Verification of Go Channels

by

Jessica Zhang

B.S. Electrical Engineering and Computer Science, MIT, 2023

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

MASTER OF ENGINEERING IN ELECTRICAL ENGINEERING AND COMPUTER SCIENCE

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

February 2025

Authored by:	Jessica Zhang Department of Electrical Engineering and Computer Science January 17, 2025
Certified by:	Nickolai Zeldovich Professor of Electrical Engineering and Computer Science, Thesis Supervisor
Accepted by:	Katrina LaCurts Chair, Master of Engineering Thesis Committee

Verification of Go Channels

by

Jessica Zhang

Submitted to the Department of Electrical Engineering and Computer Science
on January 17, 2025 in partial fulfillment of the requirements for the degree of

MASTER OF ENGINEERING IN ELECTRICAL ENGINEERING AND COMPUTER SCIENCE

ABSTRACT

Goose is a tool for translating a subset of the Go programming language into Perennial/Iris, which is an extension of Coq. However, Goose did not support channels, which are an important synchronization tool that Go is well known for.

This thesis presents an extension to Goose to support channels, including a model to represent Go channels and operations in GooseLang, the language defined in Perennial/Iris that Goose translates into, an extension to the Goose translator to support channels, and a library of separation logic specifications that define the expected behavior of channel operations on open channels. Finally, this thesis evaluates how effective this model and library is for verifying Go code containing channels, and discuss some limitations and potential future work.

Thesis supervisor: Nickolai Zeldovich
Title: Professor of Electrical Engineering and Computer Science

Contents

List of Figures

Chapter 1

Introduction

Synchronization in multi-threaded software is notoriously difficult to get right and difficult to test and debug as well. It could take running the same test suite hundreds of times for a given bug to even appear. An alternate strategy for ensuring the reliability of systems is formal verification, which is the act of proving that a piece of software always follows its intended specification. This thesis contributes a design to translate Go code centered around channels into Coq, and a framework to reason about channel behavior within Coq.

The following sections of this chapter describe the motivation behind this work, discuss previous work done in verifying Go code, outline the contributions made by this thesis, and provide an roadmap for the rest of this thesis.

1.1 Motivation

Concurrency is one of the most difficult parts of a system to get correct because it causes nondeterministic behavior and generally leads to more complex control flow than sequential logic. However, concurrency is also one of the most important properties of a system because of the performance bonuses it gives. Message passing is a common synchronization technique that relies on passing messages between different processes or threads to coordinate shared resources.

Compared to synchronization methods that rely on shared memory, message passing models tend to be easier to use and more scalable for large distributed systems, since the low-level memory management and synchronization are abstracted away by the message passing protocol. Go is a well-known programming language that makes extensive use of message-passing synchronization through its channel type [1].

Go channels work like shared queues. Some goroutines write messages to a channel, and other goroutines will eventually read those messages from the channel. These messages are used to coordinate between goroutines. Some common applications of channels in Go programs include synchornization, consolidating information from multiple goroutines, distributing work, creating futures and promises, sending notifications, and counting semaphores.

There have been attempts of formal verification on various message-passing schemes, however, there has not been an attempt to formally verify Go's channel implementation or large pieces of Go code that rely on channels for synchronization [2–5].

1.2 Verification of Go code using Goose

A formal proof of a system contains three components: an implementation of the system, a specification that defines the intended behavior of the system, and a proof that the implementation follows the specification [6]. All three pieces are represented as code in a verification system which computationally checks that the proof is valid.

Currently, the most well known formal verification system and programming language is Coq [7]. Another popular system is Iris, which is a concurrent separation logic framework implemented and verified in Coq [8]. However, for our purposes we will additionally be using Perennial, which adds stronger crash-safety reasoning features on top of Iris [9].

In 2019, a lot of work was done by Tej Chajed to advance the ease of verifying Go code [10]. As part of his thesis work, he built Goose, which is a tool that translates a subset of the Go programming language into Coq. The subset of Go that is translatable is also

referred to as Goose, and it gets translated into GooseLang, which is a semantic model of the Goose semantics in Coq. Figure 1.1 summarizes the relationship between Goose, Go, and GooseLang.

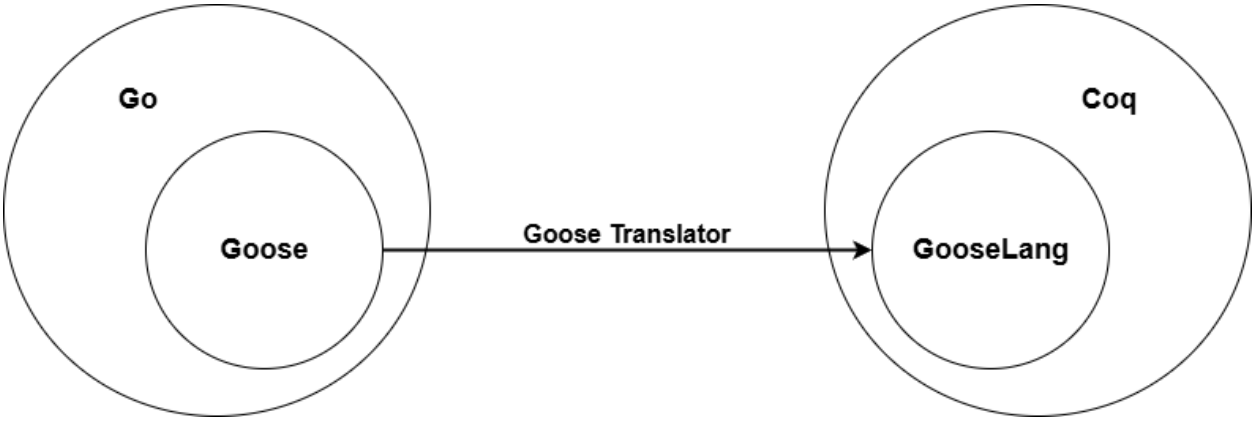

Figure 1.1: Goose is a subset of the Go programming language that can be translated into GooseLang using the Goose translator. GooseLang is a model of Goose in Coq.

A Go developer working in only Goose can verify any Goose program by using the Goose translator to translate the Goose code into GooseLang. The developer then uses the Perennial framework to verify the GooseLang program. Figure 1.2 visualizes the process of verifying Go code using Goose.

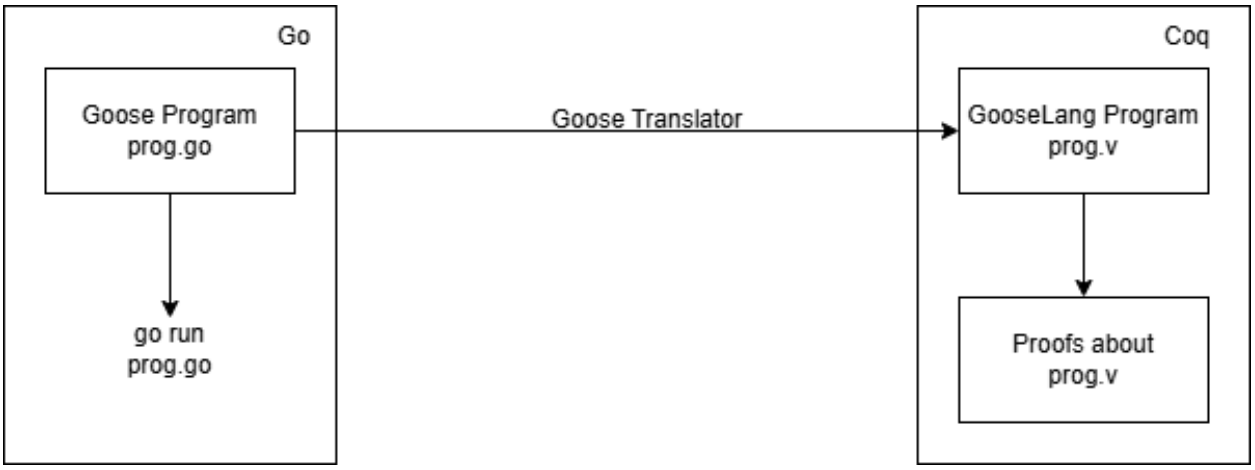

Figure 1.2: A goose program prog.go can be translated into a GooseLang program prog.v using the Goose translator. Once the program is in GooseLang, the developer can use Perennial/Iris/Coq to make proofs about prog.v.

