TLA^{+2}

A Preliminary Guide

Leslie Lamport

15 January 2014

Contents

1 Introduction

TLA$^{+\mathbf{2}}$ is the current version of TLA^{+}. It is the version supported by the latest versions of the TLA^{+} tools. TLA^{+} now means TLA$^{+\mathbf{2}}$. The previous version of TLA^{+}, which is described in the TLA^{+} book (*Specifying Systems*) is here called TLA$^{+\mathbf{1}}$. This document explains the differences between TLA$^{+\mathbf{2}}$ and TLA$^{+\mathbf{1}}$.

Most of the additions to the language in TLA$^{+\mathbf{2}}$ are for writing proofs that can be checked with TLAPS, the TLA^{+} proof system. The major change that affects specifications is that you can now write recursive operator definitions, as described in Section 2. Another change is the introduction of *lambda* expressions, explained in Section 3.

Almost all legal TLA$^{+\mathbf{1}}$ specifications are legal TLA$^{+\mathbf{2}}$ specifications. Two rather arcane changes have been made to instantiation; they are explained in Section 5. The only other change that affects TLA$^{+\mathbf{1}}$ specifications is that the following new keywords have been added in TLA$^{+\mathbf{2}}$, and thus cannot be used as identifiers.

ACTION	HAVE	PICK	SUFFICES
ASSUMPTION	HIDE	PROOF	TAKE
AXIOM	LAMBDA	PROPOSITION	TEMPORAL
BY	LEMMA	PROVE	USE
COROLLARY	NEW	QED	WITNESS
DEF	OBVIOUS	RECURSIVE	
DEFINE	OMITTED	STATE	
DEFS	ONLY		

2 Recursive Operator Definitions

The only recursive definitions allowed in TLA$^{+\mathbf{1}}$ were recursive function definitions. This restriction was inconvenient for the following reasons: (i) specifying the function's domain was sometimes difficult, (ii) checking that the function was applied to an element in the domain could significantly slow down TLC, and (iii) there was no provision for mutual recursion. I did not allow recursive operator definitions in TLA$^{+\mathbf{1}}$ because I didn't know how to assign a sensible meaning to them—for example, what should be the meaning of this silly definition?

$$F \;\triangleq\; \text{CHOOSE } v \,:\, v \neq F$$

Georges Gonthier and I have figured out how to define recursive operator definitions so they have the expected meaning when you expect them to be meaningful—namely, when the value can be computed by expanding the definition a finite number of times. The precise definition is complicated and will appear elsewhere.

In TLA^{+2}, the use of a defined operator must come after either its definition or its declaration by a *recursive* statement. For example,

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

defines $\mathit{fact}(n)$ to equal $n!$ if n is a natural number. I have no idea what it defines $\mathit{fact}(-2)$ or $\mathit{fact}(\text{“abc”})$ to equal. (Without the *recursive* declaration, *fact* could be used only after its definition, so its use in the right-hand side of the definition would be illegal.)

The syntax of the *recursive* statement is the same as that of the *constant* statement, allowing multiple declarations separated by commas. The *recursive* statement can come anywhere before the first use of the operators it declares, so it's easy to write mutually recursive definitions. However, you should put a *recursive* statement as close as possible to the definitions of the operators it declares. A tool might treat as recursive any definitions that come between an operator's *recursive* declaration and its definition.

A *recursive* statement can be used in a *let* clause to permit recursive definitions local to the *let*. A symbol declared in a *recursive* statement must later be defined to be an operator taking the correct number of arguments. Thus, recursive instantiations are *not* allowed; you cannot write

$$\begin{array}{l} \text{RECURSIVE } \mathit{Ins}(\_) \\ \mathit{Ins}(n) \;\triangleq\; \text{INSTANCE } M \text{ WITH } \;\ldots \end{array}$$

TLA^{+1} has the nice property that operator definitions are like macros. If F is defined by

$$F(x) \;\triangleq\; \ldots$$

then $F(\mathit{exp})$ is simply the expression obtained from the right-hand side of the definition by replacing every instance of x with *exp*. In TLA^{+2}, this is not true for recursively-defined operators. We do not know if $\mathit{fact}(-2)$ equals

$$\text{IF } \; n = 0 \text{ THEN } \; 1 \text{ ELSE } \; n * \mathit{fact}(-3)$$

It can be proved that

$$\mathit{fact}(42) \;\triangleq\; \text{IF } \; n = 0 \text{ THEN } \; 1 \text{ ELSE } \; n * \mathit{fact}(41)$$

However, because *fact* is defined recursively, this must be proved. The method of proving it is fairly standard; I won't discuss it here.

3 Lambda Expressions

TLA$^+$ allows you to define higher-order operators—that is, ones that take operators as arguments, such as

$$F(Op(\_, \_)) \triangleq Op(1, 2)$$

The argument of F is an operator that takes two arguments. In TLA^{+1}, such an argument had to be the name of an operator. For example, we might define

$$Id(a,\ b) \triangleq a + 2 * b$$

and write $F(Id)$. TLA^{+2} allows you to use F without having to define an operator to use as its argument. Instead of defining Id in this way and writing $F(Id)$, you can write

$$F(\textsc{lambda}\ a, b : a + 2 * b)$$

The *lambda* expression is the operator that Id is defined to equal.

A *lambda* expression can also be used in an *instance* statement to instantiate an operator parameter. For example, with the definition of Id given above, the following two statements are equivalent.

$\textsc{instance}\ M\ \textsc{with}\ Op \leftarrow Id$
$\textsc{instance}\ M\ \textsc{with}\ Op \leftarrow \textsc{lambda}\ a, b : a + 2 * b$

Syntactically, a *lambda* expression consists of the keyword LAMBDA followed by a comma-separated list of identifiers, followed by ":", followed by an expression. A *lambda* expression can be used only as the argument of a higher-order operator or to the right of a "$\leftarrow$" in an *instance* statement.

4 Theorems and Assumptions

4.1 Naming

There is no need for theorem or assumption names in a specification, since the name would be equivalent to TRUE. However, theorem and assumption names are used in writing proofs. In TLA^{+2}, you can name a theorem or assumption by inserting an optional "*identifier* $\triangleq$ " right after THEOREM or ASSUME, as in

$$\textsc{theorem}\ Fermat \triangleq \neg\exists\, n \in Nat \setminus (0\,..\,2) : \ldots$$

This is equivalent to

$$Fermat \;\triangleq\; \neg \exists\, n \in Nat \setminus (0 \,..\, 2) \,:\, \ldots$$
$$\textsc{theorem}\ Fermat$$

A theorem cannot have parameters.

TLA^{+2} allows LEMMA and PROPOSITION as synonyms for THEOREM and ASSUMPTION as a synonym for ASSUME. TLA^{+2} also allows AXIOM as almost a synonym for ASSUME and ASSUMPTION; it differs only in that (in Toolbox releases later than Version 1.1.2) TLC does not check assumptions labeled AXIOM. This is useful when writing assumptions that TLC can't check, for use in a proof.

4.2 Assume/Prove

In TLA^{+2}, a theorem can assert either a formula or an *assume/prove*. A formula is a Boolean-valued expression. However, since TLA$^+$ is untyped, the silly statement "THEOREM 42" is a legal (but unprovable) theorem. (See Section 16.1.3 of *Specifying Systems*.)

An *assume/prove* asserts a proof rule. Here is how it is used to assert a well-known rule of elementary logic that we can prove $P \Rightarrow Q$ by assuming P and proving Q.

$$\begin{array}{llll}
\textsc{theorem}\ DeductionRule \;\triangleq & \textsc{assume} & \textsc{new}\ P,\ \ \textsc{new}\ Q, & \\
 & & \textsc{assume} & P \\
 & & \textsc{prove} & Q \\
 & \textsc{prove} & P \Rightarrow Q &
\end{array}$$

Logicians often use "$\vdash$" to express such a rule, writing this as $(P \vdash Q) \vdash (P \Rightarrow Q)$. In TLA$^+$, we need to declare identifiers like P and Q before they can be used. Here is a standard proof rule of predicate logic; it asserts that we can prove $\forall\, x \in S \,:\, P(x)$ by choosing a brand-new identifier x, assuming $x \in S$, and proving $P(x)$.

$$\begin{array}{llll}
\textsc{theorem}\ \textsc{assume} & \textsc{new}\ P(_),\ \ \textsc{new}\ S, & \\
 & \textsc{assume} & \textsc{new}\ x \in S \\
 & \textsc{prove} & P(x) \\
\textsc{prove} & \forall x \in S \,:\, P(x) &
\end{array}$$

The third assumption of this rule,

$$\textsc{assume new}\ x \in S\ \ \textsc{prove}\ P(x)$$

is an abbreviation for

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

Here are some proof rules of TLA. The first asserts that a primed constant equals itself.

$$\text{THEOREM } \mathit{Constancy} \;\triangleq\; \begin{array}{ll} \text{ASSUME} & \text{CONSTANT } C \\ \text{PROVE} & C' = C \end{array}$$

Here is the standard TLA rule for proving invariance.

$$\text{THEOREM } \begin{array}{ll} \text{ASSUME} & \text{STATE } \mathit{Inv}, \\ & \text{ACTION } A, \;\; \text{STATE } v, \\ & \mathit{Inv} \land [A]_v \;\Rightarrow\; \mathit{Inv}' \\ \text{PROVE} & \mathit{Inv} \land \Box[A]_v \;\Rightarrow\; \Box \mathit{Inv} \end{array}$$

These theorems assert a rule that is valid whenever expressions or operators of the specified (or lower) level are substituted for the declared identifiers. For example, Theorem *Constancy* implies $(2+N)' = (2+N)$ if N is declared to be a constant parameter of the module. See Section 17.2 of *Specifying Systems* for an explanation of levels. (The *action* level is called transition-level there.)

The declaration NEW is equivalent to CONSTANT. If all the expressions and identifiers that appear in a theorem have constant level, then the theorem is valid when expressions of any level are substituted for the declared identifiers.

You can also use a *variable* declaration in an *assume* to state that some identifier is a TLA$^+$ variable. To illustrate the difference between a *variable* and a *state* declaration, consider this valid TLA$^+$ rule.

$$\text{THEOREM } \begin{array}{ll} \text{ASSUME} & \text{VARIABLE } x, \;\; \text{VARIABLE } y \\ \text{PROVE} & \text{ENABLED } x' \neq y' \end{array}$$

The theorem would not be valid if "VARIABLE" were replaced by "STATE" because the resulting theorem would allow any state-level expressions to be substituted for x and y. Substituting the variable z for both x and y would then yield the conclusion ENABLED $z' \neq z'$, which is false.

