Corresponding formal specifications with distributed systems

by

Minh Nhat Do

B. Eng, Nanyang Technological University, 2014

A THESIS SUBMITTED IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF

Master of Science

in

THE FACULTY OF GRADUATE AND POSTDOCTORAL STUDIES
(Computer Science)

The University of British Columbia
(Vancouver)

April 2019

The following individuals certify that they have read, and recommend to the Faculty of Graduate and Postdoctoral Studies for acceptance, the thesis entitled:

Corresponding formal specifications with distributed systems

submitted by **Minh Nhat Do** in partial fulfillment of the requirements for the degree of **Master of Science** in **Computer Science**.

Examining Committee:

Ivan Beschastnikh, Computer Science
Supervisor

William Bowman, Computer Science
Supervisory Committee Member

Abstract

As the need for computing resources grows, providers are increasingly relying on distributed systems to render their services. However, distributed systems are hard to design and implement. As an aid for design and implementation, formal verification has seen a growing interest in industry. For example, Amazon uses Temporal Logic of Actions plus (TLA$^+$) and PlusCal specification languages and tool chain to formally verify manually created specifications of their web services [8].

Nevertheless, there is currently no tool to automatically establish a correspondence between a PlusCal specification with a concrete implementation. Furthermore, PlusCal was not designed with modularity in mind, so a large PlusCal specification cannot be decomposed into smaller ones for ease of modification. This thesis proposes an extension to PlusCal, named Modular PlusCal, as well as a compiler, named PGo, which compiles Modular PlusCal and PlusCal specifications into Go programs. Modular PlusCal introduces new constructs, such as archetypes and mapping macros, to provide isolation and, as a result, modularity. By automatically compiling PlusCal and Modular PlusCal specifications into distributed system implementations, PGo reduces the burden on programmers trying to ensure the correctness of their distributed systems.

Lay Summary

Distributed software systems are notoriously hard to build correctly due to their sheer scale and complexity. Several approaches have been proposed to ensure the correctness of these systems, such as various forms of testing, monitoring, and formal verification. In formal verification, the programmer typically has to build the system twice, once as a formally verified specification and again as a real implementation.

This thesis proposes a compiler to convert a formally verified specification into a real implementation, easing the burden on programmers.

Preface

The work presented in this thesis was conducted by the author in collaboration with Finn Hackett, Renato Costa, Brendan Zhang, and Adam Geller, under the supervision of Dr. Ivan Beschastnikh.

The design of Modular PlusCal (Chapter 3) was joint work by Finn Hackett, Renato Costa, and the author. The design and implementation of the parsing pass and PGo's internal representation of (Modular) PlusCal and TLA$^+$ were joint work by Finn Hackett and the author (Section 4.1.1). The design and implementation of the configuration parsing pass were joint work by Finn Hackett and Renato Costa (Section 4.2.1). The design and implementation of the macro expansion pass were performed by Finn Hackett (Section 4.1.2). The design and implementation of the validation pass were performed by Renato Costa (Section 4.1.4). The design and implementation of the scoping pass were joint work by Finn Hackett and the author (Section 4.1.5). The design and implementation of the Modular PlusCal to Go code generation were performed by Renato Costa. The design and implementation of the PlusCal/TLA$^+$ to Go code generation were joint work by Finn Hackett and the author (Section 4.2.4). The design and implementation of the run-time system were joint work by Finn Hackett, Renato Costa, and the author. The type inference (Section 4.2.2) and atomicity inference (Section 4.2.3) stages were designed and implemented by the author.

Table of Contents

List of Tables

List of Figures

Glossary

API application programming interface

DSL domain-specific language

IR intermediate representation

TLA$^+$ Temporal Logic of Actions plus (see http://lamport.azurewebsites.net/tla/high-level-view.html)

TLAPS TLA$^+$ Proof System

TLC Temporal Logic Checker

Acknowledgments

This research was funded by the Natural Science and Engineering Research Council of Canada (NSERC) Discovery Grant.

I would like to thank my supervisor, Dr. Ivan Bestchasnikh, for his guidance throughout my master program and his support for my pursuit of academic interests. I'd like to thank Renato Costa for all the discussions about the project and life in general that kept me sane. I'd like to thank Finn Hackett, and Brendan Zhang, who put so much work into the project. Finally, I'd like to thank my wife, my mom, and NSS lab-mates for their support throughout my program.

Chapter 1

Introduction

Distributed systems are hard to reason about due to the asynchronous interactions among their constituent components. An obscure series of interactions among the components may lead to subtle yet catastrophic bugs. For example, Amazon's Elasic Compute Cloud (EC2) hit a rare race condition which caused serious downtime for the service and took down major sites on the Internet [1].

Formal verification is one approach to help programmers ensure correctness of distributed systems. In formal verification, the programmer has to provide a model and specification of the system alongside its implementation. A model of the system is a tractable representation of the implementation. A specification consists of some properties and invariants which formally establish what it means for the model to be correct. The specification and model of the system are written in a formal verification language, of which Temporal Logic of Actions plus (TLA$^+$)[1] is an example. According to TLA$^+$ documentation by Lamport [5, 17], a TLA$^+$ specification and model are together referred to as a TLA$^+$ specification, while a TLA$^+$ model is an instance of the specification with all constants instantiated with finite values. Thus, this thesis follows this convention.

Once a TLA$^+$ specification of a system is available, the programmer can use the Temporal Logic Checker (TLC), a model checker bundled in the TLA$^+$ tool box, to check for violations of important system properties. For TLA$^+$ specifications with very large state spaces, the programmer can run TLC on a group of virtual

[1] https://lamport.azurewebsites.net/tla/tla.html

machines using cloud-based distributed TLC [4] to reduce model checking time. Alternatively, the programmer can use the TLA$^+$ Proof System (TLAPS) [3] to provide proofs of the system's correctness.

Beside accepting TLA$^+$ specifications as inputs, the TLA$^+$ tool box also supports specifications written in the algorithm language PlusCal[2]. PlusCal is designed as a replacement for pseudocode so its syntax is closer to popular imperative languages than TLA$^+$'s syntax. In addition, PlusCal also has constructs for specifying nondeterminism and synchronization between asynchronous system components. Specifications written in PlusCal are compiled to TLA$^+$ so they enjoy the same support from the tool box. Thanks to these useful features, PlusCal has been adopted by large software development companies like Amazon and Microsoft [1, 2].

PlusCal allows the programmer to structure the distributed system as a collection of cooperating processes. Within one process, the programmers are free to structure the single-process computation however they choose. This is in contrast to approaches such as Mace [14, 15], and P# [9], where the specification must be structured as a state machine, complete with explicit states and state transitions. We discuss more about these languages, and other approaches in Chapter 7.

Although PlusCal has many useful features, it suffers from the lack of composability: the programmer cannot reuse concepts defined in one PlusCal specification in another PlusCal specification. This thesis proposes Modular PlusCal, an extension of PlusCal, to address this problem.

With this approach, the programmer essentially has to write the system twice: once as a specification, and again as a real implementation. Beside the duplication of effort, the real implementation may deviate from the model, violating correctness properties which are checked of the specification.

To mitigate these shortcomings, this thesis also contributes PGo, a compiler from Modular PlusCal, or PlusCal, specifications to distributed Go programs. By automating the conversion, PGo removes the duplication of effort and reduces the burden on programmers.

In summary, this thesis makes the following two contributions.

- Modular PlusCal is an extension of PlusCal with features for composing

[2] https://lamport.azurewebsites.net/tla/high-level-view.html

specifications (Chapter 3).

- PGo is a compiler from a Modular PlusCal or PlusCal specification to a distributed Go program (Chapter 4).

Chapter 2

Background

This chapter is an overview of TLA$^+$, PlusCal, and the TLA$^+$ tool box.

2.1 TLA$^+$

TLA$^+$ is a formal specification language, designed for verification of concurrent systems. A TLA$^+$ specification is composed of expressions and operators with the occasional constant/variable declarations and module extension. Figure 2.1 shows a specification of Euler's greatest common denominator algorithm in TLA$^+$.

A TLA$^+$ specification starts with the name of the module, surrounded by dashes (line 1 of Figure 2.1). The TLA$^+$ standard library includes some built-in modules, of which `Integers` is one. `Integers` includes the definitions of the less-than (`<`) and minus (`-`) operators. `EXTENDS Integers` (line 2 of Figure 2.1) pulls these definitions into scope. `CONSTANTS` declares constants to be used in the specification, which have to be specified when model checking. `VARIABLES`, as the name suggests, declares variables to be used in the following expressions or declarations. Nullary operator declaration (line 8, 11, and 20 of Figure 2.1) starts with the operator name, followed by double equals (`==`), and ends with the body of the operator, which is a TLA$^+$ expression. A TLA$^+$ conjunction can be written either as an infix operator, e.g. `TRUE /\ TRUE`, or as a prefix operator, as shown on lines 8 to 18 of Figure 2.1. When written as prefix operators, they have to align at the same column in the text. The same rules apply for disjunctions (`\/`).

```
1  ----------- MODULE GCD -----------
2  EXTENDS Integers
3
4  CONSTANTS N, M
5
6  VARIABLES n, m
7
8  Init == /\ n = N
9          /\ m = M
10
11 Next == \/ /\ n < m
12            /\ m' = m - n
13            /\ n' = n
14         \/ /\ m < n
15            /\ n' = n - m
16            /\ m' = m
17         \/ /\ n' = n
18            /\ m' = m
19
20 Spec == Init /\ [][Next]_<<n, m>>
21 ================================
```

Figure 2.1: Euler's greatest common denominator algorithm in TLA$^+$

`Init`, `Next`, and `Spec` are default names for predicates that constitute a specification in TLA$^+$. The initial predicate, called `Init` in Figure 2.1, sets the variables to their initial values. The next-state relation, called `Next` in Figure 2.1, relates the current value of a variable with its next value, for all variables, by putting an equality constraint on the variable, e.g., `n`, and its primed version, `n'`. Finally, `Spec` is a conjunction of the initial predicate `Init` and the next-state relation `Next`. The syntax `[][Next]_<<n, m>>` says that each state transition is either described by `Next` or the values of `n` and `m` are unchanged. `Spec` sets up the state space of the specfication, which can be thought of as a (possibly infinite) directed graph of states, consisting of variable-value pairs, reachable from the initial state.

For the detailed grammar of TLA$^+$, see Lamport's Specifying Systems book [16].

2.2 PlusCal algorithm language

The PlusCal algorithm language is a pseudocode-like language whose goal is to lower the learning curve for using the TLA$^+$ tool box. Figure 2.2 shows a PlusCal specification for Euler's greatest common denominator algorithm. As seen in Figure 2.2, PlusCal's syntax resembles those of imperative programming languages.

```
1  ----- MODULE GCDPlusCal -----
2  EXTENDS Integers
3
4  CONSTANTS N, M
5
6  (*
7  --algorithm GCDPlusCal {
8    variables n = N, m = M;
9    {
10     l:
11       while (TRUE) {
12         if (n < m) {
13           m := m - n;
14         } else if (m < n) {
15           n := n - m;
16         }
17       }
18   }
19 }
20 *)
21 ============================
```

Figure 2.2: Euler's greatest common denominator algorithm in PlusCal

A PlusCal specification is written in an `algorithm` block inside a TLA$^+$ comment, denoted using `(* ... *)`. All PlusCal statements beside declarations must be part of a label. In Figure 2.2, the specification consists of a single infinite `while` loop inside the label `l`. Statements within one label are executed atomically, i.e., either they all succeed or the label is not executed at all.

PlusCal has support for familiar constructs such as `if` statements, `while` statements, local binding via `with` blocks, as well as procedures and macros. Procedures in PlusCal can take a number of arguments, can have their own local variables, and can modify global variables, but have no return value. Macros in PlusCal have the same textual expansion behavior as C pre-processing macros.

To use a PlusCal specification for model checking, the programmer must compile it to TLA$^+$ using the TLA$^+$ tool box. Figure 2.3 shows the output of compiling the PlusCal specification in Figure 2.2 to TLA$^+$. Except for the slight syntactic difference, the compiled output in Figure 2.3 is equivalent to the specification in Figure 2.1.

For the detailed grammar of PlusCal, see Lamport's manual[1].

[1] https://lamport.azurewebsites.net/tla/c-manual.pdf

```
1  VARIABLES n, m
2
3  vars == << n, m >>
4
5  Init == (* Global variables *)
6          /\ n = N
7          /\ m = M
8
9  Next == IF n < m
10             THEN /\ m' = m - n
11                  /\ n' = n
12             ELSE /\ IF m < n
13                        THEN /\ n' = n - m
14                        ELSE /\ TRUE
15                             /\ n' = n
16                  /\ m' = m
17
18 Spec == Init /\ [][Next]_vars
```

Figure 2.3: Compiled TLA$^+$ output

Chapter 3

Modular PlusCal

This chapter describes Modular PlusCal, an extension of PlusCal, which provides modularity and isolation between system specification and environment specification.

3.1 Motivation

Consider a simple specification for a server and client in Figure 3.1. The specification describes simple interactions between an add server and a client, over a TCP connection. The server waits for the client to send two numbers over the network connection and sends back the sum of those two numbers to the client. The client, on the other hand, sends two numbers to the server, gets the result from the server, and prints the result.

