pressron

TLA$^+$ in Practice and Theory
Part 2: The + in TLA$^+$

01 Jun 2017

This is part 2 in a four-part series. A new post will be published every Thursday. ***Part 1***

> *Writing is nature's way of letting you know how sloppy your thinking is.*
>
> Dick Guindon

> *Mathematics is nature's way of letting you know how sloppy your writing is.*
>
> *…Formal mathematics is nature's way of letting you know how sloppy your mathematics is.*
>
> Leslie Lamport, *Specifying Systems*

In **part 1** we discussed the motivation, use and general principles guiding the design of TLA$^+$. We now turn to study the details of the language. We begin with the language elements used to describe the static state of a computation, namely the program's data and data operations, that form the primitive building blocks of computation. We will call this part of TLA$^+$ the *data logic*.

Describing Data and Operations with Math

TLA$^+$ uses the temporal logic of actions, TLA, to describe computation as a discrete dynamical system, which is analogous to the use of ordinary differential equations for describing continuous systems. Just like ODEs are defined over a state space, where the variables can take values in $\mathbb{R}$. The state space of TLA formulas is some logical structure (we'll learn precisely what that means) that contains the values which TLA variables can take at any instant in time. In TLA$^+$, sets form that structure, and the logical theory for describing elements in that structure is based on first-order-logic and Zermelo–Fraenkel set theory (ZFC) — the standard formalism of ordinary mathematics. This means that we will use formal mathematics to describe our software, or, for now, just their data structures and basic operations. "Formal" simply means working in a system (a "formal system", or a "formalism") with *precise* rules, both syntactic — how expressions must be formed — and semantic — what the expressions mean.

This static component of TLA$^+$ takes up most of the TLA$^+$ language, and if you're new to specification languages and proof assistants, understanding it will teach you all the essentials of doing formal mathematics on a computer. But from a theoretical standpoint, it is the least interesting and least essential part of TLA$^+$ (TLA, the dynamic component is the

interesting, essential part), although it is important from a design and usability perspective, as it was designed to be as easy and familiar as possible. And yet it is the most controversial aspect of TLA+. This is because Lamport chose an untyped formalism for mathematics, while most specification languages opt for a typed one.

Aside from controversial claims (on either side of the debate) regarding metrics such as correctness or productivity, types in programming languages have advantages that, I believe, most agree on: they help compilers (especially AOT compilers) generate efficient machine code, and they help organize code, by providing some form of up-to-date documentation and encouraging a sensible focus to subroutines. They also assist tooling with automatic completion and automatic refactoring. Neither of these less-controversial advantages, however, is relevant for a specification language like TLA+ because it is not compiled into an executable, and because specifications are orders of magnitude shorter than program code and so don't benefit much from features intended to help large teams of programmers maintain large codebases. The tradeoffs typed and untyped formalisms make for *specification* languages — or math in general — are therefore different from those concerning programming languages, and are well covered in the joint, and balanced, paper by Lamport and Larry Paulson (author of the proof assistant Isabelle), ***Should Your Specification Language Be Typed?***

Lamport's choice is driven by his wish for simplicity (as he sees it) for TLA+'s intended audience, namely engineers, and probably by his background as a classical mathematician. Set theory has the advantage of being familiar[1] yet powerful, and Lamport says he believes it is indeed simpler *in practice* for engineers to use in specification of systems.

Other specification languages may need type systems because they are intended for uses other than software or hardware specification. Lamport **writes**:

> *There's no good reason to write a set like {1, {2,3}}, and making it impossible to write seems like a good idea. However, I've found that there is no simple way to make it impossible to write that set without also making it impossible to write useful sets. As this implies, Coq is not simple… That doesn't meant that there's anything wrong with Coq; it just means that it's not meant for ordinary engineers. For example, if you look at a math text, you might find that the symbol + is used in a single paragraph to mean several different things. To formalize that math in TLA+, you'd have to use a different symbol for each of those different meanings. That would drive a mathematician crazy. Coq allows you to use the same symbol for all of them. So, as George Gonthier[2] will tell you, you need something like Coq for formalizing serious math. Since system builders and algorithm designers don't use that kind of math, they don't need to deal with the complexity of a language like Coq.*

The static component of TLA+ is a formal set theory that Lamport calls ZFM, ZF for Mathematics, which he explains in **this short paper** (along with a comparison with typed formalisms).

Logicians will likely find the TLA+ formalism too ordinary; perhaps even dull. The mathematician G.H. Hardy once wrote[3]: "The 'real' mathematics of the 'real' mathematicians… is almost wholly 'useless'… It is the dull and elementary parts… that work for good or ill." But TLA+ is not a tool for exploring the secrets of mathematics or novel logics, but a tool for engineers to specify systems that "work"[4]. As far as ordinary math goes, TLA+ is a clear, simple and powerful formalism, with a convenient and natural syntax[5].

TLA+ uses formal mathematics to specify software systems. As you'll see, it is not as scary as it may first sound to a programmer. Programming is turning informal requirements into a low-level formal specification of the software — the program. If you can do that, and if you understand why your program works, or at least, how it is supposed to work, you can also write a high-level mathematical specification. It just takes some practice. In the introduction to *Specifying Systems* Lamport writes:

> *The mathematics we use is more formal than the math you've grown up with… The mathematics written by most mathematicians and scientists is not really precise. It's precise in the small, but imprecise in the large.*

> *Each equation is a precise assertion, but you have to read the accompanying words to understand how the equations relate to one another and exactly what the theorems mean. Logicians have developed ways of eliminating those words and making the mathematics completely formal and, hence, completely precise… [M]athematicians and scientists think that formal mathematics, without words, is long and tiresome. They're wrong. Ordinary mathematics can be expressed compactly in a precise, completely formal language. It takes only about two dozen lines to define the solution to an arbitrary differential equation in the* `DifferentialEquations` *module… But few specifications need such sophisticated mathematics. Most require only simple application of a few standard mathematical concepts.*

I must remind you again that this is not a tutorial, and much of the material covered is not necessary to write good TLA+ specifications. While I will cover the basics and hope that the examples in this post will give you a taste of what formal mathematical specification is and how it feels similar to programming, my emphasis is the mathematical theory and design principles behind TLA+. For good hands-on tutorials on TLA+, refer to those I mentioned in **part 1**.

What Is (a) Logic?

TLA+ uses logic to specify both algorithms and their data structures. For those of you who may be rusty on mathematical logic, here's a review.

Logic Fundamentals

Formal logic, symbolic logic, or mathematical logic, is a *formal system*, or, simply a *formalism*[6]. Put simply, it is a precise language to talk about things (or, at least things that are amenable to precision). It is precise because it has precise syntax and precise semantics. The syntax defines rules for how legal (*well formed*) *expressions* (strings) in the language can be formed, and the semantics defines what those expressions *mean* by connecting them to precisely defined mathematical objects. The semantics allows us to know exactly what any phrase in the language means, and the syntax allows us to manipulate phrases in such a way that we know exactly its effect on meaning.

The syntax of the logic is made of some built-in *connectives* — usually $\wedge$ for *conjunction* ("and"), $\vee$ for *disjunction* ("or"), $\neg$ or sometimes $\sim$ for *negation* ("not"), $\Rightarrow$ or $\rightarrow$ for *implication* ("if-then"), $\equiv$ or $\Leftrightarrow$ for *equivalence* (if-and-only-if, or iff), a set of *variables* ($x, y, \ldots$) — names we use to refer to objects — and a *signature*, which is a set of symbols with a specific *arity* (how many arguments the symbol takes), like 5 (0-ary), $=$ (2-ary), $<$ (2-ary), $*$ (2-ary), or $-$ (unary minus). For example, the expression $x * 5 < -y \wedge \neg(x < 5)$ is a legal expression in the logic I've given as an example. The logic can also have *quantifiers*, the most common of which are the *universal* quantifier $\forall$ ("for all") and the *existential* quantifier $\exists$ ("there exists"). Quantifiers usually *bind* variables. For example, $\forall x \ldots$, which means "for all objects x such that …", or $\exists x \ldots$, which means "there exists an object x such that …". A *formula* is a boolean-valued expression, namely one that is either true or false. Variables that appear in a formula unbound are called *free variables*. A formula that has no free variables is called a *sentence* or a *closed formula*.

A logic also has a ***structure***, which is the logic's **domain of discourse** — *what* the logic is talking about — which gives meaning, or *semantics* to the signature. Assigning a specific structure to a logic is called an ***interpretation***. The meaning for 0-ary symbols like 2 is usually called a *constant*, while the meaning of symbols of higher-arity is usually called a *relation* (e.g., the less-than relation, $<$) or a *function*[7]. The meaning of a variable is also defined by the structure, although here the "order" of the logic matters. We'll discuss this subject a bit later, but in first-order logic, a variable can refer any value in a collection of possible values defined by the structure; that collection is called the ***universe*** of the logic.

A *model* is the relationship between the syntax and semantics: a model of a formula is a structure that *satisfies* it, namely an assignment of values to the variables that make the formula *true* (truth is a semantic property). The usual symbol for "satisfies" is $\models$. On the left is a structure that makes the formula on the right true — a model of the formula. For example, the structure for our logic can be the set of integers with multiplication, negation and the less-than relation. A model for the expression $x < 2$ could then be $x = -5$. The collection of all models of a formula A forms its *formal semantics*, and is often written $[\![A]\!]$. I will colloquially refer to that collection of models — the semantics of a formula — simply as *the*

model of the formula, and say that the model of $x < 2$ is any assignment of a number less than 2 to x, or, more simply, all numbers less than 2. We can say that that formula *specifies* all numbers less than two. A formula that is true under al! interpretations is said to be *valid*, and we write $\models A$ (with no structure on the left-hand side). The formula TRUE is satisfied by all interpretations, while the formula FALSE has no model at all.

The various logical operators interact with the model in specific ways. The model for $A \wedge B$ is the intersection of the model of A with the model of B, or $[\![A \wedge B]\!] = [\![A]\!] \cap [\![B]\!]$. The model for $A \vee B$ is the union of the model of A with the model of B, or $[\![A \vee B]\!] = [\![A]\!] \cup [\![B]\!]$. The model for $\neg A$ is the complement of the model for A, or $[\![\neg A]\!] = [\![A]\!]^c$. These are exactly the familiar definitions of the logical operators with **Venn diagrams**.

When we work with a logic, we usually work within a specific *logical theory*, which is a set of formulas called *axioms*, taken to be equivalent to TRUE. A model of a theory is a structure that satisfies all axioms of the theory; in other words, the theory characterizes, or specifies, a model. Often, therefore, a logic is not defined with a structure, just with a theory, which then characterizes all appropriate structures. For example, the **Peano axioms**, are a logical theory that characterizes the natural numbers and the familiar arithmetic operations. The natural numbers we know, with the familiar arithmetic operations, are a model of Peano arithmetic.

Of course, a logical formula, or even an entire logical theory, can have multiple interpretations over different structures. For example, the formula $x > 2$ has a different model if interpreted over the real numbers than over the naturals. The sentence $\exists\, x.\, x > 0 \wedge \forall\, y.\, y > 0 \Rightarrow y \geq x$ is true when interpreted over the integers, but false when interpreted over the reals.

A logic also usually has a *calculus*, a syntactic system for deriving expressions from other expressions, like **natural deduction**. If formula C can be derived by a finite number of application of inference rules from formulas A and B, we write $A, B \vdash C$, and say that A and B *prove* C or ***entail*** C, where A and B are the *assumptions*, and C is their *consequence*. Provability is the syntactic counterpart to the semantic notion of truth. If a formula A is entailed by the theory's axioms alone, without any other assumptions, we write $\vdash A$, and say that A is a *tautology*. If $\vdash A$ and A is not an axiom, we say that A is a *theorem*. If we want to prove the theorem A but we haven't yet done so, we call A a *proposition*.

