TLA+ in Practice and Theory
Part 4: Order in TLA+

15 Jun 2017

This is the last installment in the series. ***Part 1****,* ***Part 2****,* ***Part 3***

If you find TLA+ and formal methods in general interesting, I invite you to visit and participate in the new ***/r/tlaplus*** *on Reddit.*

In part 3 we saw how TLA, the temporal logic of actions serves as a universal mathematical framework for reasoning about discrete dynamical systems, much like ordinary differential equations are a framework for specifying and reasoning about continuous systems. Now, in our final installment in the series, we will mostly focus on those capabilities of TLA+ that are unique to the needs of reasoning about *engineered* discrete systems in general, and software in particular. We will discuss encapsulation, instantiation and information hiding, see how they can be used for composition and specifying open systems (think libraries), and introduce important notions of equivalence and order relations between algorithms.

In this post, as in part 3, I will sometimes use the word "set" loosely (as in a set of states or a set of behaviors) to mean any collection, rather than set in the ZFC set-theory sense. I wish to remind you yet again that, while comprehensive, this series is a deep-dive into the theory of TLA+ rather than a tutorial. Understanding of subjects we cover is not necessary to write good TLA+ specifications.

Encapsulation

An important ability in programming is encapsulation, or modularity, which allows us to hide internal implementation details of components. Encapsulation can also serve as an important form of abstraction, letting us instantiate components with different parameters. In mathematics, encapsulation is not as important as in programming because mathematical theorems tend to be relatively short. While software specifications are not as large as programs, they can be far more elaborate than theorems used by mathematicians[1], and so requires encapsulation. Encapsulation together with parameterized instantiation of components is also important from a theoretical perspective as a means to study composition in software.

Modules

Before we move on with the theory, we will cover the last missing technical detail of TLA+, the means by which encapsulation is achieved: the module system. Every TLA+ definition or declaration must reside inside a *module*. A module serves as (parameterized) namespace for the definitions, constant and variable declarations, and theorems and axioms it contain.

Modules are enclosed like so:

$$\text{MODULE } Foo$$

$$\ldots \text{definitions} \ldots$$

and can be nested to any depth. All definitions and declarations in an enclosing module made prior to a nested module are imported into the inner module.

Every name (definition or declaration) in a module is *exported* — i.e., made accessible to other modules — by default, unless prefaced by the keyword LOCAL.

The first (and only first) line in a module may be (and often is) $\text{EXTENDS}\ M1, M2, \ldots$ where $M1, M2, \ldots$ are modules. It imports all exported symbols from those modules as if they were all defined in the the extending module.

A module can be *instantiated* once or multiple times within another module. Every *declaration* — constant or variable, i.e. a free ordinary or temporal variable — in the instantiated module is treated as a parameter and must be assigned in a WITH clause:

$$MyFoo \triangleq \text{INSTANCE}\ Foo\ \text{WITH}\ d_1 \leftarrow e_1, \ldots, d_n \leftarrow e_n$$

where $d_1 \ldots d_n$ are the declarations in the Foo module, and $e_1 \ldots e_n$ are the meanings given to them. If a symbol with the same name is already defined or declared in the scope where INSTANCE is written, it need not be mentioned explicitly in the WITH clause but is assigned by default to the symbol with same name in the instanting scope.

A module can't be instantiated without all of its declarations assigned (it is a syntax error). Assignment substitutes the given meaning in place of the declarations. Like parameterized definitions (operators), instantiation is substitution, only at a module level. From a logic perspective, instantiation binds the free variables in the instantiated module (although it may bind them to free variables in the instantiating module, thus leaving them free).

Once a module is instantiated, its definitions can be accessed with the instance name followed by !, as in $MyFoo!Ordered(\ldots)$ (infix operator can be referenced like so: $MyFoo! \preceq (a, b)$). For each definition in $MyFoo$, $MyFoo!Def$ means the same as Def inside Foo, except with all of the declarations of Foo replaced by the meaning given to them in the WITH clause (or implicitly). For brevity in the discussion, if module Foo contains some definition X, we may sometimes call the definition after the substitution, $\overline{X}$, instead of $MyInstance!X$.

For example, in part 3 we defined "polymorphic" sorting algorithms that had the constants S and $\_ \preceq \_$ declared. If the $QuickSort$ algorithm resides in the QS module, then we can instantiate the module like this (assuming that the temporal variables A, $A0$, $done$ are declared in the instantiating module, and are therefore substituted implicitly):

$$IntegerQS \triangleq \text{INSTANCE}\ QS\ \text{WITH}\ S \leftarrow Int, \preceq \leftarrow \leq$$

Then, $IntegerQS!QuickSort$ will be a sorting algorithm for sequences of integers.

We can also instantiate a module inside a parameterized definition, like so:

$$IntQS(\_ \preceq \_) \triangleq \text{INSTANCE}\ QS\ \text{WITH}\ S \leftarrow Int$$

and then $IntQS(\leq)!QuickSort$ sorts in ascending order, while $IntQs(\geq)!QuickSort$ sorts in descending order. Note how we don't need to write $\text{WITH}\ \ldots, \preceq \leftarrow \preceq$, as the name $\preceq$ exists in the scope of the INSTANCE construct, in this case as an argument of the operator $IntQS$.

We could then write the theorem:

$$\text{THEOREM}\ IntQs(\geq)!QuickSort \Rightarrow \Box(done \Rightarrow Ordered(A, \geq))$$

Whereas $M1 \triangleq \text{INSTANCE}\ M \ldots$ creates a *named* instance — $M1$ — of the module M, we can also create *unnamed* instances simply with $\text{INSTANCE}\ M \ldots$ This has the effect of importing all the definitions and declarations in M into the instantiating module (with their assigned meanings given in the WITH clause or implicitly), and they can then be accessed without qualification. This, in a sense, turns the instantiating module itself into an instance of the instantiated module. $\text{LOCAL INSTANCE}\ M \ldots$ has the same effect, except that the symbols of M become local in the instantiating

module, and are not exported. Of course, a module can only be instantiated anonymously once within another module, otherwise the names imported by its multiple instantiations would clash.

Unnamed INSTANCE is similar to EXTENDS, except that INSTANCE requires that all declarations in the instantiated module be assigned (or re-declared), whereas EXTENDS imports all declarations as if they were declared in the extending module (for example, if the instantiated module contains $\text{CONSTANT } C$, then the result of EXTENDS would be as if $\text{CONSTANT } C$ had been declared in the instantiating module, while INSTANCE requires that C be already declared or defined in the instantiating module). Also, EXTENDS does not support substitution, and it allows naming multiple modules at once.

Because module names can only be mentioned in a MODULE, EXTENDS, or INSTANCE clause, and nothing but a module name can appear in those constructs, there is no possibility of confusing module names with other names (definitions, declarations), so TLA$^+$ allows using the same name for a module and for a definition or a declaration.

Modules are not first-class objects in TLA$^+$ in the sense that they cannot be passed as arguments to operators like ML modules or objects in object-oriented programming languages. However, the pattern,

$$
\begin{array}{l}
\text{MODULE } M1M2 \\
\text{LOCAL INSTANCE } M1 \\
\text{INSTANCE } M2
\end{array}
$$

$$MyM2 \triangleq \text{INSTANCE } M1M2$$

is very much analogous to (the illegal) $MyM2 \triangleq \text{INSTANCE } M2(M1)$, presuming that the "signatures" of $M1$ and $M2$ match, namely that $M1$ exports the names required by $M2$'s declarations.

Modules can be used to abstract away encoding ("implementation") details of mathematical objects. Here's a (very) contrived example that abstractly defines vectors without specifying any particular encoding:

$$
\begin{array}{l}
\text{MODULE } VectorSpace \\
\text{LOCAL INSTANCE } Naturals \quad \text{Like EXTENDS , but imported symbols aren't exported} \\
\text{CONSTANT } Dim, S \\
\text{CONSTANTS } ConsVector(seq), Elem(v, i) \\
\text{CONSTANTS } ElemPlus(\_,\_), ElemMinus(\_,\_) \\
\text{ASSUME } \forall\, a, b \in S : ElemPlus(a, b) = ElemPlus(b, a) \\
\text{ASSUME } \forall\, a, b, c \in S : ElemPlus(a, b) = c \equiv ElemMinus(c, a) = b \\
\\
\text{The vector space} \\
VS \triangleq \{ConsVector(seq) : seq \in [1..Dim \to S]\} \\
\text{Vector addition} \\
v1 \mathbin{++} v2 \triangleq ConsVector([i \in 1..Dim \mapsto ElemPlus(Elem(v1, i), Elem(v2, i))]) \\
\text{Vector subtraction} \\
v1 \mathbin{--} v2 \triangleq ConsVector([i \in 1..Dim \mapsto ElemMinus(Elem(v1, i), Elem(v2, i))]) \\
\\
\text{THEOREM } \forall\, u, v, w \in VS : (u \mathbin{++} v = w) \equiv (w \mathbin{--} v = u) \\
\quad \text{PROOF } \ldots
\end{array}
$$

Now we'll declare a $Vector3D$ module that extends $VectorSpace$ by using an unnamed instantiation of $VectorSpace$:

```
MODULE Vector3D
CONSTANTS ElemPlus(_, _), ElemMinus(_, _)
LOCAL ConsVector1(seq) ≜ [x ↦ seq[1], y ↦ seq[2], z ↦ seq[3]]
LOCAL Elem1(v, i) ≜ CASE i = 1 → v.x □ i = 2 → v.y □ i = 3 → v.z

Export "opaque" operators
ConsVector(seq) ≜ ConsVector1(seq)
Elem(v, i) ≜ Elem1(v, i)

INSTANCE VectorSpace WITH Dim ← 3
```

The module assigns the dimension as well as the construction and destruction operations — that, for whatever reason, use a record to encode the 3D vector — but does leaves element addition/subtraction free constants. When $Vector3D$ is instantiated, they will need to be assigned.

The module hides the encoding of the vector. If we instantiate it, like so:

```
EXTENDS Reals
V3D ≜ INSTANCE Vector3d WITH S ← Real, ElemPlus ← +, ElemMinus ← −
```

and then construct a 3D vector, $v \triangleq V3D!ConsVector(\langle 4.5, 1.6, -18.99 \rangle)$, we cannot use $v.x$ because the encoding is hidden from us (in other words, the $Vector3D$ does not export any theorem that says that its vectors are records in the set $[x : S, y : S, z : S]$)[2]. In addition, because we've hid $ConsVector$ and $Elem$'s definition inside LOCAL operators, the details of the definitions cannot be examined by proofs with BY DEF.

If a module Foo contains ASSUME A (for some assumption A) and a THEOREM T and is instantiated with $MyFoo \triangleq$ INSTANCE Foo WITH ..., then $MyFoo!T$ would refer to the theorem ASSUME $\overline{A}$ PROVE $\overline{T}$ (remember, $\overline{A}$ and $\overline{T}$ denote A and T after the substitutions supplied in the WITH clause). In other words, all assumptions in a module are added as its theorem's assumptions when instantiated. That's because the theorems in a module depends on all assumptions in the module, which they treat as axioms.

It is the assignment of temporal meaning to temporal variable declarations that will serve as the basis for the theoretical discussion that will follow. The next two sections cover some technical corner cases of substitution. If you only care about the big picture, you may skip them and go directly to the **chapter on composition**.

Substitutivity of Operators

There is the technical matter of giving a temporal meaning (an expression that involves a temporal variable) to constant declarations. A *constant module* is a module where all declarations are constants (i.e., it does not declare any temporal variables), and all its definitions are *constant-level*, meaning that they do not make use of the prime operator or any temporal operator. If a module M is *not* a constant module, its constant declarations can only be assigned a constant meaning — not a variable, state function or a temporal formula — and its operator declarations must be assigned a constant-level meaning. However, if a module *is* a constant module, then its constants may be given non-constant meaning, provided that the meanings assigned to its operator declarations are *Leibniz*, or *substitutive*, as I'll explain. Consider the following module:

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

The module is constant and the theorem is valid (i.e., it is a tautology), but if we were to instantiate the module like so:

$$\begin{array}{l}\text{VARIABLES } x, y \\ Prime(e) \triangleq e' \\ \text{INSTANCE } M \text{ WITH } C \leftarrow x, D \leftarrow y, F \leftarrow Prime\end{array}$$

The meaning of the theorem would be $(x = y) \Rightarrow (x' = y')$ which is false, and we cannot allow a theorem to become false when we substitute its free variables.

