Math

?

←

→

C

I

S

?
←
→
C
I
S

13 Arithmetic and Logic

13.1 Arithmetic

Ordinary numbers (such as 3421) and decimal fractions (such as 3.14) are built-in primitive TLA$^+$ symbols. The standard arithmetic operations on integers are defined in the standard *Integers* module:

? ← → C I S

$+$	Addition.
$-$	Subtraction and unary minus.
$*$	Multiplication.
$\hat{}$	Exponentiation, where a^b is typed `a^b`.

The module defines the usual inequality relations:

$<$	(typed `<`)	$\leq$	(typed `\leq` or `=<`)
$>$	(typed `>`)	$\geq$	(typed `\geq` or `>=`)

The *Integers* module also defines

Int	The set of all integers.
Nat	The set of all natural numbers (non-negative integers).
$\,..\,$	$m \,..\, n$ is the set of integers from m through n. More precisely:

$$m \,..\, n \;\triangleq\; \{i \in \mathit{Int} : (m \leq i) \land (i \leq n)\}$$

The ordinary division operation $/$ is not defined in the *Integers* module, since a/b need not be an integer. Instead, the module defines the operators

$\div$	Integer division, where $n \div d$ is the integer part of n/d.
$\%$	Modulus, where $n \,\%\, d$ is the remainder when n is divided by d.

More precisely, these two operators are defined so that the following two conditions hold for any integer n and positive integer d:

$$n \,\%\, d \in 0 \,..\, (d-1) \qquad\qquad n = d * (n \div d) + (n \,\%\, d)$$

All these operators except for $\div$, $\%$, and $\,..\,$ are defined so they have their usual meaning on real numbers—for example so that $.81^{.5} * 2 + .14$ equals 1.94. The *Reals* module extends the *Integers* module and also defines $/$ (real division) and the set *Real* of all real numbers. However, TLC can handle neither $/$ nor numbers that are not integers. There is also a *Naturals* module that is the same as the *Integers* module except it does not define unary minus.

Mathematically sophisticated readers who are curious about how the operators of arithmetic are defined in TLA$^+$ can find out in Section 18.4 (page 344) of *Specifying Systems*. (See the errata for a correction to the definitions.)

13.2 Mathematical Logic

In the beginning, numbers were modifiers. People talked about two goats or three pigs. Mathematics was born when numbers like *two* and *three* became nouns. Operations like addition combine nouns to form a noun—for example:

$$\underset{\text{noun}}{\boxed{\underset{\text{noun}}{\boxed{1}} \quad + \quad \underset{\text{noun}}{\boxed{1}}}}$$

?
←
→
C
I
S

In the beginning, equations were sentences and *equals* was a verb:

$$\underset{\text{subject}}{\boxed{1+1}} \quad \underset{\text{verb}}{=} \quad \underset{\text{object}}{\boxed{2}}$$

In mathematical logic, equations are nouns and *equals* is an operation, just like addition:

$$\underset{\text{noun}}{\boxed{\underset{\text{noun}}{\boxed{1+1}} \quad = \quad \underset{\text{noun}}{\boxed{2}}}}$$

Just as the noun $1+1$ names the number 2, the noun $1+1=2$ names the value TRUE. The noun $1+1=3$ names the value FALSE.

The values TRUE and FALSE are called *truth values* or *Booleans.* Truth values are to mathematical logic what numbers are to arithmetic. Mathematical logic is simpler than arithmetic because there are just two truth values, while there are infinitely many numbers.

The sentence 1+1 *equals* 2 asserts the fact that the formula (noun) $1+1=2$ is equal to TRUE. Sometimes mathematicians assert this fact by writing

$$\vdash \ 1+1=2$$

However, few mathematicians bother doing this; instead they write both the formula and the fact as $1+1=2$. You have to figure out by context which they mean.

Many people think of all mathematical expressions as sentences, reading 1+1 as *Add* 1 *and* 1. They think of $1+1=2$ as asserting

> Adding 1 and 1 produces 2.

This kind of operational thinking is a mental straight jacket that limits your ability to use mathematics. It doesn't allow you to make proper sense of formulas like

$$1+1/2+1/4+1/8+\cdots \ = \ 2$$

Is this really a formula?

Viewed operationally, this formula is an assertion about the result of performing infinitely many operations. Trying to reason about what happens when you perform an infinite number of operations can get you in trouble.

13.3 Propositional Logic

Propositional logic is the study of simple operations on Booleans (truth values), which are analogous to operations like addition and subtraction on numbers. Because there are only two Booleans, TRUE and FALSE, the operators of propositional logic are much simpler than their arithmetic cousins. We will use five of them.

?
←
→
C
I
S

13.3.1 $\land$ and $\lor$

The first two propositional logic operators we need are $\land$ (typed `/\`), called *conjunction* or *and*, and $\lor$ (typed `\/`), called *disjunction* or *or*. The names *and* and *or* describe these operators fairly well. For example, $F \land G$ equals TRUE iff both F and G are true. Thus, TRUE $\land$ FALSE equals FALSE. You can use TLC to check this. Open a brand new spec in the Toolbox with a new module named *Calculate*, using the default spec name *Calculate*. Create a new model, giving it any name. The Toolbox will show the model's Model Checking Results page. You can then type an expression for TLC to evaluate in the Evaluate Constant Expression section and run TLC to evaluate it. Start by entering

TRUE $\land$ FALSE ASCII version

(Remember that you can click on the link in the right-hand margin to get a popup from which you can copy this expression.) Running TLC shows that this expression's value is, indeed, FALSE. In this way, we could check—one at a time—the values of $F \land G$ for all four choices of the Boolean values F and G. But let's do them all at once by entering the following expression.

$\langle$TRUE $\land$ TRUE, TRUE $\land$ FALSE, FALSE $\land$ TRUE, FALSE $\land$ FALSE$\rangle$ ASCII version

Angle brackets $\langle$ and $\rangle$ enclose a tuple. This expression is a 4-tuple whose components are the four possible expressions of the form $F \land G$ when F and G are Booleans.

Next, change the four instances of $\land$ in this expression to $\lor$. Observe that TRUE $\lor$ TRUE equals TRUE. A better English name for $\lor$ would be *and/or*, since in ordinary speech we generally take "A or B will happen" to exclude the possibility that both A and B occur. However, we usually read $\lor$ as *or* because that's easier to say than *and/or*.

If this has been a detour, you may now want to return to Section 2.2 of the Starting Track.

13.3.2 Other Propositional Operators

In addition to $\land$ and $\lor$, we use these three Boolean operators:

$\Rightarrow$ Implication. Typed `=>` and read as *implies*

- $\equiv$ Equivalence. Also written $\Leftrightarrow$. Typed as `\equiv` or `<=>` and read as *is equivalent to.*

- $\neg$ Negation. Typed `~` and read as *not.*

The first two are binary (infix) operators; negation is a unary (prefix) operator.

You can use the Toolbox's Evaluate Constant Expression feature to let TLC tell you what the definitions of these operators are, just as you did for $\land$ and $\lor$. For negation, you will find that $\neg$TRUE equals FALSE and $\neg$FALSE equals TRUE. For equivalence, you will find that $F \equiv G$ equals true iff F equals G, for any Boolean values F and G.

Why write $\equiv$ instead of $=$?

You may be surprised by the definition of $\Rightarrow$ (implies). TLC reveals that FALSE $\Rightarrow G$ equals TRUE for both Boolean values of G. Of the four possibilities, only TRUE $\Rightarrow$ FALSE equals FALSE. To understand why $\Rightarrow$ means implication, consider the statement:

$$n > 5 \text{ implies } n > 3$$

This statement is true for all integers n, so the formula $(n > 5) \Rightarrow (n > 3)$ should equal TRUE for all integers n. Substituting 1 for n, the formula becomes $(1 > 5) \Rightarrow (1 > 3)$. Since $(1 > 5)$ and $(1 > 3)$ both equal FALSE, this latter formula equals FALSE $\Rightarrow$ FALSE. Hence, FALSE $\Rightarrow$ FALSE should equal TRUE. Substituting other values for n will convince you that the definition of $\Rightarrow$ is the right one.

If this has been a detour, you may now want to return to Section 2.7 of the Starting Track.

13.4 Predicate Logic

Predicate logic extends propositional logic with two operators called *quantifiers.* The first is the *universal* quantifier $\forall$, typed `\A` and read *for all.* The formula $\forall x \in S : P(x)$ is essentially the conjunction ($\land$) of the formulas $P(x)$ for all x in S. For example, the formula $\forall i \in \{1, 2, 3\} : i^2 > i$ equals the formula

$$(1^2 > 1) \land (2^2 > 2) \land (3^2 > 3)$$

which equals FALSE. In general, $\forall x \in S : P(x)$ is true iff $P(x)$ is true for all elements x in S. For example, $\forall \mathit{num} \in \mathit{Nat} : \mathit{num} + 1 > \mathit{num}$ is the (true) formula which asserts that $i + 1 > i$ is true for all natural numbers i.

The second quantifier of predicate logic is the *existential* quantifier $\exists$, typed `\E` and read *there exists.* It is the analog of $\forall$ for disjunction ($\lor$). Thus, $\exists i \in \{1, 2, 3\} : i^2 > i$ equals the formula

$$(1^2 > 1) \lor (2^2 > 2) \lor (3^2 > 3)$$

?
←
→
C
I
S

which equals TRUE. In general, $\exists\, x \in S : P(x)$ is true iff $P(x)$ is true for at least one element x in S. For example, $\exists\, id \in Int : id^2 = 9$ is the (true) formula which asserts that there is some integer n such that n^2 equals 9.

These two quantifiers are related by the following two tautologies. You should make sure that you understand why they are true.

?
←
→
C
I
S

$$\begin{aligned} \neg(\forall\, x \in S : P(x)) &\equiv \exists\, x \in S : \neg P(x) \\ \neg(\exists\, x \in S : P(x)) &\equiv \forall\, x \in S : \neg P(x) \end{aligned}$$

These two quantifiers $\forall$ and $\exists$ are said to be *bounded* because they are essentially conjunction and disjunction over formulas in some set S. Mathematicians also use unbounded versions of these operators, writing $\forall\, x : P(x)$ and $\exists\, x : P(x)$. These formulas assert that $P(x)$ is true for all ($\forall$) or for some ($\exists$) values x. Make sure that you understand why bounded and unbounded quantification are related by the following tautologies.

$$\begin{aligned} (\forall\, x \in S : P(x)) &\equiv (\forall\, x : (x \in S) \Rightarrow P(x)) \\ (\exists\, x \in S : P(x)) &\equiv (\exists\, x : (x \in S) \land P(x)) \end{aligned}$$

When writing specifications or algorithms, you will never have to use unbounded quantification. Unbounded quantification is needed only to state some mathematical laws, such as $\forall\, x : x = x$. TLC can handle only quantification over a finite set. Therefore, it cannot evaluate $\exists\, n \in Int : n^2 = 9$ or $\forall\, x : x = x$.

TLA$^+$ allows some abbreviations for nested quantification. For example:

$$\begin{aligned} &\forall\, x \in S, y \in T : P(x, y) \quad \text{means} \quad \forall\, x \in S : \forall\, y \in T : P(x, y) \\ &\exists\, x, y, z \in S : P(x, y, z) \quad \text{means} \quad \exists\, x \in S : \exists\, y \in S : \exists\, z \in S : P(x, y, z) \end{aligned}$$

There are two things about the syntax of quantifiers that may be surprising. The first is illustrated by the following syntactically incorrect definition:

Should it be $\forall\, x : P$ or $\forall\, x : P(x)$?

$$Foo \triangleq \forall\, x \in S : P \land \forall\, x \in T : Q$$

The TLA$^+$ parser will complain that the second x is a multiply-defined symbol. This is because the definition is parsed as

$$Foo \triangleq \forall\, x \in S : (P \land \forall\, x \in T : Q)$$

TLA$^+$ does not allow any symbol to be given a meaning if it already has one. The first $\forall$ assigns a meaning to x within a scope that includes the second $\forall$. Thus, the second $\forall$ assigns a meaning to x where x already has a meaning.

A quantifier is treated like a prefix operator with the lowest possible precedence, so the scope of its bound identifier extends as far as it "reasonably" can. The following are two correct versions of this definition.

$$Foo \triangleq \begin{array}{l} \land\ \forall\, x \in S : P \\ \land\ \forall\, x \in T : Q \end{array} \qquad\qquad Foo \triangleq (\forall\, x \in S : P) \land (\forall\, x \in T : Q)$$

The second possibly surprising aspect of quantification is that in the formula $\forall x \in S : P$, the expression S does not lie within the scope of the bound identifier x. Thus, S may not contain the symbol x.

If this has been a detour, you may now want to return to Section 4.1 of the Starting Track.

? ← → C I S

13.5 The CHOOSE Operator

The TLA$^+$ CHOOSE operator is closely related to the existential quantifier $\exists$. The formula $\exists x \in S : P(x)$ asserts that there is a value x for which $P(x)$ is true. If that assertion is true, then $\text{CHOOSE } x \in S : P(x)$ equals such a value. More precisely, CHOOSE is specified by the following axiom.

The fine print.

$$\textbf{Choose Axiom} \quad (\exists x \in S : P(x)) \Rightarrow \begin{array}{l} \land\ (\text{CHOOSE } x \in S : P(x)) \in S \\ \land\ P(\text{CHOOSE } x \in S : P(x)) \end{array}$$

The most common use for the CHOOSE operator is to select a unique value that is specified by the property it satisfies. For example, in Section 4.1.2□ we define the maximum of a set of numbers by using CHOOSE to select the largest element of the set. TLC can evaluate $\text{CHOOSE } x \in S : P(x)$ only if S is a finite set.

Computer scientists often think that CHOOSE must be nondeterministic. In mathematics, there is no such thing as a nondeterministic operator or a nondeterministic function. If some expression equals 42 today, then it equaled 42 yesterday and it will still equal 42 next year. The PlusCal statement

$$v := \text{CHOOSE } n \in 1\,..\,10 : \text{TRUE}$$

and its TLA$^+$ translation

$$v' = \text{CHOOSE } n \in 1\,..\,10 : \text{TRUE}$$

assign some value in $1\,..\,10$ to v. The semantics of TLA$^+$ do not specify which value in $1\,..\,10$; but it is the same one every time. If you want nondeterministic assignment, you can use the following PlusCal statement.

$$\textbf{with } (n \in 1\,..\,10)\ \{\ v := n\ \}$$

This statement sets v to a nondeterministically chosen number in $1\,..\,10$ that may differ each time it is executed. Its TLA$^+$ translation is

$$\exists n \in 1\,..\,10 : v' = n$$

If there is no element x in S satisfying $P(x)$, then we don't know anything about the value of $\text{CHOOSE } x \in S : P(x)$. For example, $\text{CHOOSE } n \in \mathit{Int} : n^2 = 2$ could be any value. TLC will report an error if it tries to evaluate such a CHOOSE.

There is also an unbounded version of CHOOSE that corresponds to unbounded existential quantification. Like unbounded quantification, it is not handled by the TLC model checker. In writing specifications, it is used only in the following idiom, which defines *NotAnS* to be an arbitrary value that is not an element of S.

$$NotAnS \;\triangleq\; \text{CHOOSE } x : x \notin S$$

For example, we might want the value of a variable v to be either an integer or else some special value that indicates an error. We could define

$$Error \;\triangleq\; \text{CHOOSE } n : n \notin Int$$

and let v satisfy the type invariant $v \in Int \cup \{Error\}$. When creating a model for a specification with this definition, the Toolbox will by default override the definition by letting *Error* be a model value.

14 Sets

This has not yet been written. See Section 1.2 of *Specifying Systems*.

?
←
→
C
I
S

14.1 An Introduction to Sets

This section will define $\{e_1, \ldots, e_n\}$ (including the empty set $\{\}$ and $\in$ (and $\notin$).

If this has been a detour, you may now want to return to Section 2.5 of the Starting Track.

14.2 Simple Set Operators

Click on the operators to see their definitions:

$$\cup, \; \cap, \; \subseteq \qquad \setminus$$

See Section 1.2 of *Specifying Systems*.

The standard *FiniteSets* module defines $Cardinality(S)$ to equal the cardinality (the number of elements in) S, if S is a finite set. The value of $Cardinality(S)$ is unspecified if S is not a finite set.

If this has been a detour, you may now want to return to Section 4.1.3 of the Starting Track.

14.3 Set Constructors

This section will define the constructs $\{x \in S : P(x)\}$ and $\{e(x) : x \in S\}$. Now, see Section 6.1 of *Specifying Systems*.

