PGo Manual

For commit da1b6e1

August 4, 2017

1 Introduction

TLA+ is a formal specification language, built on the mathematical concepts of set theory and the temporal logic of actions. Using the TLC model checker, TLA+ specifications can be checked exhaustively for specific properties, and the TLA proof system (TLAPS) allows for machine-checked proofs. PlusCal is an algorithm language which can be translated to TLA+ and uses TLA+ as its expression language. It is easy to specify and verify distributed algorithms in PlusCal, thanks to its simple constructs for nondeterminism, concurrency primitives, and rich mathematical constructs. However, PlusCal is not a programming language so it cannot be run, which limits its utility.

PGo aims to correspond a verified PlusCal specification with an executable Go implementation. PGo compiles an annotated PlusCal specification into a Go program which preserves the semantics of the specification. The main goal of PGo is to create a compiled Go program which is easy to read and edit. The intended use case for PGo is for the developer to create and verify a PlusCal spec of the system, compile it with PGo, then edit the compiled program to fully implement details not included in the specification. Programs compiled by PGo are type safe (they do not contain type assertions which panic).

2 Using PGo

2.1 Installation

Requirements: JRE8, and one of Eclipse or Ant 1.9

- Git clone the source at `https://bitbucket.org/whiteoutblackout/pgo/`
  - Option 1: Import as an Eclipse project
  - Option 2: Execute `ant pgo` assuming the project is in the `pgo/` directory

2.2 Execution

To run PGo, run the Eclipse project or run `pgo.sh`. The command-line usage is `pgo [options] pcalfile gofolder gofile`.

Optional command line arguments:

```
-h --help=<boolean> - Print usage information [default false]
-q --logLvlQuiet=<boolean> - Reduce printing during execution [default false]
-v --logLvlVerbose=<boolean> - Print detailed information during execution [default false]
-i --infile=<string> - the input pluscal file to transpile [default ]
-o --outfile=<string> - the output file to generate [default ]
-d --outfolder=<string> - the output folder to put additional packages [default ]
--writeAST=<boolean> - write the AST generated and skip the rest [default false]
```

2.3 Annotations

PGo requires annotation of the PlusCal source file to properly compile the PlusCal source. Annotations must be placed in comments within the body of the PlusCal algorithm, but there can be multiple annotations per comment. Annotations may appear in any part of the comment block containing the PlusCal algorithm. Annotations are of the form `@PGo{ <annotation> }@PGo`.

2.3.1 Annotating types

Many annotations require specifying a type name. PGo uses the following format for type names:

Primitive types	
Go type	Annotation keyword
`int`	`int`
`bool`	`bool`
`uint64`	`uint64`
`float64`	`float64`
`string`	`string`
(void type)	`void`
Collection types	
`[]<type>`	`[]<type>`
set with `<type>` elements	`set[<type>]`
map with `<K>` keys and `<V>` values	`map[<K>]<V>`
tuple with all elements of type `<E>`	`chan[<E>]`
tuple with ith element of type $\texttt{<E}_i\texttt{>}$	$\texttt{tuple[<E}_1\texttt{>,<E}_2\texttt{>,...]}$

Type names cannot contain spaces.

2.3.2 TLA+ definitions and constants declared outside of the algorithm block (Required)

The PlusCal algorithm can make use of TLA+ definitions that are found outside the algorithm block. These are not parsed by PGo and need to be in an annotation for PGo to parse them. Any TLA+ definitions found in `define` blocks also need to be annotated. The annotation for a TLA+ definition is of the form `def <name>(<params>)? <type>? == <TLA expression>`. The definition can be copied almost verbatim from the TLA+, but a parameter is specified by `<name> <type>` so typing information needs to be added to the parameters. The type that the expression should evaluate to may also be specified, if desired. A TLA+ definition without parameters is compiled into a variable, and a definition with parameters is compiled into a function.

```
1 \* outside the algorithm body
2 foo(i, j) == i + j
3 bar == {1, 3, 5}
4 \* ...
5 \* inside the algorithm body
6 (*
7     @PGo{ def foo(i int, j int) int == i + j}@PGo
8     @PGo{ def bar = {1, 3, 5} }@PGo
9 *)
```

PlusCal

```go
func foo(i int, j int) int {
    return i + j
}

var bar pgoutil.Set = pgoutil.NewSet(1, 3, 5)
```

