UiO : Department of Informatics
University of Oslo

# Shared Variables in Go

## A Semantic Analysis of the Go Memory Model

Stian Valle

Master's Thesis Autumn 2016

**Abstract**

Memory models as a part of programming language specifications have become increasingly popular the last two decades. They describe how the values that are obtained by *reads* are related to the values that are written by *writes*. To properly define this has proven particularly difficult for programming languages that allows for shared variables between multiple processes. In this thesis we formalize parts of the memory model specified by the Go language by making a structural operational semantics for it. We further use this semantics to prove that programs that are data race free will run under this semantics as they would under a *strong* memory model.

# Acknowledgments

Foremostly, I would like to thank my main supervisor, *Martin Steffen*, who showed me a direction for this thesis, and thoroughly discussed some of the concepts of the thesis with me. I learned a lot from him. Secondly, I would like to thank my secondary supervisor *Volker Stolz*. Even though I did not get to meet him very much, I believe he played a role behind the scenes.

I also want to thank *Lars Tveito*, for reading my thesis and providing feedback and suggestions, and *Carl Martin Rosenberg*, who I am certain would to the same if he had been given the opportunity. They have both been encouraging.

A mention will go to Olso Go club. Even though it's not a club for Go programmers, but rather the board game go, go is go, and this game is one of my favorite pastimes.

Last, but not least, I would like to raise a general thanks to friends and family, and in particular my brothers.

*Stian Valle*

# Contents

# List of Figures

# Chapter 1

# Introduction

Data races have been bugging programmers since the early days of concurrent programming, and will most likely continue to be something for a programmer to worry about if the programmer should embark to use memory that is shared between processes. Sure enough, idioms, like functional programming, and channel communication, have been created and/or adapted to lessen the need of shared memory, but still, there are some advantages to using shared memory that can't to date be entirely replaced by using any of the other idioms.

Most modern languages support shared memory in one way or the other, typically along with an encouragement to keep programs "data race free". However, ensuring that a program is data race free is easier said than done. In the general case, it is provably impossible for compilers to check for this feature, so determining this is left to the programmer. Some tools can be of help to accomplish this, like a runtime data race checker.

Since data races can slip into the code of even the most experienced of programmers using shared memory, and in favor of some lock-free algorithms, some languages tries to define some behavior for programs with data races as well. The rest of the languages typically keeps the behavior of such programs undefined, in a "it might work, but you're on your own"-manner. The programming language Go falls into the former category, but the creators clearly discourages programmers from taking advantage of it, as they write in the document that defines this behavior[1] that: "If you must read the rest of this document to understand the behavior of your program, you are being too clever. Don't be clever."

So, even though they do specify some behavior, they seem to deem it too obscure to be very helpful. The purpose of this thesis is to investigate some properties of the Go memory model, and thereby hopefully also bring some clarity to the model. To achieve this, we will do the following:

- Make a structural operational semantics for the *happens before*-

[1] "The Go Memory Model" [21].

relation.

- Demonstrate how the semantics works with examples.
- Prove that programs that are *data race free* will run under the semantics as they would in the semantics given by *sequential consistency*.
- Show that the semantics can be extended to a functional setting with closures.

## 1.1 Ordering of Events

When multiple processes run concurrently, it is common that they try to read from and write to variables they share with some of the other processes. Ideally it would be possible to write out the execution of all statements in all processes on a single time line. Then (still ideally) it should be possible to reason that if there was a read on a variable by any process at time $t$, the value of that read would be the same as the value of the latest write to that variable prior to $t$. It should not matter which processes that did the read and the writes.

We will hereby will refer to this ideal situation as *sequential consistency*. Sequential consistency is a term coined by Leslie Lamport [11], and is a notion that has generally proven important in discussion of programs with shared variables [1].

There is an issue, however, if multiple cores are allowed to run processes, and that would be that it is common for cores to *cache* values. In general, a core will only operate on the cached values, and only fetch or store to global memory when needed.

So how does this collide with the aforementioned notion of sequential consistency? Say that we still were able to splash out the statements on a time line as they occurred, regardless of which core they occurred on, and say that process $P$ is running on one core, and process $Q$ is running on another. Then consider a situation where $P$ make a read from variable $a$ at time $t$, and that, on the time line, process $Q$ was the last process to write a value to $a$ prior to $t$. By the sequential consistency model, process $P$ should read the same value as process $Q$ wrote. Nevertheless, on modern hardware the value written by process $Q$ could still reside only on the cache of the core that is running $Q$.

One would say that the write $Q$ did is not necessarily *visible* to $P$. There is often a *delay* between the write to a variable and the time the write is visible to other processes. If $P$ is really required to see the write that $Q$ performed, the cores needs some means of *synchronization*.

The conflict between the expectation given by sequential consistency and the actual execution is a problem. There are two possible solutions.

One is to make hardware that allow both efficiency and is conforming with the sequential consistency model. This might well be physically impossible. The other solution is to make models that are easier to implement efficiently. Such mathematical models that states which writes that are expected to be visible at which reads are *memory models*.

## 1.2 Memory models

Memory models specify what possible values one can expect to *observe* on any given *read* of a variable or memory location. For simplicity I will from here leave out the option of "memory location" and let it be implicitly understood as part of what I mean when I write "variable". A memory model will specify how values that can be obtained by *reads* are related to the values that are written by *writes*.

Sequential consistency is the base line for what we would expect from a *strong* memory model. Memory models that does not guarantee sequential consistency are referred to as *relaxed* or *weak* memory models.

Most modern memory models are weak memory models, and are today implemented both in hardware and in programming languages. The models generally focus on minimizing the times a process implicitly have to wait when other processes performs writes to shared variables, while also giving reasonable guarantees for the behavior of the programs that are to run on the model. By contrast, in strong memory models, any process that is about to access a variable must wait while and whenever another process is busy writing to it.

Memory models for concurrency were first introduced on hardware, and in the mid 90's programming languages started to try to follow suit. Java was one of the first languages trying to accomplish this, but, as it has been discovered, making a coherent memory model for a concurrent programming language is no mean feat [1, 6, 19].

The Java memory model currently builds on a notion that they refer to as *happens before*[8], another notion that can be traced back to Lamport [12]. We will investigate this notion in detail in this thesis, as it has become an integral part of the memory models of programming languages.

## 1.3 The Go memory model

Go [4, 22] is a programming language that allows for easy spawning of parallel processes, or, as they call it, goroutines. Although goroutines has some extra properties, we will think of them solely as processes, and for the rest of the text I will use the words *process* and *goroutine* interchangeably.

Since Go is a programming language dealing with concurrency, it is useful to the programmers that the language also specifies some behavior on shared variables. The Go memory model [21] is a weak memory model, and specifies a partial ordering on the execution of statements, which is referred to as *happens before*. Essentially, if the execution of statement X *happens before* the execution of statement Y, X is *not* allowed to "see" any side effects (e.g. write) of Y . On the flip side, if this is not the case, then there is a possibility that it can.

This *happens before*-relation is known to have a bit misleading name, and is thus very prone to misinterpretation. Therefore it is necessary to keep a certain cautiousness when dealing with it, so one does not draw conclusions that are not true. The problem with the name is that it tend to tend to give people expectations on the *execution order* of statements. Two statements can be in a *happens before*-relationship with each other, but still be *executed* in the reverse order of what is indicated by this relation. We will provide an example of this in the following section. To free oneself from this mind trap, it can be useful to *happens before*-relation abbreviate it to *hb*, and think of it as just a relation.

### Assignment reordering

Consider the following program where `a` and `b` are initially 0, and let `read()` and `write()` be performed concurrently by two different goroutines.

```
var a = 0
var b = 0

func write() {
   a = 1
   b = 2
}

func read() {
   fmt.Printf("b = %d\n", b) // print b
   fmt.Printf("a = %d\n", a) // print a
}
```

According to the specification of the Go memory model, the assignments `a=1` and `b=2` can be executed in reversed order (for instance due to compiler optimization). Meanwhile it also states that the *happens before* order is the same as the ones specified by the program. In effect this means that, even though they don't need to be executed in that order, the execution of `a=1` *happens before* `b=2`. We will examine why this isn't a contradiction.

The reason that the specification allow this reordering is that `a` and `b` are different variables, and they are updated without any synchronization

between them. So if there had been only one goroutine there would be no way to tell from within that goroutine whichever order they were executed anyway.

So, what happens when we have two goroutines? From the go memory model we know that `a=1` *happens before* `b=2`, and `print b` *happens before* `print a`. Nonetheless, the following table contains all possible printouts, one in each column.

| | | | |
|---|---|---|---|
| b = 0 | b = 2 | b = 0 | b = 2 |
| a = 0 | a = 0 | a = 1 | a = 1 |

We will focus on column two, as this is the column that best suggests that even though the goroutine that executes the read-function can observe that the assignment `b=2` must have happened, it still can not necessarily observe that the assignment `a=1` have happened. We can explain this behavior by pointing out that `a=1` and `b=2` may have been reordered, but again, we want to be able to explain why this still does not contradict `a=1` *happens before* `b=2` and `print b` *happens before* `print a`.

The reason is that even though we know the assignment `b=2` must somehow have been "executed before" the "print b"-statement it *does not* induce a *happens before* relation between those two events. Consequently one *can not* use transitivity to claim that the assignment `a=1` *happens before* the "print a"-statement.[2]

This is exactly what is meant in the specification: "If one goroutine executes a = 1; b = 2;, another might observe the updated value of b before the updated value of a". Note that this is what makes the Go memory model a *weak* memory model, as this is not the case in a *strong* memory model.

## 1.4 Concurrent environments

We will investigate memory models for environments where multiple processes are allowed to run concurrently and read and write to a shared set of variables. It tend to differ from system to system exactly how concurrent processes are actually executed. Two of the most common ways are:

- To let all processes be executed by a single core, having some scheduler to swap the processes in and out so that all of them gets time to execute.
- To have multiple cores to execute a subset of the processes in parallel, also with a scheduler.

In a single core environment one could in some sense apply the memory model of sequential consistency, but programmers are seldom developing

[2]Hypothetically, if also `b=2` *happens before* `print b`, then we could have used transitivity to claim that `a=1` *happens before* `print a`.

concurrent programs for such environments exclusively. Most often they are developing for any of those.

Another example of concurrent environments is distributed environments. In such instances can e.g. databases be regarded as shared memory. Even though this thesis focuses on the Go memory model, which generally assumes that the programs are run on one device, some of the concepts discussed here can be abstracted to hold for distributed environments as well.

## 1.5 Chapter overview

In Chapter 2 we make a structural operational semantics for the core parts of the Go language that we will discuss. In Chapter 3 we provide examples on how the semantics we made in Chapter 2 works. In Chapter 4 we prove that goroutine-local variables can share the same semantics as shared variables, and that data race free programs are sequentially consistent. In Chapter 5 we extend the semantics we made in Chapter 2 to work in a functional environment with closures. This extension also permits dynamic spawning of goroutines. In Chapter 6 we summarize what we have done, and point out possible directions for future work.

# Chapter 2

# Structural Operational Semantics

In this chapter we will formalize the *happens before* relation given by the Go memory model. To achieve this, we will use something known as structural operational semantics (SOS for short). An *operational semantics* is a semantics which states that the meaning of a program is determined by the *operations* it performs. That it is *structural* means that the operations a program performs is determined by the syntactic structure of the program. SOS is also known as *small step semantics*.

An SOS is given as a set of rules called *inference rules*. How these rules work will be detailed later, though they can be thought of as specifying an interpreter. After the rules are clear, we can use them to make proofs about programs. The report by Gordon Plotkin [18] can prove both as a good introduction, as well as reference, to SOS.

Even though we use Go as the inspiration for what we will model, it would be unnecessarily cumbersome to use the concrete syntax of Go. Instead we will use an abstract syntax that is much simpler. It should still be possible to fairly easily map a real Go program that is only using the selected features to a program using our abstract syntax.

## 2.1 Syntax

For the purpose of this thesis we will only need to focus on some parts of the Go Language. We will need some basic language features like `for`, `if`, functions, variables, and assignments. Furthermore we will need syntax for some Go-specific features like goroutines and channels.

We will use the following abstract syntax to define our rules:

| | | | |
|---|---|---|---|
| Program | | = | *stmts* |
| StatementList | *stmts* | = | *stmt* \| *stmt*; *stmts* |
| Statement | *stmt* | = | **for** *e* {*stmts*} \| **if** *e* {*stmts*} **else** {*stmts*} |
| | | \| | $e \mid \dot{x} = e \mid ch \leftarrow e$ |
| Expression | *e* | = | $v \mid e\ op\ e \mid \dot{x} \mid \leftarrow ch$ |
| Value | *v* | = | $\circ \mid m \mid b \mid ch$ |

The value $\circ$ is what we will call the *empty value*. It will be used to make empty statements as well. $ch \leftarrow e$ is a channel send, and $\leftarrow ch$ is a channel receive. Channels will be further described in Section 2.2.3.

### 2.1.1 Expressions

Expressions will always evaluate to a value. $\dot{x}$ are variables, and will evaluate to (one of) the value(s) "lastly" written to it. Sometimes we will also refer to $\dot{x}$ as a *write token*, since we will use it as token in *writes* (see Section 2.2.1). *op* can be taken to be a standard binary operation, e.g. $+, -, *, /, <, =$. $m$ is taken to be an integer, $b$ as a boolean, and *ch* as a reference to a channel.

### 2.1.2 Types

The type specification for Go is worthy of a study in itself, though in this thesis we will strip it down to less than the bare bones so we can since we will rather focus on aspects dealing with shared memory, which is a runtime aspect.

As in most other languages, every value is associated with a type, which is referred to as the type of the value. Common types that are also present in Go includes *integers*, *booleans*, *strings*, and arrays of these. Variables are commonly said to be of the type of the value it can store, and it is also common to associate a type to the result of an operation.

The type of a value is for the most part regarded in terms of what one can do with the value. Go is *statically typed*, which means that it is the task of the compiler to ensure that values are only used in ways that they can be used. For instance, an integer could be viewed as a value that can be e.g. added to another integer, while it is not part of being an integer that it should be possible to add a string to it.

In our semantics the variables should be thought of as typed. If a variable has a certain type, then only values of that type can be written to or read from this variable. Also, the only expressions that can appear in our programs are expressions where the values involved are only used like they can be used.

## 2.2 Approach to semantics

To understand shared variables in Go, one can profit from viewing the shared variables not only as a single point of storage. They can rather be viewed such that every goroutine has a buffer holding an instance of these variables, and that an update in one buffer can propagate to other buffers. Actually, we shall come to see that the picture is even more sophisticated than that, but this should prove as a good starting point.

The semantic rules does not to distinguish between goroutine-local variables and shared variables. Although it is not immediately clear why those two kinds of variables can share the same semantics, it is the case they can, and this will be elaborated on later (Chapter 4).

This semantics, which we will call the HB-semantics, will have another take on the semantics of a variable than one of the most traditional ones. Typically variables are regarded in the sense that there exist a mapping from the variable to its value, i.e. a variable is a location in which a value can be stored. A write is typically modeled such that it alters the mapping, and a read is typically modeled such that it applies the mapping on the variable to obtain its value.

We will only keep the idea that a variable is something that can be written a value to, and read a value from. However, in order to model the Go memory model in its purest sense, a novel take would prove necessary. Instead of making a write to alter a mapping, we are going to collect the write itself. Then, when we are doing a read, we will try to find the write that was "last" writing to the variable in question, and see what value it wrote.

### 2.2.1 Writes

Let's look at what I henceforth will call the *core rule of observable writes*. It is taken from the reference for the Go memory model [21].

> A read $r$ is *allowed* to observe a a write $w$ to $v$ if both of the following hold:
>
> 1. $r$ does not happen before $w$.
> 2. There is no other write $w'$ to $v$ that happens after $w$ but before $r$.

It will be assumed that the value obtained by a read can *only* be the value written from any write it is allowed to observe. We will not deal with the potential of out-of-thin-air results, although results like the ones described in [2] seems not to be accounted for in the Go memory model. This is actually a backside to the semantics described here, though, for the purposes of detecting data races, this semantics should still be fine.

If only one write can be observed from a read, then the read is *guaranteed* to observe that write.

So if we consider a read $r$ we have to account for two sources of writes that this read is allowed to observe, namely writes that happens before $r$, and writes that happens concurrently with $r$. We shall see in our analysis that these two sources behave differently. But first some definitions are due.

**Definition 2.1** (Concurrent)**.** If neither $e_1$ *happens before* $e_2$ nor $e_2$ *happens before* $e_1$, then we say $e_1$ *happens concurrently with* $e_2$ [21].

$e_1$ and $e_2$ should be taken to be *events*, e.g. execution of a statement.

**Definition 2.2** (Shadowed)**.** Given a read $r$ and two writes $w$ and $w'$, if $w$ *happens before* $w'$ and $w'$ *happens before* $r$, then we say that $w$ is shadowed by $w'$ (with respect to $r$).

With our definition of *shadowed* it is possible to restate the second part of the core rule of observable writes. That is: "There is no other write $w'$ such that $w$ is shadowed by $w'$."

Now, let's say we have a way to tag each write so that they all are uniquely identifiable. Then there are some things that makes sense to track for each goroutine. Given a read $r$ in a goroutine we want to know the following:

1. Which writes that happened before $r$ (all of them).
2. Which writes that are shadowed by other writes with respect to $r$.
3. Which writes that happens concurrently with $r$.

So, given a read $r$ we will refer to these sets as $\mathbf{H}(r)$, $\mathbf{S}(r)$, and $\mathbf{C}(r)$ respectively. Then the set of observable writes for read should be $(\mathbf{H}(r) \setminus \mathbf{S}(r)) \cup \mathbf{C}(r)$ (i.e. the writes that either have happened before the read but is not shadowed by other writes, or are concurrent with the read).

Instead of determining these sets when a read should be resolved, it is possible to continuously keep track of such sets so they are always up to date. Every goroutine could have a triplet of such sets associated with them. In doing so, they are no longer associated with a particular read.