There are two problems with the specification. Firstly, there is code duplication. The code blocks labeled `readA`, `readB`, and `readResult` differ only in the variable being assigned to, `a`, `b`, and `result`, respectively. Similarly, the code blocks labeled `writeResult`, `writeA`, and `writeB` share a lot in common. Secondly, there is no clear separation between system functionality specification and environment specification. The main functionality of the system, namely adding numbers on the server and printing results on the client, is mixed with the specification of the TCP connection between the client and the server, which is part of the environment.

```
1  ------------------ MODULE AddServerClientPlusCal ------------------
2  EXTENDS Integers, Sequences, TLC
3
4  CONSTANTS A, B, BufferSize, ClientId, ServerId
5
6  (*
7  --algorithm AddServerClientPlusCal {
8    variables network = [i \in {ServerId, ClientId} |-> <<>>];
9
10   process (S = ServerId)
11   variables a, b;
12   {
13     ls:
14       while (TRUE) {
15         readA:
16           await Len(network[self]) > 0;
17           a := Head(network[self]);
18           network[ServerId] := Tail(network[self]);
19         readB:
20           await Len(network[self]) > 0;
21           b := Head(network[self]);
22           network[ServerId] := Tail(network[self]);
23         writeResult:
24           await Len(network[ClientId]) < BufferSize;
25           network[ClientId] := Append(network[ClientId], a + b);
26       }
27   }
28
29   process (C = ClientId)
30   variables result;
31   {
32     lc:
33       while (TRUE) {
34         writeA:
35           await Len(network[ServerId]) < BufferSize;
36           network[ServerId] := Append(network[ServerId], A);
37         writeB:
38           await Len(network[ServerId]) < BufferSize;
39           network[ServerId] := Append(network[ServerId], B);
40         readResult:
41           await Len(network[self]) > 0;
42           result := Head(network[self]);
43           network[self] := Tail(network[self]);
44         printResult:
45           print result;
46       }
47   }
48 }
49 *)
50 ====================================================================
```

Figure 3.1: Add server and client specification in PlusCal

```
1  \* global variables
2  variables network = [i \in {ServerId, ClientId} |-> <<>>],
3            serverPkt, clientPkt;
4
5  procedure TCPRead(which)
6  {
7    lr:
8      await Len(network[which]) > 0;
9      if (which = ServerId) {
10       \* TCP read for server
11       serverPkt := Head(network[which]);
12     } else {
13       \* TCP read for client
14       clientPkt := Head(network[which]);
15     }
16     network[which] := Tail(network[which]);
17 }
18
19 procedure TCPWrite(which, pkt)
20 {
21   lw:
22     await Len(network[which]) < BufferSize;
23     network[which] := Append(network[which], pkt);
24 }
```

Figure 3.2: PlusCal procedures specifying a TCP connection

The two PlusCal syntactical constructs that are offered as solutions for reuse are procedures and macros. However, they cannot effectively address the above problems.

PlusCal procedures can modify global variables, so they can modify the `network` global variable but cannot return the packet read from the network without introducing a new global variable to store it (as shown in Figure 3.2). To use the procedure `TCPRead`, the programmer has to introduce two new global variables, namely `serverPkt` and `clientPkt`. Furthermore, the definition of `TCPRead` contains the knowledge that there are only two processes interacting with each other. This is not desirable because it reduces reusability of the specification as well as the compiled output of PGo. For example, this procedure cannot be used in a specification where there are more than two kinds of processes interacting with each other.

PlusCal macros, on the other hand, do not require new global variables, but have implicit behaviors, which are non-obvious (as shown in Figure 3.3). For example, after the macro call `TCPRead(ServerId, a)`, the value read from the TCP connection is stored in `a`, but this fact is hidden in the `TCPRead` definition. Both the

```
1  macro TCPRead(which, pkt)
2  {
3    await Len(network[which]) > 0;
4    pkt := Head(network[which]);
5    network[which] := Tail(network[which]);
6  }
7
8  macro TCPWrite(which, pkt)
9  {
10   await Len(network[which]) < BufferSize;
11   network[which] := Append(network[which], pkt);
12 }
```

Figure 3.3: PlusCal macros specifying a TCP connection

programmer and the compiler have to tease this fact out from its definition, which significantly hinders comprehension and compilation.

To cleanly address the problems of reuse and modularity, this thesis introduces Modular PlusCal, which extends PlusCal by adding archetypes (Section 3.2), instances (Section 3.4), and mapping macros (Section 3.3).

3.2 Archetypes

A PlusCal process is a unit of execution. Each process can define local variables, and can make modifications to global variables.

A Modular PlusCal *archetype* is a blueprint for a process, or a group of processes. Unlike processes, they can only access global variables which are passed in as arguments. In addition, they can only interact with their arguments through a well-defined interface. Archetypes are used for specifying system behaviors. The restrictions imposed on archetypes provide the desired isolation between system specification and environment specification. Figure 3.4 shows the grammar for archetypes.

Besides the inability to access global variables, archetypes have the same semantics as processes. They can declare their own local variables, which behave identically to local variables in PlusCal processes. Labeled blocks in their bodies are also executed atomically. They have the same labeling restrictions (see Section 4.1.4) in their bodies.

Figure 3.5 shows the specification of the add server as an archetype in Modular

⟨archetype⟩ ⊨ `archetype` ⟨identifier⟩(⟨parameter-list⟩)⟨local-variables⟩⟨body⟩
⟨parameter-list⟩ ⊨ ε | ⟨parameter⟩ | ⟨parameter⟩`,`⟨parameter-list⟩
⟨parameter⟩ ⊨ ⟨identifier⟩ | `ref` ⟨identifer⟩
⟨local-variables⟩ ⊨ ε | `variables` ⟨declaration-list⟩`;`
⟨declaration-list⟩ ⊨ ⟨variable-declaration⟩ | ⟨variable-declaration⟩`,`⟨declaration-list⟩
⟨body⟩ ⊨ {⟨labeled-statement-list⟩}

Figure 3.4: Grammar for archetypes

```
1   archetype AddServer(ref network)
2   variables a, b;
3   {
4     ls:
5       while (TRUE) {
6         readA:
7           a := network[self];
8         readB:
9           b := network[self];
10        writeResult:
11          network[ClientId] := a + b;
12      }
13  }
```

Figure 3.5: Specification for add server in Modular PlusCal

PlusCal. The `ref` keyword in `ref network` means that `network` may be modified in the archetype body, i.e., it may be assigned to in the body. It may seem that `a` is always assigned to the value of `network[self]`, as shown on line 7; however, `a` is actually assigned to whatever value is pulled out of the environment modeled by the archetype argument `network`. In this case, `a` is a number read from a network connection. Similarly, line 11 actually specifies that the sum `a + b` is sent to the client via the network.

3.3 Mapping macros

Mapping macros complement of archetypes by specifying the environment. Mapping macros define both the interface, which consists of reading and writing, and the model checking behaviors used to model the environment. A mapping macro

⟨mapping-macro⟩ ⊨ `mapping macro` ⟨identifier⟩{`read` ⟨macro-body⟩ `write` ⟨macro-body⟩}
⟨macro-body⟩ ⊨ {⟨statement-list⟩}

Figure 3.6: Grammar for mapping macros

has two parts, one for reading (captured in the read block), and the other for writing (captured in the write block). Just like the body of a PlusCal macro, the read and write blocks of a mapping macro cannot contain any labels. In other words, the statements in each block can only be a part of an atomic step in a specification. The programmer must keep this in mind to write code that accurately models the environment. Specifically, each block should not perform too much work. Figure 3.6 shows the grammar for mapping macros.

The read block of a mapping macro specifies how to pull a value out of the environment. Inside a read block of a mapping macro, the programmer can use the `$variable` special variable to refer to the global variable being used to store state for modeling a part of the environment. The write block of a mapping macro specifies how to incorporate some value into the global environment. Inside a write block, in addition to the `$variable` special variable, the programmer also have access to the `$value` special variable to refer to the value being incorporated into the environment's state. Finally, in both read and write blocks, the programmer uses a `yield` statement to specify either what value is pulled out of the environment (for a read operation) or to what value the environment modeling state is updated (for a write operation).

Figure 3.7 shows how a TCP connection can be modeled with a mapping macro. In this case, the global variable used to model a TCP connection should be initialized as a TLA$^+$ sequence. The read block specifies that a read operation performed on a TCP connection blocks until a packet has arrived (using the PlusCal `await` statement). A packet is then extracted from the connection and returned to the calling code using the `with` statement and the `yield` statement, respectively. The write block specifies that a write operation blocks until there is enough buffer to hold the packet. The packet is then sent by `Append`ing it to the connection.

```
1    mapping macro TCPConnection {
2      read {
3        await Len($variable) > 0;
4        with (msg = Head($variable)) {
5          $variable := Tail($variable);
6          yield msg;
7        }
8      }
9      write {
10       await Len($variable) < BufferSize;
11       yield Append($variable, $value);
12     }
13   }
```

Figure 3.7: Specification for a TCP connection in Modular PlusCal

⟨instance⟩ ⊨ `process` (⟨variable-declaration⟩) == ⟨archetype-instantiation⟩
⟨archetype-instantiation⟩ ⊨ `instance` ⟨identifier⟩ (⟨argument-list⟩) ⟨mapping-clause-list⟩
⟨argument-list⟩ ⊨ ε | ⟨argument⟩ | ⟨argument⟩`,`⟨argument-list⟩
⟨argument⟩ ⊨ ⟨expression⟩ | `ref` ⟨identifier⟩
⟨mapping-clause-list⟩ ⊨ ε | ⟨mapping-clause⟩ | ⟨mapping-clause⟩⟨mapping-clause-list⟩
⟨mapping-clause⟩ ⊨ `mapping` ⟨mapped-variable⟩ `via` ⟨identifier⟩
⟨mapped-variable⟩ ⊨ `@`⟨integer⟩ | ⟨identifier⟩

Figure 3.8: Grammar for instances

3.4 Instances

Since the system specification is modeled using archetypes and the environment specification is modeled using mapping macros, the programmer can develop them independently, allowing for higher level of reuse. For verification purposes, the system specification has to be composed with the environment specification. *Instances* are the compositional glue relating the two halves. Figure 3.8 shows the grammar for instances.

An instance statement instantiates a process or a group of processes using the specified archetype. Any `ref` parameter must be instantiated with a `ref` global variable, indicating that the global variable may be modified in the body of the archetype with an assignment, or an expression, indicating that the argument is

```
1    process (S = ServerId) == instance AddServer(ref conns)
2      mapping conns[_] via TCPConnection;
3
4    process (C = ClientId) == instance AddClient(ref conns)
5      mapping conns[_] via TCPConnection;
```

Figure 3.9: Examples of Modular PlusCal instances

local to the archetype. All non-`ref` parameters must not be instantiated with a `ref` global variable. Each argument may also be mapped by a mapping macro, indicating that any read and write operation to the argument must be expanded with the corresponding read or write block of the mapping macro.

In Figure 3.9, process `S` is instantiated by the `AddServer` archetype, while process `C` is instanticated by the `AddClient` archetype. Their `network` parameters are instantiated with the global variable `conns`, which is also macro-mapped via the `TCPConnection` mapping macro. The `[_]` notation following `conns` indicates that read and write operations to elements of `conns` are expanded with the corresponding read and write blocks of the `TCPConnection` mapping macro; and `conns` is said to be *function-mapped*. A mapping clause without a `[_]` notation is called *variable-mapped*; and read and write operations are expanded using the variable directly.

Figure 3.10 shows a specification of an add server and client in Modular PlusCal which is equivalent to the PlusCal specification in Figure 3.1. Having written the specification this way, the programmer can easily swap the `TCPConnection` mapping macro for a different `UDPConnection` mapping macro. Furthermore, `TCPConnection` can be reused in other specifications to model the network.

```
1  ---------- MODULE AddServerClientModularPlusCal ----------
2  EXTENDS Integers, Sequences, TLC
3
4  CONSTANTS A, B, BufferSize, ClientId, ServerId
5
6  (*
7  --mpcal AddServerClientModularPlusCal {
8    variables conns = [i \in {ServerId, ClientId} |-> <<>>];
9
10   mapping macro TCPConnection {
11     read {
12       await Len($variable) > 0;
13       with (msg = Head($variable)) {
14         $variable := Tail($variable);
15         yield msg;
16       }
17     }
18     write {
19       await Len($variable) < BufferSize;
20       yield Append($variable, $value);
21     }
22   }
23
24   archetype AddServer(ref network)
25   variables a, b;
26   {
27     ls:
28       while (TRUE) {
29         readA:
30           a := network[self];
31         readB:
32           b := network[self];
33         writeResult:
34           network[ClientId] := a + b;
35       }
36   }
37
38   archetype AddClient(ref network)
39   variables result;
40   {
41     lc:
42       while (TRUE) {
43         writeA:
44           network[ServerId] := A;
45         writeB:
46           network[ServerId] := B;
47         readResult:
48           result := network[self];
49         printResult:
50           print result;
51       }
52   }
53
54   process (S = ServerId) == instance AddServer(ref conns)
55     mapping conns[_] via TCPConnection;
56
57   process (C = ClientId) == instance AddClient(ref conns)
58     mapping conns[_] via TCPConnection;
59 }
60 *)
61 ===============================================================
```


Figure 3.10: Add server and client in Modular PlusCal

Chapter 4

Compilation with PGo

In this chapter, we explain the internals of PGo, primarily to document the current state and design of the compiler.

There are three pipelines in PGo's architecture, namely Modular PlusCal to PlusCal (Figure 4.2), PlusCal to Go (Figure 4.13), and Modular PlusCal to Go. This thesis describes the first two pipelines.

Since compilation from Modular PlusCal to PlusCal expands all accesses of archetype parameters using mapping macros, the clear boundary between system specification and environment specification is erased. Therefore, it is not appropriate to compile a Modular PlusCal to PlusCal, and then compiling the resulting PlusCal to Go. Instead, PGo compiles Modular PlusCal specifications directly to Go. However, the compilation from Modular PlusCal to Go is not described in this thesis.

Figure 4.1 shows which stages are shared between the Modular PlusCal to Go and PlusCal to Go pipelines, from parsing to type inference. In addition, configuration parsing and validation are also shared. Only atomicity inference and code generation are distinct for each pipeline.

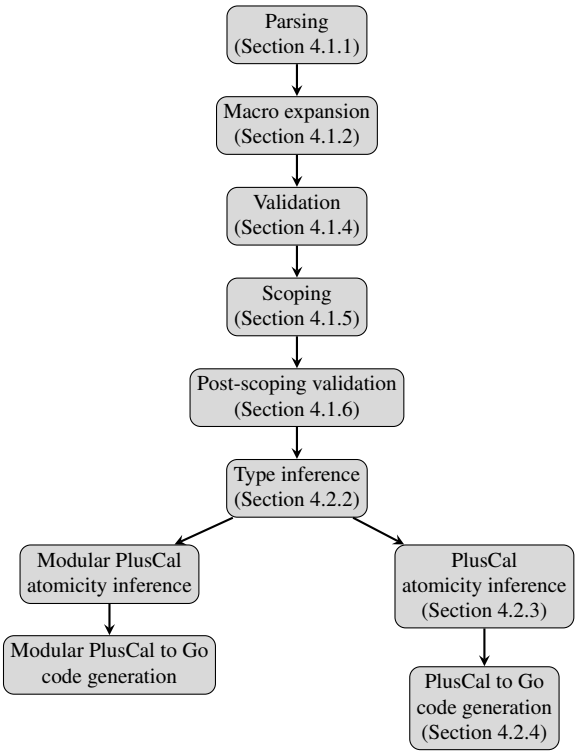

Figure 4.1: Sharing between Modular PlusCal to Go and PlusCal to Go pipelines in PGo

PGo relies on the TLA$^+$ tool box for specification verification. Since TLC only accepts TLA$^+$ specifications as input, a Modular PlusCal specification has to be compiled to TLA$^+$ for verification. PGo accomplishes this by compiling the Modular PlusCal specification to PlusCal, and then relies on the TLA$^+$ tool box for compilation from the resulting PlusCal to TLA$^+$. This pipeline is described next.

