10 The Proof Language

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

10.1 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

$$\text{THEOREM } id \triangleq A$$

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

$$\begin{array}{ll}\text{ASSUME} & A_1, \ldots, A_n \\ \text{PROVE} & B\end{array}$$

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 $v \in S$ has the same meaning as the declaration NEW v followed by the assumption $v \in S$. The assumption $v \in S$ implied by the NEW declaration is called a *domain assumption*. (The same formula $v \in S$ appearing as an assumption, where v has already been declared or defined, is not a domain assumption.)

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.)

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

$$\text{ASSUME PROVE } F$$

asserts the formula F. The assertion

$$\begin{array}{ll}\text{ASSUME} & \text{NEW } x \in S,\ P(x) \\ \text{PROVE} & Q \lor R(x)\end{array}$$

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

$$\begin{array}{ll}\text{ASSUME} & \text{NEW } P(\_),\ \text{NEW } x,\ \text{NEW } y,\ x = y \\ \text{PROVE} & P(x) = P(y)\end{array}$$

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. However, I will use such formulas for the purpose of explaining proofs.

10.2 The Hierarchical Structure of a Proof

10.2.1 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.

10.2.2 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.

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.

10.3 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 sequeunce 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 whose definitions are by default expanded in proofs.

$\mathcal{D}_U$ 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

$\langle$VARIABLE x, CONSTANT AB, CONSTANT C, 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.

10.3.1 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:

?
←
→
C
I
S

- 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.
- $\mathbf{G}$, $\mathcal{S}$, 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 $\mathbf{G}$ 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 both 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

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

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

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

to the equivalent proof

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

$\boxed{\text{Level-4 proof of } \mathbf{G}\text{, using } \forall\, i \in S : (j \in T) \Rightarrow Q(i)\,.}$

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

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

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:

$$\text{PICK } i, j \in S,\ k \in T : P(i, j, k)$$
$$\text{PICK } i,\ j,\ k : (i \notin j) \land Q(j, k)$$

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

10.3.2 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 $\langle Isa, SMT\rangle$. (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 $f(a) \triangleq a+1$
$\quad g \triangleq f(42) * b$

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 be212418458 Mon Feb 10 10:57:14 2014 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 $R \Rightarrow F$. 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 ∈ U, NEW j ∈ U, NEW k ∈ 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.

10.4 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:

$$\begin{array}{ll}
\mathcal{S}: & \langle \text{VARIABLE } x,\ \text{CONSTANT } i \in S \rangle \\
\mathcal{F}_U: & \{v < 0,\ 2 * y = 14\} \\
\mathcal{D}_U: & \{S,\ w,\ y\} \\
\mathbf{G}: & i + 3 > v + w \\
\text{BB}: & \langle \mathit{Zenon},\ \mathit{SMT} \rangle
\end{array}$$

where S, y, and w are defined by:

$$\begin{array}{lcl}
S & \triangleq & \mathit{Nat} \\
y & \triangleq & i - 1 \\
w & \triangleq & y + 2
\end{array}$$

The proof OBVIOUS then generates this proof obligation:

$$\begin{array}{ll}
\text{ASSUME} & \text{VARIABLE } x, \\
 & \text{CONSTANT } i \in \mathit{Nat} \\
 & v < 0, \\
 & 2 * (i - 1) = 14 \\
\text{PROVE} & i + 3 > v + ((i - 1) + 2)
\end{array}$$

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,

$$\begin{array}{l}
\langle 3 \rangle 8.\ \ \text{ASSUME NEW } i \in \mathit{Nat},\ P(i) \\
\qquad\ \ \text{PROVE } Q(i) \\
\quad \text{BY } \langle 2 \rangle 5,\ F > 1,\ \mathit{Isa},\ 2 \oplus 3 = 5,\ \langle 3 \rangle 8,\ \mathit{SMT},\ G(42),\ \mathit{Zenon}\ \ \text{DEF } F,\ \oplus,\ Q
\end{array}$$