14.4 SUBSET and UNION

For any set S, the set of all subsets of S is written $\text{SUBSET } S$. For example: $\text{SUBSET } \{1, 2, 3\}$ equals

$$\{\, \{\}, \; \{1\}, \; \{2\}, \; \{3\}, \; \{1, 2\}, \; \{1, 3\}, \; \{2, 3\}, \; \{1, 2, 3\} \,\}$$

Mathematicians call $\text{SUBSET } S$ the *power set* of S and write it $\mathcal{P}(S)$ or 2^S. The latter notation comes from the fact that, if S is a finite set S, then:

$$Cardinality(\text{SUBSET } S) \;=\; 2^{Cardinality(S)}$$

Note that the empty set has a single subset—namely, the empty set. Thus, $\text{SUBSET } \{\}$ equals $\{\{\}\}$.

If S is a set of sets (a set whose elements are sets), then UNION S is the union of all the elements of S. Thus, S is the set containing all the elements of elements of S. In other words:

$$(x \in \text{UNION } S) \;\equiv\; (\exists s \in S : x \in s)$$

If S equals the finite set $\{s_1, s_2, \ldots, s_n\}$ then UNION S equals

$$s_1 \cup s_2 \cup \ldots \cup s_n$$

?
←
→
C
I
S

Note that UNION $\{\}$ equals $\{\}$. Mathematicians usually write UNION as $\bigcup$.

14.5 Collections too Big to Be Sets

This will contain a discussion of Russell's paradox and what are sometimes called *classes* in set theory. See page 66 of *Specifying Systems*.

14.6 Bags

This will contain a brief discussion of bags, also called multisets, and the standard *Bags* module. See page 340 of *Specifying Systems*.

15 Functions

?
←
→
C
I
S

Functions are what programmers call arrays. However, ordinary programming languages permit only a small, rather dull class of functions/arrays. For example, they allow only finite arrays—what are known to mathematicians as functions with finite domains. Some languages permit only arrays (functions) with index sets (domains) of the form $0 \,..\, k$ for some integer k.

We begin with a description of arbitrary functions. We then discuss data types that, in TLA$^+$, are special kinds of functions. These are tuples (also known as finite sequences and called *lists* by programmers), records, and strings.

15.1 Functions and Their Domains

Mathematicians define a function to be a set of ordered pairs. In practice, it is more convenient to consider ordered pairs and other tuples to be functions. TLA$^+$ therefore takes a function to be a primitive object, without specifying how it is defined in terms of sets.

A good way to learn about functions is to let TLC evaluate expressions containing them. Open a new spec in the Toolbox. So we can use operations on integers for our examples, have the spec import the *Integers* module by adding

EXTENDS *Integers*

Create a new model, which you can use to have TLC evaluate expressions.

What programmers call the *index set* of an array, mathematicians call the *domain* of a function. The domain of a function f is written DOMAIN f. Since a tuple is a function, we can see what its domain is. Let's have TLC show us what the domain of a triple is by having it evaluate:

DOMAIN ⟨“a”, “b”, “c”⟩

```
DOMAIN <<"a", "b", "c">>
```

It tells us that its domain is the set $1 \,..\, 3$ of integers from 1 to through 3. As you can guess, an n-tuple is a function whose domain is the set $1 \,..\, n$.

What about a 0-tuple? Have TLC show you what the domain of the 0-tuple ⟨ ⟩ is.

If f is a function and x is an element of DOMAIN f, mathematicians write the value of f applied to x as $f(x)$. TLA$^+$ denotes function application with square brackets instead of parentheses, writing $f[x]$ instead of $f(x)$. (It reserves parentheses for operator application.) Have TLC evaluate the expression

What's the difference between a function and an operator?

⟨“a”, “b”, “c”⟩[2]

As you probably expected, it equals "b". If τ is an n-tuple, then $\tau[i]$ equals the i^{th} element of τ, for any i in $1\,..\,n$. Now have TLC evaluate

$$\langle \text{"a"}, \text{"b"}, \text{"c"} \rangle[4]$$

This produces the TLC error

```
Attempted to apply tuple
<<"a", "b", "c">>
to integer 4 which is out of domain.
```

?
←
→
C
I
S

For a function f, the value of $f[x]$ is unspecified if x is not an element of the domain of f.

15.2 Writing Functions

A tuple is a particular kind of function—namely, one whose domain is the set $1\,..\,n$ for some natural number n. We will need to be able to write functions with arbitrary domains. Mathematics does not provide a standard way of writing functions, so the TLA$^+$ notation for writing functions will mean nothing to you. To illustrate the notation, I'll first describe how to use the TLA$^+$ notation to write a tuple. Have TLC evaluate the following expression.

$[i \in 1\,..\,3 \mapsto i - 7]$ `[i \in 1..3 |-> i - 7]`

It reports that this expression equals the triple $\langle 1\text{–}7, 2\text{–}7, 3\text{–}7 \rangle$, which it writes as $\langle -6, -5, -4 \rangle$.

In general, the expression $[v \in S \mapsto e]$ is the function with domain S such that $[v \in S \mapsto e][x]$ equals the expression obtained from e by substituting x for v, for any x in S. For example, use TLC to check that

$[i \in \{2, 4, 6, 8\} \mapsto i - 42][4]$ `[i \in {2, 4, 6, 8} |-> i - 42][4]`

equals $4 - 42$ (which it writes as -38).

The domain of a function does not have to be a finite set. For example, the function $[i \in \mathit{Nat} \mapsto i - 42]$ has as domain the set Nat of all natural numbers. Use TLC to check that $[i \in \mathit{Nat} \mapsto i - 42][88]$ equals $88 - 42$, and that -88 is not in its domain.

Now, have TLC evaluate the (function-valued) expression

$[i \in \{2, 4, 6, 8\} \mapsto i - 42]$ `[i \in {2, 4, 6, 8} |-> i - 42]`

TLC writes its value as

$$(2 :> -40 \;@@\; 4 :> -38 \;@@\; 6 :> -36 \;@@\; 8 :> -34)$$

In TLA$^+$, the operator $:>$ has higher precedence (binds more tightly) than @@. This expression therefore equals

$$(2 \;:>\; -40) \;@@\; (4 \;:>\; -38) \;@@\; (6 \;:>\; -36) \;@@\; (8 \;:>\; -34)$$

In general, if f is a function with finite domain $\{d_1, \ldots, d_b\}$ and is not a special kind of function like a tuple, then TLC prints it as

$$(d_1 \;:>\; f[d_1] \;\;@@\;\; \ldots \;\;@@\;\; d_n \;:>\; f[d_n])$$

The operators $:>$ and @@ are defined in the standard *TLC* module. Import that module by adding it to the EXTENDS statement. (Don't forget the comma between `Integers` and `TLC`.) Save the module and have TLC evaluate

$(1 \;:>\; \text{"a"} \;@@\; 2 \;:>\; \text{"b"} \;@@\; 3 \;:>\; \text{"c"})$ `(1 :> "a"  @@  2 :> "b"  @@  3 :> "c")`

It prints this function as the tuple $\langle \text{"a"}, \text{"b"}, \text{"c"} \rangle$. Here is how the *TLC* module defines the operators $:>$ and @@:

$$d \;:>\; e \;\;\triangleq\;\; [x \in \{d\} \mapsto e]$$

$$f \;@@\; g \;\;\triangleq\;\; [x \in (\text{DOMAIN } f) \cup (\text{DOMAIN } g) \mapsto \text{IF } x \in \text{DOMAIN } f \text{ THEN } f[x] \text{ ELSE } g[x]]$$

Question 15.1 Define the function F by

$$F \;\;\triangleq\;\; \langle \text{"a"}, \text{"b"}, \text{"c"} \rangle \;@@\; (2 \;:>\; -3 \;@@\; 4 \;:>\; \langle 1, \text{"d"} \rangle \;@@\; 6 \;:>\; 37)$$

Determine the values of the following expressions, and then use TLC to check your answers.

$$\text{DOMAIN } F \qquad F[1] \qquad F[2] \qquad F[4] \qquad F[5]$$

You can define a symbol f to equal a function in the obvious way—for example, writing:

$f \;\;\triangleq\;\; [i \in 1\,..\,5 \mapsto i^2]$ `f == [i \in 1..5 |-> i^2]`

TLA$^+$ also provides a special syntax for defining a symbol to equal a function. For example, the following defines g to equal the same function as f above.

$g[i \in 1\,..\,5] \;\;\triangleq\;\; i^2$ `g[i \in 1..5] == i^2`

Add these two definitions to the module and then use TLC to check that $f = g$ equals TRUE.

The semantics of TLA$^+$ does not say what a function is—just as it does not say what a number is. All it tells us about functions is that two functions f and

g are equal if they have the same domain and $f[x]$ equals $g[x]$ for all x in that domain.

So far, we have discussed functions of a single argument. Mathematicians also use functions of multiple arguments. In TLA$^+$, a function of n arguments, where $n > 1$, is equivalent to a function whose domain is a set of n-tuples. For example, $f[a, b]$ is shorthand for $f[\langle a, b \rangle]$. Here are four ways to define a function f with domain $Nat \times Int$ such that $F[a, b] = a + b^2$ for all $a \in Nat$ and $b \in Int$.

× is defined in Section 15.5

? ← → C I S

$$\begin{aligned} f &\triangleq [a \in Nat,\ b \in Int \mapsto a + b^2] \\ f &\triangleq [u \in Nat \times Int \mapsto u[1] + u[2]^2] \\ f[a \in Nat,\ b \in Int] &\triangleq a + b^2 \\ f[u \in Nat \times Int] &\triangleq u[1] + u[2]^2 \end{aligned}$$

TLA$^+$ provides some abbreviations for writing functions of multiple arguments that are similar to the ones for nested quantification. For example, the following two expressions are equivalent.

$$[x \in Nat,\ y, z \in Int \mapsto x - (2 * y + z)]$$
$$[x \in nat,\ y \in Int, z \in Int \mapsto x - (2 * y + z)]$$

15.3 Sets of Functions

The expression $[S \to T]$ is the set of all functions f whose domain is S such that $f[x]$ is in the set T for all x in S. (The arrow → is typed `->`.) This set corresponds roughly to what a programmer would call the set of all arrays of type T indexed by S. Use TLC to see what the following two sets of functions equal.

$[\{2, 4\} \to \{“a”, “b”, “c”\}]$ `[{2, 4} -> {"a", "b", "c"}]`

$[1\,..\,3 \to \{“a”, “b”\}]$ `[1..3 -> {"a", "b"}]`

Question 15.2 If the set S has m elements and the set T has n elements, how many elements does $[S \to T]$ contain? Check your answer for $m = 0$ and for $n = 0$.

ANSWER

15.4 The EXCEPT Construct

If you have done any programming, you've probably written an assignment statement such as

```
A[3] = 42
```

With the TLA$^+$ notation that A represents the original value of A and A' represents its new value, some people think that the effect of executing this assignment is represented by the formula

$$A'[3] = 42$$

It's not, because this formula says nothing about the new value of any element of the array except $A[3]$. For example, it says nothing about the new value of $A[2]$.

A more sophisticated belief is that the assignment is represented by

$$\forall\, i \in \textsc{domain}\ A : \ A'[i] \ = \ (\textsc{if}\ i = 3\ \ \textsc{then}\ 42\ \ \textsc{else}\ A[i])$$

However, this is not correct either because it does not prohibit the domain of A from changing. For example, the old value of A could be a function whose domain is the set Nat of natural numbers, and the new value could be a function with domain $Nat \cup \{-7\}$. We could conjoin to the formula above the requirement that $\textsc{domain}\ A$ is unchanged, but that's getting pretty complicated—and it's still not right.

To specify the meaning of the assignment statement, we must state explicitly what the value A' equals. When we realize that's what we have to do, it's fairly obvious that we can do it with the formula:

$$A' \ = \ [\, i \in \textsc{domain}\ A \mapsto \textsc{if}\ i = 3\ \ \textsc{then}\ 42\ \ \textsc{else}\ A[i]\,]$$

This formula is correct, but it's a nuisance to have to write such a long formula to represent something as common as a simple assignment statement. TLA$^+$ therefore allows us to write the right-hand side of this formula as:

$$[A\ \textsc{except}\ ![3] = 42]$$

No one likes this notation. People want to know what the ! means. It doesn't mean anything; it's just a piece of syntax. The TLA$^+$ EXCEPT notation is terrible, but I don't know a better way to write the function f that is identical to A except that $f[3]$ equals 42. I've gotten used to it; in time you will too.

TLA$^+$ allows some useful generalizations of the EXCEPT notation. The function f that is identical to A except that $f[3] = 42$ and $f[6] = 24$ can be written

$$[A\ \textsc{except}\ ![3] = 42,\ ![6] = 24]$$

In general,

$$[A\ \textsc{except}\ ![i] = d,\ ![j] = e] \ \ = \ \ [\,[A\ \textsc{except}\ ![i] = d]\ \ \textsc{except}\ ![j] = e\,]$$

You can guess the meaning of

$$[A\ \textsc{except}\ ![i_1] = d_1, \ldots, ![i_n] = d_n]$$

?
←
→
C
I
S

If $A[3]$ is a function, we can represent the assignment statement `A[3][j] = "a"` by the formula

$$A' = [A \ \textsc{except}\ ![3][j] = \text{“a”}]$$

If you think about it a bit, you'll see that:

$$[A \ \textsc{except}\ ![i][j] = e] \ = \ [\,A \ \textsc{except}\ ![i] = [A[i] \ \textsc{except}\ ![j] = e]\,]$$

?
←
→
C
I
S

Question 15.3 Figure out what the following expression equals:

$$[\,\langle\,\text{“a”}, \text{“b”}, \langle\,\text{“c”}, \langle\,\text{“d”}, \text{“e”}\,\rangle\rangle\rangle \ \textsc{except}\ ![1] = \text{“X”}, \ ![3][2][1] = \text{“Y”}\,]$$

Let TLC check your answer.

The @ Notation

When writing an expression of the form

$$[A \ \textsc{except}\ ![i] = e]$$

the expression $A[i]$ often appears as a subexpression of e. TLA$^+$ allows it to be abbreviated in e as @. For example, the formula

$$A' = [A \ \textsc{except}\ ![i] = A[i] + 1]$$

can be written as

$A' = [A \ \textsc{except}\ ![i] = @ + 1]$ `A' = [A \EXCEPT ![i] = @+1]`

Similarly, the following two expressions are equivalent:

$$[A \ \textsc{except}\ ![i][j] = 2 * A[i][j]] \qquad [A \ \textsc{except}\ ![i][j] = 2 * @]$$

The @ notation doesn't save much space and is confusing to anyone not familiar with it. I therefore recommend avoiding it unless it is used often and you are writing your specification for someone already familiar with TLA$^+$.

15.5 Tuples and Finite Sequences

In Section 15.1 above, we explained that an n-tuple t is a function whose domain is the set $1\,..\,n$ of natural numbers, where $t[n]$ is the n^{th} component of t. For example, $\langle\, x + 1, 42, \text{“a”}\,\rangle[3]$ equals “a”.

Sets of tuples can be written with the Cartesian product operator ×, typed as `\X`. For example, $Nat \times Int \times \{\text{“a”}, \text{“bc”}\}$ is the set of all triples $\langle i, j, k \rangle$ where i is a natural number, j is an integer, and k is either “a” or “bc”.

Question 15.4 What do the following three sets equal?

$$S \times T \times U \qquad (S \times T) \times U \qquad S \times (T \times U)$$

Check your answer by having TLC evaluate these three expressions for particular sets S, T, and U.

?
←
→
C
I
S

Another name for a tuple is a *finite sequence*. Finite sequences are known to programmers as lists. Most programmers think of lists and tuples as having different types, and they will find it strange to learn that we consider them to be the same. However, if you forget about the idiosyncracies of programming languages, lists and tuples are both just sequences of elements.

The standard *Sequences* module defines the following operations on finite sequences.

$\mathit{Seq}(S)$ The set of all sequences of elements of the set S. For example, $\langle 3, 7 \rangle$ is an element of $\mathit{Seq}(\mathit{Nat})$. Check that TLC can evaluate the following expressions (even though $\mathit{Seq}(\mathit{Nat})$ is an infinite set).

$$\langle 3, 7 \rangle \in \mathit{Seq}(\mathit{Nat}) \qquad \langle 3, -8 \rangle \in \mathit{Seq}(\mathit{Nat})$$

$\mathit{Head}(s)$ The first element of sequence s. For example, $\mathit{Head}(\langle 3, 7 \rangle)$ equals 3.

$\mathit{Tail}(s)$ The tail of sequence s, which consists of s with its first element removed. For example, $\mathit{Tail}(\langle 3, 7, \text{“a”} \rangle)$ equals $\langle 7, \text{“a”} \rangle$.

$\mathit{Append}(s, e)$ The sequence obtained by appending element e to the tail of sequence s. For example, $\mathit{Append}(\langle 3, 7 \rangle, 3)$ equals $\langle 3, 7, 3 \rangle$.