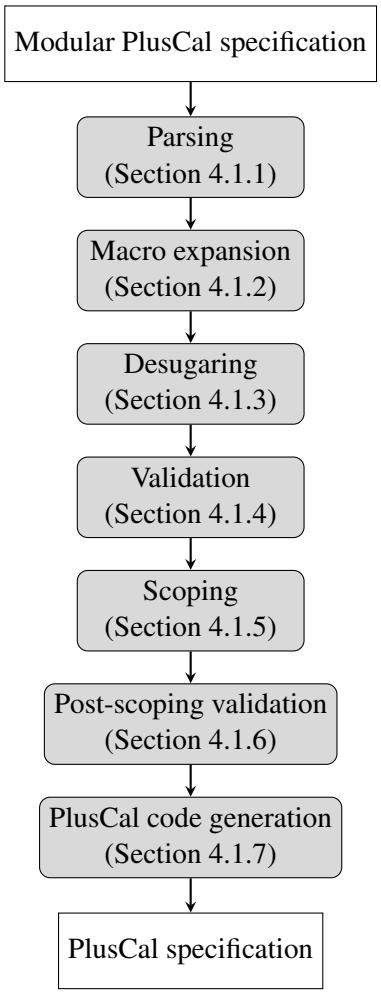

Figure 4.2: PGo compiler pipeline from Modular PlusCal to PlusCal

4.1 The Modular PlusCal to PlusCal pipeline

4.1.1 Parsing

PGo has a rich library for constructing and consuming grammars. It is developed by necessity due to the complexity of the TLA$^+$ and PlusCal grammars: operators have precedence ranges and PlusCal has C and P-syntaxes. Since PGo's `Grammar` objects are constructed in ordinary Java code, complicated constructs such

$$
\begin{array}{rcl}
\langle \text{expression} \rangle & \models & \langle \text{factor} \rangle \mid \langle \text{expression} \rangle + \langle \text{factor} \rangle \mid \langle \text{expression} \rangle - \langle \text{factor} \rangle \\
\langle \text{factor} \rangle & \models & \langle \text{unit} \rangle \mid \langle \text{factor} \rangle * \langle \text{unit} \rangle \mid \langle \text{factor} \rangle / \langle \text{unit} \rangle \\
\langle \text{unit} \rangle & \models & \langle \text{number} \rangle \mid (\langle \text{expression} \rangle)
\end{array}
$$

Figure 4.3: Grammar for simple arithmetic expressions

as precedence ranges for TLA$^+$ operators can be handled outside of the grammar algorithmically in plain Java, greatly reducing grammar development effort.

Grammar construction

PGo's parser facility provides an application programming interface (API) for piecemeal construction of grammars. The API supports string matching, regular expression pattern matching, alternative choice, and mapping. To showcase how to write down a grammar using this API, we will implement a simple arithmetic expression grammar whose grammar is shown in Figure 4.3. Note that this grammar is used only for illustration and is not part of the TLA$^+$ grammar.

The arithmetic expression grammar in Figure 4.3 supports four operators, namely addition, subtraction, multiplication, and division. Addition and subtraction have the same precedence, which is lower than the precedence of multiplication and division. Multiplication and division has the same precedence. The precedence of operators is encoded into the grammar as the productions of ⟨expression⟩ and ⟨factor⟩, which makes multiplication and division bind more tightly than addition and subtraction. All operators are left associative, which is encoded as left recursion in the grammar.

Figure 4.4 shows the abstract syntax for the arithmetic expressions. The abstract syntax consists of binary operations and numbers, both of which are subclasses of expressions. Since PGo's parser also provides source location information for parsed result, all abstract syntax classes must subclass `SourceLocatable` (line 1).

Figure 4.5 shows how the ⟨unit⟩ production can be implemented using facilities provided by PGo's parsing utility. A unit is either a number or a parenthe-

```
abstract class Expression extends SourceLocatable {}

class BinOp extends Expression {
    enum Operator {
        Add,
        Sub,
        Mul,
        Div,
    }
    final Operator operator;
    final Expression lhs;
    final Expression rhs;
    BinOp(Operator operator, Expression lhs, Expression rhs) {
        this.operator = operator;
        this.lhs = lhs;
        this.rhs = rhs;
    }
}

class Number extends Expression {
    final int value;
    Number(int value) {
        this.value = value;
    }
}
```

Figure 4.4: Abstract syntax for arithmetic expressions

```
static Grammar<Expression> number =
    matchPattern(Pattern.compile("[1-9][0-9]*"))
        .map(m -> new Number(Integer.parseInt(m.getValue().group())));

static ReferenceGrammar<Expression> expression = new ReferenceGrammar<>();

static Grammar<Expression> unit =
    parseOneOf(
        number,
        emptySequence()
            .drop(matchString("("))
            .part(expression)
            .drop(matchString(")"))
            .map(seq -> seq.getValue().getFirst()));
```

Figure 4.5: Implementation for ⟨unit⟩

sized expression, so the grammar for it is constructed with a choice grammar using `parseOneOf()` (line 8). The grammar for numbers is implemented by a regular expression matcher with pattern `[1-9][0-9]*`, which says that a number is a series of digits not starting with 0 (line 2). The matched string is then converted to a `Number` object using the `map()` transformer on the grammar (line 3). In the case of parenthesized expressions, the grammar must match a sequence of tokens, namely the open parenthesis token (`(`), a series of tokens for expressions, and the close parenthesis token (`)`). A sequence grammar starts with an empty sequence (line 10), and then is built upon using various sequence constructing methods such as `part()` (line 12) and `drop()` (lines 10 and 12), each accepting a grammar. A *drop sequence grammar* tells the parser to parse the grammar but drops the result. Any failure while parsing a dropped constituent grammar still results in a parse failure overall. Drop sequence grammars are useful in cases where tokens in the input text determine the structure of the resulting abstract syntax tree, but not the nodes in it, or cases where the token does not have a representation in the abstract syntax. The result of a sequence grammar is a heterogeneous list, where each node has its own type. Hence, a `map()` transformation is needed to extract the parsed expression (line 14). Finally, a reference grammar (line 5) acts as a pre-declaration, which allows for self-referential (e.g., the definitions of ⟨factor⟩ and ⟨expression⟩) or mutually recursive grammars (e.g., the definitions of ⟨unit⟩ and ⟨expression⟩).

Figure 4.6 shows how the ⟨factor⟩ and ⟨expression⟩ productions can be implemented. Since ⟨factor⟩ is self-referential, it is declared as a reference grammar. Since ⟨expression⟩ is both self-referential and mutually recursive (with ⟨unit⟩), it is also declared as a reference grammar. Lines 24 to 26 and 47 to 49 in Figure 4.6 shows how each nodes in the resulting heterogeneous list can be accessed. The nodes are in the reverse order of construction. A `Located` object is a simple stand-in for cases where a temporary result of a constituent part of a grammar is needed. In the case of ⟨factor⟩, the operator may be a multiplication or a division, represented as enum values, which cannot be a subclass of `SourceLocatable` (a Java restriction). Hence, it is saved into a `Located` object.

```
static ReferenceGrammar<Expression> factor = new ReferenceGrammar<>();

static {
    factor.setReferencedGrammar(
        parseOneOf(
            unit,
            emptySequence()
                .part(factor)
                .part(parseOneOf(
                    emptySequence()
                        .part(matchString("*"))
                        .map(seq ->
                            new Located<>(
                                seq.getLocation(),
                                BinOp.Operator.Mul)),
                    emptySequence()
                        .part(matchString("/"))
                        .map(seq ->
                            new Located<>(
                                seq.getLocation(),
                                BinOp.Operator.Div))))
                .part(unit)
                .map(seq -> new BinOp(
                    seq.getValue().getRest().getFirst().getValue(),
                    seq.getValue().getRest().getRest().getFirst(),
                    seq.getValue().getFirst()))));

    expression.setReferencedGrammar(
        parseOneOf(
            factor,
            emptySequence()
                .part(expression)
                .part(parseOneOf(
                    emptySequence()
                        .part(matchString("+"))
                        .map(seq ->
                            new Located<>(
                                seq.getLocation(),
                                BinOp.Operator.Add)),
                    emptySequence()
                        .part(matchString("-"))
                        .map(seq -> new Located<>(
                            seq.getLocation(),
                            BinOp.Operator.Sub))))
                .part(factor)
                .map(seq -> new BinOp(
                    seq.getValue().getRest().getFirst().getValue(),
                    seq.getValue().getRest().getRest().getFirst(),
                    seq.getValue().getFirst()))));
}
```

Figure 4.6: Implementation for ⟨factor⟩ and ⟨expression⟩

PGo's parser

PGo's parser is a recursive descent parser with backtracking and memoization. It consumes `Grammar` objects defined by the programmer, together with a string input, and produces all possible abstract syntax trees resulting from that grammar.

Since TLA$^+$ has many constructs with the same prefix, e.g., a tuple `<<e>>`, and a required action, `<<e1>>_e2`, memoization is heavily employed to keep parsing time manageable. The parser relies on the programmer to annotate which grammar should be memoized via a `memoize()` call, which creates a memoized grammar from any grammar. The memoization key is the identity of the memoized grammar and the state of the parser when it encounters that grammar. Upon the first encounter, the parser proceeds normally with the inner grammar and the result is saved into a memoization table. When there is a parse failure and the memoized grammar is retried, the memoized result is used instead of reparsing.

Internal representation

PGo has separate abstract syntax classes for TLA$^+$, PlusCal, and Modular PlusCal constructs. Each abstract syntax node has its own unique identifier, which is used as keys in constructed tables of information regarding that node. This means that their classes need not be modified to add fields for new pieces of information about the objects, which reduces churn in the code base. The class `DefinitionRegistry` in the code base acts as a centralized database for all tables of information about nodes in the abstract syntax tree.

The design of the abstract syntax classes follows the visitor pattern, which allows addition of operations on the classes without modification to them, further reducing churn.

4.1.2 Macro expansion

Macros in PlusCal and Modular PlusCal are named code fragments. They are expanded structurally during compilation. As such, they do not suffer from pitfalls of textual expansion, such operator precedence problems, which is demonstrated next. A call such as `Macro(c, 1 + 2)`, whose definition is shown in Figure 4.7, is expanded correctly to `c := (1 + 2) * 2` by PGo. This is in contrast to textual

```
1  macro Macro(a, b) {
2    a := b * 2;
3  }
```

Figure 4.7: A simple macro

```
1  lb: while (f(a)) {
2        body
3      };
4      stmt
```

```
1  lb: if (f(a)) {
2        body
3        goto lb;
4      } else {
5        stmt
6      };
```

Figure 4.8: Simple desugaring

macro expansion, where the same call is expanded incorrectly to `c := 1 + 2 * 2`.

There are restrictions on macros such as macros must not be recursive or mutually recursive, and their bodies must not contain any labels.

4.1.3 Desugaring

Since a PlusCal `while` loop must appear at the start of a labeled code block, a `while` loop in Modular PlusCal are desugared into an `if` and a `goto`. This is necessary due to the possibility that the condition of the `while` loop may contain an archetype parameter read, which may be expanded in the code generation stage (Section 4.1.7) to multiple statements. Figure 4.8 shows how this translation is performed in the simplest case. Note that desugaring is only performed in the Modular PlusCal to PlusCal pipeline.

However, blindly inserting a `goto` may result in invalid code since control flow may have terminated before the inserted `goto`. For example, the last statement of a `while` loop may be a `goto` statement. Inserting a new `goto` after that `goto` results in dead code, which is forbidden by the TLA$^+$ tool box. Therefore, desugaring has to insert `goto` strategically based on control flow.

4.1.4 Validation

The validation stage checks the input specification's compliance with labeling rules. Modular PlusCal inherits all labeling rules from PlusCal. These rules dictate where

a label must, must not, and may be present. Below are some example labeling rules.

- The first statement in the body of a process, a procedure, or a uniprocess algorithm must be labeled.
- A `while` statement must be labeled.
- A statement S in a statement sequence must be labeled if it is preceded in that sequence by any of the following:
  - A `call` statement, if S is not a `return` or a `goto`.
  - A `return` statement.
  - A `goto` statement.
  - An `if` or `either` statement that contains a labeled statement, a `goto`, a `call`, or a `return` anywhere within it.
- A `macro` body and a `with` body cannot contain any labeled statements.
- In any control path, a label must come between an assignment to a variable x and any other statement that assigns a value to x. A local variable or parameter of a procedure P is set by a `call P(...)` or `return` statement in P.

The above rules are taken from section 3.7 of the PlusCal's manual[1].

4.1.5 Scoping

Modular PlusCal, and PlusCal have static scoping, i.e., identifiers must be declared before use. TLA$^+$, on the other hand, does not allow redefinition of an identifier, so the problem of scoping becomes moot for the language.

The scoping stage builds a table matching uses of an identifier to its declaration. It does so using one `ChainMap` per lexical scope. A `ChainMap` has mappings from variable names to variable declarations, and a parent map. To find a variable declaration, it first looks up the variable name in its own mappings. If the name is not found, the `ChainMap` delegates the lookup to its parent map.

[1] https://lamport.azurewebsites.net/tla/c-manual.pdf

In addition, the scoping stage also loads additional modules as instructed by TLA$^+$ `EXTENDS` and `INSTANCE` statements in the specification. An `EXTENDS` statement pulls all definitions in a TLA$^+$ module into scope. An `INSTANCE` statement pulls all definitions in a TLA$^+$ module into scope, and also replaces some constant definitions with user-provided values.

Beside normal variables, which are declared before use, there are some special variables which are implicitly declared. For example, each process and archetype body has an implicit `self` variable, which refers to the process identifier of the process or archetype, is immutable, and is in scope for the whole body. A procedure also has an implicit `self` variable which refers to the calling process or archetype's identifier; however, PGo does not support this at the moment but only supports procedures which do not use `self`. Other special variables are `$variable` and `$value`. The `$variable` special variable is available in both read and write blocks of a mapping macro and it refers to the name of the variable being mapped (see Section 4.1.7). The `$value` special variable is available only in the write block of a mapping macro and it refers to the value being assigned to the mapped variable (see Section 4.1.7).

4.1.6 Post-scoping validation

In addition to the labeling rules, Modular PlusCal also imposes some restrictions which are checked in the post-scoping validation stage. Below are the restrictions checked in this stage.