You have probably inferred most of the grammar of *assume*/*prove* assertions:

- An *assume/prove* consists of the token ASSUME, followed by a comma-separated list of *assumptions*, followed by the token PROVE, followed by an expression.
- An *assumption* is an expression, a *declaration*, or an *assume/prove*.
- A *declaration* may be:
  - The same as a *constant* statement in the body of the module that declares a single constant parameter, except that the keyword CONSTANT may optionally be replaced by NEW, STATE, ACTION, or TEMPORAL.
  - The token NEW or CONSTANT, followed by an identifier, the token $\in$, and an expression.
  - The token VARIABLE followed by an identifier.

  An optional NEW token may precede any of these declarations except for one beginning with a NEW token. (The unnecessary "NEW" may help some people understand the meaning of the declaration.)

Indentation is not significant. (In TLA^{+2} as in TLA^{+1}, indentation matters only in bulleted lists of conjuncts and disjuncts.)

5 Instantiation

Two minor changes to instantiation have been made in TLA^{+2}: (i) there is a different syntax for instantiated in-, pre-, and postfix operators, and (ii) operator instantiation has been restricted to allow instantiation only with "Leibniz" operators, which are defined below.

5.1 Instantiating *fix Operators

If module M defines an infix operator such as &&, then in TLA^{+1} the statement

$Foo \triangleq$ INSTANCE M WITH ...

defines an infix operator *Foo*!&& that would be used in such strange expressions as

1 *Foo*!&& 2

TLA^{+2} eliminates this awkward syntax. Instead, the operator *Foo*!&& is a "normal" *nonfix* operator and not an infix one, so you write this expression as *Foo*!&&(1, 2). If this were a parameterized instantiation, so *Foo* took an argument, then you would write something like *Foo*(42)!&&(1, 2).

The analogous change has been made to postfix operators and the prefix operator unary "−", which must be written as "`-.`" after a "`!`".

For the sake of uniformity, TLA^{+2} permits any infix or postfix operator to be used as a nonfix operator. For example, `+(1,2)` is another way of writing `1+2`. (Prefix operators could always be written this way.) This alternate syntax does not apply to the left-hand side of a definition. For example, the only way to define the infix operator && is to write something like

$$a \mathbin{\&\&} b \;\triangleq\; \ldots$$

Because of a bug that is unlikely to be fixed, the current SANY parser does not accept this alternate syntax for the infix operator "−"; it accepts only `2-1` and not `-(2,1)`.

5.2 Leibniz Operators and Instantiation

Consider the following module.

```
---------------- MODULE M ----------------
CONSTANTS C, D, F(_)
THEOREM (C = D) => (F(C) = F(D))
==========================================
```

The perfectly reasonable theorem in this module is not valid in TLA^{+1} for the following reason. The semantics of TLA^{+} requires that any instantiation of a valid theorem be valid. Now consider

VARIABLES x, y
$Prime(p) \triangleq p'$
INSTANCE M WITH $C \leftarrow x$, $D \leftarrow y$, $F \leftarrow Prime$

This imports the theorem from module M as

THEOREM $(x = y) \Rightarrow (x' = y')$

which is not valid. (Equality of the values of x and y in the current state doesn't imply that they are equal in the next state.)

In TLA^{+2}, the theorem of module M is valid, which means that this INSTANCE M statement is illegal. It is illegal because TLA^{+2} allows instantiation of an operator parameter only by a Leibniz operator, and F is non-Leibniz. An operator F of a single argument is defined to be *Leibniz* iff $e = f$ implies $F(e) = F(f)$, for any expressions e and f. (Logicians generally use the term *substitutive* rather than *Leibniz*.) For an operator F that takes k arguments, F is *Leibniz* iff the value of $F(e_1, \ldots, e_k)$ remains unchanged if any of the expressions e_i is replaced by an equal expression. Constant parameters are assumed to be Leibniz, so one constant parameter can be instantiated by another.

In TLA$^+$, all built-in and definable constant operators are Leibniz. The only built-in TLA$^+$ operators that are not Leibniz are the action operators and the temporal operators, listed in Tables 3 and 4 of *Specifying Systems*. In a non-constant module, a constant parameter can be instantiated only by a constant operator. Thus, the restriction added in TLA^{+2} is automatically satisfied except when substituting non-constant operators in a constant module. However, a non-constant operator can be Leibniz—for example, the Leibniz operator G defined by

$$G(a) \;\triangleq\; x' = [x \text{ EXCEPT } ![a] = y']$$

For a defined operator to be non-Leibniz, one of its parameters must appear in the definition within an argument of a non-Leibniz operator like $'$ (prime).

6 Naming Subexpressions

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

$$\mathit{Foo}(y) \;\triangleq\; (x + y) + z$$

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

$$\begin{array}{l} \mathit{Newname}(y) \;\triangleq\; (x + y) \\ \mathit{Foo}(y) \;\triangleq\; \mathit{NewName}(y) + z \end{array}$$

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^{+2} 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.)

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.

6.1 Labels and Labeled Subexpression Names

Any subexpression of a definition can be labeled. The syntax of a labeled expression is

$$\mathit{label} :: \mathit{expression}$$

(The symbol "::" is typed "`::`".) The label applies to the largest possible expression that follows it. In other words, the end of the labeled expression is the same as the end of the expression that you would get by replacing the "*label* ::" with "$\forall x :$". However, the expression is illegal if removing the label would change the way the expression is parsed. For example,

$$a + \mathit{lab} :: b * c$$

is legal because it is parsed as $a + (\mathit{lab} :: (b * c))$, which is how it would be parsed if the label *lab* were not there. However,

$$a * \mathit{lab} :: b + c$$

is illegal because it would be parsed as $a * (\mathit{lab} :: (b + c))$ and removing the label causes the expression to be parsed as $(a * b) + c$.

Label parameters are required if labels occur within the scope of bound identifiers. Here is an example.

$$\begin{array}{l} F(a) \;\triangleq\; \forall b : l1(b) :: (a > 0) \Rightarrow \\ \qquad\qquad\qquad\qquad\qquad \land \ldots \\ \qquad\qquad\qquad\qquad\qquad \land\, l2 :: \exists c : \land \ldots \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \land\, \exists d : l3(c, d) :: a - b > c - d \end{array}$$

For this example, $F(A)!l1(B)!l2!l3(C, D)$ names the expression $A - B > C - D$. Note how the parameters of each label are the bound identifiers introduced between it and the next outer-most label. Those identifiers can appear in any order. For example, if the label $l3(c, d)$ were replaced by $l3(d, c)$, then $F(A)!l1(B)!l2!l3(C, D)$ would name the expression $A - B > D - C$.

In this example, a reference to the subexpression labeled by $l3(c, d)$ from outside the definition of F, must specify the values of all the bound identifiers a, b, c, and d. That's why labels must include the bound identifiers as parameters. Also observe that to name a labeled subexpression, we have to name all the labeled subexpressions within which it lies. We're not even allowed to eliminate the label $l2$, even though it is superfluous in this example.

Label names do not conflict with operator names. In this example, any one of the label names $l1$, $l2$, or $l3$ could be replaced by F. The rule for name conflict is the obvious one needed to guarantee that there's no ambiguity in a subexpression name (where we are not allowed to use the number of parameters to disambiguate). Thus, we cannot label the first conjunct of the $\exists\, c$ expression with $l3(c)$, but we could label it with $l1(c)$ or $l2(c)$.

For subexpressions of the definition of an infix, postfix, or prefix operator, we use the "nonfix" form. For example, a subexpression of the definition of && would have the form $\&\&(A, B)\,!\ldots$.

We can also name subexpressions of definitions in instantiated modules. For example, if we have

$$Ins(x) \triangleq \text{INSTANCE } M \text{ WITH } \ldots$$

and ν is the name of any subexpression of a definition in module M, then $Ins(exp)!\nu$ is the name of the subexpression of the instantiated definition obtained when exp is substituted for x.

We call a subexpression name having one of the forms described here a *labeled subexpression name*. We include in this category the trivial case in which there is no label name, only the name of a defined operator—possibly in an instantiated module. The precise definition is contained in the "fine print" below. You probably don't want to read it.

The Fine Print

Here is the general definition explained above with examples. We say that label $lab1$ is the *containing label* of $lab2$ iff (i) $lab2$ lies within the expression labeled by $lab1$ and (ii) if $lab2$ lies within the expression labeled by any other label, then $lab1$ also lies within that expression.

We use the notation that $f(e_1, \ldots, e_k)$ denotes f when $k = 0$. A label *lab* has the form $id(p_1, \ldots, p_k)$ where *id* and the p_i are identifiers, the p_i are all distinct, and $\{p_1, \ldots, p_k\}$ is the set of all bound identifiers p_i such that:

- Label *lab* lies within the scope of p_i.
- If *lab* has a containing label *labc*, then the expression that introduces p_i lies within the expression labeled by *labc*.

We call *id* the *name* of the label. Two labels that either have no containing label or have the same containing label must have different names.

A *simple labeled subexpression name* of a module M has the form $\mathit{prefix}\,!\,\mathit{labexp}_1\,!\ldots!\,\mathit{labexp}_n$, where *prefix* has the form $Op(e_1, \ldots, e_{k[0]})$, each labexp_i has the form $id_i(e_1, \ldots, e_{k[i]})$, *Op* and the id_i are identifiers, and the e_j are expressions. It must satisfy:

- The definition

  $$Op(p_1, \ldots, p_{k[0]}) \;\triangleq\; \ldots$$

  occurs at the top level (not inside a *let* or inner module) of M.
- id_1 must be the name of a label lab_1 in the definition of *Op* that has no containing label.
- If $i > 1$, then id_i must be the identifier of a label lab_i whose containing label is lab_{i-1}.
- $k[i]$ must equal the number of parameters in lab_i, for each $i > 0$.

This labeled subexpression name denotes the expression obtained from the expression labeled with lab_n by substituting for each parameter of *Op* and of each lab_i the corresponding argument of *prefix* and labexp_i, respectively.

A *labeled subexpression name* of a module M is either a simple labeled subexpression name of M or else has the form $Id(e_1, \ldots, e_k)\,!\,\lambda$ where there is a statement

$$Id(e_1, \ldots, e_k) \;\triangleq\; \text{INSTANCE } N \ldots$$

at the outermost level of M and λ is a labeled subexpression name of module N.

6.2 Positional Subexpression Names

Instead of using labels, we can name subexpressions of a definition by a sequence of *positional selectors* that indicate the position of the subexpression in the parse tree. Consider this example

$$\begin{array}{lcl} F(a) & \triangleq & \land \ldots \\ & & \land \ldots \\ & & \land\ Len(x[a]) > 0 \\ & & \land \ldots \end{array}$$

Here are how some of the subexpressions of this definition are named, where A is an arbitrary expression:

- $F(A)!3$ names $Len(x[A]) > 0$, the third conjunct of $F(A)$—that is, of the right-hand side of the definition with A substituted for a. We think of this conjunct list as the application of a conjunction operator that takes four arguments, the third being $Len(x[A]) > 0$.

