

TLA⁺²
A Preliminary Guide

Leslie Lamport

11 April 2008

Contents

1	Introduction	1
2	Recursive Operator Definitions	1
3	Lambda Expressions	3
4	Theorems and Assumptions	3
4.1	Naming	3
4.2	Assume/Prove	4
5	Instantiation	6
5.1	Instantiating *fix Operators	6
5.2	Leibniz Operators and Instantiation	7
6	Naming Subexpressions	8
6.1	Labels and Labeled Subexpression Names	9
6.2	Positional Subexpression Names	11
6.3	Subexpressions of LET Definitions	14
6.4	Subexpressions of an <i>Assume/Prove</i>	15
6.5	Using Subexpression Names as Operators	15
7	The Proof Syntax	15
7.1	The structure of a proof	15
7.2	Use, Hide, and By	18
7.2.1	Use and Hide	18
7.2.2	By	20
7.2.3	Obvious and Omitted	20
7.3	The Current State	21
7.4	Steps That Take Proofs	22
7.4.1	Formulas and Assume/Proves	22
7.4.2	Case	22
7.4.3	@ Steps	22
7.4.4	Suffices	23
7.4.5	Pick	24
7.4.6	QED	24
7.5	Steps That Do Not Take Proofs	24
7.5.1	Definitions	24
7.5.2	Instance	25
7.5.3	Use and Hide	25

7.5.4	Have	25
7.5.5	Take	25
7.5.6	Witness	26
7.6	Referring to Steps and Their Parts	26
7.6.1	Naming Subexpressions	27
7.6.2	Naming Facts	28
7.6.3	Naming Definitions	28

1 Introduction

TLA⁺² is an enhanced version of TLA⁺. The currently released version of TLA⁺ is here called TLA⁺¹. Most of the additions to the language in TLA⁺² are for writing proofs. 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⁺¹ specifications are legal TLA⁺² specifications. Two rather arcane changes have been made to instantiation; they are explained in Section 5. The only other change that affects TLA⁺¹ specifications is that the following new keywords have been added in TLA⁺², and thus cannot be used as identifiers.

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

2 Recursive Operator Definitions

The only recursive definitions allowed in TLA⁺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⁺¹ because I didn't know how to assign a sensible meaning—for example, what should be the meaning of

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

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,

```

RECURSIVE fact(_)
fact(n)  $\triangleq$  IF n = 0 THEN 1 ELSE n * fact(n - 1)

```

defines $fact(n)$ to equal $n!$ if n is a natural number. I have no idea what it defines $fact(-2)$ or $fact(\text{"abc"})$ 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

```

RECURSIVE Ins(_)
Ins(n)  $\triangleq$  INSTANCE M WITH ...

```

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

$$F(x) \triangleq \dots$$

then $F(exp)$ 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 $fact(-2)$ equals

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

It can be proved that

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

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

$$F(LAMBDA a, b : a + 2 * b)$$

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.

INSTANCE M WITH $Op \leftarrow Id$

INSTANCE M WITH $Op \leftarrow LAMBDA a, b : a + 2 * b$

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

$$\text{THEOREM } Fermat \triangleq \neg \exists n \in Nat \setminus (0 .. 2) : \dots$$

This is equivalent to

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

$$\begin{array}{l} \text{THEOREM } \textit{Constancy} \triangleq \\ \text{ASSUME } \text{CONSTANT } C \\ \text{PROVE } C' = C \end{array}$$

Here is the standard TLA rule for proving invariance.

$$\begin{array}{l} \text{THEOREM } \text{ASSUME } \text{STATE } \textit{Inv}, \\ \text{ACTION } A, \text{ STATE } v, \\ \textit{Inv} \wedge [A]_v \Rightarrow \textit{Inv}' \\ \text{PROVE } \textit{Inv} \wedge \Box[A]_v \Rightarrow \Box \textit{Inv} \end{array}$$

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.