Compiled Go

There will be constants in PlusCal that are not declared in the PlusCal algorithm (e.g., constant `N` for model checking). These variables will need to be specified using PGo annotations either as constants in the compiled Go program, or command line arguments to the Go program. Constants are specified as `const <varname> <type> <val>` indicating that `varname` is a Go constant of type `<type>` with initial Go value `<val>`. Command line argument variables of Go are specified as `arg <varname> <type> (<flagname>)?` indicating that variable `<varname>` is of type `<type>` and is going to be specified through a command line argument to the Go program. If no `<flagname>` is specified, then the variable will be a positional argument in the order that it appeared in the annotation. If `<flagname>` is specified, then the variable will be set on the command line using `-<flagname>=<value>`.

If a constant is not a primitive type, it cannot be declared as constant or as a command line argument in Go. The constant can instead be annotated as a TLA+ definition, where the expression is the desired constant value. This will be compiled to a global variable that is initialized with the given value. PGo provides a compile-time guarantee that the constant indeed remains constant (it is not assigned to or mutated).

```
\* outside the algorithm body
CONSTANT N, M, ProcSet
\* ...
\* inside the algorithm body
(*
    @PGo{ arg N int }@PGo
    @PGo{ arg M int PGoFlag }@PGo
    @PGo{ def ProcSet set[string] == {"a", "b", "c"} }@PGo
*)
print << N, M >>;
```

PlusCal

```go
var ProcSet pgoutil.Set = pgoutil.NewSet("a", "b", "c")
var N int
var M int

func main() {
    flag.IntVar(&M, "PGoFlag", 0, "")
    flag.Parse()
    N, _ = strconv.Atoi(flag.Args()[0])

    fmt.Printf("%v %v\n", N, M)
}
```

Compiled Go

```
$ go run CompiledGo.go -PGoFlag=1 2
2 1
```

Command-line usage

2.3.3 PlusCal procedure return values (Optional)

PlusCal has no return values, so procedures can only return values by writing to a global variable. PGo requires the developer to indicate which variable serves this purpose. In PGo, the annotation is of the form `ret <varname>`. PGo automatically scans all function definitions for the one where the variable is used. Note that using this feature will remove the specified variable from the global variables. If you rely on global state of the variable for more than just the function return value, do not specify it as a return value and use the global variable instead.

2.3.4 Variable types (Optional)

PGo will automatically infer types for variables declared in PlusCal. However, you may want to specify the types rather than using the inferred types (e.g., you want a variable to be a `uint64` rather than an `int`). This is possible by specifying `var <varname> <type>`. Annotations are required for variables that involve PlusCal tuples, since these may compile to slices or tuples depending on context. If no type annotation is provided for a variable, PGo will indicate the type it inferred in the output Go code.

2.3.5 Function signatures

Similar to specifying variable types. The annotation `func <rtype>? <funcname>() <p1type>?+` represents `<funcname>()` having a return type of `<rtype>` if specified, and parameters of type `<p1type>, <p2type>...` If any types are specified, all return types or parameters must be specified.

```
\* @PGo{ func foo() int string }@PGo
procedure foo(a, b) {
    print << a + 1, b >>;
}
```

PlusCal

```
func foo(a int, b string) {
    fmt.Println("%v %v\n", a + 1, b)
}
```

Compiled Go

3 Translation

PlusCal has many language constructs that are translated to Go in a non-trivial way. These constructs and their translations are described in this section. The `pgoutil` Go helper library, which contains some useful data structures and helper methods, is also described.

3.1 Variable declarations

In addition to the simple variable declaration `var = <val>`, PlusCal supports the declaration `var \in <set>`. This asserts that the initial value of `var` is an element of `<set>`. This is translated into a loop in which each set element is assigned to `var`. This is the same as the behaviour of TLC in model-checking mode, and is the most likely use case.

```
variables
    S = {1, 3};
    v \in S;
{
    \* algorithm body...
}
```

PlusCal

```
import "pgoutil"

var S pgoutil.Set = pgoutil.NewSet(1, 3)
var v int

func main() {
    for v_interface := range S.Iter() {
        v = v_interface.(int)
        // algorithm body...
    }
}
```

Compiled Go

(The set methods of the `pgoutil` package are described in 3.4.1.)