Prior to the work done in this thesis, Goose did not provide support for channels. So all

Goose programs had to rely on mutexes for synchronization. One of the contributions of this thesis is an extension of Goose to support channels and channel operations.

1.3 Contributions

This thesis makes the following contributions:

1.3.1 A Model of Channels in GooseLang

Prior to this thesis, Goose did not support channels, so GooseLang did not include a model or any way to represent channels. This thesis presents a model of channels and channel operations in GooseLang that mimic the behavior of channels in Go. This model adds Goose support for channel creation, the cap and len functions, channel sends, and channel receives. However, the model does not provide support for channel operations in select statements.

1.3.2 An Extension to the Goose Translator

The second contribution this thesis makes is an extension to the Goose translator to handle channels. This simplifies the process of verifying future Goose programs by automating the translation step.

1.3.3 A Library of Proofs on Go Channel Operations

The final contribution this thesis adds is a library of proofs related to Go's basic channel operations. The library provides a separation logic specification for open channels, as well as proofs that the specification is sound based on our model of channels.

1.4 Roadmap for this Thesis

This section briefly outlines the chapters of this thesis.

Chapter 2 provides more background on Goose, GooseLang, and Go's channel specifications.

Chapter 3 discusses the design and implementation of the model of channels in GooseLang.

Chapter 4 explains how the Goose translator was expanded to include channels and channel operations.

Chapter 5 introduces the separation logic specification used in the library of proofs this thesis presents.

Chapter 6 explains the proof details for the specification introduced in Chapter 5.

Chapter 7 provides some examples of how the library of theorems can be used in verification of Go code, and gives an evaluation of how useful the specification is for that task.

Chapter 8 provides a conclusion and discussion of future work.

Chapter 2

Background

This chapter will provides more background on GooseLang and Go's channel specification.

2.1 GooseLang

GooseLang is a programming language defined in Perennial/Iris for the purpose of modeling Go programs. A programming language is defined by its syntax: the form of its expressions, and its semantics: a transition system describing how expressions reduce.

At a high level, the GooseLang semantics for any translated Go function should include all the possible behaviors of the original Go function. As long as the former is true, a proof that the translated code satisfies some specification will also apply to the original Go code [10].

To keep the language simple, many Goose features are implemented in GooseLang as libraries built from simpler primitives. For example, slices are represented as a tuple of pointer, length, and capacity, and slice operations are defined as functions rather than transitions in the semantics. Our proposal to add channels will similarly be added as a library.

2.1.1 GooseLang Syntax

Figure 2.1 lists the syntax of GooseLang. The basic unit of GooseLang is the expression e, which includes basic lambda calaculus primitives such as variaribles x, values v, function applications $e\ \ e$, and recursive functions `rec` $f(x) = e$, as well as primitives for control-flow, concurrent threads, and heap operations [10].

Our model of channels, which will be discussed in more detail in Chapter 3, makes heavy use of the `InjL, InjR` structure. A value v can be declared to be either a `InjL` $\mathtt{v}_1$ or a `InjR` $\mathtt{v}_2$, where $\mathtt{v}_1$ and $\mathtt{v}_2$ are other values that are allowed to have different structures. The `InjL, InjR` structure is useful for defining a model that is known to always be in 1 of 2 possible states.

2.2 Go Channels

The last section of this chapter will go over Go's official channel specification. The full specification of Go, including channels, can be found here: https://go.dev/ref/spec [1].

A channel behaves like a shared first-in first-out queue. Concurrently running goroutines can communicate with each other by sending and receiving values of a specified type through the channel.

2.2.1 The Channel Type

The `ChannelType` is defined as

```
ChannelType = ( "chan" | "chan" "<-" | "<-" "chan" ) ElementType .
```

The `<-` is an optional operator that makes the channel directional. A `chan<- T` is a channel that is only allowed to send values of type `T`, and a `<-chan T` is a channel that is only allowed to receive values of type `T`. If no direction is specified, then the channel is

```
Syntax
    x, f ∈ Var
    l    ∈ Loc
    n    ∈ U64 ∪ U32 ∪ U8
    s    ∈ String
    ⊙ ::= + | − | ∗ | = | < | ...
    ⊖ ::= − | ToString | ToU64 | ...
    Val v  ::= () | true | false | n | l | s |
         | InjL v | InjR v | rec f(x) = e
    Exp e  ::= x | v | e e | rec f(x) = e
         | if e then e else e
         | Fork(e)
         | (e, e) | π1 e | π2 e
         | InjL e | InjR e | case e of InjL x ⇒ e or InjR x ⇒ e
         | e ⊙ e | ⊖ e | ArbitraryU64
         | AllocN(e, e) | CmpXchg(e, e, e) | Load(e)
         | PrepareWrite(e) | FinishStore(e, e) | StartRead(e) | FinishRead(e)
         | Panic(s)
         | <External>

Derived forms and notation
    λx.e ≜ rec _(x) = e
    λx,y.e ≜ λx.λy.e
    let x = e1 in e2 ≜ (λx.e2)e1
    e1;e2 ≜ let _ = e1 in e2
    ref e ≜ AllocN(1, e)
    !x ≜ Load(x)
    Store ≜ λx,e.PrepareWrite(x);FinishStore(x,e)
    x ← e ≜ Store x e
```

Figure 2.1: GooseLang's syntax [10]

bidirectional and allowed to both send and receive. Some examples of channel declarations are shown below in Figure 2.3. An uninitialized channel has the `nil` value.