$s \circ t$ The sequence obtained by concatenating the sequences s and t. For example, $\langle 3, 7 \rangle \circ \langle 3 \rangle$ equals $\langle 3, 7, 3 \rangle$. (We type $\circ$ as `\o`.)

$\mathit{Len}(s)$ The length of sequence s. For example, $\mathit{Len}(\langle 3, 7 \rangle)$ equals 2.

$\mathit{SubSeq}(s,\ m,\ n)$ The subsequence $\langle s[m],\ s[m+1],\ \ldots,\ s[n] \rangle$ consisting of the m^{th} through n^{th} elements of s. It is undefined if $m < 1$ or $n > \mathit{Len}(s)$, except that it equals the empty sequence if $m > n$.

$\mathit{SelectSeq}(s,\ \mathit{Op})$ If Op is an operator that takes a single argument, then this equals the subsequence of s consisting of the elements $s[i]$ such that $\mathit{Op}(s[i])$ equals TRUE. For example, if Op is defined by

$$\mathit{Op}(n) \ \triangleq\ n > 0$$

then $\mathit{SelectSeq}(\langle 0,\ 1,\ -1,\ 2,\ -2 \rangle)$ equals $\langle 1,\ 2 \rangle$.

Question 15.5 For what set S is $\mathit{Seq}(S)$ a finite set?

ANSWER

15.6 Records

Mathematicians represent an object with several components as a tuple. For example, a mathematician might define a graph to be a pair $\langle N, E \rangle$, where N is its set of nodes and E is its set of edges. She would then use N and E to mean the sets of nodes and edges of a graph G. If she were being completely rigorous and using the notation of TLA$^+$, she would have to write $G[1]$ and $G[2]$ instead of N and E. (If she were restricted to ordinary mathematical notation, she would have no way to be rigorous.) This would be inelegant but feasible. However, if she defined a Turing machine T to be a 7-tuple, it would be impossible to remember if its initial state was $T[4]$ or $T[5]$.

? ← → C I S

Programming languages solve this problem by introducing records. A graph G can be a record with *nodes* and *edges* field, where $G.nodes$ and $G.edges$ are its sets of nodes and edges. TLA$^+$ represents a record mathematically as a function whose domain is a finite set of strings. The record G is represented as a function whose domain is $\{$"nodes", "edges"$\}$. TLA$^+$ defines $G.nodes$ and $G.edges$ to be abbreviations of $G[$"nodes"$]$ and $G[$"edges"$]$, respectively. We adopt the terminology of programming languages, saying that *nodes* and *edges* are the *fields* of the record G.

TLA$^+$ provides a special notation for writing records. The record r with fields *nodes* and *edges* such that $r.nodes$ equals N and $r.edges$ equals E is written as:

$$[nodes \mapsto N,\ edges \mapsto E]$$

```
[nodes |-> N, edges |-> E]
```

It can also be written as $[edges \mapsto E,\ nodes \mapsto N]$. (Because of how we represent a record as a function, there is no notion of ordering of the fields.)

TLA$^+$ also provides a notation for writing sets of records. The expression

$$[h_1 : S_1, \ldots, h_n : S_n]$$

is the set of all records $[h_1 \mapsto e_1, \ldots, h_n \mapsto e_n]$ such that e_i is in S_i, for all i in $1 \,..\, n$. For example,

$$[nodes : \{Nat\},\ edges : \text{SUBSET}\ (Nat \times Nat)]$$

is the set of all records G such that $G.nodes$ equals Nat (the only element of $\{Nat\}$) and $G.edges$ is a set of pairs of natural numbers (an element of SUBSET $(Nat \times Nat)$).

See the definitions of SUBSET and of ×.

Question 15.6 Write the set $[a : A,\ b : B]$ of records using the notation for sets of functions described in Section 15.3.

ANSWER

The convention of $.nodes$ being an abbreviation for $[$"nodes"$]$ extends to the EXCEPT construct described in Section 15.4. Thus,

$$[G \text{ EXCEPT } !.nodes = NN]$$

is an abbreviation for

$$[G\ \textsc{except}\ ![\text{“nodes”}] = \mathit{NN}]$$

which is the record that is the same as G except that its *nodes* field equals NN.

This record convention also extends to PlusCal assignment statements, so $G.\mathit{edges} := \mathit{EE}$ is equivalent to $G[\text{“edges”}] := \mathit{EE}$, which sets the value of $G.\mathit{edges}$ to EE and leaves the other fields of the variable G unchanged.

?
←
→
C
I
S

15.7 Strings

A string is sequence of characters enclosed in double quotes, such as “abcd”, which is written as `"abcd"`. When writing specifications, you will almost always think of strings as an elementary data type, with no internal structure. TLA$^+$ actually defines strings to be tuples (finite sequences) of characters, though it does not specify what a character is. (The only way to represent the character a in TLA$^+$ is as part of a string, such as “abc”[1].) However, TLC has limited knowledge of strings as functions. About all that TLC can do with strings is to test if they are equal and evaluate the operators ∘ (sequence concatenation) and *Len* (the length of a sequence) of the standard *Sequences* module.

A TLA$^+$ string may contain the following characters

$$a\ b\ c\ d\ e\ f\ g\ h\ i\ j\ k\ l\ m\ n\ o\ p\ q\ r\ s\ t\ u\ v\ w\ x\ y\ z$$
$$A\ B\ C\ D\ E\ F\ G\ H\ I\ J\ K\ L\ M\ N\ O\ P\ Q\ R\ S\ T\ U\ V\ W\ X\ Y\ Z$$
$$0\ 1\ 2\ 3\ 4\ 5\ 6\ 7\ 8\ 9$$
$$\tilde{}\ @\ \#\ \$\ \%\ \hat{}\ \&\ *\ ?\ -\ +\ =\ (\ )\ \{\ \}\ [\ ]\ <\ >\ |\ /\ ,\ .\ ?\ :\ ;\ `\ '$$

plus the following six special characters, each of which is typed as a \ (backslash) followed by a second character:

`\"`	" (double quote)	`\t`	tab	`\f`	form feed
`\\`	\ (backslash)	`\n`	line feed	`\r`	carriage return

?
←
→
C
I
S

16 Miscellaneous Constructs

16.1 Conditional Constructs

16.1.1 IF / THEN / ELSE

An IF / THEN / ELSE construct has the obvious meaning: the expression

$\text{IF } x > 0 \text{ THEN } x \text{ ELSE } -x$ `IF x > 0 THEN x ELSE -x`

equals x if $x > 0$ equals TRUE, and it equals $-x$ if $x > 0$ equals FALSE. If x is a number, then this expression equals its absolute value.

The TLA$^+$ IF versus the PlusCal **if**.

When a module is parsed, the ELSE is treated as if it were a prefix operator with the lowest possible precedence. This means that as much of the text as possible that follows the ELSE is considered to be part of the ELSE expression. An ELSE clause is often terminated by the end of a definition or of a bulleted disjunction or conjunction in which it appears, as in:

$$\begin{array}{l} \mathit{AbsoluteValue}(x) \triangleq \text{IF } x > 0 \text{ THEN } x \text{ ELSE } -x \\ \mathit{Sign}(x) \triangleq \ldots \end{array}$$

or

$$\mathit{Foo} \triangleq \begin{array}{l} \land\ x' = \text{IF } y > 0 \text{ THEN } x + 1 \text{ ELSE } x - 1 \\ \land\ y' = x \end{array}$$

An IF / THEN / ELSE nested inside an outer THEN clause is often ended by the outer ELSE. If you're not sure where the parser will think it ends, enclose an IF / THEN / ELSE in parentheses.

16.1.2 CASE

The expression

$\text{CASE } p_1 \rightarrow e_1 \;\Box\; \ldots \;\Box\; p_n \rightarrow e_n$ `CASE` p_1 `->` e_1 `[]` ... `[]` p_n `->` e_n

equals e_i for some i for which p_i equals TRUE. If no p_i equals TRUE, then the value of the expression is unspecified. If p_i equals TRUE for more than one value of i, then the expression might equal any of the corresponding e_i; the semantics of TLA$^+$ do not specify which one. (As in the case of CHOOSE, there is no nondeterminism.) A CASE expression in which more than one p_i can be true is most often used when the value of the expression does not depend on which of the possible e_i the expression equals in that case. For example,

CASE expressions versus CASE proof steps.

$$\text{CASE } x \geq 0 \rightarrow x \;\Box\; x \leq 0 \rightarrow -x$$

equals x if $x > 0$ equals TRUE, $-x$ if $x < 0$ equals TRUE, and either x or $-x$ if $x = 0$ equals TRUE—which in that case both equal 0. Thus, this expression equals the absolute value of x, if x is a number.

The statement

$\text{CASE } p_1 \rightarrow e_1 \,\Box\, \ldots \,\Box\, p_n \rightarrow e_n \,\Box\, \text{ OTHER } \rightarrow e$ `CASE` p_1 `->` e_1 `[] ... []` p_n `->` e_n `OTHER ->` e

equals e if none of the p_i equals TRUE; otherwise it equals

$$\text{CASE } p_1 \rightarrow e_1 \,\Box\, \ldots \,\Box\, p_n \rightarrow e_n$$

As with an IF / THEN / ELSE, as much text as possible following a CASE expression is considered to be part of that expression.

?
←
→
C
I
S

16.2 Definitions

16.2.1 Simple Operator Definitions

Despite the important role that definitions play in mathematicians, logicians have largely ignored them, and there is no standard formal notation for writing definitions. In TLA$^+$, a definition is simply an abbreviation. There is no concept of an illegal definition; any syntactically correct definition defines something.

The simplest form of definition defines an identifier to be an abbreviation for an expression. For example,

$id \;\triangleq\; a + b$ `id == a + b`

defines id to be an abbreviation for the expression $a + b$. Thus, $2 * id$ equals the expression obtained from it by substituting $a + b$ for id. Here, substitution means semantic substitution rather than syntactic substitution, so $2 * id$ equals $2 * (a + b)$ rather than $2 * a + b$.

A defined operator is also an abbreviation. For example,

$$Op(a) \;\triangleq\; b * a + c$$

defines $Op(exp)$, for any expression exp, to be an abbreviation for the expression obtained by substituting exp for a in the expression $b * a + c$. Again, that is semantic substitution, so $Op(2 + 2)$ equals $b * (2 + 2) + c$ and not $b * 2 + 2 + c$.

Semantic substitution is tricky in the presence of bound identifiers. For example, if F is defined by

$$F(x) \;\triangleq\; \exists\, i \in S \,:\, x + 1 > i$$

then $F(i)$ does *not* equal $\exists\, i \in S : i + 1 > i$. To explicitly perform the semantic substitution, we must replace the bound identifier i by an identifier that is not defined or declared in the current context. For example, $F(i)$ equals the formula $\exists\, ii \in S : i + 1 > ii$.

TLA$^+$ provides a number of user-definable infix operators and a few user-definable postfix operators, all listed here□. They are defined like this:

$a \mathbin{++} b \;\triangleq\; (a + 2 * b) \,\%\, N$ `a ++ b == (a + 2*b) % N`

How to type symbols with non-obvious ASCII representations is shown here□.

TLA$^+$ allows higher-order operators, which are ones having operators as arguments. For example, the standard *TLC* module defines an operator *SortSeq* so that if s is a sequence and $\prec$ is an operator, then $\mathit{SortSeq}(s, \prec)$ equals a permutation of s sorted according to $\prec$. Thus, $\mathit{SortSeq}(\langle 1,\ 5,\ 3\rangle, >)$ equals $\langle 5, 3, 1\rangle$. The definition of *SortSeq* has the form:

?
←
→
C
I
S

$$\mathit{SortSeq}(s,\ \_ \prec \_)\ \triangleq\ \ldots$$

```
SortSeq(s, _ \prec _) == ...
```

It could also have been written

$$\mathit{SortSeq}(s, \mathit{LT}(\_, \_))\ \triangleq\ \ldots$$

```
SortSeq(s, LT(_, _)) == ...
```

This is the highest-order operator we can define. TLA$^+$ does not allow a higher-order operator to be an argument of another operator.

16.2.2 Function Definitions

TLA$^+$ provides a special syntax for defining functions. The following two definitions are equivalent.

$$\mathit{Successor}[i \in \mathit{Nat}]\ \triangleq\ i + 1$$
$$\mathit{Successor}\ \triangleq\ [i \in \mathit{Nat} \mapsto i + 1]$$

However, the definition syntax permits recursive function definitions, such as:

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

This definition is equivalent to

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

Function definitions for functions of multiple arguments are written in the obvious way.

You can write nonsensical recursive definitions, such as

$$\mathit{tcaf}[n \in \mathit{Nat}]\ \triangleq\ n * \mathit{tcaf}[n + 1]$$

This defines *tcaf* to equal

$$\text{CHOOSE } f : f = [n \in \mathit{Nat} \mapsto n * f[n + 1]]$$

This expression might equal $[n \in \mathit{Nat} \mapsto 0]$, since that function satisfies the CHOOSE formula that defines *tcaf*. Or, it might equal $[n \in \mathit{nat} \mapsto \infty]$ for some other value ∞. Or, it might equal some other function. The semantics of TLA$^+$ does not determine the value of *tcaf*. However, we know it is a function with domain *Nat* because there is an f satisfying the CHOOSE formula.

There are no illegal definitions in TLA$^+$, but a recursive definition might not define what you expect it to. You should use a recursive function definition only when there is a unique function satisfying the CHOOSE formula, as is the case for *factorial*.

16.2.3 Recursive Operator Definitions

With ordinary definitions, every symbol must be defined or declared before it is used. A recursive function definition allows the function being defined to appear in its defining expression. You can write recursive operator definitions by using the RECURSIVE declaration. For example, you can define a factorial operator by writing

?
←
→
C
I
S

$$\begin{array}{l}\textsc{recursive}\ \mathit{FactorialOp}(\_)\\ \mathit{FactorialOp}(n)\ \triangleq\ \textsc{if}\ n = 0\ \textsc{then}\ 1\ \textsc{else}\ n * \mathit{FactorialOp}(n-1)\end{array}$$

You can also write mutually recursive definitions. For example, in

$$\begin{array}{l}\textsc{recursive}\ F(\_,\_), G(\_)\\ H\ \triangleq\ \ldots\\ F(x,y)\ \triangleq\ \ldots\\ G(z)\ \triangleq\ \ldots\end{array}$$

F and G can appear on the right-hand side of all three definitions. (Naturally, H can appear on the right-hand side of the definitions of F and G.)

The meaning of recursive operator definitions is complicated and quite subtle. For example, what does the following definition mean?

$$X\ \triangleq\ \textsc{choose}\ x : x \neq X$$

It defines X to equal something, but certainly not to equal CHOOSE $x : x \neq X$, which is a value that doesn't equal X. Recursive operator definitions were not allowed in the first version of TLA$^+$ because I didn't know how to define what they mean. (Georges Gonthier helped figure out their meaning.)

All you need to know about a recursively defined operator F, and all you should depend upon, is this: For any value v, if the value of $F(v)$ can be computed by a finite number of "expansions" of the definition, then it equals that computed value. Thus, *FactorialOp*(7) equals $7*6*5*4*3*2*1*1$ (which equals 5040), but you should not care about the value of *FactorialOp*(-3). I would have to think hard to figure out whether or not it equals $-3 * \mathit{FactorialOp}(-4)$.

Question 16.1 Define *IntFact* by ANSWER

$$\mathit{IntFact}[n \in \mathit{Int}]\ \triangleq\ \textsc{if}\ n = 0\ \textsc{then}\ 1\ \textsc{else}\ n * \mathit{IntFact}[n-1]$$

Does *IntFact*$[-3]$ equal $-3 * \mathit{IntFact}[-4]$?

Unless what you want to define is a function, an inductive definition is usually more convenient to express with a recursive operator definition than a recursive function definition. However, TLAPS□ does not yet handle recursive operator definitions, so you must use recursive function definitions. You can turn a recursive operator definition into an ordinary definition by using a LET with recursive function definitions. For example, consider this recursive definition of the cardinality of a finite set:

```
RECURSIVE $Cardinality(\_)$
$Cardinality(S) \triangleq$
   IF $S = \{\}$ THEN 0
               ELSE $1 + Cardinality(S \setminus \{$CHOOSE $x \in S :$ TRUE$\})$
```

?
←
→
C
I
S

We can't define *Cardinality* to be a function because its domain would have to be the "set" of all sets, which isn't a set. However, we can write this definition as:

```
$Cardinality(S) \triangleq$
  LET $C[T \in$ SUBSET $S] \triangleq$
            IF $T = \{\}$ THEN 0
                        ELSE $1 + C[T \setminus \{$CHOOSE $x \in T :$ TRUE$\}]$
  IN  $C[S]$
```

TLA$^+$ does not permit recursive definitions of higher-order operators.