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

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 $\text{ENABLED } z' \neq z'$, 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 `∈`, 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

$$Foo \triangleq \text{INSTANCE } M \text{ WITH } \dots$$

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

$$1 \text{ } Foo!&& \text{ } 2$$

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

$$a \ \&\& \ b \ \triangleq \ \dots$$

Because of a bug that is unlikely to be fixed, the new 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 <i>M</i> CONSTANTS <i>C</i> , <i>D</i> , <i>F</i> (-) THEOREM (<i>C</i> = <i>D</i>) ⇒ (<i>F</i> (<i>C</i>) = <i>F</i> (<i>D</i>))

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* ← *x*, *D* ← *y*, *F* ← *Prime*

This imports the theorem from module *M* as

THEOREM (*x* = *y*) ⇒ (*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*. For an operator *F* that takes *k* arguments, *F* is *Leibniz* iff the value of *F*(*e*₁, ..., *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

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

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⁺² provides a method of naming subexpressions of a definition. If F is defined by $F(a, b) \triangleq \dots$, 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

$$\textit{label} :: \textit{expression}$$

(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,

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

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

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

is illegal because it would be parsed as $a * (\textit{lab} :: (b + c))$ 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.

$$\begin{aligned} F(a) \triangleq & \forall b : l1(b) :: (a > 0) \Rightarrow \\ & \wedge \dots \\ & \wedge l2 :: \exists c : \wedge \dots \\ & \wedge \exists d : l3(c, d) :: a - b > c - d \end{aligned}$$

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 $\&\&(A, B)! \dots$.

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

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

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 $f(e_1, \dots, e_k)$ denotes f when $k = 0$. A label lab has the form $id(p_1, \dots, p_k)$ where id and the p_i are identifiers, the p_i are all distinct, and $\{p_1, \dots, p_k\}$ 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 lab_c , then the expression that introduces p_i lies within the expression labeled by lab_c .

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 $prefix!labexp_1! \dots !labexp_n$, where $prefix$ has the form $Op(e_1, \dots, e_{k[0]})$, each $labexp_i$ has the form $id_i(e_1, \dots, e_{k[i]})$, Op and the id_i are identifiers, and the e_j are expressions. It must satisfy:

- The definition

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

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 $Id(e_1, \dots, e_k)! \lambda$ where there is a statement

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

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

$$\begin{aligned} F(a) &\triangleq \wedge \dots \\ &\quad \wedge \dots \\ &\quad \wedge Len(x[a]) > 0 \\ &\quad \wedge \dots \end{aligned}$$

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 $F(A)!3!\langle!\rangle$ or $F(A)!3!\langle!1!\rangle$ or $F(A)!3!1!\langle!2$ or \dots . As usual, “ \langle ” is typed “ $\langle\langle$ ” and “ \rangle ” is typed “ $\rangle\rangle$ ”.

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.
- $r.fld$ 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 $[fld_1 \mapsto val_1, \dots, fld_n \mapsto val_n]$ and $[fld_1 : val_1, \dots, fld_n : val_n]$ the selector $!i$ names the subexpression val_i for $i \in 1 \dots n$. 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 $\text{CASE } p_1 \rightarrow e_1 \square \dots \square p_n \rightarrow e_n$ 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 $\text{WF}_e(A)$ and $\text{SF}_e(a)$ 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 $\text{LET } \dots \text{ IN } e$ 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

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

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:

- $!(e_1, \dots, e_n)$ 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

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

then “ $Op(P_1, \dots, P_k)!$ ” names exp with P_i substituted for p_i , for each i in $1 \dots k$.

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

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

$F(7)!lab!$ 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 $\nu!Op(e_1, \dots, e_n)$ is the name of the expression $Op(e_1, \dots, e_n)$ interpreted in the context determined by ν . For example, in

$$\begin{aligned} F(a) &\triangleq \wedge \dots \\ &\wedge \text{LET } G(b) \triangleq a + b \\ &\text{IN } \dots \end{aligned}$$

$F(A)!2!G(B)$ 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, $F(A)!2!G(B)!$ names B . The naming process can be continued all the way down, naming subexpressions of *let* definitions contained within *let* definitions contained within \dots .