Since a bidirectional channel can be used in place of any directional channel, and any syntax errors caused by the misuse of directional channels can be caught by Go's compiler, we have decided to only handle bidirectional channels in Goose. From now on, all channels mentioned in this thesis are assumed to be bidirectional.

```
var a chan uint64     // a channel that can send and receive ints
var b chan<- float32  // a channel that can only send float32s
var c <-chan string   // a channel that can only receive strings
```

Figure 2.2: Examples of Go channel declarations

2.2.2 Built-in Functions

Go gives 4 built-in functions that work with channels.

make

The built-in `make(chan T, n)` function takes in a type `T` and an optional argument `n` of an integer type, and returns a channel of type `T` and capacity `n`. The capacity sets the size of the channel's buffer. If the capacity is 0, or not given, then the channel is unbuffered and messages are only transferred if both a send and receive are open at the same time. If the capacity is positive, then the channel is buffered, and the channel can hold up to `n` queued messages. If the capacity given is negative, then a run-time panic occurs.

```
var a chan int
a = make(chan int)         // an unbuffered channel
var b chan uint64
b = make(chan uint64, 100) // a buffered channel with buffer size of 100
var c chan float32
c = make(chan float32, 0)  // an unbuffered channel
var d chan bool
d = make(chan bool, -10)   // run-time panic
```

Figure 2.3: Examples of creating channels using the make function

close

The built-in `close()` function closes a channel. Attempting to send on closed channels causes a run-time panic.

```
c.close()   // closes channel c
```

cap and len

The built-in `cap(c)` function returns the capacity of channel `c`. The built-in `len(c)` function returns the number of elements queued in channel `c`'s buffer. The capacity and length of a `nil` channel is 0. At all points in time, the length of a channel must be non-negative and less than or equal to the capacity of the channel:

```
0 <= len(c) <= cap(c)
```

2.2.3 Send statements

```
ch1<-v1
```

The send operation adds the given element to the end of a channel. The expression being sent to the channel must match the channel's type. The send statement blocks if the channel is at max capacity. A send on a channel with 0 capacity is possible if a receive is called at the same time. A send on a closed channel will cause a run-time panic, and a send on a nil channel blocks forever.

Operation	cap = 0	0 < cap < len	0 < cap = len	nil channel	closed
Send	blocks until concurrent receive	succeeds	blocks	blocks forever	panic

Figure 2.4: Summary of send's behavior on open, nil, and closed channels

2.2.4 Receive statements

```
v1 := <-ch1
v2, ok := <-ch2
```

The receive operation should return the first element from the channel. The receive expression blocks until a value is available. A receive on a closed channel can always return immediately;

it returns elements in the buffer while there are still elements in the buffer, and returns the channel type's zero value if the buffer is empty.

The receive operation can also return a second optional boolean result **ok** which is **true** if the value received was from a successful send operation, and **false** if the value received was the zero value due to the channel being closed and empty.

Operation	cap = 0	open & len = 0	len > 0	nil channel	closed & len = 0
Receive	blocks until concurrent send	blocks	succeeds	blocks forever	returns zero value

Figure 2.5: Summary of receive's behavior on open, nil, and closed channels

Chapter 3

Channel Representation in GooseLang

This chapter will describe the model of channels in GooseLang. The channel representation in GooseLang should perfectly match Go's channel behavior, but does not need to mimic Go's actual channel performance, since our proofs will only be concerned with correctness and not performance.

3.1 Challenges

There were several challenges that we faced when trying to design and emulate Go channels in GooseLang that informed our design decisions.

3.1.1 The message queue

A non-zero capacity channel requires the channel to retain the messages sent, and the order they were sent in. Since messages enter and exit the channel in a first-in first-out order, we decided to use a linked list. Appending a new element onto the list involves traversing through the entire list, but removing an element is a simple operation of popping the current head of the list and making the next element the new head. Using a linked list instead of an array-like structure avoids the need to "shift" elements after removing the first element.

3.1.2 Blocking

There are several instances where a channel needs to block. Sending or receiving on a nil channel blocks forever. Sending on a channel at max capacity and receiving on an empty channel also block until the blocking condition becomes false. GooseLang's semantics do not provide a blocking primitive, so we will emulate this behavior using a polling scheme instead.

When initiating a send or receive, the function enters an infinite loop. Inside the loop, the function checks the blocking condition(for send: if the channel is at max capacity; for receive: if the channel is empty). If the blocking condition is false, the send/receive finishes and exits the infinite loop. Otherwise, it stays in the loop until the blocking condition becomes false.

However, while loops are not a low-level primitive in GooseLang, but recursive functions are. So, it is cleaner to use a recursive function to create an infinite loop. If the blocking condition is false, the send/receive finishes and returns. If the blocking condition is true, the function recursively calls itself on the same arguments, creating a loop.

3.1.3 Zero capacity channels

The last major challenge we faced when designing our channel representation was how to handle zero capacity channels. Sending on a zero capacity channel is allowed; it just requires a receiver to be active at the same time the sender. The problem is how do we tell if a receiver and sender are active at the same time? And what do we do with the sent message?

Our solution comes from the observation that when a receive on a channel is active, the channel effectively behaves as if its capacity were 1 greater. A zero capacity channel with an active receive lets a single send succeed and blocks any further sends, which is identical to how a 1-capacity channel behaves with no active receiver.

To incorporate this idea, we added an extra state parameter, called effective_capacity, to our channel representation. When a channel is initialized, effective_capacity gets set to the channel's capacity. Whenever a receive starts, the effective_capacity increases by 1, and

when a receive completes, the effective_capacity decreases by 1.

While this idea was introduced to handle zero capacity channels, the same logic can also be applied to non-zero capacity channels. Suppose a positive capacity channel is at maximum capacity. Initiating a receive will retrieve the first message from the queue, opening a spot in the message buffer, which would allow a simultaneous send to succeed. So for simplicity, we do not differentiate between zero and non-zero capacity channels.

Under this specification, the number of sends that can complete at any given point is equal to the channel's capacity plus the number of active receivers. This can be sequentialized as all the receives finishing first, and then the sends complete.

One potential issue with having an effective_capacity field that could be larger than the capacity, is how do we handle the maximum possible capacity? The capacity field of Go channels is defined to be a uint64, meaning the highest possible capacity is $2^{64} - 1$. Having an effective_capacity greater than the max int will lead to problems. However, this edge case is extremely impractical in real systems, because the operating system would run out of memory before it could have a channel with $2^{64} - 1$ elements. Since this edge case is very impractical for current systems, our model assumes that the effective_capacity will never overflow a uint64, and the GooseLang functions that change the channel's effective_capacity will assert this assumption.

3.2 Channel Representation

Now that we've discussed the reasoning behind many of our design choices, let's introduce our representation of channels in GooseLang. We propose the following scheme, which is summarized in Figure 3.1 and Figure 3.2.

```
Channel<T> = Nil()
           + Channel(capacity: int, chanref: ChanStruct<T>, lock: lock)
ChanStruct<T> = ClosedChan(contents: ChanCon<T>)
```

```
                + OpenChan(effective_capacity: int, contents: ChanCon<T>)

ChanCon<T> = Empty(zeroVal: T)

           + Cons(first: T, rest: ChanCon<T>)
```