3.2 Variable assignment

PlusCal supports multiple variable assignment statements: the statement `x := a || y := b` evaluates the right-hand sides first, then assigns the values to the left-hand sides. A common use is swapping the variables `x` and `y` with the statement `x := y || y := x`.
Go has a multiple assignment construct, but a PlusCal assignment might not translate to a Go assignment (for example, assigning a value to a particular map key translates to the `Put` method). The right-hand sides are first evaluated and assigned to temporary variables, then assigned to their corresponding variables.

```
x := y || y := x || mapping[i] := a
```

PlusCal

```
x_new := y
y_new := x
mapping_new := a
x = x_new
y = y_new
mapping.Put(i, a)
```

Compiled Go

3.3 Macros

PlusCal macros have the same semantics as C/C++ `#define` directives. PGo expands the PlusCal macro wherever it occurs.

```
variables p = 1, q = 2;
macro add(a, b) {
    a := a + b;
}
{
    add(p, q);
    print p;
}
```

PlusCal

```
import "fmt"

var p int = 1
var q int = 2

func main() {
    p = p + q
    fmt.Println("%v", p)
}
```

Compiled Go

3.4 Data types

PlusCal supports maps, sets, and tuples, which are implemented in the `pgoutil` library.

3.4.1 Sets

PlusCal sets are implemented in Go by the `pgoutil.Set` type. This container is an ordered, thread-safe set which has methods equivalent to PlusCal set operations.

```
variables
    A = {1, 2, 3};
    B = 3 .. 6; \* the set {3, 4, 5, 6}
    C = A \union B;
    D = SUBSET C; \* powerset of C
{
    print A = C;
}
```

```
import (
    "fmt"
    "pgoutil"
)

var A pgoutil.Set = pgoutil.NewSet(1, 2, 3)
var B pgoutil.Set = pgoutil.Sequence(3, 6)
var C pgoutil.Set = A.Union(B)
var D pgoutil.Set = C.PowerSet()

func main() {
    fmt.Printf("%v\n", A.Equal(C))
}
```

PlusCal also supports the typical mathematical set constructor notations, which are offloaded to library methods:

```
variables
    S = {1, 5, 6};
    T = {2, 3};
    U = {x \in S : x > 3}; \* equivalent to {5, 6}
    V = {x + y : x \in S, y \in T}; \* equivalent to {3, 4, 7, 8, 9}
\* ...
```

PlusCal

```
import "pgoutil"

var S pgoutil.Set = pgoutil.NewSet(1, 5, 6)
var T pgoutil.Set = pgoutil.NewSet(2, 3)
var U pgoutil.Set = pgoutil.SetConstructor(func(x int) bool {
        return x > 3
    }, S)
var V pgoutil.Set = pgoutil.SetImage(func(x int, y int) int {
        return x + y
    }, S, T)
// ...
```

Compiled Go

While not as concise as the PlusCal, the output Go code is still readable.
The Cartesian product operator `\X` in PlusCal is not associative. For example,

- `<<1, 2, 3>>` is in `Nat \X Nat \X Nat`,
- `<< <<1, 2>>, 3>>` is in `(Nat \X Nat) \X Nat`, and
- `<<1, <<2, 3>> >>` is in `Nat \X (Nat \X Nat)`,

and none of these are equivalent. In Go, `S.CartesianProduct(T, U, V, ...)` is equivalent to `S \X T \X U \X ...`. PGo composes these function calls to create the parenthesized expressions above.

```
variables
    S = {1, 2, 3};
    T = S \X S \X S; \* << 1, 3, 2 >> \in T
    U = S \X (S \X S); \* << 1, << 3, 2 >> >> \in U
```

PlusCal

```
import "pgoutil"

var S pgoutil.Set = pgoutil.NewSet(1, 2, 3)
var T pgoutil.Set = S.CartesianProduct(S, S)
var U pgoutil.Set = S.CartesianProduct(S.CartesianProduct(S))
```

Compiled Go

3.4.2 Maps

PlusCal "functions" are just maps in Go. `pgoutil` implements an ordered, thread-safe map in Go, which supports using slices and tuples as keys (unlike the builtin Go map). There are several syntaxes to declare maps, which are handled by library methods similar to the set methods.
A PlusCal function can be indexed by multiple indices. This is syntactic sugar for a map indexed by a tuple whose components are the indices.

```
variables
    S = {1, 2};
    f = [x \in S, y \in S |-> x + y];
    a = f[2, 2]; \* a = 4
```

PlusCal

```
import "pgoutil"

var S pgoutil.Set = pgoutil.NewSet(1, 2)
var f pgoutil.Map = pgoutil.MapsTo(func(x int, y int) int {
        return x + y
    }, S, S)
var a int = f.Get(pgoutil.NewTuple(2, 2))
```

Compiled Go

(Behind the scenes, the `pgoutil.Map` uses tuples as keys.)

