$, adds thm to both $\mathcal{D}_K$ and $\mathcal{D}_U$, and leaves $\mathcal{F}_U$, $\mathcal{S}$, and $\mathcal{B}$ unchanged. Thus, the definition of the theorem name, but not the formula asserted by the theorem, is usable by default. The statement has the same effect without the "$thm \triangleq$" except that $\mathcal{D}_K$ and $\mathcal{D}_U$ are left unchanged. An ASSUME statement (which can also include a definition) has the same effect.

EXTENDS M1, M2

Has the same effect as if the statements of modules $M1$ and $M2$ were inserted at the beginning of the current module.

An INSTANCE statement

The definitions imported by the statement are added to $\mathcal{D}_K$, and the instantiated theorems from the module are added to $\mathcal{F}_K$ as described below. The components $\mathcal{S}$, $\mathcal{D}_U$, $\mathcal{F}_U$, and $\mathcal{B}$ are unchanged.

RECURSIVE $op(\_)$

Has no effect on the proof state.

To specify the proof state at each step and each leaf proof of a theorem's proof, we do two things:

- For a proof step Σ that is not a QED step, we specify the proof state at the next proof step at the same level as Σ. (A QED step ends its level of the proof.)
- For a proof step or theorem that has a proof, we specify the proof state at the beginning of its proof—which is either its leaf proof or the first step of its non-leaf proof .

How every different kind of proof step affects the proof state is described below. Some of the descriptions are given for particular examples of the steps; the generalizations to arbitrary instances of the steps should be obvious.

Remember that a step that has a preface token like $\langle 3 \rangle 14.$ is said to be named, and $\langle 3 \rangle 14$ is its name. A step with a prefix token like $\langle 3 \rangle$ is unnamed. Similarly, a theorem that begins THEOREM $thm \triangleq$ is said to be named and have the name thm. Other theorems are said to be unnamed. In the following lists of proof-step statements, the preface tokens are omitted.

12.3.1 Steps That Can Have a Proof

ASSUME NEW $i \in S$, $j \in T$
PROVE $Q(i)$

This can be either a step or the statement of a THEOREM. If it is a step, then the proof context of the next step at the same level is obtained from the context at the step as follows:

- The formula $\forall\, i \in S : (j \in T) \Rightarrow Q(i)$ asserted by the assume/prove is added to $\mathcal{F}_K$. If the step is unnamed, this formula is also added to $\mathcal{F}_U$; otherwise $\mathcal{F}_U$ is unchanged.
- If the step is named, then its name is added to $\mathcal{D}_K$ and $\mathcal{D}_U$; otherwise $\mathcal{D}_K$ and $\mathcal{D}_U$ are unchanged.
- **G**, $\mathcal{S}$, and $\mathcal{B}$ are left unchanged.

If the step or theorem has a proof, the proof context at the beginning of the proof is obtained from the context at the step or theorem as follows:

- The current goal **G** becomes $Q(i)$.
- The declaration CONSTANT $i \in S$ is appended to $\mathcal{S}$.
- The formula $j \in T$ is added to $\mathcal{F}_K$. If this is a theorem or an unnamed step, then this formula is also added to $\mathcal{F}_U$; otherwise $\mathcal{F}_U$ is unchanged.
- If the step or theorem is named, then its name is added to $\mathcal{D}_K$ and $\mathcal{D}_U$; otherwise $\mathcal{D}_K$ and $\mathcal{D}_U$ are unchanged.
- $\mathcal{B}$ is unchanged.

If the step is named $\langle 3 \rangle 14$, then:

- Within the step's proof, $\langle 3 \rangle 14$ names the formula $j \in T$. The name can be used only as a fact—for example, in a BY proof. A formula that contains the name, such as $(j > i) \Rightarrow \langle 3 \rangle 14$, is illegal.
- Starting from the next step at the current level until the end of the current-level proof, $\langle 3 \rangle 14$ names the formula $\forall\, i \in S : (j \in T) \Rightarrow Q(i)$ asserted by the assume/prove.

Observe that facts that would be added to the set $\mathcal{F}_U$ of usable facts if the step were unnamed are not added if those facts can be named. The philosophy behind this is that the user should state explicitly (usually by name) what facts are needed to prove each step. This makes the proof easier for humans to understand and for backend provers to check.

Unlike unnamed steps, unnamed theorems are not usable by default. This means that adding a new theorem, whether named or not, will not affect

the proofs of later theorems. (Adding a new usable fact can't invalidate a proof, but it can make it harder for a prover to check it, causing TLAPS to fail to check the proof.) Since unnamed theorems can't be referred to by name, using them in a proof is inconvenient.