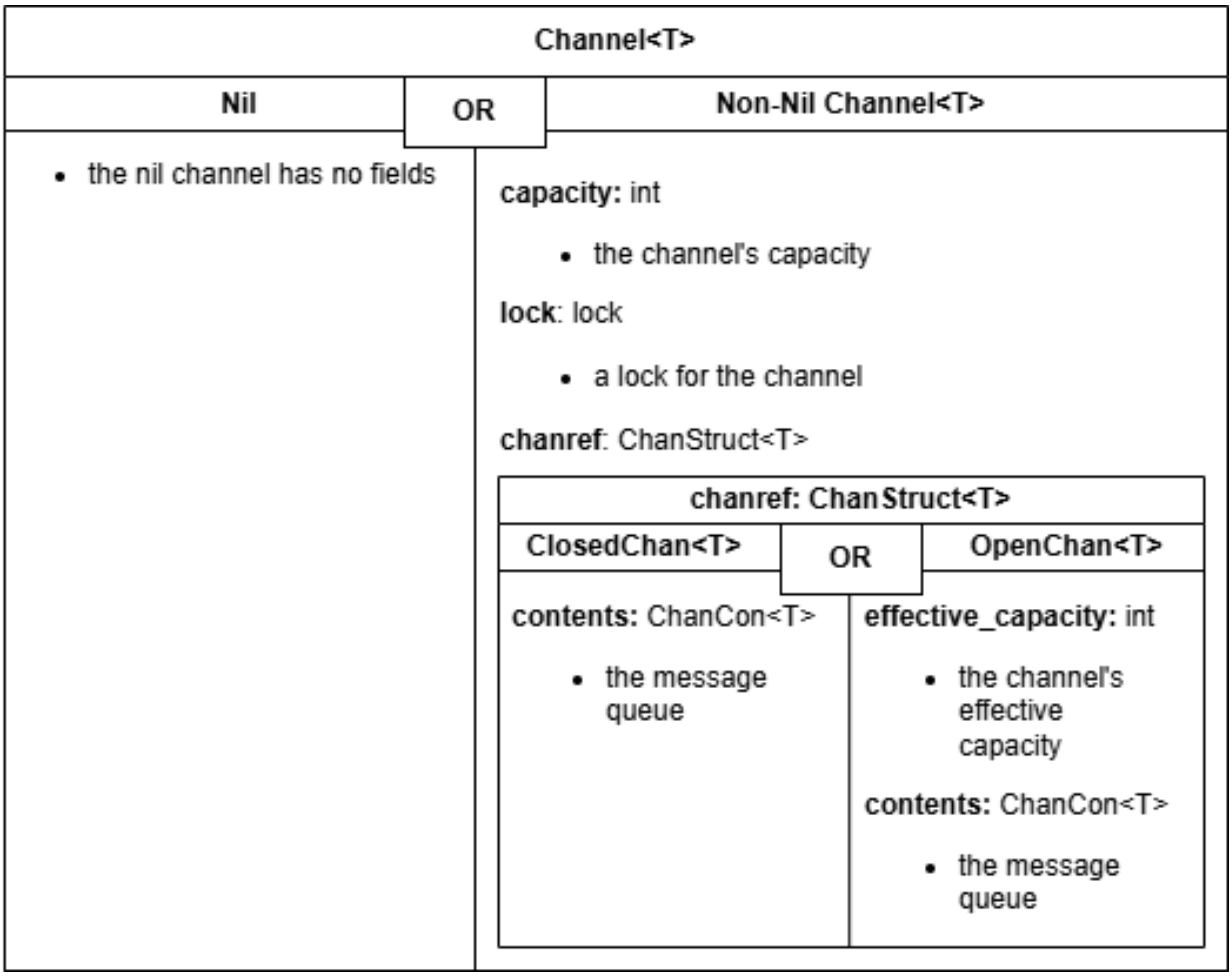

Figure 3.1: The Channel<T> type can be either Nil or Non-Nil. A Non-Nil channel contains the channel's capacity, a lock, and a ChanStruct<T>, which contains the channel's effective capacity and its contents if the channel is open, and contains just the channel's contents if the channel is closed.

Breaking it down, at the top level we have:

```
Channel<T> = Nil()
           + Channel(capacity: int, chanref: ChanStruct<T>, lock: lock)
```

A channel can either be a nil channel or not nil. A non-nil channel contains the channels capacity, a `chanref`, which is a pointer to a `ChanStruct<T>`(explained below) and a lock associated with the channel. The lock will be necessary to make certain channel operations atomic, such as modifying the contents of the channel during a send or receive. Figure 3.1 visualizes the `Channel<T>` struct.

The next level is the `ChanStruct<T>`, which contains information about the channel's current state and the elements in the channel.

```
ChanStruct<T> = ClosedChan(contents: ChanCon<T>)
              + OpenChan(effective_capacity: int, contents: ChanCon<T>)
```

Similar to the `Channel<T>`, `ChanStruct<T>` has 2 states: open and closed. In the closed state, the `ChanStruct<T>` contains just a `ChanCon<T>`, which holds the contents of the channel's message queue. In the open state, the `ChanStruct<T>` contains a `ChanCon<T>`, and an `effective_capacity` field.

The last level is the `ChanCon<T>`, which works as a linked list of the channel contents.

```
ChanCon<T> = Empty(zeroVal:T)
           + Cons(first:T, rest:ChanCon<T>)
```

A `ChanCon<T>` is either `Empty`, which means there are no elements in the channel, or a `Cons`, which contains an element `first` of type `T` and another `Cons` which contains the rest of the contents of the channel. The last element in a chain of `Cons` will always be an `Empty`. The `Empty` type contains the zero value of the type `T` associated with the channel, since a receive on a closed and empty channel should return the zero value.

Below is the declaration of the channel representation in terms of GooseLang values:

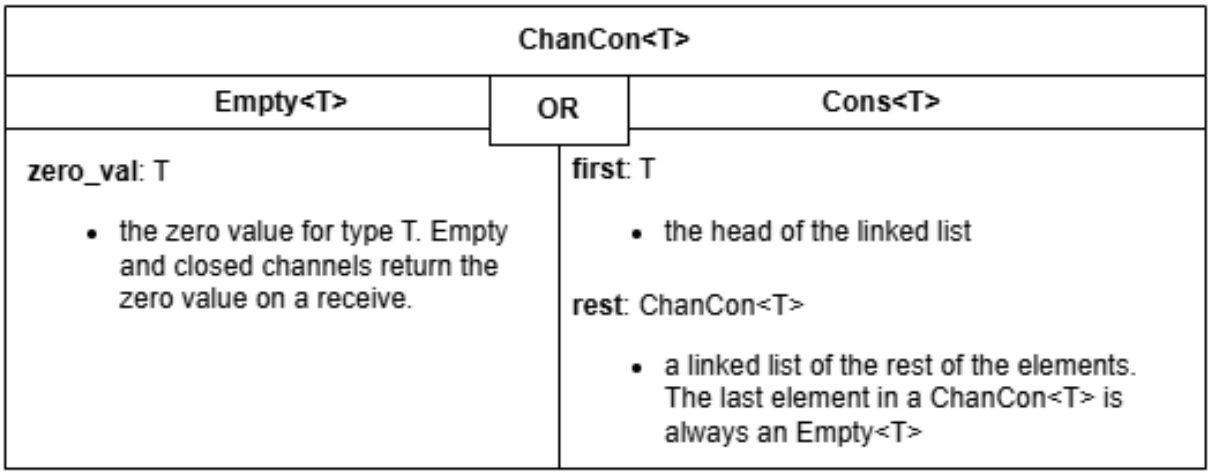

Figure 3.2: The ChanCon<T> type can be either Empty<T> or a Cons<T>. An Empty<T> list has only the zero value for type T. A Cons<T> contains a single element that is the head of the linked list, and another ChanCon<T> that represents the rest of the list. The last element in a ChanCon<T> is always an Empty<T>.

```
Notation NilChannelV := (InjLV #()).

Notation ChannelV cap chanref lock := (InjRV (cap, chanref, lock)).


Notation ChannelClosedV content := (InjLV content).

Notation ChannelOpenV eff_cap content := (InjRV (eff_cap, content)).


Notation ChanConsEmptyV zero_val := (InjLV zero_val).

Notation ChanConsV elem cons:= (InjRV (elem, cons)).
```

3.3 Channel Operations

After building the channel representation, the next step is to implement each of the channel operations discussed in section 2.2.