An operator F is said to be *Leibniz* (*substitutive*) iff $e = f \Rightarrow F(e) = F(f)$ (similarly for operators of higher arity). All builtin TLA$^+$ operators are Leibniz except for prime and the temporal operators, and all constant-level user-defined operators are Leibniz.

A non-constant operator may be Leibniz, too, for example: $G(a) \triangleq x' = [x \text{ EXCEPT } ![a] = y']$

For an operator to be non-Leibniz, one of its parameters must appear in the definition as an argument of a non-Leibniz operator like prime.

The Distributivity of Substitution

We now get to another nuance in TLA$^+$, which is of very minor importance in practice but as we're diving deep into the theory of TLA$^+$, we should explore its darkest corners (but feel free to skip this section). This nuance, however, sheds some light about the mathematical manipulation of programs in general.

We ask whether or not the substitution operation, or "barring", is *distributive* over the various logical operations. Let me explain what I mean with an example. Suppose we have the following module:

$$\begin{array}{l}\text{MODULE } M \\ \text{CONSTANTS } \ldots \\ \text{VARIABLES } \ldots \\ A \triangleq \ldots \\ B \triangleq \ldots \\ C \triangleq A \land B\end{array}$$

which we then instantiate with

$$M \triangleq \text{INSTANCE } M \text{ WITH } \ldots$$

We ask, is $M!A \land M!B$ equivalent to $M!C$? In other words, is $\overline{A} \land \overline{B} \equiv \overline{A \land B}$? If so, we say that substitution, or barring, *distributes* over $\land$.

In ***The Temporal Logic of Actions*** Lamport points out that barring distributes over most TLA operations. For example, $\overline{\Box(F \lor G)} \equiv \Box(\overline{F} \lor \overline{G})$ for any formulas F and G. Indeed, for the normal form of algorithm specification, $\overline{Init \land \Box[Next]_v \land Fairness} \equiv \overline{Init} \land \Box[\overline{Next}]_{\overline{v}} \land \overline{Fairness}$.

However, substitution may not distribute into the fairness conditions that make up $Fairness$. I.e. $\overline{\text{WF}_v(A)}$ may not be equivalent to $\text{WF}_{\overline{v}}(\overline{A})$. The reason is that the weak and strong fairness operators WF and SF are defined in terms of the ENABLED operator. ENABLED is special in that it is a state predicate but is defined using two states (ENABLED A is true in state s iff there exists *some* state t such that the transition $s \rightharpoonup t$ satisfies the action A), and it is not the case that $\overline{\text{ENABLED } \langle A \rangle_v}$ is always equivalent to ENABLED $\langle \overline{A} \rangle_{\overline{v}}$.

For example, if $A \triangleq (x' = x) \land (y' \neq y)$ and $v \triangleq \langle x, y \rangle$, then $\text{ENABLED } \langle A \rangle_v$ is always true (as there always exists a possible next state regardless of the current values of x and y), and so it is equivalent to TRUE. If our substitution is $\overline{x} = z$ and $\overline{y} = z$, then $\overline{\text{ENABLED } \langle A \rangle_v} \equiv \overline{\text{TRUE}} \equiv \text{TRUE}$, but $\text{ENABLED } \langle \overline{A} \rangle_{\overline{v}} \equiv \text{ENABLED } \langle (z' = z) \land (z' \neq z) \rangle_z$ which is always false. This happens because the primed variables are no longer free in the substituted formula, and the substitution that binds them may place constraints on them. So barring does not distribute over ENABLED and the fairness operators defined using it.

This is a technical subtlety but not a problem in practice, as we can compute $\overline{\text{ENABLED } \langle A \rangle_v}$ from $\text{ENABLED } \langle A \rangle_v$ by means other than blindly substituting all variables; it's easy to compute a state predicate that's equivalent to $\text{ENABLED } \langle A \rangle_v$ but doesn't mention primed variables.

Let's place this case in a specification, and analyze the consequences. The example would also demonstrate why we *must* define $\overline{\text{ENABLED } \langle A \rangle_v}$ such that it is still TRUE in the case above, rather than define it to be equal to $\text{ENABLED } \langle \overline{A} \rangle_{\overline{v}}$; in other words, it will help understand why barring must not distribute over ENABLED. Consider the following nested modules (and note that module $Inner$ contains the free temporal variable x as it's defined in an enclosing module prior to $Inner$'s definition):

---- MODULE $Outer$ ----

VARIABLE x

---- MODULE $Inner$ ----

VARIABLE y

$S \triangleq \{0, 1\}$
$Init \triangleq x \in S \land y \in S$
$Next \triangleq x' \in S \land y' \in S \land x' \neq y'$
$Spec \triangleq Init \land \Box[Next]_{\langle x,y \rangle} \land \text{WF}_{\langle x,y \rangle}(Next)$

THEOREM $P \triangleq Spec \Rightarrow \Diamond(x \neq y)$

$I \triangleq \text{INSTANCE } Inner \text{ WITH } y \leftarrow x$

If we look at module $Inner$ in isolation, then obviously P is a theorem. But what about module $Outer$'s $I!Spec$? What behaviors does it specify? If the fairness condition didn't exist, then behaviors where both x and y start up equal to each other and stutter forever would satisfy the specification. But the we do have a fairness condition, so **recall** its definition:

$$\text{WF}_e(A) \triangleq \Diamond\Box\text{ENABLED } \langle A \rangle_e \Rightarrow \Box\Diamond\langle A \rangle_e$$

It means that *if* the action is continuously enabled *then* it must occur infinitely often. One thing is certain: the action $Next$ cannot occur infinitely often — nor can it occur even once — in $I!Spec$. If barring distributed over ENABLED, then $\overline{\text{ENABLED } \langle Next \rangle_{\langle x,y \rangle}}$ would be equivalent to $\text{ENABLED } \langle \overline{Next} \rangle_{\overline{\langle x,y \rangle}}$, which, as we saw in the discussion above, is always false in our "barred" formula, $I!Spec$. But if that were the case, then $\text{WF}_{\langle x,y \rangle}(Next)$ would be true (an implication is true when its left operand is false), which would make $I!Spec$ true even though the $Next$ action can never occur; a stuttering behavior where x and y are constant would satisfy it. This is clearly not what we want, and worse, if that were so, then $I!P$ is not a theorem because a behavior where x and y are always equal cannot eventually reach a state where they're different. We cannot accept a situation where substitution of free variables turns a theorem false.

However, if $\overline{\text{ENABLED } \langle Next \rangle_{\langle x,y \rangle}}$ is defined to be true in this case as in the discussion above (and indeed it is), then the fairness condition $\text{WF}_{\langle x,y \rangle}(Next)$ would become equivalent (by its definition) to $\Box\Diamond\langle Next \rangle_{\langle x,y \rangle}$, which is equivalent to

FALSE in $I!Spec$, as $Next$ cannot occur. And if it's equivalent to FALSE, then so is $I!Spec$. And if $I!Spec$ is FALSE, then $I!P$ is **vacuously true**, because FALSE implies anything. All of $I!Spec$'s behavior eventually reach the impossible state where $x \neq x$ because there are none!

Also note that $Spec$ is machine-closed, yet $I!Spec$ isn't because it is the liveness property that becomes false and makes the entire formula FALSE ! — clearly a very strong safety property (we are not allowed to do anything). Substitution can, therefore, turn a machine-closed specification into a non-machine-closed one, which is yet another reason why we want to allow expressing non-machine-closed specifications.

This annoyance of substitution not being distributive over ENABLED led Lamport to observe that substitution in general does not distribute over common programming constructs (like sequential composition) in his short note, ***Substitution: Syntactic versus Semantic***.

Composition

Engineered systems are often made of components. Describing each component separately, and then the method of their composition is how engineers tackle complexity. Even when a system is made of a single component, it interacts with its environment — the user, the physical world (through sensors and actuators), or other systems — which may be thought of as another component. In this chapter, we'll learn how to model interaction and composition of components. While TLA can elegantly model arbitrary forms of composition, in the next chapter we'll see that its powerful "first-class" representation of abstraction, makes composition an unnecessary complication in practice.

Model-Based Specifications and Information Hiding

How do we specify how a system or a component interacts with its environment or with other component? In other words, how do we specify an *interface*?

Let's consider the following, very abstract, very mathematical specification of a stack. But, first, as a technical mathematical necessity I'll explain later, we'll define an operator that can be used to specify a set by some properties; it selects some "smallest" set that satisfies the property C:

$$Least(C(\_)) \triangleq \text{CHOOSE } T : \land\, C(T) \\ \land\, \forall\, U \in \text{SUBSET } T : C(U) \Rightarrow U = T$$

Now, the stack:

$$
\begin{array}{l}
\text{MODULE } Stack \\
\text{CONSTANTS } X, \quad \text{The set of elements} \\
\qquad\qquad \text{EMPTY}, \quad \text{The ``null'' stack} \\
\qquad\qquad push, pop, top \quad \text{Stack operations} \\
\\
\text{LOCAL } StackT(U) \triangleq \\
\qquad\qquad \land \text{EMPTY} \in U \\
\qquad\qquad \land \forall\, x \in X, s \in U : push[s, x] \in U \\
\qquad\qquad \land \forall\, x \in X, s \in U : push[s, x] \neq \text{EMPTY} \\
Stack \triangleq Least(StackT) \\
\\
\text{AXIOM } A1 \triangleq \land push \in [Stack \times X \to Stack \setminus \{\text{EMPTY}\}] \\
\qquad\qquad\qquad\quad \land pop \quad \in [Stack \setminus \{\text{EMPTY}\} \to Stack] \\
\qquad\qquad\qquad\quad \land top \quad \in [Stack \setminus \{\text{EMPTY}\} \to X] \\
\text{AXIOM } A2 \triangleq \forall\, x \in X, s \in Stack : top[push[s, x]] = x \\
\text{AXIOM } A3 \triangleq \forall\, x \in X, s \in Stack : pop[push[s, x]] = s
\end{array}
$$

What's going on here? We define $Stack$ as the "type" of stacks by defining its algebraic structure: a stack can be EMPTY, all stacks can be constructed by pushing elements, and EMPTY cannot be a result of a push operation.

This description should be very familiar to people who use typed functional programming languages, as it is precisely how algebraic data types are defined. Axiom *A1* (remember that AXIOM is synonymous with ASSUME) specifies the "type" of the stack operations (defined as functions), and axioms *A2* and *A3* specify the relationships among the three stack operations. The specification captures precisely what it means to be a stack without saying anything about *how* one is implemented. It's true that the specification uses functions rather than algorithms in the TLA sense, but I chose that only to make the specification be recognizable to those who are familiar with algebraic specifications in typed functional languages. Axiomatic specification of algorithms in TLA$^+$ could also be done by using temporal formulas in the axioms.

This stack specification is an example of an *algebraic* specification, which, in turn, is a special case of an *axiomatic* specification. Axiomatic specifications specify the external behavior of some system or data structure — its interface — as a list of properties is must satisfy. This approach seems very elegant at first sight, very abstract, very mathematical. But see if you can specify a FIFO queue in that way; or a B-Tree; or a concurrent B-Tree. It is very hard for people who are not logicians to figure out how to specify a system as a list of axioms. For example, the use of $Least$ is necessary to exclude infinite stacks, something which may not be obvious to non-logicians.

Here's what Lamport **says** about axiomatic specifications:

> *The lesson I learned from the specification work of the early '80s is that axiomatic specifications don't work. The idea of specifying a system by writing down all the properties it satisfies seems perfect. We just list what the system must and must not do, and we have a completely abstract specification. It sounds wonderful; it just doesn't work in practice.*
>
> *After writing down a list of properties, it is very hard to figure out what one has written. It is hard to decide what additional properties are and are not implied by the list. As a result, it becomes impossible to tell whether a specification says all that it should about a system. With perseverance, one can write an axiomatic specification of a simple queue and convince oneself that it is correct. The task is hopeless for systems with any significant degree of complexity.*

What works in practice — at least for engineers who are not logicians — is specifying by example. We write a specification of a particular, simple, implementation of a stack, and say that a stack is anything that exposes the same behavior. This approach is called *model-based* specification. In order for it to work, we need to hide the implementation details of the model because other implementations are not required to mimic those; hiding information is a cornerstone of abstraction.