QED

The context at the start of a QED step's proof is the same as for the step that simply asserted **G**, the step's current goal. No step follows a QED step at the same level.

SUFFICES ASSUME NEW $i \in S$, $j \in T$
PROVE $Q(i)$

The proof context of the next step at the same level is obtained from the context at the step as follows:

- The current goal **G** becomes $Q(i)$.
- The declaration CONSTANT $i \in S$ is appended to $\mathcal{S}$.
- The formula $j \in T$ is added to $\mathcal{F}_K$. If this is an unnamed step, then the formula is also added to $\mathcal{F}_U$; otherwise $\mathcal{F}_U$ is unchanged.
- If the step is named, then its name is added to $\mathcal{D}_K$ and $\mathcal{D}_U$; otherwise $\mathcal{D}_K$ and $\mathcal{D}_U$ are unchanged.
- $\mathcal{B}$ is unchanged.

If the step has a proof, the proof context at the beginning of the proof is obtained from the context at the step as follows:

- The current goal **G** is unchanged.
- The formula $\forall\, i \in S : (j \in T) \Rightarrow Q(i)$ asserted by the assume/prove is added to $\mathcal{F}_K$ and $\mathcal{F}_U$.
- If the step is named, then its name is added to $\mathcal{D}_K$ and $\mathcal{D}_U$; otherwise $\mathcal{D}_K$ and $\mathcal{D}_U$ are unchanged.
- **G**, $\mathcal{S}$, and $\mathcal{B}$ are left unchanged.

If the step is named $\langle 3 \rangle 14$, then:

- Starting from the next step at the same level until the end of the current-level proof, $\langle 3 \rangle 14$ names the formula $j \in T$.
- Within the step's proof, $\langle 3 \rangle 14$ names $\forall\, i \in S : (j \in T) \Rightarrow Q(i)$, the formula asserted by the assume/prove.

Observe that there is a duality between an assume/prove step and a SUFFICES assume/prove step. The proof state at the beginning of the proof of an assume/prove step is the state after the SUFFICES assume/prove step and its proof, and vice-versa. This reflects the fact that by renumbering steps, we can convert a proof

⟨3⟩14. ASSUME NEW $i \in S,\ j \in T$
PROVE $Q(i)$

Level-4 proof of $Q(i)$ using assumptions $i \in S$ and $j \in T$.

Rest of level-3 proof that proves **G** using $\forall\, i \in S : (j \in T) \Rightarrow Q(i)$.

to the equivalent proof

⟨3⟩14. SUFFICES ASSUME NEW $i \in S,\ j \in T$
PROVE $Q(i)$

Level-4 proof of **G**, using $\forall\, i \in S : (j \in T) \Rightarrow Q(i)$.

Rest of level-3 proof that proves $Q(i)$ using assumptions $i \in S$ and $j \in T$.

The proof that $\forall\, i \in S : (j \in T) \Rightarrow Q(i)$ implies **G** is usually a simple leaf proof; the proof of $Q(i)$ is often complicated, requiring multiple levels. Therefore, the SUFFICES proof usually has one fewer level. The main function of the SUFFICES construct is to reduce the depth of proofs.

CASE F

This step is equivalent to the step

ASSUME F PROVE **G**

where **G** is the current goal at the step.

PICK $i \in S,\ j \in T : P(i, j)$

This step produces the same proof state at the next statement at the same level as the step

SUFFICES ASSUME NEW $i \in S$, NEW $j \in T$
$P(i, j)$
PROVE **G**

having the same prefix token as the PICK step, and where **G** is the current goal at the step. It produces the same state at the beginning of the step's proof as the step

$\exists\, i \in S, j \in T \ :\ P(i, j)$

Thus, to prove the step, you have to prove the existence of $i \in S$ and $j \in T$ satisfying $P(i, j)$. After the step, i and j are declared to be constants, with domain assumptions $i \in S$ and $j \in T$, formula $P(i, j)$ is added to the known facts, and the current goal remains the same. (As with the corresponding assume/prove step, whether $P(i, j)$ is usable depends on whether the step is named.)

In general, a PICK statement can be anything that is a legal expression if the PICK is replaced by $\exists$—for example:

PICK $i, j \in S,\ k \in T : P(i, j, k)$

PICK $i,\ j,\ k : (i \notin j) \wedge Q(j, k)$

The meaning of these statements and their effect on the proof state should be clear.

12.3.2 Steps That Cannot Have a Proof

For a step that cannot have a proof, we need describe only how it changes the proof state at the step to obtain the proof state at the following step (which must be at the same level).