According to the axioms of most logics, if $A, B \vdash C$ then $\vdash (A \wedge B) \Rightarrow C$ and vice versa. I.e., A and B entail, or prove, C iff $A \wedge B$ *imply* C.

There are two important axioms that shape the general properties of a logic. The first is called the **principle of explosion**, and it states that for any formula $A, \vdash \neg(A \wedge \neg A)$, or equivalently $A \vdash \neg\neg A$. The second is called the **law of excluded middle**, and it states that for any formula $A, \vdash A \vee \neg A$, or equivalently, $\neg\neg A \vdash A$. A logic with both axioms is called a ***classical***, or standard, logic. A logic with the principle of explosion but without the law of excluded middle is called an ***intuitionistic***, or *constructive* logic (in which, if A is known not to be false, we cannot conclude that it must be true). A logic with the law of excluded middle but without the principle of explosion is called a ***paraconsistent*** logic (in which, if A is true, we cannot conclude that it may not also be false). Both constructive and paraconsistent logics have interesting and useful applications. With neither axiom, we have no means of relating negated and positive statements, which is tantamount to a logic without negation at all. All logic employed in TLA+ is classical.

Note that in a classical logic (but not in a non-classical logic!), the universal and existential quantifiers can each be defined in terms of the other:

$$\exists\, x.\, A \equiv \neg \forall\, x.\, \neg A$$
$$\forall\, x.\, A \equiv \neg \exists\, x.\, \neg A$$

In short we say that $\exists \equiv \neg\forall\,\neg$ and $\forall \equiv \neg\exists\,\neg$.

A logic, like all languages, expresses meaning. But the meaning of a logical statement is not always entirely captured by its formal semantics. The logician **Gottlob Frege** pointed out that a logical statement may have two kinds of meaning: **sense and reference** (the latter is sometimes also called, somewhat impreciesly, *denotation*). For example, the sentence $3 > 2$ has a semantic value of TRUE and so *refers to* the value TRUE. The sentence $2 > 1$ also refers to the value TRUE, and so is equivalent to $3 > 2$. However, the two sentences have different *senses*, as they express different ideas (one about the relationship between 3 and 2, and the other about 2 and 1). The distinction between sense and reference is a key

philosophical observation required for a full understanding of all logic (and all language, really), and some non-classical logics — in particular, intuitionistic logics — can explicitly refer to both kinds of meaning (using different notions of equivalence, one called *extensional*, and is an equivalence on reference/denotation, and the other called *intensional*, which can be seen as an equivalence on sense). But we will be dealing only with classical logic, so this important philosophical point has no practical significance.

First Order Logic and Other Orders

A **first-order logic** (or FOL) has all the regular logical connectives ($\wedge$, $\neg$, etc.), and the universal and existential quantifiers $\forall$ and $\exists$. The variables in FOL can represent values that are simple values of the universe: if the universe is the natural numbers, then a variable — either free or quantified, i.e. bound — stands for a natural number.

In a **second-order logic** (or SOL), a variable can either represent an element of the universe or a set of elements. It is easier to think of this set as a predicate; a predicate $p(x)$ is true iff the value x is in the set p. So, while in FOL all predicates must be "constant" and given in the signature, in SOL we can quantify over them and say, for example, $\forall_1 x. \exists_2 p. \forall_1 y. p(y) \equiv (y = x)$ (the symbol $\equiv$ stands for double implication, or if-and-only-if, and I've used subscripts to distinguish between first- and second-order quantification; often this distinction is clear from the context), or in words, "for any x, there exists a predicate p that is true for an argument y iff $y = x$".

Similarly, a third-order logic allows us to quantify over sets of sets of values etc. **Higher-order logic**, or HOL, allows us to quantify over variables of any order, and is usually used in typed formalisms only. The original concept of **types** was created by Bertrand Russell precisely to distinguish order: objects of different orders are of different *type*, i.e., if our "ground" values are of type one, then sets of those are of type two, sets-of-sets are of type three etc.. Types have a syntactic as well as a semantic significance: if x and y are of different types, then the expression $x = y$ is not false, but rather *ill-formed*, meaning it is not a legal expression at all. Similar to the subscripts I used above, quantifiers in HOL are distinguished by types; we write $\exists\, x : T$ (instead of $\exists_T x$) meaning "there exists an x of type T".

One of the most important theorems in all of logic is that in first-order logics (only!), semantic truth and syntactic provability coincide, i.e. $\vdash A$ iff $\models A$, or, A is a tautology iff it is valid. In other words, if a formula A is *valid* — satisfied by all interpretations (that also satisfy the axioms of the theory) — then it has a proof, and vice-versa. Similarly, $A \vdash B$ iff $[\![A]\!] \models B$. This is **Gödels' completeness theorem**. Note, however, that this doesn't mean that every sentence in FOL can be proven or disproven. If your theory is rich enough, there will always be sentences that can be neither disproven nor proven (which, in the case of FOL, means that such theories will have multiple models, in some the sentence would be true and in others false). This is a consequence of yet another famous result, known as **Gödel's first incompleteness theorem**. That theorem doesn't apply only to FOL, but to any formalism whatsoever, as it is a direct result of the halting theorem (even though the halting theorem was proven some years later).

As both the "static" or data logic and the temporal logic in TLA$^+$ only allow first-order formulas, we can just use the symbol $\vdash$ to mean both validity and provability (unless we want to specifically talk about a semantic structure on the left-hand side). But as the first incompleteness theorem applies to ZFC, not every TLA$^+$ sentence can be proven or disproven.

Because TLA$^+$ allows free variables, I'd like to point out a possible source of confusion when reading formulas containing implication ($\Rightarrow$) and free variables: The formula $x > 4 \Rightarrow x > 2$ is a theorem, i.e., $\vdash x > 4 \Rightarrow x > 2$, and is satisfied by all assignments to x, and $\forall\, x : x > 4 \Rightarrow x > 2$ is also a theorem. On the other hand, while $\forall\, x : x > 2 \Rightarrow x > 4$ is equivalent to FALSE, i.e. $\vdash \neg(\forall\, x : x > 2 \Rightarrow x > 4)$, the formula $x > 2 \Rightarrow x > 4$ does have a model: it is satisfied by all numbers that are either less than or equal to 2 or greater than 4. This means that we have to know whether we're treating $x > 2 \Rightarrow x > 4$ as a proposition — equivalent to asking whether it is valid, or satisfied by *all* assignments to x — which in our case is not true, or not, in which case it specifies a model. Describing the model for $A \Rightarrow B$ as we did above is not interesting, but the *proposition* $\vdash A \Rightarrow B$ from a semantic perspective is: $\vdash A \Rightarrow B$ iff $[\![A]\!] \subseteq [\![B]\!]$.

Now, notice that the model for a FOL formula, say $x > 2$, is always a set of values from the universe (in this case, all numbers greater than 2). So the model of a FOL formula is always a second-order object. But if the meaning of a formula is a set, a second-order object, then how can we write a general axiom such as $\vdash A \vee \neg A$ where A stands not for a *specific* formula but for *any* formula? We can't write $\forall\, A.\, A \vee \neg A$ because we can't quantify over formulas (which, as we've just

seen, stand for sets, or second-order values) nor treat them as free variables for the same reason. The answer is that we don't. We say that such an axiom is an *axiom schema* that refers to an infinite number of axioms in the logic, one for each possible formula A. The language used to write the schema is therefore not the logic itself, but the *metalogic*, or *metalanguage* of our logic.

While this is fine from a theory standpoint, this is not satisfactory for a mechanical proof software. When proving a theorem, we often need to prove may intermediate steps, and we would like to be able to state and prove general lemmas which we could reuse in many proofs, and such general second-order lemmas can be very useful.

One solution is, therefore, to adopt a higher-order logic. HOL serves as the basis for the theorem provers Isabelle/HOL and HOL Light, and more arcane type theories, like the constructive theories used in Coq or Agda, are also higher-order. But keeping the logic first-order has some advantages in terms of simplicity, model checking algorithms, general theorems (if we know that a formula is first-order, it's easier to state theorems about a formula's meaning based on its syntactic structure) — the last one is more relevant for TLA, the temporal logic part of TLA+, which we'll cover in the next installments. But it turns out that we can have our cake and eat it, too. We can keep all formulas first-order, but allow writing second-order propositions in the form $A \vdash B$, which are not in themselves formulas (and so they don't have a model). We basically provide access to the metalanguage. This is the solution adopted by TLA+, which we'll see later in the section about **proofs**.

By the way, because I mentioned model checking, I'd like to take the opportunity to clarify what may be a certain misconception about model checkers. A model checker takes its name from the definition of "model" we've just learned in the context of logic. It is a program that takes a description of a logical structure, M and a logical formula, φ, and *checks* that M is a *model* of φ, or that $M \models \varphi$.

Other than the theoretical and practical advantages of FOL, it has some theoretical disadvantages. For example, first-order theories have **non-standard models** (e.g. there are models of the first-order Peano axioms with uncountably many numbers). However, this is not a concern at all given TLA+'s intended use and audience. "I'm certainly not worried," Lamport writes, "about a bug occurring because an engineer implemented a non-standard model of the integers.[8]"

Logical Formulas and Expressions

If you're familiar with standard mathematical notation you'll find TLA+'s syntax intuitive and readable for the most part.

The static, data logic of TLA+ is first order logic with equality. It has the familiar operators[9]: $\wedge$ for conjunction, $\vee$ for disjunction, $\neg$ for negation, $\Rightarrow$ for implication, and $\equiv$ for equivalence. $\exists$ and $\forall$ are the regular first order existential and universal quantifiers. In the TLA+ syntax, we write $:$ (colon) instead of the more common single dot after a quantifier, which is used instead of parentheses to delineate the scope of the bound variables, and is read as "such that". The logical constants are TRUE and FALSE. $\wedge$ and $\vee$ have the same precedence, so the expression $a \wedge b \vee c$ is a syntax error due to the precedence ambiguity, and parentheses are required to resolve it. Implication has a lower precedence, so $A \wedge B \Rightarrow C$ is equivalent to $(A \wedge B) \Rightarrow C$.

Variables that are not part of the state of a computation (we'll learn about state in part 3), can be introduced with the quantifiers $\forall$ or $\exists$, or can be declared with the keyword $\mathrm{CONSTANT}$, in which case they are free.

In addition, TLA+ has a couple of special constructs inspired by programming languages. The construct

$$\mathrm{IF}\ p\ \mathrm{THEN}\ x\ \mathrm{ELSE}\ y$$

where p is some logical predicate and x and y are values, takes the value x if p is TRUE or the value of y otherwise. The construct

$$\mathrm{CASE}\ p_1 \rightarrow e_1 \ \square \ \ldots \ \square \ p_n \rightarrow e_n \ldots \ \square \ \mathrm{OTHER} \ \rightarrow e$$

is equal to some e_i such that p_i is true, if any, otherwise it is equal to e if the OTHER clause exists; if it does not, the value of the expression is undefined (we'll look at what precisely that means later). If more than one predicate is true, the value

of the expression would be one of them, but the language does not commit to which. Both IF/THEN and CASE are defined in terms of the CHOOSE operator we'll see later.

Because in TLA+ specifications of algorithms and systems are written as formulas, it requires some ergonomics for writing long formulas in a way that makes them easy to read. Lamport devised such a scheme in ***How to Write a Long Formula***. It uses alignment of conjunctions and disjunctions in place of parentheses, like so:

$$\begin{array}{l} \lor\ a \\ \lor\ b \land c \\ \quad\ \land d \\ \lor\ e \end{array}$$

Where a, b, c and d stand for arbitrary subformulas. Note the prefix connective that starts the disjunction and conjunction lists. The implied parentheses extend from the expression following the connective starting the line until another connective (of the same kind) is encountered in some new line aligned with the opening connective. So the above means the same as the following (which is also allowed in TLA+):

$$(a) \lor ((b \land c) \land (d)) \lor (e)$$

Definitions

The definition is the main building-block in TLA+ (in fact it is almost the only one, the only other being the module, which we'll learn about in part 4). A definition looks like so:

$$\text{Name} \triangleq e$$

Where Name is the name of our definition, and e is a TLA+ expression.