Here is a model-based specification of a stack (this time we will use algorithms):

---- MODULE $InternalStack$ ----

EXTENDS $Sequences$
CONSTANTS X The set of elements
VARIABLES buf, in, out
$vars \triangleq \langle buf, in, out \rangle$

$NOTHING \triangleq$ CHOOSE $x : x \notin X$

$Init \triangleq buf \in \langle\rangle \land in = NOTHING \land out = NOTHING$
$Push \triangleq in' \in X \land buf' = \langle in' \rangle \circ buf \land$ UNCHANGED out
$Pop \triangleq buf \neq \langle\rangle \land out' = Head(buf) \land buf' = Tail(buf) \land$ UNCHANGED in
$Top \triangleq buf \neq \langle\rangle \land out' = Head(buf) \land buf' = Tail(buf) \land$ UNCHANGED $\langle in, buf \rangle$

$IStack \triangleq Init \land \Box[Push \lor Pop \lor Top]_{vars}$

However, we're only interested in how $Push$, Pop and Top interact with the in and out variables. A stack implementation need not necessarily maintain a buf variable containing a sequence of elements. To use our specification as one describing an interface we must hide the internal implementation detail of our model implementation, namely buf. This is done using module instantiation and temporal existential quantification like so:

---- MODULE $Stack$ ----

CONSTANT X The set of elements
VARIABLES in, out

No WITH because buf is bound as a param
LOCAL $Internal(buf) \triangleq$ INSTANCE $InternalStack$

$Stack \triangleq \exists\, buf : Internal(buf)!IStack$

If we have a specification $MyStack$, the statement $MyStack \Rightarrow Stack$ means that $MyStack$ *implements* $Stack$. We will take a close look of what it means for an algorithm to implement another later, but now I'll explain how $\exists$ works to hide the buf variable.

In **part 3** we briefly encountered existential temporal quantification, $\exists\, x : F$ as saying that "there exists a temporal variable x that may take a different value at each state such that F". However, it's easier to think of $\exists\, x : F$ as "F with x hidden". This is similar to regular existential quantification in first-order logic where, for example, $\exists\, y \in Int : y * 2 = x$ uses a hidden variable y to say something about the free variable x. The bound variable y is no longer part of the model at all, which only speaks of the free variable x. This is true for temporal quantification as well.

Note that hiding buf in $Stack \triangleq \exists\, buf : Internal(buf)!IStack$ must be done in a different module from `InternalStack`, one that does *not* declare buf as a free variable in a VARIABLES clause, because TLA+ does not allow re-defining any names.

Open System Specifications

Our specifications describe a system's behavior as it interacts with its environment. If formula M describes how the system behaves, and E describes how the environment does, then the full specification — expressed as a composition — is $E \wedge M$ (remember the equivalence between IO and concurrency we've seen in part 3, and that, in practice, it's simpler to just write a single formula with nondeterminism that expresses the environment; but we're talking theory here). An *open-system* specification describes the contract between the system and the environment — think specification of a library.

The system part, M, of the conjoined specification is insufficient as an open system specification because the correctness of $E \wedge M$ depends on the behavior of E as well; M by itself would assert that the system can behave correctly no matter what the environment does, in other words, it would take E as TRUE , meaning full determinism that allows any sort of interaction with the system. Sometimes, this may be acceptable, as with our stack example above. But sometimes we'd want to specify a library or a system that only guarantees correct behavior if external components — be they other systems or other parts of the program — interact with it in a certain way. We would like to specify just what the system expects or assumes of the environment in order to function correctly rather than the system plus the environment as a whole. Such specifications are also called *rely-guarantee* or *assume-guarantee*, signifying the contract that under so and so assumptions, the system can make such and such guarantees. Such a contract requires that the system behave correctly only if the environment acts as expected.

We could use $E \Rightarrow M$ as the contract[3]. However, the contract $E \Rightarrow M$ is too lenient on the system. Consider a server that receives requests in the form of even numbers, and returns their value divided by two. If the request does not contain an even number, the system makes no guarantees on the outcome. We could specify E and M like so:

$$
\begin{aligned}
&a \div b \triangleq \text{CHOOSE } c \in Int : c * b = a \\
&Evens \triangleq \{n \in Int : n\%2 = 0\}
\end{aligned}
$$

$$
\begin{aligned}
&\text{VARIABLES } in, out \\
&E \triangleq in \in Evens \wedge \Box[in' \in Evens \wedge \text{UNCHANGED } out]_{in} \\
&M \triangleq \Box[in \in Evens \wedge out' = in \div 2 \wedge \text{UNCHANGED } in]_{out}
\end{aligned}
$$

The specification $E \Rightarrow M$ appears to say what we want, but not quite. Suppose that in its first nine steps the environment sends the system even numbers, but in the tenth step the environment sends us an odd number. This would mean that for that behavior, the formula E is false, and so the system may exhibit *any* behavior for which the formula M is false. For example, set y to 33.125 in the *second* step! This is not what we want. We want to say that M would behave correctly *as long as* E does, i.e., up until the step E starts misbehaving.

The special temporal operator $\overset{+}{\rightarrow}$ says that if $F \overset{+}{\rightarrow} G$, then G would not "become false" before F does. Our open system specification would, therefore, be $E \overset{+}{\rightarrow} M$

In practice, however, simple implication (i.e. $E \Rightarrow M$) would suffice, because it is very hard to write a specification for M — let alone a realizable, machine-closed one — that knows in advance that it may misbehave if the environment will misbehave sometime in the future.

Composing Specifications

In part 3 we saw some examples of specifications composed by the conjunction of their formulas. Those examples were of parallel composition, but we can compose specifications in many different ways.

Mathematical theories sometimes involve cumbersome constructions whose intent is just to show they are theoretically possible, and at the same time unnecessary. Turing machines are an example of that. Once their universality has been established with some tedious constructions, that construction need not be repeated, and computer scientists are free to refer to Python programs as Turing machines without writing a Python compiler to TM. We will follow a similar route. I will

show how TLA+ can support compositions that emulate programming language constructs only to later show why they are unnecessary.

Ever since structured programming became popular, subroutines have been the primary form of decomposing programs in all mainstream programming languages and in almost all programming languages in general[4]. We can write specifications as a composition of subroutines by writing some definitions that express the meaning of a subroutine. Those constructs — in particular, the $Subroutine$ operator — are far more complex than anything you're ever likely to encounter in a real TLA+ specification, but their importance lies only in showing that such compositions *can* be expressed.

MODULE $Subroutines$

VARIABLES sub, The active subroutine
$args$, Arguments for a subroutine call
$retval$, The return value from subroutine call
pc The program counter for the main subroutine

$$control \triangleq \langle sub, args, retval, pc \rangle$$

LOCAL $Main \triangleq$ "main"
$ReturnValue(r) \triangleq r' = retval$
$NoCall \triangleq$ UNCHANGED $\langle sub, args, retval \rangle$

$$Label(c) \triangleq sub = Main \land pc = c$$
$$Goto(c) \triangleq Label(c)' \land NoCall$$

$$Call(subname, a, retpc) \triangleq$$
$\land sub' = subname$
$\land pc' = retpc$
$\land args' = a$
$\land$ UNCHANGED $retval$
$\land$ UNCHANGED a

$Spec$ must assign the control vars in WITH if they're used internally

$$Subroutine(Name, Spec(\_), Args(\_,\_), Done(\_), Return(\_)) \triangleq$$
$\Box(sub = Name \Rightarrow$
$\exists\, v : \land Args(args, v) \land Spec(v)$
$\land \Box[\land sub = Name \land Done(v)$
$\land retval' = Return(v)$
$\land sub' = Main$
$\land$ UNCHANGED $\langle pc, args \rangle]_{control})$

We can now use those definitions to encapsulate modules as subroutines and call them:

MODULE $Fast$

EXTENDS $Subroutines, Naturals$

$FactSub \triangleq$ Encapsulate $Fact1$ in a subroutine

LET $Fact1(v) \triangleq$ INSTANCE $Fact1$ WITH $N \leftarrow v.N, i \leftarrow v.i, f \leftarrow v.f$

$Args(args, v) \triangleq v.N = args$

$Done(v) \triangleq v.i \geq v.N$

$Return(v) \triangleq v.f$

IN $Subroutine(\text{“fact”}, Fact1!Fact1, Args, Done, Return)$

Our algorithm

$Init \triangleq Label(\text{“a”}) \land x = 2$

$Next \triangleq \land Label(\text{“a”}) \Rightarrow Call(\text{“fact”}, x, \text{“b”})$

$\land Label(\text{“b”}) \Rightarrow \land ReturnValue(x)$

$\land Goto(\text{“a”})$

To add a subroutine to the spec, we simply conjoin it

$Spec \triangleq Init \land \Box[Next]_x \land FactSub$

It is an easy matter to add support for recursion by simulating a stack; PlusCal does exactly that.

In fact, if a and b are some syntactic constructs — *program terms* — in some deterministic programming language, however they're combined — be it as subroutines, sequentially as in $a; b$, in parallel ($a \parallel b$), as rules in some declarative language — then their composition can be expressed in TLA+ as:

$$\exists\, w : F(C_a, F_a, w) \land F(C_b, F_b, w) \land Composer(w)$$

where F_a and F_b are formulas specifying the two components, w is a tuple of variables used in the composition, F is an operator that defines the composition, and C_a and C_b are constants (one reason constants may be necessary is that $\land$ is commutative, and the composition used — e.g. $a; b$ — may not be).

This ability means that, at least in principle, we should be able to extract from a program, written in some programming language, a TLA formula that specifies it exactly, by direct transformation on the syntax. The TLA formula — or rather, its simple semantics — would then constitute a semantics for the program. This translation from program to a logic formula in some program logic is called the **characteristic formula** approach[5].

However, given the need for the constants C_a and C_b, I am not certain that such a translation would be regarded as *compositional* by programming-language theorists, as the term a may be represented in two different programs as $F(C_{a1}, F_a)$ and $F(C_{a2}, F_a)$, i.e., with two different constants. Without the constants, the target logic would need to have the same mechanisms of composing terms (sequence, parallel, function application, rule etc.) as that of the language, while TLA can be universal. Because of its applicability to both sequential and interactive or concurrent algorithms, and because of its ability to easily express rich abstraction/refinement and equivalence relations, TLA can be a particularly suitable logic for this approach.

I promised a solution to the **“parallel or” problem from part 3**. The composition presented there assumed that the given specifications have no fairness condition, for if they did, $Spec0$ could require that if its next-state action were enabled, then it must eventually be taken, yet our composition would terminate — i.e. would not take any further $Spec0$ steps — if $Spec1$ terminates first. In order to accommodate specifications with fairness conditions, we could still, however, write a composition of the kind described here, that counteracts the fairness conditions of the two specifications:

$$
\begin{array}{l}
ParallelOr(Spec0, state0, Spec1, state1) \triangleq \\
\quad \text{LET } Alternator(s) \triangleq \\
\qquad \land\ s = 0 \\
\qquad \land\ \Box[\ \lor\ s = 0 \land s' = 1 \land state0' \neq state0 \land state1' = state1 \\
\qquad\qquad \lor\ s = 1 \land s' = 0 \land state0' = state0 \land state1' \neq state1]_{\langle s, state0, state1\rangle} \\
\qquad AntiFairness(Spec, x, s) \triangleq Spec \lor \Diamond\Box(s \neq x) \\
\quad \text{IN } \exists\, s : AntiFairness(Spec0, 0, s) \land AntiFairness(Spec1, 1, s) \land Alternator(s)
\end{array}
$$

This, indeed, solves the problem as stated, but perhaps not entirely satisfactorily. A $ParallelOr$ specification would indeed terminate iff any of its two given specs terminate, but if $Spec1$ terminates first, and so the second disjunct of $AntiFairness$ for $Spec0$ is true, then $Spec0$ could be ignored completely, and $state0$ could take any values whatsoever. We may wish to create a stronger composition that truly interleaves the two specifications until one of them terminates. To do that — Stephan Merz showed me this solution — we can use the $\overset{+}{\rightarrow}$ operator we learned in the section **Open System Specifications** to define $AntiFairness$:

$$AntiFairness(Spec, x, s) \triangleq \Box\Diamond(s = x) \overset{+}{\rightarrow} Spec$$

Now $AntiFairness$ states that its operand $Spec$ must hold *as long as* the alternator would eventually execute another step of $Spec$ (by setting s to $Spec$'s index; either 0 or 1 in our case). The above formula can be read as, "as long as s will eventually be equal to x, $Spec$ must hold."