- Only `ref` parameters or local variables can be assigned to inside an archetype body.
- Parameters which are function-mapped can only be used as functions (as opposed to being used as variables).
- Parameters which are variable-mapped can only be used as variables (as opposed to being used as functions).
- A variable can only be mapped once in a Modular PlusCal instance statement.

- Parameters must be mapped consistently across different instances of the same archetype, i.e., a parameter of an archetype cannot be function-mapped in one instance but variable-mapped in another instance.

These restrictions ensure that the Modular PlusCal to Go code generation stage (not described in this thesis) can generate meaningful code.

4.1.7 PlusCal code generation

A Modular PlusCal specification is compiled into a PlusCal specification by removing the mapping macros, the archetypes, and compiling instances into processes. An instance is compiled into a process, or a group of processes, by expanding reads and writes to archetype parameters with their corresponding mapping macro read and write blocks. In other words, archetype definitions are inlined and mapping macros are expanded for instances.

For example, consider the specification in Figure 4.9. The archetype `A` writes `0` to the network (line 29), modeled as `conn`, reads from the network into a local variable `r` (line 30), and if `r` is positive (line 31), writes `r + 1` into its database at key `"k"` (line 32), and finally, prints out the value associated with the key `"k"` in its database (line 34). It does all of the above in one atomic step labeled `l`. A single process `P` (line 39), assigned an identifier of `0` (which is the value of `self` within its body), is instantiated from archetype `A`, with a TCP connection (lines 39 and 40), and a local database initialized to have the key `"k"` maps to `0` (lines 39 and 41). Figure 4.10 shows the compiled PlusCal output for this specification with added comments for clarity.

As seen in Figure 4.10, read and write expansions use temporary variables. Any write to `network` is saved into a temporary variable (e.g., `connWrite` on line 11 and `connWrite0` on line 16) because PlusCal only permits one modification per variable in a label. Any read from `network` is also saved into a temporary variable (e.g., `connRead` on line 17) to maintain a small code size. If there are multiple labels, the temporary variables are reused in each label.

```
--mpcal Spec {
  mapping macro TCPConnection {
    read {
      await Len($variable) > 0;
      with (msg = Head($variable)) {
        $variable := Tail($variable);
        yield msg;
      }
    }
    write {
      await Len($variable) < BufferSize;
      yield Append($variable, $value);
    }
  }

  mapping macro DB {
    read {
      yield $variable;
    }
    write {
      yield $value;
    }
  }

  archetype A(ref conn, ref db)
  variables r;
  {
    l:
      conn := 0;
      r := conn;
      if (r > 0) {
        db["k"] := r + 1;
      };
      print db["k"];
  }

  variables network = <<>>;

  process (P = 0) == instance A(ref network, [k \in {"k"} |-> 0])
    mapping network via TCPConnection
    mapping @2[_] via DB;
}
```

Figure 4.9: A simple Modular PlusCal specification

```
--algorithm Spec {
  variables network = <<>>,
            \* temporaries
            connWrite, connRead, connWrite0, dbWrite, dbWrite0, dbRead;
  process (P = 0)
  variables dbLocal = [k \in {"k"} |-> 0], r;
  {
    l:
      \* expanded write to conn
      await (Len(network)) < (BufferSize);
      connWrite := Append(network, 0);

      \* expanded read from conn
      await (Len(connWrite)) > (0);
      with (msg0 = Head(connWrite)) {
        connWrite0 := Tail(connWrite);
        connRead := msg0;
      };
      r := connRead;

      if (r > 0) {
        \* expanded write to db
        dbWrite := [dbLocal EXCEPT !["k"] = (r) + (1)];

        \* join write
        dbWrite0 := dbWrite;
      } else {
        \* join write
        dbWrite0 := dbLocal;
      };
      \* expanded read from db["k"]
      dbRead := dbWrite0["k"];

      \* print uses temporary
      print dbRead;

      \* write-backs
      network := connWrite0;
      dbLocal := dbWrite0;
  }
}
```

Figure 4.10: Compiled PlusCal output

Write-backs

Modular PlusCal, just like PlusCal, allows the programmer to put multiple reads and a single write to the same archetype parameter into a labeled atomic step. However, both read and write blocks of a mapping macro may modify the underlying variable (e.g., `TCPConnection` mapping macro in Figure 4.9). Therefore, PGo must generate temporary write variables to save intermediary modifications to the underlying variable. The last temporary write variable is written to the underlying variable at the end of the label.

Join writes

When control flow forks into multiple execution paths (e.g., `if` and `either` statements), each path may write a different number of times to an underlying variable. This is a problem because when the execution paths merge back after the statement, there may not be a single temporary variable that can be used for further modification. For example, in Figure 4.9, `db` is only modified when `r` is positive (line 32). Therefore, join writes are inserted (lines 26 and 28 of Figure 4.10) so that there is a single valid temporary variable to be used after the statement, regardless of which execution path was taken.

Temporary variables as global variables

Temporary variables capture unnecessary intermediary states. The intermediary states are distinct from each other due to differences in the values of the temporary variables. This leads to an exponential increase in the number of states that TLC has to check.

Originally, temporary variables are declared as local variables in compiled PlusCal processes. This further exacerbates the problem of state explosion due to how local variables in processes are compiled to TLA^+. Local variables in PlusCal processes are compiled as TLA^+ global variables whose values are functions mapping process identifiers to the local value of that variable in TLA^+.

To illustrate the problem, consider two example PlusCal outputs of the Modular PlusCal to PlusCal pipeline, shown in Figure 4.11. In Figure 4.11, `temp` serves as a PGo generated temporary variable for `var`. The only difference between the two

```
1  --algorithm Spec {
2    variables var = [
3      i \in {1, 2} |-> 0
4    ];
5
6    process (P \in {1, 2})
7    variables temp;
8    {
9      lb:
10       temp := [
11         var EXCEPT ![self] = self
12       ];
13       var := temp;
14   }
15 }
```

```
1  --algorithm Spec {
2    variables
3      temp,
4      var = [
5        i \in {1, 2} |-> 0
6      ];
7
8    process (P \in {1, 2}) {
9      lb:
10       temp := [
11         var EXCEPT ![self] = self
12       ];
13       var := temp;
14   }
15 }
```

Figure 4.11: Example local and global temporary variable outputs
Left: Local temporary variable PlusCal output.
Right: Global temporary variable PlusCal output.

specifications is whether `temp` is a local variable or a global variable.

Local variables in PlusCal processes are compiled as TLA$^+$ global functions mapping process identifiers to the local value of that variable in TLA$^+$ due to the possibility that the programmer may want to write properties and invariants involving local state of processes. Thus, lines 10 to 12 of the local temporary variable output in Figure 4.11 is actually compiled to the following nested function substitution [2] in TLA$^+$.

```
temp' = [temp EXCEPT ![self] = [var EXCEPT ![self] = self]]
```

In other words, `temp` is actually a nested function. Furthermore, process 1 can only ever read from or write to `temp[1]` in the compiled TLA$^+$. Similarly, process 2 can only read from and write to `temp[2]` in the compiled TLA$^+$.

The state spaces for the two specifications in Figure 4.11 are shown in Figure 4.12. Below are the states of the local temporary variable specification. Note that `lb` is an atomic block for both processes 1 and 2.

[2] A *function substitution* `[func EXCEPT ![x] = y]` results in a new function with the same content as `func`, except that key `x` is mapped to value `y` in the new function.

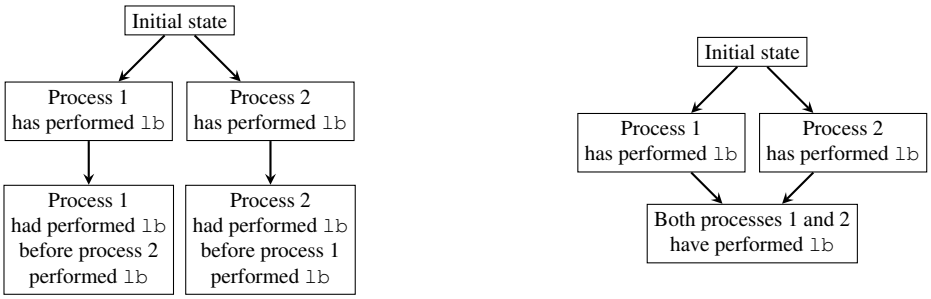

Figure 4.12: State spaces of specifications in Figure 4.11
Left: State space for local temporary variable specification.
Right: State space for global temporary variable specification.

- Initial state

```
temp = [1 |-> defaultInitValue, 2 |-> defaultInitValue]
var = [1 |-> 0, 2 |-> 0]
```

- Process 1 has performed `lb`

```
temp = [1 |-> [1 |-> 1, 2 |-> 0], 2 |-> defaultInitValue]
var = [1 |-> 1, 2 |-> 0]
```

- Process 1 had performed `lb` before process 2 performed `lb`

```
temp = [1 |-> [1 |-> 1, 2 |-> 0], 2 |-> [1 |-> 1, 2 |-> 2]]
var = [1 |-> 1, 2 |-> 2]
```

- Process 2 has performed `lb`

```
temp = [1 |-> defaultInitValue, 2 |-> [1 |-> 0, 2 |-> 2]]
var = [1 |-> 0, 2 |-> 2]
```

- Process 2 had performed `lb` before process 1 performed `lb`

```
temp = [1 |-> [1 |-> 1, 2 |-> 2], 2 |-> [1 |-> 0, 2 |-> 2]]
var = [1 |-> 1, 2 |-> 2]
```

Contrast the state space of the global temporary variable specification and that of the local temporary variable specification in Figure 4.12. The state space now has only four states, which are listed below.

- Initial state

```
temp = defaultInitValue
var = [1 |-> 0, 2 |-> 0]
```

- Process 1 has performed `lb`

```
temp = [1 |-> 1, 2 |-> 0]
var = [1 |-> 1, 2 |-> 0]
```

- Process 2 has performed `lb`

```
temp = [1 |-> 0, 2 |-> 2]
var = [1 |-> 0, 2 |-> 2]
```

- Process 2 had performed `lb` before process 1 performed `lb`

```
temp = [1 |-> 1, 2 |-> 2]
var = [1 |-> 1, 2 |-> 2]
```

As discussed above, promoting temporary variables to global variable reduces state explosion.

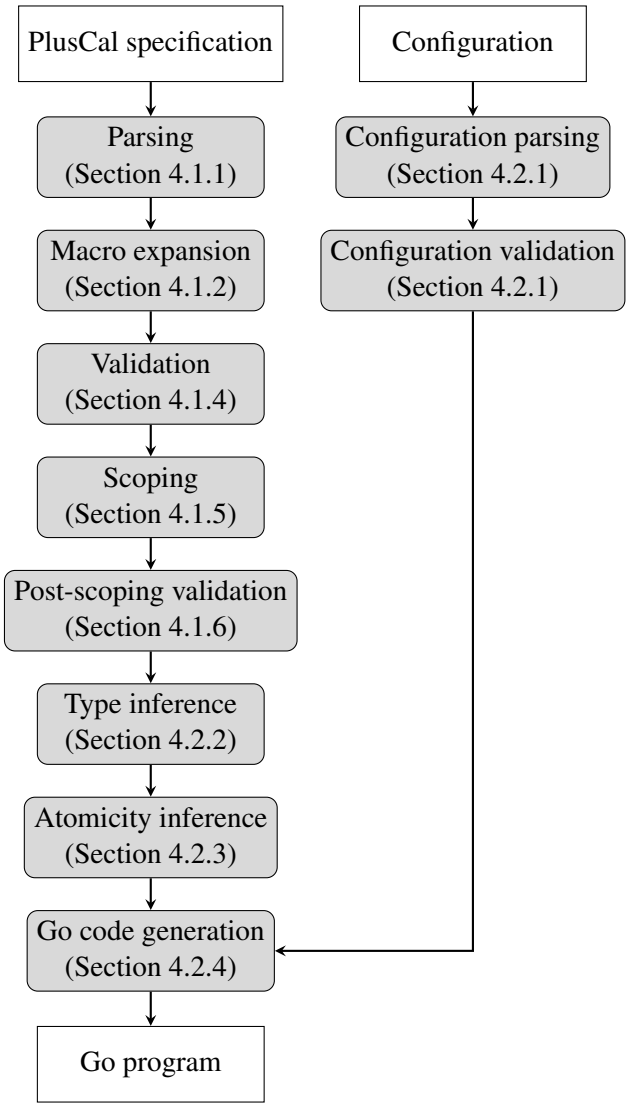

Figure 4.13: PGo compiler pipeline from PlusCal to Go

4.2 The PlusCal to Go pipeline

The PlusCal to Go pipeline takes a PlusCal specification, as well as a configuration file, as input and produces a Go program as output (see Figure 4.13).

```
Usage: pgo [options] spec
  --version=<boolean>          - Version [default false]
  -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]
  -m --mpcalCompile=<boolean>  - Compile a Modular PlusCal spec to vanilla PlusCal
                                 [default false]
  -c --configFilePath=<string> - path to the configuration file, if any
```

Figure 4.14: PGo's accepted command line arguments

```
{
  "build": {
    "output_dir": "/path/to/output/dir",
    "dest_file": "file.go",
    "dest_package": "package_name"
  },
  "networking": {
    "enabled": false,
    "state": {
      "strategy": "state-server",
      "endpoints": [],
      "peers": [],
      "timeout": 3
    }
  },
  "constants": {
    "CONST": "\"val\""
  }
}
```

Figure 4.15: Configuration file for PGo

4.2.1 Configuration parsing

The configuration parsing stage parses command line arguments and the configuration file. Figure 4.14 shows the command line arguments accepted by PGo. If the programmer turns on compilation from Modular PlusCal to PlusCal, the path to the configuration file is not required. The configuration file is expected to be a valid JSON object with attributes shown in Figure 4.15. We now overview each of the configuration parameters, starting with *build*.

Build

The `build` object is required. The `output_dir` attribute is required and must be an existing path. The programmer must provide either `dest_file` or `dest_package`

attribute; if both are provided, `dest_package` takes precedence. The `dest_file` attributes specifies the name of the output file. The `dest_package` attribute specifies the name of the output Go package.

Networking

The `networking` object is optional. If it is not specified, PGo produces a multi-threaded Go program as output. The optional `enabled` attribute specifies whether networking is enabled. The `state` attribute specifies the configuration for the networked state strategy. If the `state` object is present, `enabled` is overridden to be true. The `strategy` attribute specifies which state strategy to use. PGo currently supports `state-server` and `etcd`, with `state-server` being the default. The `endpoints` attribute specifies the etcd endpoints to connect to. The `endpoints` attribute is required if `strategy` is `etcd`. The `peers` attribute specifies the peer endpoints to connect to. The optional `timeout` attribute specifies the network timeout in seconds, which defaults to three seconds.