If the *let/in* expression is labeled, then it can be named by a labeled subexpression name λ . In that case, $\lambda!Op(e_1, \dots, e_n)$ is a labeled subexpression name that names a subexpression of the *in* clause with label $Op(p_1, \dots, p_n)$. To refer to the operator Op defined in the *let* clause, just

add a “!.” to the end of λ , writing $\lambda! :! Op(e_1, \dots, e_n)$. In particular, if H is defined to equal the *let/in* expression, then we write $H! :! Op(e_1, \dots, e_n)$, even if H is not recursively defined.

6.4 Subexpressions of an *Assume/Prove*

If we have

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

then Id is not an expression and cannot be used as one. Moreover, it is impossible to name subexpressions of the *assume/prove* from outside the proof of the theorem. (The reason for this restriction is that subexpressions can contain symbols declared in the assumptions, and the scope of those declarations does not extend outside the theorem and its proof.) Section 7.6 explains how subexpressions of the *assume/prove* are named inside the theorem’s proof.

6.5 Using Subexpression Names as Operators

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

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

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

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

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

7 The Proof Syntax

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⟩1. ...
  PROOF
    ⟨2⟩4a. ...
      OBVIOUS
    ⟨2⟩.. ...
      ⟨17⟩ ...
        PROOF OMITTED
      ⟨17⟩1. ...
      ⟨17⟩ ...
      ⟨17⟩ab QED
    ⟨2⟩11 QED
      BY ...
⟨1⟩2. ...
  BY ...
⟨1⟩3. QED

```

Figure 1: The structure of a simple proof.

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 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

- \langle (printed as “ \langle ”)
- a number called the step’s *level number* or a + or * character. (The

meaning of $+$ and $*$ is explained below.)

- $\langle \rangle$ (printed as “ $\langle \rangle$ ”)
- 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.
- an optional sequence of periods.

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 “ \langle ” and “ \rangle ” rather than “ $\langle\langle$ ” and “ $\rangle\rangle$ ”.) 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 “ $\langle 2 \rangle \dots$ ” tokens could be replaced by “ $\langle * \rangle \dots$ ”. 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.

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*. 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. 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. We call these the sets of *current declarations*, *current definitions*, and *known facts*.

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 a proof tool will expand and the facts that it will try to apply when trying to prove something. Here are the default values of these subsets. (These default values should be viewed as an initial proposal, subject to change.)

- All definitions are usable.
- A theorem or assumption is usable iff it is unnamed. (Section 4.1 explains how theorems are named.)

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:

- An expression. The expression is usually the name of a theorem or assumption. For a *use* statement, it could in principle be any fact easily derivable from the set of known facts—for example, it could be the fact $a + b = b + a$ if the set of known facts includes the commutativity rule of arithmetic. Most of the time, it will be the name of a theorem or assumption or the name of a subexpression of a theorem or assumption. For example, if *NumFact* is the name of a theorem of the form $\forall n \in \text{Nat} : \dots$, then one might use *NumFact!*(42) as a fact. For a *hide* statement, the expression is usually the name of a theorem or assumption, or perhaps a fact that appeared in a previous *use* statement
- `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

$$Id \triangleq \text{INSTANCE } M \dots$$

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 $Id(x) \triangleq \dots$

It is expected that a proof tool would have some special standard modules whose theorems would invoke decision procedures or proof tactics. For example, there might be an *Arithmetic* module such that the statement `USE MODULE Arithmetic` would cause the tool to apply certain decision procedure for arithmetic when trying to prove something.

A definition specifier is one of the following:

- The name of a defined operator. For example,
 - *F* if the module contains the definition $F(x, y) \triangleq \dots$
 - *Ins!F* if the current module contains $Ins(a) \triangleq \text{INSTANCE } M \dots$ and *F* is defined in *M*.
 - *F!@!Bar* if the current module contains

$$F(x, y) \triangleq \forall z \in S : \text{LET } Bar(w) \triangleq \dots \text{ IN } \dots$$

(See Section 6.5.)

- `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