- $F(A)!3!1$ names $Len(x[A])$, the first argument of $>$, the top-level operator of the expression $F(A)!3$

- $F(A)!3!1!1$ names $x[A]$, the first (and only) argument of the top-level operator of the expression $F(A)!3!1$.

- The naming of subexpressions of $x[A]$ is based on the realization that this expression represents the application of a function-application operator to the two arguments x and A. Thus, $F(A)!3!1!1!1$ names x and $F(A)!3!1!1!2$ names A

The positional selector "$!\langle$" is always synonymous with $!1$, and "$!\rangle$" is synonymous with $!2$ when selecting the second argument of an operator that takes two arguments. Thus, instead of $F(A)!3!1!1!2$, we could write $F(A)!3!\langle!\langle!\rangle$ or $F(A)!3!\langle!1!\rangle$ or $F(A)!3!1!\langle!2$ or $\ldots$. As usual, "$\langle$" is typed "`<<`" and "$\rangle$" is typed "`>>`".

The use of positional selectors to pick an argument of an operator is self-evident for most operators that do not introduce bound identifiers. Here are the cases that are not obvious.

- In $[f \text{ EXCEPT } ![a] = g, ![b].c = h]$ we select f with $!1$, g with $!2$, and h with $!3$. No other subexpressions of the EXCEPT construct can be named.

- $r.\mathit{fld}$ is an application of a record-field selector operator to the two arguments r and "fld", so $!1$ selects r. (You can also use $!2$ to select "fld", but there's no reason to name a simple string constant with a subexpression name.)

- In $[\mathit{fld}_1 \mapsto \mathit{val}_1, \ldots, \mathit{fld}_n \mapsto \mathit{val}_n]$ and $[\mathit{fld}_1 : \mathit{val}_1, \ldots, \mathit{fld}_n : \mathit{val}_n]$ the selector $!\,i$ names the subexpression val_i for $i \in 1 \,..\, n$. The field names fld_i cannot be selected. (There is no point naming fld_i, since it's just a string constant.)

- In $\text{IF } p \text{ THEN } e \text{ ELSE } f$ the selector $!1$ names p, the selector $!2$ names e, and the selector $!3$ names f.

- In $\text{CASE } p_1 \to e_1 \;\Box\; \ldots \;\Box\; p_n \to e_n$ the selector $!i!1$ names p_i and $!i!2$ names e_i. If p_n is the token OTHER, then it cannot be named.

- In $\mathrm{WF}_e(A)$ and $\mathrm{SF}_e(A)$ the selector $!1$ names e and $!2$ names A.

- In $[A]_e$ and $\langle A \rangle_e$ the selector $!1$ names A and $!2$ names e.

- In $\text{LET } \ldots \text{ IN } e$ the selector $!1$ names e. This is rather subtle because we are naming an expression that contains operators defined in the LET clause that are not defined in the context in which the subexpression name appears. Consider this example

  $$\begin{array}{lcl} F & \triangleq & \text{LET } G \triangleq 1 \text{ IN } G + 1 \\ G & \triangleq & 22 \\ H & \triangleq & F!1 \end{array}$$

  The $F!1$ in the definition of H names the expression $G + 1$ in which G has the meaning it acquires in the LET definition. Thus, H is equal to 2, not to 23.

  We will see below how to name subexpressions of LET definitions, such as the first (local) definition of G above.

I now describe selectors for subexpressions of constructs that introduce bound identifiers. Consider this example:

$$R \triangleq \exists x \in S, y \in T : x + y > 2$$

- $R!(X, Y)$ names $X + Y > 2$, for any expressions X and Y.

- $R!1$ names S.

- $R!2$ names T.

In general, for any construct that introduces bound identifiers:

- $!(e_1, \ldots, e_n)$ selects the body (the expression in which the bound identifiers may appear) with each expression e_i substituted for the i^{th} bound identifier.

- If the bound identifiers are given a range by an expression of the form "$\in S$", then $!i$ selects the i^{th} such range S.

For example, in the expression

$$[x, y \in S,\ z \in T \mapsto x + y + z]$$

the selector $!1$ names S, the selector $!2$ names T, and the selector $!(X, Y, Z)$ names $X + Y + Z$.

Parentheses are "invisible" with respect to naming. For example, it doesn't matter if ν names the subexpression $a + b$ or the subexpression $((a + b))$; in either case, $\nu!\langle$ names a.

We usually don't need to name the entire expression to the right of a "$\triangleq$" because the operator being defined names it. However, as observed in Section 2, this is not true for recursively defined operators. If *Op* is recursively defined by

$$Op(p_1, \ldots, p_k) \ \triangleq\ exp$$

then "$Op(P_1, \ldots, P_k)!:$" names *exp* with P_i substituted for p_i, for each i in $1 \,..\, k$.

A *positional subexpression name* consists of a labeled subexpression name (defined in Section 6.1 above) followed by a sequence of positional selectors. For example, in

$$F(c) \ \triangleq\ a \ * \ lab :: (b + c * d)$$

$F(7)!lab!\rangle$ names $7 * d$. Remember that a labeled subexpression need not contain labels—for example, $F(7)$ is a labeled subexpression name.

6.3 Subexpressions of LET Definitions

If a positional subexpression name ν names a *let/in* expression and *Op* is an operator defined in the *let* clause, then $\nu!Op(e_1, \ldots, e_n)$ is the name of the expression $Op(e_1, \ldots, e_n)$ interpreted in the context determined by ν. For example, in

$$\begin{array}{lll} F(a) & \triangleq & \land \ \ldots \\ & & \land \ \text{LET } G(b) \ \triangleq \ a + b \\ & & \quad\ \text{IN} \ \ \ldots \end{array}$$

$F(A)!2!G(B)$ names the expression $G(B)$, where the definition of G is interpreted in a context in which A is substituted for a. This expression of course equals $A+B$. (However, if G were recursively defined, $F(A)!2!G(B)$ might not be so simply related to the expression to the right of the "$\triangleq$" in G's definition.) We can also name subexpressions of the definition of G. For example, $F(A)!2!G(B)!\rangle$ names B. The naming process can be continued all the way down, naming subexpressions of *let* definitions contained within *let* definitions contained within . . . .

If the *let*/*in* expression is labeled, then it can be named by a labeled subexpression name λ. In that case, $\lambda!\,Op(e_1, \ldots, e_n)$ is a labeled subexpression name that names a subexpression of the *in* clause with label $Op(p_1, \ldots, p_n)$. To refer to the operator *Op* defined in the *let* clause, just add a "$!:$" to the end of λ, writing $\lambda!:!\,Op(e_1, \ldots, e_n)$. In particular, if H is defined to equal the *let*/*in* expression, then we write $H!:!\,Op(e_1, \ldots, e_n)$, even if H is not recursively defined.

6.4 Subexpressions of an *Assume*/*Prove*

If we have

$$\text{THEOREM } Id \;\triangleq\; \text{ASSUME } A_1, \ldots, A_n \text{ PROVE } G$$

then *Id* is not an expression and cannot be used as one. Subexpressions of an *assume*/*prove* can be named with labels or positionally, where $Id!i$ names A_i if $1 \leq i \leq n$, and $Id!n{+}1$ names G. However, the assumptions can contain declarations like NEW C, so it is possible to name a subexpression of an *assume*/*prove* that contains identifiers declared within the *assume*/*prove*. Such a name can be used only within the scope of those declarations. For example, consider

$$\begin{array}{l} \text{THEOREM } T \triangleq \begin{array}{ll} \text{ASSUME} & x > 0,\ \text{NEW } C \in Nat,\ y > C \\ \text{PROVE} & x + y > C \end{array} \\ \quad\vdots \\ Foo \triangleq \ldots \end{array}$$

Then $T!1$ names the expression $x > 0$, which can be used in the definition of *Foo*. However, $T!3$ names the expression $y > C$ that contains the constant C, and the definition *Foo* is not within the scope of the declaration of C, so $T!3$ cannot be used within the definition of *Foo*. In fact, $T!3$ can be used only within the proof of T. (Proofs are discussed in Section 7.)

6.5 Using Subexpression Names as Operators

Subexpression names can be used as operator names by replacing every part of the form $!id(e_1, \ldots, e_n)$ by $!id$, and every selector $!(e_1, \ldots, e_n)$ by $!@$. For example, consider:

$$F(Op(\_,\_,\_)) \triangleq Op(1,2,3)$$
$$G \triangleq \forall x : P \subseteq \{\langle x,\ y+z \rangle : y \in S,\ z \in T\}$$

Then $G!(X)!\rangle!(Y, Z)$ is the expression $\langle X,\ Y+Z \rangle$, so $G!@!\rangle!@$ is the operator

$$\text{LAMBDA } x,\ y,\ z : \langle x,\ y+z \rangle$$

and $F(G!@!\rangle!@)$ equals $\langle 1,\ 2+3 \rangle$.

7 The Proof Syntax

This section describes the syntax of proofs and how proofs are checked by TLAPS, the TLA$^+$ proof checker.

7.1 The structure of a proof

A theorem is optionally followed by a proof. A proof is either a terminal proof or a sequence of steps, some of which have proofs. Figure 1 shows a possible proof structure, where the actual assertions made by the steps or by the terminal proofs are elided. This example is a proof having level number 1 and consisting of three steps named $\langle 1 \rangle 1$, $\langle 1 \rangle 2$, and $\langle 1 \rangle 3$. Step $\langle 1 \rangle 1$ has a level-2 proof that consists of three steps, one named $\langle 2 \rangle 4a$, an unnamed step (marked by the token "$\langle 2 \rangle$"), and a QED step named $\langle 2 \rangle 11$. Step $\langle 2 \rangle 4a$ has a terminal proof. The unnamed level-2 proof step has a four-step proof with level number 17. Only its first step has a proof—a terminal proof asserting that the actual proof is omitted.

A proof may optionally begin with the token PROOF. Thus, the PROOF token that begins the proof of step $\langle 1 \rangle 1$ and that precedes the token OMITTED could be removed, and a PROOF token could be added before step $\langle 1 \rangle 1$, before the "OBVIOUS" terminal proof, before the first level $\langle 17 \rangle$ step, and before either of the BY proofs. The formatting is for readability only; indentation has no significance.

In general, a proof consists of the optional keyword PROOF followed by either a terminal proof or else by a sequence of steps followed by a QED

$\langle 1\rangle 1.$ ...

  PROOF

    $\langle 2\rangle 4a.$ ...

      OBVIOUS

    $\langle 2\rangle$ ...

      $\langle 17\rangle$ ...

        PROOF OMITTED

      $\langle 17\rangle 1.$ ...

      $\langle 17\rangle$ ...

      $\langle 17\rangle ab$ QED

    $\langle 2\rangle 11$ QED

      BY ...