Constants

The `constants` object is optional. It contains key-value pairs whose key is the name of the constant, encoded as a JSON string, and whose value is a TLA$^+$ expression, also encoded as a JSON string.

4.2.2 Type inference

Since Go, the target language, is a statically typed language, PGo introduces a type system for effective compilation. The type inference stage is the last stage shared between the Modular PlusCal to Go and the PlusCal to Go pipelines.

Types

PGo's types are defined inductively, as shown in Figure 4.16. The $\vec{\tau}$ notation indicates a sequence of types, e.g., $\texttt{Tuple}[\texttt{Bool}, \texttt{Tuple}[\texttt{Int}]]$ is a valid type. The $\overrightarrow{x : \tau}$ indicates a sequence of fields and types, e.g., $\texttt{Record}[dst : \texttt{String}, src : \texttt{String}]$ is a valid type.

$$
\begin{aligned}
\tau ::= & \texttt{Bool} \mid \texttt{Int} \mid \texttt{Real} \mid \texttt{String} \mid \texttt{Interface} \\
& \mid \texttt{Set}[\tau] \mid \texttt{NonEnumerableSet}[\tau] \mid \texttt{Slice}[\tau] \\
& \mid \texttt{Tuple}[\vec{\tau}] \mid \texttt{Map}[\tau_1]\tau_2 \mid \texttt{Record}[\overrightarrow{x : \tau}] \\
& \mid \texttt{Function}(\vec{\tau})\tau_2 \mid \texttt{Procedure}(\vec{\tau}) \\
& \mid \texttt{AR}[\texttt{read}(\tau_1);\texttt{write}(\tau_2)] \mid \texttt{ARC}[\tau_1][\texttt{read}(\tau_2);\texttt{write}(\tau_3)]
\end{aligned}
$$

Figure 4.16: PGo's types

The type `NonEnumerableSet`$[\tau]$ is reserved for sets that are not finitely enumerable, such as built-in constructs like `Nat` and `Integers`, which represent the sets of natural numbers and integers, respectively. An example record type is `Record`$[dst : \texttt{String}, data : \texttt{String}]$, which is inferred for the TLA$^+$ expression `[dst |-> "1.1.1.1", data |-> "payload"]`. PlusCal processes and procedures, as well as Modular PlusCal archetypes and instances, have type `Procedure`$(\vec{\tau})$.

The type `AR`$[\texttt{read}(\tau_1);\texttt{write}(\tau_2)]$ is read as archetype resource with read type τ_1 and write type τ_2. Similarly, the type `ARC`$[\tau_1][\texttt{read}(\tau_2);\texttt{write}(\tau_3)]$ is read as archetype resource collection with key type τ_1, read type τ_2, and write type τ_3. These two types are the types of archetype parameters.

Type constraints

A typing judgement has the form $\Gamma \vdash t : \texttt{T} \mid C$, which is read as "term t has type `T` under assumptions Γ whenever constraints C are satisfied". Below are some example type rules in PGo's type system.

$$
\frac{n \in \{\texttt{FALSE}, \texttt{TRUE}\}}{\Gamma \vdash n : \texttt{Bool} \mid \emptyset}\ \text{CT-Bool} \qquad \frac{n \in \{\ldots, \texttt{-1}, \texttt{0}, \texttt{1}, \ldots\}}{\Gamma \vdash n : \texttt{Int} \mid \emptyset}\ \text{CT-Int} \qquad \frac{\texttt{x} : \texttt{T} \in \Gamma}{\Gamma \vdash \texttt{x} : \texttt{T} \mid \emptyset}\ \text{CT-Var}
$$

$$
\frac{n \in \{\ldots, \texttt{-0.1}, \ldots, \texttt{0}, \ldots, \texttt{0.1}, \ldots\}}{\Gamma \vdash n : \texttt{Real} \mid \emptyset}\ \text{CT-Real} \qquad \frac{n \in \{\texttt{""}, \texttt{"a"}, \ldots\}}{\Gamma \vdash n : \texttt{String} \mid \emptyset}\ \text{CT-String}
$$

$$\frac{\begin{array}{c}\Gamma \vdash x : \mathtt{T}_1 \mid C_1 \qquad \Gamma \vdash y : \mathtt{T}_2 \mid C_2 \\ \mathtt{boolOp} \in \{\wedge, \vee, \Rightarrow, \equiv\} \\ C = C_1 \cup C_2 \cup \{\mathtt{T}_1 = \mathtt{Bool}, \mathtt{T}_2 = \mathtt{Bool}\}\end{array}}{\Gamma \vdash x\ \mathtt{boolOp}\ y : \mathtt{Bool} \mid C}\ \text{CT-BoolOp}$$

$$\frac{\begin{array}{c}\Gamma \vdash x : \mathtt{T}_1 \mid C_1 \qquad \Gamma \vdash y : \mathtt{T}_2 \mid C_2 \\ \mathtt{setMemberOp} \in \{\in, \notin\} \\ C = C_1 \cup C_2 \cup \{\mathtt{T}_2 = \mathtt{Set}[\mathtt{T}_1]\}\end{array}}{\Gamma \vdash x\ \mathtt{setMemberOp}\ y : \mathtt{Bool} \mid C}\ \text{CT-SetMemberOp}$$

The typing rules are implemented in multiple classes in PGo's code base. For example, typing rules for built-in TLA$^+$ operators are implemented in the `TLABuiltins` class.

The most common type constraints in PGo's type system are equality type constraints, as shown in the example typing rules. An *equality type constraint* states that a type $\mathtt{T}_1$ must be structurally equal to another type $\mathtt{T}_2$. In other words, all constituent types within $\mathtt{T}_1$, if there are any, must be structurally equal to constituent types within $\mathtt{T}_2$, recursively.

In addition to type equality constraints, PGo also supports has-field constraints. A *has-field constraint* states that a type `T` must be a `Record` and it must have a field f with type $\mathtt{T}_f$. This type constraint is necessary since definition and usage of records may span multiple processes or archetypes.

Type equality constraints and has-field constraints comprise *basic type constraints*. These basic type constraints are then wrapped in monomorphic type constraints or polymorphic type constraints. A *monomorphic type constraint* is just a simple wrapper for a basic type constraint. A *polymorphic type constraint* wraps multiple basic constraints, which are considered alternatives. Since a collection of type constraints encodes the conjunction of them, a polymorphic type constraint allows encoding of disjunctions of basic constraints.

Type inference algorithm

The type inference stage has two phases: type constraint collection, and type solving. After type constraints are collected, they are solved by the `TypeSolver`. The resulting solution is a mapping between type variables (stand-ins for unknown types during type constraint collection) to concrete types. If a type variable is not mapped

to any concrete type, which happens when there are not enough constraints on it, it is mapped to an `Interface`. PGo's `Interface` type is translated to Go's `interface{}` type in code generation.

The `TypeSolver` works by going through the constraints one by one. If the current constraint is a monomorphic constraint, the solver extracts out the basic constraint. If the current constraint is a polymorphic constraint, the solver sets up a backtracking point by taking a snapshot of its own state, and pushes the snapshot onto a state stack. It then extracts the first basic constraint from the polymorphic constraint, and proceeds to solve it. When there's a need to backtrack, the solver pops a snapshot from the state stack, reloads its state from the snapshot, and extracts the next basic constraint to be solved. If there's no basic constraint left, it returns an error via the issue context.

After extracting a basic type constraint, the `TypeSolver` proceeds to solve it. If the extracted basic constraint is a has-field constraint, the solver looks up the record in its record groups to constrain that record if it's present; otherwise, it adds a new entry for that record. If the extracted basic constraint is not a has-field constraint, it must be a type equality constraint. The solver checks if the left-hand side or the right-hand side of the equality constraint is a type variable. If so, it looks up the type variable in its variable groups. If both are type variables, the two groups to which they belong are merged, and the types to which the two groups map are added as a type equality constraint to be solved. The left hand side and right hand side are then substituted to gain further structural knowledge of their types. The resulting types are checked to see if they are the same type and their constituent types are added as constraints to be solved.

The above process repeats until there are no constraints left, at which point the solver constructs and returns a substitution as the result. In other words, the solver returns the first solution it can find.

Another point of note is that since TLA$^+$ expressions are ambiguous (e.g., `<<1, 2>>` can be a tuple or a sequence initialized with two elements), PGo prefers constraints which give the most amount of information. For example, the TLA$^+$ function call `a[b]` is constrained with $(\texttt{a} : \texttt{Slice}[\texttt{T}] \wedge \texttt{b} : \texttt{Int}) \vee (\texttt{a} : \texttt{Map}[\texttt{T}_1]\texttt{T}_2 \wedge \texttt{b} : \texttt{T}_1) \vee (\texttt{a} : \texttt{Function}(\texttt{T}_1)\texttt{T}_2 \wedge \texttt{b} : \texttt{T}_1)$. The order of basic constraints in the disjunction matters, since `TypeSolver` returns the first found solution. For example, if

selecting `a` : `Slice`[`T`] /\ `b` : `Int` leads to a solution, `TypeSolver` does not consider the other constraints.

4.2.3 Atomicity inference

The atomicity inference stage is the first stage where Modular PlusCal and PlusCal compilation to Go diverge. This thesis describes only the PlusCal to Go pipeline.

PlusCal requires that a statement be part of a label. A block of statements in a label is executed atomically, i.e., they all succeed or are not executed. Therefore, PGo has to determine if a global variable needs synchronization to maintain these semantics in the generated Go code.

Global variables that need synchronization are grouped into collections that are called lock groups. The lock groups are computed by traversing labeled blocks in procedures and processes, putting all global variables accessed within a labeled block into the same group, and merging groups with a common global variable.

4.2.4 Go code generation

PlusCal has many language constructs that are translated to Go in non-trivial ways. These constructs and their translations are described in this section, starting with an overview of supported PlusCal and TLA$^+$ constructs.

PlusCal support

PGo supports both the C and P-syntaxes of PlusCal. PlusCal's P-syntax denotes code blocks with keywords like `begin` and `end`, while its C-syntax denotes code blocks with curly braces. Note that unused labels are removed from the Go output and that fresh variable names and labels are generated to avoid name capture.

PlusCal feature	**PGo support**
Line comment `\* line comment`	Supported; removed from output

PlusCal feature	**PGo support**
Block comment `(* block comment *)`	Supported; removed from output

Labeled statements
`label:`
`  stmt1;`
`  stmt2;`
`  \* ...`
	Compiled with a mutex or a distributed mutex around the statements

Assignment `x := exp;`	Supported; compiled as expected
Multiple variable assignment `x := y || y := x + y;`	Supported; compiled as multiple assignment in Go

While loop
`while (condition) {`
`  body;`
`}`
	Compiled as
	`for {`
	`  if !condition {`
	`  break`
	`  }`
	`  body`
	`}`

PlusCal feature	**PGo support**

If statement
`if (condition) {`
`  thenPart;`
`} else {`
`  elsePart;`
`}`
	Supported; compiled as expected

Return statement `return;`	Supported; compiled as expected
Skip statement `skip;`	Supported; removed from output
Call statement `call proc(arg1, arg2);`	Supported; compiled as expected
Macro call `macro1(arg1, arg2);`	Supported; expanded during compilation

PlusCal feature	**PGo support**

Either statement
`either { stmt1; stmt2; }`
`    or { stmt3; stmt4; }`
`    or { stmt5; stmt6; }`
`    \* ...`
	Compiled as
	`case0:`
	`  stmt1`
	`  stmt2`
	`  goto endEither`
	`case1:`
	`  stmt3`
	`  stmt4`
	`  goto endEither`
	`case2:`
	`  stmt5`
	`  stmt6`
	`  goto endEither`
	`// ...`
	`endEither:`
	Each case is tried deterministically from top to bottom (i.e., `case0` is tried before `case1`, etc.). Case N is tried only after case 0 to N-1 have failed because await conditions in those cases are not met.

With statement
`with (x = exp1,`
`      y \in exp2) {`
`  body;`
`}`
	Supported; compiled as variable assignment with fresh names. In the example code, `y` is assigned the first element of `exp2`.

Print statement `print exp;`	Compiled as `fmt.Printf("%v", exp)`

PlusCal feature	**PGo support**

Assert statement
`assert condition;`
	Compiled as
	`if !condition {`
	`  panic("condition");`
	`}`

Await statement
`await condition;`
	Compiled as
	`awaitLabel:`
	`  if !condition {`
	`  goto awaitLabel`
	`  }`

Goto statement `goto label;`	Supported; compiled as expected

Single process algorithm
`--algorithm Algo {`
`  variables x = exp1,`
`  y \in exp2;`
`  {`
`  body;`
`  }`
`}`
	Supported; compiled as a single-threaded single-process Go program

PlusCal feature	**PGo support**

Multiprocess algorithm
`--algorithm Algo {`
`  variables x = exp1,`
`  y = exp2;`
`  process (P \in exp3)`
`  variables local = exp4;`
`  {`
`  body;`
`  }`
`}`
	Supported; compiled with various strategies configured by the user

Table 4.1: PGo support for PlusCal constructs

TLA$^+$ support

Below are the TLA$^+$ constructs. Note that PGo makes liberal use of temporary variables to compile complex TLA$^+$ constructs.

TLA$^+$ feature	**PGo support**

Function call
`x[exp1]`
`\* or`
`x[exp1, exp2, exp3]`
`\* or`
`x[<<exp1, exp2, exp3>>]`
`\* or`
`x[[field1 |-> e1,`
`  field2 |-> e2]]`
	Supported; compiled code dependent on the type of `x` (the function). For example, when `x` is a `Slice`$[\tau]$, the function call `x[i]` is compiled to a simple slice indexing.