To understand why simple logical conjunction is a universal mechanism of composition[6], like many things in TLA, it is helpful to think of nondeterminism. The specification of a component imposes certain constraints on the behavior of the system. But a component also interacts with its environment — the user, the network, the operating system or other components — on which the component generally imposes no constraints (except, possibly, as requirement for its correct behavior, as we've seen when discussing **open system specifications**). To the component, the environment appears nondeterministic. The composition of multiple components is, therefore, the intersection of the constraints imposed by all components, or, in logic terms, the conjunction of the formulas describing them. A conjunction of formulas — an intersection of their behavior — represents the combined constraints imposed by them, and forms a decrease in the total nondeterminism of the specification, as we specify the behavior of more of its parts. This observation becomes obvious and precise when we learn, in the next chapter, that there is a partial-order relation on TLA specifications defined by their nondeterminism — where "bigger" is more abstract or less deterministic — and, in fact, they form a boolean lattice (almost). The least deterministic ("biggest") specification that still determines both A and B (i.e., "smaller" than both A and B) is their *meet*, $A \land B$. This is just a semantic way of expressing this simple deduction:
$Spec \Rightarrow A, Spec \Rightarrow B \vdash Spec \Rightarrow A \land B$

It should not bother you or offend your aesthetic sensibilities that we need to do things that appear low-level to a programmer, like simulating a stack, as the entire mechanism can be hidden inside a module and thus completely abstracted. In fact, there is never a need to actually write a specification composed so finely. The whole point of this section was to demonstrate that TLA+ can represent the low-level mechanisms of programming languages; what matters now that we know it *can* be done, is demonstrating that it need not be done. As we'll now see, we can equivalently and much more easily specify one component at a time, abstracting all others.

Equivalence, Abstraction and Implementation

Equality and Equivalence

In every mathematical theory that defines some object, an equality relation is of the utmost importance, as you don't really define an object uniquely unless you also say when two descriptions in your theory refer to the same one. The objects TLA is concerned with are discrete dynamical systems, and so we must define when two of them are equal. As we've learned, a TLA in formula specifies a set of behaviors. Two algorithms are equivalent, then, iff their respective sets of behaviors are equal[7]. At the logic (i.e. syntax) level, two formulas denote the same set of behaviors iff they are *logically* equivalent (i.e., one is true iff the other is true), which is denoted by the simple logical connective $\equiv$, called "if-and-only-if" or "equivalent".

TLA's equality is a very strong form of equivalence. We might sometimes care about weaker forms of equivalence. For example, we can give different significance to different parts of the algorithm's state. We can say that an algorithm's behavior is a sequence of states, where each state is a pair of *visible* state and *hidden* state, and that two algorithms are equivalent if the set of *observable* behaviors they denote — when only the observable part of the state is taken into account — are the same. You can think of this as a *projection* of behavior along some dimension that only captures the observable state.

For example, in part 3 **we saw** three specifications of algorithms computing the factorial function, and I mentioned that $Fact1$ and $Fact3$ are equivalent, i.e. they're the same algorithm, only $Fact3$ has more detail (the stack). Let's see how. We'll assume that each specification is defined in its own module, named $Fact1$, $Fact2$, and $Fact3$ respectively. In a separate module, we can then write:

$$\text{VARIABLES } N, f, i$$

$$\begin{aligned} Fact1 &\triangleq \text{INSTANCE } Fact1 \\ Fact3(pc, stack) &\triangleq \text{INSTANCE } Fact3 \end{aligned}$$

Then,

$$\text{THEOREM } Fact1 ! Fact1 \equiv \exists\, pc, stack : Fact3(pc, stack) ! Fact3$$

Meaning that $Fact1$ and $Fact3$ are equivalent if we disregard (hide) $Fact3$'s variables pc and $stack$, considering them the hidden part of the state, and the variables N, f and i the visible part of the state (which we leave free). This is a stronger statement than $Fact3 \Rightarrow Fact1$. In ordinary (non-modal) logic, too, if G is a formula with two free variables, x and y (and so, for clarity, we'll write $G(x, y)$) and F is a formula with just the free variable x (so we'll write $F(x)$), then,

$$F(x) \equiv \exists\, y : G(x, y) \vdash G(x, y) \Rightarrow F(x)$$

but the entailment doesn't work in the other direction (consider the case $F(x) = \text{TRUE}$).

We might care about even weaker forms of equivalence. A common form of weak equivalence, relevant for sequential programs and is central in the theory of pure functional programs is *extensional* equality, which says that two programs are equivalent if they produce (or "reduce to", in PL-theory parlance) equal outputs for equal inputs. It is an equivalence that mimics the notion of equality of functions[8]. If we treat computations as behaviors, as we do in TLA, the functional form of extensional equality compares just the initial (input) and final (output) states by hiding all intermediate states.

First I'll show how this can be done by changing the original specifications a bit; later I'll show how this can be done without any changes.

Here are the two specification, modified by the addition of a new variable, out:

$$\text{MODULE } Fact1$$

$$\text{EXTENDS } Naturals$$

$$\text{VARIABLES } N, i, f, out$$

$$\text{NOTHING} \triangleq \text{CHOOSE } x \notin Nat$$

$$\begin{aligned} Fact1 \triangleq\ & \land N \in Nat \land f = 1 \land i = 2 \land out = \text{NOTHING} \\ & \land \Box[\text{IF } i \leq N \text{ THEN } f' = f * i \land i' = i + 1 \land \text{UNCHANGED } \langle out, N \rangle \\ & \qquad \text{ELSE } \ out' = f \land \text{UNCHANGED } \langle f, i, N \rangle]_{\langle f, i, out, N \rangle} \end{aligned}$$

$$\text{MODULE } Fact2$$

$$\text{EXTENDS } Naturals$$

$$\text{VARIABLES } N, i, f, out$$

$$\text{NOTHING} \triangleq \text{CHOOSE } x \notin Nat$$

$$\begin{aligned} Fact2 \triangleq\ & \land N \in Nat \land f = 1 \land i = N \land out = \text{NOTHING} \\ & \land \Box[\text{IF } i > 1 \text{ THEN } f' = f * i \land i' = i - 1 \land \text{UNCHANGED } \langle out, N \rangle \\ & \qquad \text{ELSE } \ out' = f \land \text{UNCHANGED } \langle f, i, N \rangle]_{\langle f, i, out, N \rangle} \end{aligned}$$

Now we'll establish the extensional equivalence of $Fact1$ and $Fact2$:

$$\text{MODULE } FactAlg$$

$$\text{VARIABLES } N, out$$

$$Fact1(i, f) \triangleq \text{INSTANCE } Fact1$$
$$Fact2(i, f) \triangleq \text{INSTANCE } Fact2$$

$$\text{THEOREM } (\exists\, i, f : Fact1(i, f)!Fact1) \equiv (\exists\, i, f : Fact2(i, f)!Fact2)$$

By hiding variables i and f we're still just projecting the states onto a lower dimension, but because we are left only with N — which never changes — and out which changes only at the last step, we get a projection with stuttering steps, which is equivalent to the following algorithm with a single step:

$$\begin{aligned} & \text{VARIABLES } N, out \\ & FactAlg \triangleq \text{LET } fact[n \in Nat] \triangleq \text{IF } n \leq 1 \text{ THEN } 1 \text{ ELSE } n * fact[n - 1] \\ & \qquad \text{IN } N \in Nat \land \Box[out' = fact[N]]_{\langle N, out \rangle} \end{aligned}$$

We can even extend this kind of extensional view to nonterminating computations. Remember the complicated subroutine construction we've seen? Well, when hiding the control variables with

$$\begin{aligned} & \text{VARIABLE } x \\ & FastWithSub(sub, args, retval, pc) \triangleq \text{INSTANCE } Fast \\ & Fast1 \triangleq \exists\, sub, args, retval, pc : FastWithSub(sub, args, retval, pc) \end{aligned}$$

our algorithm is equivalent to:

$$\begin{array}{l}\text{VARIABLE } x \\ Fast \triangleq \text{LET } fact[n \in Nat] \triangleq \text{IF } n \leq 1 \text{ THEN } 1 \text{ ELSE } n * fact[n-1] \\ \qquad \text{IN } x = 2 \land \Box[x' = fact[x]]_x\end{array}$$

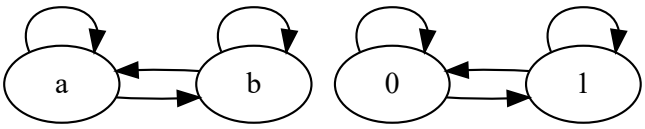
This is very important because it means that we are mathematically justified in replacing subroutines with functions that represent them extensionally if we're not interested in the dynamics of the subroutine. If you're interested in the theory of programming languages, this gives us a very precise relationship between denotational and operational semantics. We can regard the factorial completely denotationally, as a function, or operationally, as an algorithm or a subroutine, and the relationship between the two is exactly an equivalence with hidden state. Later we'll see that this is a special case of abstraction/refinement relation.

We can take equivalence even further. Consider the following two specifications:

$$\begin{array}{l}\text{MODULE } M1 \\ \text{VARIABLE } x \\ Spec1 \triangleq x \in \{\text{“A”}, \text{“B”}\} \land \Box[x' = (\text{IF } x = \text{“A”} \text{ THEN “B” ELSE “A”})]_x\end{array}$$

$$\begin{array}{l}\text{MODULE } M2 \\ \text{EXTENDS } Naturals \\ \text{VARIABLE } x \\ Spec2 \triangleq x \in \{0, 1\} \land \Box[x' = 1 - x]_x\end{array}$$

These two algorithms are clearly not equal, but they are also equivalent in a very clear sense. Here are their respective state diagrams (every state also leads to itself because of stuttering invariance):



The two algorithms only differ in the *values* of their states — also called *labels* in the theory of program analysis — but the structure of their state transitions is identical.

We can express their equivalence like so:

$$\begin{array}{l}\text{VARIABLE } x \\ M1 \triangleq \text{INSTANCE } M1 \\ M2 \triangleq \text{INSTANCE } M2 \text{ WITH } x \leftarrow \text{IF } x = \text{“A”} \text{ THEN } 0 \text{ ELSE } 1 \\ \\ \text{THEOREM } M1!Spec1 \equiv M2!Spec2\end{array}$$

This equivalence is the equality of $Spec1$ and $\overline{Spec2}$, under a mapping of the states. In this case, the mapping is 1-1 and onto, i.e. a bijection, but it doesn't have to be. Let's go back to our factorials and do things a little differently this time. We'll now specify the $FactAlg$ module like this:

```
------------------------ MODULE FactAlg ------------------------
EXTENDS Naturals
VARIABLES N, out
NOTHING ≜ CHOOSE x ∉ Nat
FactAlg ≜ LET fact[n ∈ Nat] ≜ IF n ≤ 1 THEN 1 ELSE n * fact[n − 1]
          IN N ∈ Nat ∧ out = NOTHING ∧ □[out′ = fact[N]]⟨N,out⟩
----------------------------------------------------------------
```

And we'll go back to our original definition of $Fact1$:

```
------------------------ MODULE Fact1 ------------------------
EXTENDS Naturals
VARIABLES N, i, f
vars≜ ⟨f, i, N⟩

Fact1 ≜ ∧ N ∈ Nat ∧ f = 1 ∧ i = 2
        ∧ □[IF i ≤ N THEN f′ = f * i ∧ i′ = i + 1 ∧ UNCHANGED N
                     ELSE UNCHANGED vars]vars

NOTHING ≜ CHOOSE x ∉ Nat
Done ≜ i > N
FA ≜ INSTANCE FactAlg WITH out ← IF Done THEN f ELSE NOTHING

THEOREM Fact1 ⇒ FA!FactAlg
--------------------------------------------------------------
```

We can do something nearly identical with $Fact2$:

```
------------------------ MODULE Fact2 ------------------------
EXTENDS Naturals
VARIABLES N, i, f
vars≜ ⟨f, i, N⟩

Fact2 ≜ ∧ N ∈ Nat ∧ f = 1 ∧ i = N
        ∧ □[IF i > 1 THEN f′ = f * i ∧ i′ = i − 1 ∧ UNCHANGED N
                     ELSE UNCHANGED ⟨f, i, N⟩]vars

NOTHING ≜ CHOOSE x ∉ Nat
Done ≜ i < 1
FA ≜ INSTANCE FactAlg WITH out ← IF Done THEN f ELSE NOTHING

THEOREM Fact2 ⇒ FA!FactAlg
--------------------------------------------------------------
```

Instead of hiding variables in $Fact1$ and $Fact2$, we used a mapping of states, only this time, unlike in our example of the two-state machines, the mapping is *contractive*; multiple states of $Fact1$ (and $Fact2$) are mapped to a single state of $FactAlg$. The ability to replace hiding (i.e., temporal existential quantification) with a substitution, or a state mapping is related to an important theorem we'll get to shortly.