$\langle 1\rangle 2.$ ...

  BY ...

$\langle 1\rangle 3.$ QED

Figure 1: The structure of a simple proof.

step. A step or a QED step may have a proof, which is called a *subproof* of the proof containing the step. A terminal proof consists of the keyword OBVIOUS or OMITTED or else begins with the keyword BY.

Each step begins with a *step-starting token* that consists of a *step name* followed by an optional sequence of periods. A step name consists of

- `<` (printed as "$\langle$")
- a number called the step's *level number* or a $+$ or $*$ character. (The meaning of $+$ and $*$ is explained below.)
- `>` (printed as "$\rangle$")
- an optional string of letters and/or digits. If this string is present, then the step is said to be *named* and its *step name* consists of the entire token up to and including this string.

Since a step-starting token is a single token, it may not contain spaces. (Note that a step-starting token is the one place in which "$\langle$" and "$\rangle$" are typed "`<`" and "`>`" rather than "`<<`" and "`>>`".) All the steps of a proof have the same level number, which is less than that of any of its subproofs. A step with a greater level number than the preceding step begins the proof of that preceding step, whether or not it is preceded by a PROOF token.

Named steps are referred to by their step names. The scope of a level k step name (the part of a proof within which it can be used) consists of the step's proof (if it has one), all the level-k steps in the same proof that follow it and in those steps' proofs. A step name cannot be used within its scope to label another step. However, the same step name can be used in different subproofs of a proof. For example, step names $\langle 2\rangle 4a$ and $\langle 17\rangle 1$ could be used in a proof of step $\langle 1\rangle 3$.

The level number of a step may be written implicitly with a "$*$" or a "$+$". To explain the meaning of such a level number, let us define the *current level* at a proof step to equal -1 for the first step of the entire proof, and otherwise to equal the level of the latest preceding step that is neither a QED step nor followed by a QED step of the same level. In the example above, the current level at step $\langle 1\rangle 1$ is -1, the current level at step $\langle 2\rangle 4a$ is 1, and the current level at step $\langle 2\rangle 11$ is 2. Let L be the current level at a step whose step-starting token begins with "$\langle * \rangle$" or "$\langle + \rangle$". Then

- a "$+$" is equivalent to the number $L+1$, and
- a "$*$" is equivalent to the number $L+1$ if it immediately follows a PROOF token or is at the beginning of the entire proof; otherwise it is equivalent to the number L.

In the above example, $\langle 1\rangle 1$ can be replaced by either $\langle + \rangle 1$ or $\langle * \rangle 1$; $\langle 2\rangle 4a$ can be replaced by $\langle + \rangle 4a$ or $\langle * \rangle 4a$; and either of the other two "$\langle 2\rangle \ldots$" tokens could be replaced by "$\langle * \rangle \ldots$". If the PROOF token before it were missing, then $\langle 2\rangle 4a$ could be replaced only by $\langle + \rangle 4a$ and not by $\langle * \rangle 4a$. In all cases, it makes no difference if we use the "$*$" or "$+$" or the equivalent explicit level number.

A "$*$" can also be used instead of a level number in a reference to a proof step, in which case it stands for the current level. For example, you can write $\langle * \rangle 4a$ instead of $\langle 2\rangle 4a$ in the *by* statement that is the proof of step $\langle 2\rangle 11$. Again, it makes no difference if you write "$*$" or the equivalent explicit level number.

> TLAPS (the TLA$^+$ prover) does not yet support references to proof steps that use "$*$" as level number.

7.2 Use, Hide, and By

7.2.1 Use and Hide

At any point in a module, there is a set of *current declarations*, a set of *current definitions*, and a set of *known facts*. Outside a proof, the current declarations come from *constant* or *variable* declarations within the module and within modules it extends; the current definitions come from definitions within the module and within extended or instantiated modules; and the facts come from assumptions and theorems asserted thus far in the module and in extended modules, and from assertions imported thus far by instantiation. Each theorem in an instantiated module yields the assertion that the instantiated theorem follows from the instantiation of the module's assumptions. For example, if module M contains the single assumption

$\textsc{assume}\ A$

and the theorem

$\textsc{theorem}\ \mathit{Thm} \triangleq\ T$

then the statement

$\mathit{Mod} \triangleq \textsc{instance}\ M\ \textsc{with}\ \ldots$

imports a theorem named $\mathit{Mod}!\mathit{Thm}$ that asserts

$\textsc{assume}\ \overline{A}$
$\textsc{prove}\ \overline{T}$

where $\overline{A}$ and $\overline{T}$ are the formulas obtained from A and T by performing the substitutions specified by the INSTANCE statement's WITH clause.

There are also subsets of the sets of current definitions and known facts called the *usable definitions* and the *usable facts*. These are the definitions that TLAPS expands and the facts that it tries to apply when trying to prove something. (The definitions referred to here are "outer-level" definitions and not LET definitions, which are always expandable.) Here are the default values of these subsets at points in a module outside a proof.

- Only the definitions of theorem names are usable. (Section 4.1 explains how theorems are named.)

- No theorems or assumptions are usable.

The defaults can be overridden by *use* and *hide* statements. Such statements can appear anywhere in the body of the module—that is, at the "top level", not inside any other statements. A *use* or *hide* statement consists of the keyword USE or HIDE followed by an optional list of facts, optionally followed by the keyword DEF or DEFS and a list of definition specifiers. (It must include at least one fact or definition specifier.)

A fact is one of the following:

- The name of a theorem, assumption, or proof step.
- An arbitrary formula—but only in a USE statement, not a HIDE statement. The formula must be easily provable from the currently usable facts and the preceding facts in the USE statement. "Easily provable" means that a proof tool should be able to find the proof without any help from the user.

The parser also allows the following two kinds of "facts" in a USE or HIDE statement. However, they are not supported by TLAPS and are likely to be removed from the language.

- MODULE *Name*, indicating that all known facts obtained from the module *Name* are to be added or removed from the set of usable facts. The module name must appear in an EXTENDS or INSTANCE statement or else be the name of the current module.
- An identifier *Id* that appears in a statement of the form

  $$Id \;\triangleq\; \text{INSTANCE } M \;\ldots$$

  It adds or removes from the set of usable facts all facts imported from module M. The INSTANCE statement cannot have parameters—that is, it can't be of the form $Id(x) \;\triangleq\; \ldots.$

Theorems in certain special standard modules will direct TLAPS to use decision procedures or proof tactics. For example, there will be a theorem named *SimpleArithmetic* that causes TLAPS to apply a certain decision procedure for arithmetic when trying to prove something.

A definition specifier is the name of a defined operator—for example,

- F if the module contains the definition $F(x, y) \triangleq \ldots.$
- $Ins!F$ if the current module contains $Ins(a) \triangleq$ INSTANCE M ... and F is defined in M.

The SANY parser also accepts the following two kinds of definition specifiers. However, they are not supported by TLAPS and will probably be eliminated from the language.

- MODULE *Name*, indicating that all definitions from the module *Name* are to be added or removed from the set of usable definitions. The module name must appear in an EXTENDS or INSTANCE statement or else be the name of the current module.

- An identifier *Id* that appears in a statement

  $$Id(p_1, \ldots, p_k) \;\triangleq\; \text{INSTANCE } M \;\ldots$$

  (possibly with $k = 0$). It indicates that all the definitions imported from the instantiation are to be added or removed.

7.2.2 By

A terminal *by* proof has the same syntax as a *use* statement, except that it starts instead with the keyword BY. As explained below, at any point in a proof there will be sets of known and usable facts and of current and usable definitions. There will also be a current goal. A *by* proof asserts that this goal follows easily from the set of usable facts together with the set of facts specified in the *by* statement, using only those definitions contained in the set of usable definitions or specified by the statement. "Easily" means that a proof tool should be able to find the proof without any help from the user.

In addition to names of theorems, assumptions, and steps, a fact in a BY statement can be an arbitrary formula. Such a fact must follow easily from the set of usable facts together with the previous facts in the BY statement. For example, suppose the set of currently usable facts includes the fact $e \in S$. You might write

$$\begin{array}{l} \langle 3\rangle 1.\ \forall x \in S : P(x) \\ \langle 3\rangle 2.\ P(e+1) \\ \qquad \text{BY } \langle 3\rangle 1,\ P(e) \end{array}$$

The fact $P(e)$, which follows from $e \in S$ and $\langle 3\rangle 1$, makes the proof easier to understand (and easier for a prover to check) by alerting the reader that to prove $P(e+1)$ from usable facts and the fact $\langle 3\rangle 1$, he (or it) should first note that $P(e)$ follows from these facts. Arbitrary expressions can also be used as facts in a USE, but not in a HIDE.

A *by only* proof begins with the keywords BY ONLY. Unlike in an ordinary *by* proof, the current goal must follow easily from just the specified facts and the currently known domain formulas, without using any other usable facts. TLAPS uses all currently usable definitions plus the ones specified by the DEF clause.

TLAPS also allows the use in a BY or BY ONLY proof or in a USE statement of a fact that is trivially equivalent to a known (but not necessarily usable) fact. For example, if a module contains

$$\text{THEOREM } \mathit{Elementary} \;\triangleq\; 1 + 1 = 2$$

then the facts *Elementary* and $1+1 = 2$ can be used interchangebly anywhere within the scope of the definition of *Elementary*.

7.2.3 Obvious and Omitted

The terminal proof OBVIOUS asserts that the current goal follows easily from the set of known facts and the definitions contained in the set of usable definitions.

The terminal "proof" OMITTED means that the user is asserting the validity of the step without providing a proof. It asserts that the user has deliberately chosen not to provide a proof, and has not omitted it either accidentally or temporarily while writing other parts of the proof.

A proof is incomplete if it contains a statement with no proof. Incomplete proofs will be the norm while a user is developing the proof. TLAPS attempts to check a step only if the step has a proof other than the terminal "proof" OMITTED.

7.3 The Current State

At each point in a proof there is a current state that consists of:

- The set of current declarations.
- The set of current definitions and a subset consisting of the usable definitions.
- A set of currently known facts and a subset consisting of the usable facts.
- A *current* goal, which is a formula.

Recall that at the start of a theorem, there are sets of current declarations, current and usable definitions, and facts and usable facts described above. The state at the start of the theorem's proof is obtained by adding to these sets the following:

- If the theorem asserts a formula, then the formula becomes the current goal.
- If the theorem asserts an *assume/prove*, then the declarations in the assumptions are added to the set of current declarations. The set of formulas and *assume/prove*s asserted in the assumptions is added to the set of known facts; it is also added to the set of usable facts iff

the theorem has no name. The *prove* formula becomes the current goal. (If an assumption is an *assume/prove*, then the declarations of the inner *assume* are not added to the set of current declarations.)