Binary operator call `x /\ y = z + 1`	Supported; compiled as expected

Records
`[field1 |-> exp1,`
`field2 |-> exp2]`
	Compiled as Go maps
	`map[string]interface{}{`
	`  "field1": exp1,`
	`  "field2": exp2,`
	`}`

Function set
`[Nat -> Nat]`
`\* or`
`[Nat -> 1..3]`
`\* or`
`[1..3 -> 1..3]`
	Unsupported

TLA$^+$ feature	PGo support

Function substitution
`[f EXCEPT ![exp1] = exp2]`
`\* or`
`[f EXCEPT !.field = exp]`
	Unsupported

If expression
`if condition`
`  then thenExp`
`  else elseExp`
	Compiled as
	`var result type;`
	`if condition {`
	`  result = thenExp`
	`} else {`
	`  result = elseExp`
	`}`
	`// result is used in place`
	`// of the expression`
	`// hereafter`

Tuple (as slice) `<<exp1, exp2, exp3>>`	Compiled as slice when all its contents are of the same type `[]type{exp1, exp2, exp3}`

Tuple (as struct)
`<<exp1, exp2, exp3>>`
	Compiled as a struct when at least one element is of a different type from the others' types.
	`struct {`
	`  e0 type`
	`  e1 type`
	`  e2 type`
	`}{exp1, exp2, exp3}`

TLA+ feature	PGo support

Case expression
`CASE x -> y`
`  [] z -> p`
`  [] OTHER -> other`
	Compiled as
	`var result type;`
	`if x {`
	`  result = y`
	`  goto matched`
	`}`
	`if z {`
	`  result = p`
	`  goto matched`
	`}`
	`result = other`
	`matched:`
	`// result is used in place`
	`// of the expression`
	`// hereafter`

Existental
`\E a, b, c : exp`
`\* or`
`\EE a, b, c : exp`
	Unsupportable; TLC throws an error when given this expression

Universal
`\A a, b, c : exp`
`\* or`
`\AA a, b, c : exp`
	Unsupportable; TLC throws an error when given this expression

Let expression
`LET op(a, b, c) == exp1`
`    fn[d \in D] == exp2`
`    e == exp3`
`IN exp`
	Unsupported

TLA$^+$ feature	**PGo support**

Assumption
`ASSUME exp`
`\* or`
`ASSUMPTION exp`
`\* or`
`AXIOM exp`
	Unsupported

Theorem `THEOREM exp`	Unsupported
Maybe action `[exp1]_exp2`	Unsupported
Required action `<<exp1>>_exp2`	Unsupported
Operator `Op(arg1, arg2) = exp`	Compiled as a Go function
Operator call `Op(exp1, exp2)`	Supported; compiled as a function call

TLA+ feature	PGo support

Quantified existential
`\E a \in exp1,`
`  b \in exp2 : exp3`
	Compiled as
	`exists := false`
	`for _, a := range exp1 {`
	`  for _, b := range exp2 {`
	`  if exp3 {`
	`  exists = true`
	`  goto yes`
	`  }`
	`  }`
	`}`
	`yes:`
	`// exists is used in place`
	`// of the expression`
	`// hereafter`

Quantified universal
`\A a \in exp1,`
`  b \in exp2 : exp3`
	Compiled as
	`forAll := true`
	`for _, a := range exp1 {`
	`  for _, b := range exp2 {`
	`  if !exp3 {`
	`  forAll = false`
	`  goto no`
	`  }`
	`  }`
	`}`
	`no:`
	`// forAll is used in place`
	`// of the expression`
	`// hereafter`

Set constructor `{exp1, exp2, exp3}`	Compiled as sorted slice `[]type{exp1, exp2, exp3}`

TLA$^+$ feature	PGo support

Set comprehension
`{exp : a \in exp1,`
`b \in exp2}`
	Compiled as
	`tmpSet := make([]type, 0)`
	`for _, a := range exp1 {`
	`  for _, b := range exp2 {`
	`  tmpSet = append(tmpSet,`
	`  exp)`
	`  }`
	`}`
	`// more code to ensure`
	`// elements in tmpSet`
	`// are unique and sorted`
	
	`// tmpSet is used in place`
	`// of the expression`
	`// hereafter`

Set refinement
`{a \in exp1 : exp}`
	Compiled as
	`tmpSet := make([]type, 0)`
	`for _, v := range exp1 {`
	`  if exp {`
	`  tmpSet = append(tmpSet,`
	`  v)`
	`  }`
	`}`
	`// tmpSet is used in place`
	`// of the expression`
	`// hereafter`

Table 4.2: PGo support for TLA$^+$ constructs

Structure of a compiled Go program

A compiled Go program has the following structure: an `init` function, compiled procedures, compiled user-defined operators, compiled processes (as Go func-

tions), and a `main` function.

The `init` function contains initialization code for constants. It also contains initialization code for global variables, as well as client initialization code for the global state backing store (etcd or state server), when compiling a multi-process specification.

When compiling a single-process specification, the `main` function contains the compiled output of the single process's body. When compiling a multi-process specification, it only contains a "process switch", which is a `switch` statement selecting the right process based on the command line argument passed to it at runtime. When compiling a multi-process specification using the state server strategy, there are two synchronization barriers before and after the process switch. These synchronization barriers ensure states are initialized correctly and processes terminate only when all other processes have finished their work.

Critical section tracking

Since PlusCal execution model requires that a statement must be part of exactly one label, PGo employs mutual exclusions (mutexes) to satisfy this requirement. At the start and end of a compiled labeled block, PGo inserts mutex lock and unlock code respectively. However, statements with differing execution paths, e.g., `if`, `while`, and `either` statements, pose a challenge to this simple compilation strategy, since each execution path may execute different labels as well as a different number of labels. Thus, PGo employs critical section tracking to handle this issue.

In critical section tracking, PGo keeps the current active label and lock group as part of its state. When a statement with diverging execution paths is compiled, PGo makes copies of the tracking state, and follows different paths with different copies. Thus, compilation interacts with labeling rules (see Section 4.1.4) in non-trivial ways. For example, Figure 4.17 shows a simple `while` loop whose condition `cond` is in label `cond_label`, which is different from its `body`'s label `body_label`, as well as the pseudocode for the output, where `CS` is short for critical section. Before entering the loop, PGo inserts code to enter the critical section for `cond_label` (line 1). After testing the condition, PGo exits the critical section for `cond_label` (line 7), and enters the critical section for `cond_body` (line 8) to execute the body of the loop

```
1  cond_label: while (cond) {
2  body_label:   body
3              };
```

```
1   enter CS for cond_label
2   for {
3     if !cond {
4       exit CS for cond_label
5       break
6     }
7     exit CS for cond_label
8     enter CS for body_label
9     body
10    exit CS for body_label
11    enter CS for cond_label
12  }
```

Figure 4.17: Example pseudocode output of `while` loop

(line 9). At the end of the loop, PGo exits the critical section for `body_label` (line 10), and enters the critical section for `cond_label` (line 11) so that the condition `cond` can be safely tested again.

Chapter 5

Evaluation

All tests in this chapter are preformed on an Intel Core i7-4720HQ CPU @ 2.60GHz with 16GB of RAM running Fedora 29 and OpenJDK 1.8.0_201. The TLA$^+$ toolbox version is Version 1.5.7 of 18 July 2018.

5.1 How effective is the compiled PlusCal output for model checking?

Since PGo relies on TLC to model check the Modular PlusCal specifications and TLC only accepts TLA$^+$ specifications as input, PGo has to compile Modular PlusCal to TLA$^+$. PGo does this by compiling Modular PlusCal to PlusCal, and then relies on the TLA$^+$ tool box to convert the resulting PlusCal to TLA$^+$.

This experiment tries to determine if there is any degradation in model checking performance and how much degradation is introduced by PGo's approach to compilation. In this experiment, three Modular PlusCal specifications, written by the PGo development team, with varying degrees of complexity are used as input to PGo.

The source Modular PlusCal specifications are formatted to maximize comprehension. All environment-modeling mapping macros are included in the specification. Each variable declaration is put on a new line with an accompanying comment explaining what the variable is. Thus, the source specifications are quite verbose.

On the other hand, the compiled PlusCal specifications are only formatted slightly with indentations by PGo. However, variable declarations are put on the same line.

- **Distributed queue.** The distributed queue specification is the simplest of the three specifications. The specification models a single-producer multiple-consumer queue. The producer is modeled as an archetype with two parameters, namely the network connections and a task queue. The producer continually listens for a work request from any client and then picks a task from the queue to send to the requesting client. A client is modeled as an archetype with two parameters, namely the network connections, and a task processor. A client continually sends a work request to the producer, gets back a task, and works on the task by invoking the task processor. The full Modular PlusCal specification is 55 lines long. The compiled PlusCal output has 59 lines of code.

- **Load balancer** The load balancer specification models a load balancing system with multiple clients and multiple servers, multiplexed by a single load balancer. The clients send requests to the load balancer, which forwards them to the servers in a round-robin fashion. The specification models the file system abstractly. The full Modular PlusCal specification is 87 lines long. The compiled PlusCal output has 93 lines of code.

- **Replicated key-value store** The replicated key-value store specification is the most complex of the three specifications. It models a replicated key-value store where the replicas only talk to the clients and state modification is only applied when the network message containing the command is stable. Message stability detection is implemented via buffered messages with logical time stamps. The full Modular PlusCal specification is 296 lines long. The compiled PlusCal output has 338 lines of code.

Each Modular PlusCal specification is compiled to PlusCal using PGo into two versions, one where the temporary variables are local to the processes, the other where the temporary variables are global variables. Additionally, the author manually creates another PlusCal version by removing all the temporary variables,

i.e., this version is the optimal output an ideal compiler can generate. The metrics collected for model checking are time (in seconds), diameter (the length of the longest behavior found), states found (the total number of states TLC found), and distinct states (the number of distinct states among the states found).

Parameters	**Version**	**Time** (seconds)	**Diameter**	**States Found**	**Distinct States**
1 consumer buffer size of 1	Local	5	13	25	18
	Global	5	13	25	18
	Optimal	5	10	21	15
2 consumers buffer size of 2	Local	5	48	8291	3858
	Global	5	30	1583	758
	Optimal	5	30	709	336
3 consumers buffer size of 3	Local	109	125	40285707	12840589
	Global	5	62	123723	45943
	Optimal	5	59	25065	8635
4 consumers buffer size of 4	Local	timeout	N/A	N/A	N/A
	Global	23	106	11842873	3790768
	Optimal	6	100	794705	214176
5 consumers buffer size of 5	Local	timeout	N/A	N/A	N/A
	Global	timeout	N/A	N/A	N/A
	Optimal	30	153	24708973	5459069
6 consumers buffer size of 6	Local	timeout	N/A	N/A	N/A
	Global	timeout	N/A	N/A	N/A
	Optimal	timeout	N/A	N/A	N/A

Table 5.1: Model checking results for distributed queue

Table 5.1 shows the model checking results for the distributed queue specification. The specification has two parameters: number of consumers and buffer size. With one consumer and a buffer size of one, the global temporary variable version does not have any model checking performance advantage compared to the local temporary variable version. Both versions degrade model checking performance when compared to the optimal version. The model checking time in seconds, however, shows that model checking for the three versions terminates in the same amount of time (five seconds). This may indicate that a significant portion of the time is spent in initializing TLC. With two consumers and a buffer

size of two, the local temporary variable version starts to show further degradation in model checking performance than the global temporary variable version in all collected metrics except for running time. Although the state space of the global temporary variable version has the same diameter as the optimal version, it contains more than twice as many distinct states as the optimal version. This means that the state spaces for the local and global temporary variable versions are going to grow much faster than that of the optimal version. However, the model checking time for the three versions are the same, and is also equal to the model checking time when there is only one consumer with buffer size of one. This strengthens the hypothesis that most of the five seconds is spent in initializing TLC. Increasing the values of the parameters significantly increases the size of the state space. However, the global temporary variable version shows a much slower growth than the local temporary variable version. Nevertheless, the global temporary variable version still shows much faster growth than the optimal version.

Overall, Table 5.1 shows that even such a simple specification as the distributed queue specification quickly runs into the state explosion problem. It also shows that declaring temporary variables as global variables is a huge state space reduction when compared to declaring them as local variables. However, that strategy is still not doing enough to keep the state space as small as the version where there are no temporary variables.

Parameters	**Version**	**Time** (seconds)	**Diameter**	**States Found**	**Distinct States**
1 client	Local	5	19	85	48
1 server	Global	5	19	85	48
buffer size of 1	Optimal	5	19	85	48
2 clients	Local	35	62	651053	221800
2 servers	Global	10	53	121153	41832
buffer size of 2	Optimal	5	53	16261	5592
3 clients	Local	timeout	N/A	N/A	N/A
3 servers	Global	timeout	N/A	N/A	N/A
buffer size of 3	Optimal	15	101	5316875	1338572
3 clients	Local	timeout	N/A	N/A	N/A
3 servers	Global	timeout	N/A	N/A	N/A
buffer size of 3	Optimal	timeout	N/A	N/A	N/A

Table 5.2: Model checking results for load balancer

Table 5.2 tells the same story as Table 5.1: global temporary variables are a huge help but are nowhere near enough to keep state space down.

Unfortunately, the state space for the replicated key-value store is infinite, so the programmer can only rely on bounded model checking which does not explore the entire state space. Running TLC for bounded time or bounded number of states does not show any difference among the three versions, but only shows the rate of exploration of the state space in terms of number of explored states, which is about 10 million distinct states per minute for all three versions. In addition, TLC does not show any other indication of progress, such as how much of the state space is explored in terms of percentage of the overall state space. Thus, no metrics are collected for the outputs of this specification.

5.2 How complex a specification can PGo compile?

This experiment tries to determine if PGo can compile various PlusCal specifications. In this experiment, six PlusCal specifications, which are a mix of pre-existing and newly written specifications, are used as input to PGo. Table 5.3 shows the sizes of the single-process specifications and their outputs. Table 5.4 shows the sizes of the multi-process specifications and their outputs. Below are the descriptions of the specifications.

- **Euclid.** The Euclid specification is a simple single-process PlusCal implementation of Euclid's algorithm for finding the greatest common denominator of two numbers. It does so by continually subtracting the smaller number from the larger number, and stopping only when the the larger number is 0 after subtraction. The result is printed to the screen. This specification is written by a PGo team member.

- **Queens.** The queens specification is a single-process PlusCal implementation to solve the n-queens problem. The n-queens problem asks for all placements of n queens on an n-by-n chessboard, such that no queen can attack another. The PlusCal implementation is a brute-force solution that tries to place the next queen such that it cannot attack any previously placed

queens, keeping only successful placements along the way. This specification is taken from the TLA$^+$ examples repository.

- **Counter.** The counter specification has a number processes all trying to increment a counter a number of times. This specification checks whether global state management is implemented correctly, by checking if the final value of the counter equals to the number of processes times the number of iterations. This specification is written by a PGo team member.
- **Round robin.** The round robin specification is just the counter specification with an added twist: each process is blocked waiting for its turn at the start of each iteration. At the start of execution, all processes race to get the first turn. This specification is written by a PGo team member.
- **Distributed queue.** The distributed queue specification models a single-producer multiple-consumer queue. The queue is modeled as a sequence of tasks. This specification is different from the one with the same name described in Section 5.1. This specification is written by a PGo team member.
- **Dijkstra's mutex.** The Dijkstra's mutex specification is a PlusCal implementation of the first mutual exclusion algorithm by Dijkstra [11]. This specification is taken from the TLA$^+$ examples repository.