3.3.1 Make

The built-in `make(chan T, n)` should create an empty, non-nil `Channel` with type `T` and capacity `n`.

```
Definition NewChan (t:ty) : val :=
  λ: "cap",
    let: "chanref" := Alloc (InjR (Var "cap", (ChanConsEmptyV (zero_val t)))) in
    let: "lock" := lock.new #() in
    (InjR (Var "cap", Var "chanref", Var "lock")).
```

In the above code,

`"chanref" := Alloc (InjR (Var "cap", (ChanConsEmptyV (zero_val t))))`

creates an open channel called `chanref` with effective_capacity `cap` and an empty channel contents. The following line creates a new lock, and the function returns the channel comprised of `cap`, `chanref`, and `lock`.

3.3.2 Close

The `close` function takes in a channel and closes it if it is open, and causes a run-time panic if the channel is nil or already closed. We accomplish this behavior by causing a panic if we detect a nil or closed channel. Otherwise, the channel is open and of the form `InjR (eff_cap, content)`. To close it, we replace the chanref with a `InjL content`, representing a closed channel with contents `content`.

```
Definition CloseChan: val :=
  λ: "channel",
    match: "channel" with
      InjL "nullv" ⇒ Panic("close of nil channel")
    | InjR "chan" ⇒
        let: "cap" := Fst (Fst "chan") in
```

```
    let: "chanref" := Snd (Fst "chan") in
    let: "lock" := Snd "chan" in
    lock.acquire "lock";;
    match: !"chanref" with
      InjL "empty" ⇒ Panic ("close closed channel")
    | InjR "open" ⇒
        let: "con" := Snd "open" in
          "chanref" ← InjL "con";;
          lock.release "lock"
    end
  end.
```

3.3.3 Cap

The `cap` function returns `0` if the channel is a nil channel, and otherwise returns the capacity defined during initialization.

```
Definition ChanCap: val :=
  λ: "channel",
    match: "channel" with
      InjL "nullv" ⇒ #0
    | InjR "chan" ⇒ Fst (Fst "chan")
    end.
```

3.3.4 Len

Because of how we handle effective capacity, there are two possible definitions of `len` we could use. The first is the length of a channel as defined in the Go spec. The second is the "effective length" of a channel. Because of our introduction of the "effective capacity", it is possible for the length of a `ChanCon<T>` to be higher than a channel's capacity.

The `len` function as defined by Go's spec should never return a number higher than the channel's capacity, but a channel's effective length will be important to us when we try to make the `send` function. Since we need both definitions of `len`, we built two different `len` functions, one that follows Go's spec, and another that returns the effective length of the channel.

Let's take a look at the effective length function first. We define `ChanLen'`, which takes in a `ChanCon<T>`(the contents of a channel) and returns the length of the list.

```
Definition ChanLen': val :=
  λ: "chancon",
    (rec: "chanLen" "c" :=
      match: "c" with
        InjL "empty" ⇒ #0
      | InjR "content" ⇒ #1 + "chanLen" (Snd "content")
     end) ("chancon").
```

This function recursively calls itself on `ChanCons<T> c`. If `c` is `Empty(zeroVal:T)`, then it returns 0, otherwise `c` is a `Cons(first:T, rest:ChanCon<T>)` and the function returns $1 +$ `ChanLen'(rest)`.

Using `ChanLen'`, we build `ChanLen`, which takes in a `Channel<T>` and returns the length of a channel as defined in Go's spec.

```
Definition ChanLen: val :=
  λ: "channel",
    match: "channel" with
      InjL "nullv" ⇒ #0
    | InjR "chan" ⇒
        let: "cap" := Fst (Fst "chan") in
        let: "chanref" := Snd (Fst "chan") in
        let: "lock" := Snd "chan" in
          lock.acquire "lock";;
```

```
    let: "eff_len" := (let: "con" :=
      (match: !"chanref" with
        InjL "closed" ⇒ "closed"
      | InjR "open" ⇒ Snd "open"
      end) in ChanLen' "con") in
      lock.release "lock";;
      if: ("eff_len" < "cap") then "eff_len"
      else "cap"
end.
```

`ChanLen` returns `0` if the channel is nil. If the channel is not nil, then `ChanLen` acquires the lock for the channel, so that no additional sends and receives can modify the state of the channel, and then calls `ChanLen'` on the channel's contents. The effective length gets stored in the variable `eff_len` and the lock is released.

Finally, `ChanLen` checks if the effective length is less than the capacity; if it is, then `ChanLen` returns `eff_len`, otherwise it returns `cap`. This guarantees that

$$0 \leq \texttt{len(c)} \leq \texttt{cap(c)}$$

The only time this edge case occurs is when the channel is at maximum capacity, and there are simultaneous sends and receives at the same time as the `len` function. In which case, `len` could return multiple values depending on how it interleaves with the sends and receives. In cases where `len` sees effective length $>$ capacity, it is behaviorally equivalent to reorder some of the receives to complete first, then the `len`, then the rest of the operations, such that `len` would return `cap(c)`.

For example, say c is a 0-capacity channel, and there are simultaneously 2 sends, 2 receives, and 1 len. The cases where `len` sees an effective length $>$ capacity is when the sends happen before the `len`, and the receives happen after the `len`. This is behaviorally equivalent to bumping the receives to happen before the `len`, so `len` sees a length of capacity.

3.3.5 Send

The channel send operation is considerably more complicated than the other functions we've discussed so far. The main considerations that went into our design of the send operations were:

1. `Send` must acquire the channel's lock before it is allowed to modify the contents of the channel
2. `Send` must block until it succeeds
3. A send operation is tied to a particular instance of a channel, not the chan variable(ie, a send on a nil channel should block forever, even if the channel variable gets reassigned later)

The combination of constraints 1 and 2 demand that `Send` repeatedly acquires the lock, attempts to send, then releases the lock(regardless of whether it succeeded or not), until the send finally succeeds. The third constraint says we should only read the pointer to the `ChanStruct` and the lock once, and continually reference the same pointer and lock.

With this in mind, let's build the helper function `TrySend`, which does 1 iteration of attempting to send. `TrySend` takes in 3 arguments: `chanref`, the pointer to the `ChanStruct`, `lock`, the lock associated with this instance of the channel, and `v`, the value to send, and returns `true` if the send is successful and `false` if it was not.

```
Definition TrySend: val :=
  λ: "chanref" "lock" "v",
    lock.acquire "lock";;
    match: !"chanref" with
      InjL "empty" ⇒ Panic ("send on closed channel")
    | InjR "open" ⇒
        let: "eff_cap" := Fst "open" in
```

```
    let: "con" := Snd "open" in
    let: "len" := ChanLen' "con" in
    Assume ("eff_cap" < #(2^64 − 1));;
    if: "eff_cap" > "len" then
      "chanref" ← InjR ("eff_cap", ChanAppend "con" "v");;
      lock.release "lock";;#true
    else lock.release "lock";;#false
  end.
```

`TrySend` first acquires the lock for the channel. Then it checks whether the channel is closed or open. If the channel is closed, it panics. Otherwise, it calls `ChanLen'` on the channel's contents to get the effective length and checks if the effective capacity is strictly greater than the length. If it is, then we append `v` to the end of the channel's contents, release the lock, and return `true`. If the effective capacity is not greater than the length, then we just release the lock and return `false`.