Definitions can be parameterized, in which case they're called operators:

$$Double(x) \triangleq 2 * x$$

Operators are *not* functions; those have a precise meaning, and we'll learn the distinction later. Operators can be second-order:

$$ApplyTwice(Op(\_), x) \triangleq Op(Op(x))$$

Defining some symbolic operators is also supported:

$$a \preceq b \triangleq a\%b = 0$$

We can also use symbolic names for operator arguments:

$$Foo(x, y, \_ \prec \_) \triangleq \text{IF } x \prec y \text{ THEN } x \text{ ELSE } y$$

(any 2-ary operator can be passed in place of $\prec$, not just ones defined as symbolic operators).

Recursive operators are also allowed:

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

Anonymous operators can be defined inline with LAMBDA [10], as in $ApplyTwice(\text{LAMBDA } x : x^2, 3)$.

Definitions have a scope of the module in which they are defined (we will learn about modules in part 4), and a definition must precede its use. Definitions that are local to an expression are introduced with the $\text{LET}\ ..\ \text{IN}$ construct:

$$
\begin{aligned}
Foo(a, b) \triangleq \text{LET } x \triangleq \text{IF } a \leq b \text{ THEN } a \text{ ELSE } b \\
y \triangleq x * a \\
\text{IN } y * b
\end{aligned}
$$

LET definitions can be made inside any expression.

TLA+ does not let you name variables (or definitions) that are already bound in scope, so this is not allowed:

$$
\begin{aligned}
x \triangleq 3 \\
Foo(x) \triangleq x + 1
\end{aligned}
$$

and neither is this:

$$
\begin{aligned}
Foo(x) \triangleq \\
\quad \text{LET } Bar(x) \triangleq x + 1 \\
\quad \text{IN } x * 2
\end{aligned}
$$

But this is allowed because x is not in scope when Foo is defined:

$$
\begin{aligned}
Foo(x) \triangleq x + 1 \\
x \triangleq 3
\end{aligned}
$$

The values of the logic are the objects of its structure. As we'll see momentarily, the data logic — the language used to describe the static data of algorithms — in TLA+ is a first-order logic over set theory, and so the values of the logic are all sets. Operators are not themselves values, and we cannot quantify over them. For example, we cannot write:

$$\forall C, D, F : (C = D) \Rightarrow (F(C) = F(D))$$

Operators work by syntactic substitution[11]; from a practical point of view they work similarly to hygienic macros, but they must evaluate to a value. We, therefore, cannot define the following operator, as the LAMBDA expression is an operator, not a value:

$$Add(x) \triangleq \text{LAMBDA } y : x + y$$

This could, however, be easily achieved with functions, as we'll see later.

From a logic point-of-view, operators are second-order objects, or objects of the meta-language. TLA+ allows treating them as free variables denoting second-order objects using the CONSTANT construct we'll learn **later**, as well as in the **proof language**.

Sets

The data logic in TLA+ is based on **ZFC set theory**, so all values in TLA+ are formally sets. Even the number 1 is a set, TRUE is a set, and the function $\tan$ is a set. But as we'll see later, how those values are encoded as sets is hidden, and their internal structure is inaccessible to us that it may as well not exist at all, and we can safely treat them as primitive values.

When working with set theory, we usually use first-order logic. However, as the values of the universe are sets, the expressivity of the logic is similar to that of higher-order logic[12]. In fact, FOL over ZFC is considered the "standard" formal system for (ordinary) mathematics.

Set Fundamentals

The constant $\{\}$ is the empty set, and there are two binary relations that are defined for all sets: $\in$ is the membership relation, and $=$ is equality, where sets are equal by extensionality, i.e., iff they have the same elements.

Sets can be constructed only in accordance with the axioms of ZFC[13]:

If a and b are sets, then $\{a, b\}$ is another set (axiom of pairing).

If S is a set and P is some predicate, then $\{x \in S : P(x)\}$ is a set containing all members of S that satisfy P (axiom of specification/separation). For example, $\{n \in Nat : n\%2 = 0\}$ is the set of even natural numbers. Note that a colon is used instead of the more common $|$ in the set comprehension expression.

If S is a set and F some expression then $\{F(x) : x \in S\}$ is a set, constructed by replacing each member of S with its image under the operator F (axiom of replacement)[14]. For example, $\{2 * n : n \in Nat\}$ is also the set of even natural numbers.

If S is a set, then $\text{UNION } S$ is a set containing all members of the members of S (axiom of union). E.g. $\text{UNION } \{\{1, 2\}, \{2, 3\}\} = \{1, 2, 3\}$.

If S is a set, then $\text{SUBSET } S$ is the **power set** of S, namely the set of all subsets of S (axiom of power set). For example, $\text{SUBSET } \{1, 2, 3\} = \{\{\}, \{1\}, \{2\}, \{3\}, \{1, 2\}, \{1, 3\}, \{2, 3\}, \{1, 2, 3\}\}$.

Quantifiers can (and usually do) appear bounded (e.g. $\forall x \in S : \ldots$), in which case they satisfy the following:

$$(\forall\, x \in S : P(x)) \equiv (\forall\, x : x \in S \Rightarrow P(x))$$
$$(\exists\, x \in S : P(x)) \equiv (\exists\, x : x \in S \land P(x))$$

TLA+ has the usual set operators $\cup$ (union), $\cap$ (intersection), $\subseteq$ (subset or equal), $\subset$ (strict subset), \ (set difference) etc., all trivially defined using the fundamental operations above.

The set $\text{BOOLEAN} \triangleq \{\text{TRUE}, \text{FALSE}\}$ is built-in. TLA+ does not distinguish between the elements of the set BOOLEAN and logical truth values. So, for example, $\forall\, p \in \text{BOOLEAN} : p \lor (5 < 6)$ is true, one can write $(3 < 4) = (8 > 2)$, using $=$ instead of $\equiv$ even though the latter is a logical connective while the former is a relation on sets, and $(1 > 0) \in \text{BOOLEAN}$ is true. A formula (and a predicate) is, therefore, any boolean-valued expression.

Let's see some examples of TLA+ definitions using sets. Here is an example of a useful operator that states that there exists one and only one member of a set satisfying some predicate:

$$ExistsOne(S, P(\_)) \triangleq \exists\, x \in S : P(x) \land \forall\, y \in S : P(y) \Rightarrow y = x$$

(this is a common pattern in logic to express a unique value; a value is unique iff two variables referring to it must be equal).

Now let's define some richer mathematical concepts. We'll define a *proset* — a set with a **preorder** — a ***poset*** — a partially-ordered set — and a **totally-ordered** set:

$$\begin{aligned} Proset(S, \_ \preceq \_) \triangleq\ & \land \forall\, a \in S : a \preceq a && \text{Reflexivity} \\ & \land \forall\, a, b, c \in S : (a \preceq b \land b \preceq c) \Rightarrow a \preceq c && \text{Transitivity} \end{aligned}$$

$$\begin{aligned} Poset(S, \_ \preceq \_) \triangleq\ & \land Proset(S, \preceq) \\ & \land \forall\, a, b \in S : (a \preceq b \land b \preceq a) \Rightarrow a = b && \text{Antisymmetry} \end{aligned}$$

$$\begin{aligned} Toset(S, \_ \preceq \_) \triangleq\ & \land Poset(S, \preceq) \\ & \land \forall a, b \in S : a \preceq b \lor b \preceq a && \text{Totality} \end{aligned}$$

And here are some important algebraic structures, the **semigroup**, the **monoid** and the **group**[15]:

$$\begin{aligned} Semigroup(S, \_ \cdot \_) \triangleq & \land \forall a, b \in S : a \cdot b \in S && \text{Closure} \\ & \land \forall a, b, c \in S : (a \cdot b) \cdot c = a \cdot (b \cdot c) && \text{Associativity} \end{aligned}$$

$$\begin{aligned} Monoid(M, \_ \cdot \_) \triangleq & \land Semigroup(M, \cdot) \\ & \land \exists id \in M : \forall a \in M : \land id \cdot a = a \land a \cdot id = a && \text{Identity element} \end{aligned}$$

$$\begin{aligned} Group(G, \_ \cdot \_) \triangleq & \land Monoid(G, \cdot) \\ & \land \exists id \in G : \forall a \in G : \land id \cdot a = a \land a \cdot id = a \\ & \qquad \land \exists b \in G : a \cdot b = id \land b \cdot a = id && \text{Inverse element} \end{aligned}$$

$$\begin{aligned} AbelianGroup(G, \_ \cdot \_) \triangleq & \land Group(G, \_ \cdot \_) \\ & \land \forall a, b \in G : a \cdot b = b \cdot a && \text{Commutativity} \end{aligned}$$

I could have defined $Group$ directly in terms of $Semigroup$, as we must repeat the identity condition (or use a construct we haven't yet learned), but I wanted to emphasize that a group is a monoid. That the identity element is unique is a theorem we will formally state and prove later.

Finally, the **quotient set** of a set S contains its equivalence classes with respect to an equivalence $\sim$ (often written $S/\sim$) and can be defined like so:

$$\begin{aligned} Quotient(S, \_ \sim \_) \triangleq \{x \in \text{SUBSET } S : & \land x \neq \{\} \\ & \land \forall a, b \in x : a \sim b \\ & \land \forall a \in S : (\exists b \in x : a \sim b) \Rightarrow a \in x\} \end{aligned}$$

The first conjunct says that x — a member of the quotient — must not be empty, the second says that all members of x are equivalent, and the third conjunct says that x is maximal, i.e., contains *all* members of S that are equivalent to those in x; therefore x is an equivalence class.

People who are not accustomed to writing formal definitions — and even those who are — may neglect to write the third conjunct, but the TLA+ tooling can help. $Quotient(\{1, 2, 3, 4\}, \text{LAMBDA } a, b : a\%2 = b\%2)$ specifies the equivalence classes of numbers that are of equal parity: $\{\{1, 3\}, \{2, 4\}\}$. The TLA+ toolbox allows you to evaluate expressions using the model checker TLC[16]; not quite a REPL, but almost. If you try to evaluate the above expression without the third conjunct in the definition you'll get $\{\{2\}, \{3\}, \{4\}, \{1, 3\}, \{2, 4\}\}$ and realize your mistake.

There are often many equivalent ways of defining mathematical concepts. For example, here is another way of defining $Quotient$:

$$\begin{aligned} ClassOf(a, S, \_ \sim \_) &\triangleq \{b \in S : b \sim a\} \\ Quotient(S, \_ \sim \_) &\triangleq \{ClassOf(a, S, \sim) : a \in S\} \end{aligned}$$

The `FiniteSets` module (which is part of the TLA+ standard module library), defines useful set operators, like $IsFiniteSet(S)$, which is true iff S is finite, and $Cardinality(S)$, which is the number of elements in S if S is finite (we define $Cardinality$ ourselves **later on**).

The CHOOSE Operator and the Meaning of Undefined

Completing the picture of the fundamental operators is CHOOSE, which is **Hilbert's epsilon operator**. Technically, CHOOSE can be added to any first-order logic and doesn't rely in any way on our theory of sets, but as it is a powerful general mechanism for talking about specific values, I thought that we should first get to know the values in our theory.