The USE and HIDE Steps

These two steps modify the sets $\mathcal{F}_U$ and $\mathcal{D}_U$ of usable facts and definitions; a USE step can also modify $\mathcal{B}$. They can appear either as a proof step with a preface token, or as a top-level module statement with no preface token. When they appear as a step, it makes no difference whether they are named or not. A USE or HIDE step can have a name, but that name can't be used anywhere.

USE $\langle 2\rangle 2$, *Isa*, $i > 1$, *thm*, *SMT* DEF $F, \oplus$

where *thm* is a theorem name. The step adds to $\mathcal{F}_U$ the formula $i > 1$ and the facts named by the step name $\langle 2\rangle 2$ and the theorem name *thm*. It adds to $\mathcal{D}_U$ the symbols F and $\oplus$, which must be in $\mathcal{D}_K$. It leaves $\mathbf{G}$, $\mathcal{S}$, $\mathcal{F}_K$, and $\mathcal{D}_K$ unchanged, and it makes $\mathcal{B}$ equal $\langle \mathit{Isa}, \mathit{SMT}\rangle$. (A USE step that specifies no backend provers leaves $\mathcal{B}$ unchanged.) This step produces the same proof obligations as the step

$\langle 3\rangle 14$. TRUE
BY $\langle 2\rangle 2$, *Isa*, $i > 1$, *thm*, *SMT* DEF $F, \oplus$

The keywords DEF and DEFS are equivalent.

Remember that the "facts" *Isa* and *SMT* specify backend provers.

HIDE $\langle 2\rangle 2$, *thm* DEF $F, \oplus$

Removes from $\mathcal{F}_U$ the facts named by the step name $\langle 2\rangle 2$ and the theorem name *thm*. It removes from $\mathcal{D}_U$ the symbols F and $\oplus$. It leaves $\mathbf{G}$, $\mathcal{S}$, $\mathcal{F}_K$, and $\mathcal{D}_K$ unchanged.

Observe that while a USE step can add arbitrary formulas to $\mathcal{F}_U$, a HIDE step can remove only named facts from $\mathcal{F}_U$.

The DEFINE Step

This step makes definitions that are local to the current level of the proof and its subproofs.

DEFINE $f(a) \triangleq a + 1$
$g \triangleq f(42) * b$

adds to $\mathcal{D}_K$ and $\mathcal{D}_U$ the symbols f and g, which have the specified definitions everywhere within the scope of the DEFINE—which is the rest of the current proof (and its subproofs). The other proof-state components are unchanged. The step may be named, but its name should not be used.

We may change TLAPS to make the step name refer to both definitions when used in the DEF clause of a USE or HIDE step.

Observe that, unlike ordinary definitions in the module, definitions made in a DEFINE step are usable by default. They can be hidden (removed from $\mathcal{D}_U$) with a HIDE step.

Other Steps That Cannot Have a Proof

WITNESS $n - 2 \in Nat,\ 2 * m \in 1 \,..\, n$

For this step to be legal, the current goal **G** must be obviously equivalent to

G0. $\exists\, a \in S,\ b \in T\ :\ P(a, b)$

for some identifiers a and b, expressions S and T, and operator P. For example, **G** might be the formula

G1. $\exists\, i, j \in Int\ :\ i + j \leq 3 * (n + 1)$

To prove G0, it suffices to prove $P(v, w)$ for particular values v in S and w in T. In our example, v is $n - 2$, w is $2 * m$, and S and T both equal Int. The WITNESS step we would generally use to prove G1 is

WITNESS $n - 2 \in Int,\ 2 * m \in Int$

I have chosen a different, rather silly example to explain how a WITNESS step works in general. Our example WITNESS step is equivalent to the step

SUFFICES ASSUME $n - 2 \in Nat,\ 2 * m \in 1 \,..\, n$
PROVE $(n - 2) + (2 * m) \leq 3 * (n + 1)$

with the proof

BY $1 \,..\, n \subseteq Int,\ 2 * m \in 1 \,..\, n,\ Nat \subseteq Int,\ n - 2 \in Nat$

Thus, the step changes **G** to $(n - 2) + (2 * m) \leq 3 * (n + 1)$. If the step is named, it adds formulas $n - 2 \in Nat$ and $2 * m \in 1 \,..\, n$ to $\mathcal{F}_K$; if it is unnamed, it adds these formulas to $\mathcal{F}_K$ and $\mathcal{F}_U$.