3.4.3 Tuples

PlusCal tuples are used in several different contexts, so variables involving tuples must be annotated with their types. Tuples can store homogeneous data, in which case they correspond to Go slices. Tuple components may be of different types, which correspond to `pgoutil.Tuple`s. Most often in multiprocess algorithms, tuples are used for communication, in which case `pgoutil.Tuple`s are used. PlusCal tuples are 1-indexed, but Go tuples and slices are 0-indexed, so 1 is subtracted from all indices in Go.

```
(* @PGo{ var slice []string }@PGo
   @PGo{ var tup tuple[int,string] }@PGo *)
variables
    slice = << "a", "b", "c" >>;
    tup = << 1, "a" >>;
{
    print slice[2]; \* "b"
}
```

PlusCal

```
import (
    "fmt"
    "pgoutil"
)

var slice []string = []string{"a", "b", "c"}
var tup pgoutil.Tuple = pgoutil.NewTuple(1, "a")

func main() {
    fmt.Printf("%v", slice[2 - 1])
}
```

Compiled Go

3.5 Predicate operations

PlusCal supports the mathematical quantifiers ∀ and ∃. PGo compiles these to the library methods `ForAll` and `Exists`.

```
variables
    S = {1, 2, 3};
    T = {4, 5, 6};
    b1 = \E x \in S : x > 2 \* TRUE
    b2 = \A x \in S, y \in T : x + y > 6 \* FALSE
```

PlusCal

```
import "pgoutil"

var S pgoutil.Set = pgoutil.NewSet(1, 2, 3)
var T pgoutil.Set = pgoutil.NewSet(4, 5, 6)
var b1 bool = pgoutil.Exists(func(x int) bool {
        return x > 2
    }, S)
var b2 bool = pgoutil.ForAll(func(x int, y int) bool {
        return x + y > 6
    }, S, T)
```

Compiled Go

Related is Hilbert's ε operator (called `CHOOSE` in PlusCal). In PlusCal, `CHOOSE x \in S : P(x)` is defined to be some arbitrary value in `S` such that `P(x)` is true. If there is no such `x`, the expression is unspecified. Furthermore, `CHOOSE` is deterministic and always selects the same `x` given equal sets and equivalent predicates. The Go equivalent function `pgoutil.Choose` selects the least element that satisfies the predicate. This guarantees the determinism that the operator provides, and is the same as TLC's behaviour. If no element satisfies the predicate `P`, `pgoutil.Choose`

panics, similar to TLC returning an error when it encounters unspecified behaviour.

```
variables
    S = {1, 2, 3};
    a = CHOOSE x \in S : x >= 1
    b = CHOOSE y \in S : y < 0 \* TLC error
```

PlusCal

```
import "pgoutil"

var S pgoutil.Set = pgoutil.NewSet(1, 2, 3)
var a int = pgoutil.Choose(func(x int) bool {
        return x >= 1
    }, S).(int) // a == 1
var b int = pgoutil.Choose(func(y int) bool {
        return y < 0
    }, S).(int) // panic (no element satisfies predicate)
```

Compiled Go

3.6 With

The PlusCal `with` statement has the syntax

```
variables S_1 = {1, 2, 3}, a = "foo";
\* ...
with (x_1 \in S_1, x_2 = a, ...)
{
    \* do stuff with the x_i ...
}
```

PlusCal

This construct selects a random element from each `S_i` and assigns them to the local variables `x_i`. If the syntax `x_i = a` is used, this simply assigns `a` to `x_i`. In Go, this translates to

```go
import (
    "math/rand"
    "pgoutil"
    "time"
)

var S_1 pgoutil.Set = pgoutil.NewSet(1, 2, 3)
var a string = "foo"
// ...
func main() {
    rand.Seed(time.Now().UnixNano())
    // ...
    {
        x_1 := S_1.ToSlice()[rand.Intn(S_1.Size())].(int)
        x_2 := a
        // ...
        // do stuff with the x_i ...
    }
}
```

Compiled Go

In Go the random number generator is seeded with the current Unix time at the beginning of the `main` function. The local variables declared by the `with` and its body are contained in a separate code block, to preserve the variables' local scope.