16.2.4 Recursive Or Inductive?

The terms *recursive definition* and *inductive definition* seem to be used interchangeably. I will distinguish them as follows:

- A *recursive* definition is one in which the symbol being defined appears in its definition.

- An *inductive* definition is one in which a function or operator is defined by (a) defining its value on its smallest arguments and (b) defining its value on any other argument in terms of its values on smaller arguments (for some appropriate definition of "small").

An inductive definition is therefore a special case of a recursive one. The definition of factorial for natural numbers is the classic example of an inductive definition. The definition of *tcaf* above is an example of a recursive definition that is not inductive.

The recursive definitions used by mathematicians seem to all be inductive. That's not true of computer scientists, who might write the following definition of the set of all finite sequences of integers:

> A finite sequence of integers is either the empty sequence or equals the prepending of an integer to a finite sequence of integers.

This informal definition is not inductive, and a mathematician would probably consider it to be incorrect because it is satisfied by both the set of all finite sequences and the set of all finite and infinite sequences. Computer scientists generally define such a recursive definition in terms of a least fixed point, so it defines the smallest set satisfying the recursion relation—in this case, the set of finite sequences. TLA$^+$ does not use a least fixed point semantics, so you should write only recursive definitions that are inductive.

Mathematics is so expressive that recursion is needed much less often in TLA$^+$ than in most specification languages used by computer scientists. For example, the set of all finite sequences of integers can be written in TLA$^+$ as:

? ← → C I S

$$\text{UNION } \{[1\,..\,n \rightarrow \mathit{Int}] : n \in \mathit{Nat}\}$$

16.3 The LET / IN Construct

A specification is a formula. In principle we can write it as a single large formula without using any definitions. We make the formula easier to understand by decomposing it with the aid of definitions. For example, we might decompose the definition of an operator H by introducing operators F and G. If F and G are used only in the definition of H, the complete specification might be easier to understand by making the definitions of F and G local to the definition of H. Local definitions are made with the LET / IN construct.

A LET / IN construct is an expression, and it can appear as a subexpression of any expression. However, it is most often used as the right-hand side of a definition, or as "most of" the right-hand side of a definition. For example, the definition of H might have the form

$$H(a, b) \;\triangleq\; \exists\, i \in \mathit{Nat} : \text{LET } \ldots \text{ IN } \ldots$$

where the LET clause is a list of definitions and the IN clause is an expression in which the defined symbols can be used. The symbols a, b, and i can be used in both the definitions and the IN clause. Here is a simple, meaningless example:

$$\begin{array}{lll} H(a, b) \;\triangleq\; \exists i \in \mathit{Nat} : & & \\ \quad \text{LET} & F(u) \;\triangleq\; u + i & \\ & G \;\triangleq\; a * F(2) & \\ \quad \text{IN} & G > F(b) & \end{array}$$

It defines $H(x, y)$ to equal $x * (2 + i) > y + i$.

Recursive function definitions□ and recursive operator definitions□ can appear in a LET clause. The RECURSIVE declaration is used the same way as in module-level definitions.

16.4 The LAMBDA Construct

A higher-order operator takes an operator as an argument. What can that argument be? It can't be an expression, since the value of an expression is not an operator. It can be the name of an already-defined operator; for example, we wrote $SortSeq(\langle 1, 5, 3\rangle, >)$ above, using the operator $>$ as an argument of the higher-order operator *SortSeq*. We might want to apply *SortSeq* to an operator that is not already defined. One way is to define the operator locally using a LET / IN, as in the expression

$$\begin{array}{ll} \text{LET} & LT(x, y) \ \triangleq \ x[1] > y[1] \\ \text{IN} & SortSeq(\langle\langle 1, \text{“a”}\rangle, \langle 5, \text{“c”}\rangle, \langle 3, \text{“x”}\rangle\rangle, \ LT\rangle \end{array}$$

?
←
→
C
I
S

which equals

$$\langle\langle 5, \text{“c”}\rangle, \langle 3, \text{“x”}\rangle, \langle 1, \text{“a”}\rangle\rangle$$

Another way is to write the operator as a LAMBDA expression:

$$SortSeq(\langle\langle 1, \text{“a”}\rangle, \langle 5, \text{“c”}\rangle, \langle 3, \text{“x”}\rangle\rangle, \ \text{LAMBDA } x, y : x[1] > y[1])$$

It should be obvious from this example how to write any ordinary operator as a LAMBDA expression. Higher-order operators cannot be written as LAMBDA expressions.

A LAMBDA expression can appear only as an operator argument of a higher-order operator or as a substitution for a constant operator parameter in an INSTANCE statement, as in:

$$\text{INSTANCE } M \text{ WITH } N \leftarrow 42, \ LT \leftarrow \text{LAMBDA } x, y : x[1] > y[1]$$

You cannot use a LAMBDA expression as an operator anyplace else. Even though it makes perfect sense, you can't write:

$$(\text{LAMBDA } x, y : x[1] > y[1])\,(4, 2) \qquad \text{This is illegal.}$$

?
←
→
C
I
S

17 Temporal Logic

Temporal logic is not difficult, but it is different from the ordinary logic that we use most of the time. That difference can lead to mistakes in reasoning. It's therefore important that you understand temporal logic clearly—especially if you want to write temporal logic proofs.

From the point of view of logic, mathematics is just a game of manipulating formulas. We show that certain formulas are theorems by applying formal rules of reasoning—rules that are independent of what those formulas mean. This hyperbook is about systems, not about logic. To use math in studying systems, we must understand what our formulas mean. The study of the meaning of formulas is called *semantics*.

We can define something only in terms of something else. In science and engineering, meaning is ultimately defined in terms of ordinary mathematics. So, I will define the meaning of temporal logic formulas in terms of ordinary mathematical expressions. (A formula is a Boolean-valued expression.) Before getting to temporal logic, I will precisely define the meaning of the non-temporal expressions that we have been using.

The expressions of ordinary math are what I have been calling *constant expressions*—ones like $[\{0,1\} \rightarrow \text{BOOLEAN}]$. A constant expression can also contain unspecified constants. In TLA$^+$, such constants are introduced by CONSTANT statements and as bound identifiers. For example, in the expression $\{i \in Nat : i > 2\}$, the identifier i is a constant in the subexpression $i > 2$. Unspecified constants are what mathematicians usually call variables. (Temporal logicians call them *rigid variables*.)

The simplest non-constant expressions we have seen are *state expressions* (often called state functions). A state expression is like an ordinary mathematical expression except it can also contain (unprimed) variables—symbols that in TLA$^+$ are declared in VARIABLE statements. (Temporal logicians call them *flexible variables*.) If pc is a variable and i is a constant, then $pc[i] = \text{“cs”}$ is a state expression. (The string “cs” is an ordinary mathematical value, just like the empty set $\{\}$ or the number 42.) A *state formula*, also called a *state predicate*, is a Boolean-valued state expression.

The meaning of a state expression E is a mapping $\llbracket E \rrbracket$ from states to constant expressions. I have written that a state is an assignment of values to variables. More precisely, a state assigns a constant expression to every possible variable name. We can consider a state s to be a record with an infinite number of components—one for every variable name—that assigns to any variable v the constant expression $s.v$. The meaning $\llbracket pc[i] = \text{“cs”} \rrbracket$ of the state expression $pc[i] = \text{“cs”}$ is the mapping that assigns to each state s the constant expression $(s.pc)[i] = \text{“cs”}$. In general, for any state expression E, we define $\llbracket E \rrbracket(s)$ to be the constant expression obtained by replacing every occurrence of every variable v in E by $s.v$.

I have defined an action to be a formula that may contain both primed and unprimed variables. More generally, let a *transition expression* be like an ordinary mathematical expression except that it may contain primed and unprimed variables. The meaning $\llbracket T \rrbracket$ of a transition expression T is the mapping that assigns to any pair $\langle s, t \rangle$ of states the constant expression obtained from T by, for each variable v, substituting $s.v$ for each unprimed occurrence of v in T and $t.v$ for each occurrence of v' in T. A state expression E is a transition expression that contains no primed variables, so $\llbracket E \rrbracket(\langle s, t \rangle)$ equals $\llbracket E \rrbracket(s)$ for all states s and t. An *action* is a Boolean-valued transition expression.

?
←
→
C
I
S

A temporal formula is one obtained by combining actions (and state predicates) with temporal operators and the ordinary operators of logic. Those ordinary operators of logic include unbounded quantification and bounded quantification over constant sets. Thus, $\forall N \in S : F$ is a temporal formula if S is a constant expression and F is a temporal formula.

TLA$^+$ allows S to be a state expression, not just a constant expression, in $\forall N \in S : F$; but we never write such a formula.

A *behavior* is an infinite sequence of states. The meaning of a temporal formula F is a mapping that assigns a Boolean-valued constant expression $\llbracket F \rrbracket(\sigma)$ to every behavior σ. An action A is considered to be a temporal formula such that if σ equals the sequence $s_1, s_2, \ldots$ of states, then $\llbracket A \rrbracket(\sigma)$ equals $\llbracket A \rrbracket(\langle s_1, s_2 \rangle)$. Thus, $\llbracket P \rrbracket(\sigma)$ equals $\llbracket P \rrbracket(s_1)$ if P is a state predicate.

The meaning of the temporal operator $\Box$ is defined by letting $\llbracket \Box F \rrbracket(\sigma)$ be true iff $\llbracket F \rrbracket(\tau)$ is true for all suffixes τ of σ. More precisely, if σ equals s_1, s_2, $\ldots$, let σ^{+i} be the suffix s_{i+1}, s_{i+2}, $\ldots$ of σ. Then $\llbracket \Box F \rrbracket(\sigma)$ is defined to equal $\forall\, i \in \mathit{Nat} : \llbracket F \rrbracket(\sigma^{+i})$.

We have defined the temporal operator $\Diamond$ by letting $\Diamond F$ equal $\neg \Box \neg F$. Since $\neg \forall\, i \in S : \neg \ldots$ equals $\exists\, i \in S : \ldots$, it follows that $\llbracket \Diamond F \rrbracket(\sigma)$ equals $\exists\, i \in \mathit{Nat} : \llbracket F \rrbracket(\sigma^{+i})$. That is, $\llbracket \Diamond F \rrbracket(\sigma)$ is true iff F is true for some suffix of σ (where a sequence is considered to be a suffix of itself).

The meanings of ordinary logical operators applied to temporal formulas is obvious. For example, $\llbracket (F \land G) \rrbracket(\sigma)$ is defined to equal $\llbracket F \rrbracket(\sigma) \land \llbracket G \rrbracket(\sigma)$. Similarly, $\llbracket (\forall\, i \in S : F_i) \rrbracket(\sigma)$ equals $\forall\, i \in S : \llbracket F_i \rrbracket(\sigma)$, for any constant expression S.

I have said nothing about formulas containing user-defined symbols. The meaning of such a formula is the meaning of the formula obtained by expanding the definitions of all user-defined symbols. It should be clear what this means in the absence of recursive definitions, so defined symbols can be removed by a finite number of expansions. Section 16.2.2 describes the meaning of recursive function definitions, showing how they can be expanded. The meaning of recursive operator definitions can be defined in a similar but considerably more complicated fashion.

17.1 Understanding Temporal Formulas

Intuitively, a temporal formula F is an assertion about a behavior. The formula $\Box F$ asserts that F is *always* true, meaning that it is true for all the behavior's suffixes. The formula $\Diamond F$ asserts that F is *eventually* true, meaning that it is true for some suffix of the behavior. (When discussing temporal logic, we take *eventually* to include *now*.)

?
←
→
C
I
S

Here are some common temporal formulas. You should think carefully about them until you find their meanings perfectly obvious.

$\Box\Diamond F$ True of a behavior σ iff F is true for infinitely many suffixes of σ. We read $\Box\Diamond$ as *infinitely often*.

$\Diamond\Box F$ True of a behavior σ iff F is true for all suffixes of some suffix τ of σ. We read $\Diamond\Box$ as *eventually always*.

The following formulas are tautologies, meaning that, for all temporal formulas F, G, H, and F_i, they are true for all behaviors. You should convince yourself that they are, indeed, tautologies. Note that $\Box$ and $\Diamond$ have higher precedence (bind more tightly) than $\wedge$, and $\vee$, which have higher precedence than $\Rightarrow$, $\equiv$, and $\leadsto$.

$$\neg\Box F \;\equiv\; \Diamond\neg F$$

$$\neg\Diamond F \;\equiv\; \Box\neg F$$

$$\Box\Box F \;\equiv\; \Box F$$

$$\Diamond\Diamond F \;\equiv\; \Diamond F$$

$$\Box(F \wedge G) \;\equiv\; (\Box F) \wedge (\Box G)$$

More generally: $\Box(\forall\, i \in S : F_i) \;\equiv\; (\forall\, i \in S : \Box F_i)$, for any constant expression S.

$$\Diamond(F \vee G) \;\equiv\; (\Diamond F) \vee (\Diamond G)$$

More generally: $\Diamond(\exists\, i \in S : F_i) \;\equiv\; (\exists\, i \in S : \Diamond F_i)$, for any constant expression S.

All such temporal logic tautologies can be proved from a small number of axioms and proof rules. However, with practice, you should understand the temporal operators $\Box$ and $\Diamond$ well enough that whether or not simple formulas like these are tautologies becomes obvious.

Question 17.1 Which of the following formulas are tautologies? Find counterexamples for those that aren't.

ANSWER

(a) $\Diamond\Box(F \wedge G) \;\equiv\; (\Diamond\Box F) \wedge (\Diamond\Box G)$

(b) $\Diamond\Box(F \vee G) \;\equiv\; (\Diamond\Box F) \vee (\Diamond\Box G)$

(c) $(\Box F \;\equiv\; F) \;\Rightarrow\; (F \wedge \Box G \;\equiv\; \Box(F \wedge G))$

(d) $(F \leadsto G) \vee (F \leadsto H) \;\Rightarrow\; (F \leadsto G \vee H)$

(e) $(F \leadsto G) \wedge (G \leadsto H) \;\Rightarrow\; (F \leadsto H)$

(f) $\Box\Diamond(F \wedge G) \;\equiv\; (\Box\Diamond F) \wedge (\Box\Diamond G)$

(g) $\Box\Diamond(F \vee G) \;\equiv\; (\Box\Diamond F) \vee (\Box\Diamond G)$

(h) $(\Diamond F \;\equiv\; F) \;\Rightarrow\; (F \vee \Diamond G \;\equiv\; \Diamond(F \vee G))$

(i) $(F \leadsto G) \wedge (F \leadsto H) \;\Rightarrow\; (F \leadsto G \wedge H)$

(j) $(F \wedge H \leadsto G) \;\Rightarrow\; (F \wedge \Box H \leadsto G \wedge \Box H)$

If you correctly answered this question, you should have no trouble understanding any of the temporal formulas that arise in reasoning about algorithms—assuming that you understand the state predicates and actions that occur in those formulas.

Fortunately, there's an easy way to check if a temporal logic formula is a tautology: let the TLAPS prover□ do it for you. For example, you can check if $\Diamond\Box(F \land G) \;\equiv\; (\Diamond\Box F) \land (\Diamond\Box G)$ is a tautology by asking TLAPS to check:

?
←
→
C
I
S

```
THEOREM ASSUME TEMPORAL F, TEMPORAL G
        PROVE  <>[](F /\ G) \equiv (<>[]F) /\ (<>[]G)
BY PTL
```

The PTL backend prover will succeed in proving the theorem (coloring it green) only if the formula is a tautology. If it fails, the formula is almost certainly not a tautology.

17.2 Proof Rules and Proofs

Truth versus provability.

Now that you understand temporal formulas, we can examine theorems and their proofs. A temporal formula is a *theorem* iff it is true for all behaviors. Let $\vdash F$ be the assertion that F is a theorem, so $\vdash F$ can be informally defined to equal $\forall\,\sigma : [\![F]\!](\sigma)$. The proof rule

$$\text{PR.}\quad \frac{F_1, \ldots, F_n}{G}$$

asserts $(\vdash F_1) \land \ldots \land (\vdash F_n) \;\Rightarrow\; (\vdash G)$. We often omit the commas when writing PR. Also, to save space, I will sometimes write this proof rule as $F_1, \ldots, F_n \vdash G$.

