A TLA$^+$ and PlusCal Overview

Some koans and exercises

Rafael Luque

September 17, 2019

OSOCO

Table of contents

1. Introduction

2. TLA$^+$ and PlusCal Semantics

3. PlusCal Koans

4. Test the Water

5. Workshop of Spec Writing

Introduction

TLA$^+$ and PlusCal Semantics

About PlusCal

- Developed by Lamport in 2009 to make TLA$^+$ more accessible to programmers.
- PlusCal provides a pseucode-like structure on top of TLA$^+$.
- Adds additional syntax: assignments and labels.
- PlusCal is language that compiles down to TLA$^+$.

Spec layout

```
---- ❶ MODULE example ❷ ----
EXTENDS Integers ❸

(* --algorithm wire ❹

    variables ❺
        people = { "bob", "alice" },
        acc = [ alice |-> 5, bob |-> 5 ];

    begin ❻
        skip;
    end algorithm; ❹ *)
==== ❶
```

Spec layout

1. TLA$^+$ specs must start with at least four - at each side of `MODULE` and four = at the end.
2. The module name must be the same as filename.
3. `EXTENDS` is the import keyword.
4. `(*...*)` is the block comment. PlusCal spec starts with `--algorithm <name>` and ends with `end algorithm`.
5. Initialization of variables.
6. Where the algorithm is implemented.

Basic values

Four basic values in **TLA**$^+$:

String	Must be written en double quotes.
Integer	Floats are not supported.
Boolean	Written as TRUE and FALSE.
Model value	A kind of symbol value.

Standard operators

Operator	Meaning	Example
`x = y`	Equals	`>> 1 = 2` FALSE
`x /= y`	Not Equals	`>> 1 /= 2` TRUE
`x /\ y`	And	`>> TRUE /\ FALSE` FALSE
`x \/ y`	Or	`>> TRUE \/ FALSE` TRUE
`~x`	Not	`>> ~TRUE` FALSE
`x := y`	Assignment	PlusCal only

Arithmetic Operators

- If you `EXTENDS Integers` get the arithmetic operators: `+`, `-`, `%` and `*`.
- Decimal division is not supported, only the integer division: `\div`.
- You also get the range operator `..` where `a..b` is the set `{a, a+1, ..., b-1, b}`.

Complex Types

TLA$^+$ has four complex types:

- Sets.
- Tuples or sequences.
- Structures.
- Functions.

Sets

Sets

Unordered collections of elements of the same type.

```
{1, 2, 42}
```

```
{{TRUE}, {FALSE, TRUE}, {}}
```

Set Operators

Operator	Meaning	Example
`x \in set`	Is member of	`>> 1 \in 1..3` TRUE
`x \notin set`	Is not member of	`>> 1 \notin 1..2` FALSE
`set1 \subseteq set2`	Is subset of	`>> {1, 2} \subseteq {1, 2, 3}` TRUE
`set1 \union set2`	Union	`>> {1, 2} \union (2..3)` `{1, 2, 3}`
`set1 \intersect set2`	Intersection	`>> {1, 2} \intersect (2..3)` `{2}`
`set1 \ set2`	Difference	`>> {1, 2} \ (2..3)` `{1}`
`Cardinality(set)`	Cardinality (requires EXTENDS FiniteSets)	`>> Cardinality({1, 2})` 2

Set Transformations

Filter sets

```
{x \in set: conditional}
```

```
>> {x \in 1..3: x > 2}
{3}
```

Map sets

```
{expression: x \in set}
```

```
>> {x * 3: x \in 1..3}
{3, 6, 9}
```

Tuples or Sequences

Tuples or Sequences

Ordered collections of elements with the index starting at 1.

```
tuple := <<1, TRUE, {1, 2}>>;

>> tuple[1]
1
>> tuple[3]
{1, 2}
```

Sequence Operators

If you **`EXTENDS Sequences`** you get the following additional operators:

Operator	Meaning	Example
`Head(seq)`	Head	`>> Head(<<1, 2>>` 1
`Tail(seq)`	Tail	`>> Tail(<<1, 2, 3>>` `<<2, 3>>`
`Append(seq, x)`	Append	`>> Append(<<1, 2>>, 3` `<<1, 2, 3>>`
`seq1 \o seq2`	Combine	`>> <<1, 2>> \o <<3>>` `<<1, 2, 3>>`
`Len(seq)`	Length	`>> Len(<<1, 2>>)` 2

Structures or Structs

Structures or Structs

A map of strings to values.

```
>> [a |-> 1, b |-> <<1, {2, 3}>>].b
   <<1, {2, 3}>>
```

Assignments

Assign an existing variable to a value with `:=`.