3.7 Processes

The PlusCal algorithm body can either contain statements in a uniprocess algorithm, or process declarations in a multiprocess algorithm. Processes can be declared with the syntax `process (Name \in S)` or `process (Name = Id)`. The first construct spawns a set of processes, one for each ID in the set `S`, and the second spawns a single process with ID `Id`. A process can refer to its identifier with the keyword `self`. In TLC, multiprocess algorithms are simulated by repeatedly selecting a random process then advancing it one atomic step (if it does not block).

PGo converts each process body to a function and spawns a goroutine per process. The function takes a single parameter `self`, the process ID. There are two semantic considerations: the main goroutine should not exit before all goroutines finish executing, and the time at which all child goroutines begin executing should be synchronized. To preserve these semantics, PGo uses a global waitgroup which waits on all goroutines, and each process body pulls from a dummy channel before beginning execution. When all goroutines have been initialized, the dummy channel is closed so that the channel pull no longer blocks, allowing for a synchronized start.

The following is a simple example:

```
variables
    idSet = {1, 2, 3};
    id = "id";

process (PName \in idSet)
    variable local;
{
    local := self;
}

process (Other = id) {
    print self;
}
```

PlusCal

```
import (
    "fmt"
    "pgoutil"
    "sync"
)

var idSet pgoutil.Set = pgoutil.NewSet(1, 2, 3)
var id string = "id"
var PGoWait sync.WaitGroup
var PGoStart chan bool = make(chan bool)

func PName(self int) {
    var local int
    defer PGoWait.Done()
    <-PGoStart
    local = self
}

func Other(self string) {
    defer PGoWait.Done()
    <-PGoStart
    fmt.Println("%v\n", self)
}

func main() {
    // spawn goroutines for PName processes
    for procId := range idSet.Iter() {
        PGoWait.Add(1)
        go PName(procId.(int))
    }
    PGoWait.Add(1)
    go Other(id)
    close(PGoStart)
    PGoWait.Wait()
}
```

Compiled Go

Note that all processes use the same waitgroup (`PGoWait`) and dummy channel (`PGoStart`).

3.8 Labels

In PlusCal, labels are used as targets for `goto` statements and also to specify atomic operations. If a statement is labeled, all statements until the next label are considered to be a single atomic operation. PGo does not yet support the second use of labels. In Go, unused labels cause compile errors so PGo only inserts labels when they are used in `goto` statements.

3.9 Limitations

Not all PlusCal specifications can be compiled by PGo. This is an overview of some important PlusCal features that are currently unsupported.

- Using labels to denote atomic operations
- The `await` keyword
- The `either` construct
- TLA+ features:
  - Alignment of boolean operators in bulleted lists determining precedence
  - Records (structs)
  - Bags (multisets)
  - `IF ... THEN ... ELSE` and `CASE` expressions
  - The `LET .. IN` construct
  - Temporal logic operators
  - Recursive definitions
  - Operator definition (use definitions with parameters instead)

The following are features that may be desirable for a developer working with the compiled program but are not yet considered by PGo:

- Support for networking
- Avoiding data races caused by variable assignments in multithreaded programs
- Interfacing with other programs
- Reading input from sources other than the command line

4 Notes on the Go library

The `pgoutil` Go package provides some useful data structures and methods that correspond to PlusCal language features. The data structures are generic and provide no runtime type safety. PGo guarantees at compile-time that the compiled code does not contain any type assertions that will panic. This section contains some useful information about these containers for developers wishing to modify and optimize the compiled Go code.

4.1 Maps

The Go library map is mutable, but the PlusCal map is not. An assignment to a component of a PlusCal map does not change the value of other references to that map:

```
variables
    S = {1, 2, 3};
    M = [x \in S |-> 2 * x];
    T = {M};
{
    print M \in T; \* TRUE
    M[2] := 3;
    print M \in T; \* FALSE
}
```

The `pgoutil.Map` clones any map that is to be inserted into a map or a set. This preserves semantics but may unnecessarily slow the program, if the map is never mutated. Similarly, since slices are mutable but PlusCal tuples are immutable, slices are cloned before being used as map keys or values.
The Go map provides three iterators: `Keys` and `Values` range over the keys and values stored in the map ordered by keys, and `Iter` ranges over `KVPair`s, which are `struct`s with two fields: `Key` and `Val`.
If a PlusCal map is indexed by multiple indices, the Go map internally uses a `Tuple` of the indices as its key. The value `M[a, b]` in PlusCal can be recovered in Go by calling `M.Get(pgoutil.NewTuple(a, b))`.
The `pgoutil.MapsTo` method makes extensive use of reflection. Performance-critical sections of the application should be rewritten to avoid the use of this method.
The `pgoutil` package exposes a `Map` interface. A custom data type which implements `pgoutil.Map` is interoperable with the builtin type.