Remember that the assumption NEW $C \in S$ is an abbreviation for the declaration NEW C and the assertion $C \in S$. An assertion of the form $C \in S$ obtained from a declaration is called a *domain formula.* Domain formulas are always added to the set of usable facts as well as to the set of known facts, even if the theorem is unnamed.

After the theorem's proof (if any), the current state reverts to the state right before the theorem, with the theorem added to the set of known facts iff it is named. An unnamed theorem can never be used in a proof.

To explain the meaning of a step, we describe the relation between the state of the proof at the (beginning of the) step and

- the state at the beginning of the statement's proof (if it has one), and
- the state immediately after the statement and its proof (if it has one).

7.4 Steps That Take Proofs

In the following descriptions, σ will be used to denote an arbitrary step-starting token.

7.4.1 Formulas and Assume/Proves

A step that asserts a formula or an *assume/prove* affects the state exactly the same way as a theorem. It makes either the formula or the *prove* assertion the current goal of the step's proof. The formulas and *assume/prove*s from the *assume* clause are added to the set of usable facts iff the step is unnamed. However, domain formulas obtained from NEW clauses are always added to the set of usable facts.

After the step and its proof, the step's assertion is added to the set of usable facts iff the step is unnamed. (An unnamed step can never be referred to in a *by* or *use*, so the step's assertion must be put into the set of usable facts for it ever to be used.)

7.4.2 Case

A *case* step consists of the step-starting token followed by the keyword CASE and a formula. The step "σ CASE F" is equivalent to

σ ASSUME F PROVE G

where G is the current goal. (Since G is already the current goal, this means that the current goal remains the same.)

7.4.3 @ Steps

A common method of proving an inequality is by proving a sequence of inequalities. For example, to prove $A \leq D$, we might prove $A \leq B \leq C \leq D$. Such a proof might appear inside a proof as follows (where the proofs of the individual steps are omitted).

$\langle 2 \rangle 3.\ A \leq D$
 $\langle 3 \rangle 1.\ A \leq B$
 $\langle 3 \rangle 2.\ B \leq C$
 $\langle 3 \rangle 3.\ C \leq D$
 $\langle 3 \rangle 4.$ QED
  BY $\langle 3 \rangle 1,\ \langle 3 \rangle 2,\ \langle 3 \rangle 3$

It's a nuisance to have to write B and C twice if they're large formulas. TLA^{+2} provides the following abbreviated way of writing this proof.

$\langle 2 \rangle 3.\ A \leq D$
 $\langle 3 \rangle 1.\ A \leq B$
 $\langle 3 \rangle 2.\ @ \leq C$
 $\langle 3 \rangle 3.\ @ \leq D$
 $\langle 3 \rangle 4.$ QED
  BY $\langle 3 \rangle 1,\ \langle 3 \rangle 2,\ \langle 3 \rangle 3$

This style of reasoning can be used with any transitive operator or combination of operators, such as

$$\begin{array}{ccccccc} A & = & B & = & C & = & D \\ A & \Rightarrow & B & \Rightarrow & C & \Rightarrow & D \\ A & \subseteq & B & \subseteq & C & \subseteq & D \\ A & \leq & B & < & C & \leq & D \end{array}$$

However, the token @ followed by an infix operator followed by an expression can be used in a step that follows any @ step or any formula step in which the formula's top-level operator is an infix operator. The "@" then refers to the right-hand side of the preceding step's formula. Although it's bad style and you shouldn't do it, you could write

$\langle 3 \rangle 4.\ A \leq B$
$\langle 3 \rangle 5.\ @ > C$

where the @ stands for B.

7.4.4 Suffices

The step σ SUFFICES A asserts that proving A proves the current goal, where A can be a formula or an *assume/prove*. At the beginning of the step's proof, A is added to the set of known facts and to the set of usable facts. (The proof must prove the current goal.) After the step and its proof:

- If A is a formula, then it is made the current goal.
- If A is an *assume/prove*, then:
  - The declarations in its assumptions are added to the set of current declarations and the domain formulas from those declarations are added to the sets of known and usable facts.
  - The assertions among its assumptions (the formulas and the *assume/proves*) are added to the set of known facts. They are added to the set of usable facts iff the step is not named.
  - The *prove* formula is made the current goal.

7.4.5 Pick

A *pick* step has the same syntax as one that asserts a $\exists$ formula, except with the $\exists$ replaced by the token PICK. For example, the step

σ PICK $x \in S,\ y \in T\ :\ P(x, y)$

asserts that there exist values x in S and y in T satisfying $P(x, y)$, and then declares x and y to be equal to an arbitrary pair of such values. The state at the start of the step's proof is the same as for the formula obtained by replacing PICK by $\exists$. After the proof:

- *Constant* declarations of the identifiers introduced by the step are added to the set of declarations and the domain formulas of those declarations are added to the sets of known and usable facts. (In this example, the domain formulas are $x \in S$ and $y \in T$.)
- The body of the PICK (in this example, the formula $P(x, y)$) is added to the set of known facts. It is added to the set of usable facts iff the step is not named.

A *pick* step is effectively translated to two steps. For example, the step and its proof

$$\begin{array}{l} \sigma \ \ \text{PICK } x \in S,\ y \in T : P(x, y) \\ \qquad \text{PROOF } \Pi \end{array}$$

are translated to

$$\begin{array}{l} \rho \ \ \exists\, x \in S,\ y \in T : P(x, y) \\ \qquad \text{PROOF } \Pi \end{array}$$

$$\begin{array}{llll} \sigma & \text{SUFFICES ASSUME} & \text{NEW } x \in S,\ \text{NEW } y \in T \\ & & P(x, y) \\ & \qquad\quad \text{PROVE} & G \end{array}$$

and σ contains a proof asserting that it follows from ρ, where ρ is a new step name and G is the current goal. This translation is relevant to the meaning of the step name σ. (See Section 7.6.2 on page 30.)

7.4.6 QED

The state at the beginning of a *qed* step's proof is unchanged. After the step and its proof, the state is determined by the rule for the step whose proof the *qed* step ends.

7.5 Steps That Do Not Take Proofs

7.5.1 Definitions

In a *definition* step, the step-starting token is followed by the optional token DEFINE and a sequence of operator definitions, function definitions, and/or module definitions, where a module definition is something like

$$Ins(x) \ \triangleq \ \text{INSTANCE } M \text{ WITH } \ \ldots$$

It has the same effect on the state as the corresponding (top-level) statements. The definitions introduced by the step (which are the definitions of the imported and renamed operators for a module definition) are added to both the set of definitions and the set of usable definitions.

7.5.2 Instance

An *instance* step consists of a step-starting token followed by an ordinary *instance* statement (one that begins with the keyword INSTANCE). It has the same effect on the state as the corresponding (top-level) statement.

7.5.3 Use and Hide

A *use* or *hide* step has the same syntax as the corresponding (top-level) statement, except preceded by the step-starting token. It affects the sets of usable facts and definitions the same way as the corresponding *use* or *hide* statement. As explained in Section 7.6 below, a *use* or *hide* step can name facts or definitions made in earlier steps.

There is also a *use only* step, in which the keyword USE is followed by the keyword ONLY. It sets the usable facts to be only known domain facts and facts specified by the step. It affects the set of usable definitions the same way as an ordinary *use* step.

7.5.4 Have

A *have* step consists of a step-starting token followed by HAVE and a formula. For the statement

$$\sigma \text{ HAVE } F$$

to be correct, the current goal must be syntactically of the form $H \Rightarrow G$ for some formulas H and G, and the formula $H \Rightarrow F$ must be an obvious consequence of the known facts and usable definition. In that case, the step is equivalent to

$$\begin{array}{ll} \sigma \text{ SUFFICES} & \text{ASSUME } F \\ & \text{PROVE } G \end{array}$$

plus a BY ONLY proof that permits using only the fact ASSUME F PROVE B (a fact that must therefore be easily provable with no assumptions). Thus, this step means that we are going to prove the current goal by assuming F and proving G.

7.5.5 Take

A *take* step consists of a step-starting token followed by TAKE followed by anything that could come between "∀" and its matching ":"—for example

$$\sigma \text{ TAKE } x, y \in S,\ z \in T$$

This step is typically used when the current goal is

$$\forall\, x, y \in S,\ z \in T : G$$

for some formula G. It means that we are going to prove this goal by declaring x, y, z to be constants, assuming $x \in S$, $y \in S$, and $z \in T$, and proving G. More precisely this TAKE statement is equivalent to

$$\begin{array}{rl} \sigma \text{ SUFFICES ASSUME} & \text{NEW CONSTANT } x \in S, \\ & \text{NEW CONSTANT } y \in S, \\ & \text{NEW CONSTANT } z \in T \\ \text{PROVE} & G \end{array}$$

followed by a proof that permits using only the domain formulas $x \in S$, $y \in S$, and $z \in T$.

In general, for the step σ TAKE τ to be correct, the current goal should be obviously equivalent to $\forall \tau : G$ for some formula G. (Again, the meaning of "obviously equivalent" is not specified.) In that case, G is made the current goal, constant declarations of the bound identifiers in τ are added to the current set of declarations, and any formulas of the form $id \in e$ in τ are added to the set of known facts and to the set of usable facts.

7.5.6 Witness

A *witness* step consists of a step-starting token, followed by WITNESS, followed by a comma-separated list of expressions. A *witness* step is used to prove an existentially quantified formula by specifying instantiations of its bound identifiers. There are two cases in which the statement σ WITNESS $e_1, \ldots, e_k$ is correct:

- The current goal is obviously equivalent to a formula $\exists\, id_1, \ldots, id_k : G$. In this case, the WITNESS step is equivalent to

  $$\begin{array}{l} \sigma \text{ SUFFICES } \overline{G} \\ \quad \text{PROOF OBVIOUS} \end{array}$$

  where $\overline{G}$ is the formula obtained by substituting each e_j for id_j in G, for j in $1 \,..\, k$.

- The current goal is obviously equivalent to a formula $\exists\, \iota_1 \in S_1, \ldots, \iota_k \in S_k : G$ where each ι_j is an identifier, there is some substitution of expressions for these identifiers that transforms each $\iota_j \in S_j$ to e_j, and each e_j is easily provable from the current set of usable facts. In this case, the formula obtained from G by the aforementioned substitution of expressions for the identifiers in the ι_j is made the current goal, and the domain formulas e_j are added to the set of known facts and to the set of usable facts. (Adding a fact that is easily provable to the set of usable facts might make additional facts easily provable from that set.) For example, if the current goal is

$$\exists\, x, y \in S,\ z \in T : G(x, y, z)$$

then the step

$\langle 3\rangle 4.$ WITNESS $\mathit{expX} \in S,\ \mathit{expY} \in S,\ \mathit{expZ} \in T$

is equivalent to

$\langle 3\rangle 4.$ SUFFICES $G(\mathit{expX},\ \mathit{expY},\ \mathit{expZ})$
  $\langle 4\rangle 1.$ $\mathit{expX} \in S$
    PROOF OBVIOUS
  $\langle 4\rangle 2.$ $\mathit{expY} \in S$
    PROOF OBVIOUS
  $\langle 4\rangle 3.$ $\mathit{expZ} \in T$
    PROOF OBVIOUS
  $\langle 4\rangle 4.$ QED
    BY ONLY $\langle 4\rangle 1, \langle 4\rangle 2, \langle 4\rangle 3$

7.6 Referring to Steps and Their Parts

Within a proof, steps and their parts can be named in three contexts: as ordinary expressions, as facts in a *by*, *use*, or *hide*, and in the DEF clause of one of those statements. We now consider these three possibilities.