Abstraction and Refinement

We've talked about what it means for algorithms to be equivalent to one another, and we've seen how we can define coarse or fine equivalence relations using hiding or substitution. But it is very common in computer science to speak of an algorithm *implementing* some interface, or of a type or an interface being an *abstraction*. TLA allows us to speak precisely about those concepts, and so we will give precise meaning to the familiar intuitive notions of implementation and abstraction.

There are many ways we think of abstraction. We can think of it as a distillation of the important features of some system, or we can think of it as identifying commonalities in a set of things and factoring them out, or we can think of them as a screen hiding irrelevant detail. There are probably other ways people think of when they talk of abstraction, but mathematically we can describe them all — as they all mean the same thing — as a superset. There are red apples, red cars and other red things, and the abstract notion of redness can be represented as a set that includes them all. Of course, nothing stops us from having another set of apples of all colors, representing the essence of "appleness", or a set of all small things etc. An abstraction abstracts many things — called *instances* of the abstraction — and a thing can be abstracted by many abstractions. There are even sub-abstractions — red apples — that have their own instances (red apples of all sizes) and their own sub-abstractions (small red apples). A sub-abstraction, as well as an instance, is always more specific, provides more details, than its abstraction.

When we talk of algorithms and programs, we call an instance of an abstraction an *implementation*. We will also use another name that is common in program analysis: *refinement*. As we've seen in part 3, TLA makes no distinction between an algorithm and an algorithm property (i.e. a set of algorithms), so we will use the terms implementation and refinement interchangeably.

An algorithm, algorithm property, or any discrete system A, is an *implementation* or a *refinement* of a system B iff the set of behaviors of A is a subset of that of B. In which case we say that B is an abstraction of A; sometimes people also call B a *specification* of A, but I will not use that word as it may be confusing.

The abstraction/refinement relation in TLA defines a partial order on the universe of behaviors. Indeed that universe forms a bounded **lattice**, with refinement being set inclusion. Syntactically, the bottom element is FALSE, top is TRUE, and the join and meet are just $\vee$ and $\wedge$ respectively. In short, it forms a **boolean algebra**[9].

The refinement relation, A implements B (or A's set of behaviors is a subset of B's) is represented by simple logical implication, $\Rightarrow$. Even if little in the previous paragraph made sense to you, you can appreciate that $\Rightarrow$ is a "less than" order relation in boolean algebra: look at **its truth table** with 0 as false and 1 as true, and see that it is the same as $\leq$.

For example, a web service that emits values of type `int` can be represented as $x \in Int \wedge \Box[x' \in Int]_x$ (or just $\Box(x \in Int)$), which then forms an abstraction for a service that emits *incrementing* integer values. Indeed:

$$(x = 0 \wedge \Box[x' = x + 1]_x) \Rightarrow (x \in Int \wedge \Box[x' \in Int]_x)$$

And, as we've seen in part 3, a clock that displays both the hour and the minute is a refinement of a clock that just displays the hour.

As we've done with equivalence, we can say that an algorithm's behavior has a visible part and a hidden part, and that algorithm A is a refinement of B if the visible behaviors of A form a subset of those of B. Hiding detail — with the temporal existential quantifier — is quite a general form of abstraction.

If the algorithms/systems are represented by formulas F and G, in TLA the abstraction/refinement relation becomes:

$$(\exists\, w : F) \Rightarrow (\exists\, v : G)$$

where w is a tuple of those variables in F we wish to hide, and v is a tuple of the variables of G we want to hide. As with any existential quantification in a logic that permits free variables, the statement above is logically equivalent to:

$$F \Rightarrow (\exists\, v : G)$$

In general $(\exists\, v : \Psi) \Rightarrow \Phi$ is eqivalent to $\Psi \Rightarrow \Phi$, provided that the variables v do not appear free in Φ. Intuitively, if the existence alone of appropriate values is enough to imply the right-hand side then so does every concrete satisfying assignment; e.g., the proposition $(\exists\, x : x > 0 \land y \geq x) \Rightarrow y > 0$ is equivalent to the proposition $x > 0 \land y \geq x \Rightarrow y \geq 0$. Using the same ordinary logic analogy, it is easy to see that the converse does not hold, i.e., $\Psi \Rightarrow (\exists\, w : \Phi)$ is *not* equivalent to $\Psi \Rightarrow \Phi$.

To give the above formula a more concrete meaning, recall our discussion of model-based specification. We state that a specification $MyStack$ implements a stack, as specified by our model, with the proposition:

$$MyStack \Rightarrow (\exists\, buf : IStack)$$

This is obviously something we are interested in verifying, but how do we do that? Unfortunately, the TLC model checker cannot check formulas containing temporal quantification. The problem is, in general, co-NP-hard *in the number of states*, which is big to begin with, although an algorithm that is efficient in practice may be found. Proving it directly is also hard.

But recall that in the **previous section** we saw how an equivalence with hidden variables (extensional equivalence of factorial algorithms) could also be expressed using substitution. The model checker *can* check formulas of the form $F \Rightarrow \overline{G}$, and as both F and G are presumably written as state machines in normal form and specify successor state step-by-step, *proving* such a mapping is not as hard as proving implication that uses states hidden with a temporal quantifier. In 1988, Martín Abadi and Leslie Lamport published an important result about that very idea in their paper, ***The Existence of Refinement Mappings*** (1991).

A function from F's states to G's states is called a *refinement mapping*. Unfortunately, even if F implements G in the sense above — namely that the visible portion of F's behaviors is a subset of G's — such a mapping does not always exist. Consider some examples:

- Algorithm B computes a running average of input numbers that flow into the system one by one. B does so by maintaining a (hidden, internal) growing sequence of all inputs and emitting their average. But an algorithm A exhibiting the same visible behavior — emitting a running average — is not required to maintain such a list, and, instead, can make do with (internally) keeping a running sum and count. There is no mapping from A's state of a sum and a count to B's state of a list of all previously observed numbers, as the latter requires more information.
- Algorithm B emits a sequence of ten nondeterministically chosen numbers. B does so by (internally) nondeterministically choosing a sequence of ten numbers in the initial state, and then emitting them one by one. Algorithm A could exhibit the same visible behavior by nondeterministically choosing a number at each state. There is no mapping from the state of A, that only has the currently emitted value, to that of B, containing all future outputs.
- Algorithm B specifies an hour clock, but internally keeps track of minutes, too, while algorithm A only emits the time in hours. Both have the same visible external behavior, but B "runs slower". There is no function mapping A's state — which changes only once an hour — to the multiple states representing that same hour in B.

Abadi and Lamport proved that if A is machine-closed, B has finite invisible nondeterminism (both discussed in part 3), plus another condition on B that we didn't cover, then there exists a refinement mapping from A to B, or from a modified version of A that contains auxiliary variables — either *history* or *prophecy* variables — to B, where a history variable records the history of the state (or a part of the state) and a prophecy variable nondeterministically chooses the *future* states. In other words, the theorem is that the addition of history and/or prophecy variables is enough to guarantee a refinement mapping for any abstraction/refinement.

Consider the following algorithm that emits ten nondeterministically chosen numbers:

$$
\begin{aligned}
&\text{VARIABLES } n, x \\
&vars \triangleq \langle n, x \rangle \\
&Init \triangleq \land n = 9 \\
&\qquad\quad \land x \in Int \\
&Next \triangleq \land n > 0 \\
&\qquad\quad \land n' = n - 1 \\
&\qquad\quad \land x' \in Int \\
&Spec \triangleq Init \land \Box[Next]_{vars}
\end{aligned}
$$

Here is the algorithm with an added history variable, h:

$$
\begin{aligned}
&\text{VARIABLES } n, x \\
&\text{VARIABLE } h \\
&vars \triangleq \langle n, x \rangle \\
&Init \triangleq \land n = 9 \\
&\qquad\quad \land x \in Int \\
&\qquad\quad \land h = \langle x \rangle \\
&Next \triangleq \land n > 0 \\
&\qquad\quad \land n' = n - 1 \\
&\qquad\quad \land x' \in Int \\
&\qquad\quad \land h' = Append(h, x) \\
&Spec \triangleq Init \land \Box[Next]_{\langle vars, h \rangle}
\end{aligned}
$$

And here it is with an added prophecy variable, p:

$$
\begin{aligned}
&\text{VARIABLES } n, x \\
&\text{VARIABLE } p \\
&vars \triangleq \langle n, x \rangle \\
&Init \triangleq \land n = 9 \\
&\qquad\quad \land x \in Int \\
&\qquad\quad \land p \in [1..n \to Int] \\
&Next \triangleq \land n > 0 \\
&\qquad\quad \land n' = n - 1 \\
&\qquad\quad \land x' = Head(p) \\
&\qquad\quad \land p' = Tail(p) \\
&Spec \triangleq Init \land \Box[Next]_{\langle vars, p \rangle}
\end{aligned}
$$

But how do we prove that adding the auxiliary variables does not restrict F's behaviors, and that our refinement mapping is *sound*? We need to prove that $F \Rightarrow \exists\, aux : F\_aux$ which brings us back to the original problem.

The situation with history variables is simple because if F_h is F with a history variable added, and we define

$$H \triangleq h = vars \land \Box[h' = Append(h, vars)]_{\langle vars, h \rangle}$$

(where $vars$ is the tuple of variables in F) then:

$$
\begin{aligned}
F\_h \equiv F \land H \Rightarrow &\exists\, h : (F \land H) \\
&\equiv F \land \exists\, h : H \\
&\equiv F \land \text{TRUE} \quad \text{Because the action in } H \text{ is always enabled} \\
&\equiv F
\end{aligned}
$$

But unfortunately, this simple decomposition cannot be done for prophecy variables. Lamport and Stephan Merz recently published ***Auxiliary Variables in TLA+*** which gives syntactic rules for using prophecy variables that are proven to be sound

(i.e., don't restrict behaviors).

Refinement mapping is a very powerful verification technique not only in theory, but also in practice. It is sometimes beneficial to specify an algorithm at several levels of detail, say, A, B and C, with A being the most detailed (refined) and C the most abstract, and then verify that for some property P, $C \Rightarrow P$. Because $A \Rightarrow B \Rightarrow C$, we learn that $A \Rightarrow P$, and A is specified at a level that is close enough to code to be readily implemented.

The Meaning of Abstraction

The equivalences and abstraction/refinement relations we've seen mean that there is rarely a need to compose specifications. A component can more easily be specified in isolation, with simple abstractions standing in for its cooperator components.