Now, these sets will contain writes, and for each write we will want to know which variable that was written to, which value that was written to it, and also the tag of the write. The tag is meant to make the writes distinguishable. So we will model a write as a triple $\langle id, \dot{x}, v \rangle$, where these corresponds to tag, variable, and value respectively.

For convenience, let $\mathbf{W}$ be the set of all writes ever made so far by any goroutine. This set will be global in our model.

If we take $X$ to be a set of writes, we will make the notation that $X_{\dot{x}} = \{\langle id, \dot{x}', v \rangle \in X : \dot{x}' = \dot{x}\}$ (i.e. the subset of writes on a certain variable).

### 2.2.2 Goroutines

Goroutines will be modeled so that they each have an a program (i.e. list of statements – *stmts*), and a state $\sigma$. The state $\sigma$ will be a tuple with sets **H** and **S** that will behave as described above. A set **C** could be calculated by doing $\mathbf{W} \setminus \mathbf{H}$.

Even though states are *tuples* of sets, they will be operated under some of the set operations, like subset and union. In doing this, we will mean the element-wise operation on the elements (which are the sets) of the tuple. More precisely, if $\sigma = \langle \mathbf{H}, \mathbf{S} \rangle$ and $\sigma' = \langle \mathbf{H}', \mathbf{S}' \rangle$, then

- $\sigma \cup \sigma' = \langle \mathbf{H} \cup \mathbf{H}', \mathbf{S} \cup \mathbf{S}' \rangle$.
- $\sigma \subseteq \sigma' = \mathbf{H} \subseteq \mathbf{H}' \wedge \mathbf{S} \subseteq \mathbf{S}'$

Also if $\sigma = \langle \mathbf{H}, \mathbf{S} \rangle$ then $\sigma_{\dot{x}} = \langle \mathbf{H}_{\dot{x}}, \mathbf{S}_{\dot{x}} \rangle$

The sets **W**, **H**, and **S** correspond respectively to the sets "allWrites", "previous", and "overwritten" in [13]

We observe that **H** and **S** both must be subsets of **W**, so they must both be an element in $\mathscr{P}(\mathbf{W})$ ($\mathscr{P}$ denotes the power set of a set). Hence, states, which are tuples of these, must be an element of $\mathscr{P}(\mathbf{W})^2$.

In Go, the goroutines resides in a hierarchy in the sense that if a goroutine spawns another goroutine, then the spawner becomes the parent of the spawnee. If a parent terminates, then so will also all the other goroutines further down in the hierarchy. Nonetheless, in this description of the semantics, we will assume a flat structure. This is to focus on certain other aspects of *happens before*.

### 2.2.3 Channels

Every channel will be modeled as a tuple $\langle s, r, c, \mathscr{M}, \mathscr{V} \rangle$, where

- $s$ is the number of sends that have been performed.
- $r$ is the number of receives that have been performed.
- $c$ is the capacity. Unbuffered channels have capacity 0.
- $\mathscr{M} : \{-c, \ldots, 0\} \cup \mathbb{N} \to \mathscr{P}(\mathbf{W})^2$ is a mapping such that given $i$, if

  $r \leq i$: $\mathscr{M}$ is a mapping from the $i$-th send, to the state of the goroutine performing the send (at the time of the send).

  $i < r$: $\mathscr{M}$ is a mapping from the $i$-th receive, to the state of the goroutine performing the receive.

- $\mathscr{V}$ is the mapping from the $i$-th send, to the value that was actually sent on that send.

An illustration of a channel is given in Figure 2.1 on page 16. It will be explained in further detail later, but $\sigma_i$ and $v_i$ corresponds to $\mathcal{M}(i)$ and $\mathcal{V}(i)$ respectively. $r$ and $s$ will be initially be 0 when the channel is created. For $i < 0$, $\mathcal{M}(i) = \langle \emptyset, \emptyset \rangle$. This is to make the rules we make in Section 2.3.5 simpler.

We let $C$ be a mapping from channel identifiers (*id*s for short) to channels. In the semantics we will assume that these *id*s are referred to directly in the programs, though in a more thorough description there would be a layer of dereference from channel name in program to channel. $\mathcal{M}$ will initially map to tuples with only empty sets. Later we will use the notation $\mathcal{M}[i \mapsto \sigma]$ to mean a new mapping such that

$$\mathcal{M}[i \mapsto \sigma](x) = \begin{cases} \sigma & \text{if } x = i \\ \mathcal{M}(x) & \text{otherwise} \end{cases}$$

## 2.3 Inference rules

Inference rules are a tool to formalize semantics of programs. Each rule has the form

$$\textsc{Rule-name}\ \frac{\textit{Premises}}{\textit{Conclusion}}$$

A running program is always in some *state*. The state describes everything one would expect from the state of an executing program, like the values of the variables, and at what point the execution is in the program. The state of a program is an element of a *state space*, usually denoted $\Gamma$. This state space holds all possible states of the running program.

The inference rules describes the possible *transitions* between the states in $\Gamma$. There are other ways to describe such transitions than using inference rules, but in this work we will assume transitions to be solely described by these. The transitions makes a binary relation (usually irreflexive, sometimes also asymmetric) on $\Gamma$, and is commonly denoted $\rightarrow$. We say that a program in state $\gamma_1$ can transition (or step) to state $\gamma_2$ if $\gamma_1 \rightarrow \gamma_2$.

This constitutes a graph, where the possible states are the nodes, and the transitions are directed edges between them. The program execution can then be thought of as a chess piece on one of the nodes, and it can transition to other nodes if there are edges that indicates that this is possible.

The transitive closure of $\rightarrow$ is often denoted $\rightarrow^*$. $\gamma_1 \rightarrow^* \gamma_2$ thus means that a program in state $\gamma_1$ can get to state $\gamma_2$ by zero or more transition steps. The execution of a program will start at a specific state, which we can call $\gamma_s$. We say that the execution of a program can *reach* a state $\gamma$ if $\gamma_s \rightarrow^* \gamma$.

There are three different kinds of states to consider, namely those with no transitions from it, those with one transition from it, and those with more than one transitions from it. Given the program is in a state $\gamma$, and we let

$\Gamma_\gamma = \{\gamma' \in \Gamma : \gamma \to \gamma'\}$ (i.e. $\Gamma_\gamma$ is the set of states that $\gamma$ can transition to), then if $\Gamma_\gamma$ has

- no element, the execution will be forced to halt. $\gamma$ is to be considered a terminating state.
- exactly one element, the execution is forced to make the transition from $\gamma$ to the state in $\Gamma_\gamma$.
- more than one element, the execution can make the transition from $\gamma$ to any one of the states in $\Gamma_\gamma$. If there exist programs whose execution of the program can reach such a state, then we will say that the inference rules are *nondeterministic*. Inference rules for concurrent systems are usually nondeterministic, while for nonconcurrent systems they are usually not.

Now we have what we need to refine what a inference rule will look like. It will typically have a form where the conclusion has the form $\gamma \to \gamma'$, i.e.

$$\textsc{Rule-name}\ \frac{\textit{Premises}}{\gamma \to \gamma'}$$

This simply means that if the premises are true, then there is a transition from $\gamma$ to $\gamma'$. Such a transition exists if and only if there is a rule that permits it.

### 2.3.1 The states

Now it its time do define what the states in our semantics of the memory model look like. Actually, we have in large already defined what states will contain by starting to formalize writes, goroutines, channels, and syntax above. It is only a matter of putting it together. We will call the states in our semantics for *configurations*. This is to avoid confusion when we refer to other things as states.

The configurations will be put together of two parts, namely a global state part, and a part that will be managing the individual goroutines. These two parts will be separated by a triangle ($\triangleright$). This is actually just a tuple, but it is often relieving to break things up by introducing some different notation. As we will come to see, there will be plenty of other tuples with the standard tuple notation anyway.

Goroutines running in parallel could have the notation $g_1 \| \cdots \| g_n$, but to keep things short I will use the notation $g \| G$ where $g$ is thought to be any one of $g_1, \ldots, g_n$, running in parallel with the rest (which are implied by the big $G$).

### 2.3.2 Basic rules

Let's take a shot at what will be our first rule.

$$\textsc{Par-step}\ \frac{\gamma \rhd g \to \gamma' \rhd g'}{\gamma \rhd g \| G \to \gamma' \rhd g' \| G}$$

Here the the global parts are denoted $\gamma$, and $\gamma'$. What this rule says is that a goroutine $g$ that is running in parallel with other goroutines is allowed to take a step as if it was the only running goroutine. This rule actually makes it possible to make rules that only considers configurations with one goroutine, as this rule neatly extends those rules to work with configurations with more than one running goroutine.

When we later say that a goroutine $g$ takes a step, we mean that we can apply this rule to alter $g$ to $g'$.

For the next rules we will write goroutines as tuples $\langle stmts, \sigma \rangle$. The $\sigma$ has been described before, it's the local state of the goroutine. *stmts* will typically be altered when the goroutine takes a step. It is a list of statements that are yet to be executed.

The next rule is a little like the previous rule. After this rule is defined we can rather look at rules with only one statement.

$$\textsc{Seq}\ \frac{\gamma \rhd \langle stmt, \sigma \rangle \to \gamma' \rhd \langle stmts, \sigma' \rangle}{\gamma \rhd \langle stmt\mathbf{;}\ stmts^\star, \sigma \rangle \to \gamma' \rhd \langle stmts\mathbf{;}\ stmts^\star, \sigma' \rangle}$$

It is worth noticing that the $stmts^\star$ stays the same as before the step. The *stmt* is allowed to expand to more statements, as when the rule is used in combination with for instance IF-TRUE, IF-FALSE, or FOR-TRUE. It is also allowed to transform one statement to another. This is actually the common case.

Then we have a rule that describes when a program is allowed to proceed to the next statement. This is when the current statement is just a value, and in particular the value $\circ$.

$$\textsc{Skip}\ \frac{}{\gamma \rhd \langle v\mathbf{;}\ stmts^\star, \sigma \rangle \to \gamma \rhd \langle stmts^\star, \sigma \rangle}$$

### 2.3.3 Control structures

The rules here will not be used very much in the rest of the text, but there is one important aspect to them that needs to be mentioned all the same: They don't directly alter the global state $\gamma$, or the local state $\sigma$ of the goroutine, they only *allow* for an alternation, just in case this is needed for evaluating an expression that contains a channel receive. The fact that they themselves don't do any alternations will mean that these rules can safely be ignored in proofs we will do later in Chapter 4.

$$\textsc{If-eval-expr}\ \frac{\gamma \rhd \langle e, \sigma \rangle \to \gamma' \rhd \langle e', \sigma' \rangle}{\gamma \rhd \langle \mathbf{if}\ e\ \{stmts_1\}\ \mathbf{else}\ \{stmts_2\}, \sigma \rangle \to \gamma' \rhd \langle \mathbf{if}\ e'\ \{stmts_1\}\ \mathbf{else}\ \{stmts_2\}, \sigma' \rangle}$$

$$\textsc{If-true}\ \frac{v = \textsc{True}}{\gamma \rhd \langle \mathbf{if}\ v\ \{stmts_1\}\ \mathbf{else}\ \{stmts_2\}, \sigma \rangle \to \gamma \rhd \langle stmts_1, \sigma \rangle}$$

$$\textsc{For-prep}\ \frac{}{\gamma \rhd \langle \textbf{for}\ e\ \{stmts\}, \sigma\rangle \to \gamma \rhd \langle \underline{\textbf{for}}\ \langle e, e\rangle\ \{stmts\}, \sigma\rangle}$$

$$\textsc{For-eval-expr}\ \frac{\gamma \rhd \langle e_1, \sigma\rangle \to \gamma' \rhd \langle e_1', \sigma'\rangle}{\gamma \rhd \langle \underline{\textbf{for}}\ \langle e_1, e_2\rangle\ \{stmts\}, \sigma\rangle \to \gamma' \rhd \langle \underline{\textbf{for}}\ \langle e_1', e_2\rangle\ \{stmts\}, \sigma'\rangle}$$

$$\textsc{For-true}\ \frac{v = \textsc{True}}{\gamma \rhd \langle \underline{\textbf{for}}\ \langle v, e\rangle\ \{stmts\}, \sigma\rangle \to \gamma \rhd \langle stmts\textbf{; for}\ e\ \{stmts\}, \sigma\rangle}$$

$$\textsc{For-false}\ \frac{v = \textsc{False}}{\gamma \rhd \langle \underline{\textbf{for}}\ \langle v, e\rangle\ \{stmts\}, \sigma\rangle \to \gamma \rhd \langle \circ, \sigma\rangle}$$

The rules should be quite clear, so I won't give much explanation to them. The only thing I will elaborate on is the structure of the **for** statements. In order to maintain the original expression for an eventual later evaluation, we need to copy it. This is done by the rule FOR-PREP. So we have one copy to do the evaluation on, and one copy that should be maintained.

The **for** statement is our first example of something called *runtime syntax*. Runtime syntax is not allowed to be present in a starting configuration, but is allowed to be present in any other configuration. The **for** statement is not the same as the **for** statement, though they are related. Runtime syntax will continue to be presented as we go, and it can be recognized by an underline.

### 2.3.4 Reads and writes

$$\textsc{Write-eval-expr}\ \frac{\gamma \rhd \langle e, \sigma\rangle \to \gamma' \rhd \langle e', \sigma'\rangle}{\gamma \rhd \langle \dot{x} = e, \sigma\rangle \to \gamma' \rhd \langle \dot{x} = e', \sigma'\rangle}$$

This rule will eventually reduce the expression to a value. Similar rules will not be explicitly written out later, e.g. the SEND rule that is given later should be thought to have a corresponding SEND-EVAL-EXPR rule.

$$\textsc{Write}\ \frac{\begin{array}{cc} id = fresh & \sigma = \langle \mathbf{H}, \mathbf{S}\rangle \\ w = \langle id, \dot{x}, v\rangle & \sigma' = \langle \mathbf{H} \cup \{w\}, \mathbf{S} \cup \mathbf{H}_{\dot{x}}\rangle \end{array}}{\langle \mathbf{W}, C\rangle \rhd \langle \dot{x} = v, \sigma\rangle \to \langle \mathbf{W} \cup \{w\}, C\rangle \rhd \langle \circ, \sigma'\rangle}$$

This is a rule that assumes that the next instruction of a goroutine is an assignment to a variable $\dot{x}$. This rule suggests that in that case we construct a new write $w$ and put it among the other writes in $\mathbf{W}$, as outlined in Section 2.2.1. It also puts the write into the $\mathbf{H}$ of the current goroutine. This is natural, since this step is to be considered as happening before subsequent steps by this goroutine. As for the writes to $\dot{x}$ that happened before this step, they are in this goroutine henceforth shadowed by this write to $\dot{x}$, and are thus put into $\mathbf{S}$.

$$\textsc{Read}\ \frac{\sigma = \langle \mathbf{H}, \mathbf{S}\rangle \qquad \langle id, \dot{x}, v\rangle \in \mathbf{W}_{\dot{x}} \setminus \mathbf{S}_{\dot{x}}}{\langle \mathbf{W}, C\rangle \rhd \langle \dot{x}, \sigma\rangle \to \langle \mathbf{W}, C\rangle \rhd \langle v, \sigma\rangle}$$

When a variable is read, it can assume the value of any write to it that has not been shadowed. Later, in chapter 5, we will make rules to ensure that

there exists at least one such write to $\dot{x}$. In the case where there are more than one such write, this rule will contribute to induce nondeterministic behavior, as the rule is agnostic to which write it picks the value from.

This rule can be debated, since it is based on the aforementioned assumption of no out-of-thin-air results.

### 2.3.5 Channel communication

The rules of *happens before* for channel communication that we will focus on are:

(R1) A send on a channel *happens before* the corresponding receive from that channel completes.

(R2) The $k$-th receive on a channel with capacity $C$ *happens before* the $k + C$-th send from that channel completes.

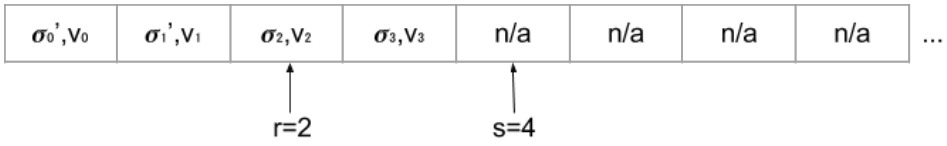


Figure 2.1: Illustration of a channel.

$$\textsc{Send}\ \frac{\begin{array}{c} C(ch) = \langle s, r, c, \mathcal{M}, \mathcal{V} \rangle \qquad s \leq c + r \\ \mathcal{M}' = \mathcal{M}[s \mapsto \sigma] \qquad \mathcal{V}' = \mathcal{V}[s \mapsto v] \\ C' = C[ch \mapsto \langle s+1, r, c, \mathcal{M}', \mathcal{V}' \rangle] \end{array}}{\langle \mathbf{W}, C \rangle \triangleright \langle ch \leftarrow v, \sigma \rangle \rightarrow \langle \mathbf{W}, C' \rangle \triangleright \langle \underline{\mathbf{pend}}\ \langle ch, s \rangle, \sigma \rangle}$$

This rule, and the following rules, are based on quite simple ideas, and still they may be quite daunting when typed out. The first thing to sort out is that this rule is for the situation when the next command is a send statement (i.e. $ch \leftarrow v$). If that is the case, and all of the premises are true, the statement is transformed to a **pend** statement, and the channel is updated.

The way the channel is updated is outlined by the premises. Note that this rule can only be applied if $s \leq c + r$, i.e. the channel has capacity to do the send (this is the only premise that should be read as a conditional, the rest should rather be read like steps of an algorithm).

Actually, the rule allows to put *one more* than the capacity allows, and this is where the **pend** statement comes into the picture. It puts the goroutine into a pending state. If the capacity wasn't overflowed by the send, the goroutine can step out of the pending state immediately. If the capacity was overflowed by the send (by one), the goroutine must wait for a receive. It will be as if the send hasn't really been performed yet. Both of these cases are modeled by the rule PEND later.