7.6.1 Naming Subexpressions

Formulas The name of a step that asserts a formula names that formula. For example, the step

$\langle 2\rangle 3.$ $x + y = z$

defines $\langle 2\rangle 3$ to equal $x + y = z$. The step name $\langle 2\rangle 3$ can be used like any other defined symbol—for example:

$$\langle 2\rangle 3 \wedge (z \in Nat) \;\Rightarrow\; (x + y - z = 0)$$

We can also use labels and/or positional selectors to name subexpressions of $\langle 2\rangle 3$ the same way we name subexpressions of other defined symbols—for example, $\langle 2\rangle 3!\langle$ names the subexpression $x + y$. (See Section 6.)

Assume/Prove Steps The parts of an *assume/prove* step are named as explained in Section 6.4, where the step number names the *assume/prove*. Thus, in

$$\begin{array}{lll} \langle 3\rangle 4.\ \textsc{assume} & P, & \textsc{assume}\ \ Q \\ & & \textsc{prove}\ \ \ \ R \\ \qquad\ \ \textsc{prove} & S & \end{array}$$

$\langle 3\rangle 4!1$ names P, $\langle 3\rangle 4!2!\langle$ names Q and $\langle 3\rangle 4!3$ names S.

As explained in Section 6.4, subexpressions of an *assume/prove* can be used only within the scope of any identifier that could appear in that subexpression (even if it that identifier doesn't actually appear in it).

Case, Have, Suffices, and Witness Steps Expressions within a *case*, *have*, *suffices*, or *witness* step are named as if CASE, HAVE, SUFFICES, and WITNESS were prefix operators—CASE, HAVE, and SUFFICES taking a single argument and WITNESS taking an arbitrary number of arguments. Thus, in

$\langle 2\rangle 3.\ \textsc{case}\ x + y > 0$
$\langle 2\rangle 4.\ \textsc{witness}\ y,\ x + 1$

$\langle 2\rangle 3!1$ equals $x+y > 0$ and $\langle 2\rangle 3!1!\langle$ equals $x+y$, while $\langle 2\rangle 4!2$ equals $x+1$. The "argument" of SUFFICES can be an *assume/prove*, whose subexpressions are named as described above for an *assume/prove* step.

Pick and Take A subformula of a *pick* step is named as if the *pick* were replaced by $\forall$. For example, in

$\langle 3\rangle 4.\ \textsc{pick}\ x \in S,\ y \in T\ :\ x + y > 0$

$\langle 3\rangle 4!2$ names T and $\langle 3\rangle 4!(e, f)$ names $e + f > 0$. The naming of a *take* step is similar, except that there is no "body" to name, only the sets that follow an " $\in$ ".

Note that the symbols introduced in a *pick* step are not declared within the proof of that step, but they are declared after the proof. However, references to the body of the *pick* are made the same way in both places.

7.6.2 Naming Facts

Syntactically, any expression can be used as a fact. (A proof tool might accept only a restricted set of expressions as facts.) Any named step that makes an assertion can also be used as a fact. The only kinds of steps that can *not* be used as facts are *use*, *hide*, *definition*, *instance*, and *qed*.

The scope of a step name includes the proof of the step. Thus, it is legal to use a subexpression of a step named σ within that step's proof—for example, to name an assumption if σ is an *assume/prove* step.

When used by itself (and not in the name of one of its subexpressions), a step name denotes the fact or facts that the step adds to the current set of known facts. We now explain exactly what that means.

We have described above how every step that makes an assertion is equivalent to one of the form A or SUFFICES A, where A is either a formula or an *assume/prove*. If we consider a formula G to be equivalent to ASSUME TRUE PROVE G, then any step σ is equivalent to a step of the form $\sigma\ A$ or σ SUFFICES A for an *assume/prove* A.

- For the step $\sigma\ A$, within the proof's step the step name σ denotes the set of assumptions of A; outside the proof it denotes A.

- For the step σ SUFFICES A, within the proof's step the step name σ denotes A; outside the proof it denotes the set of assumptions of A.

It is quite useful to have a step name σ refer to the known facts introduced into the current context by the step, since those facts are not automatically added to the set of usable facts. However, it has the unfortunate effect of making a proof look circular when σ is used as a fact within the proof of the step named σ. Readers and writers of TLA$^+$ proofs should quickly get used to this convention.

One may want to refer to a long formula G inside a step $\sigma\ G$. For example, we can assume $\neg G$ in a proof by contradiction of the step. However, by these rules, σ names TRUE within the proof of σ, so we cannot write $\neg G$ as $\neg\sigma$. We can write $\neg G$ as $\neg\sigma!:$ instead.

7.6.3 Naming Definitions

Only the names of defined operators may appear in the DEF clause of a *by* proof or a *use* or *hide* step. These include the names of operators defined in *let* clauses. The step name of a *define* step may also be used in a DEF clause. If the step defines more than one operator, then the step name applies to all of them—but not to any operators defined in *let* clauses within those definitions.

Remember that an operator name does not contain any parameters or any parentheses. For example, the expression $Ins(42)!Foo(x, y)$ is an application of the operator named $Ins!Foo$ to the three arguments 42, x, and y. Section 6.5 explains how to name operators defined in a *let* clause.

7.7 Referring to Instantiated Theorems

Suppose module M contains a theorem

$$\textsc{theorem}\ T$$

and another module MI imports M with the statement

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

As explained in Section 17.5.5 of *Specifying Systems*, this imports the theorem that we can write in TLA^{+2} as

$$\begin{array}{l}\textsc{assume}\ \overline{A_1}, \ldots \overline{A_k} \\ \textsc{prove}\ \overline{T}\end{array}$$

where $A_1, \ldots, A_k$ are the assumptions asserted by ASSUME statements in M, and $\overline{A_i}$ and $\overline{T}$ are the formulas obtained from A_i and T by performing the substitutions specified by the WITH clause.

If the theorem is named, as in

$$\textsc{theorem}\ Thm \triangleq T$$

then $I!Thm$ names the imported fact—the ASSUME / PROVE above. However, the rules of Section 6.4 above for naming parts of an ASSUME / PROVE do not apply to this fact. The name $I!Thm!:$ refers to the formula $\overline{T}$. A formula $\overline{A_i}$ can be referred to in module MI only if it has been assigned a name in module M, in which case it is named $I!\ldots$ as usual. Because TLC cannot do anything with an ASSUME / PROVE, it treats $I!Thm$ (as well as $I!Thm!:$) as the name of $\overline{T}$.

A proof of module MI that uses the imported theorem Thm generally wants to use $\overline{T}$. To do that, it must prove all its hypotheses $\overline{A_i}$. Often, the formulas $\overline{A_i}$ are simple consequences of the assumptions of module MI. In that case, a proof can simply use $I!Thm$ in a context in which those assumptions of MI are usable facts.

7.8 Temporal Proofs

Temporal-logic reasoning has not yet been implemented in TLAPS. We give here a tentative description of how it will work. Section 8 more fully describes the semantic issues that must be addressed to handle temporal logic.

THEOREM $Init \land \Box[Next]_{vars} \land \mathrm{WF}_{vars} \Rightarrow \Diamond P$

  $\langle 1 \rangle 1.$ $Init \land \Box[Next]_{vars} \Rightarrow \Box Inv$

    $\langle 2 \rangle 1.$ $Init \Rightarrow Inv$

    $\langle 2 \rangle 2.$ $Inv \land [Next]_{vars} \Rightarrow Inv'$

    $\langle 2 \rangle 3.$ $Inv \land \Box[Next]_{vars} \Rightarrow \Box Inv$

      BY $\langle 2 \rangle 2$

    $\langle 2 \rangle 4.$ QED

      BY $\langle 2 \rangle 1$, $\langle 2 \rangle 3$

  $\langle 1 \rangle 2.$ $\Box Inv \land \Box[Next]_{vars} \land \mathrm{WF}_{vars} \Rightarrow \Diamond P$

  $\langle 1 \rangle 3.$ QED

    BY $\langle 1 \rangle 1$, $\langle 1 \rangle 2$

Figure 2: A temporal proof, with lower-level proofs omitted.

7.8.1 The Short Story

A temporal TLA$^+$ formula is one that contains any of these temporal operators:

$$\Box \quad \Diamond \quad \leadsto \quad \mathrm{WF} \quad \mathrm{SF} \quad \stackrel{+}{\rhd} \quad \boldsymbol{\exists} \quad \boldsymbol{\forall}$$

(Don't confuse the temporal operators $\boldsymbol{\exists}$ and $\boldsymbol{\forall}$ with the ordinary non-temporal operators $\exists$ and $\forall$.) A temporal theorem or proof step is one that asserts a temporal formula. Figure 2 is an example of a temporal theorem and its proof, with most of the lower-level subproofs omitted. The three level-1 steps are temporal steps, as are steps $\langle 2 \rangle 3$ and $\langle 2 \rangle 4$, the latter asserting the (temporal) formula of step $\langle 1 \rangle 1$.

Some of the rules of ordinary mathematical reasoning are not valid for temporal formulas. To make writing proofs easier, TLA^{+2} defines the precise meaning of temporal theorems and proof steps differently than for non-temporal ones. This difference is explained in Section 7.8.2 below. However, you won't have to worry about it as long as you obey these rules:

- Don't write temporal *Assume*/*Prove* theorems or proof steps.
- Don't use a HAVE, CASE, TAKE, or WITNESS step when the current goal is a temporal formula.

If you follow the rules, you're unlikely to do anything wrong.

7.8.2 The Fine Print

The main difference between temporal and non-temporal formulas in TLA^{+2} lies in the interpretation of the assertion ASSUME F PROVE G. For non-temporal F and G, this essentially asserts the truth of the formula $F \Rightarrow G$. The two steps

$\langle 3 \rangle 4.$ ASSUME F PROVE G $\qquad\qquad$ $\langle 3 \rangle 4.\ F \Rightarrow G$

differ only in the goal and the set of known facts for their proofs. The proof of the first has G as the goal and F added to the known facts; the proof of the second has $F \Rightarrow G$ as the goal and no additional known facts. These two steps make very different assertions if F or G is a temporal formula. To understand the difference, we need some notation.