Rule of thumb

If it's the first time you're using the variable, = is initialization.
Every other time, = is equality and := is assignment.

assert

assert

`assert TRUE` does nothing, `assert FALSE` raises an error.

In order to use assertions you need to add `EXTENDS TLC` to the spec.

skip

- A no-op.
- To fill parts of the spec that we haven't filled out yet or conditionals that don't update anything.

if

```
if condition1 then
  body1
elsif condition2 then
  body2
else
  body3
end if;
```

while loop

```
while condition do
  body
end while;
```

macros

To avoid duplications in your specs you can add macros before the begin of the algorithm:

```
macro name(arg1, arg2) begin
  \* macro's body
end macro;

begin
  name(x, y);
end algorithm;
```

Macros limitations

You can place assignments, assertions, and if statements in macros, but not while loops. You also cannot assign to any variable more than once.

Multiple Starting States i

We need a way to specify not just one setup, but an entire space of setups to check our specifications.

Multiple Starting States ii

We initialize variables with =, but we can also initialize them with **\in**.

TLC will try running the algorithm with any possible element in the set:

```
(* --algorithm in
variables x \in 1..3};

begin
  assert x < 3;
end algorithm; *)
```

Multiple Starting States iii

BOOLEAN set

TLA$^+$ defines **BOOLEAN** as the set `{TRUE, FALSE}`. This can be useful if you have a variable `isReady \in BOOLEAN`.

Multiple Starting States iv

SUBSET

SUBSET is the power set, or the set of all subsets.

```
>> SUBSET 1..2
   {{}, {1}, {2}, {1, 2}}
```

Multiple Starting States v

\X

set1 \X set2 is the set of all tuples where the first element is in set1 and the second element in set2.

```
>> (1..2) \X BOOLEAN
   {<<1, TRUE>>, <<1, FALSE>>, <<2, TRUE>>, <<2, FALSE>>}
```

Multiple Starting States vi

[key: set]

If `x \in [key: set]`, then x is an structure where the value of key is an element in set.

```
>> [a: (1..2), b: BOOLEAN]
{[a |-> 1, b |-> TRUE], [a |-> 2, b |-> TRUE],
  [a |-> 1, b |-> FALSE], [a |-> 2, b |-> FALSE]}
```

Beware of state explosion i

```
variables
  capacity = [trash |-> 10, recycle |-> 10],
  items = <<
    [type |-> "trash", size |-> 5],
    [type |-> "recycle", size |-> 3],
    [type |-> "recycle", size |-> 4],
    [type |-> "trash", size |-> 2]
  >>;
```

Beware of state explosion ii

```
variables
  capacity \in [trash: 1..10, recycle: 1..10],
  item \in [type: {"trash", "recycle"}, size: 1..6],
  items \in item \X item \X item \X item;
```

1 initial state vs. $10x10x(2x6)^4 = $ **2,073,600 initial states**

Simulating Nondeterminism i

For single process algorithms PlusCal provides two constructs to simulate nondeterminism:

Simulating Nondeterminism ii

```
either
  /* branch 1
or
  /* branch 2
/* ...
or
  /* branch n
end either;
```

- TLC will check all branches.
- No way to make one branch more likely than others.
- If all branches are *macro-valid*, we can place an either inside a macro.

Simulating Nondeterminism iii

There are two ways to use the **with** statement:

```
with var = value do
  /* body
end with;
```

```
with var \in set do
  /* body
end with;
```

- The former creates a temporary variable, the second is nondeterministic.
- TLC will check what happens to all possible assignments of var to elements of set.
- with follows macro rules: no double assignments and no while loops.
- You can place with statements inside macros.