There is also an unbounded form (without the $\in$) of the WITNESS statement:

WITNESS $n - 2,\ 2 * m$

that can be used if the goal is of the form $\exists\, a, b \,:\, P(a, b)$. It changes **G** the same way as the corresponding bounded WITNESS, but leaves the other state components unchanged. It generates no proof obligations.

A WITNESS statement helps the backend provers by explicitly telling them how to prove an existentially quantified formula. They seldom need this help. The provers will usually deduce G0 by themselves from the facts $v \in S$, $w \in T$, and $P(v, w)$.

The HAVE and TAKE steps that are described next were added to the language to save some typing. I never use them, preferring the equivalent SUFFICES steps. Readers encountering TLA$^+$ proofs for the first time can find them forbidding. For such readers, it's a good idea to use as few different kinds of steps as you can. If you use HAVE and TAKE steps, it's best to do so only in the lowest-levels of the proof.

HAVE F

where F is an arbitrary formula. The current goal **G** must be of the form $P \Rightarrow Q$, in which case the step is equivalent to

SUFFICES ASSUME F
PROVE Q

with a leaf proof OBVIOUS. To check this leaf proof, TLAPS has to prove $P \Rightarrow F$. This statement is most often used with F equal to P.

TAKE $i, j \in U,\ k \in V$

For this statement to be legal, the current goal **G** must be equivalent to

$$\forall\, a, b \in S,\ c \in T \,:\, P(a, b, c)$$

In this case, the step is equivalent to the step

SUFFICES ASSUME NEW $i \in U$, NEW $j \in U$, NEW $k \in V$
PROVE $P(i, j, k)$

with the leaf proof

BY $T \subseteq V$, $S \subseteq U$

The TAKE step is almost always used with $U = S$ and $V = T$. In this case, the SUFFICES step can be generated with the Toolbox's Decompose proof command by selecting the Use suffices option.

There is also an analogous unbounded version:

TAKE i, j, k

It is equivalent to

SUFFICES ASSUME NEW i, NEW j, NEW k
PROVE $P(i, j, k)$

with proof OBVIOUS.

An INSTANCE Statement

The INSTANCE step has the same syntax as a module level INSTANCE step (not preceded by $\triangleq$). It leaves the current goal **G** unchanged and changes the other components of the proof state the same way that an ordinary INSTANCE statement in the module does. TLAPS does not (yet) handle INSTANCE steps.

12.4 Proof Obligations

Proof obligations are generated by leaf proofs and by the following kinds of steps that do not take proofs: USE, WITNESS, TAKE, and HAVE. These four kinds of steps are explained above in terms of equivalent steps with OBVIOUS or BY leaf proofs. The proof obligations generated by the steps are the ones generated by those leaf proofs. We therefore need to consider only the proof obligations generated by an OBVIOUS or BY leaf proof.

An OBVIOUS proof generates a single proof obligation. Suppose the proof context at the proof has these components:

$\mathcal{S}$: $\langle$VARIABLE x, CONSTANT $i \in S\rangle$
$\mathcal{F}_U$: $\{v < 0,\ 2 * y = 14\}$
$\mathcal{D}_U$: $\{S,\ w,\ y\}$
G: $i + 3 > v + w$
$\mathcal{B}$: $\langle Zenon,\ SMT\rangle$

where S, y, and w are defined by:

$S \triangleq Nat$
$y \triangleq i - 1$
$w \triangleq y + 2$

The proof OBVIOUS then generates this proof obligation:

ASSUME VARIABLE x,
CONSTANT $i \in Nat$
$v < 0$,
$2 * (i - 1) = 14$
PROVE $i + 3 > v + ((i - 1) + 2)$

Note how all occurrences of S, y, and w have been replaced by their definitions. The obligation is sent to *Zenon* and, if *Zenon* fails to prove it, it is sent to *SMT*.

To describe the obligations generated by a BY proof, we consider this proof step:

$\langle 3\rangle 8$. ASSUME NEW $i \in Nat,\ P(i)$
PROVE $Q(i)$
BY $\langle 2\rangle 5,\ F > 1,\ Isa,\ 2 \oplus 3 = 5,\ \langle 3\rangle 8,\ SMT,\ G(42),\ Zenon$ DEF $F,\ \oplus,\ Q$

where step $\langle 2\rangle 5$ is

$\langle 2\rangle 5$. *Step*2_5

for some formula *Step*2_5. Step $\langle 3\rangle 8$ and its proof are then equivalent to the following steps. Note how each BY fact that isn't a name of a previously asserted fact must be proved using the preceding BY facts; and the definitions of symbols in the DEF clause are expanded in all these proofs.