Note that both a value is sent, as well as the $\sigma$ with the *happens before*-set and the shadowing set. This is in accordance with (R1).

$$\textsc{Recv}\ \frac{\begin{array}{c} C(ch) = \langle s, r, c, \mathscr{M}, \mathcal{V} \rangle \qquad r < s \\ \mathscr{M}(r) = \sigma' \qquad \mathcal{V}(r) = v \qquad \mathscr{M}' = \mathscr{M}[r \mapsto \sigma] \\ C' = C[ch \mapsto \langle s, r+1, c, \mathscr{M}', \mathcal{V} \rangle] \end{array}}{\langle \mathbf{W}, C \rangle \rhd \langle \leftarrow ch, \sigma \rangle \rightarrow \langle \mathbf{W}, C' \rangle \rhd \langle v, \sigma \cup \sigma' \rangle}$$

This rule should be read in the same pattern as the rule SEND. Evidently $r$ must be less than $s$ for this rule to be applied, otherwise we would try to receive something that hasn't been sent yet.

In this rule we take the union of two $\sigma$'s, recall that this is defined to mean the element-wise union of the elements (see Section 2.2.2).

In the same breath the goroutine writes back it's *old* $\sigma$. This is in accordance with (R2), and this state can be picked up by a pending goroutine. It can be discussed whether the goroutine should rather send back it's *new* $\sigma$, the rules are actually not entirely clear on this.[1] The $\sigma$ that is written back is depicted in Figure 2.1 on the preceding page as a $\sigma$ with a mark.

$$\textsc{Pend}\ \frac{C(ch) = \langle \_, r, c, \mathscr{M}, \_ \rangle \qquad r > s - c \qquad \mathscr{M}(s-c) = \sigma'}{\langle \mathbf{W}, C \rangle \rhd \langle \underline{\mathbf{pend}}\ \langle ch, s \rangle, \sigma \rangle \rightarrow \langle \mathbf{W}, C' \rangle \rhd \langle \circ, \sigma \cup \sigma' \rangle}$$

In this rule we find that we must have $r > s - c$. This just means that the $s - c$-th element must have been received, this is to make sure that the receiving goroutine has written its $\sigma$ to $\mathscr{M}(s-c)$, since it is required that the *sending* goroutine is updated on what has happened before this point in the receiving goroutine. Note that the $s$ here is taken from the **pend** -statement, not the channel. It's actually possible for the $s$ of the channel to be much higher at the time when this rule is applied, as it doesn't have to be applied immediately when it can, but for instance after a lot of sends and receives on this channel by other goroutines.

Recall that if $s - c < 0$ then $\mathscr{M}(s-c) = \langle \emptyset, \emptyset \rangle$.

If we look at Figure 2.1 and imagine that it is a channel with capacity of one, then the buffer is currently overflowed. That means there *must* be a goroutine with a **pend** $\langle ch, 3 \rangle$ as its next statement. When another goroutine does a receive, $r$ in that channel becomes 3. Since now $r = 3 > s - c = 3 - 1 = 2$, the PEND rule may now be applied.

[1] The ambiguity lies in the phrasing. For instance is the *completion* of a send somehow *after*, or a part of the send? And should we take "a send *happens before*" to mean that the entire send *happens before*, or that the start of the send *happens before*? The rules are built on the assumption that the answers to the questions are "somehow after", and "the start of the send" respectively. Another set of rules would be necessary if the answers should be "part of", and "the entire send" instead.

# Chapter 3

# Examples

This chapter is provided as to give the reader a feeling for how the semantics works. We will provide a way to write configurations, and explanations as to how we use the rules to prove that one configuration can step to another. We will start by explaining how a simple Go program can be translated to a starting configuration, and afterwards provide three examples of programs. The three examples are explained on different levels of detail, the first one being most low-level, and the last being most high-level.

## 3.1 Introductory example

### 3.1.1 Translating to a starting configuration

The rules described so far are sufficient to describe the behavior of a lot of programs, if we just convert a program into an appropriate starting configuration. Since we do not yet have a means of spawning new goroutines, we must for the time being assume that the goroutines we need are already there in the starting configuration. Dynamic spawning of goroutines will be discussed in chapter 5, along with function calls, and more ado about variables. Until then, Figure 3.1 on the following page can give a tentative understanding of how programs can be translated.

According to the figure, the main function will not be a part of the starting configuration. However, we see that its purpose is to create two goroutines that are running in parallel. The translation will have these two goroutines in the configuration. Notice also that two writes have already been done; they reflect the writes that are done in the global scope.

### 3.1.2 New notation for goroutine states

Since it can be a little cumbersome to write out the entire local states (the $\sigma$'s), we can abbreviate it. For instance, imagine that the goroutine with

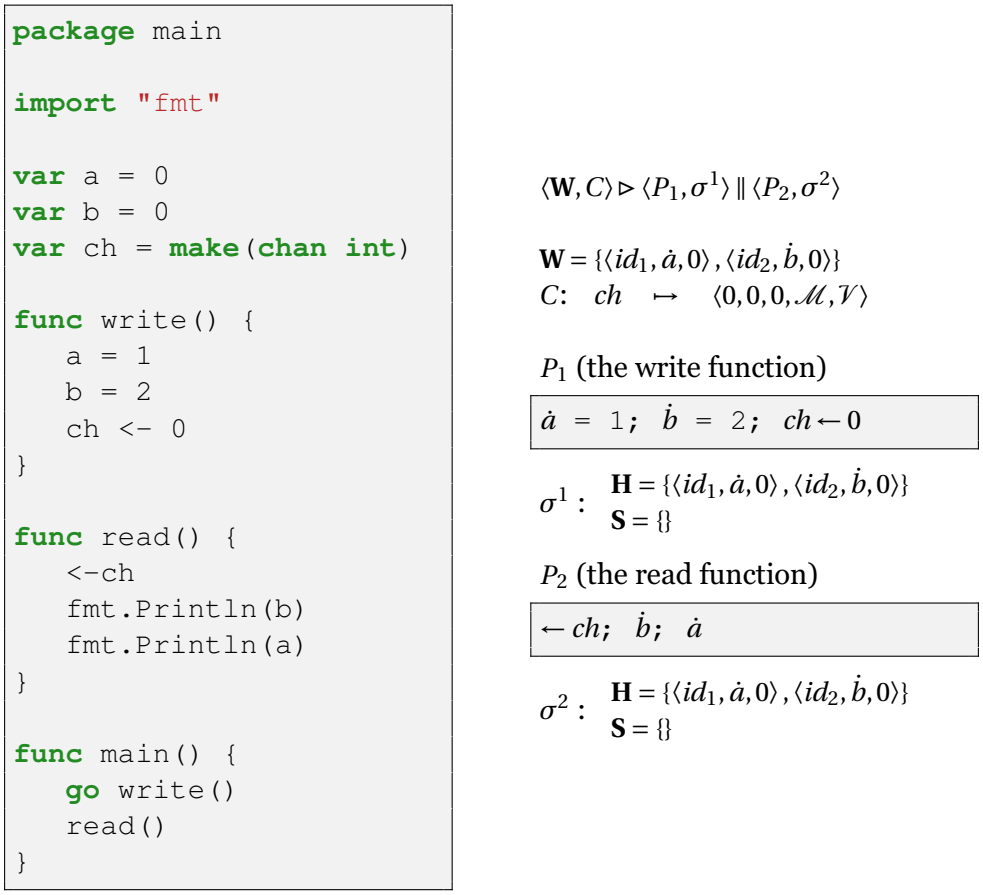


Figure 3.1: Translation of a program to a starting configuration

$\langle P_1, \sigma^1 \rangle$ (we will later refer to it as $g_1$, the other one as $g_2$) executes one WRITE-step. Then the state of the goroutine would change, as well as the **W**. We will go through the details later, but for now we will skip to the results.

The $\sigma^1$ would be transformed to

$$\sigma^{1\prime}: \begin{array}{l} \mathbf{H} = \{\langle id_1, \dot{a}, 0\rangle, \langle id_2, \dot{b}, 0\rangle, \langle id_3, \dot{a}, 1\rangle\} \\ \mathbf{S} = \{\langle id_1, \dot{a}, 0\rangle, \langle id_2, \dot{b}, 0\rangle\} \end{array}$$

The $\sigma^2$ will remain unchanged. The **W** would be transformed to

$$\mathbf{W}' = \{\langle id_1, \dot{a}, 0\rangle, \langle id_2, \dot{b}, 0\rangle, \langle id_3, \dot{a}, 1\rangle\}$$

Now, arguably the following representation for $\sigma^{1\prime}$ should be easier to read:

| | $\dot{a}$ | $\dot{b}$ |
|---|---|---|
| $\sigma^{1\prime}$ | 1 0 | 0 |

Each number correspond to the value of a write, while the columns tells which variable the write was to. The gray numbers correspond to the values of the writes in the **S** of the $\sigma$. If there are concurrent writes to a variable that is not in $\sigma$, these will be depicted in a parenthesis behind the writes

that are in $\sigma$. In addition, it is possible to have more than one row with $\sigma$'s. To illustrate, we will add in a row with $\sigma^2$:

| | $\dot{a}$ | $\dot{b}$ |
|---|---|---|
| $\sigma^{1\prime}$ | 1 0 | 0 |
| $\sigma^2$ | 0 (1) | 0 |

Notice how all elements of **W** are present in every row. If we have a row with $\sigma = \langle \mathbf{H}, \mathbf{S} \rangle$, what we can see from the row in terms of **H**, **S** and **W** is:

**W**: The entire row.

**H**: All elements that are not in parenthesis.

**S**: All elements that are gray.

**W** \ **S**: All black elements (including those in parenthesis). These are the *observable* writes. One can use **O** to denote such sets. We will do it on occasion in this chapter, and the reader can use it if the reader later needs to take notes in the margin of this thesis.

**W** \ **H**: All elements in parenthesis. These are the writes that are *concurrent*. Can be denoted by **C**.

**H** \ **S**: All black elements that are not in parenthesis. This set can be thought of as the *moment* of $\sigma$. One can use **M** to denote such sets.

When considering a cell for a variable $\dot{x}$, one will find that corresponding explanations works to understand the contents of the cell, e.g. $\mathbf{W}_{\dot{x}}$ is the entire cell, $\mathbf{S}_{\dot{x}}$ are all elements that are gray.

On the step we did above, we used a $'$ to mark that $\sigma^1$ and $\sigma^{1\prime}$ are not the same. If make a new change, this time on $\sigma^{1\prime}$, we would end up on $\sigma^{1\prime\prime}$ if we are to apply the same reasoning. After more changes there would become only more marks, and it would become tedious to count them in order to see which states are the same, and which are not. From this point, instead of writing $\sigma^{1\prime}$, we will write $\sigma^{1,1}$, and instead of $\sigma^{1\prime\prime}$, we will write $\sigma^{1,2}$. Hopefully the reader get the idea. The first number is the number of the goroutine, the second number is how many times the state of that goroutine has been changed. Similarly we will use $\mathbf{W}^n$ to denote the **W** after $n$ changes to it.

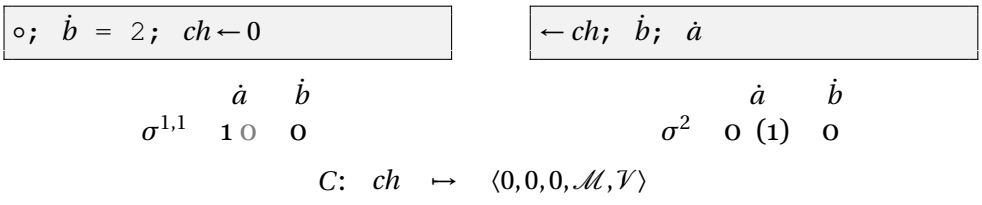


Figure 3.2: New notation of goroutine states.

We will write the new configuration as depicted by Figure 3.2. Everything that is a part of a configuration is present in this representation, though the mappings $\mathscr{M}$, and $\mathscr{V}$ are not yet written entirely out. Also, some information

is lost as to which writes that are corresponding to each other. This will be more evident later, when we are to take the union of two $\sigma$'s. We will deal with channels in a way similar to the $\sigma$'s later. The contents of $g_1$ is on the left side, the contents of $g_2$ on the right. The contents of the global state is in the middle on the bottom. Note that we have omitted explicit mention of the $\mathbf{W}$, as it would be redundant. After all, it is present in the row of a $\sigma$.

### 3.1.3 Reasoning about possible steps

In the previous section we said we let $g_1$ make a WRITE-step, but we did neither explain why we could do so with any reference to the inference rules, nor did we explain how we arrived at the result by applying the rules. We will do this now. We know that the configuration at the time when we would do the WRITE-step had the following form:

$$\langle \mathbf{W}, C\rangle \triangleright \langle \dot{a} = 1; \; stmts^\star, \sigma^1\rangle \,\|\, G \quad (1)$$

Here $G$ is just $g_2$. We will take a bottom-up approach, and we will use numbers (1)-(6) to indicate configurations that look identical. This is also the order we figure out what the configurations must look like. We know that

$$\textsc{Par-step}\frac{(2)\quad \langle \mathbf{W}, C\rangle \triangleright \langle \dot{a} = 1; \; stmts^\star, \sigma^1\rangle \to \gamma' \triangleright g_1'}{(1)\quad \langle \mathbf{W}, C\rangle \triangleright \langle \dot{a} = 1; \; stmts^\star, \sigma^1\rangle \,\|\, G \to \gamma' \triangleright g_1' \,\|\, G}$$

Further we know that

$$\textsc{Seq}\frac{(3)\quad \langle \mathbf{W}, C\rangle \triangleright \langle \dot{a} = 1, \sigma^1\rangle \to \gamma' \triangleright \langle stmts, \sigma^{1\prime}\rangle}{(2)\quad \langle \mathbf{W}, C\rangle \triangleright \langle \dot{a} = 1; \; stmts^\star, \sigma^1\rangle \to \gamma' \triangleright \langle stmts; \; stmts^\star, \sigma^{1\prime}\rangle}$$

Here we take $\sigma^{1\prime}$ to mean that the state may or may not change, we will know that later, and we reserve $\sigma^{1,1}$ to the occasion that we actually know we must make a new $\sigma$. Lastly, we know that

$$\textsc{Write}\frac{\sigma^1 = \langle \mathbf{H}, \mathbf{S}\rangle \qquad w = \langle id_3, \dot{a}, 1\rangle \qquad \sigma^{1,1} = \langle \mathbf{H} \cup \{w\}, \mathbf{S} \cup \mathbf{H}_{\dot{a}}\rangle}{(3)\quad \langle \mathbf{W}, C\rangle \triangleright \langle \dot{a} = 1, \sigma^1\rangle \to \langle \mathbf{W} \cup \{w\}, C\rangle \triangleright \langle \circ, \sigma^{1,1}\rangle \quad (4)}$$

Going down again, we see for the SEQ-rule what $\gamma'$, *stmts*, and $\sigma^{1\prime}$ must equate to (indeed, we need to replace $\sigma^{1\prime}$ by $\sigma^{1,1}$).

$$\textsc{Seq}\frac{(3)\quad \langle \mathbf{W}, C\rangle \triangleright \langle \dot{a} = 1, \sigma^1\rangle \to \langle \mathbf{W} \cup \{w\}, C\rangle \triangleright \langle \circ, \sigma^{1,1}\rangle \quad (4)}{(2)\quad \langle \mathbf{W}, C\rangle \triangleright \langle \dot{a} = 1; \; stmts^\star, \sigma^1\rangle \to \langle \mathbf{W} \cup \{w\}, C\rangle \triangleright \langle \circ; \; stmts^\star, \sigma^{1,1}\rangle \quad (5)}$$

Finally, the PAR-STEP-rule would be

$$\textsc{Par-step}\frac{(2)\quad \langle \mathbf{W}, C\rangle \triangleright \langle \dot{a} = 1; \; stmts^\star, \sigma^1\rangle \to \langle \mathbf{W} \cup \{w\}, C\rangle \triangleright \langle \circ; \; stmts^\star, \sigma^{1,1}\rangle \quad (5)}{(1)\quad \langle \mathbf{W}, C\rangle \triangleright \langle \dot{a} = 1; \; stmts^\star, \sigma^1\rangle \,\|\, G \to \langle \mathbf{W} \cup \{w\}, C\rangle \triangleright \langle \circ; \; stmts^\star, \sigma^{1,1}\rangle \,\|\, G \quad (6)}$$

Putting it all together we have the following proof that we can take the WRITE-step:

$$\textsc{Par-step}\ \frac{\textsc{Seq}\ \dfrac{\textsc{Write}\ \dfrac{\sigma^1 = \langle \mathbf{H}, \mathbf{S}\rangle \qquad w = \langle id_3, \dot{a}, 1\rangle \qquad \sigma^{1,1} = \langle \mathbf{H}\cup\{w\}, \mathbf{S}\cup\mathbf{H}_{\dot{a}}\rangle}{\langle \mathbf{W}, C\rangle \triangleright \langle \dot{a} = 1, \sigma^1\rangle \to \langle \mathbf{W}\cup\{w\}, C\rangle \triangleright \langle \circ, \sigma^{1,1}\rangle}}{\langle \mathbf{W}, C\rangle \triangleright \langle \dot{a} = 1\mathbf{;}\ stmts^\star, \sigma^1\rangle \to \langle \mathbf{W}\cup\{w\}, C\rangle \triangleright \langle \circ\mathbf{;}\ stmts^\star, \sigma^{1,1}\rangle}}{\langle \mathbf{W}, C\rangle \triangleright \langle \dot{a} = 1\mathbf{;}\ stmts^\star, \sigma^1\rangle \parallel G \to \langle \mathbf{W}\cup\{w\}, C\rangle \triangleright \langle \circ\mathbf{;}\ stmts^\star, \sigma^{1,1}\rangle \parallel G}$$

The effect of taking this step can be illustrated as

Before:

| | $\dot{a}$ | $\dot{b}$ |
|---|---|---|
| $\sigma^1$ | 0 | 0 |
| $\sigma^2$ | 0 | 0 |

After:

| | $\dot{a}$ | $\dot{b}$ |
|---|---|---|
| $\sigma^{1,1}$ | 1 0 | 0 |
| $\sigma^2$ | 0 (1) | 0 |

That is, for the $\sigma$ that is changed, all writes that were in $\mathbf{M}_{\dot{a}}$ will be moved into **S** (i.e. black writes that were not in parentheses are grayed out), and the new write is added to the **M**. For other goroutines, the new write is added to **C** (i.e. added to the writes in parentheses).