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

(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* or *hide* 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 set of goals. A *by* proof asserts that one of those goals 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.

Consider the proof in Figure 1, and suppose that all the steps simply assert formulas. There are three goals at statement $\langle 17 \rangle 1$: the statement of the theorem of which this is a proof, the level-1 goal asserted by step $\langle 1 \rangle 1$, and the level-2 goal asserted by the unnamed level-2 step whose proof contains $\langle 17 \rangle 1$. We normally think of the latter level-2 formula as *the* goal of the level-17 proof. However, instead of proving the level-2 goal, the proof could prove the level-1 goal instead. Of course, if the level-17 proof actually proved $\langle 1 \rangle 1$, then there would be no need for the unnamed level-2 step or for the *qed* step $\langle 2 \rangle 11$. However, what sometimes happens is that the level $\langle 17 \rangle$ proof proves the level-2 goal in one case and the level-1 goal in another case. If $\langle 1 \rangle 1$ asserted a formula *F*, then this situation would be described in an informal prose proof by:

In this case, *F* would be true and we would be done.

7.2.3 Obvious and Omitted

The terminal proof `OBVIOUS` asserts that one of the current goals 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, rather than having omitted it accidentally.

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. A proof tool will attempt to check a step only if it 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 set of goals, which are formulas. If this set is nonempty, one of the goals is designated the *current* goal.

A theorem starts with the sets of current declarations, current and usable definitions, and facts and usable facts described above, and with an empty set of goals. 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 is added to the (empty) set of goals and 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/proves* asserted in the assumptions is added to the set of known facts and is also added to the set of usable facts iff the theorem has no name, and 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 $\text{NEW } C \in S$ is an abbreviation for the declaration $\text{NEW } C$ and the assertion $C \in S$.

After the theorem’s proof (if any), the current state reverts to the state right before the theorem with the theorem’s assertion added to the set of known facts and to the set of usable facts iff the theorem is unnamed.

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, adding either the formula or the *prove* assertion to the set of goals as the current goal. The formulas and *assume/proves* from the *assume* clause are added to the set of usable facts iff the step is unnamed. 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

$$\sigma \text{ ASSUME } F \text{ PROVE } G$$

where G is the current goal. (Since G is already the current goal, the set of goals and the current goal remain 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).

⟨2⟩3. $A \leq D$
 ⟨3⟩1. $A \leq B$
 ⟨3⟩2. $B \leq C$
 ⟨3⟩3. $C \leq D$
 ⟨3⟩4. QED
 BY ⟨3⟩1, ⟨3⟩2, ⟨3⟩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.

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

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

$A = B = C = D$
 $A \Rightarrow B \Rightarrow C \Rightarrow D$
 $A \subseteq B \subseteq C \subseteq D$
 $A \leq B < C \leq D$

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

⟨3⟩4. $A \leq B$
 ⟨3⟩5. @ $> C$

where the @ stands for B .

7.4.4 Suffices

The step σ SUFFICES A asserts that proving A proves one of the goals in the state (at any level), 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 is added to the set of usable facts iff the step is unnamed. (The proof must prove an existing goal.) After the step and its proof:

- If A is a formula, then it is added to the set of goals as the current goal.
- If A is an *assume/prove*, then the declarations in its assumptions are added to the set of current declarations, the assertions among its assumptions (the formulas and *assume/proves*) are added to the set of known facts and are also added to the set of usable facts iff the step is unnamed, and the *prove* formula is added to the set of goals as 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

$$\sigma \text{ PICK } x \in S, y \in T : P(x, y)$$

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 at the top of the stack, and the facts asserted by the *pick* are added to the set of facts and, if the step is unnamed, to the set of usable facts. In this example, the facts asserted by the statement are the three formulas $x \in S$, $y \in T$, and $P(x, y)$.

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