$\text{CHOOSE } x : P(x)$, is equal to *some* value (i.e. set) satisfying the predicate P, if one exists. CHOOSE satisfies these two rules:

$$(\exists\, x : P(x)) \equiv\ P(\text{CHOOSE } x : P(x))$$
$$(\forall\, x : P(x) \equiv Q(x)) \Rightarrow (\text{CHOOSE } x : P(x)) = (\text{CHOOSE } x : Q(x))$$

The first rule shows how the quantifier $\exists$ can be defined in terms of CHOOSE (and therefore so can $\forall$). The second rule is sometimes called *right-uniqueness*, and means that CHOOSE is *not* nondeterministic, but rather always picks a specific value (although not one that is dictated by the logic). In other words, there is just one model for the formula $a = \text{CHOOSE } x : \text{TRUE}$, although the axioms of the logic don't allow us to know which particular model it is. This is a common misunderstanding among new TLA+ users.

Both IF/THEN and CASE constructs are defined in terms of CHOOSE. See *Specifying Systems*, p. 298 for the precise definitions.

CHOOSE (and the quantifiers, too) allows to use as its bound variable the name that is currently being defined, so we can, and often do, write $x \triangleq \text{CHOOSE } x : P(x)$.

Like the quantifiers, CHOOSE usually appears bounded, as in, $\text{CHOOSE } x \in S : P(x)$, which is shorthand for $\text{CHOOSE } x : x \in S \land P(x)$.

If no value satisfies the predicate P, the value of $\text{CHOOSE } x : P(x)$ is *undefined*, which means that it is equal to *some* value — could be 42, could be $\{3, \{4, 5\}\}$ — which we simply cannot determine (meaning, prove in the logic). In other words, "undefined" means "we don't know and we don't care".

When an undefined value is part of an expression that always yields a boolean value (like expressions involving $\in$ or $=$, which are defined on all sets), the value of that expression is some indeterminable BOOLEAN; for example: $3 \in (\text{CHOOSE } x \in \{\} : x > 0)$. In such cases I will also say that the value of the formula is undefined even though it is known to be a BOOLEAN, because the logic says nothing about whether that value is TRUE or FALSE. We can similarly use the word "undefined" loosely to refer to an indeterminable value in general, even if we do know it is a member of some set.

$ChooseOne$ is a useful operator, whose value is defined only if *exactly one* value matches the predicate:

$$ChooseOne(S, P(\_)) \triangleq \text{CHOOSE } x \in S : P(x) \land \forall\, y \in S : P(y) \Rightarrow y = x$$

Another useful operator is $AnyOf$, which picks an element from a set (and is undefined if the set is empty):

$$AnyOf(S) \triangleq \text{CHOOSE } x \in S : \text{TRUE}$$

CHOOSE only *selects* values, it does not construct them, so you can't CHOOSE the Russell paradox "set" because one does not exist, as it cannot be constructed using any of the set construction operations. The value of $\text{CHOOSE } x : \forall\, s : s \in x \equiv s \notin s$ (which says that x is the set of all sets that do not contain themselves — a paradox) is, therefore, undefined, as no such set x exists; the right-hand side of the CHOOSE expression is false for all sets in ZFC.

It is important to emphasize yet again that CHOOSE, despite being named as a verb — a confusing choice for programmers, I believe[17] — does not imply any algorithm, or indeed any dynamic process for finding a value. It "statically" describes *what* something is, not *how* to find it. For example $\text{CHOOSE } x \in S : x > 0$ means "*some* positive element in the set S", not "*find* a positive element in the set S".

Some Important Sets

STRING is another built-in set, which is the set of all finite character strings. Strings are *sequences* of characters (we'll cover sequences in the next section), and characters are primitive values with an opaque encoding (TLA+ has no syntax for character literals).

The modules `Naturals`, `Integers`, and `Reals` define the sets Nat ($\mathbb{N}$), Int ($\mathbb{Z}$), and $Real$ ($\mathbb{R}$) respectively, with $Nat \subset Int \subset Real$, along with the familiar arithmetic operators, order relations ($\leq$) etc.. Importing the public definitions

from a module is done with the EXTENDS statement, as in $\text{EXTENDS}\ Naturals$. You can't use arithmetic operators in a TLA+ specification (the numeric literals are built-in) unless you import one of those modules that defines them.

The special syntax $a\,..\,b$ defines the set $\{n \in Nat : n \geq a \land n \leq b\}$, so $2..5 = \{2, 3, 4, 5\}$. TLA+ currently has no special syntax for complex numbers or matrices.

In set theory, all values are encoded as some set (so the number 1 is somehow encoded as a set). However, TLA+ lets us hide encodings within modules so that their details do not escape the module boundary and appear to external modules as primitives (we will cover modules in part 4), and so the value of $1 \in 2$ or of $1 = \text{``hi''}$ is undefined (i.e., indeterminable). Because equality (and membership) is defined on all sets to have a boolean value we know that $1 = \text{``hi''}$ must be some boolean, namely, either TRUE or FALSE, but we have no way of determining which. In a typed formalism, the expression $1 = \text{``hi''}$ is simply illegal (ill-formed); in a dynamically-typed programming language it evaluates to false. But in TLA+ it is simply nonsensical — we cannot assign it any meaning, i.e. a value[18] — like the english expression "Thursday is purple", which is grammatically legal, or well-formed, yet carries no well-accepted meaning. If you'd like to revisit the tradeoffs of this choice, I'll refer you to the two papers I linked to above[19].

As a result, we don't know whether the set $\{1, \text{``hi''}\}$ contains one or two elements because we cannot know if 1 is equal to "hi" or not as we don't know how "hi" is encoded (TLC forbids such a set altogether, as its elements are incomparable). So what do we do if we want to have a set that contains the values 0, 1, and UNKNOWN? We use CHOOSE to define UNKNOWN like so:

$$\text{UNKNOWN} \triangleq \text{CHOOSE}\ x : x \notin \{0, 1\}$$

and then we can write the set $\{0, 1, \text{UNKNOWN}\}$, now known to contain three elements.

Division for real numbers (defined in the `Reals` module) could be defined like so:

$$a/b \triangleq \text{CHOOSE}\ c \in Real : a = b * c$$

This definition immediately tells us, by the meaning of the CHOOSE operator, that $1/0$ is undefined[20], in the very precise sense explained in our discussion of CHOOSE.

With numbers, arithmetic operations and the order relation on numbers we can start doing some more concrete math. We'll begin with something simple: a definition of the operator $Prime$, which is a predicate saying whether a number is prime (other definitions are also possible):

$$Divides(p, n) \triangleq \exists\, q \in Int : p * q = n$$
$$Prime(n) \triangleq n > 1 \land \forall\, p \in Nat : Divides(p, n) \Rightarrow p = n \lor p = 1$$

Now let's define the GCD operator, the greatest common divisor of two natural numbers:

$$DivisorsOf(n) \triangleq \{p \in Int : Divides(p, n)\}$$
$$SetMax(S) \triangleq \text{CHOOSE}\ x \in S : \forall\, y \in S : x \geq y$$
$$GCD(m, n) \triangleq SetMax(DivisorsOf(m) \cap DivisorsOf(n))$$

Notice how straightforward is the GCD definition: first we define what a divisor is; then we define what *the divisors* of a number are. The common divisors are then just the intersection of the two numbers' divisors, and the GCD is the greatest among them.

We can use the sets we've seen and some definitions we made above to state a couple of interesting theorems:

$$\text{THEOREM } AbelianGroup(Int, +)$$

$$\begin{aligned}\text{THEOREM } \forall\, n \in Nat \setminus \{0\} : \text{LET } a \sim b &\triangleq a\%n = b\%n && \text{Equality modulo } n\\ a \oplus b &\triangleq ClassOf(AnyOf(a) + AnyOf(b), Int, \sim) && \text{Sum on equiv. classes}\\ \text{IN } &AbelianGroup(Quotient(Int, \sim), \oplus)\end{aligned}$$

It is important to point out that Nat is the set of *all* natural numbers, Int is the set of *all* integers, and $Real$ is the set of the actual real numbers — all uncountably many of them. If we want to work with, say, 32-bit integers or 64-bit floating point numbers, we need to define them and the arithmetic operations on them. You may wonder what good are so many real numbers considering that a program can only represent a small, finite subset of them. The answer is that TLA+ is not a programming language but a specification language, and it is useful to specify and reason about more than just the actual variables of your program. For example, if you're specifying a numerical algorithm, it is easier to analyze its error if you can actually express the precise result it tries to approximate. Or if you are building a cyber-physical system — a discrete system that interacts with objects in the real world; think drone, sensor or robot — you may want to also model the system's environment, which is easier to do by describing it as a continuous system using familiar physics equations (we will talk about cyber-physical systems in part 4).

Functions, Sequences and Records

Most interesting objects we deal with both in mathematics and programming are not normally thought of as sets. TLA+ lets us cleanly express standard mathematical and programming objects.

Functions

Usually, a **function** is defined to be a one-valued relation, where a relation is a set of pairs — in other words, a function is defined by its **graph** — but in TLA+ functions are not defined as sets of pairs but as primitives (meaning that, like numbers, their encoding as sets is unknown, or opaque[21]). In fact, it is pairs that are actually functions in TLA+ as they're a special case of sequences, which are in turn a special case of functions, as we'll see. In any event, functions in TLA+ are not computations; they have no dynamic behavior and no computational complexity. They are just values in the state space of algorithms. Programmers may best think of them as associative arrays, albeit possibly infinite in size (even uncountably big).

If A and B are sets, $[A \to B]$ is the set of all functions from A to B. Function application is denoted with square brackets ($f[x]$) to syntactically differentiate it from operator "application" (really, substitution). Unlike an operator, whose arguments can be any value in the universe or even other operators, every function has a domain — a set — which is obtained with the DOMAIN operator like so $\forall\, f \in [A \to B] : \text{DOMAIN } f = A$. Because a function has a set domain, we cannot have a function whose parameter can be *any* set or even any function (as we can with operators), because that would make the domain of the function "too large to be a set", namely, a proper class, which doesn't exist in ZFC.

Functions are (first-order) objects in our logic — they are just (opaque) sets themselves — while operators are not (they are second-order objects, or objects in the metalanguage). We can therefore quantify over functions as we do over any other values, as in $\forall\, f \in [A \to B] : \ldots$. Because a function's encoding as a set is opaque, statements like $3 \in f$, for some function f, are undefined and nonsensical.

The **image** of a function can be defined as:

$$Image(f) \triangleq \{f[x] : x \in \text{DOMAIN } f\}$$

Equality for functions is *extensional*, meaning, for two functions, f and g,

$$(f = g) \equiv (\text{DOMAIN } f = \text{DOMAIN } g \land \forall\, x \in \text{DOMAIN } f : f[x] = g[x])$$

We could define a specific function $double$ on the natural numbers like so,

$$double \triangleq \text{CHOOSE } f \in [Nat \to Nat] : \forall\, n \in Nat : f[n] = 2 * n$$

but that would be very cumbersome and so TLA+ allows us to define it like so:

$$double \triangleq [n \in Nat \mapsto 2 * n]$$

or like so,

$$double[n \in Nat] \triangleq 2 * n$$

The latter form is syntactic sugar for

$$double \triangleq \text{CHOOSE } f : f = [n \in Nat \mapsto 2 * n]$$

Because of this, functions defined in this form can be recursive:

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

as this is just shorthand for

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

which defines $fact$ as a **fixed-point** (or fixpoint). To see that more clearly, we can define:

$$Fixpoint(F(\_)) \triangleq \text{CHOOSE } x : x = F(x)$$

and then:

$$fact \triangleq Fixpoint(\text{LAMBDA } f : [n \in Nat \mapsto \text{IF } n \leq 1 \text{ THEN } 1 \text{ ELSE } n * f[n - 1]])$$

Note that recursive functions, because they are really specified as fixpoints, may turn out to be undefined if the fixpoint does not exist, as in the following case:

$$f[n \in Nat] \triangleq 1 + f[n]$$

So when using recursive function definitions in proofs, you may need to prove that they define actual functions by showing that they indeed match a value for every value in their domain (or in CS speak, that they're *total*).