where step $\langle 2\rangle 5$ is

$$\langle 2\rangle 5.\quad \mathit{Step2\_5}$$

for some formula $\mathit{Step2\_5}$. 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.

```
⟨3⟩8. ASSUME  NEW $i \in Nat$, $P(i)$
      PROVE   $Q(i)$
  ⟨4⟩     USE DEF $F$, $\oplus$, $Q$
  ⟨4⟩1. ASSUME  NEW CONSTANT $i \in Nat$, $Step2\_5$
        PROVE   $F > 1$
    OBVIOUS
  ⟨4⟩     USE $Isa$
  ⟨4⟩2. ASSUME  NEW CONSTANT $i \in Nat$, $Step2\_5$, $F > 1$
        PROVE   $2 \oplus 3 = 5$
    OBVIOUS
  ⟨4⟩     USE $Isa$, $SMT$
  ⟨4⟩3. ASSUME  NEW CONSTANT $i \in Nat$, $Step2\_5$, $F > 1$, $2 \oplus 3 = 5$, $P(i)$
        PROVE   $G(42)$
    OBVIOUS
  ⟨4⟩     USE $Isa$, $SMT$, $Zenon$
  ⟨4⟩4. ASSUME  NEW CONSTANT $i \in Nat$, $Step2\_5$, $F > 1$, $2 \oplus 3 = 5$, $P(i)$, $G(42)$
        PROVE   $Q(i)$
    OBVIOUS
  ⟨4⟩4. QED
⟨3⟩     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.

10.5 Further Details

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

10.5.1 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:

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

In this case, the @ in step $\langle 3 \rangle 7$ is an abbeviation for b, and the @ in step $\langle 3 \rangle 8$ is an abbeviation 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.

Implicit Level Specifiers

You can avoid writing explicit level numbers in a level specifier by using the level specifiers $\langle * \rangle$ and $\langle + \rangle$, whose meanings are defined as follows.

In a proof with level number i, the level specifier $\langle * \rangle$ is euivalent to $\langle i \rangle$. A proof has level number i if either (i) it begins with a step having the level specifier $\langle i \rangle$; or (ii) it begins with PROOF $\langle * \rangle$ and is the proof of a level $i - 1$ step; or (iii) $i = 0$, it is the top-level proof of a theorem, and it begins with either $\langle * \rangle$ or PROOF $\langle * \rangle$. A proof beginning with either $\langle + \rangle$ or PROOF $\langle + \rangle$ has the same level as if it began with PROOF $\langle * \rangle$.

As an example, here are two equivalent ways to number proof steps.

```
THEOREM ...                    THEOREM ...
  ⟨∗⟩1. ...                      ⟨0⟩1. ...
    ⟨3⟩2. ...                      ⟨3⟩2. ...
    ⟨∗⟩3. QED                      ⟨3⟩3. QED
  ⟨∗⟩4. ...                      ⟨0⟩4. ...
    ⟨+⟩5. ...                      ⟨1⟩5. ...
    ⟨∗⟩ 6. QED                     ⟨1⟩6. QED
      PROOF                          PROOF
        ⟨∗⟩7. ...                      ⟨2⟩7. ...
        ⟨∗⟩8. QED                      ⟨2⟩8. QED
    ⟨∗⟩9. QED                      ⟨1⟩9. QED
  ⟨∗⟩10. QED                     ⟨0⟩10. QED
```

You can use $\langle * \rangle$ and $\langle + \rangle$ in exactly the same way in preface tokens of unnamed proof steps. You can also use a step name like $\langle * \rangle 7$ as a fact in a BY proof; it is then equivalent to $\langle i \rangle 7$ where i is the level number of the step being proved by the BY proof.

TLAPS currently does not accept step names like $\langle * \rangle 7$ in a BY proof.

The only reason to use $\langle * \rangle$ and $\langle * \rangle$ is because it makes it easier to reorganize a proof, since you can move a step with preface token $\langle * \rangle 3$ from one proof to another one without having to rename it because the proofs are at a different level. You can

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

$$\begin{aligned} Newname(y) &\;\triangleq\; (x+y) \\ Foo(y) &\;\triangleq\; NewName(y)+z \end{aligned}$$

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

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.

10.5.2 Importing

Instantiated Theorems

The statement

$$I \;\triangleq\; \textsc{instance}\ M\ \ \textsc{with}\ \ldots$$

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 $\overline{exp}$, 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

$$\begin{aligned} \textsc{theorem}\ \ Thm \;\triangleq\; &\textsc{assume}\ \ A \\ &\textsc{prove}\ \ \ \Gamma \end{aligned}$$

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

$$\textsc{assume}\ B \qquad \text{and} \qquad \textsc{assume}\ C$$

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

$$\begin{array}{ll} \textsc{assume} & \overline{B},\ \overline{C},\ \overline{A} \\ \textsc{prove} & \overline{\Gamma} \end{array}$$

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

$$\mathit{Naturals} \qquad \mathit{Integers} \qquad \mathit{Sequences} \qquad \mathit{TLAPS} \qquad \mathit{TLC}$$

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

$$\textsc{local}\ L \triangleq 22 * i G \triangleq L + 14$$

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

$$\textsc{use}\ \ \textsc{def}\ G$$

(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.

10.5.3 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

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

is treated as if it were

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

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

Recursive operator definitions are more problematic. The statements

$$\begin{array}{l} \text{RECURSIVE } \mathit{Fact}(\_) \\ \mathit{Fact}(n) \;\triangleq\; \text{IF } n = 0 \text{ THEN } 1 \text{ ELSE } n * \mathit{Fact}(n-1) \end{array}$$

are equivalent to a definition

$$\mathit{Fact}(n) \;\triangleq\; \ldots$$

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:

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

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

10.5.4 The Fine Print

Some of the explanations of proofs and the proof language given above were not completely accurate. Here is the full story.

- $\mathcal{F}_K$ 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:

$$\begin{array}{ll} \langle 2 \rangle & x = 2 \\ \langle 2 \rangle 1. & x = 2 \\ \langle 2 \rangle & \text{USE } \langle 2 \rangle 1 \end{array}$$

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

$\langle 2\rangle$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

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