The definition of a refinement/abstraction relation in TLA$^+$, $B \Rightarrow A$, is that the behaviors of B are all behaviors of A. One way to look at it is that algorithm B implements abstraction A if all of B's behaviors are allowed by A or, conversely, that B never exhibits a behavior that is forbidden by A. Looking at it from another perspective, abstraction — like input and concurrency — is just nondeterminism. A is a nondeterministic descriptions of all algorithms that implement it.

An abstraction, then, is a description with less detail, or more nondeterminism, but one that fully delineates the possible behaviors of implementations. It's a conservative approximation, one that helps reasoning by being on the one hand simpler than its concrete instances, and on the other, sound with respect to them, meaning a property that's true for the abstraction must be true for all implementations.

It is therefore clear that we do not want to limit the how far our abstractions stray from physical realizability, as the more abstract they become, the easier it is to prove things about them. On the other hand, we want flexibility to lower our abstractions arbitrarily, to make sure that what we want to verify is still true for them. We want our abstractions just concrete enough for the properties we're interested in verifying are true, but not lower. The ability to ignore detail flexibly and selectively — which is at the very core of TLA$^+$'s design — is of the utmost important in formal methods. The cost of verifying the correctness of a program — be it with automatic tools or manually — grows very rapidly with the addition of detail.

In the next section I'll mention an even more general description of abstraction than TLA's, employing something called Galois connections. It is perhaps the most general concept, one which does not rely on any specific description of computation and encompasses all other particular notions of abstraction, but the general principle holds: abstraction is nondeterminism.

Once you understand this, it becomes easy to see that the very application of a (sound) mathematical description of reality is an abstraction — in this very sense — of reality, or that reality is a refinement of the description. The question of how closely should the mathematical framework match reality becomes merely one of asking at which level of abstraction do we wish to think.

Other Notions of Equivalence and Order

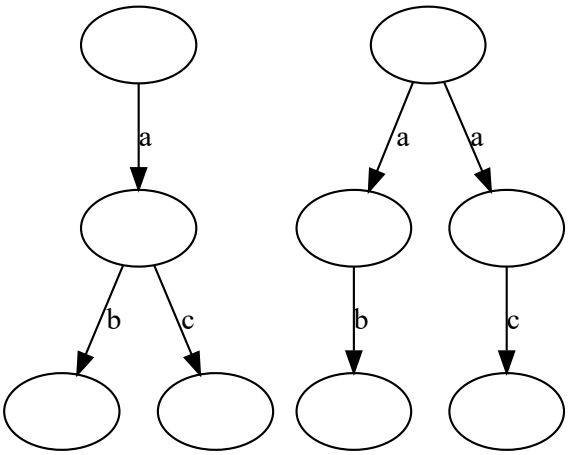
There are other notions of equivalence and order (abstraction/refinement) on programs and algorithms in academic literature. I will mention a few of them, and show their relationship to TLA.

Process Algebras

Around the same time Lamport and others were working on specifying systems using temporal logic, others were working on an algebraic, programming-language-like approaches called process calculi or process algebras, most notable among them were Robin Milner and Tony Hoare.

In a programming language one usually doesn't mention the program's state, only the actions it performs. If we want to reason about algorithms in a pseudo-programming language like a process algebra, we also need notions of equivalence

and order, but things may get complicated when we don't wish to mention state. Comparing traces of events alone (I will not call them actions to not confuse with TLA actions) for equality or inclusion yields an imprecise equivalence. For example, the following two state machines share the same set of action traces, $\{a \to b, a \to c\}$, and yet are clearly very different:

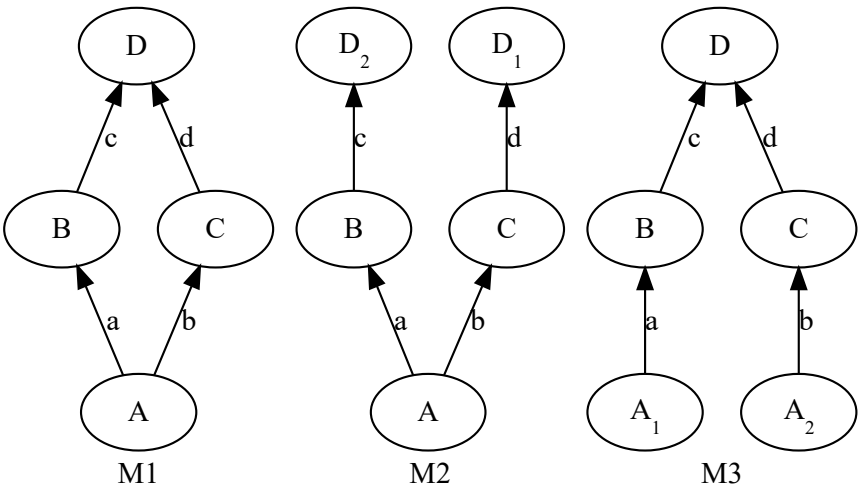


The order relation suggested by Milner to represent implementation is called **simulation**. Given two event-labeled state machines, P and Q, a *simulation* over $\langle P, Q \rangle$ is a relation $R \subseteq P \times Q$, such that for every $\langle p, q \rangle \in R$, if p is an initial state of P then q is an initial state of Q, and if $p \xrightarrow{a} p'$ then $q \xrightarrow{a} q'$ and $\langle p', q' \rangle \in R$. If such a relation exists, we say that Q simulates P, or that P is *similar* to Q. Intuitively, we can think of simulation as a game: Q simulates P if for every move taken by P, Q can match it with the same move.

Simulation can be used to define an equivalence called bisimulation: if both R and R^{-1} (the relation with its pairs reversed) are simulations, then R is said to be a *bisimulation*. If there exists a bisimulation between systems P and Q, they are said to be *bisimilar*. Using our game analogy, P and Q are bisimilar if at every turn, no matter which goes first, the other can match the move. Note that if P simulates Q and Q simulates P, then they are not necessarily bisimilar, as bisimilarity requires that the same relation and its inverse are simulations. This is why simulation is a ***preorder*** rather than a partial order, as it is not antisymmetric — if two processes simulate one another, they are not necessarily equivalent.

Milner's simulation relation is sometimes also called *forward simulation*, and a dual notion can be defined as follows: a relation $R \subseteq P \times Q$ is a *backward simulation* if for every $\langle p', q' \rangle \in R$, if $p \xrightarrow{a} p'$ then there exists a q such that $q \xrightarrow{a} q'$ and $\langle p, q \rangle \in R$.

Consider the following three state machines:



M1 simulates both M2 and M3. M2 forward-simulates M1 (in fact, they are bisimilar) but does not backward-simulate it, and M3 does not forward-simulate M1 but it does backward-simulates it.

How are these relations related to those of TLA? $M2 \Rightarrow \overline{M1}$ and $M3 \Rightarrow \overline{M1}$ under the refinement mappings that map states D_1 and D_2 to D and A_1 and A_2 to A, respectively. But what about the other direction? Note that M2 is exactly M1 with a history variable added (a single state in M1 corresponds to multiple states in M2 iff it can have different histories), so $M1_h \Rightarrow M2$ (in fact, they are equivalent), and M3 is exactly M1 with a prophecy variable added (a single state in M1 corresponds to multiple states in M3 iff it can have different futures), so $M1_p \Rightarrow M3$. Forward simulation, then, corresponds to the addition of a history variable (if necessary), and backward simulation corresponds to the addition of a prophecy variable.

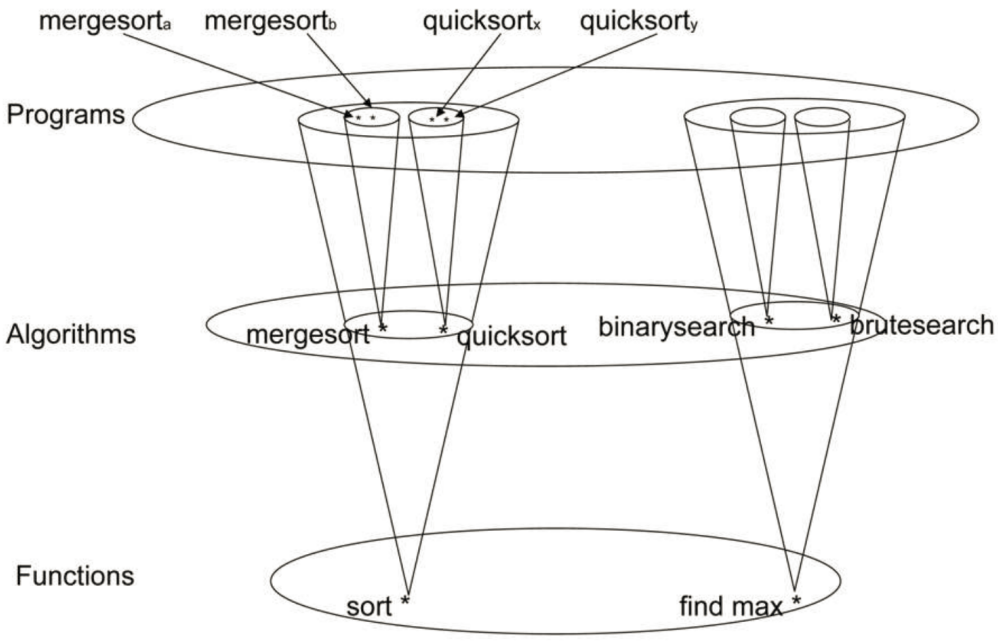
TLA manages to be simpler than simulation relations — as those require some structural analysis of the processes rather than just defining relation on their traces — because it mentions state explicitly.

Categorical Algorithms and Refinement Calculus

Another approach for defining equivalence relations over algorithms can be found in the 2010 paper, ***Towards a Definition of an Algorithm***, by Noson Yanofsky. The paper is quite readable, but I'll quickly go over the salient details. Yanofsky defines an algorithm to be an equivalence class on programs. In the paper, he uses a pseudo-programming language that is made of a tree whose nodes are the primitive operations in the definition of **primitive recursive functions**, and calls those trees *programs*. Those programs compute (primitive-recursive) functions. He then defines a **multigraph** of objects that are similar to states, except they don't represent intermediate stages of computation, but inputs and outputs. The programs are the edges of the graph, connecting inputs to outputs.

He then notes that the multigraph is transitive: if there is a program f connecting node a to node b, and program g from node b to node c, then there must be a program $f \circ g$ from a to c. However, this multigraph is not a **category** because it is not associative: the two programs $h \circ (g \circ f)$ and $(h \circ g) \circ f$ are represented by edges connecting the same node, but as the two programs (i.e., trees) are not the same, the two edges are distinct. He then goes on to define a series of equivalence relations on programs (trees) which he says preserve their essence, and shows that if we take a quotient of the program graph with respect to the equivalence relations, we get a category of algorithms.

Yanofsky's approach only covers equivalence, not order, but the three concentric equivalence relations he defines — on programs, algorithms and functions — essentially create an order consisting of exactly three tiers. Programs refine algorithms, and algorithms refine functions, but there is no general order relation that puts all three levels on some continuous spectrum, as seen in this image from the paper:



The reason Yanofsky needs to define programs structurally (or intensionally), and only then algorithms as a quotient of programs with respect to structural equivalence, rather than extensionally, by how they behave, is due to the original sin of

those approaches that view programs (or algorithms) as denoting functions rather than discrete dynamical processes[10].

A similarly structural approach, but one that does create an order relation on programs, is Ralph-Johan Back's refinement calculus and Joakim von Wright's refinement algebra. The idea behind **refinement calculus** is defining some transformations that make a program more refined or abstract. **Refinement algebra** is surprisingly similar to the ideas we've seen in TLA, only expressed as an algebra rather than a logic, and it does not extend to concurrent algorithms. Back and von Wright wrote **a book** together about their approach that is freely available online. Refinement algebra/calculus is similar in some ways to TLA (sometimes very similar), but is less universal and more complicated; on the other hand, it can — at least in principle — be applied directly to programs written in a programming language. This is another example of how much expressive power a simple formalism like TLA can gain by giving up on attempting to directly talk about programming languages.