A TLA$^+$ expression e assigns a value we write $\sigma \models e$ to any behavior σ. Only if e is a temporal formula does this value depend on all of σ. For example, if v is a variable, then:

- $\sigma \models 42$ is independent of σ.
- $\sigma \models (v + 42)$ depends only on the first state of σ.
- $\sigma \models (v' + 42 > v)$ depends only on the first two states of σ.
- $\sigma \models \Box(x > 0)$ depends on all of σ.

If F is a formula (a Boolean-valued expression), then $\sigma \models F$ is a *meta-formula* asserting that F is true in behavior σ. We let $\models F$ be the meta-formula $\forall \sigma : F$, which asserts that F is true in all behaviors. If $\models F$ equals TRUE, we say that F is *valid*.

In TLA^{+2}, a theorem or proof step that asserts a non-temporal formula F is considered to assert the meta-formula $\tau \models F$ for some particular but arbitrary behavior τ. The proof fragment

$\langle 3 \rangle 2.\ x > 3$
$\langle 3 \rangle 3.\ x > 2$
BY $\langle 3 \rangle 2$

is correct because $\langle 3 \rangle 2$ asserts $\tau \models (x > 3)$ and $\langle 3 \rangle 3$ asserts $\tau \models (x > 2)$, and $\tau \models (x > 3)$ implies $\tau \models (x > 2)$. The theorem

THEOREM $(x \in Nat) \land (x' = x + 1) \Rightarrow (x' \in Nat)$

asserts the meta-formula

$$\tau \models (x \in Nat) \land (x' = x + 1) \Rightarrow (x' \in Nat)$$

Since this is true for an arbitrary behavior τ, it is true for all behaviors, so the meta-formula

$$\models (x \in \mathit{Nat}) \wedge (x' = x + 1) \Rightarrow (x' \in \mathit{Nat})$$

is also true. Hence asserting that a particular formula is true for the arbitrary behavior τ means that the formula is valid. Our interpretation of non-temporal theorems is consistent with the usual idea that a theorem asserts the validity of a formula.

For non-temporal formulas F and G, we interpret a theorem or proof step that asserts ASSUME F PROVE G to assert the meta-formula $\tau \models (F \Rightarrow G)$, which is equivalent to the meta-formula $(\tau \models F) \Rightarrow (\tau \models G)$.

In TLA^{+2}, a theorem or proof step that asserts a temporal formula F is considered to assert the meta-formula $\models F$—that is, the validity of F. The proof fragment

$\langle 1 \rangle 2.\ \ \Diamond F$

$\langle 1 \rangle 3.\ \ \Box\Diamond F$
BY $\langle 1 \rangle 2$

is correct because $\langle 1 \rangle 2$ asserts $\models \Diamond F$ and $\langle 1 \rangle 3$ asserts $\models \Box\Diamond F$, and the truth of $\Diamond F$ for all behaviors implies the truth of $\Box\Diamond F$ for all behaviors. Deducing $\Box\Diamond F$ from $\Diamond F$ is standard temporal-logic reasoning. This interpretation of temporal assertions yields the proof rules we expect. Note that $\tau \models \Diamond F$ does not imply $\tau \models \Box\Diamond F$ because the truth of $\Diamond F$ for an individual behavior τ does not imply that $\Box\Diamond F$ is true for τ.

If F or G is a temporal formula, then we interpret a theorem or proof step that asserts ASSUME F PROVE G to assert the meta-formula $(\models F) \Rightarrow (\models G)$, which is *not* equivalent to the meta-formula $\models (F \Rightarrow G)$. (Remember that $\models F$ means $\forall \sigma : \sigma \models F$, and $(\forall x : A) \Rightarrow (\forall x : B)$ is not equivalent to $\forall x : (A \Rightarrow B)$.) The meta-formula $\models (F \Rightarrow G)$ is the interpretation of the assertion $F \Rightarrow G$. Thus, for temporal formulas, ASSUME F PROVE G does not assert the same thing as $F \Rightarrow G$.

We have chosen not to make ASSUME F PROVE G equivalent to $F \Rightarrow G$ in order to make it possible to express temporal logic proof rules in TLA^{+2}. In fact, the examples in Section 4.2 have the meaning given to them there because of this interpretation. As an example of how those rules are used, the proof of step $\langle 2 \rangle 3$ of Figure 2 uses the TLA proof rule

THEOREM ASSUME STATE Inv,
ACTION A, STATE v,
$\mathit{Inv} \wedge [A]_v \Rightarrow \mathit{Inv}'$
PROVE $\mathit{Inv} \wedge \Box[A]_v \Rightarrow \Box\mathit{Inv}$

mentioned on page 5. It is sound to use this proof rule in this case because step $\langle 2 \rangle 2$ asserts the truth of the meta-formula

$$\tau \models Inv \land [A]_v \;\Rightarrow\; Inv'$$

in a context containing no assumptions about the particular behavior τ, so it implies the truth of

$$\models Inv \land [A]_v \;\Rightarrow\; Inv'$$

which is the hypothesis of this TLA proof rule.

Temporal *Assume*/*Prove* assertions seem to be useful only for stating proof rules. They will appear only in the statement or proof of a proof rule. You will probably have no occasion to prove proof rules, so you're unlikely ever to write a temporal formula inside an *Assume*/*Prove*.

There is one important case in which $(\forall x : A) \Rightarrow (\forall x : B)$ is equivalent to $\forall x : (A \Rightarrow B)$—namely, when A does not depend on x. The meta-formula $\sigma \models F$ is independent of σ if F is a constant. Hence, ASSUME F PROVE G is equivalent to $F \Rightarrow G$ if F is a constant, even if G is a temporal formula.

The proof language we have described thus far can lead to proofs of incorrect temporal formulas. To make it sound for temporal reasoning, we must add further restrictions on proofs of temporal formulas. The precise rules have not yet been determined. See Section 8 for a sufficient rule.

8 The Semantics of Proofs

8.1 The Interpretation of Boolean Operators

As discussed in Section 16.1.3 of *Specifying Systems*, there are various ways to define the meanings of the Boolean operators on non-Boolean arguments. For example, we know that $x \land y$ equals $y \land x$ for any Booleans x and y. However, is $5 \land 7$ equals to $7 \land 5$? TLAPS uses what is called in *Specifying Systems* the *liberal* interpretation. In this interpretation, $(5 \land 7) = (7 \land 5)$ is true and TLAPS will prove it.

The precise interpretation of the Boolean operators is in terms of an operator *ToBoolean* such that $ToBoolean(x)$ is some Boolean that equals x if x is a Boolean. More precisely, *ToBoolean* is assumed to satisfy

$$\begin{array}{l} \land\ \forall x : ToBoolean(x) \in \mathrm{BOOLEAN} \\ \land\ \forall x \in \mathrm{BOOLEAN} : ToBoolean(x) = x \end{array}$$

For example, we can define conjunction $\land$ by

$$x \land y \;\triangleq\; ToBoolean(x) \mathbin{\underline{\land}} ToBoolean(y)$$

where $\triangle$ is ordinary conjunction on Booleans. The operator *ToBoolean* exists only in the semantics and is not defined at the TLA$^+$ level. However, it follows from the assumptions about *ToBoolean* that it can be defined in TLA$^+$ by

$$\mathit{ToBoolean}(x) \;\triangleq\; (x \equiv \text{TRUE})$$

8.2 The Problem with Temporal Logic

Logicians write $\vdash B$ to mean that B is provable, and $A \vdash B$ to mean that proving A proves B. Ordinary mathematical reasoning relies on the *Deduction Principle* $(A \vdash B) \vdash (A \Rightarrow B)$, which asserts that we can prove $A \Rightarrow B$ by assuming A and proving B. This proof rule is not sound for temporal logic (or other modal logics). To see why, let's briefly review temporal logic.

A (linear-time) temporal logic formula is an assertion about a behavior, which is a sequence of states. Let $[F]_\beta$ be the meta-formula asserting that F is true of behavior β. (This meta-formula is written $\beta \models F$ in Section 7.8.2.) We assume that you are familiar with the semantics of the temporal logic TLA. Here are some of the rules that define it, where P is any state predicate and F is any temporal formula:

- $[P]_\beta$ is true iff P is true in the first state of β.
- $[P']_\beta$ is true iff P is true in the second state of β.
- $[\Box F]_\beta$ is true iff F is true for any suffix of β.
- $[F \wedge G]_\beta = [F]_\beta \wedge [G]_\beta$
- $[\neg F]_\beta = \neg[F]_\beta$

In the last two equalities, $\wedge$ and $\neg$ on the left-hand side are operators of the TLA logic, and on the right-hand side are meta-operators (operators of the semantic domain). As illustrated by these equalities, the operators of ordinary logic distribute over $[\;]_\beta$.

In temporal logic, $F \vdash G$ asserts the provability of

$$(\forall \beta : [F]_\beta) \Rightarrow (\forall \beta : [G]_\beta)$$

The proof rule $F \vdash \Box F$ is valid, since if F is true for all behaviors β, then it is true for all suffixes of all behaviors β. From this and the Deduction Principle, we can prove $(x = 1) \Rightarrow \Box(x = 1)$, which asserts that for any behavior β, if $x = 1$ is true in the first state of β, then it is true in all states of β. This is obviously wrong, which shows why the Deduction Principle is invalid for temporal logic.

The Deduction Principle lies at the heart of all ordinary mathematical reasoning. That it does not hold means temporal logic is evil. Unfortunately, it is a necessary evil because it's the best way to reason about liveness properties. TLA is the least evil temporal logic I know because it relies as much as possible on non-temporal reasoning. The semantics of TLA$^+$ proofs is defined to make non-temporal reasoning as convenient as possible by maintaining the validity of the Deduction Principle for such reasoning.

8.3 The Meaning of an Assertion

The free identifiers of an ordinary TLA$^+$ formula are declared variables and declared constants. The meta-formula $[F]_\beta$ asserts that F is true of behavior β for all values of the declared constants. An *assume/prove* statement can contain declarations of other kinds of identifiers with associated levels. For example NEW ACTION A declares the identifier A to have action level. For convenience, we eliminate each such declaration and substitute for all occurrences of A a new free identifier, having action level, that cannot appear in any other formula or *assume/prove*. In examples, we take $A\$$ to be that new identifier. A meta-formula is valid if it is valid when any possible TLA$^+$ expression of the correct level is substituted for all its free identifiers. An expression has the correct level iff its level is at most equal to the level of the identifier.