$\langle 3\rangle 8$. ASSUME NEW $i \in Nat$, $P(i)$
PROVE $Q(i)$
$\langle 4\rangle$ USE DEF F, $\oplus$, Q
$\langle 4\rangle 1$. ASSUME NEW CONSTANT $i \in Nat$, *Step*2_5
PROVE $F > 1$
OBVIOUS
$\langle 4\rangle$ USE *Isa*
$\langle 4\rangle 2$. ASSUME NEW CONSTANT $i \in Nat$, *Step*2_5, $F > 1$
PROVE $2 \oplus 3 = 5$
OBVIOUS
$\langle 4\rangle$ USE *Isa*, *SMT*
$\langle 4\rangle 3$. ASSUME NEW CONSTANT $i \in Nat$, *Step*2_5, $F > 1$, $2 \oplus 3 = 5$, $P(i)$
PROVE $G(42)$
OBVIOUS
$\langle 4\rangle$ USE *Isa*, *SMT*, *Zenon*
$\langle 4\rangle 4$. ASSUME NEW CONSTANT $i \in Nat$, *Step*2_5, $F > 1$, $2 \oplus 3 = 5$, $P(i)$, $G(42)$
PROVE $Q(i)$
OBVIOUS
$\langle 4\rangle 4$. QED
$\langle 3\rangle$ USE *Isa*, *SMT*, *Zenon*

The USE DEF step is omitted if the BY proof has no DEF clause. The other USE steps are omitted if the BY facts do not specify any backend prover. In that case, the value of $\mathcal{B}$ at step $\langle 3\rangle 8$ determines the backend provers used to check the proofs.

12.5 Further Details

Here are some miscellaneous facts about proofs and the proof language.

12.5.1 Additional Language Features

@ Expressions

Suppose you want to prove an equality $a > d$ by proving $a \geq b$, $b = c$, and

$c > d$, where a, b, c, and d may be large expressions. To save some typing, you can write:

$\langle 3 \rangle 6.$ $a \geq b$
$\langle 3 \rangle 7.$ @ $= c$
$\langle 3 \rangle 8.$ @ $> d$

In this case, the @ in step $\langle 3 \rangle 7$ is an abbreviation for b, and the @ in step $\langle 3 \rangle 8$ is an abbreviation for c. The symbol @ does not mean b in a proof of $\langle 3 \rangle 7$, nor does it mean c in a proof of $\langle 3 \rangle 8$. The symbol @ can be used in the same way in subproofs of those steps' proofs.

Using Unnamed Facts

The description of TLAPS given thus far makes it seem impossible to use an unnamed fact such as

ASSUME $N + 1 > M$

in a proof. There is one additional feature of TLAPS that makes it possible to use such a fact: TLAPS will accept as proved a formula F appearing in a BY clause or USE statement if F is identical to a fact in $\mathcal{F}_K$. This allows the use of unnamed facts in a proof. For example, the statement adds to $\mathcal{F}_K$ the fact $N + 1 > M$. Thus the assumption above can be used in a proof as follows:

$\langle 3 \rangle 7a.$ $2 * N + 2 > 2 * M$
BY $N + 1 > M$

The proof BY $1 + N > M$ will fail, with TLAPS complaining that it can't prove $1 + N > M$ (assuming that this formula is not implied by facts in $\mathcal{F}_U$). To use a formula F this way, F must have the same parse tree as a formula in $\mathcal{F}_K$.

Implicit Level Specifiers

You can write a proof consisting entirely of unnamed steps without writing explicit level numbers. Just write $\langle + \rangle$ (typed `<+>`) as the level specifier of the first step and $\langle * \rangle$ (typed `<*>`) as the level specifiers of subsequent steps. For example, here is a possible subproof structure.

$\langle 4 \rangle 2.$ ...
$\langle + \rangle$ ...
$\langle * \rangle$ ...
$\langle * \rangle$ ...
$\langle + \rangle$ ...
$\langle * \rangle$ ...
$\langle * \rangle$ QED
$\langle * \rangle$ ...
$\langle * \rangle$ QED

You can write PROOF $\langle * \rangle$ or PROOF $\langle + \rangle$ instead of $\langle + \rangle$.

Each $\langle + \rangle$ or $\langle * \rangle$ is equivalent to $\langle i \rangle$ for a suitable level number i. You can even mix steps labeled $\langle * \rangle$ and steps labeled $\langle i \rangle$ or $\langle i \rangle j$ in the same proof. (A little experimentation will reveal what the appropriate level i is.) However, this is a bad idea. You should use $\langle + \rangle$ and $\langle * \rangle$ only in short, low-level subproofs that need no step names.