Functions can be defined on any set (including sets of functions) and "return" any value, including other functions, so we can define the "higher-order" function:

$$add[x \in Nat] \triangleq [y \in Nat \mapsto y + x]$$

and then,

$$inc \triangleq add[1]$$

Formally, however, unlike in typed formalisms, functions such as add are not really higher-order in TLA+, as all functions — all values really — have the same type; they are all values in the universe of sets.

We are now in a position to define a few **common important properties of functions**: *injection* (a one-to-one mapping), *surjection* (an onto mapping) and *bijection*:

$$\begin{aligned} Injection(f) &\triangleq \forall x, y \in \text{DOMAIN } f : x \neq y \Rightarrow f[x] \neq f[y] \\ Surjection(f, S) &\triangleq \forall y \in S : \exists x \in \text{DOMAIN } f : f[x] = y \\ Bijection(f, S) &\triangleq Surjection(f, S) \wedge Injection(f) \end{aligned}$$

Where the variable S represents the function's codomain. Note that we could have also defined $Injection$ as $\forall y \in Image(f) : ExistsOne(\text{DOMAIN } f, \text{LAMBDA } x : f[x] = y)$.

Now let's define a function composition operator (I will use the ● operator because ∘, the common symbol for function composition, is normally used in TLA+ for sequence concatenation):

$$g \bullet f \triangleq [x \in \text{DOMAIN } f \mapsto g[f[x]]]$$

Note that $g \bullet f$ is only defined if f and g are composable, meaning $Image(f) \subseteq \text{DOMAIN } g$ — remember, there is no such thing as a partial function in ordinary math, at least not in the same sense as in programming. If we want function composition that works like that of partial functions in programming, we could define:

$$\begin{aligned} g \star f \triangleq \text{LET } Preimage(h, S) \triangleq \{x \in \text{DOMAIN } h : h[x] \in S\} \\ \text{IN } [(x \in \text{DOMAIN } f \cap Preimage(g, Image(f))) \mapsto g[f[x]]] \end{aligned}$$

Unlike $g \bullet f$, $g \star f$ is *always* a function, but it is not necessarily a function on $\text{DOMAIN } f$ but potentially only on a subset of it, and it may even be the empty function (the function on the empty set, one that is undefined for any argument) if $Image(f) \cap \text{DOMAIN } g = \{\}$.

The following operator defines an identity function on an arbitrary set:

$$Identity(S) \triangleq [x \in S \mapsto x]$$

The following is a theorem about $Identity$:

$$\forall S : \forall x \in S : Identity(S)[x] = x$$

Let's also define an $Inverse$ operator for the inverse of any invertible function (the definition says: for any y in the function's image we pick an x in the domain that is mapped to y, provided that x is the only point that maps to y):

$$Inverse(f) \triangleq [y \in Image(f) \mapsto ChooseOne(\text{DOMAIN } f, \text{LAMBDA } x : f[x] = y)]$$

A function is invertible if for every element in its codomain, there is one and only one element in its domain that is mapped to it, or in other words, if it is a bijection. We would like to state that as a general theorem about functions, but here we have a problem: the collection of all functions is not a set (but a proper class), so we cannot write $\forall f \in Function : \ldots$ (unlike in typed functional languages, where the type `forall a b. a → b` denotes all functions). However, we can define the predicate Fn, with a seemingly silly definition, which is true iff its argument is any function:

$$Fn(f) \triangleq f = [x \in \text{DOMAIN } f \mapsto f[x]]$$

The following is, then, a theorem:

$$\begin{aligned} \forall S, f : Fn(f) \land Bijection(f, S) \Rightarrow \land\ & Inverse(f) \bullet f = Identity(\text{DOMAIN } f) \\ \land\ & f \bullet Inverse(f) = Identity(S) \end{aligned}$$

as is:

$$\begin{aligned} \forall f : Fn(f) \land Injection(f) \Rightarrow \land\ & Inverse(f) \bullet f = Identity(\text{DOMAIN } f) \\ \land\ & f \bullet Inverse(f) = Identity(Image(f)) \end{aligned}$$

We can manipulate functions in all sorts of ways. For example, if inc is the function defined above that increments every natural number by one, we can, of course, define the following:

$$g \triangleq [x \in Nat \mapsto \text{IF } x \geq 1 \land x \leq 2 \text{ THEN } inc[x] * 10 \text{ ELSE } inc[x]]$$

But the EXCEPT construct, which allows us to "change" specific function values, makes it easier:

$$g \triangleq [inc \text{ EXCEPT } ![1] = inc[1] * 10,\ ![2] = inc[2] * 10]$$

The above could also be written as:

$$g \triangleq [inc \text{ EXCEPT } ![1] = @ * 10,\ ![2] = @ * 10]$$

where @ refers to the original function's value at that point.

We also have syntactic sugar to define and apply functions with multiple parameters. We'll talk about it once we've learned about tuples.

Now let's use functions and one of the sets we've seen to formally define some more important mathematical concepts[22]:

$$\begin{aligned}
&RealFunction \triangleq \text{UNION } \{[S \to Real] : S \in \text{SUBSET } Real\} \\
&AbsoluteValue(a) \triangleq \text{IF } a \geq 0 \text{ THEN } a \text{ ELSE } -a \\
&OpenBall(a, e) \triangleq \{x \in Real : AbsoluteValue(x - a) < e\} \\
&PosReal \triangleq \{x \in Real : x > 0\} \\
&Limit(f, a) \triangleq \text{CHOOSE } l \in Real : \quad \text{This is the famous } (\epsilon, \delta) \text{ definition of the limit} \\
&\qquad\qquad \forall\, e \in PosReal : \exists\, d \in PosReal : \\
&\qquad\qquad\quad \forall\, x \in OpenBall(a, d) \setminus \{a\} : f[x] \in OpenBall(l, e) \\
&Derivative(f, a) \triangleq \text{LET } e \triangleq \text{CHOOSE } e \in PosReal : OpenBall(a, e) \subseteq \text{DOMAIN } f \\
&\qquad\qquad\quad \text{IN } Limit([x \in OpenBall(a, e) \setminus \{a\} \mapsto (f[x] - f[a])/(x - a)], a)
\end{aligned}$$

It is as easy to define non-computable functions as it is to define computable ones. Here is an example of a non-computable function:

$$\begin{aligned}
&dirichlet[x \in Real] \triangleq \\
&\quad \text{LET } IsRational(z) \triangleq \exists p \in Int, q \in Nat \setminus \{0\} : z = p/q \\
&\quad \text{IN} \quad \text{IF } IsRational(x) \text{ THEN } 1 \text{ ELSE } 0
\end{aligned}$$

As is, by the way, this one, even though at first it seems so simple and natural:

$$step[x \in Real] \triangleq \text{IF } x < 0 \text{ THEN } 0 \text{ ELSE } 1$$

While we can define any computable function (and many non-computable ones, too), those functions are *not* how we describe *computations* in TLA+. Unlike in specification languages based on functional programming, computations in TLA+ are not functions but dynamical systems (like ODEs). Instead, we use functions as data structures (associative arrays) or, like operators, as the primitive operations of our computations. For example, when we write a high-level specification and don't wish to model the dynamic behavior of say, the factorial subroutine, but would rather consider it a primitive operation, we specify it as a function.

Sequences and Tuples

A sequence is a finite or infinite ordered list of values. In TLA+, sequences are defined as functions on some prefix of $Nat \setminus \{0\}$, and so use 1-based indexing as is the usual practice in math, but not in most programming languages, where 0-based indexing is common.

The sequence of all even numbers in ascending order is just $[n \in Nat \setminus \{0\} \mapsto 2 * (n - 1)]$, and the sequence of all even numbers between 2 and 200 is $[n \in 1..100 \mapsto 2 * n]$.

The Sequences module contains several useful definitions for working with sequences. $Seq(S)$ is the set of all finite sequences over the set S (the set of infinite sequences is just $[Nat \setminus \{0\} \to S]$), and is defined like so:

$$Seq(S) \triangleq \text{UNION } \{[1..n \to S] : n \in Nat\}$$

The $Len(s)$ operator is the length of a sequence (which can be defined as $\text{CHOOSE } n \in Nat : \text{DOMAIN } s = 1..n$); the $\circ$ operator concatenates two sequences, the $Append(seq, x)$ operator appends the value x to the end of the sequence seq, $Head(seq)$ operator is the first element in the sequence ($seq[1]$) and $Tail(seq)$ is the tail (

$[i \in 1..Len(seq) - 1 \mapsto seq[i+1]]$). $SubSeq(seq, i, j)$ is the subsequence of seq between i and j inclusive, and $SelectSeq(seq, P(\_))$ is the sequence filtered to contain only those elements x such that $P(x)$ is true.

As strings are just sequences of characters, the `Sequences` module's $\circ$ operator also concatenates strings, $SubSeq$ selects a substring etc..

Defining the familiar `map` operation from functional programming on sequences is easy:

$$Map(F(\_), seq) \triangleq [i \in \text{DOMAIN } seq \mapsto F(seq[i])]$$

`flatmap` requires a bit more work and the use of a recursive operator (the inner helper operator is required as TLA+ does not allow recursive operators with operator parameters):

$$\begin{aligned} FlatMap(F(\_), seq) \triangleq\ & \text{LET RECURSIVE } Helper(\_) \\ & \quad Helper(s) \triangleq \text{IF } Len(s) = 0 \text{ THEN } \langle\rangle \\ & \qquad\qquad \text{ELSE } F(Head(s)) \circ Helper(Tail(s)) \\ & \text{IN } Helper(seq) \end{aligned}$$

TLA+ has special syntax for finite sequence literals, also called tuples (or lists, if you like). The tuple $\langle 10, \text{“hi ”}, [x \in N \mapsto x+1]\rangle$ is simply syntax sugar for:

$$[i \in 1..3 \mapsto \text{CASE } i = 1 \to 10 \ \square\ i = 2 \to \text{“hi ”} \ \square\ i = 3 \to [x \in N \mapsto x+1]]$$

If A and B are sets, then $A \times B$ is their ***Cartesian product***, $\{\langle a, b\rangle : a \in A, b \in B\}$. Similarly, $A \times B \times C = \{\langle a, b, c\rangle : a \in A, b \in B, c \in C\}$ etc.

For convenience, tuples can be used as quantified variables: instead of $\exists\, pair \in A \times B : pair[1] > pair[2]$, we can write $\exists\, \langle a, b\rangle \in A \times B : a > b$.

Functions that take multiple parameters are syntactic sugar defined in terms of tuples. We can write:

$$[x \in Nat, y \in \text{STRING } \mapsto x + Len(y)]$$

which is just syntax sugar for

$$[\langle x, y\rangle \in Nat \times \text{STRING } \mapsto x + Len(y)]$$

and, similarly, we can define the same function as

$$foo[x \in Nat, y \in \text{STRING}] \triangleq x + Len(y)$$

We can apply such functions with the expression $foo[3, \text{“hi ”}]$, which is just syntactic sugar for $f[\langle 3, \text{“hi ”}\rangle]$.