4.2 Sets and predicates

The `pgoutil.Set` is implemented using the `pgoutil.Map` as a base, so much of the same semantics apply.
The sets in the Go set library are mutable, whereas TLA+ sets are immutable. However, all operations equivalent to TLA+ set operations are implemented as methods which do not mutate the Go set. For performance purposes, sets are not cloned when inserted into a map or a set. The developer modifying the compiled Go program should exercise caution when making sets of sets or maps with sets if mutating the sets that are inserted. The sets should be copied with the `Clone` method before insertion.
Simple optimizations that can be made by the developer when dealing with sets include replacing the PlusCal `S := S \union {elt}` with simply adding `elt` to `S` and similarly replacing `S := S \ {elt}` with removing `elt` from `S`. Since this mutates the set, the set must be cloned if it is inserted into a set or used with a map.
Elements inserted into a set must be of the same type; otherwise, the program will panic at runtime. This is the same behaviour as TLC, which throws an error when one set contains elements of different types. At compile-time, PGo will throw an error if elements in a set are not of the same type.
The `SetConstructor` and `SetImage` methods, along with the quantifiers `ForAll` and `Exists` and the `Choose` method, make use of the `reflect` package. This drastically improves the readability of the compiled Go program, but it also incurs a significant performance penalty. Performance-critical sections of the application should be rewritten to avoid use of these methods.
Since slices are mutable, but PlusCal tuples are immutable, slices are cloned before insertion into a set.
The `pgoutil` library exposes a `Set` interface. The developer modifying the Go code may wish to write a custom set container (say, which is specialized for a specific data type). If this container implements the `pgoutil.Set` interface, it is interoperable with the builtin set.

4.3 Tuples

The `pgoutil.Tuple` implements an immutable tuple type which holds elements of arbitrary type, backed by a slice. The `At` and `Set` tuple methods panic if the index accessed is out of range.

5 Example programs

5.1 Euclidean algorithm

The Euclidean algorithm is a simple algorithm that computes the greatest common divisor of two integers, and is a good example PlusCal algorithm.

```
------------------------ MODULE Euclid ----------------------------
EXTENDS Naturals, TLC
CONSTANT N

(*
--algorithm Euclid {
(* @PGo{ arg N int }@PGo *)
  variables u = 24;
            v \in 1 .. N;
            v_init = v;
  {
  a: while (u # 0) {
      if (u < v) {
          u := v || v := u;
      };
  b: u := u - v;
    };
    print <<24, v_init, "have gcd", v>>
  }
}
*)
=============================================================================
```

PlusCal

```
package main

import (
    "flag"
    "fmt"
    "pgoutil"
    "strconv"
)

var v int // PGo inferred type int
var u int // PGo inferred type int
var v_init int // PGo inferred type int
var N int


func main() {
    flag.Parse()
    N, _ = strconv.Atoi(flag.Args()[0])

    for v_interface := range pgoutil.Sequence(1, N).Iter() {
        v = v_interface.(int)
        u = 24
        v_init = v
        for u != 0 {
            if u < v {
                u_new := v
                v_new := u
                u = u_new
                v = v_new
            }
            u = u - v
```

```
32          }
33          fmt.Printf("%v %v %v %v\n", 24, v_init, "have gcd", v)
34      }
35  }
```

Compiled Go

The constant `N` needs to be annotated since its declaration does not appear in the comment containing the algorithm. Note the code to swap `u` and `v` in lines 26-29 of the Go program.