Types

Typed programming languages have not one but two notions of implementation order: subtypes and type inhabitants (instances). A type is an abstraction of both its subtypes as well as its concrete instances; conversely a subtype is a refinement of its supertypes, and a value is a refinement of its type. While we can create any hierarchy of abstraction relations using subtypes, we cannot directly show that one *program* — a type inhabitant — implements another, even in formalisms with dependent types; a program is a leaf in the order relation[11]. If our types are rich enough, we can simulate such a relation by writing our algorithm at the type level, only then it is no longer a program but a type. Equivalence in those formalisms is also complicated.

TLA makes no such distinctions between types (properties) and their inhabitants (algorithms). Both are just formulas — or, semantically, sets of behaviors — and are part of the same abstraction/refinement partial lattice, that of implication (or set inclusion). There are just different levels of detail.

The semantics of TLA is too simple to allow making any useful distinction between ***denotation***[12] and ***operation*** — an interesting distinction in the theory of programming languages — but as we've seen, it is a simple matter to hide any detail we deem irrelevant for the sake of a particular abstraction/refinement relation and choose the precise level of detail for the sake of comparisons, one of which may correspond exactly to the level of detail provided by your programming language of choice, be it SQL, Java, assembly, or digital circuits.

Of course, types are normally used for the more difficult task of specifying an actual runnable program, and they can potentially be used for some advanced techniques such as proof-carrying code. The distinction between types and their inhabitant programs is necessitated by the need to ensure programs are sufficiently detailed to be compilable to efficient executables. Types also have benefits beyond specification (for which, I believe, other approaches such as contracts are more appropriate) such as assisting with tooling, maintenance and code organization.

Galois Connections

All specific definitions of abstraction/implementations we've seen — TLA refinements, process simulations, subtyping and type inhabitants — are encompassed by the very general, very abstract concept of a **Galois connection**. A clear introduction to abstraction relations as Galois connections can be found in ***Binary Relations for Abstraction and Refinement*** by David A. Schmidt. The discussion in the paper (except for the last part, which specifically deals with temporal logic) is completely detached from any particular formalism, and helps to think of programs abstractly and mathematically, detached from their expression in any particular formalism. Another introduction to Galois connections and their power, although less in the context of algorithms, can be found in Peter Smith's, ***The Galois Connection between Syntax and Semantics***. I will not attempt to discuss Galois connections in any depth, just quickly define them and show their relevance to the subject of abstraction and refinement in the context of TLA.

If $\mathcal{P} = \langle P, \preceq \rangle$ and $\mathcal{Q} = \langle Q, \sqsubseteq \rangle$ are posets (partially ordered sets), and $f_* : P \to Q$, $f^* : Q \to P$, are functions, then the pair $\langle f_*, f^* \rangle$ is a *Galois connection* if for all $p \in P, q \in Q$: $f_*(p) \sqsubseteq q$ iff $p \preceq f^*(q)$. f_* is then called the lower, or left, adjoint (of the connection), and f^* the upper, or right, adjoint. f_* and f^* uniquely determine each other; setting one forces the other.

In the context of TLA, if we have a refinement mapping such that $F \Rightarrow \overline{G}$, we can call that mapping, when seen as a function on complete behaviors, f. We then we take P and Q to be the *powersets*[13] of the behaviors of F and G respectively, and define f_* as a function on sets, such that it applies f element-wise to every behavior in the a set. f_* is then a function from P to Q, mapping the elements of P — subsets of the behaviors of F — to elements of Q — subsets of G's behaviors. We define f^* to be the preimage of f. We get a Galois connection if we take the partial orders on both posets as just set inclusion.

Galois connections serve as the basis for **abstract interpretation**, a central concept in the theory of program analysis, developed by Patrick and Radhia Cousot in the 1970s. In TLA terms, abstract interpretation is a process by which the more abstract specification, G, is automatically generated from the more concrete one, F. Abstract interpretation places different kinds of programs semantics on a spectrum of abstraction, from behaviors to functional denotational semantics. Many ideas in program analysis, including type checking, model checking and deductive proofs can all be viewed as forms of abstract interpretation. In TLA+ this idea is made explicit and formally manipulable, though less general.

TLA manages to express equivalence and order extensionally — based on what the program does, not how it is constructed — rather than structurally, like the process calculi, refinement calculus and Yanofsky's categories. And yet, unlike types, TLA is able to express arbitrary level of detail because of two features: it treats algorithms as denoting behaviors rather than functions, and it explicitly models state. The structural refinement calculus/algebra is similar in its expressiveness as far as sequential algorithms are concerned.

Parameterization vs. Nondeterminism

In part 1 I mentioned a claim made that TLA+ is not higher-order, suggesting it has cannot easily describe higher-order algorithms. While I think that by this point it should be clear how universal TLA+ is, the point is still worth revisiting, if only to discuss what Lamport calls the "Whorfian syndrome" — the confusion of language with reality.

Whether a program is "higher-order" or not is not a property of what it does, but how it is described; in other words, higher-orderness is a property of a formalism, not of reality. Just as continuous dynamical systems that are themselves modified by other dynamical systems are usually not expressed as higher-order ODEs but as first-order ODEs of a higher state dimension, so too whether or not algorithms that are modified by other algorithms are expressed as higher-order constructs depends on the formalism, and in TLA, every composition of algorithms can be expressed as a conjunction of formulas (and adding variables, possibly hidden, to describe the state of each component; i.e., increasing the dimension of the state).

Let's consider higher-order functions, in the functional programming sense. A higher-order function in is a program (subroutine) that is parameterized by another program. We've seen how that could be expressed using a conjunction of formulas, but before we get back to that we'll look at abstracting the "higher-order" parameter as a function — which is justified by an abstraction relation.

Here is one formulation of the flatmap operation familiar to functional programmers. Normally wouldn't specify such a simple operation as an algorithm but rather abstract it as a function or operator, but this is just for the sake of discussion:

$$\begin{array}{l}\text{CONSTANTS } S, T \\ \text{VARIABLES } x, y\end{array}$$

$$Flatmap(f, s) \triangleq x = s \land y = \langle\rangle \land \Box[x \neq \langle\rangle \land y' = y \circ f[Head(x)] \land x' = Tail(x)]_{\langle x,y\rangle}$$

(we could have made f a parameter of the module itself, with $\text{CONSTANT } f$ and $\text{ASSUME } f \in [S \to Seq(T)]$).

The following theorem expresses a very simple type property:

$$\text{THEOREM } \forall\, s \in Seq(S), f \in [S \to Seq(T)] : Flatmap(s, t) \Rightarrow \Box[y \in Seq(T)]$$

It is expressed analogously to how it would be expressed in a typed programming language: with a universal quantifier over the inhabitants of the input types. However, we could express the algorithm and theorem *equivalently* as follows:

$$\begin{array}{l}\text{CONSTANTS } S, T \\ \text{VARIABLES } x, y, f \\ \\ Flatmap \triangleq \land f \in [S \to Seq(T)] \land x \in Seq(S) \land y = \langle\rangle \\ \qquad\qquad \land \Box[x \neq \langle\rangle \land y' = y \circ f[Head(x)] \land x' = Tail(x) \land \text{UNCHANGED } f]_{\langle x,y,f\rangle} \\ \\ \text{THEOREM } Flatmap \Rightarrow \Box[y \in Seq(T)]\end{array}$$

You can think of the former formulation as a description of a single, parameterized, computation, and of the latter as a description of all possible computations or as a nondeterministic algorithm, but the theorem is equivalent in both formulations. Nondeterminism is just a different expression of parameterization.

From a formal logic perspective, the difference is one between a formalism that requires all variables to be bound (by quantifiers or lambda expressions) and one that allows free variables. For example, in the latter, instead of writing $\forall\, x \in Int : x > 0 \Rightarrow -x < 0$ (all variables are bound) you can write $x \in Int \land x > 0 \Rightarrow -x < 0$, and instead of $\forall\, y \in Real : (\exists\, x \in Real : x > 0 \land y \geq x) \Rightarrow y > 0$, you can write $x \in Real \land y \in Real \land x > 0 \land y \geq x \Rightarrow y > 0$.

Programming languages are formalisms that generally require all variables to be bound. This requires higher-order constructs where a logic allowing free variables doesn't. Free variables are just how nondeterminism is expressed syntactically.

But you may object and say that my trick of abstracting the subroutine parameter as a function — while justified as a valid abstraction — is sidestepping the issue. In TLA+, algorithms (such as subroutines) are temporal formulas, whereas functions are simple values no different from integers. While abstraction allows us to evade the issue of "higher-order" algorithms, is the theory able to tackle them directly?

The answer is yes. Suppose we have the specification $Spec(G) \triangleq F \land G$, which is parameterized by the algorithm specification G (e.g. G can be a subroutine like we saw above, or a concurrent process that interacts with F). If in a typed language we wanted to prove that $Spec(G : Q) : P$, or that for G of type Q, $Spec$ is of type P, then in a logic like TLA we would think to express the same proposition like so:

$$\text{THEOREM } \forall\, G : (G \Rightarrow Q) \Rightarrow (Spec(G) \Rightarrow P)$$

However, the above is not a TLA formula: the data logic of TLA+ allows us to quantify over constants (i.e. non-temporal variables), and TLA additionally allows us to quantify over temporal variables, but G is neither — it is a temporal formula. The model of a TLA formula is a set of behaviors, but an algorithm (or a property) is not a TLA object and we cannot quantify over it (nor have a free variable that denotes a set of behaviors). The formula above is a second-order proposition, as it universally quantifies over G, an algorithm (a set of behaviors).

However, as we've seen, while all TLA formulas are first-order, the TLA+ proof language does allow us to write second-order propositions that are not themselves formulas. We can therefore write the proposition like so:

$$\begin{array}{l}\text{THEOREM ASSUME TEMPORAL } G, G \Rightarrow Q \\ \text{PROVE} \quad Spec(G) \Rightarrow P\end{array}$$

But this is both overkill and cannot be checked by the TLC model checker. More importantly, the proposition is a very mundane, reasonable one about algorithm correctness, and it stands to reason that a good formalism would allow us to directly specify it as a formula.

As it turns out, this is easy. Because tautologically $\forall\, A : (A \Rightarrow B) \Rightarrow (A \Rightarrow C)$ is equivalent to $B \Rightarrow C$[14], the above is equivalent to the much simpler, first-order proposition:

$$F \land Q \Rightarrow P$$

which we can write as a TLA formula. Moreover, as F is presumably written in normal form and Q is likely a simple safety property, i.e., an invariant, $F \land Q$ could be easily rewritten as a single, normal form formula, which would allow us to check that it satisfies P using the model checker.

We are saved, again, by nondeterminism. TLA does not distinguish between algorithms and properties of algorithms (or types) — they're both just formulas, or semantically, sets of behaviors. We don't need to quantify over all algorithms G that satisfy the property Q; we just write Q.

Various Cool Stuff

Before concluding this treatment of the theory and design of TLA+, I want showcase some uses of TLA+ that you're unlikely to encounter when modeling "ordinary" software but are of great importance for special classes of software, as they demonstrates the power of the formalism. Rather than a thorough treatment, I will just give a taste of what's possible.

Real Time

TLA+ can be used to model all kinds of systems, among them realtime systems. The following example is taken from *Specifying Systems*.

Let's revisit our specification of the hour clock from part 3:

$$\begin{array}{l} \text{VARIABLE } h \\ Next \triangleq \text{IF } h \neq 12 \text{ THEN } h + 1 \text{ ELSE } 1 \\ HourClock \triangleq h \in 1..12 \land \Box[Next]_h \land \text{WF}_h(Next) \end{array}$$

While this describes the behavior of an hour clock at the level of detail that may be of interest to some, it says nothing about the actual time duration between ticks, which may be of great interest to others. To model real time, we do what scientists normally do and introduce a real-valued variable, t to represent it. The variable t (in second units) will change between ticks of the clock, but not during the tick — we'll assume that the tick is instantaneous (this is just a modeling choice). If we wanted to implement our clock in a realtime system, we would want to specify that it ticks approximately every hour, allowing for some margin of error, which we'll call k. We'll introduce a variable, $elapsed$ to represent the elapsed time between now and the last tick:

$$\begin{array}{lll} \text{CONSTANT } k & & \\ \text{VARIABLES } h, & & \text{The hour} \\ \qquad\qquad\quad t, & & \text{Time, in second units} \\ \qquad\qquad\quad elapsed & & \text{How much time has elapsed since the last tick} \end{array}$$

$$\begin{array}{l} HCTInit \triangleq h \in 1..12 \land elapsed = 0 \\ HCTNext \triangleq \lor \land h' = \text{IF } h \neq 12 \text{ THEN } h + 1 \text{ ELSE } 1 \\ \qquad\qquad\quad \land elapsed' = 0 \\ \qquad\qquad \lor \land \text{UNCHANGED } h \\ \qquad\qquad\quad \land elapsed' = elapsed + (t' - t) \end{array}$$

We can model the requirement that no more than $3600 + k$ elapse between ticks like so:

$$MaxTime \triangleq \Box(elapsed \leq 3600 + k)$$

We also want to specify that at least $3600 - k$ seconds elapse between consecutive ticks:

$$MinTime \triangleq \Box[Next \Rightarrow (t \geq 3600 - k)]_h$$

(remember, an action can only follow $\Box$ within $[]_e$ to ensure invariance under stuttering; in this case e is just h because we don't care about actions that don't change the value of h).