Records

The record $r \triangleq [a \mapsto 10, b \mapsto 20, c \mapsto 30]$ is syntax sugar for the function $[f \in \{\text{“a ”}, \text{“b ”}, \text{“c ”}\} \mapsto \text{CASE } f = \text{“a ”} \to 10 \ \square\ f = \text{“b ”} \to 20 \ \square\ f = \text{“c ”} \to 30]$, and $r.a$ is syntax sugar for $r[\text{“a ”}]$. EXCEPT also has special syntax for records, so a record that is equal to r but whose “field” b is 200, could be created with $[r \text{ EXCEPT } !.b = 200]$ (or with $[r \text{ EXCEPT } !.b = @ * 10]$). Finally, we have special syntax for sets of records:

$$[name : \text{STRING}, id : Nat]$$

is the set of all records with the fields $name$ and id, whose $name$ field is a string and whose id is a natural number.

Operators vs. Values

Should you express mathematical functions as TLA+ functions or as operators? Similarly, should you express relations as sets of pairs or as binary operators? That's entirely up to you. Expressing them as objects in the theory, i.e. as functions or sets, "reifies" them allowing to CHOOSE them or quantify over them, which may or may not be necessary in your specification. Using operators may be more convenient in terms of syntax (e.g. $x \preceq y$ looks nicer than $\langle x, y \rangle \in OrderRel$). Sometimes, the choice is dictated by the capabilities of the tools you'd like to use in verifying your specification. For example, the mechanical proof system TLAPS currently does not support recursive operator definitions, but it does support recursive functions (which are just syntax sugar for CHOOSE)[23]. Also, operators can express things that functions simply can't. For example, the sequence length operator or the set cardinality operator (which we'll see later) could not have been defined as functions because they're not functions in set-theory, as their domain is not a set but "too many" sets (i.e. a collection, a *class* that is cannot be constructed as a set in ZFC).

Constants

In addition to definitions, we may also declare *constants*. They are values (i.e. sets) or operators that are not given a definition, but we may specify assumptions that they must satisfy. One way to think about constants is as unknowns; yet another is to think of them as analogous to type parameters of polymorphic definitions in typed programming languages. In part 4 we'll see how those constants can be seen as an input or parameters when we learn of modules and module instantiation.

The name *constant* means that the declared value/operator is not a temporal one — we'll learn about temporal variables in part 3, when we learn how to write algorithms in TLA+.

Constants are introduced with the keyword CONSTANT (or its synonym CONSTANTS), and assumptions about them are introduced with the keyword ASSUME. For example:

$$\begin{array}{l} \text{CONSTANTS } M, N \\ \text{ASSUME } \land M \in Nat \\ \qquad\quad\ \land N \in Nat \end{array}$$

Assumptions can also be named ($\text{ASSUME } A1 \triangleq \ldots$) so that they can be referred to in proofs, where they're treated as axioms. In fact, the keyword AXIOM is synonymous with ASSUME (with the minor difference that the model checker TLC checks assumptions declared with ASSUME but not with AXIOM).

Now let's look at a more interesting example:

$$\begin{array}{l} \text{CONSTANTS } S, \_ \preceq \_ \\ \text{ASSUME } Poset(S, \preceq) \end{array}$$

We can think of S as a type parameter that has a partial order relation defined on it, similar to a typeclass in some functional programming languages, or interfaces in object-oriented ones.

From a formal logic perspective, constants are just free variables. However, as constants can be not only values but also operators, they are technically free second-order variables. The reason is that we can write something like the following (we'll introduce the THEOREM construct when we talk about **proofs**, but its meaning should be intuitive here):

$$\begin{array}{l} \text{CONSTANTS } C, D, F(\_) \\ \text{THEOREM } (C = D) \Rightarrow (F(C) = F(D)) \end{array}$$

This theorem treats F as *any* operator, i.e. as a free variable, which is equivalent to the quantified second-order formula (which we cannot write in TLA+):

$$\forall\, C, D, F : (C = D) \Rightarrow (F(C) = F(D))$$

However, TLA+ does not define an equality relation over those second-order objects (i.e., operators). Writing,

$$\begin{array}{l}\text{CONSTANT } F(\_), G(\_)\\ \text{ASSUME } F = G\end{array}$$

is a syntax error. One must write $\text{ASSUME } \forall\, x : F(x) = G(x)$.

Constants are one way in which TLA+ allows us to state second-order propositions even though it is a first-order logic. We'll say a more about that in the section on proofs.

Putting it All Together

Now let's use everything we've seen to define the $Cardinality(S)$ operator, which is the number of elements in the finite set S (the operator is provided by the standard module `FiniteSets`).

We can use a computer-sciency definition using recursion, based on the observation that the cardinality of a finite set is 1 plus the cardinality of the set after removing an arbitrary element from it:

$$\begin{array}{l}\text{RECURSIVE } Cardinality(\_)\\ Cardinality(s) \triangleq \text{IF } s = \{\} \text{ THEN } 0\\ \qquad\qquad\qquad\qquad \text{ELSE LET } x \triangleq \text{CHOOSE } x \in s : \text{TRUE}\\ \qquad\qquad\qquad\qquad\quad \text{IN} \quad 1 + Cardinality(s \setminus \{x\})\end{array}$$

We can also opt for a more "mathematical" definition. We'll define cardinality by finding a bijection from a range of natural numbers to the set.

$$Cardinality(s) \triangleq \text{CHOOSE } n \in Nat : \exists\, f \in [1..n \to s] : Bijection(f, s)$$

We can show off some other TLA+ features by using the following mathematical definition, which finds the largest natural number n such that there is injection (a one-to-one mapping) from $1..n$ to s:

$$\begin{array}{l}Cardinality(s) \triangleq\\ \quad \text{LET } TheLargestSuch(S, \_ \succ \_, P(\_)) \triangleq\\ \qquad\qquad \text{CHOOSE } x \in S : P(x) \land \forall\, y \in S : y \succ x \Rightarrow \neg P(y)\\ \quad \text{IN } TheLargestSuch(Nat, >, \text{LAMBDA } n : \exists\, f \in [1..n \to s] : Injection(f))\end{array}$$

What do we do with all this? We use TLA+ values to define the data of our program at any step of its execution, and the primitive operations that our computation can perform in one step. Data structures can be easily defined using the objects we learned: Arrays and lists can be modeled as sequences; structures can be modeled as records; maps can be modeled as functions, and sets as, well, sets.

What about things like the heap and the stack? What about processes or threads? In TLA+, you choose the level of abstraction. If you wish, you could model memory at a low level, as an array containing bytes (say, as natural numbers between 0 and 255) and write definitions for memory allocation and deallocation operations like `malloc` and `free`, as well as definitions that take care of encoding and decoding constructs like strings, arrays, floating point numbers etc. to and from bytes. In most circumstances, however, you're more likely to choose to model those constructs purely abstractly without worrying about their memory layout, or you could choose to find some middle ground where memory is allocated and deallocated in objects, but without worrying about lower-level representation, instead modeling interesting techniques for dealing with shared pointers like **separation logic**. As to processes, we'll see how those are modeled in part 3.

What good are the non-computable functions or operators (like $Inverse$ or $dirichlet$) we've seen when specifying real software systems? For that matter, even when a definition is computable, it may not suggest a feasible (**tractable**) way of computing it, which is just as bad. As we'll see in the following installments, such definitions can be convenient abstract representations of actual executable algorithms. They specify the *what* rather than the *how* of our program or some small part of it, and we can then choose to define the *how* and verify that it conforms to the *what*, or choose to leave things at that, and not worry about an implementation.

For example, when specifying some elaborate algorithm that relies on a sorting function, we may know that we already have an efficient library subroutine for sorting and we're not interested in the details of its implementation. So instead of specifying the sorting subroutine as an algorithm or defining it in a way that mimics an efficient implementation, we may choose to simply define its behavior:

$$\begin{aligned} Permutations(seq) \triangleq\ & \text{LET } N \triangleq Len(seq) \\ & \qquad PermsOfNums \triangleq \{f \in [1..N \to 1..N] : Injection(f)\} \\ & \text{IN } \{seq \bullet perm : perm \in PermsOfNums\} \end{aligned}$$

$$Ordered(seq, \_ \preceq \_) \triangleq \forall\, i, j \in 1..Len(seq) : i < j \Rightarrow seq[i] \preceq seq[j]$$

$$Sort(seq, \_ \preceq \_) \triangleq \text{CHOOSE } out \in Permutations(seq) : Ordered(out, \preceq)$$

To make things more concrete, let's consider a specification of linked lists. Say we decide to model the list as linked nodes, but we choose not to go into the detail of actual pointer arithmetic, and so we will not model memory directly.

Even though it is not necessary in TLA+, we can begin by specifying the "type" of our linked list node, where by type I mean the general structure of the node (without getting into memory layout). The elements of our list can be any members of the set S.

$$\begin{aligned} &\text{CONSTANT } S \\ &\text{RECURSIVE } Node \\ &\text{NULL} \triangleq \text{CHOOSE } x : x \notin Node \\ &Node \triangleq [elem : S, next : Node \cup \{\text{NULL}\}] \end{aligned}$$

We can later use this definition when we define algorithms over linked list to show that the type of the list is preserved. This turns the familiar notion of type into just another, relatively simple property we can specify and verify. For example, $x \in Node$ is just another proposition, like $a + b = 0$.

Now, let's define a view that presents a linked list as a sequence. For simplicity, let's assume we only allow finite lists. Remember, a finite sequence is just a function of some 1..n:

$$\begin{aligned} &\text{RECURSIVE } LenList(\_) \\ &LenList(ll) \triangleq \text{IF } ll = \text{NULL THEN } 0 \text{ ELSE } 1 + LenList(ll.next) \end{aligned}$$

$$\begin{aligned} ListAsSeq(ll) \triangleq\ & \text{LET RECURSIVE } View(\_, \_) \\ & \qquad View(i, node) \triangleq \text{IF } i = 1 \text{ THEN } l.elem \\ & \qquad\qquad\qquad\qquad \text{ELSE } View(i - 1, l.next) \\ & \text{IN } [i \in 1..LenList(ll) \mapsto View(i, ll)] \end{aligned}$$

$$\begin{aligned} &\text{RECURSIVE } SeqAsList(\_) \\ &SeqAsList(s) \triangleq \text{IF } s = \langle\rangle \text{ THEN NULL} \\ &\qquad\qquad\qquad\qquad \text{ELSE } [elem \mapsto s[1], next \mapsto SeqAsList(Tail(s))] \end{aligned}$$

The operators $ListAsSeq$[24] and $SeqAsList$ allow us to move between two levels of abstractions and write a theorem like

$$\text{THEOREM } \forall\, s \in Seq(S) : s = ListAsSeq(SeqAsList(s))$$

We could then prove the theorem or check it on a finite model in the model checker to verify that our concrete implementation of lists behaves like an abstract sequence. This is a very simple form of something called a refinement relation between specifications; in particular, this is a *data* refinement, as it pertains to a data structure. In parts 3 and 4 we'll see what refinement means exactly, as well as how we can use refinement relations of algorithms.

I hope that it is now clear why I opened by saying that this is the least essential part of TLA+: any other system for defining data and operations on it would do. For example we could have used HOL instead of ZFC, for a slightly different set of tradeoffs, although when it comes to actual work, the similarities would be greater than the differences; after all, math ıs math. The interesting — and unique — power of TLA+ comes from TLA, the logic used to define algorithms, which we'll cover in the upcoming installments. Nevertheless, the data formalism chosen by TLA+ is one that is both powerful and relatively familiar to engineers, most of whom learned about sets and functions in a discrete math course.

In fact, this system is powerful enough that we don't really need anything else. We could define computations as as discrete dynamical processes that change state in time by modeling (discrete) time as the index of an infinite sequence[25]. However, this is not the path taken by TLA+, which offers a more structured and convenient way of describing algorithms and computations using the temporal logic of actions.

Proofs

The most important skill when using TLA+ (or any other specification language) is the ability to express your ideas precisely in the language of mathematics. If there is an automatic verification tool for your language of choice — like the TLC model checker for TLA+ — this is often all you need. However, the most important capability of logic after serving as a precise language, is the ability to use syntactic manipulation rules — often called inference rules, deduction rules or a proof calculus — to write proofs, using the syntax of the logic to arrive at truth.