Subexpression Names

When writing proofs, it is often necessary to refer to subexpressions of a formula. In theory, one could use definitions to name all these subexpressions. For example, if

$$Foo(y) \triangleq (x + y) + z$$

and we need to mention the subexpression $(x + 13)$ of $Foo(13)$, we could write

$$Newname(y) \triangleq (x + y)$$
$$Foo(y) \triangleq NewName(y) + z$$

This doesn't work in practice because it results in a mass of non-locally defined names, and because we may not know when we define the formula which subformulas will need to be mentioned later.

TLA$^+$ provides a method of naming subexpressions of a definition. If F is defined by $F(a, b) \triangleq \ldots$, then any subexpression of the formula obtained by substituting expressions A for a and B for b in the right-hand side of this definition has a name beginning "$F(A, B)$!". (Although this is a new use of the symbol "!", it is a natural extension of its use with module instantiation.) Here is a complete explanation of subexpression names.

You can use subexpression names in any expression. When writing a specification, you can define operators in terms of subexpressions of the definitions of other operators. Don't! Subexpression names should be used only in proofs. In a specification, you should use definitions to give names to the subexpressions that you want to re-use in this way.

12.5.2 Importing

Instantiated Theorems

The statement

$$I \triangleq \text{INSTANCE } M \text{ WITH } \ldots$$

imports definitions and theorems into the current module. If module M defines

$$D \triangleq \psi$$

for some expression ψ, then the INSTANCE statement defines $I\,!\,D$ in the current module to equal $\overline{\psi}$, which is the formula obtained from ψ by performing the substitutions for the CONSTANT and VARIABLE parameters of M specified by the WITH clause. Suppose module M contains the theorem

THEOREM $Thm \triangleq$ ASSUME A
PROVE Γ

and that this theorem is preceded in module M by the two assumptions

ASSUME B and ASSUME C

The INSTANCE statement then imports the theorem $I\,!\,Thm$, which asserts

ASSUME $\overline{B}$, $\overline{C}$, $\overline{A}$
PROVE $\overline{\Gamma}$

This is the case even if there are additional assumptions following theorem *Thm* in module M.

Everything works the same if the "$I \triangleq$" is removed from the INSTANCE statement, except that definitions and theorem names are imported from M without renaming.

Special Modules

There are certain special modules whose defined operators are treated as if they were built-in operators. That is, knowledge about the meanings of those operators are built into the backend provers. Putting those operators in the DEF clause of a USE or HIDE statement has no effect. Those special modules are

Naturals *Integers* *Sequences* *TLAPS* *TLC*

Although all the operators defined in the TLC module are treated by TLAPS like built-in operators, the backend provers have useful knowledge only about $:>$ and @@.

Local Definitions

You probably did not realize that TLA$^+$ has LOCAL definitions, and it's unlikely that you will ever have any reason to use them. But if you do, here's what you need to know if your proofs use facts or definitions imported from modules containing local definitions. Suppose module M contains

LOCAL $L \triangleq 22 * i$
$G \triangleq L + 14$

In a module that imports M, the definition of G can be expanded in a proof by

USE DEF G

(If the module is instantiated with renaming, G is replaced with something like $I!G$.) However, the definition of L can't be expanded because L cannot be referenced in the importing module. Currently, the definition of L is automatically expanded if module M is imported with the EXTENDS statement. It is left unexpanded if M is imported with instantiation. This may change.

12.5.3 Recursively Defined Functions and Operators

A recursive function definition is treated as if it were the equivalent non-recursive definition in terms of CHOOSE. For example

$$fact[n \in Nat] \triangleq \text{IF } n = 0 \text{ THEN } 1 \text{ ELSE } n * fact[n-1]$$

is treated as if it were

$$fact \triangleq \text{CHOOSE } f : f = [n \in Nat \mapsto \text{IF } n = 0 \text{ THEN } 1 \text{ ELSE } n * f[n-1]]$$

The library modules *NaturalsInduction* and *WellFoundedInduction* provide useful theorems for reasoning about recursively defined functions.