If we now wonder whether $g_2$ can now make a Recv-step, we have to see if we can make a proof that it can. The configuration looks like this

$$\langle \mathbf{W}, C\rangle \triangleright \langle \leftarrow ch\mathbf{;}\ stmts^\star, \sigma^2\rangle \parallel G \quad (1)$$

where this time $G$ is just $g_1$. Climbing up the Par-step-, and Seq-rule as we did in the demonstration of the Write-step, we arrive at

$$\langle \mathbf{W}, C\rangle \triangleright \langle \leftarrow ch, \sigma^2\rangle \quad (3)$$

However, $C(ch) = \langle s, r, c, \mathcal{M}, \mathcal{V}\rangle = \langle 0, 0, 0, \mathcal{M}, \mathcal{V}\rangle$, so $s = 0$, and $r = 0$. But the Recv-rule requires $r < s$, and since this condition is not fulfilled, we cannot prove that we can make a Recv-step. Thus, we can't.

We can rather use the following proof to make a Skip-step for $g_1$:

$$\textsc{Par-step}\ \frac{\textsc{Skip}\ \dfrac{}{\gamma \triangleright \langle \circ\mathbf{;}\ stmts^\star, \sigma^{1,1}\rangle \to \gamma \triangleright \langle stmts^\star, \sigma^{1,1}\rangle}}{\gamma \triangleright \langle \circ\mathbf{;}\ stmts^\star, \sigma^{1,1}\rangle \parallel G \to \gamma \triangleright \langle stmts^\star, \sigma^{1,1}\rangle \parallel G}$$

From this point on we will not provide full proofs of steps, the reader may rather refer to this section to see how it is done if the reader is uncertain that a step that is claimed to be possible is indeed so.

### 3.1.4 New notation for channels

If we let $g_1$ take another Write-step and subsequently another Skip-step, we will arrive at the following configuration:

| $ch \leftarrow 0$ |
|---|

| | $\dot{a}$ | $\dot{b}$ |
|---|---|---|
| $\sigma^{1,1}$ | 1 0 | 2 0 |

| $\leftarrow ch;\quad \dot{b};\quad \dot{a}$ |
|---|

| | $\dot{a}$ | $\dot{b}$ |
|---|---|---|
| $\sigma^2$ | 0 (1) | 0 (2) |

$$C\colon\ ch \ \mapsto\ \langle 0,0,0,\mathscr{M},\mathscr{V}\rangle$$

We will hereby use the following notation for channels:

| | s | r | c | 0 | 1 | 2 |
|---|---|---|---|---|---|---|
| $ch$ | 3 | 2 | 10 | $\sigma^{1,1}$,12 | $\sigma^{1,4}$,42 | $\sigma^{2,1}$,9 |

Again multiple lines can be used, this time in the case where there are more than one channel. The tuple at column $i$ corresponds to $\mathscr{M}(i),\mathscr{V}(i)$ of that channel. Values and states that are grayed out are to be considered as obtained. Hence we can interpret the cells in the following way:

$\sigma, v$: The $\sigma$ and $v$ were just posted from a *sending* goroutine.

$\sigma, v$: The $\sigma$ was just posted from a *receiving* goroutine. The value $v$ has been consumed by this goroutine.

$\sigma, v$ The $\sigma$ has been consumed by a *pending* goroutine.

As usual,$s$ is the index of the next cell that will be posted to by a sending goroutine, and $r$ is the index of the next cell that will be consumed by a receiving goroutine.

$g_1$ is allowed to do a SEND-step, and after that we will have the configuration depicted in Figure 3.3.

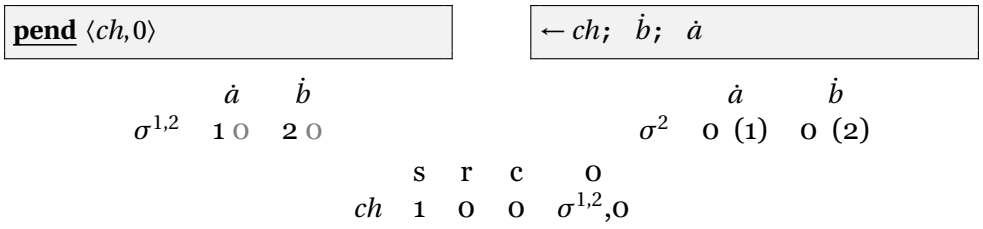


Figure 3.3: New notation for channels.

Now $g_2$ is at last allowed to do its RECV-step we get to the configuration in Figure 3.4. There are several things to notice in that picture. Firstly $\sigma^2$ is inserted to the channel, while at the same time $\sigma^2$ is replaced by $\sigma^{2,1} = \sigma^2 \cup \sigma^{1,2}$ as the state of $g_2$. We could refer to earlier figures to see what $\sigma^2$ is, but we have chosen to include it in the figure for completeness. When a state $\sigma$ is not a state of a particular goroutine, which writes that are in the **C** of that $\sigma$ becomes irrelevant, and we will thus omit it (i.e. omit the writes in the parentheses).

Another thing to notice is that it is not immediately clear from Figure 3.3 what the union of $\sigma^{1,2}$ and $\sigma^2$ would be, since, as mentioned, some information about which numbers that corresponds to the same write is lost in our notation. Anyway, the correct result is depicted.

We let $g_2$ do a skip and then the READ from $\dot{b}$. Note that $g_1$ can do a PEND-step, but as long as other goroutines also can do steps, it can't be forced by this semantics to make a step.

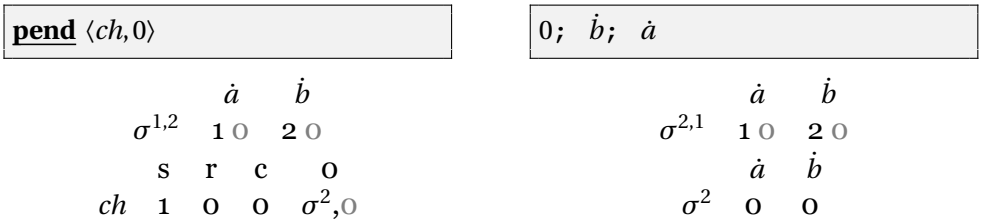


Figure 3.4: Configuration after RECV.

$g_2$ can only observe a write with value 2 associated with $\dot{b}$, so the READ-step is bound to transform $\dot{b}$ to this value. Hence, the new configuration would be that of Figure 3.5.

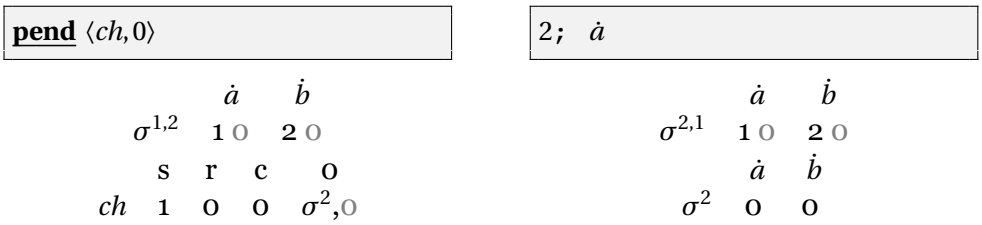


Figure 3.5: Configuration after READ of $\dot{b}$.

Now we let both goroutines run until they are finished. The order of the steps is irrelevant; we will arrive at the same final configuration for all cases. It is depicted in Figure 3.6.

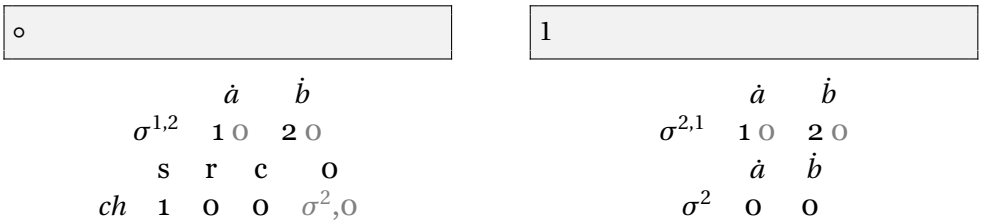


Figure 3.6: Final configuration of first example.

This was actually an example of a properly synchronized program.

## 3.2 Guard variables

In the days of strong memory models, a frequently applied technique was to use guard variables to ensure that other data could only be accessed when it was ready. Such techniques can still be applied under certain circumstances. We will examine one such pattern, and see if it can safely be applied in Go.

This time we will start directly from a starting configuration in our semantics, depicted in Figure 3.7.

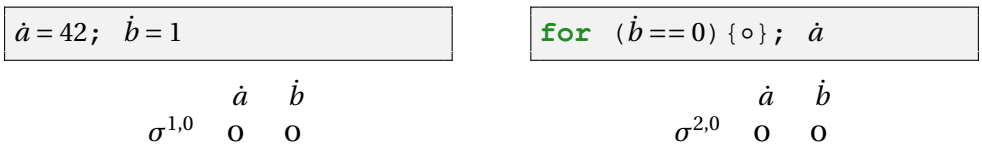


Figure 3.7: Starting configuration of guard example.

The purpose of this program is that the left goroutine, which we in the spirit of the previous example will refer to as $g_1$, should be able to write the value 42 to $\dot{a}$ before $\dot{a}$ is read by $g_2$. We want to see whether the read from $\dot{a}$ *must* evaluate to 42. In this example $\dot{b}$ plays the role of the guard variable.

Actually, this program can livelock under our semantics, as we have no means of preventing the **for**-statement from being the only goroutine that takes a step. Such a case would happen in a cycle depicted in Figure 3.8.

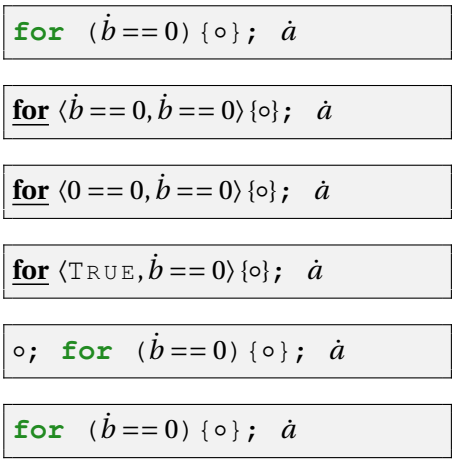


Figure 3.8: Steps of an infinite loop.

To get past this infinite loop, $g_2$ must be able to observe another value for $\dot{b}$ than 0. So we let $g_1$ run to and with the write on $\dot{b}$. We will also run $g_2$ to the point where it is about to read from $\dot{b}$. After this we get the configuration of Figure 3.9.

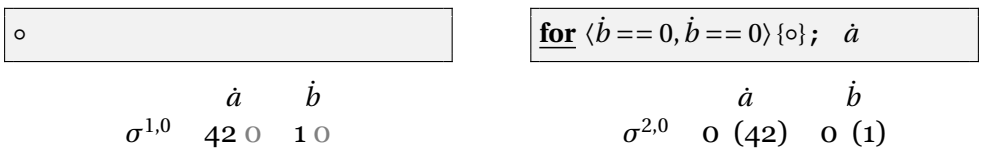


Figure 3.9: The guard variable is set.

When $g_2$ now read from $\dot{b}$, we see that both 0 and 42 is among its observable values, so it can evaluate to either. If it evaluates to 0, it will just cause another spin in the loop depicted in Figure 3.8. Although there is nothing in the semantics that stops this from going on forever, lets say that $g_2$ eventually reads 42, then the steps of Figure 3.10 will take place.

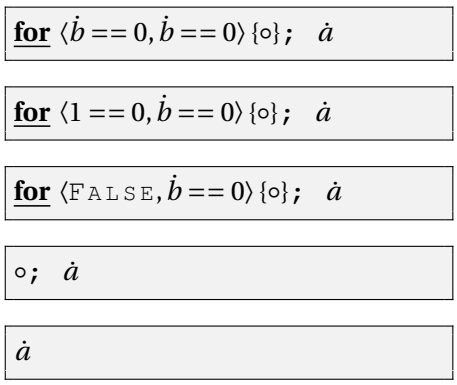


Figure 3.10: Exiting the loop.

Since the state of $g_2$ have not changed since Figure 3.9 we still have that

$$\begin{array}{cccc} & \dot{a} & & \dot{b} \\ \sigma^{2,0} & 0\ (42) & & 0\ (1) \end{array}$$

.

Since 0 is still among the observable values for $\dot{a}$, this variable can indeed evaluate to something else than 42. Bottom line seems to be that it is not safe to assume that synchronization by means of guard variables would work.

## 3.3 Lamport's Bakery Algorithm

In an article from 1974, Lamport reports a solution to the mutual exclusion problem, namely the Bakery Algorithm [10]. This algorithm is specified without any requirement for any other synchronization primitives, like locks or semaphores. Also, it does not require any central process to coordinate the other processes.

Nevertheless, the algorithm presented has certain requirements on the execution model, and is generally known to break under weak memory models. We will demonstrate that this will also be the case in our semantics. In this example we will make a full naive implementation, and then reduce it to a starting configuration that is easier to work with. This is also in part to touch on the limitations of what programs we can model. This is also the only the only full fledged algorithm we are going to have a look at.

### 3.3.1 Description of algorithm

When Lamport describes the algorithm, he pictures a bakery. The baker can only serve one customer at a time, so the customers must follow some kind of policy to know who will be up next. So, upon entering the bakery, a customer must stop at the doorstep to write a number on a ticket. She must

```
func bakery(i int) {
   for {
      choosing[i] = 1
      number[i] = 1 + max(number)
      choosing[i] = 0
      for j := 0; j < n; j++ {
         for choosing[j] != 0 {
         }
         for number[j] != 0 &&
            pairLess(number[j], j, number[i], i) {
         }
      }
      critical(i) // critical section here
      number[i] = 0
      // noncritical section here
   }
}
```

Listing 3.1: Bakery algorithm

do this by looking at the tickets of the other customers and write a number that is higher than every number she sees. If there are other customers on the doorstep, they will be writing as well, but it does not really matter what she sees on their tickets.

When the customer is finished writing the number, she will proceed into the bakery, where she will wait until it is her turn. She will know that it is her turn when she is the one in the bakery who has the lowest number on her ticket. In case there is two or more people who are tied on having the lowest number, they will use a predefined order among them, e.g. the person with the person with the lowest cell phone number goes first.

When a customer is served and leaves the bakery, her ticket becomes irrelevant.

In Lamport's algorithm, the customers are analogous to processes, and getting served by the baker is analogous to entering a critical section.

### 3.3.2 Naive implementation

The algorithm is seemingly nicely mapped to the Go function in listing 3.1. `choosing` and `number` are arrays with `n` elements. `i` is the index of the process. It should be a unique integer in the range [0,n) and is given to the goroutine when it is spawned. Note that the `pairLess` function evaluates if the pair `(number[j],j)` is less than the pair `(number[i],i)` (a pair $(a,b) < (c,d)$ if *either* $a < c$ *or* $a = c$ and $b < d$).

Even in theory there are several problems with this Go code:

1. We are not given that the statements within a goroutine are actually executed in the order given in the code [21].
2. Lack of synchronization (e.g. channel communication) actually allows a compiler to optimize away the spawning of the goroutines [21].
3. Even within a goroutine, a statement is not necessarily done with execution before the execution of another statement starts.

A potential issue could also have been that the different elements of an array would not act as a variable in itself. That could have meant that a writes to different elements in the array would have to be regarded as different writes to the array as a whole. However, that seems to not be the case as the specification states that each element of an array acts as a variable [23].

### 3.3.3 Run with semantics

We will show that the mutual exclusion property does not hold, not even in the special case where there are only two goroutines running. We will show this by showing that it is possible to reach a configuration where the next statement in both goroutines is a write to a variable which we will name `critical`.

Since we not yet have support for function calls, the calls to `max` and `pairLess` will have to be inlined, and we will let `choosing[0]`, `choosing[1]`, `number[0]`, and `number[1]` be represented by write tokens `c0`, `c1`, `n0`, and `n1` respectively. We will for aesthetic reasons omit the dot above these write tokens, as they have more than one letter.

Also, since there are in this case only two goroutines, some for-statements can be reduced to simpler statements. The trouble can be detected within the first iteration of the outer for-loop, so we omit this as well. The resulting starting configuration should correspond to that of Figure 3.11

```
c0 = 1
if n0 > n1
{ max0 = n0 }
{ max0 = n1 }
n0 = 1+max0
c0 = 0
for c1 != 0 {o}
for n1 != 0 && n1 <= n0
{o}
critical = 1
```

| | c0 | n0 | c1 | n1 |
|---|---|---|---|---|
| $\sigma^{1,0}$ | o | o | o | o |

```
c1 = 1
if n0 > n1
{ max1 = n0 }
{ max1 = n1 }
n1 = 1+max1
c1 = 0
for c0 != 0 {o}
for n0 != 0 && n0 < n1
{o}
critical = 2
```

| | c0 | n0 | c1 | n1 |
|---|---|---|---|---|
| $\sigma^{2,0}$ | o | o | o | o |

Figure 3.11: Initial configuration of the Bakery Algorithm.