The TLA+ proof language is more readable than any I've seen, so I will cover its main features, as they're important from a design, if not theory, perspective. The proof language is a relatively late addition to TLA+, added in 2008, along with an early version of TLAPS, the mechanical proof system. The design principles of the proof language, with comparisons to other proof languages, is covered in its **introduction paper**. Details of usage are covered in detail in section 7 of the **TLA+2 Preliminary Guide**, and a good tutorial, which includes the use of the mechanical proof system, is found in "The TLA+ Proof Track" of the **hyperbook**. More information about TLAPS can be found on **its website**. TLAPS can mechanically verify a useful subset of TLA+ proofs and specifications. TLAPS is not a standalone proof assistant, but a frontend, which uses multiple automated solvers and the **Isabelle proof assistant** as backends. It can even check and certify the automated provers in Isabelle (currently only Zenon, but certifying the SMT solvers is planned). You may be interested in reading some of the papers explaining TLAPS's operation **here** and **here**. An interesting real-world specification of a real-time OS kernel scheduler has recently **been mechanically proven with TLAPS**.

In practice, when writing real specifications, you'll find that using the model-checker has a much higher return-on-investment than writing proofs (by an order of magnitude). Not only does the model checker save you a great deal of effort, but you can only prove things that are true, and very often, you'll make assertions that are wrong. The model checker will let you know exactly *why* it is that your assertions are wrong; struggling with a proof may lead you to your mistake, but only after considerable effort.

Nevertheless, sometimes you may find it useful to write a proof. TLA+ has a rich structured and declarative proof language, based on Lamport's interesting ideas for a structured proof system that assists both in the readability and the rigor of mathematical proofs. The system is detailed in ***How to write a 21st century proof*** (with an older discussion in ***How to Write a Proof***), which is an interesting read regardless of whether or not you're interested in TLA+. Lamport first gave a talk explaining his ideas on structured proofs in 1991. **It was not well received**:

> *Lots of people jumped on me for trying to take the fun out of mathematics. The strength of their reaction indicates that I hit a nerve. Perhaps they really do think it's fun having to recreate the proofs themselves if they want to know whether a theorem in a published paper is actually correct, and to have to struggle to figure out why a particular step in the proof is supposed to hold.*

Twenty years later, attitudes have changed, although not practice:

> *The talk was received much more calmly than my earlier one, and the mathematicians were open to considering that I might have something interesting to say about writing proofs. Perhaps in the last 20 years I have learned to be more persuasive, or perhaps the mathematicians in the audience had just grown older and calmer. In any case, they were still not ready to try changing how they write their own proofs.*

TLA+'s proof language is rich with constructs designed to make it easier for people to read and write proofs, and can be learned in a day or two once you know TLA+. The proof language deviates somewhat from the relative minimalism that characterizes the rest of TLA+, and focuses more on readability that evokes prose proofs as much as possible, using a wide selection of keywords — over 25 — many of which are synonyms and syntactic sugar. I will not cover them all, but just highlight the core elements.

The paper introducing the proof language[26] reads:

> *The goal of the language is to make proofs easy to read and write for someone with no knowledge of how the proofs are being checked. This leads to a mostly declarative language, built around the uses and proofs of assertions rather than around the application of proof-search tactics.*

While formal proofs are some sequence (or a tree) of applications of the logic's inference rules, writing proofs in this way is untenable, as the rules are too primitive, and each moves you forward towards the goal too slowly. So mechanical proof assistants often have a higher-level proof languages. Some, like Coq, have an imperative proof language, where a proof is constructed by chaining *tactics*. You can think of tactics as macros for the primitive inference rules. Others, like TLA+ have a declarative proof language (and Isabelle has both imperative and declarative proof languages). Declarative proofs are designed to be easily readable by people, and resemble how humans write proofs. At every step of the proof they only list those facts — axioms, other lemmas or theorems, and previous proof steps — required to prove the current statement, without explaining *how*. They basically list the input and the output for some chain of deduction steps, leaving the actual steps for the tool to figure out. This, however, means that there is a difference between a *legal* proof and a *useful* proof. For example, the following is (probably) a formal proof in TLA+

$$\begin{aligned}&\text{THEOREM } FermatsLastTheorem \triangleq \\ &\qquad \neg\exists a, b, c, n \in Nat \setminus 0 : n > 2 \land a^n + b^n = c^n \\ &\text{PROOF BY } PeanoAxioms\end{aligned}$$

but, of course, it will convince no mathematician and certainly not the mechanical proof system, which cannot deduce the steps from the input to the output. This, however:

$$\begin{aligned}&\text{THEOREM } GoldbachConjecture \triangleq \\ &\quad \forall n \in Nat : n > 2 \land n\%2 = 0 \Rightarrow \\ &\qquad\qquad \exists p, q \in Nat : IsPrime(p) \land IsPrime(q) \land p + q = n \\ &\text{PROOF BY } PeanoAxioms\end{aligned}$$

may or may not be a valid proof, because we don't know whether Goldbach's conjecture is a theorem or not, and if it is, whether or not the Peano axioms suffice to prove it.

A useful proof is one that can be relatively easily verified either by a human reader and/or by the mechanical proof system. This happens when it is obvious how proof goal (theorem/lemma or a proof step) is entailed by the facts listed as its proof. What “obvious" means depends, of course, on the capabilities of the person or algorithm verifying the proof, where “verification" ultimately means becoming convinced (perhaps even supplying a formal certificate — the long chain of primitive inference rules connecting the assumptions to the consequence) that the goal is indeed entailed by the facts listed.

Both imperative and declarative proof languages have their pros and cons (and both are quite tedious). Imperative proofs are so hard to read that they are effectively meaningless to a human; on the other hand, they let you know exactly what the verifier knows at each step, so they are easier to write by comparison. Declarative proofs are easy to read, but writing them is often a game of trial and error, where you supply more detail and hope that would do the trick, while the feedback you get is just whether the verifier has managed to prove the goal from the supplied facts or not.

The TLA+ proof language does not presuppose any specific capabilities on the part of the verifier, be it human or mechanical, instead allowing hierarchically refining every proof step by making it into a proof goal of its own and recursively breaking it down into another, hopefully simpler, series of proof steps, and so on until finally we get proof steps that are each obvious enough for the verifier. This is the idea behind Lamport's structured hierarchical proofs.

At any point in a TLA+ proof, there is a current *proof obligation*, which claims that a proof *goal* is entailed by a set of facts (other theorems axioms or proof steps) and definitions called a *context*. A *proof goal* is anything that requires proof. It can be a theorem introduced with the THEOREM keyword (or one of its synonyms LEMMA, COROLLARY, PROPOSITION), or, recursively, a proof step of one. A (declarative) proof is either omitted (with the OMITTED clause), stated to be OBVIOUS if it is a direct result of logical inference rules and other built-in axioms that don't require further assumptions, or supplied in a BY (or, optionally, PROOF BY) clause, which lists all facts from which the obligation can be deduced as well as any definitions that the obligation depends on and must be examined ("expanded").

A theorem can be stated in two ways. It can be a logical proposition given as a formula (like $\forall r, s, t : r \subseteq s \land s \subseteq t \Rightarrow r \subseteq t$ or the above statement of Fermet's Last Theorem), or as a more powerful, more general $\text{ASSUME}/\text{PROVE}$ pair, where the ASSUME clause lists some assumptions that, if true, would entail the consequence in the PROVE clause. The $\text{ASSUME}/\text{PROVE}$ pair basically means $(\text{ASSUME clause}) \vdash (\text{PROVE clause})$. For example:

$$\begin{array}{l}\text{THEOREM } ModusPonens \triangleq \text{ASSUME NEW } P, \text{NEW } Q, \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad P, P \Rightarrow Q \\ \qquad\qquad\qquad\qquad\qquad\quad \text{PROVE } Q \\ \quad \text{PROOF OBVIOUS}\end{array}$$

or:

$$\begin{array}{l}\text{THEOREM } ImplicationIntr \triangleq \\ \quad \text{ASSUME NEW } P, \text{NEW } Q, \\ \qquad\qquad \text{ASSUME } P \\ \qquad\qquad \text{PROVE } \quad Q \\ \qquad \text{PROVE } P \Rightarrow Q \\ \quad \text{OBVIOUS}\end{array}$$

The keyword NEW introduces a new variables, as in "let P be any...". NEW by itself is shorthand for NEW CONSTANT, or just CONSTANT, signifying that the variable is a constant rather than a temporal one that can refer to different values at different times in an algorithm; this is a CONSTANT in the exact sense as we learned above in the section ***Constants***. If unbounded, it means any value or any (non-temporal) operator or formula. It is a second-order free variable. $\text{ASSUME } A \text{ PROVE } B$ is a proposition but it is not a TLA+ formula; it doesn't have a model in the TLA+ logic, so the formulas of the logic are still all first-order. See the discussion to follow, as well as in the section ***First Order Logic and Other Orders***, for the reasoning behind this design.

For example, the following theorem demonstrates a bounded NEW, as well as introducing parameterized formulas:

$$\begin{array}{l}\text{THEOREM } ForallIntr \triangleq \\ \quad \text{ASSUME NEW } P(\_), \text{NEW } S, \\ \qquad\qquad \text{ASSUME NEW } x \in S \\ \qquad\qquad \text{PROVE } \quad P(x) \\ \quad \text{PROVE } \forall x \in S : P(x) \\ \text{OBVIOUS}\end{array}$$

Note how we assume the existence of a theorem (a true proposition) in the nested $\text{ASSUME}/\text{PROVE}$.

The ability to write second-order theorems is important in a proof system, as it allows the user to make general, reusable lemmas for use in proofs. For example, an important theorem (exported by the `NaturalsInduction` module provided with TLAPS) that can be used in many circumstances is the following, which defines induction over the naturals:

$$\begin{array}{l}\text{THEOREM } NatInduction \triangleq \\ \quad \text{ASSUME NEW } P(\_), \\ \qquad\qquad P(0), \\ \qquad\qquad \forall\, n \in Nat : P(n) \Rightarrow P(n+1) \\ \quad \text{PROVE } \forall\, n \in Nat : P(n)\end{array}$$

Instead of $\forall\, n \in Nat : P(n) \Rightarrow P(n+1)$ above, we could have used the nested $\text{ASSUME NEW } n \in Nat, P(n) \text{ PROVE } P(n+1)$.

We can then use that theorem to prove the following (silly) one:

$$\begin{array}{l}\text{THEOREM ASSUME NEW } x \in Nat \\ \qquad\qquad\qquad \text{PROVE} \quad x = 0 \lor x - 1 \geq 0 \\ \quad \text{BY } NatInduction\end{array}$$

(OBVIOUS would have sufficed in this case, both for a human reader as well as for TLAPS, and we could have stated the theorem more simply as the formula $\forall x \in Nat : x = 0 \lor x - 1 \geq 0$). Notice there is no need to actually define an operator of the form $P(n)$; it is automatically inferred from the formula $x = 0 \lor x - 1 \geq 0$ that it can be understood as an operator parameterized by x.

Theorems of the kind we've seen can be introduced as axioms to define inference rules for new logical connectives (for example, for something like **separation logic**). However, built-in axioms cannot be removed (so we can't remove the law of excluded middle as an axiom to ensure our reasoning is constructive). Axioms are declared with the keyword AXIOM or with ASSUME, the construct we covered in the section about **constants**.