5.2 N-Queens problem

This PlusCal algorithm computes all possible ways to place n queens on an $n \times n$ chessboard such that no two queens attack each other. It demonstrates the expressive power of PlusCal's set constructs, as the algorithm is very concise.

```
1  ------------------------- MODULE QueensPluscal -----------------------------
2  EXTENDS Naturals, Sequences
3  (***************************************************************************)
4  (* Formulation of the N-queens problem and an iterative algorithm to solve *)
5  (* the problem in TLA+. Since there must be exactly one queen in every row *)
6  (* we represent placements of queens as functions of the form *)
7  (* queens \in [ 1..N -> 1..N ] *)
8  (* where queens[i] gives the column of the queen in row i. Note that such *)
9  (* a function is just a sequence of length N. *)
10 (* We will also consider partial solutions, also represented as sequences *)
11 (* of length \leq N. *)
12 (***************************************************************************)
13
14 CONSTANT N \** number of queens and size of the board
15 ASSUME N \in Nat \ {0}
16
17 (* The following predicate determines if queens i and j attack each other
18    in a placement of queens (represented by a sequence as above). *)
19 Attacks(queens,i,j) ==
20   \/ queens[i] = queens[j] \** same column
21   \/ queens[i] - queens[j] = i - j \** first diagonal
22   \/ queens[j] - queens[i] = i - j \** second diagonal
23
24 (* A placement represents a (partial) solution if no two different queens
25    attack each other in it. *)
26 IsSolution(queens) ==
27   \A i \in 1 .. Len(queens)-1 : \A j \in i+1 .. Len(queens) :
28        ~ Attacks(queens,i,j)
29
30 (***************************************************************************)
31 (* We now describe an algorithm that iteratively computes the set of *)
32 (* solutions of the N-queens problem by successively placing queens. *)
33 (* The current state of the algorithm is given by two variables: *)
34 (* - todo contains a set of partial solutions, *)
35 (* - sols contains the set of full solutions found so far. *)
36 (* At every step, the algorithm picks some partial solution and computes *)
37 (* all possible extensions by the next queen. If N queens have been placed *)
38 (* these extensions are in fact full solutions, otherwise they are added *)
39 (* to the set todo. *)
40 (***************************************************************************)
41
42 (* --algorithm QueensPluscal
```

```
    (** @PGo{ arg N int }@PGo
        @PGo{ var todo set[[]int] }@PGo
        @PGo{ var sols set[[]int] }@PGo
        @PGo{ def Attacks(queens []int,i int,j int) ==
              \/ queens[i] = queens[j] \** same column
              \/ queens[i] - queens[j] = i - j \** first diagonal
              \/ queens[j] - queens[i] = i - j \** second diagonal }@PGo
        @PGo{ def IsSolution(queens []int) ==
              \A i \in 1 .. Len(queens)-1 : \A j \in i+1 .. Len(queens) :
              ~ Attacks(queens,i,j) }@PGo **)
    variables
      todo = { << >> };
      sols = {};

    begin
nxtQ: while todo # {}
      do
        with queens \in todo,
             nxtQ = Len(queens) + 1,
             cols = { c \in 1..N : ~ \E i \in 1 .. Len(queens) :
                                    Attacks( Append(queens, c), i, nxtQ ) },
             exts = { Append(queens,c) : c \in cols }
        do
          if (nxtQ = N)
          then todo := todo \ {queens}; sols := sols \union exts;
          else todo := (todo \ {queens}) \union exts;
          end if;
        end with;
      end while;
    end algorithm
*)
==========================================================================
```

PlusCal

```
package main

import (
    "flag"
    "math/rand"
    "pgoutil"
    "strconv"
    "time"
)

var todo pgoutil.Set = pgoutil.NewSet([]int{})
var sols pgoutil.Set = pgoutil.NewSet()
var N int

func Attacks(queens []int, i int, j int) bool {
    return queens[i - 1] == queens[j - 1] || queens[i - 1] - queens[j - 1] == i - j || queens[j - 1] - quee
}
func IsSolution(queens []int) bool {
    return pgoutil.ForAll(func(i int) bool {
                return pgoutil.ForAll(func(j int) bool {
                           return !Attacks(queens, i, j)
                       }, pgoutil.Sequence(i + 1, len(queens)))
```

```
            }, pgoutil.Sequence(1, len(queens) - 1))
}

func main() {
    rand.Seed(time.Now().UnixNano())
    flag.Parse()
    N, _ = strconv.Atoi(flag.Args()[0])

    for !pgoutil.NewSet().Equal(todo) {
        {
            queens := todo.ToSlice()[rand.Intn(todo.Size())].([]int)
            nxtQ := len(queens) + 1
            cols := pgoutil.SetConstructor(pgoutil.Sequence(1, N), func(c int) bool {
                   return !pgoutil.Exists(func(i int) bool {
                               return Attacks(append(queens, c), i, nxtQ)
                           }, pgoutil.Sequence(1, len(queens)))
                })
            exts := pgoutil.SetImage(func(c int) []int {
                   return append(queens, c)
                }, cols)
            if (nxtQ == N) {
                todo = todo.Difference(pgoutil.NewSet(queens))
                sols = exts.Union(sols)
            } else {
                todo = exts.Union((todo.Difference(pgoutil.NewSet(queens))))
            }
        }
    }
}
```