Recursive operator definitions are more problematic. The statements

$$\text{RECURSIVE } Fact(\_)$$
$$Fact(n) \triangleq \text{IF } n = 0 \text{ THEN } 1 \text{ ELSE } n * Fact(n-1)$$

are equivalent to a definition

$$Fact(n) \triangleq \ldots$$

whose right-hand side is very complicated and approximately incomprehensible. We hope eventually to provide library modules that make it possible to prove things about recursively defined operators. For now, operators that are declared in a RECURSIVE statement are treated by TLAPS like declared operators rather than defined operators. Their definitions cannot be expanded, and there is no way to prove anything about them from their definitions. If you must use a recursively defined operator like *Fact* now, you should assume without proof a theorem like:

$$\text{LEMMA } FactDef \triangleq \forall n \in Nat : Fact(n) = (\text{IF } n = 0 \text{ THEN } 1 \text{ ELSE } n * Fact(n-1))$$

You should use TLC to check the theorem. When TLAPS handles recursively defined operators, you should be able to prove it.

12.5.4 The Fine Print

Some of the explanations of proofs and the proof language given above were not completely accurate. Here is what was omitted.

- $\mathcal{F}_K$ and $\mathcal{F}_U$ are not really sets of formulas; they are actually sets of formulas and names of steps and theorems. For example, the sequence of steps:

  $\langle 2 \rangle \quad x = 2$
  $\langle 2 \rangle 1. \; x = 2$
  $\langle 2 \rangle \quad \text{USE } \langle 2 \rangle 1$

  adds the formula $x = 2$ and the name $\langle 2 \rangle 1$ to $\mathcal{F}_U$. The step

  $\langle 2 \rangle \text{HIDE } \langle 2 \rangle 1$

  removes the name $\langle 2 \rangle 1$ from $\mathcal{F}_U$, but not the formula $x = 2$.

  The step HIDE $\langle 2 \rangle 1$ removes that step name from $\mathcal{F}_U$. It does not remove the name of a step that names the same formula as $\langle 2 \rangle 1$, nor the fact named by $\langle 2 \rangle 1$ if that formula is also in $\mathcal{F}_U$.

- Examining TLAPS's console output reveals that, in addition to the proof obligations described above, there are some trivial obligations that TLAPS proves easily.

13 The Bounded Buffer Proof

In Section 8.3.2, we showed informally that the bounded buffer algorithm of module *PCalBoundedBuffer* implements the bounded channel specification of module *PCalBoundedChannel* under a refinement mapping. We now examine the TLA$^+$ proof of that result.

First, download the file `PCalBoundedBuffer.tla` that contains the specification's root module with the proof, as well as the file `PCalBoundedChannel.tla` that contains an instantiated module. I suggest that you put these files in a different folder (directory) than the files by that name that you have already created, but you can overwrite the existing files if you wish. (If you want to overwrite the existing files, make sure that the Toolbox does not have a specification open that contains either of those files.) Open the specification in the Toolbox.

Click here if you have trouble downloading these files.

Definitions and Assumptions

This version of module *PCalBoundedBuffer* is the same, up through the statement of the algorithm, as the one you saw in Section 8.2.2. However, I have given the assumption on line 15 the name *NAssump* so it can be used in the proof.

The definitions of *PCInv*, *TypeOK*, and *Inv* (lines 79–85) are the same as before, except that I defined *TypeOK* in terms of the new state function *BufCtr* that is defined on line 77. When defining formulas to be used in a TLAPS proof, we use auxiliary definitions to break them into smaller pieces than we would for use in a specification or a hand proof. We do this are two reasons:

How to go to line 79.

- TLAPS proofs are generally pretty long, and using auxiliary definitions to abbreviate subexpressions can save a fair amount of typing.
- Replacing the subexpression $0\,..\,(2 * N - 1)$ by the name *BufCtr* helps the theorem provers, since they then don't have to consider facts about multiplication, subtraction, 0, 1, 2, or the operator "$..$" when searching for a proof.

The algorithm uses the modulus operator $\%$. To prove properties of the algorithm, we need to reason about this operator. The backend provers do not yet know anything about $\%$, so we need to assert some facts about it. The most elegant way to do this would be to assert the defining properties of $\%$ and $\div$, explained in Section 13.1. This would lead us to the following axiom:

$$\begin{aligned} \forall n \in \mathit{Int}, d \in \mathit{Nat} \setminus \{0\} : & \land\ n \,\%\, d \in 0\,..\,(d-1) \\ & \land\ n \div d \in \mathit{Int} \\ & \land\ n = d * (n \div d) + (n \,\%\, d) \end{aligned}$$

However, I decided that it would be easier to assume Lemma *ModDef* (line 93). This lemma should be strong enough to prove the axiom above from properties of integers if $n \div d$ were defined to equal

$$\text{CHOOSE } q \in Int : n - (n \,\%\, d) = d * q$$

so it should be the only assumption about % that we need. However, I found that I also needed Lemma *ModMod* (line 100). Although *ModMod* can be proved from *ModDef* using only the properties of integers, the proof would be rather tiresome. Therefore, I decided to assume *ModMod* as well.