While TLAPS has been designed to be fully backend-agnostic, when using it you may sometimes find it necessary to use specific solver features. For example, $\text{BY } Z3$ can tell TLAPS to use the **Z3 SMT solver** to discharge (verify) the particular proof, and you can even — although it's not recommended and you're unlikely to need it in practice — make use of Isabelle tactics. In fact, the proof of the $NatInduction$ theorem above provided with TLAPS is $\text{BY } IsaM(\text{``(intro natInduct, auto) ''})$.

Let's look at an example that requires expanding a definition in the proof. The TLAPS solvers are powerful enough to prove the following theorems about the algebraic structures we defined in the section **Set Fundamentals** directly from their definition, without any added assistance:

$$\begin{array}{l}\text{THEOREM } UniqueIdentity \triangleq \\ \quad \text{ASSUME NEW } M, \text{NEW } \_ \cdot \_, Monoid(M, \cdot) \\ \quad \text{PROVE } \forall\, x, y \in M : \land\, \forall\, a \in M : x \cdot a = a \land a \cdot x = a \\ \qquad\qquad\qquad\qquad\qquad \land\, \forall\, a \in M : y \cdot a = a \land a \cdot y = a \\ \qquad\qquad\quad \Rightarrow x = y \\ \text{BY DEF } Monoid\end{array}$$

$$\begin{array}{l}\text{THEOREM } UniqueInverse \triangleq \\ \quad \text{ASSUME NEW } G, \text{NEW } \_ \cdot \_, Group(G, \cdot), \\ \qquad\qquad \text{NEW } id \in G, \forall\, a \in G : id \cdot a = a \land a \cdot id = a \\ \quad \text{PROVE} \quad \forall\, a, b, c \in G : (a \cdot b = id \land b \cdot a = id \land a \cdot c = id) \Rightarrow b = c \\ \text{BY DEF } Group, Monoid, Semigroup\end{array}$$

BY DEF $Monoid$ says that the proof relies on the content of the definition of $Monoid$, as no facts are used nor any definitions expanded unless we explicitly say so (inside long proofs — which we'll get to right away — there are ergonomic constructs designed to save us repetitive mentions of facts or definitions in multiple proof steps). The second theorem requires examining the inner definitions of $Monoid$ and $Semigroup$ used in the definition of $Group$, as it requires the associativity property, defined in $Semigroup$.

More interesting theorems cannot be proven satisfactorily with a single BY clause, and must be broken down into smaller proof steps. Each proof step states a proposition and requires proof (or OMITTED) with a BY clause, or, recursively, by a nested series of proof steps, thus forming a tree-like hierarchy. Every step is labeled with "$\langle depth \rangle name$.", where *depth* is the depth of the step in the hierarchy, and *name* is an optional label for the step, usually just the index of the step in the sequence. A named step can be referenced as a fact in a BY clause as if it were a named theorem. Every sequence of proof steps, at any level of the tree, must end with a QED step, whose goal is the parent proposition of the sequence, known as the *current goal*. Here's an example of a proof structure:

```
THEOREM ...
    ⟨1⟩1. ... BY ...
    ⟨1⟩2. ...
          ⟨2⟩1. ... BY ...
          ⟨2⟩2. ... BY ...
          ⟨2⟩3. QED BY ...
    ⟨1⟩3. ...
          ⟨2⟩1.  ...
                ⟨3⟩1. ... BY ...
                ⟨3⟩2. QED BY ...
          ⟨2⟩2. ... BY ...
          ⟨2⟩3. QED BY ...
    ⟨1⟩4. QED BY ...
```

Note that a proof may contain multiple steps with the same name (e.g. $\langle 2 \rangle 1$, $\langle 2 \rangle 2$, $\langle 2 \rangle 3$. all appear twice in the outline above), however, there can be no ambiguity when referencing steps, as you can only reference steps appearing previously in the same sequence or in higher levels of the hierarchy. The idea of a hierarchical proof is that each nested sequence provides the details of how its goal (i.e., parent) is proven, until the leaf nodes are obvious enough. If, as a reader, you're not interested in the details or the step is obvious enough without the more detailed steps, you can collapse the tree in the TLA+ Toolbox. This hierarchical structure results in very readable proofs, as you can easily grasp the outline of the proof, and then delve deeper for more detail if you like.

The proof language has some constructs that allow for a more natural expression of the proof (similar to common techniques in prose proofs), or to improve readability. Instead of listing the set of facts and definitions that are supposed to entail the goal, the keyword USE introduces facts and definitions that are then implicitly added to all subsequent steps in the same step sequence. The matching HIDE keyword does the opposite, removing implicit facts from consideration. Because USE doesn't introduce a proposition, it is usually written in an unnamed step (e.g. $\langle 2 \rangle$ as opposed to, say, $\langle 2 \rangle 5$), as there's never a reason to reference it.

A more interesting ergonomic construct is SUFFICES. Say that assuming *A*, we want to prove *C*, and that we could do that by first proving $A \vdash B$ and then $B \vdash C$. Sometimes it is more convenient (to guide the reader in the intent of the proof) to first prove $B \vdash C$ and only then $A \vdash B$. In a prose proof, we usually say "suffices to prove *B* because..." and then move on to proving *B*. We can do the same in TLA+ with the SUFFICES construct. When used in an unnamed step, it changes the current goal as seen in the example below, taken from the hyperbook, which demonstrates the use of SUFFICES as well as gives you a sense of what a real proof looks like. The example proves some theorems that can be mechanically checked by TLAPS about the GCD operator we defined above in the section **Some Important Sets**[27]:

```
THEOREM ∀ m, n ∈ Nat \ {0} : GCD(m, n) = GCD(n, m)
  BY DEF GCD

THEOREM ∀ m ∈ Nat \ {0} : GCD(m, m) = m
  ⟨1⟩ SUFFICES ASSUME NEW m ∈ Nat \ {0}
               PROVE  GCD(m, m) = m
    OBVIOUS
  ⟨1⟩1. Divides(m, m)
    BY DEF Divides
  ⟨1⟩2. ∀ i ∈ Nat : Divides(i, m) ⇒ i ≤ m
    BY DEF Divides
  ⟨1⟩ QED
    BY ⟨1⟩1, ⟨1⟩2 DEF GCD, SetMax, DivisorOf

THEOREM ∀ m, n ∈ Nat \ {0} : n > m ⇒ GCD(m, n) = GCD(m, m − n)
  ⟨1⟩ SUFFICES ASSUME NEW m ∈ Nat \ {0}, NEW n ∈ Nat \ {0},
                      n > m
               PROVE  GCD(m, n) = GCD(n, n − m)
    OBVIOUS
  ⟨1⟩ ∀ i ∈ Int : Divides(i, m) ∧ Divides(i, n) ≡ Divides(i, m) ∧ Divides(i, n − m)
    BY DEF Divides
  ⟨1⟩ QED
    BY DEF GCD, SetMax, DivisorOf
```

The CASE construct helps writing a proof by cases. For example, instead of:

```
⟨1⟩1. ∀ n ∈ Int : P(n)
      ⟨2⟩1. ASSUME n ≥ 0 PROVE P(n)
            ...
      ⟨2⟩2. ASSUME n < 0 PROVE P(n)
            ...
      ⟨2⟩3. QED
            BY ⟨2⟩1, ⟨2⟩2
```

you can write:

```
⟨2⟩1. ∀ n ∈ Int : P(n)
      ⟨2⟩1. CASE n ≥ 0
            ...
      ⟨2⟩2. CASE n < 0
            ...
      ⟨2⟩3. QED
            BY ⟨2⟩1, ⟨2⟩2
```

Finally, let's take a look at the PICK construct. If we have an assumption of the form $\exists x : P(x)$ we can use it by PICKing a fresh variable x for which $P(x)$ is assumed (it works similarly to NEW x, $P(x)$). Here is an example adapted from the TLAPS documentation. In addition to PICK, it makes use of SUFFICES and also demonstrates a proof of contradiction[28]:

$$
\begin{array}{l}
\text{THEOREM } \neg\exists\, x \in Nat : x + 1 = 0 \\
\quad \langle 1 \rangle \text{ SUFFICES ASSUME } \exists\, x \in Nat : x + 1 = 0 \\
\qquad\qquad\qquad\qquad \text{PROVE FALSE} \\
\quad\ \text{OBVIOUS} \\
\quad \text{Goal is now FALSE, i.e., a contradiction} \\
\quad \text{The assumption } \exists\, x \in Nat : x + 1 = 0 \text{ is now in the implied list of facts} \\
\quad \langle 1 \rangle \text{ PICK } u \in Nat : u = -1 \\
\qquad \langle 2 \rangle 1.\ \forall\, n \in Nat : n + 1 = 0 \Rightarrow n = -1 \\
\qquad\quad \text{OBVIOUS} \\
\qquad \langle 2 \rangle 2.\ \text{QED BY } \langle 2 \rangle 1 \\
\quad \text{We now have } u \in Nat : u = -1 \text{ in the implied list of facts} \\
\quad \langle 1 \rangle \text{ QED BY } -1 \notin Nat
\end{array}
$$

There are quite a few other constructs that capture common proof techniques; you can find a list of them **here** (as you can see, their meaning is precisely defined by the more basic constructs we've covered). There are also constructs that allow adding local definitions to a proof, a scheme that allows naming and referencing parts of formulas, and a convenient syntax for writing proofs by a sequence of equalities or inequalities.

For an example of the last, if TLAPS weren't able to prove the theorem $UniqueInverse$ automatically — or if the proof, although obvious to the mechanical prover, is not obvious to the human reader — we could help the system or the reader by writing the following detailed proof, where @ refers to the right-hand side of the relation mentioned in the previous step:

$$
\begin{array}{lll}
\text{THEOREM } UniqueInverse \triangleq & & \\
\qquad \text{ASSUME NEW } G, \text{NEW } \_ \cdot \_, Group(G, \cdot), & & \\
\qquad\qquad \text{NEW } id \in G, \forall\, a \in G : id \cdot a = a \wedge a \cdot id = a & & \\
\qquad \text{PROVE} \quad \forall\, a, b, c \in G : (a \cdot b = id \wedge b \cdot a = id \wedge a \cdot c = id) \Rightarrow b = c & & \\
\quad \langle 1 \rangle \text{ SUFFICES ASSUME NEW } a \in G, \text{NEW } b \in G, \text{NEW } c \in G, & & \\
\qquad\qquad\qquad\qquad a \cdot b = id, b \cdot a = id, a \cdot c = id & & \\
\qquad\qquad\qquad \text{PROVE} \quad b = c & & \\
\qquad\ \text{OBVIOUS} & & \\
\quad \langle 1 \rangle\ b & = b \cdot id & \text{OBVIOUS} \\
\quad \langle 1 \rangle\ @ & = b \cdot (a \cdot c) & \text{OBVIOUS} \\
\quad \langle 1 \rangle\ @ & = (b \cdot a) \cdot c & \text{BY DEF } Group, Monoid, Semigroup \\
\quad \langle 1 \rangle\ @ & = c & \text{OBVIOUS} \\
\quad \langle 1 \rangle \text{ QED} & & \text{OBVIOUS}
\end{array}
$$

The appendix of ***How to write a 21st century proof*** contains a full, mechanically verified, TLA+ proof of a simple theorem in calculus along with all necessary definitions.

While TLA+ *can* be used to prove general mathematical theorems, it was not designed for that task. Mathematicians interested in formal proofs would be better served by tools designed for that task, like Isabelle or Coq. Engineers who choose to write formal proofs are encouraged not to waste their time on proving mathematical theorems, but to either use the library of proven mathematical theorems supplied with TLAPS or to merely state a mathematical theorem and omit the proof. They should concentrate on proving the correctness of their algorithms or system designs. In part 3 we'll see how that's done.

Conclusion

Using formal math to precisely define objects and their properties is pretty straightforward, can be learned easily by engineers, and doing that forms the core of the work in TLA+, and, in fact, in any specification language or proof assistant.