Specification	**Size** (lines of code)	**Output size** (lines of code)
Euclid	21	36
Queens	49	203

Table 5.3: Sizes of single-process specifications and their outputs

Specification	Size (lines of code)	Multi-threaded output size (lines of code)	Distributed system output size (lines of code)
Counter	27	55	88
Round robin	30	66	115
Distributed queue	43	70	117
Dijkstra's mutex	41	173	340

Table 5.4: Sizes of multi-process specifications and their outputs

The results in Table 5.3 and Table 5.4 show that PGo can handle moderate-size specifications with non-trivial TLA$^+$ functionalities, such as set operations. PGo successfully compiles the Dijkstra's mutex specification which is a realistic specification from the TLA$^+$ examples repository. For examples of the Go outputs, see Section 5.2.1 and Section 5.2.2.

Since this thesis does not describe PGo's run-time system, and discussions about the performance of the generated Go program is not meaningful without discussing the run-time system, no metrics on the performance of the generated Go programs are collected. Studying the performance of PGo's compiled Go programs are deferred to future work.

To give an intuition about the Go outputs, we provide walkthroughs of two specifications in the next two subsections.

5.2.1 Walkthrough of the queens specification and its compiled single-threaded implementation

Figure 5.1 shows the queens specification in PlusCal. Line 1 pulls arithmetic operators on natural numbers from `Naturals`, sequence operators (such as `Len`, `Seq` (sequence constructor), and `Append`) from `Sequences`, and printing operator and procedure (`print` in PlusCal and `printS` in TLA$^+$) from `TLC`. The constant `N` denotes the size of the board (line 4), which is assumed to be positive (line 5). The `Attacks` operator (lines 7 to 10) checks if two queens on rows `i` and `j` can attack each other by checking whether they are on the same column (line 8), on the same first diagonal (line 9), or on the same second diagonal (line 10). The `IsSolution` operator (lines

12 to 14) checks if a placement is a solution by making sure that all queens cannot attack one another. The `Solutions` operator (line 16) contains all solutions for `N` queens, which is constructed by filtering all N^N placements using the `IsSolution` operator. The `IsSolution` and `Solutions` operators are not used in the PlusCal specification but they are used in checking whether the PlusCal specification is correct.

The `sols` variable (line 21) contains all solutions found so far, which is initialized as an empty set. The `todo` variable contains the tasks to be done, which is initialized as a set containing an empty placement (no queens are placed on the board). Whenever there is work to be done (line 24), a placement `queens` is extracted from `todo` (line 26). The next row, where the new queen is placed, `nxtQ`, is extracted on line 27. The `cols` variable (lines 28 and 29) is the set of all columns where the next queen can be placed. The `exts` variable (line 30) is the set of new placements, constructed by extending the placement on line 26 with the columns on line 28. If the next row `nxtQ` is the last row to be filled (line 32), then `todo` is updated by removing the placement `queens`, and the solution set `sols` is extended with the new solutions, now in `exts` (line 33). Otherwise, the placement `queens` is removed from `todo` and new placements, `exts`, are added to `todo`. When there are no tasks left in `todo`, the set of solutions is printed (line 37).

There are some additional invariants to check during model checking. `TypeInvariant` (lines 42 to 44) states that `todo` is a subset of the power set of the sequence `1 .. N` (line 43), all placements in `todo` must have length less than `N` (line 43), `sols` is also a subset of the power set of the sequence `1 .. N` (line 44), and all solutions in `sols` must have length exactly `N`. `Invariant` (lines 46 to 49) states that must always be a subset of `Solutions` (defined on line 16), and when there is no more work (`todo` is empty), `Solutions` must be a subset of `sols`, which, together with the previous condition, means that `Solutions` and `sols` must be equal.

The compiled single-threaded Go program has 203 lines of code. The configuration for this specification only contains the output path and file name, as well as the value for `N`, the problem size.

As show in Figure 5.2, it starts with the package name (line 1), imported packages (lines 3 to 6), declaration of `N` as a Go variable (line 8), and the `init` function, which initializes `N` to `8`, as specified in the configuration. Since only the `Attacks` operator is used in the PlusCal specification, only that operator is compiled. Lines

```
1  -------------------------- MODULE Queens -----------------------------
2  EXTENDS Naturals, Sequences, TLC
3
4  CONSTANT N              \** number of queens and size of the board
5  ASSUME N \in Nat \ {0}
6
7  Attacks(queens,i,j) ==
8    \/ queens[i] = queens[j]                \** same column
9    \/ queens[i] - queens[j] = i - j        \** first diagonal
10   \/ queens[j] - queens[i] = i - j        \** second diagonal
11
12 IsSolution(queens) ==
13   \A i \in 1 .. Len(queens)-1 : \A j \in i+1 .. Len(queens) :
14        ~ Attacks(queens,i,j)
15
16 Solutions == { queens \in [1..N -> 1..N] : IsSolution(queens) }
17
18 (* --algorithm QueensPluscal
19      variables
20        todo = { << >> };
21        sols = {};
22
23      begin
24 nxtQ:  while todo # {}
25        do
26          with queens \in todo,
27               nxtQ = Len(queens) + 1,
28               cols = { c \in 1..N : ~ \E i \in 1 .. Len(queens) :
29                                       Attacks( Append(queens, c), i, nxtQ ) },
30               exts = { Append(queens,c) : c \in cols }
31          do
32            if (nxtQ = N)
33            then todo := todo \ {queens}; sols := sols \union exts;
34            else todo := (todo \ {queens}) \union exts;
35            end if;
36          end with;
37        end while;
38        print sols;
39      end algorithm
40 *)
41
42 TypeInvariant ==
43   /\ todo \in SUBSET Seq(1 .. N) /\ \A s \in todo : Len(s) < N
44   /\ sols \in SUBSET Seq(1 .. N) /\ \A s \in sols : Len(s) = N
45
46 Invariant ==
47   /\ sols \subseteq Solutions
48   /\ todo = {} => Solutions \subseteq sols
49 =============================================================================
```

Figure 5.1: Queens specification

```
1  package main
2
3  import (
4          "fmt"
5          "sort"
6  )
7
8  var N int
9
10 func init() {
11         N = 11
12 }
13
14 func Attacks(queens []int, i int, j int) bool {
15         return queens[i-1] == queens[j-1] ||
16                 queens[i-1]-queens[j-1] == i-j ||
17                 queens[j-1]-queens[i-1] == i-j
18 }
19
20 func main() {
21         todo := [][]int{[]int{}}
22         sols := [][]int{}
```

Figure 5.2: Compiled queens specification - preamble

```
1          for {
2                  if !(len(todo) != 0) {
3                          break
4                  }
```

Figure 5.3: Compiled queens specification - `while` condition

14 to 18 shows the compiled output of the `Attacks` operator, with the expression split on multiple lines to fit on the page. The operator is compiled into a Go function with three parameters. PGo correctly infers the types of the parameters and the return type of the operator, without aid from the programmer.

The rest of the specification is compiled into the `main` function in Go, starting with the declaration and initialization of `todo` and `sols`. The variables `todo` and `sols` are inferred to be sets of sequences of integers, and TLA$^+$ sets are compiled as sorted slices while sequences are compiled as slices, the two variables are of type slices of slices of integers in Go. Since `todo` is initialized as a set containing an empty sequence in PlusCal, it is initialized as a slice containing a single empty slice (line 21). Since `sols` is initialized as an empty set, it is initialized as an empty slice (line 22).

Figure 5.3 shows how the `while` condition on line 24 of Figure 5.1 is compiled.

```
1                queens := todo[0]
2                nxtQ := len(queens) + 1
3                tmpSet := make([]int, 0)
4                tmpRange := make([]int, N-1+1)
5                for i := 1; i <= N; i++ {
6                        tmpRange[i-1] = i
7                }
8                for _, c := range tmpRange {
9                        exists := false
10                       tmpRange0 := make([]int, len(queens)-1+1)
11                       for i := 1; i <= len(queens); i++ {
12                               tmpRange0[i-1] = i
13                       }
14                       for _, i := range tmpRange0 {
15                               tmpSlice := make([]int, len(queens), len(queens)+1)
16                               copy(tmpSlice, queens)
17                               tmpSlice = append(tmpSlice, c)
18                               if Attacks(tmpSlice, i, nxtQ) {
19                                       exists = true
20                                       break
21                               }
22                       }
23                       if !exists {
24                               tmpSet = append(tmpSet, c)
25                       }
26               }
27               cols := tmpSet
```

Figure 5.4: Compiled queens specification - `queens`, `nxtQ`, and `cols`

Since this is a single-process output, there is no need to insert locking code. Note that the check agains an empty collection is compiled as a length check against zero.

Figure 5.4 shows how lines 26 to 29 of Figure 5.1 are compiled. Note that the variable `queens` is assigned to the zeroth element of `todo`. Lines 4 to 7 show how `1 .. N` is compiled. Lines 8 to 27 show how a set refinement, which removes elements for which the check returns false, is compiled, with the `Attacks` check on line 29 of Figure 5.1 compiled on line 18. There is no need to sort the result, since element removal maintains the sortedness property.

Figure 5.5 shows how line 30 of Figure 5.1, which is a set comprehension, is compiled. It looks involved because it contains code for set element construction (lines 2 to 7), sorting the constructed elements to establish sortedness (lines 8 to 19), and removal of duplicate elements to establish uniqueness (lines 20 to 40).

Since the condition for the `if` condition on line 32 of Figure 5.1 is a simple

```
tmpSet0 := make([][]int, 0)
for _, c := range cols {
        tmpSlice := make([]int, len(queens), len(queens)+1)
        copy(tmpSlice, queens)
        tmpSlice = append(tmpSlice, c)
        tmpSet0 = append(tmpSet0, tmpSlice)
}
sort.Slice(tmpSet0, func(i int, j int) bool {
        less := len(tmpSet0[i]) < len(tmpSet0[j])
        if len(tmpSet0[i]) == len(tmpSet0[j]) {
                for i0 := 0; i0 < len(tmpSet0[i]); i0++ {
                        less = tmpSet0[i][i0] < tmpSet0[j][i0]
                        if tmpSet0[i][i0] != tmpSet0[j][i0] {
                                break
                        }
                }
        }
        return less
})
if len(tmpSet0) > 1 {
        previousValue := tmpSet0[0]
        currentIndex := 1
        for _, v := range tmpSet0[1:] {
                eq := len(previousValue) == len(v)
                if eq {
                        for i0 := 0; i0 < len(previousValue); i0++ {
                                eq = previousValue[i0] == v[i0]
                                if !eq {
                                        break
                                }
                        }
                }
                if !eq {
                        tmpSet0[currentIndex] = v
                        currentIndex++
                }
                previousValue = v
        }
        tmpSet0 = tmpSet0[:currentIndex]
}
exts := tmpSet0
```

Figure 5.5: Compiled queens specification - `exts`

```
1                       tmpSet1 := make([][]int, 0, len(todo))
2                       for _, v := range todo {
3                               eq := len(v) == len(queens)
4                               if eq {
5                                       for i0 := 0; i0 < len(v); i0++ {
6                                               eq = v[i0] == queens[i0]
7                                               if !eq {
8                                                       break
9                                               }
10                                      }
11                              }
12                              if !eq {
13                                      tmpSet1 = append(tmpSet1, v)
14                              }
15                      }
16                      todo = tmpSet1
```

Figure 5.6: Compiled queens specification - update `todo` when solutions are found

integer comparison, it is compiled simply as `nxtQ == N`. Figure 5.6 shows how the update to `todo` on line 33 of Figure 5.1, which is a set difference, is compiled.

Figure 5.7 shows how the update to `sols` on line 33 of Figure 5.1, which is a set union, is compiled. It looks involved because sortedness and uniqueness have to be re-established (lines 4 to 15 and lines 16 to 36, respectively).

Since lines 34 of Figure 5.1 is just a combination of a set difference and a set union, which are already shown in Figure 5.6 and Figure 5.7, its compiled output will not be shown.

From the compiled output, the following observations can be made.

- Constants are compiled as Go variables. This is the case due to the possibility of complex initialization, e.g., set refinement.
- PGo makes liberal use of temporary variables in its Go compiled output.
- PlusCal `while` loops are compiled as infinite loops with in-body checks. This is the case also due to the possibility of complex loop condition.
- TLA$^+$ sets are compiled as immutable sorted slices in Go.

```
1                       tmpSet2 := make([][]int, len(sols), len(sols)+len(exts))
2                       copy(tmpSet2, sols)
3                       tmpSet2 = append(tmpSet2, exts...)
4                       sort.Slice(tmpSet2, func(i0 int, j0 int) bool {
5                               less0 := len(tmpSet2[i0]) < len(tmpSet2[j0])
6                               if len(tmpSet2[i0]) == len(tmpSet2[j0]) {
7                                       for i1 := 0; i1 < len(tmpSet2[i0]); i1++ {
8                                               less0 = tmpSet2[i0][i1] < tmpSet2[j0][i1]
9                                               if tmpSet2[i0][i1] != tmpSet2[j0][i1] {
10                                                      break
11                                              }
12                                      }
13                              }
14                              return less0
15                      })
16                      if len(tmpSet2) > 1 {
17                              previousValue := tmpSet2[0]
18                              currentIndex := 1
19                              for _, v := range tmpSet2[1:] {
20                                      eq := len(previousValue) == len(v)
21                                      if eq {
22                                              for i1 := 0; i1 < len(previousValue); i1++ {
23                                                      eq = previousValue[i1] == v[i1]
24                                                      if !eq {
25                                                              break
26                                                      }
27                                              }
28                                      }
29                                      if !eq {
30                                              tmpSet2[currentIndex] = v
31                                              currentIndex++
32                                      }
33                                      previousValue = v
34                              }
35                              tmpSet2 = tmpSet2[:currentIndex]
36                      }
37                      sols = tmpSet2
```

Figure 5.7: Compiled queens specification - update `sols` when solutions are found

5.2.2 Walkthrough of round robin specification and its compiled distributed sytem implementation

Figure 5.8 shows the round robin specification.

The constant `procs` (line 4) is the number of processes in the distributed systems. The constant `iters` (line 4) is the number of iterations each process performs. The `token` (line 8) represents whose turn it is to increment the `counter` (line 9).