Since `TrySend` is an operation that could increase the size of the channel contents, we add the line `Assume ("eff_cap" < #(2^64-1))` to assert our assumption that the effective_capacity will never exceed the max uint64.

The `ChanAppend` function is a helper function that takes in a channel contents and a value to append, appends the value to the end of the contents, and returns the new contents. Since the channel contents is comprised of a nested chain of `(elem, rest_of_content)`, the `ChanAppend` function works by recursively calling `ChanAppend` on the `rest_of_content` until we get to the end of the list, which is a `InjL "empty"`. We replace the `InjL "empty"` with `InjR ("v", InjL "empty")`, which makes `v` the last element of the list.

```
Definition ChanAppend: val :=
  λ: "con" "v",
  (rec: "chanAppend" "con" :=
    match: "con" with
```

```
    InjL "empty" ⇒ InjR ("v", InjL "empty")
  | InjR "elemCon" ⇒
    let: "elem" := Fst "elemCon" in
    let: "con2" := Snd "elemCon" in
      InjR ("elem", "chanAppend" "con2")
  end
) ("con").
```

Finally, we are ready to build `ChannalSend`. `ChannalSend` takes in a channel and a value to send and blocks until it successfully sends the value to the channel. If the channel is a nil channel, then `ChannalSend` should block forever, which we implement using the following infinite loop code:

```
Definition InfiniteLoop: val :=
  (rec: "loop" <> := Var "loop" #()).
```

Otherwise, the channel is not nil, so we extract the `chanref` and `lock`. Then we define a recursive function which runs `TrySend "chanref" "lock" "v"`, and if the `TrySend` returns `true`, `ChannelSend` terminates. Otherwise, the recursive function calls itself again. This causes `ChannelSend` to infinitely loop until `TrySend` succeeds.

```
Definition ChannelSend: val :=
  λ: "channel" "v",
  match: "channel" with
    InjL "nullv" ⇒ InfiniteLoop #()
  | InjR "chan" ⇒
    let: "cap" := Fst (Fst "chan") in
    let: "chanref" := Snd (Fst "chan") in
    let: "lock" := Snd "chan" in
    (rec: "chanSend" "c" :=
      let: "r" := TrySend "c" "lock" "v" in
```

```
        if: "r" then #()
        else "chanSend" "c"
    ) ("chanref")
end.
```

3.3.6 Receive

Finally, we have the receive operation, which we will build incrementally like send. All the considerations that went into the send operation also apply to the receive: the receive operation requires the lock to modify the channel, but should not block other channel operations while it is blocking, and each receive operation is tied to a specific instance of a channel.

To accommodate for 0 capacity channel sends and receives, our specification causes each call to receive on a channel to increase the effective capacity of the channel by 1, and it decreases the effective capacity by 1 once the receive completes. Under this specification, it is possible to do `capacity` + `# of active receivers` send operations at the same time without any of the send operations blocking. In order to increase the effective capacity, we need to acquire the channel's lock, but sends can't happen while we are holding the lock. Therefore, we need to release the lock after increasing effective capacity to give sends an opportunity to finish, and reacquire the lock before attempting to receive.

In pseudocode, receive should do the following:

```
while(1):
    acquire lock
    increase effective_capacity
    release lock
    acquire lock
    ret, success = TryReceive
    decrease effective_capacity
```

```
        release lock
        if(success) return ret
```

Let's take a look at the increase and decrease effective_capacity functions first.

```
Definition IncCap: val :=
  λ: "chanref",
    match: !"chanref" with
      InjL "closed" ⇒ #()
    | InjR "open" ⇒
        let: "eff_cap" := Fst "open" in
        let: "con" := Snd "open" in
        Assume ("eff_cap" < #(2^64 − 1));;
        "chanref" ← InjR (#1 + "eff_cap", "con")
    end.

Definition DecCap: val :=
  λ: "chanref",
    match: !"chanref" with
      InjL "closed" ⇒ #()
    | InjR "open" ⇒
      let: "eff_cap" := Fst "open" in
      let: "con" := Snd "open" in
      Assume ("eff_cap" > #0);;
      "chanref" ← InjR("eff_cap" − #1, "con")
    end.
```

Both the `IncCap` and `DecCap` functions take in a `ChanStruct<T>` pointer. If the channel is closed, the function does nothing. Otherwise, the channel is open, and the function extracts the capcity, the effective_capacity, and the contents of the channel and replaces the `ChanStruct<T>` with another `ChanStruct<T>` with the same capacity and contents, and

effective_capacity incremented/demcremented by 1.

Both `IncCap` and `DecCap` modify the value of effective_capacity, and so must have `Assume` statements to guarantee that the effective_capacity will not overflow or underflow a uint64.

The next step in receive is performing a single receive attempt. For this, we define another helper function, `InnerReceive`, which takes in a `ChanStruct<T>` pointer and returns the value from receive, whether the channel is closed and empty(the optional second return value of a receive operation), and whether the receive succeeded or not.

```
Definition InnerReceive: val :=
  λ: "chanref",
    match: !"chanref" with
      InjL "closed" ⇒
        match: "closed" with
          InjL "empty" ⇒ ("empty", #false, #true)
        | InjR "con" ⇒
            let: "elem" := Fst "con" in
            let: "rest" := Snd "con" in
              "chanref" ← InjL("rest");;
              ("elem", #true, #true)
        end
    | InjR "open" ⇒
        let: "eff_cap" := Fst "open" in
        let: "con" := Snd "open" in
        match: "con" with
          InjL "empty" ⇒ ("empty", #false, #false)
        | InjR "con" ⇒
            let: "elem" := Fst "con" in
            let: "rest" := Snd "con" in
              "chanref" ← InjR("eff_cap", "rest");;
```

```
            ("elem", #true, #true)
        end
    end.
```

`InnerReceive` takes in a `ChanStruct<T>` pointer. If the channel is closed and empty, it returns the zero value of type `T`, `false` for the second return value(since the channel is both closed and empty), and `true` for the third return value since the receive successfully completed. If the channel is closed and not empty, then it returns the first element in contents and sets the channel's contents to the rest of the list.

If the channel is open and empty, then it should be blocking. So it returns `false` for the third return and the other 2 return values don't matter, but will be defaulted to the zero value and `false`. Finally, if the channel is open and not empty, then it also returns the first element in contents and sets the channel's contents to the rest of the list.

Using `InnerReceive`, we now build `TryReceive`. Similar to `TrySend`, `TryReceive` takes in `chanref`, the pointer to the `ChanStruct` and `lock`, the lock associated with this instance of the channel. `TryReceive` then wraps a call to `InnerReceive` in between an `IncCap` and a `DecCap`.

```
Definition TryReceive: val :=
  λ: "chanref" "lock",
    lock.acquire "lock";;
    IncCap "chanref";;
    lock.release "lock";;
    lock.acquire "lock";;
    let: "r" := InnerReceive "chanref" in
      DecCap "chanref";;
      lock.release "lock";;
      "r".
```

`TryReceive` makes sure to acquire the lock before increasing the effective capacity, but

then releases the lock immediately after. This is to give a potential sender the opportunity to send a value on the channel now that the effective capacity has been increased. `TryReceive` then performs a `InnerReceive` on the channel, decrements the effective capacity, releases the lock, and returns the result of the `InnerReceive`.