If we first run the first (i.e. left) goroutine to the first `for` statement, and then do the same for the second goroutine we *can* (for instance if the goroutines only consider values from their **M**) obtain

```
for c1 != 0 {∘}
for n1 != 0 && n1 <= n0
{∘}
critical = 1
```

```
for c0 != 0 {∘}
for n0 != 0 && n0 < n1
{∘}
critical = 2
```

| | c0 | n0 | c1 | n1 |
|---|---|---|---|---|
| $\sigma^{1,4}$ | 0 1 0 | 1 0 | 0 (0 1) | 0 (1) |

| | c0 | n0 | c1 | n1 |
|---|---|---|---|---|
| $\sigma^{2,4}$ | 0 (0 1) | 0 (1) | 0 1 0 | 1 0 |

From this point there are no writes to any of the write tokens, so for any of the reads, the goroutines can pick any of the numbers given in their $\mathbf{W} \setminus \mathbf{S}$ (i.e. their set of observable writes **O**). If we let the first goroutine select 0-writes (i.e. write tuples where the value component is zero) when doing a READ on the write tokens `c1`, and `n1`. That would make the `for`-statements be reduced to nothing. If we do the same for `c0`, and `n0` in the second goroutine we find that the configuration is now

```
critical = 1
```

```
critical = 2
```

| | c0 | n0 | c1 | n1 |
|---|---|---|---|---|
| $\sigma^{1,4}$ | 0 1 0 | 1 0 | 0 (0 1) | 0 (1) |

| | c0 | n0 | c1 | n1 |
|---|---|---|---|---|
| $\sigma^{2,4}$ | 0 (0 1) | 0 (1) | 0 1 0 | 1 0 |

# Chapter 4

# Some Properties of the Model

Some properties are outlined below, along with proofs. Even though the rules were written with these properties in mind, the fact that they are true is not always trivial. Section 4.2 can be considered the crux of this thesis, as it proves that the semantics described herein is actually sound with respect to the semantics given by sequential consistency.

However, we will start a bit easier, by examining programs in which for any given variable only one goroutine will ever write to it. The intuitions collected by doing this will be useful when we later consider the general case.

## 4.1 Goroutine-local variables

These propositions in this section will build up to the fact that goroutine-local variables modeled in this way will exhibit the same behaviors as if they were modeled by a mapping from variables to values that is associated with the goroutine. This will be proven in Theorem 4.6.

**Lemma 4.1.** *If only one goroutine is ever writing to a variable $\dot{x}$, then all writes in $\mathbf{W}_{\dot{x}}$ are present in $\mathbf{H}_{\dot{x}}$ of that goroutine. The converse is also the case, so $\mathbf{W}_{\dot{x}} = \mathbf{H}_{\dot{x}}$*

*Proof.* The only rule that adds writes to **W** is WRITE. If we assume the invariant is initially true, we only have to check that it is still true after applying that rule. As we can clearly see, the write is also added to **H** of that goroutine. No rule removes writes from either set.

The converse holds trivially, as **W** will hold *all* writes. □

**Lemma 4.2.** *If an invariant holds for the $\sigma$'s in all goroutines, then given a channel, the same invariant must hold for the all $\sigma$'s from $0$ to $s-1$ (inclusive) in the $\mathscr{M}$ of that channel.*

*Proof.* The lemma is vacuously true when there has been no send to the channel, since in that case $s = 0$. The only rule that alters $s$ is SEND. Given that the lemma is true when $s = k$ we see that after applying the SEND rule the lemma is also true when $s = k + 1$, since the rule inserts a $\sigma$ where the invariant holds into $\mathscr{M}(k)$.

The only other rule that can alter the $\mathscr{M}$ of the channel in the interval in question is RECV. Since this rule also only inserts $\sigma$'s where the invariant holds, this lemma must be true. □

The reader should notice that since the choice of channel in the preceding lemma is arbitrary, the lemma holds for *all* channels.

**Proposition 4.3.** *If only one goroutine $g$ is ever writing to a variable $\dot{x}$, then the following three invariants holds:*

1. *For all goroutines, either $\mathbf{H}_{\dot{x}} \setminus \mathbf{S}_{\dot{x}}$ contains only one element, or $\mathbf{H}_{\dot{x}} = \emptyset \wedge \mathbf{S}_{\dot{x}} = \emptyset$.*
2. *The $\sigma_{\dot{x}}$ of any goroutine is a subset of the $\sigma_{\dot{x}}$ of $g$ (by subset we mean element-wise subset, see Section 2.2.2).*
3. *Given a $\sigma$ in a goroutine, for every other $\sigma'$ (i.e. $\sigma$'s in other goroutines) we have either $\sigma_{\dot{x}} \subseteq \sigma'_{\dot{x}}$ or $\sigma'_{\dot{x}} \subseteq \sigma_{\dot{x}}$.*

*Proof.* This proposition builds on the assumption that $\mathbf{H}_{\dot{x}} = \emptyset \wedge \mathbf{S}_{\dot{x}} = \emptyset$ is initially true in all goroutines, in which case we can verify that all three invariants are initially true. We examine the rules that can alter the $\sigma$'s of the goroutines:

WRITE After this rule is applied $\mathbf{S}'_{\dot{x}}$ will be the same as $\mathbf{H}_{\dot{x}}$, and $\mathbf{H}'_{\dot{x}}$ will be the same as $\mathbf{H}_{\dot{x}}$ with one more write. Hence, after the rule is applied $\mathbf{H}'_{\dot{x}} \setminus \mathbf{S}'_{\dot{x}}$ will contain one element, and the first invariant is still true.

The second invariant will also still be true, as the only $\sigma$ that had added to its elements is the $\sigma$ of $g$.

The third invariant is maintained, as the subset relation between the $\sigma$'s in the other goroutines are not altered.

RECV If this rule can be applied (i.e. $r < s$), we know that the $\sigma' = \mathscr{M}(r)$ must keep the three invariants, as according to Lemma 4.2. Since the $\sigma$ of the goroutine doing the receive also holds the invariants, $\sigma_{\dot{x}} \cup \sigma'_{\dot{x}}$ is simply the bigger of the two (one of them must be a subset of the other). Thus the invariants still holds. Note that if $\sigma$ is belonging to $g$, then $\sigma_{\dot{x}}$ will be the bigger one (that applies to PEND as well).

PEND Let $\sigma' = \mathscr{M}(s-c)$. If $s-c \geq 0$ the invariants holds for $\sigma'$ because of Lemma 4.2. If $s-c < 0$ then $\sigma' = \langle \emptyset, \emptyset \rangle$, and also in this case the invariants holds for $\sigma'$. Again $\sigma_{\dot{x}} \cup \sigma'_{\dot{x}}$ is just the bigger of the two, so the invariants are maintained.

□

Note that the following theorem is dealing with **W**, not **H** as the preceding proposition do.

**Theorem 4.4.** *If only one goroutine is ever writing to a variable $\dot{x}$, then $\mathbf{W}_{\dot{x}} \setminus \mathbf{S}_{\dot{x}}$ has invariantly one element in that goroutine (from the first time $\dot{x}$ is written to).*

*Proof.* From Lemma 4.1 we know that $\mathbf{W}_{\dot{x}} = \mathbf{H}_{\dot{x}}$. Since $\mathbf{W}_{\dot{x}}$ contains all writes to $\dot{x}$ neither set can be empty after the first write to $\dot{x}$. Hence, according to the first invariant of Proposition 4.3, $\mathbf{H}_{\dot{x}} \setminus \mathbf{S}_{\dot{x}}$ must contain exactly one element, and so must $\mathbf{W}_{\dot{x}} \setminus \mathbf{S}_{\dot{x}}$. □

**Definition 4.5** (Goroutine-local variable)**.** A goroutine-local variable is a variable that can only ever be written to or read from by a single goroutine. It is said to belong to this goroutine. It must be written to at least once before it can be read from.

**Theorem 4.6.** *Given a goroutine-local variable $\dot{x}$, assume a varying function $V(\dot{x})$ that always give the value that is lastly written to $\dot{x}$ (i.e. whenever the goroutine writes a value $v$ to $\dot{x}$, $V$ is altered to $V[\dot{x} \mapsto v]$). Then, whenever the goroutine does a read from $\dot{x}$ it will find $\langle id, \dot{x}, v \rangle$ as the only element of $\mathbf{W}_{\dot{x}} \setminus \mathbf{S}_{\dot{x}}$, and $v = V(\dot{x})$.*

*Proof.* The only rule that alters $\mathbf{W}_{\dot{x}}$ and $\mathbf{S}_{\dot{x}}$ of the goroutine is the WRITE rule. The fact that neither the RECV or the PEND alters $\mathbf{S}_{\dot{x}}$ of this goroutine is remarked at the end of the RECV part of the proof of Proposition 4.3. Let $w = \langle id, \dot{x}, v \rangle$ be the write we put into **W** when applying the WRITE rule, and it will not be put into **S**. At this point the theorem is trivially true. $\mathbf{W}_{\dot{x}} \setminus \mathbf{S}_{\dot{x}} = \{w\}$ will be invariant until the next WRITE, and so will $V(\dot{x}) = v$. □

## 4.2 Data race free programs

We will in this section show that if a program is data race free, it would be sequentially consistent, and we could use another semantics, using the following rules instead of the previously defined rules WRITE, and READ. This other semantics, which I will call the SC-semantics, is similar to the HB-semantics, but instead of keeping track of the *writes* (i.e. **W** and local states $\sigma$'s), it only keeps track of a global mapping $V$ from variables to values. The SC-semantics corresponds to what one would expect of a *strong* memory model.

$$\textsc{Write-SC}\frac{V' = V[\dot{x} \mapsto v]}{\langle V, C\rangle \triangleright \langle \dot{x} = v\rangle \rightarrow \langle V', C\rangle \triangleright \langle \circ\rangle}$$

$$\textsc{Read-SC}\frac{V(\dot{x}) = v}{\langle V, C\rangle \triangleright \langle \dot{x}\rangle \rightarrow \langle V, C\rangle \triangleright \langle v\rangle}$$

The other rules can be translated to rules in this semantics by omitting the *write*-tracking sets from the configurations. We can use the HB-semantics to show properties in the SC-semantics and vice versa. In this thesis we will show that if a program is data race free in the sense of the HB-semantics, it will also be data race free in the sense of the SC-semantics. We will define this clearer later, in Section 4.2.1, and Section 4.2.3.

The main difference between the two semantics is that the rule READ-SC is always deterministic, that is, whenever we apply the rule READ-SC there is only one possible value we can obtain from using the rule.

Configuration in the HB-semantics will be denoted $\gamma^{\text{HB}}$, and for the SC-semantics they will be denoted $\gamma^{\text{SC}}$. We will take them to be elements of the respective state spaces $\text{SPACE}_{\text{HB}}$, and $\text{SPACE}_{\text{SC}}$. Additionally, we will refer to executions under the different semantics as HB-executions, resp. SC-executions.

In order to compare HB-executions to SC-executions, we will use *bisimulation*. We will define this notion in Section 4.2.2. This notion of simulation was pioneered by Robin Milner [14, 15], where he describes under which conditions one system can be used to simulate another system. Initially he used bisimulation to define an equivalence relation on single process programs, such that programs in the same equivalence class could be said to be running the same algorithm. Later the notion was extended to cover programs with more processes.

In the following propositions, conditions that are labeled with HB, resp. SC are thought to be predicates on configurations under the HB-semantics, resp. SC-semantics. Given a configuration and a condition $C$, we use $C(\gamma)$ to mean that the condition is satisfied on the configuration. We will further use the notation $C^*(\gamma)$ to mean that there exists a configuration that can be reached from gamma that satisfies the condition, i.e. $\exists\gamma'(\gamma \rightarrow^* \gamma' \wedge C(\gamma'))$.

We will in this section also introduce the notation $\mathbf{H}(\sigma)$, and $\mathbf{S}(\sigma)$ to denote the $\mathbf{H}$ and $\mathbf{S}$ of a given $\sigma$.

### 4.2.1 Data race in HB-semantics

The Go reference [3] defines a data race to be when a write to a variable is concurrent with either another write or a read on the same variable. We will translate this to the following definition:

**Definition 4.7** (HB-race condition)**.** A program (i.e. starting configuration) has a *data race* if it can reach a configuration that satisfies one of the following:

$WW_{\text{HB}}$ A goroutine can perform a write on $\dot{x}$, but $\mathbf{W}_{\dot{x}} \setminus \mathbf{H}_{\dot{x}}$ already has one or more elements. This is a write-write conflict.

$RW_{\text{HB}}$ A goroutine can perform a read from $\dot{x}$, but $\mathbf{W}_{\dot{x}} \setminus \mathbf{H}_{\dot{x}}$ already has one or more elements. This is a write-read conflict.

A program without any data races will be referred to as *data race free*. If a starting configuration $\gamma$ is data race free, we will use the notation $\text{DRF}^*_{\text{HB}}(\gamma)$ to denote this.

Note that we in this definition implicitly defined the predicates $WW_{\text{HB}}$, $RW_{\text{HB}}$, $WW^*_{\text{HB}}$, and $RW^*_{\text{HB}}$. Note also that $\text{DRF}^*_{\text{HB}}(\gamma) = \neg WW^*_{\text{HB}}(\gamma) \wedge \neg RW^*_{\text{HB}}(\gamma)$.

We will examine configurations where $WW_{\text{HB}}$ is not satisfied. It should be fairly easy to see from the definition that this is the case if and only if for any goroutine that can perform a WRITE on $\dot{x}$, we have $\mathbf{H}_{\dot{x}} = \mathbf{W}_{\dot{x}}$. This works similarly for $RW_{\text{HB}}$.

**Definition 4.8** (Ambiguous read)**.** A program can perform an ambiguous read if it can reach a configuration that satisfies the following:

$R_{AMB}$ A goroutine can perform a read from $\dot{x}$, but $\mathbf{W}_{\dot{x}} \setminus \mathbf{S}_{\dot{x}}$ has two or more elements.

Note that the following lemma speaks about *strict* subsets.

**Lemma 4.9.** *For all goroutines* $\mathbf{S} \subset \mathbf{H}$ *is invariant.*

*Proof.* The only three rules that alters **W**, **H**, and **S** for the goroutine are WRITE, RECV, and PEND. If we assume the invariant is initially true, we have to verify that it still holds after applying any of these.

For the WRITE rule, the invariant holds trivially, since the only elements added to **S** are elements that are already in **H**. For the RECV and PEND it suffices to keep in mind that the $\sigma$ attained from the channel must also be from a goroutine where the invariant also was initially true (Lemma 4.2). If $\mathbf{S} \subset \mathbf{H}$ in both $\sigma$ and $\sigma'$, then that must also be the case in the union. □

**Lemma 4.10.** *Given a configuration* $\gamma^{\text{HB}}$*, and assuming that* $\neg WW^*_{\text{HB}}(\gamma^{\text{HB}})$*, then we have the following invariants on all* $\sigma$*'s in all configurations of* $\text{reach}(\gamma^{\text{HB}})$*:*

1. $|\mathbf{H}_{\dot{x}}(\sigma) \setminus \mathbf{S}_{\dot{x}}(\sigma)| = 1$.
2. *For every other* $\sigma'$*, we have either* $\sigma_{\dot{x}} \subseteq \sigma'_{\dot{x}}$ *or* $\sigma'_{\dot{x}} \subseteq \sigma_{\dot{x}}$.

*Proof.* We will prove these invariants by induction:

WRITE If the first invariant is true, it will continue to be true, analogous to the proof of the first invariant of Proposition 4.3.

Let $\sigma^g$ be the state of the goroutine performing the write. Because of our assumption that $WW_{\text{HB}}$ is not satisfied, we must have that $\mathbf{W}_{\dot{x}} = \mathbf{H}_{\dot{x}}(\sigma^g)$. Under the assumption that the first invariant is true, we have that for any other $\sigma'$, no $\mathbf{S}_{\dot{x}}(\sigma')$ can have any more elements than

$\mathbf{S}_{\dot{x}}(\sigma^g)$. This means that $\sigma'_{\dot{x}} \subseteq \sigma^g_{\dot{x}}$ (after all, according to the second invariant, one must be a subset of the other), and the rest of the proof can be done analogously to the proof of WRITE for the first and second invariants in Proposition 4.3.

RECV The proof of the first and second invariants are analogous to the proof of RECV for respectively the first and second invariants in Proposition 4.3.

PEND Analogous to RECV.

□

**Corollary 4.11.** *Given a configuration $\gamma^{\text{HB}}$, and assuming that $\neg WW^*_{\text{HB}}(\gamma^{\text{HB}})$, then we have the following invariant on all $\sigma$'s in all configurations of* $\text{reach}(\gamma^{\text{HB}})$*:*

$|\mathbf{W}_{\dot{x}} \setminus \mathbf{S}_{\dot{x}}(\sigma)| \geq 2$ *implies* $|\mathbf{W}_{\dot{x}} \setminus \mathbf{H}_{\dot{x}}(\sigma)| \geq 1$
*(contrapositive:* $|\mathbf{W}_{\dot{x}} \setminus \mathbf{H}_{\dot{x}}(\sigma)| = 0$ *implies* $|\mathbf{W}_{\dot{x}} \setminus \mathbf{S}_{\dot{x}}(\sigma)| = 1$*).*

*Proof.* We can prove the contrapositive of the invariant from the first invariant of Lemma 4.10. If $|\mathbf{W}_{\dot{x}} \setminus \mathbf{H}_{\dot{x}}(\sigma)| = 0$, then $\mathbf{W}_{\dot{x}} = \mathbf{H}_{\dot{x}}(\sigma)$. Substituting $\mathbf{W}_{\dot{x}}$ into the first invariant of Lemma 4.10, we see that we must have $|\mathbf{W}_{\dot{x}} \setminus \mathbf{S}_{\dot{x}}(\sigma)| = 1$. □

**Theorem 4.12.** *A program $\gamma^{\text{HB}}$ can perform an ambiguous read only if it has a data race (i.e. $R^*_{AMB}(\gamma^{\text{HB}}) \Rightarrow \neg\text{DRF}^*_{\text{HB}}(\gamma^{\text{HB}})$)*

*Proof.* Assume that $\neg WW^*_{\text{HB}}(\gamma^{\text{HB}})$, then by Corollary 4.11 we know that $|\mathbf{W}_{\dot{x}} \setminus \mathbf{S}_{\dot{x}}(\sigma)| \geq 2$ implies $|\mathbf{W}_{\dot{x}} \setminus \mathbf{H}_{\dot{x}}(\sigma)| \geq 1$. So under this assumption, this means that any configuration in $\text{reach}(\gamma^{\text{HB}})$ that satisfies $R_{AMB}$ will also satisfy $RW_{\text{HB}}$. The only other option is that the assumption is false, but this immediately implies a data race. □

### 4.2.2 Bisimulation

We proceed by making a definition of *bisimulation* that fits our purposes, and that is in spirit of traditional definitions. It will help us to compare programs that are running under different semantics. We can take $X$ and $Y$ to be state spaces, and $\rightarrow_X$ and $\rightarrow_Y$ to be their respective transition relations.