Proof rules cannot be written in TLA$^+$. The theorem

$$\begin{array}{lll}\textsc{theorem} & \textsc{assume} & F_1, \ldots, F_n \\ & \textsc{prove} & G\end{array}$$

asserts $\vdash (F_1 \land \ldots \land F_n \;\Rightarrow\; G)$, which is very different from Rule PR. A simple and striking example of the difference is provided by the proof rule $F \vdash \Box F$. This rule asserts that if F is true for all behaviors, then so is $\Box F$. The rule is valid because $\Box F$ asserts of a behavior σ that F is true for all suffixes of σ, and every suffix of σ is a behavior. Therefore $\Box F$ is true for every behavior if F is true for all behaviors. On the other hand, for a behavior σ, the formula $F \Rightarrow \Box F$ asserts $[\![F \Rightarrow \Box F]\!](\sigma)$, which is equivalent to $[\![F]\!](\sigma) \Rightarrow [\![\Box F]\!](\sigma)$. Hence it asserts that if F is true of σ, then it is true of all suffixes of σ. This is clearly not true for an arbitrary formula F and behavior σ, so $F \Rightarrow \Box F$ is not a tautology. It is a theorem for certain formulas F, for example if F equals $\Box(x = 1)$.

Question 17.2 Show that $\Box(x = 1) \Rightarrow \Box(\Box(x = 1))$ is a theorem. ANSWER

Temporal logic proof rules can be deduced from tautologies and a small number of basic proof rules. For example, the proof rule $F \vdash \Box F$ can be deduced from the proof rule $(F \Rightarrow G) \vdash (\Box F \Rightarrow \Box G)$, the tautology $\Box$ TRUE, and ordinary logic. You should be able to determine if a simple proof rule is valid by translating it into a statement about behaviors.

When reasoning about temporal formulas, it is surprisingly easy to get into trouble by confusing proof rules with formulas—for example, confusing the rule $F \vdash \Box F$ with the formula $F \Rightarrow \Box F$. Here's a very blatant example of this.

?
←
→
C
I
S

> THEOREM *Fallacy* $\triangleq$ $(y = 0) \Rightarrow \Box(y = 0)$
>
> Proof: To prove $A \Rightarrow B$, it suffices to assume A and prove B. Hence, it suffices to assume $y = 0$ and prove $\Box(y = 0)$. From $y = 0$ and the rule $F \vdash \Box F$, we deduce $\Box(y = 0)$. QED

The theorem is clearly not correct, and the error in the proof becomes clear if we express all the statements explicitly in terms of assertions about behaviors. However, the virtue of temporal logic is that it lets us reason about temporal formulas without having to translate those formulas into explicit assertions about behaviors. To understand how to do that without making mistakes, we need to examine proofs more closely.

A theorem

> THEOREM T

asserts that the formula T is true for all behaviors—that is, that $\vdash T$ is true. We prove this theorem by showing that T is true for some arbitrarily chosen behavior. In other words, we prove $[\![T]\!](\sigma)$ for some behavior σ. Consider a hierarchically structured proof of the theorem. The proof of the first top-level step can use only previously proved theorems, which are true for all behaviors σ. Hence, the first top-level step must be true for every behavior σ. Since subsequent top-level statements can use only previously-proved theorems or top-level statements, they too must be true for every behavior σ. In other words, every top-level statement is itself a theorem. Hence, we can apply the proof rule $F \vdash \Box F$ to every top-level statement F.

Suppose that the theorem contains no ASSUME/PROVE steps, including no SUFFICES ASSUME/PROVE steps. It's then easy to see that every step, not just the top-level ones, is proved assuming only formulas that are true for all behaviors, and hence every step is a theorem. We can therefore apply the rule $F \vdash \Box F$ to the formula F asserted by any step in the proof.

The problem comes when a proof contains an ASSUME/PROVE. Here is the "proof" of theorem *Fallacy* written more rigorously:

> 1. Suffices Assume: $y = 0$
> Prove: $\Box(y = 0)$
> Proof: Obvious.

2. Q.E.D.
   PROOF: By step 1 and the proof rule $F \vdash \Box F$, with $y = 0$ substituted for F.

The theorem asserts that $[\![(y = 0) \Rightarrow \Box(y = 0)]\!](\sigma)$ is true for all behaviors σ. The proof attempts to show that it is true for some particular arbitrary behavior σ. The SUFFICES statement asserts that to prove this, it suffices to assume $[\![y = 0]\!](\sigma)$ is true and prove that $[\![\Box(y = 0)]\!](\sigma)$. Thus, in the Q.E.D. proof, we can assume only that $[\![y = 0]\!](\sigma)$ is true for this particular behavior σ. However, the hypothesis of the proof rule $F \vdash \Box F$ asserts that $[\![F]\!](\tau)$ is true for all behaviors τ, not just that $[\![F]\!](\sigma)$ is true. Hence, that hypothesis is not satisfied for F equal to $y = 0$, and the proof rule cannot be applied.

?
←
→
C
I
S

In general, a formula A asserted in an ASSUME cannot be used to prove the hypothesis F of a proof rule, because the ASSUME asserts that A is true for the particular behavior under consideration, and the hypothesis of the proof rule asserts that F is true for all behaviors. Since formulas proved without using any assumptions in an ASSUME clause are true of all behaviors, they can be used as hypotheses of a proof rule. We can therefore stay out of trouble by never doing any temporal reasoning within the scope of an assumption from an ASSUME/PROVE or SUFFICES ASSUME/PROVE step.

While eschewing ASSUMEs permits us to use temporal proof rules freely, it's a Draconian restriction. There are some formulas that can be ASSUMEd without causing problems. For example, a constant formula in an ASSUME is harmless. Since it does not depend on the value of any variable, a constant formula is true of some single behavior iff it is true of every behavior. We can use some non-constant formulas in ASSUMEs, but to see why requires introducing a different kind of proof rule.

The temporal logic proof rules that we use do not require their hypotheses to be true for *all* behaviors. They require only that the hypotheses are *always* true. More precisely, to deduce that the conclusion is true for any behavior σ, they require only that their hypotheses be true for all suffixes of σ. We we write such a rule as:

$$\text{TR.}\quad \begin{array}{c} F_1, \ldots, F_n \\ \hline\hline G \end{array}$$

This asserts that for any behavior σ, formula G is true for σ if $F_1, \ldots, F_n$ is true for all suffixes of σ. In other words, it asserts:

$$\forall \sigma : (\forall i \in Nat : [\![F_1]\!](\sigma^{+i})) \land \ldots \land (\forall i \in Nat : [\![F_n]\!](\sigma^{+i})) \Rightarrow [\![G]\!](\sigma)$$

To save space, I sometimes write this rule as $F_1, \ldots, F_n \Vdash G$. Rule TR is stronger than rule PR, since it makes a weaker assumption about its hypotheses F_i. I will call a rule having the form TR a $\Vdash$ rule.

The symbol $\Vdash$ and $\Vdash$ rules are not standard terminology.

Instead of using the rule $F \vdash \Box F$, we can use the stronger rule $F \Vdash \Box F$. It asserts that, for any behavior σ, if F is true for all prefixes of σ then $\Box F$ is true for σ. It is valid by definition of $\Box$. In fact, rule TR is equivalent to

$$\text{TR}_{\Rightarrow}.\ \vdash\ \Box F_1 \land \ldots \land \Box F_n\ \Rightarrow\ G$$

For example, $F \Vdash \Box F$ is really the trivial assertion $\vdash \Box F \Rightarrow \Box F$. The reason we write TR rather than $\text{TR}_{\Rightarrow}$ is that there are rules in which some $\Box F_i$ may not be a legal TLA$^+$ formula. For example,

?
←
→
C
I
S

$$\begin{array}{c} P \Rightarrow P' \\ \hline\hline P \Rightarrow \Box P \end{array}$$

is a valid rule for any state predicate P. However, $\Box(P \Rightarrow P')$ is not a legal (syntactically correct) TLA$^+$ formula because an action (a non-temporal formula containing primes) A can occur in a temporal formula only in a subexpression of the form $\Box[A]_v$ or $\Diamond\langle A\rangle_v$. This restriction is to ensure that you can't write temporal formulas that you shouldn't, and that there is no need for you to write. However, all of our proof rules and all the reasoning we do here remain valid even if we do not place this restriction on our temporal formulas. (Our rules and reasoning can be invalid only for formulas that contain a **∃** or **∀** operator.)

Our $\Vdash$ rules make it unnecessary to write illegal formulas like $\Box(P \Rightarrow P')$ when reasoning about specifications. However, those formulas can be useful in understanding and reasoning about our proof rules. I will therefore sometimes use them in the rest of Section 17. I will use a red $\Box$ or $\Diamond$ when the operator is applied illegally to an action formula, as in $\Box(P \Rightarrow P')$.

Of course, TLAPS will only accept legal TLA$^+$ formulas. However, if each $\Box F_i$ is a TLA$^+$ formula, you can check that a rule of the form TR is valid by having TLAPS check if $\text{TR}_{\Rightarrow}$ is true.

Question 17.3 Prove the validity of the rule $(P \Rightarrow P') \Vdash (P \Rightarrow \Box P)$ for P a state predicate. ANSWER

If $\Box G$ is a legal formula, then rule TR is equivalent to

$$\text{TR}_{\Box}.\ \begin{array}{c} F_1, \ldots, F_n \\ \hline\hline \Box G \end{array}$$

Since $\Box G \Rightarrow G$ is a tautology, it's easy to see that $\text{TR}_{\Box}$ implies TR. The proof that TR implies $\text{TR}_{\Box}$ is as follows. To prove $\text{TR}_{\Box}$, we let σ be any behavior, assume that each F_i is true for every suffix of σ, and prove that $\Box G$ is true for σ. By definition of $\Box$, we do this by showing that if τ is any suffix of σ, then G is true for τ. Since a suffix of a suffix is a suffix, every suffix of τ is a suffix of σ. Hence, each F_i is true for every suffix of τ, which by TR implies that G is true for τ, completing the proof. (If this is too confusing, write the proof in terms of $[\![G]\!]$ and the $[\![F_i]\!]$.)

When we write a proof of a temporal formula, we are proving that the formula is true for every particular behavior σ. For every step not within the scope of an ASSUME, or in the scope of only constant assumptions, this proves that the step is true for all behaviors. If the step appears in the scope of a non-constant assumption F, then we have proved the step only for behaviors σ for which F is true. If F equals $\Box G$ for some formula G, then the behaviors for which F is true include all suffixes of σ. Hence, if the only non-constant assumptions within whose scope a step appears are all of the form $\Box G$, then the step is true for all suffixes of σ. Any such step can be used as a hypothesis of a $\Vdash$ rule, which by the reasoning that led to $\text{TR}_{\Box}$ shows that the conclusion is also true for all suffixes of σ. Hence, our proofs remain sound even with ASSUME/PROOFS as long as the only non-constant assumptions equal $\Box G$ for some G, and the only temporal proof rules we use are $\Vdash$ rules.

Since $\Box\Box G$ is equivalent to G, a formula F equals $\Box G$ for some G iff F is equivalent to $\Box F$. Such a formula is called a $\Box$ *formula*. More precisely, F is a $\Box$ formula iff $\vdash (F \equiv \Box F)$ is true. Note that a constant formula is a $\Box$ formula. Since all the rules we use for temporal reasoning are $\Vdash$ rules, we have seen that we can stay out of trouble by obeying the following rule when writing proofs:

> A proof step asserting a temporal logic formula can appear in the scope of an ASSUME iff all the assumptions of the ASSUME are $\Box$ formulas.

Here are five simple rules for showing that a formula is a $\Box$ formula.

- A constant formula is a $\Box$ formula.
- $\Box F$, $\Box\Diamond F$, and $\Diamond\Box F$ are $\Box$ formulas, for any formula F.
- The conjunction and disjunction of $\Box$ formulas are $\Box$ formulas.
- If $P(i)$ is a $\Box$ formula for every i in a constant set S, then $\forall i \in S : P(i)$ and $\exists i \in S : P(i)$ are $\Box$ formulas.

These rules and the definitions of WF and SF given below imply that $\text{WF}_v(A)$ and $\text{SF}_v(A)$ are $\Box$ formulas, for any v and A.

Since $\vdash (\Box F \Rightarrow F)$ is a tautology, F is a $\Box$ formula if $\vdash (F \Rightarrow \Box F)$ is true. This observation will help you answer:

Question 17.4 Verify the five rules for $\Box$ formulas given above.

Question 17.5 Explain why $P \vdash P'$ is a valid proof rule, for any state predicate P, and why $\Box P \Rightarrow P'$ is not equivalent to any legal TLA$^+$ formula.

? ← → C I S

?
←
→
C
I
S

17.3 Rules for Proving Safety

The most fundamental safety property is invariance. The assertion that a state predicate P is an invariant of a specification *Spec* is expressed in temporal logic by the formula $\mathit{Spec} \Rightarrow \Box P$. The formula is proved by finding an inductive invariant$^{\Box}$ *Inv* that implies P. The proof is based on the following proof rule:

$$\text{INV1:}\quad \frac{\mathit{Inv} \land [\mathit{Next}]_{\mathit{vars}} \Rightarrow \mathit{Inv}'}{\mathit{Inv} \land \Box[\mathit{Next}]_{\mathit{vars}} \Rightarrow \Box \mathit{Inv}}$$

If *Spec* equals $\mathit{Init} \land \Box[\mathit{Next}]_{\mathit{vars}}$, possibly conjoined with a liveness property, then the proof looks like this:

THEOREM: $\mathit{Spec} \Rightarrow P$

1. $\mathit{Init} \Rightarrow \mathit{Inv}$

2. $\mathit{Inv} \land \Box[\mathit{Next}]_{\mathit{vars}} \Rightarrow \mathit{Inv}'$

   2.1. It suffices to prove $\mathit{Inv} \land \mathit{Next} \Rightarrow \mathit{Inv}'$

   PROOF: $[\mathit{Next}]_{\mathit{vars}}$ equals $\mathit{Next} \lor \text{UNCHANGED } \mathit{vars}$ and

   $$\mathit{Inv} \land \text{UNCHANGED } \mathit{vars} \Rightarrow \mathit{Inv}'$$

   is obvious because the tuple *vars* of variables contains all the variables that occur in *Inv*.

   ⋮

3. $\mathit{Inv} \Rightarrow P$

4. Q.E.D.

   PROOF: By the definition of *Spec*, steps 1–3, and rule INV1.

Here is a more complete proof of the Q.E.D. step, which we would normally not bother to write.

4. Q.E.D.

   4.1. $\mathit{Inv} \land \Box[\mathit{Next}]_{\mathit{vars}} \Rightarrow \Box \mathit{Inv}$

   PROOF: By step 2 and rule INV1.

   4.2. $\mathit{Init} \land \Box[\mathit{Next}]_{\mathit{vars}} \Rightarrow \Box \mathit{Inv}$

   PROOF: By steps 1 and 4.1.

   4.3. $\Box \mathit{Inv} \Rightarrow \Box P$

   PROOF: By step 3 and the proof rule $(F \Rightarrow G) \vdash (\Box F \Rightarrow \Box G)$.

   4.4. Q.E.D.

   PROOF: By 4.1–4.3 and the definition of *Spec*.

The other kind of safety property that we prove is that one specification implements the safety part of another specification under a refinement mapping. As explained in Section 6.8$^{\Box}$, such a property is expressed by a formula of the form:

$$I \land \Box[M]_v \;\Rightarrow\; J \land \Box[N]_w$$

To prove it, we prove the following properties for a suitable invariant *Inv*

?
←
→
C
I
S

$$\begin{aligned} & I \Rightarrow J \\ & I \land \Box[M]_v \Rightarrow \Box \mathit{Inv} \\ & \mathit{Inv} \land \mathit{Inv}' \land [M]_v \Rightarrow [N]_w \end{aligned}$$

and apply this proof rule:

$$\text{INV2:}\quad \frac{\mathit{Inv} \land \mathit{Inv}' \land [M]_v \;\Rightarrow\; [N]_w}{\Box \mathit{Inv} \land \Box[M]_v \;\Rightarrow\; \Box[N]_w}$$

plus simple propositional reasoning.

Question 17.6 Show that rule INV2 is valid.

17.4 Leads To

The temporal operator $\leadsto$, read as *leads to*, is defined by

$$F \leadsto G \;\triangleq\; \Box(F \Rightarrow \Diamond G)$$

Thus, $F \leadsto G$ is true of a behavior σ iff, for every suffix τ of σ, if F is true for τ then G is true for some suffix of τ. In other words, it asserts that if F ever becomes true, then G will be true then or later. We type $\leadsto$ as `~>` in TLA$^+$.

The formula $F \leadsto G$ is a liveness property, and the $\leadsto$ operator is fundamental in reasoning about liveness. A system satisfies a liveness property because the system's fairness property implies certain elementary leads-to properties—ones that assert that if F is true, then a system step must eventually occur that makes G true. We will see how to prove such elementary leads-to properties below, when we examine fairness. Here, we consider how to deduce leads-to properties from safety properties and other leads-to properties.