Finally, we are ready to make `ChannelReceive`, which performs a blocking receive. Similar to `ChannelSend`, `ChannelReceive` keeps recursively calling `TryReceive`, until `TryReceive` is finally successful.

```
Definition ChannelReceive: val :=
  λ: "channel",
  match: "channel" with
    InjL "nullV" ⇒ InfiniteLoop #()
  | InjR "chan" ⇒
      let: "cap" := Fst (Fst "chan") in
      let: "chanref" := Snd (Fst "chan") in
      let: "lock" := Snd "chan" in
      (rec: "chanRec" "c" :=
        let: "r" := TryReceive "c" "lock" in
        let: "v" := Fst (Fst ("r")) in
        let: "non-empty" := Snd (Fst ("r")) in
        let: "valid" := Snd "r" in
          if: "valid" then ("v", "non-empty")
          else "chanRec" "c"
      ) ("chanref")
  end.
```

If the channel is nil, `ChannelReceive` enters an infinite loop. Otherwise, it extracts the channel pointer and lock and calls `TryReceive` until the third return from `TryReceive` is true(the receive is successful). Once the `TryReceive` succeeds, `ChannelReceive` returns the first two return values of `TryReceive`.

Chapter 4

Translating Go Code to GooseLang

Now that we have a framework to represent channels and channel operations, the next step is to build a translator from the Go code to GooseLang. This chapter describes how the Goose translator handles channels.

4.1 The Goose Translator

The Goose translator translates Goose code into GooseLang, and the majority of it was written by Tej Chajed as part of his thesis [10]. The code for the translator can be found here: https://github.com/goose-lang/goose. The Goose translator makes heavy use of the go/ast and go/types packages from the Go standard library. This both simplifies the parsing step, and ensures that we parse Go code in the same manner that Go does.

The full documentation of the ast package can be found here: https://pkg.go.dev/go/ast. For our purposes, we will be paying particular attention to the `type ChanType`, `type SendStmt`, and the `token.ARROW`.

4.2 Channel Declarations and the Make function

Channel declarations use the same structure as all other declarations. Variable declarations have `type ValueSpec` and are handled using the following function:

```
func (ctx Ctx) varSpec(s *ast.ValueSpec) coq.Binding {
    lhs := s.Names[0]
    ...
    var rhs coq.Expr
    if len(s.Values) == 0 {
        ty := ctx.typeOf(lhs)
        rhs = coq.NewCallExpr(coq.GallinaIdent("ref"),
            coq.NewCallExpr(coq.GallinaIdent("zero_val"), ctx.coqTypeOfType(s, ty)))
    } else {
        rhs = ctx.referenceTo(s.Values[0])
    }
    return coq.Binding{
        Names: []string{lhs.Name},
        Expr:  rhs,
    }
}
```

If the declaration has no initialization, the variable gets set to a reference of the zero value of that type. In the case of channels, it gets set to the nil channel. If the declaration comes with an initialization, then the variable gets set to a reference of the right hand expression.

Channel initialization is done through the make function, which is handled in the following code along with slices and maps:

```
func (ctx Ctx) makeExpr(args []ast.Expr) coq.CallExpr {
```

```
    ...
    switch ty := ctx.typeOf(args[0]).Underlying().(type) {
    ...
    case *types.Chan:
        return coq.NewCallExpr(coq.GallinaIdent("NewChan"),
            ctx.coqTypeOfType(args[0], ty.Elem()), ctx.expr(args[1]))
    default:
            ...
    }
    return coq.CallExpr{}
}
```

The `makeExpr` function cases on the type of the first argument, and in the case of a channel, it creates a new channel in GooseLang using a call to the GooseLang function `NewChan`.

4.3 Cap and Len

`cap`, `len`, and `make` are all examples of `type CallExpr`, which are handled in the following function:

```
func (ctx Ctx) callExpr(s *ast.CallExpr) coq.Expr {
    if isIdent(s.Fun, "make") {
 return ctx.makeExpr(s.Args)
    }
    if isIdent(s.Fun, "len") {
 return ctx.lenExpr(s)
    }
    if isIdent(s.Fun, "cap") {
```

```
 return ctx.capExpr(s)
    }
    ...
```

In each of these cases, the function gets translated to GooseLang by calling the GooseLang equivalent of the function.

4.4 Send and Receive

Sends are their own type of statements, the `type SendStmt`. When a send statement is found, it calls the following function, which replaces the send statement with tbe GooseLang function `ChannelSend`.

```
func (ctx Ctx) sendStmt(s *ast.SendStmt) coq.Binding {
    channel := s.Chan
    value := ctx.expr(s.Value)
    return ctx.channelSend(channel, value)
}

func (ctx Ctx) channelSend(channel ast.Expr, value coq.Expr) coq.Binding {
    return coq.NewAnon(coq.NewCallExpr(
        coq.GallinaIdent("ChannelSend"),
        ctx.expr(channel), value))
}
```

Receives on the other hand are a type of unary expression. Receives are also unique in that they can have 2 return values: the value returned and whether the return came from a non-empty channel. Consequently, the `unaryExpr` function had to be expanded to include a second argument, `isSpecial`, which is set to true if a second return is detected. When `isSpecial` is false, the return of `ChannelReceive` is truncated to only return the first value.

```
func (ctx Ctx) unaryExpr(e *ast.UnaryExpr, isSpecial bool) coq.Expr {
    ...
    if e.Op == token.ARROW {
        e2 := ctx.receiveExpr(e)
        if !isSpecial {
            e2 = coq.NewCallExpr(coq.GallinaIdent("Fst"), e2)
        }
        return e2
    }
    ...
}

func (ctx Ctx) receiveExpr(e *ast.UnaryExpr) coq.Expr {
    return coq.NewCallExpr(
        coq.GallinaIdent("ChannelReceive"),
        ctx.expr(e.X))
}
```

4.5 Future Work: Select statements

There is one last Go feature that interacts with channels, which is the select statement. A select statement consists of several cases, which are allowed to contain channel operations like send and receive. The select statement blocks until any case is allowed to proceed. If multiple cases are allowed to proceed, it randomly chooses one to finish.

We have decided not to extend Goose to support select statements at this time. Chapter 8 will discuss the possibility of adding support for select statements in more detail.

Chapter 5

Separation Logic Specifications

With the ability to translate Go code containing channels into GooseLang, it is now possible to start building proofs. As part of this thesis, we build a simple library of proofs around the correctness of Go's base channel operations. This library can be used and expanded upon to make more complicated proofs. This chapter introduces the specifications of our proofs.

5.1 Definitions and Invariants

Given the setup of our proofs, it will be useful to first define a few invariants about channels, and also a better way to describe a channel's current state.