Operators i

Operators

An **operator** is the **TLA**$^+$ equivalent to procedures in programming languages.

```
OpWithArgs(Arg1, Arg2) == Expression

OpWithoutArgs == Expression
```

Operators ii

Higher-order Operators

Operators that take other opertators as arguments.

```
Add(a, b) == a + b
Apply(Op(_, _), x, y) == Op(x, y)

>> Apply(Add, 2, 3)
5
```

Operators iii

Anonymous Operators

You can define anonymous operators with `LAMBDA param1, param2, paramN: body`.

They can only be used as arguments of other operators, not as standalone operators.

```
>> Apply(LAMBDA x, y: x * y, 2, 3)
6
```

Operators iv

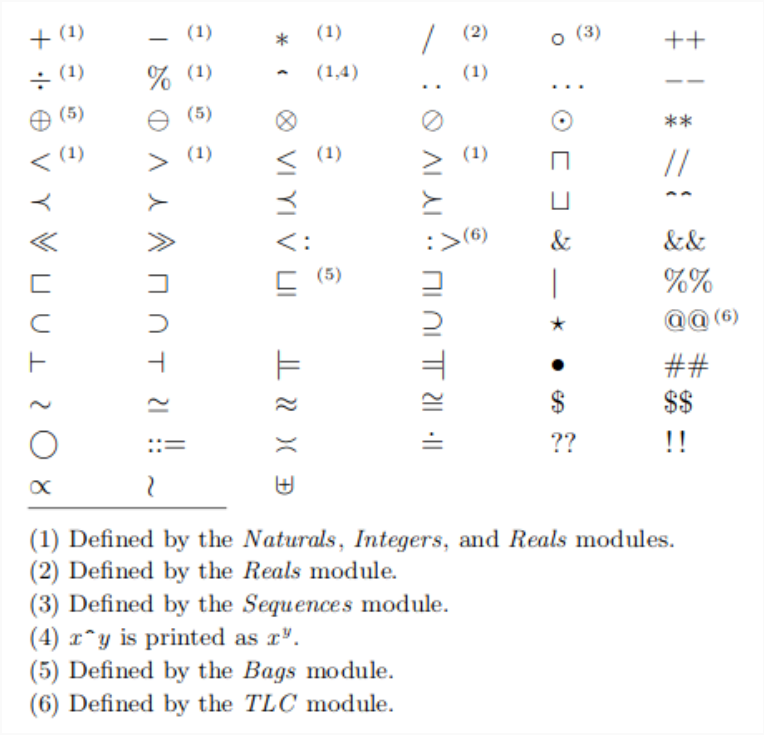
TLA$^+$ also permits definitions of binary (infix) operators.

For example, the following defines $\oplus$ (typed "(+)") to mean addition modulo N:

```
a (+) b == (a + b) % N
```

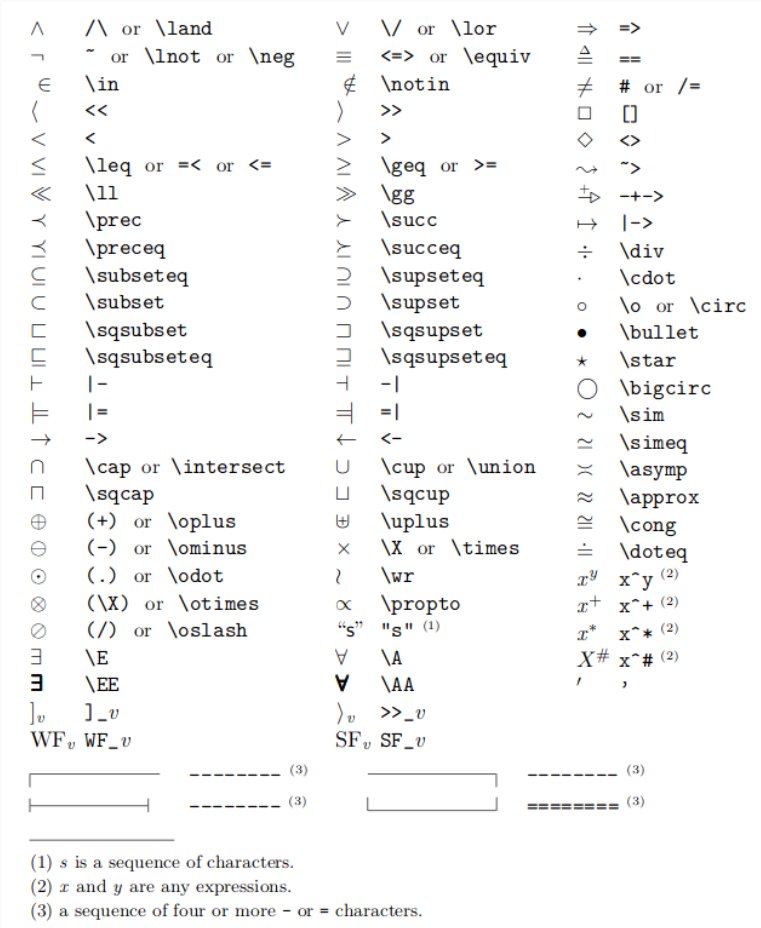
There is a table with the user-definable operator symbols at [**?**].