Note that $F \leadsto G$ is a $\Box$ property, since by definition it is of the form $\Box H$. Also, the proof rule $H \Vdash \Box H$ and the tautology $G \Rightarrow \Diamond G$ imply the proof rule $(F \Rightarrow G) \Vdash (F \leadsto G)$.

17.4.1 The Leads-To Induction Rule

?
←
→
C
I
S

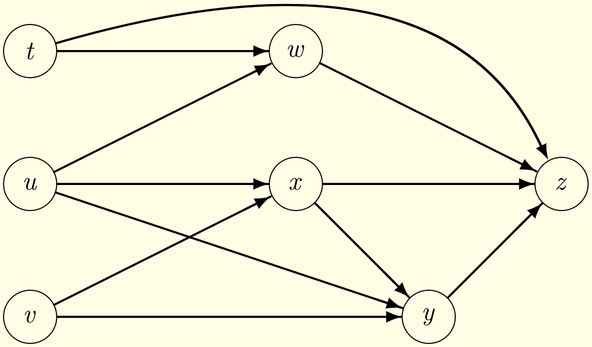
The following tautology, which asserts the transitivity of $\leadsto$, allows us to deduce leads-to properties from simpler leads-to properties:

$$(F \leadsto G) \land (G \leadsto H) \Rightarrow (F \leadsto H)$$

By induction, we can generalize this to

$$(F_n \leadsto F_{n-1}) \land (F_{n-1} \leadsto F_{n-2}) \land \ldots \land (F_1 \leadsto F_0) \Rightarrow (F_n \leadsto F_0)$$

We can further generalize this as follows. Suppose we have a finite directed graph with no cycles that contains a single sink node. Let's take it to be the following graph that has the sink z; we'll generalize later.



Suppose that we start with a token on any of the nodes and move it according to the following rule: if the token is not on the sink node, then it must eventually move along one of the edges from its current node to another node. For example, if the token is on node x, it must eventually move to node y or node z. Obviously, the token must eventually end up on the sink node.

Now, let's assign to each node n a formula F_n. In the token game, let's interpret the token being on node n to mean that F_n is true. The rule about how tokens must move corresponds to the assumption that if F_n is ever true, then F_q must eventually become true for some node pointed to by an edge from n. In other words, we assume that F_n leads to the disjunction of all formulas F_q for which there is an edge from n to q. The conjunction of all those assumptions is the formula:

$$(F_t \leadsto (F_w \lor F_z)) \land (F_u \leadsto (F_w \lor F_x \lor F_y)) \land \ldots \land (F_y \leadsto F_z)$$

We observed that if the token is placed on any node, then it eventually reaches the sink node. Therefore, if we start with at least one formula F_t true, then eventually F_z must be true. In other words, the formula we are assuming implies

$$(F_t \lor F_u \lor \ldots \lor F_z) \leadsto F_z$$

A little thought shows that we can generalize this to any directed graph satisfying two conditions: (i) it has no infinite path starting from any node (which implies that there are no cycles) and (ii) it has a unique sink.

Given a graph with a set $\mathcal{N}$ of nodes, let's define the relation $\succ$ by letting $m \succ n$ be true iff there is an edge from m to n. Condition (i) can then be expressed by the assertion that there is no "infinite chain"

$$n_1 \succ n_2 \succ n_3 \succ \ldots$$

?
←
→
C
I
S

of elements in $\mathcal{N}$. The relation $\succ$ is said to be *well-founded* on the set $\mathcal{N}$ iff this condition holds.

For any element n of $\mathcal{N}$, define $\mathcal{N}_{n\,\succ}$ to be the set of all elements m of $\mathcal{N}$ such that $n \succ m$. Our token-game assumption is that, from any node n other than z, the token eventually moves to some node in $\mathcal{N}_{n\,\succ}$. The corresponding temporal property for node n is $F_n \leadsto (\exists m \in \mathcal{N}_{n\,\succ} : F_m)$. Our general rule states that, if this formula is true for all nodes other than z, then if there exists some node n with F_n true, eventually F_z is true:

> **Leads-To Induction Rule** If $\succ$ is a well-founded relation on a set $\mathcal{N}$ and $z \in \mathcal{N}$, then:
>
> $$\begin{array}{l}(\forall\, n \in \mathcal{N} \setminus \{z\} : F_n \leadsto (\exists\, m \in \mathcal{N}_{n\,\succ} : F_m)) \\ \qquad \Rightarrow \\ ((\exists\, n \in \mathcal{N} : F_n) \leadsto F_z)\end{array}$$

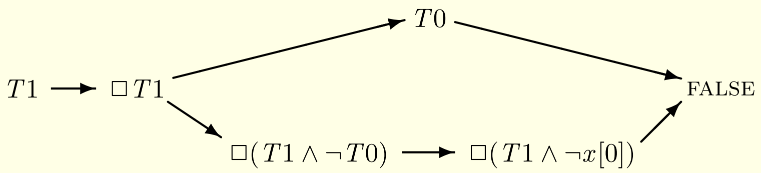
The assumption (ii) of our example, that z is the unique sink node, has disappeared from the rule. The reason is that if n is a sink node, then $\mathcal{N}_{n\,\succ}$ is the empty set, so the hypothesis $F_n \leadsto (\exists\, m \in \mathcal{N}_{n\,\succ} : F_m)$ is equivalent to $F_n \leadsto \text{FALSE}$. The proof rule $(G \Rightarrow H) \vdash (G \leadsto H)$ implies that $\text{FALSE} \leadsto F_z$ is a tautology, so $F_n \leadsto \text{FALSE}$ implies $F_n \leadsto F_z$. There is thus an implicit edge from every sink node other than z to node z. Since we don't draw any edges leading out of z, adding these implicit edges produces a graph in which z is the unique sink. While the rule is valid for any z in $\mathcal{N}$, it is always used with z the minimum element of $\mathcal{N}$ under the ordering $\succ$.

Of course, the graph has disappeared from the rule, being replaced by the well-founded relation $\succ$. Also gone is the assumption that the graph is finite. The rule is valid for infinite sets $\mathcal{N}$ as well. The argument about the token game does not depend on having a finite number of nodes, only on there not being an infinite path having a first node.

The most common example of a well-founded relation on an infinite set $\mathcal{N}$ with minimum element is the relation $>$ on the set *Nat* of natural numbers, which has minimum element 0. Another useful well-founded relation on an infinite set is lexicographical ordering on the set of k-tuples of natural numbers, for some integer $k > 0$. This ordering is defined by letting $\langle a_1, \ldots a_k \rangle \succ \langle b_1, \ldots b_k \rangle$ be true iff there is some i in $1 \,..\, k$ such that $a_i > b_i$ and $a_j = b_j$ for all j in $1 \,..\, (i-1)$. This is a well-founded ordering with minimum element $\langle 0, \ldots, 0 \rangle$.

?
←
→
C
I
S

When explaining a use of the Leads-To Induction Rule in a proof, it usually helps to draw the directed graph of the relation $\succ$ with the actual formulas F_n as the nodes. For example, here is the graph used in a proof of $T1 \leadsto \text{FALSE}$.



The graph shows that $T1 \leadsto \text{FALSE}$ follows from these formulas:

$$
\begin{array}{ll}
T1 \leadsto \Box T1 & \Box(T1 \land \neg T0) \leadsto \Box(T1 \land \neg x[0]) \\
T0 \leadsto \text{FALSE} & \Box(T1 \land \neg x[0]) \leadsto \text{FALSE} \\
\Box T1 \leadsto T0 \lor \Box(T1 \land \neg T0) &
\end{array}
$$

I call such a graph a *proof graph*. It's clear that the token game, starting with the token on the $T1$ node, shows that these $\leadsto$ formulas imply $T1 \leadsto \text{FALSE}$. This follows from the Leads-To Induction Rule, the transitivity of $\leadsto$, and the observation that for any $m \in \mathcal{N}$, the proof rule $(G \Rightarrow H) \vdash (G \leadsto H)$ implies $F_m \leadsto (\exists\, n \in \mathcal{N} : F_n)$.

The Leads-To Induction Rule was originally called the *Lattice Rule*, which was a misleading name because the rule has nothing to do with what mathematicians call lattices. The following question explains why I prefer to call it Leads-To Induction.

Question 17.7 The validity of the Leads-To Induction Rule depends only on the transitivity of $\leadsto$. Ordinary implication is also transitive. Therefore, the rule obtained from Leads-To Induction by replacing $\leadsto$ with $\Rightarrow$ is also valid. Show that the following substitutions then produce the rule for ordinary mathematical induction:

$$F_n \leftarrow \neg F_n \qquad \mathcal{N} \leftarrow \mathit{Nat} \qquad \succ \leftarrow > \qquad z \leftarrow 0$$

Question 17.8 Define a TLA$^+$ operator *LeadsToInduction* so that ANSWER

$$\mathit{LeadsToInduction}(F, \mathcal{N}, \succ, z)$$

expresses the Leads-To Induction Rule, writing $F(n)$ instead of F_n and $\mathit{LTSet}(\mathcal{N}, \succ)$ instead of $\mathcal{N}_{n\succ}$. You should include the definition of *LTSet* and of a well-founded relation with minimum element.

17.4.2 The $\Box \leadsto$ Rule

The formula $\Box F \land G \leadsto H$ is true of a behavior σ iff, for every suffixe τ of σ, if $\Box F$ and G are true of τ, then H is true of some suffix ρ of τ. By definition of $\Box$, if $\Box F$ is true of τ then it is also true of its suffix ρ. Therefore, the following formula is a tautology.

$$(\Box F \land G \leadsto H) \;\Rightarrow\; (\Box F \land G \leadsto \Box F \land H)$$

?
←
→
C
I
S

Suppose we're proving a formula $(\Box P \land Q) \leadsto R$ by Leads-To Induction using a graph having $\Box P \land Q$ as its only source node. This tautology implies that we can let $\Box P$ be a conjunct of all the formulas in the proof graph. Instead of explicitly making it a conjunct, we can simply assume $\Box P$ in the proof of $Q \leadsto R$, usually employing this proof structure:

$\langle i \rangle j.\ (\Box P \land Q) \leadsto R$

  $\langle i+1 \rangle 1.$ SUFFICES ASSUME: $\Box P$
                  PROVE: $Q \leadsto R$

The justification for the SUFFICES step is this proof rule:

$$\Box \leadsto \text{Rule:}\quad \frac{\Box P \;\Rightarrow\; (Q \leadsto R)}{\Box P \land Q \;\leadsto\; \Box P \land R}$$

For a short Leads-To Induction proof, we may not bother with this rule and just conjoin $\Box P$ to the formulas in the proof graph.

Question 17.9 Prove the validity of the $\Box \leadsto$ Rule. Show by an example that the corresponding implication

$$(\Box P \Rightarrow (Q \leadsto R)) \;\Rightarrow\; (\Box P \land Q \leadsto \Box P \land R)$$

is not a tautology.

17.4.3 Proving $\leadsto$ Formulas by Contradiction

Proofs by contradiction are nice because they allow us to use an additional hypothesis—namely, the negation of the formula we are trying to prove. This is a very strong hypothesis because, if what we are trying to prove is true, then the hypothesis is equivalent to FALSE, which is the strongest possible hypothesis.

A proof by contradiction of a $\leadsto$ formula is based on the tautology:

$$F \leadsto (G \lor (F \land \Box \neg G))$$

This is a tautology because, at any point in an execution, either G eventually becomes true or it is always false. Hence, to prove $F \leadsto G$, it suffices to prove that $(F \land \Box \neg G)$ can never be true. We do that by proving $(F \land \Box \neg G) \leadsto$ FALSE, since only FALSE can lead to FALSE. In other words, a proof by contradiction of $F \leadsto G$ uses this tautology:

$$((F \wedge \Box \neg G) \leadsto \text{FALSE}) \;\Rightarrow\; (F \leadsto G)$$

To prove $(F \wedge \Box \neg G) \leadsto \text{FALSE}$, we often use the $\Box \leadsto$ Proof Rule, which implies that it suffices to assume $\Box \neg G$ and prove $F \leadsto \text{FALSE}$.

?
←
→
C
I
S

If this section on Temporal Logic has been a detour, you may now want to return to Section 7.6.

17.5 Fairness

The concept of weak fairness of an action was introduced informally in Section 6.4□ and made more precise in Section 6.7.2□. We now formally define weak and strong fairness. To do this, we must first introduce the ENABLED operator.

17.5.1 The ENABLED Operator

The TLA$^+$ primitive operator ENABLED is defined so that, for an action A, the formula ENABLED A is a state predicate asserting that A is enabled. Action A is enabled in a state s iff there exists a state t such that $s \rightarrow t$ is an A step. Thus, ENABLED A is defined semantically by:

$$[\![\text{ENABLED } A]\!](s) \;\equiv\; \exists\, t : [\![A]\!](\langle s, t\rangle)$$

Suppose A contains no user-defined operators, no ENABLED operator, and no "$\cdot$" operator. We can then construct ENABLED A as follows:

What is the $\cdot$ operator?

- Let $v_1, \ldots, v_n$ be all the distinct variables that occur primed in A, and let $\widehat{v_1}, \ldots, \widehat{v_n}$ be n identifiers that do not occur in A.
- Let $\widehat{A}$ be the formula obtained from A by replacing each occurrence of v_i' by $\widehat{v_i}$, for each variable v_i.
- Then ENABLED A is the formula $\exists\, \widehat{v_1}, \ldots, \widehat{v_n} : \widehat{A}$.

For an arbitrary action A, let the *full expansion* of A be the action obtained by recursively expanding all definitions of user-defined symbols in A. We can compute ENABLED A by applying the procedure above to the full expansion of A. If there are nested ENABLED operators, the procedure is applied in a bottom-up fashion (constructing the innermost ENABLED formulas first). How to handle occurrences of the "$\cdot$" operator is left as an exercise for the reader.

For an action A defined as part of the definition of a next-state relation, ENABLED A can usually be computed using the following rules, where A, B, and the $A(i)$ are actions, P is a state predicate, *exp* is a state function (an expression with no primed variables), and v is a variable.

E1. $\textsc{enabled}\ (P \land A) \ \equiv\ P \land (\textsc{enabled}\ A)$

E2. $\textsc{enabled}\ (A \lor B) \ \equiv\ (\textsc{enabled}\ A) \lor (\textsc{enabled}\ B)$

E3. $\textsc{enabled}\ (\exists\, i \in \mathit{exp} : A(i)) \ \equiv\ \exists\, i \in \mathit{exp} : \textsc{enabled}\ A(i)$

E4. If the full expansions of A and B have no primed variable in common, then

$$\textsc{enabled}\ (A \land B) \ \equiv\ (\textsc{enabled}\ A) \land (\textsc{enabled}\ B)$$

?
←
→
C
I
S

E5. $\textsc{enabled}\ (v' = \mathit{exp}) \ \equiv\ \text{TRUE}$

E6. $\textsc{enabled}\ (v' \in \mathit{exp}) \ \equiv\ \mathit{exp} \neq \{\}$

For example, if x and y are variables, then:

$$\begin{array}{lll}
\multicolumn{2}{l}{\textsc{enabled}\ ((x \subseteq \{y, y+1\}) \land (x' = \{\}) \land (y' \in x))} & \\
\equiv & (x \subseteq \{y, y+1\}) \land \textsc{enabled}\ ((x' = \{\}) \land (y' \in x)) & \text{by E1} \\
\equiv & (x \subseteq \{y, y+1\}) \land \textsc{enabled}\ (x' = \{\}) \land \textsc{enabled}\ (y' \in x) & \text{by E4} \\
\equiv & (x \subseteq \{y, y+1\}) \land (x \neq \{\}) & \text{by E5 and E6}
\end{array}$$

Unfortunately, fairness involves not $\textsc{enabled}\ A$ for such an action A, but rather $\textsc{enabled}\ \langle A \rangle_v$, where $\langle A \rangle_v$ is defined to equal $A \land (v' \neq v)$ and v is usually the tuple of all specification variables. The rules above do not allow us to compute such an $\textsc{enabled}$ expression.

In some cases, the action A does not permit stuttering steps, so A and $\langle A \rangle_v$ are equivalent. If they are, then we can apply the proof rule

$$(A \equiv B) \vdash ((\textsc{enabled}\ A) \equiv (\textsc{enabled}\ B))$$

to deduce that $\textsc{enabled}\ \langle A \rangle_v$ equals the more easily computed formula $\textsc{enabled}\ A$. It's rarely the case that A and $\langle A \rangle_v$ are equivalent on all steps $s \rightarrow t$. However, they often are equivalent on steps $s \rightarrow t$ in which s satisfies a type invariant. This rule and E1 implies that the following rule is true for any actions A and B and any state predicate P.