At the begining of each iteration, each process waits for its turn (line 15). It then increments `counter` (line 16), passes the token to the next process (line 17),

```
1  --------------------------- MODULE round_robin ---------------------------
2  EXTENDS Integers, TLC
3
4  CONSTANT procs, iters
5
6  (*
7  --algorithm round_robin {
8    variables counter = 0,
9              token = -1;
10
11   fair process (P \in 0..procs-1)
12   variables i = 0;
13   {
14       w: while (i < iters) {
15           waitToken:  await token = -1 \/ token = self;
16           incCounter: counter := counter + 1;
17                       token := (self + 1) % procs;
18                       print counter;
19           nextIter:   i := i + 1;
20       }
21   }
22 }
23 *)
24
25 TokenWithinBounds ==
26   token = -1 \/ token \in 0..procs-1
27
28 CounterConverges ==
29     (\A self \in ProcSet: pc[self] = "Done") => (counter = procs * iters)
30 ===============================================================================
```

Figure 5.8: Round robin specification

and prints the value of `counter`. Each process performs the above steps `iters` times (lines 14 and 19).

The `TokenWithinBounds` invariant (lines 25 and 26) checks the value of `token` is never out of bounds. The `CounterConverges` invariant (lines 28 and 29) states that when all processes are done, the value of `counter` must be equal to the number of processes, `procs`, times the number of iterations, `iters`.

Figure 5.9 shows how line 15 of Figure 5.8 is compiled. Lines 2 to 8 obtain the write locks on `counter` and `token`. These locks are saved into a handle named `refs` (line 2). The local values of `counter` and `token` are updated on lines 9 and 10. The PlusCal `await` statement is compiled as an `if` statement with a `goto` statement, whose target is the current label (`waitToken`), so that the condition can be tested again.

The `globalState` variable is a handle on the distributed state of the program.

```
1           waitToken:
2                   refs, err = globalState.Acquire(&distsys.BorrowSpec{
3                           ReadNames:  []string{},
4                           WriteNames: []string{"counter", "token"},
5                   })
6                   if err != nil {
7                           panic(err)
8                   }
9                   counter = refs.Get("counter").(int)
10                  token = refs.Get("token").(int)
11                  if !(token == -1 || token == self) {
12                          err = globalState.Release(refs)
13                          if err != nil {
14                                  panic(err)
15                          }
16                          goto waitToken
17                  }
18                  refs.Set("counter", counter)
19                  refs.Set("token", token)
20                  err = globalState.Release(refs)
21                  if err != nil {
22                          panic(err)
23                  }
```

Figure 5.9: Compiled distributed system for round robin specification

Each piece of state is a protected global variable used in the compiled program. Each process has ownership of a global variable once it has obtained a lock on that variable. The ownership is moved among processes during execution. When a lock on a variable is released, the ownership of that variable still stays with the process until another process obtains the lock on that variable.

From Figure 5.9, the following observations can be made.

- PlusCal `await` statements are compiled into busy wait loops. This may hinder performance if there are too many retries.
- PGo's atomicity inference is imprecise. The `counter` variable is not used in the `waitToken` label but it is still locked in the compiled output.

Chapter 6

Discussion

6.1 Limitations

Modular PlusCal limits the interface of archetype parameters to only reading and writing. This limits the kinds of systems that can be modeled using Modular PlusCal. For example, a programmer may find it hard to model a file system that has an interface with three functionalities: read from, write to, and seek within a file.

PGo makes liberal use of temporary variables when compiling from Modular PlusCal to PlusCal. These temporary variables capture intermediate states, which greatly increases the state space for TLC to explore. Currently, PGo's PlusCal outputs can be used during system development by setting parameter values to be small. However, for production deployment, larger values for parameters are used to provide higher assurance, which greatly increases model checking time. More effort in eliminating these temporary variables must be invested to manage model checking time.

PGo's type system, while being an improvement over the previous iteration, is still limited. The lack of support for more advanced typing paradigms, such as recursive types, coupled with poor error reporting, makes for a frustrating user experience.

PGo's atomicity inference algorithm is prone to lumping all global variables into one lock group, which effectively eliminates concurrency, turning the system into a single-threaded program. A different approach to discovering finer grained

lock groups would mitigate this problem.

6.2 Future work

In addition to being a verification tool, TLC may be used in PGo's compilation process as an inference tool. For example, prefetching may be framed as an inference problem, where correlation among usages of variables may be encoded as a synthetic variable whose value is checked by TLC. To achieve this, PGo must be extended with a mechanism to manipulate generated TLA$^+$ output, to invoke TLC on its own, and to parse the output of TLC to obtain the value of the synthetic variable.

Currently, PGo is unverified. Verifying PGo will remove it from the trusted computing base, increasing the programmer's confidence in the correctness of the output implementation. One method to verify PGo is to formally model Modular PlusCal, PlusCal, TLA$^+$, and Go, using various dynamic semantics, such as small-step operational semantics. With these semantics available, equivalence between the source specification and the compiled output can be formally established.

More effort should be invested into producing quality Go code to exact more performance at runtime. For example, currently, PGo outputs mostly immutable data structures by making multiple copies of the data. Switching to in-place modification may provide performance enhancement to PGo-generated programs.

PGo only outputs to Go. However, due to factors such as library availability and team consensus, it may not always be possible to incorporate Go tools into an existing team's workflow. Thus, it is desirable for PGo to output to other languages, such as Rust or Java, to promote adoption. Since Go has a simple type system and requires explicit error handling, Go programs may not be a suitable intermediate representation (IR) for code generation to other languages. On the other hand, lower-level IR, such as LLVM IR, may be too low level, which removes high level details about PlusCal execution model. Instead, a new IR for PGo is recommended for ease of multi-target code generation.

Chapter 7

Related work

Domain-specific languages.

There are many languages that are tailored for writing distributed systems. However, they do not provide facilities for checking the correctness of the distributed system implementation. For example, the Emerald programming language [7] requires the programmer to structure the distributed system as distributed objects. The smallest unit of execution in Emerald is an active object, which is an object with an associated process. Emerald objects expose methods that provide functionalities for other objects. The Argus programming language [18] also requires the programmer to structure the distributed system as a collection of distributed objects, called guardians. Guardians are periodically serialized to disk so that they can be restored after crashes. In addition to guardians, Argus also introduces actions, which are executed atomically as composable transactions. An action or a group of composed actions are executed in its entirety (committed) or have no visible effect on the whole system (failed). The Erlang programming language [6] is a functional programming language based on the actor model. The unit of execution in Erlang is called a process, which is a green thread in modern parlance. Processes in Erlang communicate by passing messages. They are categorized into workers and monitors. Workers provide the functionalities required from the system, while monitors are in charge of restarting the workers when one of them fails. These languages provide the programmers with tools to structure their distributed systems. In contrast to these languages and systems, PGo provides tools

to check for correctness via the TLA$^+$ tool box.

Correspondence using proof assistants.

There have been many attempts at bridging the gap between formally verified specifications and system implementations. PGo differs from these approaches by not requiring the user to write proofs for verification. Instead, PGo relies on model checking, using TLC for verification. For example, Verdi [20] requires the programmer to write the specification, the implementation, and the proof that the implementation matches the specification in Coq [19]. The specification , and the implementation are written against an ideal networking environment (e.g., no packet drop, reordering, nor corruption). Verdi provides system transformers to transform the system of specification, implementation and proof into one that can handle faults. The transformed system is then verified by the Coq theorem prover. The implementation is then extracted to an executable using Coq's tools. IronFleet [13] proposes a methodology to structure the implementation of a distributed system and its increasingly abstract specifications as layers to allow feasible verification of practical distributed system implementations. Both Verdi and IronFleet place a big burden of writing the proofs on the programmer. This burden is not trivial, since it has been shown in both projects that the proof effort is at least ten times the implementation effort in terms of lines of code. Instead of requiring formal proofs, PGo relies on model checking for verification. This erases the burden of proof at the expense of verification time due to state-space explosion.

Correspondence using model checkers.

There are also other efforts of bridging the gap between formal specification and system implementation using model checking. PGo differs from these projects in that its input languages, namely PlusCal and Modular PlusCal, does not require the programmer to structure the system as a state machine. For example, Mace [14, 15] is a source-to-source compiler from a highly structured domain-specific language (DSL) to C++. It requires the programmer to write distributed systems as state machines, complete with explicit states and transitions. This makes the implementation amenable to model checking. P [10] provides a DSL for asynchronous event-driven programs and a model checker to explore the state spaces for verification. Its successor, P# [9], extends P for writing distributed systems, and provides a test environment for systematic exploration of the state space.

Instead of requiring the system to be written as a state machine, PGo provides more freedom in how the specification is structured.

Checking concrete implementations.

Instead of checking specifications, there have been approaches to directly checking the implementation. PGo's approach, on the other hand, provides efficient verification and automatic translation from abstract model to concrete implementation. For example, Verisoft [12] aims to check concrete programs via stateless search. It employs partial-order reduction techniques and bounds on the number of states explored to keep checking time manageable. MODIST [21] extends this to distributed systems. Work in this category has the benefit of being a drop-in solution for existing implementations. In contrast, PGo's approach provides automated verification and translation from abstract model to concrete implementation.

Chapter 8

Conclusion

This thesis presents Modular PlusCal, an extension of PlusCal with the goal of adding a clear distinction between system specification and environment specification, and PGo, a compiler from Modular PlusCal and PlusCal specifications to Go distributed system implementations. PGo reduces the burden of translating a specification to an implementation, which increases programmer productivity.

Bibliography

[1] Summary of the Amazon EC2 and Amazon RDS Service Disruption in the US East Region. URL https://web.archive.org/web/20190407054157/https://aws.amazon.com/message/65648/ (Accessed on 2019-03-14). → pages 1, 2

[2] Azure Cosmos TLA+ specifications. Contribute to Azure/azure-cosmos-tla development by creating an account on GitHub. URL https://web.archive.org/web/20190415013638/https://github.com/Azure/azure-cosmos-tla (Accessed on 2019-04-11). → page 2

[3] TLA+ Proof System, . URL https://web.archive.org/web/20181127125818/http://tla.msr-inria.inria.fr/tlaps/content/Home.html (Accessed on 2019-04-11). → page 2

[4] Cloud based distributed TLC, . URL https://web.archive.org/web/20190415013939/https://tla.msr-inria.inria.fr/tlatoolbox/doc/cloudtlc/ (Accessed on 2019-04-11). → page 2

[5] The TLA+ Hyperbook. URL https://web.archive.org/web/20190217125233/http://lamport.azurewebsites.net/tla/hyperbook.html (Accessed on 2019-03-14). → page 1

[6] J. Armstrong. Making Reliable Distributed Systems in the Presence of Software Errors. URL https://web.archive.org/web/20190405190606/http://erlang.org/download/armstrong_thesis_2003.pdf (Accessed on 2019-04-14). → page 73

[7] A. P. Black, N. C. Hutchinson, E. Jul, and H. M. Levy. The Development of the Emerald Programming Language. In *Proceedings of the Third ACM*

SIGPLAN Conference on History of Programming Languages, HOPL III, pages 11–1–11–51. ACM. ISBN 978-1-59593-766-7. doi:10.1145/1238844.1238855. → page 73

[8] F. Z. B. M. M. B. M. C. N. Deardeuff, Tim Rath. How Amazon Web Services Uses Formal Methods. URL https://web.archive.org/web/20190302044811/https://cacm.acm.org/magazines/2015/4/184701-how-amazon-web-services-uses-formal-methods/abstract (Accessed on 2018-05-02). → page iii

[9] P. Deligiannis, A. F. Donaldson, J. Ketema, A. Lal, and P. Thomson. Asynchronous Programming, Analysis and Testing with State Machines. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '15, pages 154–164. ACM. ISBN 978-1-4503-3468-6. doi:10.1145/2737924.2737996. → pages 2, 74

[10] A. Desai, V. Gupta, E. Jackson, S. Qadeer, S. Rajamani, and D. Zufferey. P: Safe Asynchronous Event-driven Programming. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '13, pages 321–332. ACM. ISBN 978-1-4503-2014-6. doi:10.1145/2491956.2462184. → page 74

[11] E. W. Dijkstra. Solution of a Problem in Concurrent Programming Control. 8(9):569–. ISSN 0001-0782. doi:10.1145/365559.365617. → page 60

[12] P. Godefroid. Model Checking for Programming Languages Using VeriSoft. In *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '97, pages 174–186. ACM. ISBN 978-0-89791-853-4. doi:10.1145/263699.263717. → page 75

[13] C. Hawblitzel, J. Howell, M. Kapritsos, J. R. Lorch, B. Parno, M. L. Roberts, S. Setty, and B. Zill. IronFleet: Proving Safety and Liveness of Practical Distributed Systems. 60(7):83–92. ISSN 0001-0782. doi:10.1145/3068608. → page 74

[14] C. Killian, J. W. Anderson, R. Jhala, and A. Vahdat. Life, Death, and the Critical Transition: Finding Liveness Bugs in Systems Code. In *Proceedings of the 4th USENIX Conference on Networked Systems Design & Implementation*, NSDI'07, pages 18–18. USENIX Association, . → pages 2, 74

[15] C. E. Killian, J. W. Anderson, R. Braud, R. Jhala, and A. M. Vahdat. Mace: Language Support for Building Distributed Systems. In *Proceedings of the*

28th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '07, pages 179–188. ACM, . ISBN 978-1-59593-633-2. doi:10.1145/1250734.1250755. → pages 2, 74

[16] L. Lamport. *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley Longman Publishing Co., Inc., . ISBN 978-0-321-14306-8. → page 5

[17] L. Lamport. A High-Level View of TLA+, . URL https://web.archive.org/web/20190415014843/https://lamport.azurewebsites.net/tla/high-level-view.html (Accessed on 2019-04-11). → page 1

[18] B. Liskov. Distributed Programming in Argus. 31(3):300–312. ISSN 0001-0782. doi:10.1145/42392.42399. → page 73

[19] T. C. D. Team. The Coq Proof Assistant, version 8.9.0. URL https://web.archive.org/web/20190415015254/https://zenodo.org/record/2554024 (Accessed on 2019-04-01). → page 74

[20] J. R. Wilcox, D. Woos, P. Panchekha, Z. Tatlock, X. Wang, M. D. Ernst, and T. Anderson. Verdi: A Framework for Implementing and Formally Verifying Distributed Systems. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '15, pages 357–368. ACM. ISBN 978-1-4503-3468-6. doi:10.1145/2737924.2737958. → page 74

[21] J. Yang, T. Chen, M. Wu, Z. Xu, X. Liu, H. Lin, M. Yang, F. Long, L. Zhang, and L. Zhou. MODIST: Transparent Model Checking of Unmodified Distributed Systems. In *Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation*, NSDI'09, pages 213–228. USENIX Association. → page 75