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

It has the same effect on the state as the corresponding (top-level) statements. The definitions 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.

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 should be obviously equivalent to $F \Rightarrow G$ for some formula G . In that case, the state after the step is obtained by adding G to the set of goals, making it the current goal, adding F to the set of known facts, and adding it to the set of usable facts iff the step is unnamed. Thus, this step means that we are going to prove the current goal by assuming F and proving G .

It's up to a proof tool to determine what “obviously equivalent” means. We would expect that $G \vee \neg F$ is obviously equivalent to $F \Rightarrow G$, but that $\forall x : F \Rightarrow G(x)$ is probably not obviously equivalent to $F \Rightarrow \forall x : G(x)$ (assuming x does not occur free in F , so the latter formula is syntactically correct).

7.5.5 Take

A *take* step consists of a step-starting token followed by `TAKE` followed by anything that could come between “ \forall ” and its matching “ $:$ ”—for example

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

This step is typically used when the current goal is

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

for some formula G . It means that we are going to prove this goal by declaring x, y, z to be constants, assuming $x \in S$ and $\langle y, z \rangle \in T$, and proving G .

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 added to the set of goals as 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 $id \in e$ or $\langle id_1, \dots, id_k \rangle \in e$ in τ are added to the set of known facts; they are also added to the set of used facts iff the step is unnamed.

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, \dots, e_k is correct:

- The current goal is obviously equivalent to a formula $\exists id_1, \dots, id_k : G$. In this case, the formula obtained by substituting each e_j for id_j in G , for j in $1 \dots k$, is added to the set of goals and made the current goal.
- The current goal is obviously equivalent to a formula $\exists \iota_1 \in S_1, \dots, \iota_k \in S_k : G$ where each ι_j is either an identifier or a tuple of identifiers, there is some substitution of expressions for these identifiers that transforms each ι_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 added to the set of goals and made the current goal, and the facts e_j are added to the set of known facts and to the set of usable facts if the step is not named. (Adding a fact that is easily provable to the set of usable facts might make additional facts easily provable from that set.)

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.)

For naming purposes, a *suffices* step that asserts a formula is treated the same way as a step that simply asserts the formula. For example,

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

defines $\langle 2 \rangle 3$ to equal $x + y = z$ and $\langle 2 \rangle 3! \langle$ to equal $x + y$.

Assume/Proof Steps The parts of an *assume/proof* step or a *suffices assume/proof* step can be named either with labels or positionally. In the step

$$\begin{array}{l} \sigma \text{ ASSUME } A_1, \dots, A_k, \\ \text{PROVE } G \end{array}$$

$\sigma!i$ names A_i , for $1 \leq i \leq k$, and $\sigma!k+1$ names the formula G . Labeled subexpressions are referred to as expected with names beginning “ $\sigma!$ ”. Thus, in

$$\begin{array}{l} \langle 3 \rangle 4. \text{ ASSUME } P, \text{ ASSUME } Q \\ \text{PROVE } R \\ \text{PROVE } S \end{array}$$

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

Subexpressions of an ordinary *assume/proof* step can be named only inside the proof of that step, not after the step. (The scope of symbol declarations in the assumptions ends with the end of the proof.) However, for a *suffices assume/proof* step, the opposite holds. Subexpressions of such a step can be named only after the step’s proof, where the scope of any symbols declared in the assumptions begins.

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

$\langle 2 \rangle 3$. CASE $x + y > 0$

$\langle 2 \rangle 4$. 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 $x + 1$.

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

$\langle 3 \rangle 4$. 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’s syntactically legal to write a *use* step that uses the thing we are trying to prove. Of course, this would not be a proper proof. (If what we are trying to prove is easily derived from the usable facts, then we should simply write PROOF OBVIOUS.) It is legal 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. If σ asserts a formula, then ordinary logic allows us to use $\neg\sigma$ within its proof. (This means that the proof is by contradiction.)

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.