E7. $(P \Rightarrow (A \equiv B)) \vdash (P \Rightarrow ((\textsc{enabled}\ A) \equiv (\textsc{enabled}\ B))$

We most often apply this rule with B equal to $\langle A \rangle_v$ and P an invariant that implies type-correctness. Note that E7 is an ordinary proof rule, not a $\Vdash$ rule.

Question 17.10 Explain why the formula ANSWER

$$(A \equiv B) \ \Rightarrow\ ((\textsc{enabled}\ A) \equiv (\textsc{enabled}\ B))$$

and the rule

$$(A \equiv B) \ \Vdash\ ((\textsc{enabled}\ A) \equiv (\textsc{enabled}\ B))$$

are not necessarily true.

17.5.2 Weak Fairness

?
←
→
C
I
S

We saw in Section 6.7.2□ that the formula $\mathrm{WF}_v(A)$ asserts of a behavior σ that σ does not contain a suffix in which an $\langle A \rangle_v$ step is always enabled but never occurs. Since $\langle A \rangle_v$ is defined to equal $A \wedge (v' \neq v)$, if v is the tuple of all variables in a system specification, then an $\langle A \rangle_v$ step is a non-stuttering A step. Recall that $[A]_v$ is defined to equal $A \vee (v' = v)$. This calculation

$$\begin{aligned}
\langle A \rangle_v &\equiv A \wedge (v' \neq v) && \text{by definition of } \langle \ldots \rangle_v \\
&\equiv \neg\neg(A \wedge (v' \neq v)) && \text{because } \neg\neg F \equiv F \\
&\equiv \neg(\neg A \vee (v' = v)) && \text{because } \neg(F \wedge G) \equiv (\neg F \vee \neg G) \\
&\equiv \neg[\neg A]_v && \text{by definition of } [\ldots]_v
\end{aligned}$$

shows that the operators $\langle \ldots \rangle_v$ and $[\ldots]_v$ bear the same relation to each other that $\Diamond$ and $\Box$ do—that is, $\langle A \rangle_v \equiv \neg[\neg A]_v$ and $\Diamond F \equiv \neg\Box\neg F$. This relation is sometimes called *duality*. These two duality relations imply

$$\Diamond\langle A \rangle_v \equiv \neg\Box[\neg A]_v$$

for any action A and state expression v.

We now express weak fairness in terms of temporal operators and ENABLED. The formula $\mathrm{WF}_v(A)$ asserts of a behavior σ that there is not a suffix τ of σ such that ENABLED $\langle A \rangle_v$ is true in every state of τ and there is no $\langle A \rangle_v$ step in τ. "ENABLED $\langle A \rangle_v$ is true in every state of τ" is expressed by $\tau \vdash \Box(\textsc{enabled}\ \langle A \rangle_v)$; and "there is no $\langle A \rangle_v$ step in τ" is expressed by $\tau \vdash \neg\Diamond\langle A \rangle_v$. Therefore, $\mathrm{WF}_v(A)$ can be written as the following formula:

$$\neg\Diamond(\Box(\textsc{enabled}\ \langle A \rangle_v) \wedge \neg\Diamond\langle A \rangle_v)$$

This formula probably looks inscrutable to you. However, like all mathematical formulas, we can understand it a piece at a time. Moreover, we can use tautologies to simplify it as follows:

$$\begin{aligned}
&\neg\Diamond(\Box(\textsc{enabled}\ \langle A \rangle_v) \wedge \neg\Diamond\langle A \rangle_v) \\
&\quad \equiv \Box\neg((\Box\textsc{enabled}\ \langle A \rangle_v) \wedge \neg\Diamond\langle A \rangle_v) && \text{by } \neg\Diamond F \equiv \Box\neg F \\
&\quad \equiv \Box((\neg\Box\textsc{enabled}\ \langle A \rangle_v) \vee \Diamond\langle A \rangle_v) && \text{by } \neg(F \wedge G) \equiv (\neg F \vee \neg G) \text{ and } \neg\neg F \equiv F \\
&\quad \equiv \Box(\Diamond(\neg\textsc{enabled}\ \langle A \rangle_v) \vee \Diamond\langle A \rangle_v) && \text{by } \neg\Box F \equiv \Diamond\neg F \\
&\quad \equiv \Box\Diamond((\neg\textsc{enabled}\ \langle A \rangle_v) \vee \langle A \rangle_v) && \text{by } \Diamond F \vee \Diamond G \equiv \Diamond(F \vee G) \\
&\quad \equiv \Box\Diamond(\neg\textsc{enabled}\ \langle A \rangle_v) \vee \Box\Diamond\langle A \rangle_v && \text{by } \Box\Diamond(F \vee G) \equiv (\Box\Diamond F \vee \Box\Diamond G)
\end{aligned}$$

We can therefore define:

$$\mathrm{WF}_v(A) \;\triangleq\; \Box\Diamond\neg\textsc{enabled}\ \langle A \rangle_v \;\vee\; \Box\Diamond\langle A \rangle_v$$

Thus, $\mathrm{WF}_v(A)$ asserts of a behavior that either $\langle A \rangle_v$ is infinitely often disabled (not enabled), or there are infinitely many $\langle A \rangle_v$ steps.

Question 17.11 Show that $\mathrm{WF}_v(A)$ is equivalent to each of the following two formulas:

$$\begin{array}{l}
\Diamond\Box\,(\textsc{enabled}\ \langle A\rangle_v) \;\Rightarrow\; \Box\Diamond\langle A\rangle_v \\
\neg\,\Diamond\,(\Box(\textsc{enabled}\ \langle A\rangle_v) \wedge \Box[\neg A]_v)
\end{array}$$

?
←
→
C
I
S

17.5.3 Strong Fairness

The weak fairness formula $\mathrm{WF}_v(A)$ asserts of a behavior σ that σ does not contain a suffix in which an $\langle A\rangle_v$ step is always enabled but never occurs. The corresponding strong fairness formula $\mathrm{SF}_v(A)$ asserts of a behavior σ that σ does not contain a suffix in which an $\langle A\rangle_v$ step is infinitely often enabled but never occurs. Since *always true* is stronger than (implies) *infinitely often true*, the condition that something *isn't* true infinitely often is stronger than its not being always true. Hence, $\mathrm{SF}_v(A)$ implies $\mathrm{WF}_v(A)$, so strong fairness is stronger than weak fairness. *Doesn't contain a suffix satisfying F* is expressed in temporal logic by $\neg\Diamond F$, and *infinitely often* is expressed as $\Box\Diamond$. Hence $\mathrm{SF}_v(A)$ can be written as

$$\neg\Diamond(\Box\Diamond(\textsc{enabled}\ \langle A\rangle_v) \wedge \neg\Diamond\langle A\rangle_v$$

We can use tautologies to simplify this formula as follows:

$$\begin{array}{lll}
\multicolumn{3}{l}{\neg\Diamond(\Box\Diamond(\textsc{enabled}\ \langle A\rangle_v) \wedge \neg\Diamond\langle A\rangle_v)} \\
\equiv & \Box\neg((\Box\Diamond\textsc{enabled}\ \langle A\rangle_v) \wedge \neg\Diamond\langle A\rangle_v) & \text{by } \neg\Diamond F \equiv \Box\neg F \\
\equiv & \Box((\neg\Box\Diamond\textsc{enabled}\ \langle A\rangle_v) \vee \Diamond\langle A\rangle_v) & \text{by } \neg(F \wedge G) \equiv (\neg F \vee \neg G) \text{ and } \neg\neg F \equiv F \\
\equiv & \Box(\Diamond\Box(\neg\textsc{enabled}\ \langle A\rangle_v) \vee \Diamond\langle A\rangle_v) & \text{by } \neg\Box\Diamond F \equiv \Diamond\Box\neg F \\
\equiv & \Box\Diamond\Box(\neg\textsc{enabled}\ \langle A\rangle_v) \vee \Box\Diamond\langle A\rangle_v) & \text{by } \Box(\Diamond F \vee \Diamond G) \equiv \Box\Diamond F \vee \Box\Diamond G \\
\equiv & \Diamond\Box(\neg\textsc{enabled}\ \langle A\rangle_v) \vee \Box\Diamond\langle A\rangle_v & \text{by } \Diamond\Box\Diamond F \equiv \Box\Diamond F
\end{array}$$

We can therefore define:

$$\mathrm{SF}_v(A) \;\triangleq\; \Diamond\Box\neg\textsc{enabled}\ \langle A\rangle_v \;\vee\; \Box\Diamond\langle A\rangle_v$$

Observe that replacing $\Diamond\Box\neg\textsc{enabled}\ \langle A\rangle_v$ by the weaker formula $\Box\Diamond\neg\textsc{enabled}\ \langle A\rangle_v$ in this formula yields the (weaker) formula $\mathrm{WF}_v(A)$.

Question 17.12 Show that the following formulas are all tautologies, for any temporal formulas F and G:

$$\neg\Box\Diamond F \equiv \Diamond\Box\neg F \qquad \Diamond\Box\Diamond F \equiv \Box\Diamond F \qquad \Diamond\Box F \Rightarrow \Box\Diamond F \qquad \Box(\Diamond F \vee \Diamond G) \equiv \Box\Diamond F \vee \Box\Diamond G$$

?
←
→
C
I
S

17.5.4 Proving $\leadsto$ Properties with Fairness

Section 17.4 gives rules for deriving $\leadsto$ from other $\leadsto$ properties and safety properties. However, except for the trivial rule $(F \Rightarrow G) \vdash (F \leadsto G)$ (which essentially says that something happening now happens eventually), it provides no rule for deriving $\leadsto$ properties without starting from $\leadsto$ properties. Elementary $\leadsto$ properties are derived from fairness assumptions.

To prove $P \leadsto Q$ for state predicates P and Q, we must show that if P ever becomes true, then Q will eventually becomes true. We use $\mathrm{WF}_v(A)$ to prove this by contradiction by showing that if P is true and Q never becomes true, then an $\langle A \rangle_v$ step must eventually make Q true. This in turn follows from these three conditions:

- Once true, P remains true until Q becomes true.
- Any $\langle A \rangle_v$ step starting in a state with P true makes Q true.
- If P remains true forever, then an $\langle A \rangle_v$ step must eventually occur.

Assuming a next-state action $\mathcal{N}$, so every step is a $[\mathcal{N}]_v$ step, the first two conditions are expressed as:

$$\begin{array}{l}(1) P \land [\mathcal{N}]_v \Rightarrow (P' \lor Q') \\ (2) P \land \langle A \rangle_v \Rightarrow Q'\end{array}$$

The third condition is implied by $\mathrm{WF}_v(A)$ and:

$$(3) P \Rightarrow \textsc{enabled}\ \langle A \rangle_v$$

Since we assume every non-stuttering step is an $\mathcal{N}$ step, we can strengthen (2) by replacing $\langle A \rangle_v$ with $\langle \mathcal{N} \land A \rangle_v$. (In almost all applications, $\mathcal{N} \land A$ will equal A.) This gives us the following proof rule:

$$\text{WF1.} \quad \begin{array}{l} \quad P \land [\mathcal{N}]_v \Rightarrow (P' \lor Q') \\ \quad P \land \langle \mathcal{N} \land A \rangle_v \Rightarrow Q' \\ \quad P \Rightarrow \textsc{enabled}\ \langle A \rangle_v \\ \hline\hline \Box[\mathcal{N}]_v \land \mathrm{WF}_v(A) \Rightarrow (P \leadsto Q) \end{array}$$

To apply this rule, we have to be able to prove its hypotheses. We will seldom be able to prove the hypotheses without additional assumptions unless P rules out states in which variables have values "of the wrong type". We could let P have a conjunct I that is an invariant asserting type-correctness and other useful properties. Instead, we usually apply WF1 in a proof in the scope of the assumption $\Box I$, so we can assume I and I' in the proof of these hypotheses. (Of course, I' would be useful only in the proof of the first two.)

There is an analogous rule to WF1 for strong fairness. With the hypothesis $\mathrm{SF}_v(A)$, to deduce that an $\langle A \rangle_v$ step eventually occurs, it suffices to show that

$\langle A \rangle_v$ is infinitely often enabled. We can therefore weaken the third hypothesis of WF1 to obtain:

$$\text{SF1.}\quad \begin{array}{l} P \land [\mathcal{N}]_v \Rightarrow (P' \lor Q') \\ P \land \langle \mathcal{N} \land A \rangle_v \Rightarrow Q' \\ \Box P \land \Box[\mathcal{N}]_v \Rightarrow \Diamond \text{ENABLED } \langle A \rangle_v \\ \hline\hline \quad \Box[\mathcal{N}]_v \land \text{SF}_v(A) \Rightarrow (P \leadsto Q) \end{array}$$

?
←
→
C
I
S

As with rule WF1, we usually need an assumption $\Box I$ to prove the hypotheses, for a suitable invariant I. We often need an additional assumption $\Box F$ to prove the third hypothesis, for some temporal formula F. Equivalently, we can use the following generalization of SF1.

$$\text{SF1a.}\quad \begin{array}{l} P \land [\mathcal{N}]_v \Rightarrow (P' \lor Q') \\ P \land \langle \mathcal{N} \land A \rangle_v \Rightarrow Q' \\ \Box P \land \Box[\mathcal{N}]_v \land \Box F \Rightarrow \Diamond \text{ENABLED } \langle A \rangle_v \\ \hline\hline \quad \Box[\mathcal{N}]_v \land \text{SF}_v(A) \land \Box F \Rightarrow (P \leadsto Q) \end{array}$$

Question 17.13 Use semantic reasoning to show the soundness of rule SF1. HINT

17.5.5 Proving Fairness

As we saw in Section 6.8□, to prove that a specification $Spec_H$ implements a specification $Spec_A$ under a refinement mapping, we must prove $Spec_H \Rightarrow \overline{Spec_A}$, where $\overline{F}$ is the formula obtained from a formula F by substituting, for each variable x of $Spec_A$ the formula $\overline{x}$. Formula $Spec_A$ usually has the form $Init_A \land \Box[Next_A]_w \land \mathcal{F}$, where w is the tuple of all the specification's variables and $\mathcal{F}$ is the conjunction of fairness formulas. In that case, $\overline{Spec_A}$ equals $\overline{Init_A} \land \Box[\overline{Next_A}]_{\overline{w}} \land \overline{\mathcal{F}}$. We saw in Section 17.3 how to prove the $Spec_H$ implies $\overline{Init_A} \land \Box[\overline{Next_A}]_{\overline{w}}$. We now see how to prove that it implies $\overline{\mathcal{F}}$.

If $\mathcal{F}$ is the conjunction of fairness properties, then we can prove those properties one and a time. We therefore need to know only how to prove $Spec_H \Rightarrow \overline{\mathcal{F}}$ when $\mathcal{F}$ is a weak or strong fairness formula. We begin with weak fairness.

Recall that since barring a formula means substituting for its variables, barring distributes over most mathematical operators, including $\lor$, $\neg$, $\Box$, and $\Diamond$. For example, $\overline{\Box\Diamond P}$ equals $\Box\Diamond\overline{P}$. Since $\text{WF}_w(B)$ equals $\Box\Diamond\neg\text{ENABLED } \langle B \rangle_w \lor \Box\Diamond\langle B \rangle_w$, this implies

$$\overline{\text{WF}_w(B)} \quad \equiv \quad \Box\Diamond\neg\overline{\text{ENABLED } \langle B \rangle_w} \lor \Box\Diamond\langle \overline{B} \rangle_{\overline{w}}$$

We cannot simplify this formula further because $\overline{\text{ENABLED } \langle B \rangle_w}$ need not equal Why not?

$\text{ENABLED}\ \overline{\langle B\rangle_w}$.

The only mathematical operators you are likely to use that barring does not distribute over are ENABLED and the operators WF and SF defined in terms of it. (The only other TLA$^+$ operator that barring doesn't distribute over is the action composition operator "$\cdot$".)

?
←
→
C
I
S

We now come to a rule for deducing $\overline{\text{WF}_w(B)}$ from a fairness property $\text{WF}_v(A)$. The complete rule, which is used in practice, is rather complicated. We will approach it gradually, starting with this simple rule:

$$\begin{array}{c} \langle A\rangle_v \land \overline{\text{ENABLED}\ \langle B\rangle_w} \ \Rightarrow\ \langle\overline{B}\rangle_{\overline{w}} \\ \overline{\text{ENABLED}\ \langle B\rangle_w} \ \Rightarrow\ \text{ENABLED}\ \langle A\rangle_v \\ \hline\hline \text{WF}_v(A) \ \Rightarrow\ \overline{\text{WF}_w(B)} \end{array}$$

To show the soundness of this rule, as explained above, we use some illegal TLA$^+$ formulas to reason about a proof rule $F_1, \ldots, F_n \ \Vdash\ G$ as if it were $\Box F_1 \land \ldots \land \Box F_n \Rightarrow G$. Here is the proof of soundness of this rule.