We can now write the specification for our realtime hour clock:

$$HCTime \triangleq HCTInit \land \Box[HCTNext]_{\langle h,t,elapsed \rangle} \land MaxTime \land MinTime$$

Finally, we need to specify how t can change. Time can move forward by any amount as long as h doesn't change (the tick is instantaneous):

$$\begin{aligned} TNext \triangleq & \land t' \in \{r \in Real : r > t\} \\ & \land \text{UNCHANGED } h \end{aligned}$$

The safety part of the time specification would then be $(t \in Real) \land \Box[TNext]_t$, but what about liveness? Requiring that $TNext$ would occur infinitely often, thus ensuring t increases indefinitely, is not enough. Because t is a real-valued variable, even though $TNext$ requires that h is unchanged and therefore t cannot grow by more than $3600 + k$ at a time, $TNext$ can still occur infinitely often, and t grow indefinitely while never reaching $3600 + k$ and the clock ticking even once. This is what's known as a *Zeno behavior*, after **Zeno's paradox of Achilles and the tortoise**. To rule it out, and to ensure t grows without bound, we'll use the following fairness condition

$$\forall r \in Real : \mathrm{WF}_t(TNext \land t' > r)$$

which is equivalent to $\forall r \in Real : \Diamond(t' > r)$ but makes it clear that this is indeed a fairness property. We can now finish the time-flow part of the specification:

$$\begin{aligned} RT \triangleq & \land t \in Real \\ & \land [TNext]_t \\ & \land \forall r \in Real : \mathrm{WF}_t(TNext \land t' > r) \end{aligned}$$

The complete specification will then be:

$$RTHourClock \triangleq RT \land HCTime$$

It is also true that $RTHourClock \Rightarrow HourClock$; our realtime clock *implements* the less detailed $HourClock$ specification, which serves as an abstraction of the realtime clock.

Specifying Systems shows how we can define just two general realtime operators that can then be conjoined with an untimed specifications to create a realtime specification. The two operators are responsible for the two roles we've seen: specifying that actions happen within a certain time threshold from a deadline constraint, and specifying that time grows unbound. For even more information on using TLA+ for specifying realtime systems, see Lamport and Abadi's 1993 paper, ***An Old-Fashioned Recipe for Real Time***, or Lamport's 2005 paper, ***Real Time is Really Simple***.

Cyber-Physical Systems

Cyber-physical systems, also called hybrid systems, are discrete systems that interact with the physical world through sensors and/or actuators. While the digital system is specified as a (usually realtime) algorithm, the physical world it interacts with is best modeled as a continuous dynamical system.

The idea of how to do this is very similar to the one used in our realtime specification, only instead of a single continuous variable representing time, we have other variables representing continuous quantities that continuously change over time. The following example is also taken from *Specifying Systems*:

Consider a cyber-physical system where a discrete switch affects the behavior of some object moving along a single dimension. We will denote the object's position as x, and its motion is described by the equation:

$$\ddot{x} + c\dot{x} + f[t] + (\text{IF } switchOn \text{ THEN } kx \text{ ELSE } 0) = 0$$

where c and k are some constants, f is some continuous function, t is time, and $\dot{x}$ and $\ddot{x}$ are the first and second derivatives, respectively, of x with respect to t. In TLA+, we can write it as the equation $D[t, x, \dot{x}, \ddot{x}] = 0$ where D is:

$$D[t, x0, x1, x2 \in Real] \triangleq x2 + x * x1 + f[t] + (\text{IF } switchOn \text{ THEN } k * x0 \text{ ELSE } 0)$$

We then define an operator, $Integrate(t, D, t0, \langle x_0, \ldots, x_n - 1\rangle)$ whose value is the value at time t of the tuple $\langle x, \dot{x}, \ldots, \overset{(n-1)}{x}\rangle$, the time-derivatives of x that form the solution to the ordinary differential equation $D[t, x, \dot{x}, \ldots, \overset{(n-1)}{x}] = 0$, and $\langle x_0, \ldots, x_n - 1\rangle$ are the values of those derivatives at time $t0$. You can find the definition of $Integrate$ on page 178 of ***Specifying Systems***. The partial specification (showing only the continuous part) of our hybrid system, where v stands for the particle velocity $\dot{x}$, is:

$$\text{VARIABLES } x, v, t, switchOn$$

$$\begin{aligned} TNext \triangleq\ & \wedge\ t' \in \{r \in Real : r > t\} \\ & \wedge\ \langle x, v\rangle' = Integrate(t', D, t, \langle x, v\rangle) \\ & \wedge\ \text{UNCHANGED } switchOn \qquad \text{The discrete variable changes instantaneously} \end{aligned}$$

For an example of a hybrid specification with a correctness proof, see ***Hybrid Systems in TLA+***.

A simplified version of TLA that is not stuttering-invariant, basically rTLA from part 3, was implemented in Coq as part of the **VeriDrone project** to reason about quadcopter controllers precisely in this manner.

Biology

A **metabolic network** can also be easily modeled as a nondeterministic state machine. For example:

$$
\begin{array}{l}
\text{VARIABLE } m \text{ The number of each kind of molecule} \\
State \triangleq [a : Nat, \\
\qquad\quad b : Nat, \\
\qquad\quad c : Nat, \\
\qquad\quad d : Nat] \\
TypeOK \triangleq m \in State
\end{array}
$$

$$
\begin{array}{l}
a + b \to c + d \\
Reaction1 \triangleq \land m.a > 0 \\
\qquad\qquad\quad \land m.b > 0 \\
\qquad\qquad\quad \land m' = [m \text{ EXCEPT } !.a = @ - 1 \\
\qquad\qquad\qquad\qquad\qquad\quad\ !.b = @ - 1 \\
\qquad\qquad\qquad\qquad\qquad\quad\ !.c = @ + 1 \\
\qquad\qquad\qquad\qquad\qquad\quad\ !.d = @ + 1]
\end{array}
$$

$$
\begin{array}{l}
c + d \to a \\
Reaction2 \triangleq \land m.c > 0 \\
\qquad\qquad\quad \land m.d > 0 \\
\qquad\qquad\quad \land m' = [m \text{ EXCEPT } !.a = @ + 1 \\
\qquad\qquad\qquad\qquad\qquad\quad\ !.c = @ - 1 \\
\qquad\qquad\qquad\qquad\qquad\quad\ !.d = @ - 1]
\end{array}
$$

$$
\begin{array}{l}
Init \triangleq m \in [a : 0..1000, \\
\qquad\qquad\quad b : 0..1000, \\
\qquad\qquad\quad c : 0..1000, \\
\qquad\qquad\quad d : 0..1000] \\
Next \triangleq Reaction1 \lor Reaction2 \\
Spec \triangleq Init \land [Next]_m \land \mathrm{WF}_m(Next)
\end{array}
$$

We can then state propositions about the system, such as that it has three static attractors:

$$\exists\, atts \in \text{SUBSET } State : Cardinality(atts) = 3 \land \Diamond\Box(m \in atts)$$

Of course, we can make much more complex propositions, such as that the system has no more than two periodical attractors of period less than 100.

See ***A Computational Framework For Complex Diseases*** for a brief mention of TLA in this context.

Conclusion

The goal of this series was to show why a mathematical specification of software systems, rather than a programming language-based one, is often a good choice for reasoning about systems as it is both very powerful and relatively simple, how TLA+ allows such mathematical reasoning, and how this form of reasoning provides insight into similarities between concepts and algorithms that are often obscured in programming languages.

However, I think that the choice of a general, ordinary math formalism for reasoning about algorithms is not enough in itself to rid us of the Whorfian syndrome, as that syndrome has two components. The first is due to the choice of formalisms, like those based on programming languages, that are intentionally restrictive, either for aesthetic or technical reasons, or both. This effect can be seen in a **blog post** by an accomplished engineer:

> *Over the years I've seen a few people argue that there's something fundamentally wrong with the notion of the algorithm because it doesn't apply to the kind of open-ended loop we see in operating systems and interactive applications. Some have even gone further to suggest that somehow mathematics and computer*

> *science are fundamentally different because mathematics can't seek to describe these kinds of open-ended phenomena… [N]ot only are there nice ways to look at open-ended computations, but from a mathematical perspective they are precisely dual, in the category theoretical sense, to terminating computations. It may be true that mathematicians sometimes spend more time with things, and computer scientists with cothings. But this really isn't such a big difference and the same language can be used to talk about both… [I realised] there was this entire co-universe out there…*

By "co-universe", he's referring to pure-FP's way of modeling infinite data streams and infinite computations. Clearly, "this entire co-universe" is not really "out there", but very much deep inside his chosen formalism, one that chooses (for historical and aesthetic reasons) computable functions as the fundamental building blocks of computation. The notion of co-data or co-computation, doesn't really provide any fundamental insight into nonterminating computations, but only into their description in that specific formalism. Bartosz Milewski — a physicist, no less! — also repeats this sentiment in his lectures on category theory in the context of functional programming, **saying**, "time is really hard to describe in mathematics". But mathematics has no difficulty whatsoever in expressing time or "open-ended" discrete systems any more that it finds it difficult to describe open-ended continuous systems. As should now be clear, computations of all kinds are easily and elegantly, expressed in mathematics as infinite sequences of states. The difficulty arrises only when using specific formalisms that explicitly choose to *disregard* time. Whether or not that is a useful simplification is a separate discussion, but the difficulty is not with mathematics.

Formalisms are important, but their importance cannot be properly appreciated if one does not distinguish between insights a formalism provides into its own operation and the far more valuable insights a formalism offers that help discover features of reality. An ideological, almost religious, adherence to specific formalisms, seems to have been common among some logicians almost hundred years ago, long before it became common among programmers; that ideological perspective was no more conducive to understanding mathematics back then as it is for understanding computation now[15]. In this sense, the Whorfian syndrome is a form of an ideological conviction that blinds more than it enlightens.

But the Whorfian syndrome has another component, one that cannot be avoided even by choosing a very general, relatively simple, mathematical framework such as TLA for reasoning about computation. *All* formalisms are merely *descriptions* of computations. Computation is neither a "thing" nor a "cothing"; it isn't an infinite sequence of states, either, but a discrete dynamical system that we can choose to describe and denote in many different formalisms, picking the one that is most convenient for the task at hand. Whichever language we choose to describe reality, we run the risk of confusing the description with reality. Understanding abstraction, and how it also relates a description of reality to reality helps, but it also takes practice.

This is **a post to the TLA$^+$ mailing list**, where Lamport discusses this more general Whorfian syndrome in TLA$^+$

> *You originally wrote:*
>
> > *My conundrum is linked to the fact that if the action is taken the primed variable equations have immediate effect, that is the state transition is instantaneous.*
>
> *An action is a formula. It makes no sense to say that a formula has immediate effect. Does the formula 2+3=5 have immediate effect? That no one else has pointed this out shows that many people don't appreciate the distinction between math and reality.*
>
> *I have a digital clock on my desk. If I cover part of the display, the clock behaves as follows. It shows a pattern of light and dark regions. For about an hour, that pattern remains quite stable, with barely perceptible changes in light intensity. Then, for a fraction of a second, the pattern fluctuates wildly. It then reaches a*