**Definition 4.13** (Bisimulation)**.** Given two systems $(X, \rightarrow_X)$, $(Y, \rightarrow_Y)$, a relation $R$ on $X \times Y$ is a *bisimulation* if $x\ R\ y$ implies both of the following:

1. $\forall x'(x \rightarrow_X x' \Rightarrow \exists y'(y \rightarrow_Y y' \wedge x'\ R\ y'))$
2. $\forall y'(y \rightarrow_Y y' \Rightarrow \exists x'(x \rightarrow_X x' \wedge x'\ R\ y'))$

The preceding definition says that if $R$ is a bisimulation, and given two configurations $x, y$ such that $x\ R\ y$, then whichever step that can be taken from the configuration $x$, there exist a similar step that can be taken from

$y$, such that the new configurations are still related, and vice versa. The following lemma just extends this by stating that they can follow each other in this manner any number of steps.

**Lemma 4.14.** *Given two systems* $(X, \to_X)$, $(Y, \to_Y)$, *and a bisimulation* $R$ *on* $X \times Y$, *if* $x\ R\ y$ *then both of the following holds:*

1. $\forall x'(x \to_X^* x' \Rightarrow \exists y'(y \to_Y^* y' \wedge x'\ R\ y'))$
2. $\forall y'(y \to_Y^* y' \Rightarrow \exists x'(x \to_X^* x' \wedge x'\ R\ y'))$

*Proof.* Simple induction. □

**Definition 4.15** (Bisimilar)**.** Given two systems $(X, \to_X)$, $(Y, \to_Y)$, then $x \in X, y \in Y$ are *bisimilar* if there exist a bisimulation $R$ such that $x\ R\ y$. We write $x \sim y$ for this.

If we regard $x$ and $y$ in the above definition as program states, the intuition is that if $x \sim y$, these states are kind of equal to each other. We will not use this definition in any serious proofs, but we will leave still leave it here for completion, and since it is interesting in itself.

For the following theorems we will define a binary relation $R$ on $\text{SPACE}_{\text{HB}} \times \text{SPACE}_{\text{SC}}$ such that if $\gamma^{\text{HB}} = \langle \mathbf{W}, C^{\text{HB}} \rangle \rhd G^{\text{HB}}$ and $\gamma^{\text{SC}} = \langle V, C^{\text{SC}} \rangle \rhd G^{\text{SC}}$, then $\gamma^{\text{HB}}\ R\ \gamma^{\text{SC}}$ if $\neg R^*_{AMB}(\gamma^{\text{HB}})$ *and* all of the following relations holds:

$R_G$ They have equally many goroutines which are running the same programs as each other.

$R_C$ The channels in $C^{\text{HB}}$ are corresponding to the channels in $C^{\text{SC}}$.

$R_V$ For all $\dot{x}$, the value of $\dot{x}$ in $V$ is in a write to $\dot{x}$ in $\mathbf{W}$ that is visible to all goroutines in $\gamma^{\text{HB}}$, i.e.

$$\forall \dot{x} \in \text{Dom}(V)(V(\dot{x}) = v \to \exists w \in \mathbf{W}(\text{Val}(w) = v \wedge w \notin \bigcup_{g \in G^{\text{HB}}} \mathbf{S}_{\dot{x}}(\sigma^g))).$$

**Theorem 4.16.** *$R$ is a bisimulation of* $(\text{SPACE}_{\text{HB}}, \to_{\text{HB}})$ *and* $(\text{SPACE}_{\text{SC}}, \to_{\text{SC}})$.

*Proof.* Assume $\gamma^{\text{HB}}\ R\ \gamma^{\text{SC}}$. We must show that for any step $\gamma^{\text{HB}}$ may take, $\gamma^{\text{SC}}$ can take a similar step, such that the two new configurations are still related by $R$. Similarly, we have to show that for any step $\gamma^{\text{SC}}$ can make, $\gamma^{\text{HB}}$ can make a similar step.

We will use $\to$ to mean $\to_{\text{HB}}$ or $\to_{\text{SC}}$, and we will take it to be evident from the context which that is meant.

If $\gamma^{\text{HB}} \to \gamma^{\text{HB}\prime}$ and the transition is not a WRITE-, READ-, RECV-, or PEND-step, we can apply a similar step from $\gamma^{\text{SC}}$ to obtain a $\gamma^{\text{SC}\prime}$ such that $\gamma^{\text{HB}\prime}\ R\ \gamma^{\text{SC}\prime}$. This applies the other way as well.

WRITE $\gamma^{\text{HB}} \to \gamma^{\text{HB}\prime}$: The new write $w$ with value $v$ to $\dot{x}$ is not in $\mathbf{S}_{\dot{x}}$ of any of the goroutines. When we make the corresponding WRITE-step $\gamma^{\text{SC}} \to \gamma^{\text{SC}\prime}$, we obtain $V'(\dot{x}) = V[\dot{x} \mapsto v](\dot{x}) = v$, and thus $\gamma^{\text{HB}\prime}\ R\ \gamma^{\text{SC}\prime}$. This also applies the other way around.

READ $\gamma^{\text{HB}} \to \gamma^{\text{HB}\prime}$: Let $g$ be the goroutine taking the step, $\dot{x}$ be the variable written to, $V$ be the variable map of $\gamma^{\text{SC}}$, and $V(\dot{x}) = v$. Since, by the definition of $R$, we have that $\neg R^*_{AMB}(\gamma^{\text{HB}})$, we must have that there is only one write in $\mathbf{W} \setminus \mathbf{S}(\sigma^g)$, and this is required by the relation $R$ to have the value $v$. So we have for the corresponding READ-step $\gamma^{\text{SC}} \to \gamma^{\text{SC}\prime}$ that $\gamma^{\text{HB}\prime}\, R\, \gamma^{\text{SC}\prime}$.

Note that if the requirement that $\neg R^*_{AMB}(\gamma^{\text{HB}})$ was not a part of the definition of $R$, then $R$ would not be necessarily be a bisimulation, since if $R_{AMB}(\gamma^{\text{HB}})$, there would be two or more writes in $\mathbf{W} \setminus \mathbf{S}(\sigma^g)$, and only one of them would *required* by the relation $R$ to have the value $v$, while one of the other writes could have another value. [1] On the other hand, the corresponding READ-step $\gamma^{\text{SC}} \to \gamma^{\text{SC}\prime}$ *must* obtain the value $v$.

RECV Here we only have to confirm that the corresponding steps does not alter the status of the relation $R_V$. For any $\dot{x}$ we know that there is a $w$ such that the formula of $R_V$ is true. Since $w$ is not in the $\mathbf{S}$ of any $\sigma$, it will not be present in the union of two $\sigma$'s either, so the $R_V$ relation will hold after the steps as well.

PEND Analogous to RECV.

□

**Corollary 4.17.** *If* $\gamma^{\text{HB}}\, R\, \gamma^{\text{SC}}$ *then* $\gamma^{\text{HB}} \sim \gamma^{\text{SC}}$.

*Proof.* Follows immediately from Definition 4.15. □

Actually, we have now proved that if a program under the HB-semantics is data race free, then it will run similarly to (not to say identically as) the program under the SC-semantics.

### 4.2.3 Data race in SC-semantics

This section will show that if a program is data race free when running under the HB-semantics it will also be data race free in an another sense when run under the SC-semantics.

**Definition 4.18** (SC-race condition)**.** A SC-program (i.e. starting configuration) has a *data race* if it can reach a configuration that satisfies one of the following:

$WW_{\text{SC}}$ A goroutine can perform a WRITE on $\dot{x}$, while another goroutine can also perform a WRITE on $\dot{x}$. This is a write-write conflict.

$RW_{\text{SC}}$ A goroutine can perform a READ from $\dot{x}$, while another goroutine can perform a WRITE to $\dot{x}$. This is a write-read conflict.

[1] Note, however, that under some circumstances, all of them can have the value $v$, for instance if the program writes the value $v$ at *every* WRITE to $\dot{x}$

We will use the notation $\text{DRF}^*_{\text{SC}}(\gamma^{\text{SC}})$ to denote a program $\gamma^{\text{SC}}$ that can *not* reach a configuration that satisfies either $WW_{\text{SC}}$ or $RW_{\text{SC}}$.

Let $R^-$ be the same relation as $R$, but without the requirement that $\neg R^*_{AMB}(\gamma^{\text{HB}})$.

**Theorem 4.19.** *Given starting configurations* $\gamma_s^{\text{HB}}$ *and* $\gamma_s^{\text{SC}}$ *such that* $\gamma_s^{\text{HB}}\ R^-\ \gamma_s^{\text{SC}}$*, then* $\text{DRF}^*_{\text{HB}}(\gamma_s^{\text{HB}}) \Rightarrow \text{DRF}^*_{\text{SC}}(\gamma_s^{\text{SC}})$.

*Proof.* We will assume $\text{DRF}^*_{\text{HB}}(\gamma_s^{\text{HB}})$, and $\neg\text{DRF}^*_{\text{SC}}(\gamma_s^{\text{SC}})$, and then show that this leads to contradiction. According to the contrapositive of Theorem 4.12, we have that $\neg R^*_{AMB}(\gamma_s^{\text{HB}})$, and so we have $\gamma_s^{\text{HB}}\ R\ \gamma_s^{\text{SC}}$. By the assumption that $\neg\text{DRF}^*_{\text{SC}}(\gamma_s^{\text{SC}})$, we know that there exist a configuration $\gamma^{\text{SC}}$ such that $\gamma_s^{\text{SC}} \rightarrow^* \gamma^{\text{SC}}$, and that satisfies either $WW_{\text{SC}}$, or $RW_{\text{SC}}$. We consider the two cases separately:

$WW_{\text{SC}}$ In this case, we know by Lemma 4.14 that there exist configuration $\gamma^{\text{HB}}$ such that $\gamma_s^{\text{HB}} \rightarrow^* \gamma^{\text{HB}}$ and $\gamma^{\text{HB}}\ R\ \gamma^{\text{SC}}$. $\gamma^{\text{HB}}$ will actually itself satisfy $WW_{\text{SC}}$. Say that $g_1$ and $g_2$ are the goroutines that can make the WRITE-steps on $\dot{x}$, we can clearly see that if we first let $g_1$ perform its WRITE-step, the write $w$ it puts in $\mathbf{W}$ will not be in $\mathbf{H}(\sigma^{g_2})$. So if we now look at $g_2$, that is still allowed to perform its WRITE-step, we know that $\{w\} \subseteq \mathbf{W}_{\dot{x}} \setminus \mathbf{H}_{\dot{x}}(\sigma^{g_2})$, making this (reachable) configuration satisfy $WW_{\text{HB}}$, which contradicts the assumption $\text{DRF}^*_{\text{HB}}(\gamma_s^{\text{HB}})$.

$RW_{\text{SC}}$ This is analogous to the previous case, though we must let the goroutine that is allowed to do the WRITE-step take its step before the goroutine that is allowed to do the READ-step.

□

# Chapter 5

# First Class Functions

Go treats functions as first class citizens,[1] and it even has support for lambdas and closures. Therefore modeling a function call can't be done as simply as introducing a stack frame for variables that just disappears when the function is done.

This chapter will show how to extend the HB-semantics described in Chapter 2 to a functional setting with closures. In addition to simply demonstrate that it can be done at all, it will make far more language constructs in Go to be more directly translated to a starting configuration in our abstract syntax. Also it will show that reasoning about whether a variable is goroutine-local may not be trivial, not to say sometimes impossible.

Closures in Go has been examined before in [20], along with the language constructs `defer`, `panic`, and `recover`. It does not deal with shared variables, so for analysis of programs without shared variables, the semantics described there has fewer and simpler rules.

Consider Listing 5.1 on the following page. The output would be "2, 3, 11". The lambda function we have written to `f_inner` we will refer to as `f_inner`. This program demonstrates at least the following:

- The variable `x` inside the body of `f_inner` is referring to the `x` in `f` at the time when the lambda is defined. This is known as lexical binding.
- `x` is still referable even after `f` has returned, so the program isn't just depending on a stack to store function-local variables.
- The two lambdas that are eventually bound to `g` and `h` are referring to two different "versions" of `x`.
- When the lambdas are called, they can not only read, but also write to their version of `x`.

[1] First class citizens of a language are entities that "supports all the operations generally available to other entities" [5]. For Go, these operators includes being passed as argument, returned from a function, written to a variable, and sent over a channel.

```
package main

import "fmt"

func f(param int) func() int {
   x := param
   f_inner := func() int {
      x++
      return x
   }
   return f_inner
}

func main() {
   g := f(1)
   h := f(10)
   fmt.Printf("%d, ", g())
   fmt.Printf("%d, ", g())
   fmt.Printf("%d\n", h())
}
```

Listing 5.1: Closure example

The way this usually works is that when a function or lambda is defined, a reference to a *frame* is stored with it. We will call such a frame a *referenced frame*. A frame traditionally stores the bindings of variables. Whenever a function is called a new frame is created and will hold the bindings of the parameters passed to the functions, as well as the local variables. It will also hold a reference to the referenced frame of the function. We will call such a frame a *local frame*. The local frame of a currently executing function we will call an *active frame*.

All functions declared at top level will have the *global frame* (i.e. the frame holding the bindings of all global variables) as their referenced frame. If a lambda is defined during the execution of a function call, a reference to the active frame is stored with it. Whenever a variable is referenced, the execution will first try to find its value in the active frame. If the binding isn't defined there, it will look for the value in the referenced frame instead, and if it isn't there either, it will keep looking all the way to the global frame.

The frames that are created when Listing 5.1 is executed is illustrated in Figure 5.1 on the next page. The figure captures the following moments:

a) Before main is called.

b) Just after main is called.

c) After the first call to f is done.

d) After the second call to f is done.

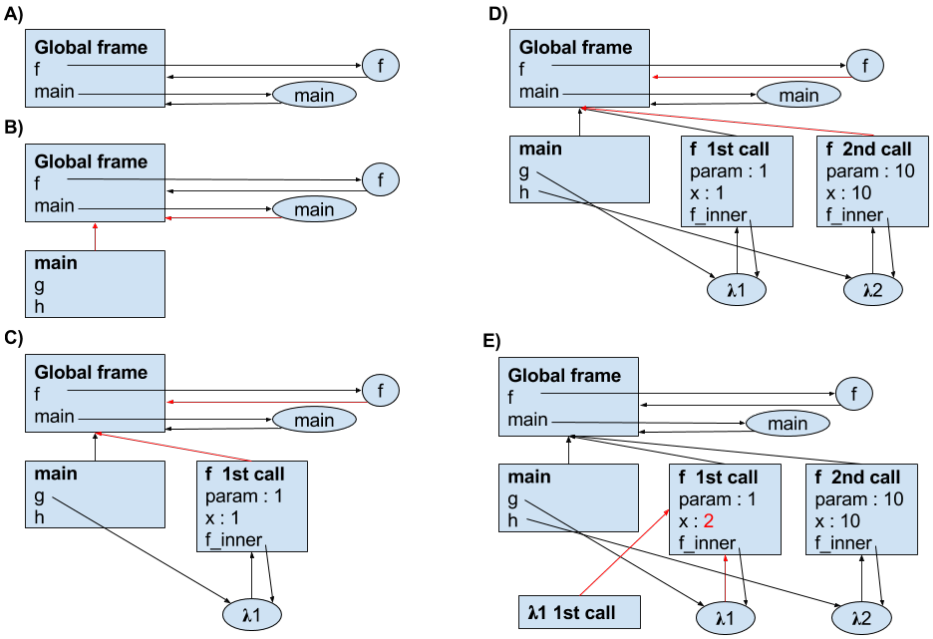


Figure 5.1: When functions are called, a new frame is created.

e) After the first call to g.

## 5.1 Contexts

The model with frames depends on the view that each variable neatly maps to its value (i.e. there exists a map from variables to values). Since this is not the case in our model, we can rather make each variable map to a something we will call a *write token* instead. Earlier we said that we wrote to variables, from this point we will only write to write tokens.

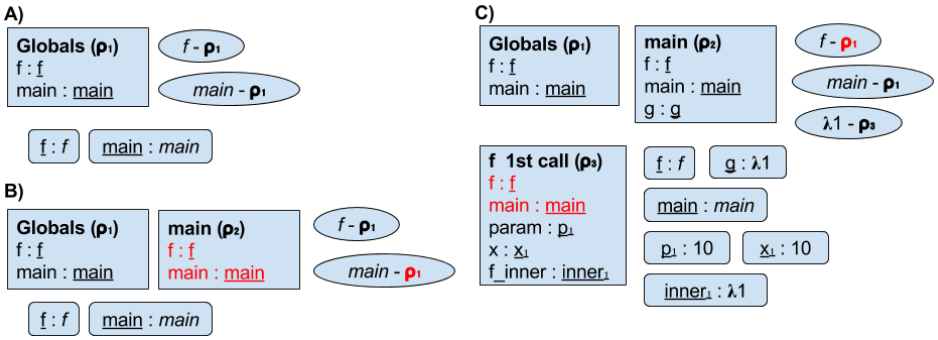


Figure 5.2: Contexts corresponding to A), B), and C) in figure Figure 5.1. Red bindings are bindings copied from the context also highlighted in red.