An assertion is either a formula or an *assume/prove*. We now define the meaning of an assertion. For a formula G, we define the meaning $[\![G]\!]$ of G to be $[G]_\beta$ if G is a non-temporal formula and to be $\forall\,\beta : [G]_\beta$ if G is a temporal formula. We define the meaning of an *assume/prove* by defining $[\![\text{ASSUME } A_1, \ldots, A_n \text{ PROVE } G]\!]$ to equal:

- $[\![A_1]\!] \wedge \ldots \wedge [\![A_n]\!] \Rightarrow [G]_\beta$ if G is a non-temporal formula.
- $(\forall\,\beta : [\![A_1]\!]) \wedge \ldots \wedge (\forall\,\beta : [\![A_n]\!]) \Rightarrow \forall\,\beta : [G]_\beta$ if G is a temporal formula.

We say that A_i is a *β-assumption* in this *assume/prove* iff G is a non-temporal formula and β occurs free in the formula $[\![A_i]\!]$.

The Deduction Principle for non-temporal formulas can be written in

TLA$^+$ as follows:

$$\text{THEOREM } \mathit{DeductionPrinciple} \;\triangleq\; \begin{array}{ll} \text{ASSUME} & \text{NEW ACTION } A, \\ & \text{NEW ACTION } B, \\ & \text{ASSUME } A \text{ PROVE } B \\ \text{PROVE} & A \Rightarrow B \end{array}$$

The meaning of this theorem is

$$([A\$]_\beta \Rightarrow [B\$]_\beta) \Rightarrow [A\$ \Rightarrow B\$]_\beta$$

where $A\$$ and $B\$$ are new action-level identifiers. This is a valid formula because $\Rightarrow$ distributes over $[\,]_\beta$. If the ACTION declarations were replaced by TEMPORAL declarations (so the goal is considered to be a temporal formula), the meaning of assertion *DeductionPrinciple* would be

$$(\forall \beta : ((\forall \beta : [A\$]_\beta) \Rightarrow (\forall \beta : [B\$]_\beta))) \Rightarrow (\forall \beta : [A\$ \Rightarrow B\$]_\beta)$$

Since the first $\forall \beta$ is quantifying a formula with no free occurrence of β, this is equivalent to

$$((\forall \beta : [A\$]_\beta) \Rightarrow (\forall \beta : [B\$]_\beta)) \Rightarrow (\forall \beta : [A\$ \Rightarrow B\$]_\beta)$$

Because $\Rightarrow$ distributes over $[\,]_\beta$, this is in turn equivalent to

$$((\forall \beta : [A\$]_\beta) \Rightarrow (\forall \beta : [B\$]_\beta)) \Rightarrow (\forall \beta : ([A\$]_\beta \Rightarrow [B\$]_\beta))$$

This formula is clearly not valid for arbitrary $A\$$ and $B\$$.

8.4 Primitive Proofs

Our object is to determine the restrictions on proofs that lead to sound proofs. Recall that in Section 7.3 we defined the set of known facts at any point in a module. For simplicity, we do not consider USE and HIDE statements and pretend that each BY proof explicitly lists all required known facts. Each known fact comes either from a proved assertion (a theorem or proof step) or an assumption of an *assume/prove* assertion. We assume that the set of known facts identifies the source of each of its facts.

We can write the rules of first-order logic and of TLA as TLA$^+$ theorems. We assume that these theorems are in the set of known facts at the beginning of a module. We define a proof to be *primitive* if each leaf assertion follows from the known facts listed in its BY proof using only the following two inference rules:

I1. Replace all instances of a free identifier in a fact with any expression of the right level—meaning that either the expression has level at most equal to the level of the identifier, or else the fact (and hence the identifier) has constant level.

I2. Deducing a new fact G from the known facts $A_1, \ldots, A_n$, and $\textsc{assume}\ A_1, \ldots, A_n\ \textsc{prove}\ G$.

In principle, every proof can be expanded (by replacing BY proofs with additional levels of proof) to a primitive proof.

8.5 The Problem

Our problem is that rule I2 is not *a priori* sound for temporal logic. As an example, consider the following two valid proof rules.

$$
\begin{array}{llll}
\textsc{theorem}\ \mathit{AddBox} & \triangleq & \textsc{assume} & \textsc{new state}\ P, \\
 & & & P \\
 & & \textsc{prove} & \Box P
\end{array}
$$

$$
\begin{array}{llll}
\textsc{theorem}\ \mathit{RemoveBoxPrime} & \triangleq & \textsc{assume} & \textsc{new state}\ P, \\
 & & & \Box P \\
 & & \textsc{prove} & P'
\end{array}
$$

Here's how we can use them to prove an incorrect theorem.

THEOREM $(x = 1) \Rightarrow (x = 1)'$
$\langle 1\rangle 1.$ ASSUME $x = 1$ PROVE $(x = 1)'$
  $\langle 2\rangle 1.$ $\Box(x = 1)$
    BY $x = 1$, *AddBox*
  $\langle 2\rangle 2.$ QED
    BY $\langle 2\rangle 1$, *RemoveBoxPrime*
$\langle 1\rangle 2.$ QED
  BY $\langle 1\rangle 1$, *DeductionPrinciple*

The problem with this proof lies in the proof of $\langle 2\rangle 1$. We are using rule I1 and *AddBox* to deduce

$\langle 3\rangle 1.$ ASSUME $x = 1$ PROVE $\Box(x = 1)$

and then using I2 to deduce $\Box(x = 1)$ from $x = 1$ and $\langle 3\rangle 1$. However, for this to be sound, we need to be able to deduce $[\![\Box(x = 1)]\!]$ from $[\![x = 1]\!]$ and

$[\![\langle 3\rangle 1]\!]$. However, we have

$$
\begin{aligned}
[\![\Box(x = 1)]\!] &= \forall\,\beta : [\Box(x = 1)]_\beta \\
[\![x = 1]\!] &= [x = 1]_\beta \\
[\![\langle 3\rangle 1]\!] &= (\forall\,\beta : [x = 1]_\beta) \Rightarrow (\forall\,\beta : [\Box(x = 1)]_\beta)
\end{aligned}
$$

so $[\![x = 1]\!]$ and $[\![\langle 3\rangle 1]\!]$ do not imply $[\![\Box(x = 1)]\!]$. For the deduction to be valid, we need to be able to use $\forall\, b : [\![x = 1]\!]$, not just $[\![x = 1]\!]$.

8.6 The Solution

Recall that we defined a β-assumption in Section 8.3 essentially to be a non-temporal assumption that occurs in an *assume/prove* having a non-temporal goal. Each known fact comes from a particular assertion or assumption. A known fact is a β-assumption if it comes from a β-assumption of an *assume/prove* assertion.

A proof of an assertion $\mathcal{A}$ is valid if it proves the validity of $[\![\mathcal{A}]\!]$. Our fundamental result is:

Proposition 1 *A primitive proof is valid if no* assume/prove *fact having a temporal goal appears in a* BY *proof if the set of (all) known facts contains a β-assumption.*

Proof: We define a semantic proof that corresponds to the TLA$^+$ proof. We replace each assertion (including that of the theorem) $\mathcal{A}$ with $[\![\mathcal{A}]\!]$. However, if $\mathcal{A}$ is an *assume/prove*, so $[\![\mathcal{A}]\!]$ has the form $F_1 \land \ldots \land F_n \Rightarrow H$, then we prove it as if it were the corresponding assertion

ASSUME $F_1, \ldots, F_n$ PROVE H

In other words, we implicitly add as the first step of the proof the step

$\langle + \rangle$ SUFFICES ASSUME $F_1, \ldots, F_n$ PROVE H

The assertions of the semantic proof are formulas of ordinary math, so ordinary mathematical reasoning is valid. We must show that the semantic proof is a valid proof of ordinary math. We do this by defining the set of known facts at each point of the semantic proof and show that each BY step of the original proof can be transformed into a valid leaf proof of the semantic proof.

The set of known facts at each point in the semantic proof is the same as for a TLA$^+$ proof, with the following exception: If β does not occur free in any of the known facts under which an assertion $\mathcal{A}$ is proved (that is, the set of known facts right before the assertion) and β occurs free in $[\![\mathcal{A}]\!]$, then we add $\forall \beta : [\![\mathcal{A}]\!]$ instead of $[\![\mathcal{A}]\!]$ to the set of known facts. This is clearly valid, since if we prove a fact about β using assumptions that don't mention β, then the fact must be true for all β. To each TLA$^+$ rule that appears in the initial set of facts, we also include the corresponding semantic fact, universally quantified by β if β occurs free in it. (Any such fact is obviously true for all β.) To every known fact at a point in the TLA$^+$ proof, we have defined a corresponding known fact at the corresponding point of the semantic proof.

Since the TLA$^+$ proof is primitive, each leaf step is justified by a proof using only known facts and inference rules I1 and I2. To every application of I1 to a known fact of the TLA$^+$ proof, there is a corresponding application of I1 to the corresponding fact of the semantic proof. (A free identifier *Id* appears inside the semantic proof in expressions of the form $[Id]_\beta$.) For each application of I2 of the TLA$^+$ proof, there is a corresponding deduction of the form:

I2s. Deducing a new fact H from the known facts $F_1, \ldots, F_n$, and $G_1 \land \ldots \land G_n \Rightarrow H$

where each F_i is the same as G_i except that one but not the other may contain an outer $\forall \beta$. The deduction is sound except if some G_i is universally quantified by β, but F_i has β as a free variable. By the definition of the meaning of an *assume/prove*, this can happen only if the fact $G_1 \land \ldots \land G_n \Rightarrow H$ comes from a proof rule having a temporal goal. Moreover, it can only happen if the set of known facts contains a fact having β as a free variable. However, our rule for constructing the known facts of the semantic proof implies that the set of known facts at any point can contain one having β as a free variable only if it contains a β-assumption. (The "first" such fact that gets added to the set of known facts can only be a β-assumption.) Our hypothesis applies that we cannot use a proof rule having a temporal goal in a BY step that occurs when the set of known facts contains a β-assumption. Therefore, every deduction I2s required to justify a step in the semantic proof is valid. $\square$

We can't prove a temporal formula without using proof rules having temporal goals. The hypothesis of our proposition effectively means that we can't prove any temporal steps when the current set of known facts includes a β-assumption. Remember that the set of known facts includes a β-assumption only if we are essentially inside the proof of an *assume/prove* having a non-temporal goal. (This can occur because the current goal has been replaced by a non-temporal goal by a SUFFICES *assume/prove* step.)

We don't write primitive proofs. The proof manager must check that, if a BY step were expanded to make it primitive, then the hypothesis of the proposition is satisfied. This effectively means that it must ensure that no proof rule with a temporal goal is used in a BY step occurring within the scope of a β-assumption.

There is no restriction on where we can use known facts that are temporal formulas or proof rules with temporal assumptions. In particular, we are always free to use the following valid proof rule:

$$
\begin{array}{llll}
\text{THEOREM } \mathit{Unbox} & \triangleq & \text{ASSUME} & \text{NEW STATE } P, \\
 & & & \Box P \\
 & & \text{PROVE} & P \land P'
\end{array}
$$