As I explained when we introduced *GCD*1–*GCD*3 in the proof of Euclid's algorithm, we should use TLC to check anything that we assume without proof. We can check Lemmas *ModDef* and *ModMod* the same way we checked *GCD*1–*GCD*3: by substituting finite sets of numbers for *Int* and $Nat \setminus \{\}$.

Lemmas *ModDef* and *ModMod* should be the only facts that we need to assume to prove that *Inv* is an invariant of the algorithm. However, a bug in an earlier backend prover that was used for reasoning about arithmetic prevented it from proving that $2 * N$ is an integer. I got around that bug by assuming that $2 * N$ equals $N + N$. Because that prover had trouble handling $2 * N$, it seemed best to hide $2 * N$ from it by defining K to equal $2 * N$ and to use the definition of K only to prove some simple properties of it. The expression $2 * N$ therefore appears in almost no proof obligations. The definition of K is on line 113, and the assertion that it equals $N + N$ appears as the lemma *KDef* on line 114, which the *SMT* prover checks easily.

Navigating the Module

Before getting into the actual proofs, let's examine the commands provided by the Toolbox for navigating through specifications and proofs. None of the following navigation commands work if the Toolbox reports a parsing error in the specification.

Finding a Definition or Declaration

The Goto Declaration command jumps to the definition or declaration of a symbol. You select the symbol by putting the cursor anywhere in or just to the left of an occurrence of it or by selecting any portion of the occurrence.

Select the symbol *Msg* on line 81 and execute the Goto Declaration command—either by right-clicking anywhere and choosing it from the menu or by hitting the F3 key. This will jump to and highlight the symbol *Msg* in the CONSTANT declaration. Jump back to the *Msg* on line 81 by executing the Return from Goto Declaration command—either from the menu raised by right-clicking or by hitting the F4 key.

Another way to jump to a symbol's definition or declaration is to hold down the control key and move the mouse pointer over the symbol. The symbol should

become marked as a link—perhaps by changing color and being underlined. Clicking on the symbol then jumps to its declaration or definition.

The Show Declarations command produces a pop-up menu with a list of all symbols that have a meaning in the module, including ones imported from other modules. However, the list does not include any symbols defined in a standard module or any bound identifiers. The command can be executed either from the menu raised by right-clicking or by hitting the F5 key. You can jump to a symbol's definition or declaration by clicking on its name in the list. You can also select the symbol by moving up and down the list with the arrow keys; hitting the Enter key jumps to the definition or declaration of the highlighted symbol. You can also pare the list of symbols displayed by typing into the menu. Only symbols that begin with the letters you have typed (ignoring case) will be shown.

The Return from Goto Declaration command returns the cursor to where it was when you last executed a Goto Declaration or Show Declarations command.

The Show Uses command highlights all uses of a symbol and jumps to the first use. To execute it, select an occurrence of the symbol and execute the command either by right-clicking and selecting it from the menu or by pressing the F6 key. The Goto Next Use (F8) and Goto Prev Use (F7) commands let you cycle through all the uses of the symbol. Remember that you can return to the symbol's definition or declaration with the Goto Declaration command.

Experiment with these commands. Execute the Show Declarations command and choose $C!Init$. This will take you to the definition of *Init* in the instantiated module *PCalBoundedChannel*. See what happens when you execute Show Uses for *Init*.

Return to module *PCalBoundedBuffer*. Observe that the Show Uses command also works for bound identifiers such as the symbols i and j in Lemma *ModDef*. (If you've forgotten where that lemma is, use the Show Declarations command to find it.)

Find the uses of Lemma *ModDef*. Cycle up (F7) to the next to last use, which is on line 1003.

Observe that a step name such as $\langle 2 \rangle 1$ is considered to be a (non-global) defined symbol, so the Goto Declaration and Show Uses commands work for it.

Go to the comment that begins on line 468 and observe that you can jump to the definition of a symbol even from an occurrence inside a comment.

Check that you can't go to the definition of any symbol defined in a standard module. However, you can show the uses of such a symbol. You can show neither definitions nor uses of built-in TLA$^+$ operators like $\Rightarrow$.

These navigation commands may do strange things if the module has been modified since it was last saved and parsed. They usually act as if a symbol's occurrences are at the location in the file where they were when the module was last parsed. Most of the commands do nothing if the Toolbox reports a parsing error in the specification.

The rest of the description of the bounded buffer proof has not yet been written. However, comments in the TLA$^+$ files explain the proof. You should try reading those files.