If we take a look on Figure 5.2 we will see something that resembles frames which are the big, squared boxes. We will call these *contexts*. In this figure

we find the write tokens as the names with an underscore, and we see that within the contexts each variable maps to a write token. In the text we will denote a write token by putting a dot above it, e.g. $\dot{x}$ is a write token.

The figure depicts writes as small, rounded boxes. a box with "$\underline{x}_1 : 10$" will correspond to a write $\langle id, \dot{x}_1, 10 \rangle$. each of the boxes will have a distinct id, although it is not explicitly depicted. All writes are collected in $\mathbf{W}$.

Each of the contexts are denoted with a $\rho$. When a function is declared, the context in which it was declared is stored with the function. When a function is called, a new context is created, and until other variables are declared in the function, it is just a copy of the context stored with it. Note that the reason that it is possible to just copy the contexts (as opposed to making the contexts refer to each other like the frames do) is that the write tokens are *immutable* [7].

As we can see, Figure 5.2 is already getting cluttered compared to Figure 5.1. Still the figure can be helpful if one keeps in mind that the figures, for the time being, corresponds to each other. When describing the rules for functions, we will extend on Figure 5.2 as a means of providing examples.

## 5.2 Syntax

We will extend the syntax from Chapter 2 to cover variable declaration, variable assignment, function declarations, and function calls. The **go** - statement is also introduced here, and will finally make it possible for us to make programs that dynamically spawns goroutines. We use the notation $|=$ to mean that we keep the possible productions from Chapter 2, in addition to the productions we introduce here.

| | | | |
|---|---|---|---|
| Statement | *stmt* | $\mid=$ | **var** $x = e$ \| **var** $form = elist$ \| $e = e$ \| $elist = elist$ |
| | | \| | $elist = f(elist)$ \| **go** $f(elist)$ \| **return** $elist$ |
| VariableList | *form* | = | $\cdot$ \| $x.form$ |
| ExpressionList | *elist* | = | $\cdot$ \| $e.elist$ |
| Expression | $e$ | $\mid=$ | $x$ \| **func** $(form)\{stmts\}$ |
| Value | $v$ | $\mid=$ | $f$ |

## 5.3 Semantics

Previously we have had $C$ to be a function from *id*s to channels. From this point the $C$ can map to other things as well, like function definitions, arrays, or references to write tokens (although, we will only provide rules here for function definitions). It should always be clear from the context what we are trying to retrieve from $C$, for instance, we will use $ch$ to denote an $id$ to a channel, or $f$ to denote an $id$ to a function. Such *id*s are values, so whenever we find $v$ in a rule, it can be one of these as well. This also applies to previous rules.

### 5.3.1 Variables

$$\textsc{Decl-form}\ \frac{}{\gamma \rhd \langle \textbf{var}\ x.form = v.vals, \sigma, R\rangle \rightarrow \gamma \rhd \langle \textbf{var}\ x = v\textbf{;}\ \textbf{var}\ form = vals, \sigma, R\rangle}$$

$$\textsc{Decl}\ \frac{\dot{x} = fresh \qquad \rho' = \rho[x \mapsto \dot{x}]}{\gamma \rhd \langle \textbf{var}\ x = v, \sigma, \rho.R\rangle \rightarrow \gamma \rhd \langle \dot{x} = v, \sigma, \rho'.R\rangle}$$

This corresponds to adding a row in the current context, like for instance the row with $g$ was added to the $\rho_2$ context in Figure 5.2 (i.e. the row is not present in B), but is present in C)).

$$\textsc{Eval-lvar}\ \frac{e \neq \dot{x} \qquad e \rightarrow e'}{\gamma \rhd \langle e = e, \sigma, R\rangle \rightarrow \gamma \rhd \langle e' = e, \sigma, R\rangle}$$

This rule requires that the expression on the left hand side will eventually evaluate to a write token. Like types, this will be the job of the compiler to make sure of. Similar syntax could be introduced for channels, i.e. $e \leftarrow e$ where the left hand side must evaluate to a channel, and $\leftarrow e$.

$$\textsc{Deref-var}\ \frac{\rho(x) = \dot{x}}{\gamma \rhd \langle x, \sigma, \rho.R\rangle \rightarrow \gamma \rhd \langle \dot{x}, \sigma, \rho.R\rangle}$$

### 5.3.2 Functions

A function will be modeled by $\lambda\langle form, stmts, \rho\rangle$. The $\lambda$ in front of the tuple is just to mark that it is a function, and thus make it visually separable from channels. For the following rules, it should be noted that *vals* is an *elist* in which all expressions are values. Rules that evaluates an *elist* to *vals* is given in the rules Eval-list-head, and Eval-list-tail below. Rules that utilizes these works analogously to the *-eval-expr rules from Chapter 2, and should be made for the Apply-* rules (both to resolve the function *id* (i.e. $f$), and the parameters), and the **return** *elist* statement, but neither will be explicitly typed out here.

$$\textsc{Func-decl}\ \frac{f = fresh \qquad C' = C[f \mapsto \lambda\langle form, stmts, \rho\rangle]}{\langle \mathbf{W}, C\rangle \rhd \langle x = \textbf{func}\ (form)\{stmts\}, \sigma, \rho.R\rangle \rightarrow \langle \mathbf{W}, C'\rangle \rhd \langle x = f, \sigma, \rho.R\rangle}$$

The declared function captures the current (i.e. topmost) context of the goroutine.

$$\textsc{Apply-async}\ \frac{C(f) = \lambda\langle form, stmts, \rho\rangle}{\gamma \rhd \langle \textbf{go}\ f(vals), \sigma, R\rangle \rightarrow \gamma \rhd \langle \circ, \sigma, R\rangle \parallel \langle \textbf{var}\ form = vals\textbf{;}\ stmts, \sigma, \rho\rangle}$$

$$\textsc{Apply-sync}\ \frac{\begin{array}{c} C(f) = \lambda\langle form, stmts, \rho\rangle \\ f_{call} \equiv \{\textbf{var}\ form = vals\textbf{;}\ stmts\}\end{array}}{\gamma \rhd \langle elist = f(vals), \sigma, R\rangle \rightarrow \gamma \rhd \langle elist = f_{call}, \sigma, \rho.R\rangle}$$

Note that the $\rho$ is pushed onto the context stack of the goroutine.

$$\textsc{Func-step}\ \frac{stmts \neq \textbf{return}\ vals\textbf{;}\ stmts'' \qquad \gamma \rhd \langle stmts, \sigma, R\rangle \to \gamma' \rhd \langle stmts', \sigma', R'\rangle}{\gamma \rhd \langle elist = \{stmts\}, \sigma, R\rangle \to \gamma' \rhd \langle elist = \{stmts'\}, \sigma', R'\rangle}$$

$$\textsc{Func-return}\ \frac{stmts = \textbf{return}\ vals\textbf{;}\ stmts'}{\gamma \rhd \langle elist = \{stmts\}, \sigma, \rho\textbf{.}R\rangle \to \gamma \rhd \langle elist = vals, \sigma, R\rangle}$$

Here $\rho$ is popped off the stack of the goroutine.

### 5.3.3 Expression lists

$$\textsc{Assign-list}\ \frac{}{\gamma \rhd \langle e_1\textbf{.}elist_1 = e\textbf{.}elist_2, \sigma, R\rangle \to \gamma \rhd \langle e_1 = e_2\textbf{;}\ elist_1 = elist_2, \sigma, R\rangle}$$

$$\textsc{Assign-list-discard}\ \frac{}{\gamma \rhd \langle \cdot = vals, \sigma, R\rangle \to \gamma \rhd \langle \circ, \sigma, R\rangle}$$

$$\textsc{Eval-list-head}\ \frac{\gamma \rhd \langle e, \sigma, R\rangle \to \gamma \rhd \langle e', \sigma, R\rangle}{\gamma \rhd \langle e\textbf{.}elist, \sigma, R\rangle \to \gamma \rhd \langle e'\textbf{.}elist, \sigma, R\rangle}$$

$$\textsc{Eval-list-tail}\ \frac{\gamma \rhd \langle elist, \sigma, R\rangle \to \gamma \rhd \langle elist', \sigma, R\rangle}{\gamma \rhd \langle e\textbf{.}elist, \sigma, R\rangle \to \gamma \rhd \langle e\textbf{.}elist', \sigma, R\rangle}$$

## 5.4 New translation to starting configuration

With the extended semantics, it is now possible to map a program more directly to a starting configuration. All starting configurations can now initially have only one goroutine, where the $\sigma$ is $\langle \emptyset, \emptyset \rangle$. The program of the goroutine can be mapped along the lines of Figure 5.3 on the next page.

## 5.5 Visibility of write tokens

Earlier it was much easier to reason about which goroutines that was reading from or writing to which write tokens; it was plainly written in the code. Now we have provided rules that makes it possible to make closures, and if we keep in mind that closures can be sent through channels, we must realize that we have introduced the possibility of creating programs where it may be virtually impossible to track which goroutines that can get a hold of which references. First class arrays, or variable references introduces the same problem.

Still, the introduction of these rules does not invalidate the theorems in Chapter 2.

```
package main

import "fmt"

func f(p int) func() int {
    x := p
    f_i := func() int {
        x++
        return x
    }
    return f_i
}

func main() {
    g := f(1)
    h := f(10)
    fmt.Println(g())
    fmt.Println(g())
    fmt.Println(h())
}
```

```
var f = func(p){
  var x = p
  var f_i = func(·){
    x = x + 1
    return x
  }
  return f_i
}

var main = func(·){
  var g = ∘
  g = f(1)
  var h = ∘
  h = f(10)
  · = g(·)
  · = g(·)
  · = h(·)
}

· = main(·)
```

Figure 5.3: New translation to starting configuration.

# Chapter 6

# Conclusion

In this thesis we started by providing a structural operational semantics that reflects the Go memory model for *reads*, *writes*, and channel communication. We found that we would model the *happens before* relations by means of *sets*, in particular the sets:

**W**: The set of all writes.

**H**: The set of all writes that has *happened before*.

**S**: The set of all writes that has been *shadowed* (or in terminology of [13] *overwritten*).

We modeled **W** to be a part of the *global state* of the *configurations* we defined, while **H** and **S** was modeled as part of the goroutine local states which we referred to as $\sigma$'s. When modeling channel communication we made the channels pass $\sigma$'s in addition to passing the ordinary values.

We proceeded by providing examples on how to use the semantic rules to step through programs. In doing this we also introduced notation to make it easier to type out the configurations.

Next, we proved that if a program is data race free under our semantics, then it is sequentially consistent. This property is generally deemed to be *required* by any sound memory model.

Lastly, we provided a framework for extending our semantics into a functional setting with closures, and dynamic spawning of goroutines.

## Discussion and future work

The semantic rules may not encompass the issue of *out of thin air* results correctly, in the sense that the Go language seems to allow behavior that is not allowed by our semantics. We will illustrate this by referring to one of the situations discussed in [2]. Consider the situation in Figure 6.1. According to the Go memory model the statements of the assignments of

```
r1 = x
y = r1
```

```
r2 = y
x = 42
```

Initially `x=y=0`. Should `r1=r2=42` be allowed?

Figure 6.1

the right thread can be reordered, since neither of the assignments depends on the other. After such a reordering we can clearly have `r1=r2=42` as a possible outcome. In [2] they claim that this possibility is even "generally accepted as benign". Our semantics can not, however, produce this result, as `r2` can not assume the value `42`.

To which extent the Go memory model allows the other situations described in [2] is unclear. That depends on which compiler optimizations the Go language allows, aside of the mentioned kind of assignment reordering. It also depends on whether the language specification in some way prohibits the speculations that are described in [2]. Since such speculations seems not to be explicitly prohibited, it can seem like some of the problematic situations are allowed.

*Out of thin air* results and how to prohibit them is also discussed in [17].

Locks has not been discussed in this thesis, even though behavior for them is defined on the specification web page [21]. Arguably, adding semantic rules for them should not invalidate the theorems of Chapter 2, but still formalizing rules for them could be interesting. The reason that we did not do this is that locks are required to be imported through the `sync` package, so we did not consider them a part of the core language.

We used Lamport's bakery algorithm to demonstrate the workings of our semantics. It is one of many algorithms that are known to work under a strong memory model (e.g. the SC-semantics), but break under a weak memory model. A technique known as *inferring fences* has been developed to mend such algorithms [9, 16]. Such techniques may not exist out of the box in Go, but even so, it would be interesting to see how the semantics would be different.

In Appendix B we suggest a version of the Bakery algorithm that uses channels as a means for fences. It may or may not be proven that this version will have the mutual exclusion property under our semantics, but if it does, it would prove as an example of an algorithm that is not properly synchronized, but still do what is expected from it.

Chapter 5 shows how to extend the semantics to include for instance functions, and shows how this makes it hard to reason about the visibility of write tokens. For instance, it is harder to reason whether a variable is goroutine local or not.

It should be entirely possible to add arrays and references into the semantics as well.

The semantics can be used as a theoretical base for implementing a data race checker. However, since this implementation would have to update sets at every assignment, it would slow down a program considerably. Variables that are goroutine local would not be required to be tracked.

# Appendix A

# Inference rules

In this appendix we have collected all the inference rules from both Chapter 2 and Chapter 5, as well as the complete abstract syntax.

## Syntax

| | | | |
|---|---|---|---|
| Program | | = | *stmts* |
| StatementList | *stmts* | = | *stmt* \| *stmt*; *stmts* |
| Statement | *stmt* | = | **for** *e* {*stmts*} \| **if** *e* {*stmts*} **else** {*stmts*} |
| | | \| | **var** *x* = *e* \| **var** *form* = *elist* \| *e* = *e* \| *elist* = *elist* |
| | | \| | *elist* = *f*(*elist*) \| **go** *f*(*elist*) \| **return** *elist* |
| | | \| | *e* \| $\dot{x}$ = *e* \| *ch* ← *e* |
| VariableList | *form* | = | · \| *x*.*form* |
| ExpressionList | *elist* | = | · \| *e*.*elist* |
| Expression | *e* | = | *v* \| *e op e* \| $\dot{x}$ \| ← *ch* \| *x* \| **func** (*form*){*stmts*} |
| Value | *v* | = | ∘ \| *m* \| *b* \| *ch* \| *f* |

## Rules

### Basic rules

$$\textsc{Par-step}\ \frac{\gamma \rhd g \to \gamma' \rhd g'}{\gamma \rhd g \| G \to \gamma' \rhd g' \| G}$$

$$\textsc{Seq}\ \frac{\gamma \rhd \langle stmt, \sigma\rangle \to \gamma' \rhd \langle stmts, \sigma'\rangle}{\gamma \rhd \langle stmt;\ stmts^\star, \sigma\rangle \to \gamma' \rhd \langle stmts;\ stmts^\star, \sigma'\rangle}$$

$$\textsc{Skip}\ \frac{}{\gamma \rhd \langle v;\ stmts^\star, \sigma\rangle \to \gamma \rhd \langle stmts^\star, \sigma\rangle}$$

## Control structures

$$\textsc{If-eval-expr}\ \frac{\gamma \rhd \langle e, \sigma\rangle \to \gamma' \rhd \langle e', \sigma'\rangle}{\gamma \rhd \langle \textbf{if}\ e\ \{stmts_1\}\ \textbf{else}\ \{stmts_2\}, \sigma\rangle \to \gamma' \rhd \langle \textbf{if}\ e'\ \{stmts_1\}\ \textbf{else}\ \{stmts_2\}, \sigma'\rangle}$$

$$\textsc{If-true}\ \frac{v = \textsc{True}}{\gamma \rhd \langle \textbf{if}\ v\ \{stmts_1\}\ \textbf{else}\ \{stmts_2\}, \sigma\rangle \to \gamma \rhd \langle stmts_1, \sigma\rangle}$$

$$\textsc{For-prep}\ \frac{}{\gamma \rhd \langle \textbf{for}\ e\ \{stmts\}, \sigma\rangle \to \gamma \rhd \langle \underline{\textbf{for}}\ \langle e, e\rangle\ \{stmts\}, \sigma\rangle}$$

$$\textsc{For-eval-expr}\ \frac{\gamma \rhd \langle e_1, \sigma\rangle \to \gamma' \rhd \langle e_1', \sigma'\rangle}{\gamma \rhd \langle \underline{\textbf{for}}\ \langle e_1, e_2\rangle\ \{stmts\}, \sigma\rangle \to \gamma' \rhd \langle \underline{\textbf{for}}\ \langle e_1', e_2\rangle\ \{stmts\}, \sigma'\rangle}$$

$$\textsc{For-true}\ \frac{v = \textsc{True}}{\gamma \rhd \langle \underline{\textbf{for}}\ \langle v, e\rangle\ \{stmts\}, \sigma\rangle \to \gamma \rhd \langle stmts\textbf{;}\ \textbf{for}\ e\ \{stmts\}, \sigma\rangle}$$

$$\textsc{For-false}\ \frac{v = \textsc{False}}{\gamma \rhd \langle \underline{\textbf{for}}\ \langle v, e\rangle\ \{stmts\}, \sigma\rangle \to \gamma \rhd \langle \circ, \sigma\rangle}$$

## Reads and writes

$$\textsc{Write-eval-expr}\ \frac{\gamma \rhd \langle e, \sigma\rangle \to \gamma' \rhd \langle e', \sigma'\rangle}{\gamma \rhd \langle \dot{x} = e, \sigma\rangle \to \gamma' \rhd \langle \dot{x} = e', \sigma'\rangle}$$

$$\textsc{Write}\ \frac{\begin{array}{cc} id = fresh & \sigma = \langle \mathbf{H}, \mathbf{S}\rangle \\ w = \langle id, \dot{x}, v\rangle & \sigma' = \langle \mathbf{H} \cup \{w\}, \mathbf{S} \cup \mathbf{H}_{\dot{x}}\rangle \end{array}}{\langle \mathbf{W}, C\rangle \rhd \langle \dot{x} = v, \sigma\rangle \to \langle \mathbf{W} \cup \{w\}, C\rangle \rhd \langle \circ, \sigma'\rangle}$$

$$\textsc{Read}\ \frac{\sigma = \langle \mathbf{H}, \mathbf{S}\rangle \qquad \langle id, \dot{x}, v\rangle \in \mathbf{W}_{\dot{x}} \setminus \mathbf{S}_{\dot{x}}}{\langle \mathbf{W}, C\rangle \rhd \langle \dot{x}, \sigma\rangle \to \langle \mathbf{W}, C\rangle \rhd \langle v, \sigma\rangle}$$