Operators v



(1) Defined by the *Naturals*, *Integers*, and *Reals* modules.
(2) Defined by the *Reals* module.
(3) Defined by the *Sequences* module.
(4) x^y is printed as x^y.
(5) Defined by the *Bags* module.
(6) Defined by the *TLC* module.

Figure 1: User-definable operator symbols



(1) s is a sequence of characters.
(2) x and y are any expressions.
(3) a sequence of four or more - or = characters.

Figure 2: ASCII for typeset symbols

Operators vi

Operators as constants

If a set of possible values is constant, we define it as an operator instead of a variable. This prevents us from accidentally modifying the set in the algorithm.

```
BinTypes == { "trash", "recycle" }
Items == [ type: BinTypes, size: 1..6 ]
SetsOfFour(set) == set \X set \X set \X set

(* --algorithm recycler
variables
  items \in SetsOfFour(Items);
  ...
```

Operators vii

Operators using PlusCal variables

If you want to define an operator using the variables of a PlusCal algorithm, you should place it in a define block.

Definitions goes below variable definitions and above macro definitions.

Operators viii

```
(* --algorithm recycler
variables
    capacity = [ trash |-> 10, recycle |-> 10 ],
    bins = [ trash |-> {}, recycle |-> {} ],
    count = [ trash |-> 0, recycle |-> 0 ];

define
  NoBinOverflow ==
    capacity.trash >= 0 /\ capacity.recycle >= 0
  CountsMatchUp =
    /\ Len(bins.trash) = count.trash
    /\ Len(bins.recycle) = count.recycle
end define;

\* macros...
```

Invariants

Operators as Invariants

We can use operators as **invariants**. Invariants are boolean expressions that are checked at the end of every state of the model execution. If it's ever false, the model fails.

Logical Operators i

\A

\A means “all elements in a set”. It’s used in the form `\A x \in set: P(x)`, which means “for all elements in the set, P(x) is true”.

```
AllLessThan(set, max) == \A x \in set: x < max

>> AllLessThan({1, 3}, 4)
TRUE

>> AllLessThan({1, 3}, 2)
FALSE
```

Logical Operators ii

\E

\E means “there exists some element in the set”. It’s used in the form `\E x \in set: P(x)`, which means “there is at least one element in the set where P(x) is true”.

```
SeqOverlapsSet(seq, set) == \E x \in 1..Len(seq): seq[x] \in

>> SeqOverlapsSet(<<1, 2, 3>>, {2, 4})
TRUE

>> SeqOverlapsSet(<<1, 2, 3>>, {4, 5})
FALSE
```

Logical Operators iii

`=>`

`P => Q` means that if P is true, then Q is true.

It's equivalent to writing `~P \/ Q`.

Logical Operators iv

<=>

`P <=> Q` means that either P and Q are both true or P and Q are both false.

```
Xor(A, B) == (~A /\ B) \/ (A /\ ~B)
AlternativeXor(A, B) == ~A <=> B

>> \A A, B \in BOOLEAN: Xor(A, B) = AlternativeXor(A, B)
TRUE
```

PlusCal Koans

Test the Water

Workshop of Spec Writing

Questions?

References i