ASSUME: A1. $\Box(\langle A\rangle_v \land \overline{\text{ENABLED}\ \langle B\rangle_w} \Rightarrow \langle\overline{B}\rangle_{\overline{w}})$

A2. $\Box(\overline{\text{ENABLED}\ \langle B\rangle_w} \Rightarrow \text{ENABLED}\ \langle A\rangle_v)$

PROVE: $\text{WF}_v(A) \ \Rightarrow\ \overline{\text{WF}_w(B)}$

1. SUFFICES ASSUME: $\text{WF}_v(A) \ \land\ \Diamond\Box\overline{\text{ENABLED}\ \langle B\rangle_w}$

   PROVE: $\Box\Diamond\langle\overline{B}\rangle_{\overline{w}}$

   PROOF: By definition of WF, the tautology $\neg\Box\Diamond\neg F \ \equiv\ \Diamond\Box F$, and propositional logic, $\text{WF}_v(A) \ \Rightarrow\ \overline{\text{WF}_w(B)}$ is equivalent to:

   $$\text{WF}_v(A) \ \land\ \Diamond\Box\overline{\text{ENABLED}\ \langle B\rangle_w} \ \Rightarrow\ \Box\Diamond\langle\overline{B}\rangle_{\overline{w}}$$

Note that this assumption is a □ formula.

2. $\Diamond\Box\text{ENABLED}\ \langle A\rangle_v$

   PROOF: By step 1, assumption A2, and the rule

   $(F \Rightarrow G) \Vdash (\Diamond\Box F \Rightarrow \Diamond\Box G)$.

3. $\Box\Diamond\langle A\rangle_v$

   PROOF: By step 2, since $\text{WF}_v(A)$ is equivalent to

   $\Diamond\Box\text{ENABLED}\ \langle A\rangle_v \ \Rightarrow\ \Box\Diamond\langle A\rangle_v$.

4. $\Box\Diamond(\langle A\rangle_v \land \overline{\text{ENABLED}\ \langle B\rangle_w})$

   PROOF: By step 3 and the step 1 assumption, since $\Box\Diamond F \land \Diamond\Box G$ implies $\Box\Diamond(F \land G)$.

5. Q.E.D.

   PROOF: By step 4, assumption A1, and the rule

   $(F \Rightarrow G) \Vdash (\Box\Diamond F \Rightarrow \Box\Diamond G)$.

? ← → C I S

You should study this rule until you understand it intuitively. We obtain the complete rule by incrementally modifying it. We won't worry much about soundness of the intermediate rules; we will just prove that the complete rule is sound.

The first change is to introduce a next-state action $\mathcal{N}$ and tuple v of variables, so we can use $\Box[\mathcal{N}]_v$ to prove $\overline{\text{WF}_w(B)}$:

$$\begin{array}{l}
\langle \mathcal{N} \wedge A\rangle_v \wedge \overline{\textsc{enabled}\ \langle B\rangle_w} \;\Rightarrow\; \langle \overline{B}\rangle_{\overline{w}} \\
\overline{\textsc{enabled}\ \langle B\rangle_w} \;\Rightarrow\; \textsc{enabled}\ \langle A\rangle_v \\
\hline\hline
\quad \Box[\mathcal{N}]_v \wedge \text{WF}_v(A) \;\Rightarrow\; \overline{\text{WF}_w(B)}
\end{array}$$

Soundness follows from soundness of the original rule because if $\Box[\mathcal{N}]_v$ is true of a behavior σ, then a step of σ is an $\langle A\rangle_v$ step iff it is an $\langle \mathcal{N} \wedge A\rangle_v$ step.

Next, we make a modification that by itself accomplishes nothing, but will allow further transformations. We introduce a new action C that's stronger than A:

$$\begin{array}{l}
\langle \mathcal{N} \wedge C\rangle_v \;\Rightarrow\; \langle \overline{B}\rangle_{\overline{w}} \\
\langle \mathcal{N} \wedge A\rangle_v \wedge \overline{\textsc{enabled}\ \langle B\rangle_w} \;\Rightarrow\; C \\
\overline{\textsc{enabled}\ \langle B\rangle_w} \;\Rightarrow\; \textsc{enabled}\ \langle A\rangle_v \\
\hline\hline
\quad \Box[\mathcal{N}]_v \wedge \text{WF}_v(A) \;\Rightarrow\; \overline{\text{WF}_w(B)}
\end{array}$$

Soundness of this rule is obvious, since the first two hypotheses imply the first hypothesis of the previous rule.

From here on, we show the position of our successive rules within the complete rule, the parts to be added later shown in gray. The next step is to strengthen the second and third hypotheses to make use of an invariant P, adding another hypothesis that proves the invariance of P.

$$\begin{array}{l}
\langle \mathcal{N} \wedge C\rangle_v \;\Rightarrow\; \langle \overline{B}\rangle_{\overline{w}} \\
P \wedge P' \wedge \langle \mathcal{N} \wedge A\rangle_v \wedge \overline{\textsc{enabled}\ \langle B\rangle_w} \;\Rightarrow\; C \\
P \wedge \overline{\textsc{enabled}\ \langle B\rangle_w} \;\Rightarrow\; \textsc{enabled}\ \langle A\rangle_v \\
\Box[\mathcal{N} \wedge \neg C]_v \wedge \text{WF}_v(A) \wedge \Box\overline{\textsc{enabled}\ \langle B\rangle_w} \;\Rightarrow\; \Diamond\Box P \\
\hline\hline
\qquad\qquad \Box[\mathcal{N}]_v \wedge \text{WF}_v(A) \;\Rightarrow\; \overline{\text{WF}_w(B)}
\end{array}$$

Finally, we weaken the fourth hypothesis because it's not necessary that P be an invariant; we just need to show that $\Diamond\Box P$ is satisfied, which $\text{WF}_v(A)$ can help us prove. This yields our complete rule:

WF2.
$$\begin{array}{c}
\langle \mathcal{N} \wedge C \rangle_v \;\Rightarrow\; \langle \overline{B} \rangle_{\overline{w}} \\
P \wedge P' \wedge \langle \mathcal{N} \wedge A \rangle_v \wedge \overline{\text{ENABLED } \langle B \rangle_w} \;\Rightarrow\; C \\
P \wedge \overline{\text{ENABLED } \langle B \rangle_w} \;\Rightarrow\; \text{ENABLED } \langle A \rangle_v \\
\Box[\mathcal{N} \wedge \neg C]_v \wedge \text{WF}_v(A) \wedge \Box\overline{\text{ENABLED } \langle B \rangle_w} \;\Rightarrow\; \Diamond\Box P \\
\hline\hline
\Box[\mathcal{N}]_v \wedge \text{WF}_v(A) \;\Rightarrow\; \overline{\text{WF}_w(B)}
\end{array}$$

Click here for the proof of rule WF2.

As with rule SF1, we often need an additional assumption $\Box F$ to prove the last hypothesis, for some temporal formula F. We can similarly generalize WF2 to:

WF2a.
$$\begin{array}{c}
\langle \mathcal{N} \wedge C \rangle_v \;\Rightarrow\; \langle \overline{B} \rangle_{\overline{w}} \\
P \wedge P' \wedge \langle \mathcal{N} \wedge A \rangle_v \wedge \overline{\text{ENABLED } \langle B \rangle_w} \;\Rightarrow\; C \\
P \wedge \overline{\text{ENABLED } \langle B \rangle_w} \;\Rightarrow\; \text{ENABLED } \langle A \rangle_v \\
\Box[\mathcal{N} \wedge \neg C]_v \wedge \text{WF}_v(A) \wedge \Box\overline{\text{ENABLED } \langle B \rangle_w} \wedge \Box F \;\Rightarrow\; \Diamond\Box P \\
\hline\hline
\Box[\mathcal{N}]_v \wedge \text{WF}_v(A) \wedge \Box F \;\Rightarrow\; \overline{\text{WF}_w(B)}
\end{array}$$

The analogous rule to WF2 for deducing $\overline{\text{SF}_w(B)}$ from $\text{SF}_v(A)$ is:

SF2.
$$\begin{array}{c}
\langle \mathcal{N} \wedge C \rangle_v \;\Rightarrow\; \langle \overline{B} \rangle_{\overline{w}} \\
P \wedge P' \wedge \langle \mathcal{N} \wedge A \rangle_v \;\Rightarrow\; C \\
P \wedge \overline{\text{ENABLED } \langle B \rangle_w} \;\Rightarrow\; \text{ENABLED } \langle A \rangle_v \\
\Box[\mathcal{N} \wedge \neg C]_v \wedge \text{SF}_v(A) \wedge \Box\Diamond\overline{\text{ENABLED } \langle B \rangle_w} \;\Rightarrow\; \Diamond\Box P \\
\hline\hline
\Box[\mathcal{N}]_v \wedge \text{SF}_v(A) \;\Rightarrow\; \overline{\text{SF}_w(B)}
\end{array}$$

It's identical to WF2 except with SF replacing WF, with $\Box\Diamond\overline{\text{ENABLED } \langle B \rangle_w}$ replacing the stronger condition $\Box\overline{\text{ENABLED } \langle B \rangle_w}$ in the fourth hypothesis, and with the stronger second hypothesis. The proof of soundness is similar to that for WF2 and is left as an exercise for the motivated reader. Just as we generalized WF2 to WF2a to make use of an assumption $\Box F$, we can generalize SF2 to:

SF2a.
$$\begin{array}{c}
\langle \mathcal{N} \wedge C \rangle_v \;\Rightarrow\; \langle \overline{B} \rangle_{\overline{w}} \\
P \wedge P' \wedge \langle \mathcal{N} \wedge A \rangle_v \;\Rightarrow\; C \\
P \wedge \overline{\text{ENABLED } \langle B \rangle_w} \;\Rightarrow\; \text{ENABLED } \langle A \rangle_v \\
\Box[\mathcal{N} \wedge \neg C]_v \wedge \text{SF}_v(A) \wedge \Box\Diamond\overline{\text{ENABLED } \langle B \rangle_w} \wedge \Box F \;\Rightarrow\; \Diamond\Box P \\
\hline\hline
\Box[\mathcal{N}]_v \wedge \text{SF}_v(A) \wedge \Box F \;\Rightarrow\; \overline{\text{SF}_w(B)}
\end{array}$$

?
←
→
C
I
S

We now have rules to deduce $\overline{\mathrm{WF}_w(B)}$ from $\mathrm{WF}_v(A)$ and $\overline{\mathrm{SF}_w(B)}$ from $\mathrm{SF}_v(A)$. This leaves two other deductions we might want to perform:

- Deducing $\overline{\mathrm{WF}_w(B)}$ from $\mathrm{SF}_v(A)$.
- Deducing $\overline{\mathrm{SF}_w(B)}$ from $\mathrm{WF}_v(A)$.

?
←
→
C
I
S

We can do the first by using SF2 to prove $\overline{\mathrm{SF}_w(B)}$, which implies $\overline{\mathrm{WF}_w(B)}$. We can do the second if $\mathrm{WF}_v(A)$ is equivalent to $\mathrm{SF}_v(A)$ so we can use SF2, or if $\overline{\mathrm{SF}_w(B)}$ is equivalent to $\overline{\mathrm{WF}_w(B)}$ so we can use WF2. Let's first consider when $\mathrm{WF}_v(A)$ is equivalent to $\mathrm{SF}_v(A)$.

We know that $\mathrm{SF}_v(A)$ implies $\mathrm{WF}_v(A)$, so to show them equivalent, it suffices to show $\neg\mathrm{SF}_v(A)$ implies $\neg\mathrm{WF}_v(A)$. Since $\neg\Box\Diamond\langle A\rangle_v \equiv \Diamond\Box\neg\langle A\rangle_v$ and $\neg\langle A\rangle_v \equiv [\neg A]_v$, it follows from the definitions of WF and SF that:

$$\begin{array}{lcl} \neg\,\mathrm{WF}_v(A) & \equiv & \Diamond\Box\textsc{enabled}\ \langle A\rangle_v \ \land\ \Diamond\Box[\neg A]_v \\ \neg\,\mathrm{SF}_v(A) & \equiv & \Box\Diamond\textsc{enabled}\ \langle A\rangle_v \ \land\ \Diamond\Box[\neg A]_v \end{array}$$

Consider a behavior σ that satisfies $\mathrm{SF}_v(A)$. It has a suffix τ that satisfies $\Box[\neg A]_v$ and $\textsc{enabled}\ \langle A\rangle_v$. If τ satisfies

$$\textsc{enabled}\ \langle A\rangle_v \ \land\ \Box[\neg A]_v \ \Rightarrow\ \Box\textsc{enabled}\ \langle A\rangle_v$$

then it satisfies

$$\Box\textsc{enabled}\ \langle A\rangle_v \ \land\ \Box[\neg A]_v$$

which implies that σ satisfies $\neg\mathrm{WF}_v(A)$. By this argument, we have derived the following rule:

$$\begin{array}{c} \textsc{enabled}\ \langle A\rangle_v \ \land\ \Box[\neg A]_v \ \Rightarrow\ \Box\textsc{enabled}\ \langle A\rangle_v \\ \hline\hline \mathrm{WF}_v(A) \ \equiv\ \mathrm{SF}_v(A) \end{array}$$

We will apply this rule in a context in which we have a next-state action $\mathcal{N}$ and an invariant I. Propositional logic shows that $\vdash [\mathcal{N}]_v \land [\neg A]_v \equiv [\mathcal{N} \land \neg A]_v$, which implies $\vdash \Box[\mathcal{N}]_v \land \Box[\neg A]_v \equiv \Box[\mathcal{N} \land \neg A]_v$. The proof rule above therefore implies

$$\begin{array}{c} \textsc{enabled}\ \langle A\rangle_v \ \land\ \Box I \ \land\ \Box[\mathcal{N} \land \neg A]_v \ \Rightarrow\ \Box\textsc{enabled}\ \langle A\rangle_v \\ \hline\hline \Box I \ \land\ \Box[\mathcal{N}]_v \ \Rightarrow\ (\mathrm{WF}_v(A) \ \equiv\ \mathrm{SF}_v(A)) \end{array}$$

Exactly the same reasoning used above proves the corresponding rule for $\overline{\mathrm{SF}_w(B)}$ and $\overline{\mathrm{WF}_w(B)}$:

$$\begin{array}{c} \overline{\textsc{enabled}\ \langle B\rangle_w} \ \land\ \Box[\neg\overline{B}]_{\overline{w}} \ \Rightarrow\ \Box\overline{\textsc{enabled}\ \langle B\rangle_w} \\ \hline\hline \overline{\mathrm{WF}_v(B)} \ \equiv\ \overline{\mathrm{SF}_v(B)} \end{array}$$

We can use a next-state action $\mathcal{N}$ and invariant I with the aid of this tautology:

$$(\text{UNCHANGED } v \Rightarrow \text{UNCHANGED } \overline{w}) \;\Rightarrow\; ([\mathcal{N} \land \neg\overline{B}]_v \Rightarrow [\mathcal{N}]_v \land [\neg\overline{B}]_{\overline{w}})$$

This tautology and the preceding proof rule imply the soundness of:

$$\begin{array}{c} \text{UNCHANGED } v \Rightarrow \text{UNCHANGED } \overline{w} \hspace{20em} \\ \overline{\text{ENABLED } \langle B\rangle_w} \land \Box I \land \Box[\mathcal{N} \land \neg\overline{B}]_{\overline{w}} \;\Rightarrow\; \Box\overline{\text{ENABLED } \langle B\rangle_w} \\ \hline\hline \Box I \land \Box[\mathcal{N}]_v \;\Rightarrow\; (\overline{\text{WF}_v(B)} \equiv \overline{\text{SF}_v(B)}) \end{array}$$

When using rule WF1, WF2, SF1, or SF2, I find it helpful to prove a step that corresponds to the rule's conclusion with a sequence of substeps that correspond to the rule's hypotheses followed by a QED step whose BY clause mentions those substeps and the rule. When writing a formal TLA$^+$ proof to be checked by TLAPS, there's no need to mention the rule (except perhaps in a comment) because the TLP backend will deduce the rule by itself. I like to use local definitions that make it clear what formulas are being substituted for what identifiers in the rule. For example, if I'm using rule SF2, I will define P to equal the formula to be substituted for P in the rule. Not only does this make it easier to understand the proof, but it ensures that I've substituted exactly the same formula for each instance of P. The TLP backend will probably not be able to check the proof if I were to substitute $P_1 \land P_2$ for P in one hypothesis and $P_2 \land P_1$ for it in another. In fact, for complicated formulas, you may have to hide these local definitions for TLP to prove the QED step.

?
←
→
C
I
S