## Channel communication

$$\textsc{Send}\ \frac{\begin{array}{c} C(ch) = \langle s, r, c, \mathcal{M}, \mathcal{V}\rangle \qquad s \le c + r \\ \mathcal{M}' = \mathcal{M}[s \mapsto \sigma] \qquad \mathcal{V}' = \mathcal{V}[s \mapsto v] \\ C' = C[ch \mapsto \langle s + 1, r, c, \mathcal{M}', \mathcal{V}'\rangle] \end{array}}{\langle \mathbf{W}, C\rangle \rhd \langle ch \leftarrow v, \sigma\rangle \to \langle \mathbf{W}, C'\rangle \rhd \langle \underline{\textbf{pend}}\ \langle ch, s\rangle, \sigma\rangle}$$

$$\textsc{Recv}\ \frac{\begin{array}{c} C(ch) = \langle s, r, c, \mathcal{M}, \mathcal{V}\rangle \qquad r < s \\ \mathcal{M}(r) = \sigma' \qquad \mathcal{V}(r) = v \qquad \mathcal{M}' = \mathcal{M}[r \mapsto \sigma] \\ C' = C[ch \mapsto \langle s, r + 1, c, \mathcal{M}', \mathcal{V}\rangle] \end{array}}{\langle \mathbf{W}, C\rangle \rhd \langle \leftarrow ch, \sigma\rangle \to \langle \mathbf{W}, C'\rangle \rhd \langle v, \sigma \cup \sigma'\rangle}$$

$$\textsc{Pend}\ \frac{C(ch) = \langle \_, r, c, \mathcal{M}, \_\rangle \qquad r > s - c \qquad \mathcal{M}(s - c) = \sigma'}{\langle \mathbf{W}, C\rangle \rhd \langle \underline{\textbf{pend}}\ \langle ch, s\rangle, \sigma\rangle \to \langle \mathbf{W}, C'\rangle \rhd \langle \circ, \sigma \cup \sigma'\rangle}$$

## Variables

$$\textsc{Decl-form}\ \frac{}{\gamma \rhd \langle \textbf{var}\ x\textbf{.}form = v\textbf{.}vals, \sigma, R\rangle \to \gamma \rhd \langle \textbf{var}\ x = v\textbf{;}\ \textbf{var}\ form = vals, \sigma, R\rangle}$$

$$\textsc{Decl}\ \frac{\dot{x} = \mathit{fresh} \qquad \rho' = \rho[x \mapsto \dot{x}]}{\gamma \rhd \langle \mathbf{var}\ x = v, \sigma, \rho\mathbf{.}R\rangle \to \gamma \rhd \langle \dot{x} = v, \sigma, \rho'\mathbf{.}R\rangle}$$

$$\textsc{Eval-lvar}\ \frac{e \neq \dot{x} \qquad e \to e'}{\gamma \rhd \langle e = e, \sigma, R\rangle \to \gamma \rhd \langle e' = e, \sigma, R\rangle}$$

$$\textsc{Deref-var}\ \frac{\rho(x) = \dot{x}}{\gamma \rhd \langle x, \sigma, \rho\mathbf{.}R\rangle \to \gamma \rhd \langle \dot{x}, \sigma, \rho\mathbf{.}R\rangle}$$

### Functions

$$\textsc{Func-decl}\ \frac{f = \mathit{fresh} \qquad C' = C[f \mapsto \lambda\langle \mathit{form}, \mathit{stmts}, \rho\rangle]}{\langle \mathbf{W}, C\rangle \rhd \langle x = \mathbf{func}\ (\mathit{form})\{\mathit{stmts}\}, \sigma, \rho\mathbf{.}R\rangle \to \langle \mathbf{W}, C'\rangle \rhd \langle x = f, \sigma, \rho\mathbf{.}R\rangle}$$

$$\textsc{Apply-async}\ \frac{C(f) = \lambda\langle \mathit{form}, \mathit{stmts}, \rho\rangle}{\gamma \rhd \langle \mathbf{go}\ f(\mathit{vals}), \sigma, R\rangle \to \gamma \rhd \langle \circ, \sigma, R\rangle \parallel \langle \mathbf{var}\ \mathit{form} = \mathit{vals}\mathbf{;}\ \mathit{stmts}, \sigma, \rho\rangle}$$

$$\textsc{Apply-sync}\ \frac{\begin{array}{c} C(f) = \lambda\langle \mathit{form}, \mathit{stmts}, \rho\rangle \\ f_{\mathit{call}} \equiv \{\mathbf{var}\ \mathit{form} = \mathit{vals}\mathbf{;}\ \mathit{stmts}\} \end{array}}{\gamma \rhd \langle \mathit{elist} = f(\mathit{vals}), \sigma, R\rangle \to \gamma \rhd \langle \mathit{elist} = f_{\mathit{call}}, \sigma, \rho\mathbf{.}R\rangle}$$

$$\textsc{Func-step}\ \frac{\begin{array}{c} \mathit{stmts} \neq \mathbf{return}\ \mathit{vals}\mathbf{;}\ \mathit{stmts}'' \\ \gamma \rhd \langle \mathit{stmts}, \sigma, R\rangle \to \gamma' \rhd \langle \mathit{stmts}', \sigma', R'\rangle \end{array}}{\gamma \rhd \langle \mathit{elist} = \{\mathit{stmts}\}, \sigma, R\rangle \to \gamma' \rhd \langle \mathit{elist} = \{\mathit{stmts}'\}, \sigma', R'\rangle}$$

$$\textsc{Func-return}\ \frac{\mathit{stmts} = \mathbf{return}\ \mathit{vals}\mathbf{;}\ \mathit{stmts}'}{\gamma \rhd \langle \mathit{elist} = \{\mathit{stmts}\}, \sigma, \rho\mathbf{.}R\rangle \to \gamma \rhd \langle \mathit{elist} = \mathit{vals}, \sigma, R\rangle}$$

### Expression lists

$$\textsc{Assign-list}\ \frac{}{\gamma \rhd \langle e_1\mathbf{.}\mathit{elist}_1 = e\mathbf{.}\mathit{elist}_2, \sigma, R\rangle \to \gamma \rhd \langle e_1 = e_2\mathbf{;}\ \mathit{elist}_1 = \mathit{elist}_2, \sigma, R\rangle}$$

$$\textsc{Assign-list-discard}\ \frac{}{\gamma \rhd \langle \cdot = \mathit{vals}, \sigma, R\rangle \to \gamma \rhd \langle \circ, \sigma, R\rangle}$$

$$\textsc{Eval-list-head}\ \frac{\gamma \rhd \langle e, \sigma, R\rangle \to \gamma \rhd \langle e', \sigma, R\rangle}{\gamma \rhd \langle e\mathbf{.}\mathit{elist}, \sigma, R\rangle \to \gamma \rhd \langle e'\mathbf{.}\mathit{elist}, \sigma, R\rangle}$$

$$\textsc{Eval-list-tail}\ \frac{\gamma \rhd \langle \mathit{elist}, \sigma, R\rangle \to \gamma \rhd \langle \mathit{elist}', \sigma, R\rangle}{\gamma \rhd \langle e\mathbf{.}\mathit{elist}, \sigma, R\rangle \to \gamma \rhd \langle e\mathbf{.}\mathit{elist}', \sigma, R\rangle}$$

## Alternative rules for channels

The rules for channels given in chapter 2 may be a bit inconvenient since the function $\mathscr{M}$ really have to "remember" arbitrarily many $\sigma$'s, as the pending goroutines can wait arbitrarily long before stepping out of the pending state even when they can.

This alternative semantics will only allow a limited number of goroutines to pend on it, and it will make a FIFO for it. In this semantics, a channel will be represented by a tuple $\langle c, V, S, R, P\rangle$ where

- $c$ is the capacity of the channel.
- $V$ is a FIFO of the values sent over the channel.
- $S$ is a FIFO of $\sigma$'sthat are put in by sending goroutines and picked up by the corresponding receiving goroutines.
- $R$ is a FIFO of $\sigma$'sthat are put in by the receiving goroutines and picked up by the corresponding pending goroutines.
- $P$ is a FIFO of tokens corresponding to goroutines that are pending on the channel.

$$\textsc{Send}\ \frac{\begin{array}{c} C(ch) = \langle c, V, S, R, P\rangle \qquad \|S\| \leq c \\ token = fresh \qquad C' = C[ch \mapsto \langle c, v.V, \sigma.S, R, token.P\rangle] \end{array}}{\langle \mathbf{W}, C\rangle \rhd \langle ch \leftarrow v, \sigma\rangle \rightarrow \langle \mathbf{W}, C'\rangle \rhd \langle \underline{\mathbf{pend}}\ \langle ch, token\rangle, \sigma\rangle}$$

$$\textsc{Recv}\ \frac{\begin{array}{c} C(ch) = \langle c, V.v, S.\sigma', R, P\rangle \qquad \|R\| \leq c \\ C' = C[ch \mapsto \langle c, V, S, \sigma.R, P\rangle \end{array}}{\langle \mathbf{W}, C\rangle \rhd \langle \leftarrow ch, \sigma\rangle\ \langle \mathbf{W}, C'\rangle \rhd \langle \circ, \sigma \cup \sigma'\rangle}$$

$$\textsc{Pend}\ \frac{\begin{array}{c} C(ch) = \langle c, V, S, R.\sigma', P.token\rangle \\ C' = C[ch \mapsto \langle c, V, S, R, P\rangle] \end{array}}{\langle \mathbf{W}, C\rangle \rhd \langle \underline{\mathbf{pend}}\ \langle ch, token\rangle, \sigma\rangle\ \langle \mathbf{W}, C'\rangle \rhd \langle \circ, \sigma \cup \sigma'\rangle}$$

# Appendix B

# Listings

Complete listing of naive implementation

```
package main

import (
    "fmt"
    "math/rand"
    "time"
)

var n = 5
var choosing = make([]int, n)
var number = make([]int, n)

func max(number []int) int {
    res := 0
    for _, v := range number {
        if v > res {
            res = v
        }
    }
    return res
}

func pairLess(a, b, c, d int) bool {
    isLess := false
    if (a < c) || (a == c && b < d) {
        isLess = true
    }
    return isLess
}

func critical(i int) {
    fmt.Printf("%d: Start critical\n", i)
```

```
    fmt.Printf("%d: number[%d]=%d\n", i, i, number[i])
    duration := time.Duration(rand.Intn(1000)) *
        time.Millisecond
    fmt.Printf("%d: Sleeping %v\n", i, duration)
    time.Sleep(duration)
    fmt.Printf("%d: End critical\n", i)
}

func bakery(i int) {
    for {
        choosing[i] = 1
        number[i] = 1 + max(number)
        choosing[i] = 0
        for j := 0; j < n; j++ {
            for choosing[j] != 0 {
            }
            for number[j] != 0 &&
                pairLess(number[j], j, number[i], i) {
            }
        }
        critical(i) // critical section here
        number[i] = 0
        // noncritical section here
    }
}

func main() {
    rand.Seed(time.Now().UnixNano())
    for i := 1; i < n; i++ {
        go bakery(i)
    }
    bakery(0)
}
```

Complete listing of mended implementation

```
package main

import (
    "fmt"
    "math/rand"
    "time"
)

var n = 5
var choosing = make([]int, n)
var number = make([]int, n)
var ch = make(chan int)
```

```
func MB() {
   ch <- 0
}

func max(number []int) int {
   res := 0
   for _, v := range number {
      if v > res {
         res = v
      }
   }
   return res
}

func pairLess(a, b, c, d int) bool {
   isLess := false
   if (a < c) || (a == c && b < d) {
      isLess = true
   }
   return isLess
}

func critical(i int) {
   fmt.Printf(" %d: Start critical\n", i)
   fmt.Printf(" %d: number[%d]=%d\n", i, i, number[i])
   duration := time.Duration(rand.Intn(1000)) *
      time.Millisecond
   fmt.Printf(" %d: Sleeping %v\n", i, duration)
   time.Sleep(duration)
   fmt.Printf(" %d: End critical\n", i)
}

func noncritical(i int) {
   fmt.Printf("N%d: Start noncritical\n", i)
   fmt.Printf("N%d: number[%d]=%d\n", i, i, number[i])
   duration := time.Duration(rand.Intn(1000)) *
      time.Millisecond
   fmt.Printf("N%d: Sleeping %v\n", i, duration)
   time.Sleep(duration)
   fmt.Printf("N%d: End noncritical\n", i)
}

func bakery(i int) {
   for {
      choosing[i] = 1
      MB()
      number[i] = 1 + max(number)
```

```
        MB()
        choosing[i] = 0
        MB()
        for j := 0; j < n; j++ {
            for choosing[j] != 0 {
                MB()
            }
            for number[j] != 0 &&
                pairLess(number[j], j, number[i], i) {
                MB()
            }
        }
        critical(i) // critical section here
        MB()
        number[i] = 0
        MB()
        noncritical(i) // noncritical section here
        MB()
    }
}

func listen() {
    for {
        <-ch
    }
}

func main() {
    rand.Seed(time.Now().UnixNano())
    for i := 0; i < n; i++ {
        go bakery(i)
    }
    listen()
}
```

# Bibliography


[1] Sarita V. Adve and Hans-J. Boehm. “Memory models: a case for rethinking parallel languages and hardware”. In: *Communications of the ACM* 53.8 (Aug. 2010), pp. 90–101.

[2] Hans-J. Boehm and Brian Demsky. “Outlawing Ghosts: Avoiding Out-of-thin-air Results”. In: *Proceedings of the Workshop on Memory Systems Performance and Correctness*. MSPC ’14. Edinburgh, United Kingdom: ACM, 2014, 7:1–7:6.

[3] *Data Race Detector – The Go Programming Language*. URL: https://golang.org/doc/articles/race_detector.html (visited on 09/23/2016).

[4] Alan A. A. Donovan and Brian W. Kernighan. *The Go Programming Language*. Addison-Wesley, 2015.

[5] *First-Class Citizen*. In: *Wikipedia, the Free Encyclopedia*. Page Version ID: 739727114. Sept. 16, 2016.

[6] Brian Goetz. “Java theory and practice: Fixing the Java Memory Model, Part 1”. In: *IBM developerWorks* (2004).

[7] *Immutable Object*. In: *Wikipedia, the Free Encyclopedia*. Page Version ID: 738814146. Sept. 11, 2016.

[8] *Java Memory Model FAQ*. URL: https://www.cs.umd.edu/~pugh/java/memoryModel/jsr-133-faq.html (visited on 09/23/2016).

[9] Michael Kuperstein, Martin Vechev, and Eran Yahav. “Automatic Inference of Memory Fences”. In: *Proceedings of Formal Methods in Computer-Aided Design (FMCAD), 2010*. 2010.

[10] Leslie Lamport. “A New Solution of Dijkstra’s Concurrent Programming Problem”. In: *Commun. ACM* 17.8 (Aug. 1974), pp. 453–455.

[11] Leslie Lamport. “How to Make a Multiprocessor Computer that Correctly Executes Multiprocess Programs”. In: *IEEE Transactions on Computers* C-28.9 (Sept. 1979), pp. 690–691.

[12] Leslie Lamport. “Time, Clocks, and the Ordering of Events in a Distributed System”. In: *Commun. ACM* 21.7 (July 1978), pp. 558–565.

[13] Jeremy Manson and William Pugh. *Semantics of Multithreaded Java*. Technical Report CS-TR-4215. Dept. of Computer Science, University of Maryland, College Park, Mar. 2001.

[14] Robin Milner. "An Algebraic Definition of Simulation between Programs". In: *Proceedings of the Second International Joint Conference on Artificial Intelligence*. William Kaufmann, 1971, pp. 481–489.

[15] Robin Milner. *Communication and Concurrency*. Prentice Hall, 1989.

[16] Chinmay Narayan, Shibashis Guha, and S. Arun-Kumar. "Inferring Fences in a Concurrent Program Using SC Proof of Correctness". In: (Apr. 10, 2013). arXiv: 1304.2936 [cs].

[17] Jean Pichon-Pharabod and Peter Sewell. "A Concurrency-Semantics for Relaxed Atomics that Permits Optimisation and avoids Out-Of-Thin-Air Executions". In: *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '16. New York, NY, USA: ACM, 2016, pp. 622–633.

[18] Gordon D. Plotkin. *A Structural Approach to Operational Semantics*. Report DAIMI FN-19. Computer Science Department, Aarhus University, Sept. 1981.

[19] William Pugh. "The Java memory model is fatally flawed". In: *Concurrency: Practice and Experience* 12.6 (2000), pp. 445–455.

[20] Martin Steffen. "A Small-Step Semantics of a Concurrent Calculus with Goroutines and Deferred Functions". In: *Theory and Practice of Formal Methods*. Ed. by Erika Ábrahám, Marcello Bonsangue, and Einar Broch Johnsen. Lecture Notes in Computer Science 9660. \. Springer International Publishing, 2016, pp. 393–406.

[21] *The Go Memory Model – The Go Programming Language*. URL: https://golang.org/ref/mem (visited on 01/22/2016).

[22] *The Go Programming Language Specification – The Go Programming Language*. URL: https://golang.org/ref/spec (visited on 06/19/2016).

[23] *The Go Programming Language Specification – The Go Programming Language – Variables*. URL: https://golang.org/ref/spec#Variables (visited on 04/10/2016).