Compiled Go

Note that the variables `cols` and `exts` do not require a type annotation since PGo can infer their type based on the type of `todo`.

5.3 Dijkstra's mutex algorithm

This is a multiprocess algorithm which only allows one process to be in the critical section at one time.

```
--------------------------- MODULE DijkstraMutex ---------------------------
EXTENDS Integers
(*****************************************************************************)
(* There is no reason why the processes need to be numbered from 1 to N. *)
(* So, we assume an arbitrary set Proc of process names. *)
(*****************************************************************************)
CONSTANT Proc

(*********
Here is the PlusCal version of this algorithm.
The algorithm was modified from the original by adding the variable temp2,
to avoid a type consistency conflict when temp changes type at Li4a.
(* @PGo{ def Proc == 1 .. 10 }@PGo *)

 --algorithm Mutex
 { variables b = [i \in Proc |-> TRUE], c = [i \in Proc |-> TRUE], k \in Proc;
   process (P \in Proc)
     variable temp, temp2 ;
     { Li0: while (TRUE)
```

```
            { b[self] := FALSE;
              Li1: if (k # self) { Li2: c[self] := TRUE;
                                   Li3a: temp := k;
                                   Li3b: if (b[temp]) { Li3c: k := self } ;
                                   Li3d: goto Li1
                                  };
             Li4a: c[self] := FALSE;
                   temp2 := Proc \ {self};
             Li4b: while (temp2 # {})
                    { with (j \in temp2)
                       { temp2 := temp2 \ {j};
                         if (~c[j]) { goto Li1 }
                       }
                    };
               cs: skip; \* the critical section
              Li5: c[self] := TRUE;
              Li6: b[self] := TRUE;
              ncs: skip \* non-critical section ("remainder of cycle")
            }
    }
 }
*********)
==============================================================================
```

PlusCal

```
package main

import (
    "math/rand"
    "pgoutil"
    "sync"
    "time"
)

var k int // PGo inferred type int
var b pgoutil.Map // PGo inferred type map[int]bool
var c pgoutil.Map // PGo inferred type map[int]bool
var Proc pgoutil.Set
var PGoWait sync.WaitGroup
var PGoStart chan bool = make(chan bool)

func P(self int) {
    var temp int // PGo inferred type int
    var temp2 pgoutil.Set // PGo inferred type set[int]
    defer PGoWait.Done()
    <-PGoStart
    for true {
        b.Put(self, false)
        Li1:
        if k != self {
            c.Put(self, true)
            temp = k
            if b.Get(temp).(bool) {
                k = self
            }
            goto Li1
```

```
        }
        c.Put(self, false)
        temp2 = Proc.Difference(pgoutil.NewSet(self))
        for !pgoutil.NewSet().Equal(temp2) {
            {
                j := temp2.ToSlice()[rand.Intn(temp2.Size())].(int)
                temp2 = temp2.Difference(pgoutil.NewSet(j))
                if !c.Get(j).(bool) {
                    goto Li1
                }
            }
        }
        // TODO skipped from pluscal
        c.Put(self, true)
        b.Put(self, true)
        // TODO skipped from pluscal
    }
}

func main() {
    rand.Seed(time.Now().UnixNano())
    for k_interface := range Proc.Iter() {
        k = k_interface.(int)
        b = pgoutil.MapsTo(func(i int) bool {
                return true
            }, Proc)
        c = pgoutil.MapsTo(func(i int) bool {
                return true
            }, Proc)
        Proc = pgoutil.Sequence(1, 10)
        for procId := range Proc.Iter() {
            PGoWait.Add(1)
            go P(procId.(int))
        }
        close(PGoStart)
        PGoWait.Wait()
    }
}
```

Compiled Go

The constant `Proc` is defined to be `1 .. 10` in the annotation. If the process set needs to be changed, only the annotation needs to be edited.