Most of the proofs we provide use one of the following invariants:

```
Definition is_channel_alloc (chanref: loc) lk (closed: bool) ty (P: val → iProp ∑): iProp ∑ :=
    is_lock nroot lk (∃ eff_cap l, own_chan chanref eff_caf closed ty l P).

Definition own_chan (chanref: loc) (eff_cap: Z) (closed: bool) ty (l: list val) (P: val → iProp ∑):
        iProp ∑ :=
    (if closed then chanref ↦ ChannelClosedV (chan_contents (zero_val ty) l)
     else chanref ↦ ChannelOpenV #eff_cap (chan_contents (zero_val ty) l))%I ∗
    ⌜ has_zero ty ⌝ ∗ ([∗ list] _ ↦ elem ∈ l, P elem ∗ ⌜ val_ty elem ty ⌝) ∗
```

```
    ⌜ length l < 2^64 ⌝ ∗ ⌜ 0 ≤ eff_cap < 2^64 ⌝.

Definition is_channel c closed ty (P: val → iProp Σ): iProp Σ :=
    ⌜ c = InjLV #() ⌝ ∗ ⌜ has_zero ty ⌝ ∨
    (∃ cap chanref lk, is_channel_alloc chanref lk closed ty P ∗
    ⌜ c = InjRV (#cap, #chanref, lk) ⌝ ∗ ⌜ 0 ≤ cap < 2^64⌝).
```

These invariants are complicated and contain many subpieces. Let's build up to them from the very basics: the definition of a channel.

A channel can either be nil or non-nil. The nil channel has no fields and is equal to `InjLV #()`. A non-nil channel is defined by a uint64 capacity, a pointer to a `ChanStruct<T>`, and a lock. Combining these facts, we get the following invariant:

```
Definition is_channel c closed ty (P: val → iProp Σ): iProp Σ :=
    ⌜ c = InjLV #() ⌝ ∨ (∃ cap chanref lk, is_channel_alloc chanref lk closed ty P ∗
    ⌜ c = InjRV (#cap, #chanref, lk) ⌝ ∗ ⌜ 0 ≤ cap < 2^64⌝).
```

A non-nil channel's capacity, `ChanStruct<T>` pointer, and lock never change unless the channel is reassigned to a different channel. Meanwhile, the values inside the `ChanStruct<T>` change often. Our current invariant does not give us a way to talk about the values inside the `ChanStruct<T>`, so let's make a new invariant.

A non-nil channel can either be open or closed. A closed channel is represented by a `ChannelClosedV` which only contains a `ChanCon<T>`. An open channel is represented by a `ChannelOpenV` which contains an effective capacity along with a `ChanCon<T>`. A `ChanCon<T>` is just another way of representing a list. The following Fixpoint turns a list of vals l with associated zero value `zero` into a `ChanCon<T>`.

```
Fixpoint chan_contents (zero: val) (l : list val): val :=
    match l with
    | nil ⇒ InjLV zero
    | cons x l' ⇒ InjRV (x, (chan_contents zero l'))
```

```
end.
```

We now define the invariant `own_chan`, which says given a pointer `chanref`, an integer `eff_cap`, a boolean `closed`, a type `ty`, and a list of values `l`, if `closed` is true, then `chanref` points to a `ChannelClosedV` with contents `chan_contents (zero_val ty) l`. If `closed` is false, then `chanref` points to a `ChannelOpenV` with effective capacity `eff_cap` and contents also equal to `chan_contents (zero_val ty) l`.

```
Definition own_chan (chanref: loc) (eff_cap: Z) (closed: bool) ty (l: list val): iProp Σ :=
  (if closed then chanref ↦ ChannelClosedV (chan_contents (zero_val ty) l)
   else chanref ↦ ChannelOpenV #eff_cap (chan_contents (zero_val ty) l))%I.
```

Between our current definitions of `is_channel` and `own_chan`, these give the invariant that a channel will always follow the structure outlined in Section 3.2. Now, let's add 3 more properties to this invariant.

All elements in a channel will always share the same type as the channel's type. In other words, for a list of values `l` and type `ty`, every element `elem` in `l` has the following property:

```
[∗ list] _ ↦ elem ∈ l, ⌜ val_ty elem ty⌝
```

Additionally, all valid channel types have a defined zero value(`0` for ints, `false` for booleans, `[]` for arrays, etc.). So, for the given type `ty`,

```
⌜ has_zero ty ⌝
```

Finally, we know that the length of the list `l` and the value `eff_cap` must be uint64s. The lengths of lists is already implicitly at least 0. So, we should add the following to our invariant:

```
⌜ length l < 2^64 ⌝ ∗ ⌜ 0 ≤ eff_cap < 2^64 ⌝
```

Putting these all together, we get the following definition of `own_chan`:

```
Definition own_chan (chanref: loc) (eff_cap: Z) (closed: bool) ty (l: list val): iProp Σ :=
  (if closed then chanref ↦ ChannelClosedV (chan_contents (zero_val ty) l)
```

```
  else chanref ↦ ChannelOpenV #eff_cap (chan_contents (zero_val ty) l))%I ∗
⌜ has_zero ty ⌝ ∗ ([∗ list] _ ↦ elem ∈ l, ⌜ val_ty elem ty ⌝ ∗
⌜ length l < 2^64 ⌝ ∗ ⌜ 0 ≤ eff_cap < 2^64 ⌝).
```

`own_chan` is useful for describing how single channel operations affect the channel. For example, the increase effective capacity helper function increases the effective capacity by 1 when the channel is open and keeps everything else the same, which we can express using the following theorem:

```
Theorem wp_IncCap_Open (chanref: loc) (eff_cap: Z) ty (l: list val):
    {{{ own_chan chanref eff_cap false ty l}}}
        IncCap #chanref
    {{{ RET #(); own_chan chanref (eff_cap + 1) false ty l}}}
```

`own_chan` is a very powerful tool when we know exactly how our channel changes. However, once you introduce multiple threads, it becomes impossible to know how exactly a channel behaves, and `own_chan` becomes difficult to use. The above theorem only works because `IncCap` is always wrapped inside locked code, so nothing else is allowed to change the channel during an `IncCap`.

To solve this issue, we introduce `is_channel_alloc` as a more generalized version of the `own_chan` invariant.

```
Definition is_channel_alloc (chanref: loc) lk ty: iProp Σ :=
  is_lock nroot lk (∃ eff_cap l closed, own_chan chanref eff_cap closed ty l).
```

`is_lock nroot lk I` gives the guarantee that the invariant `I` will be true when the lock `lk` is acquired, and that `I` will also be true when `lk` is released. So, `is_channel_alloc` says that for a pointer `chanref`, lock `lk`, and type `ty`, on acquiring the channel's lock, there will exist some value of `eff_cap`, `l`, and `closed` such that `own_chan chanref eff_cap closed ty l` is true. And when the lock is released, there will be (potentially different) values of `eff_cap`, `l`, and `closed` such that `own_chan chanref eff_cap closed ty l` is still true.

This definition of `is_channel_alloc` allows a channel's parameters to change in the middle of channel operations, which accurately models how channels behave, but introduce other problems when it comes to proofs. Namely, sends require an open channel to succeed. Sending on a closed channel causes a Panic, and sending on a nil channel causes an infinite loop, both of which should be avoided, and it would be useful to be able to prove that a piece of code will not cause a Panic or an infinite loop.

Proving that a channel is not nil is easy, since channels never go from allocated to nil, so you only need to prove that the channel gets allocated before the send is called. Proving that a channel is open works similarly, since an allocated channel always starts as open. However, there does exist an operation that closes channels, and our current definition of `is_channel_alloc` allows for a channel to get closed at any time, which makes it difficult to prove that a channel will be open at the beginning of a send.

Our work-around for this problem is to take out the `closed` bool from the exists statement, to get the following:

```
Definition is_channel_alloc (chanref: loc) lk (closed: bool) ty: iProp Σ :=
  is_lock nroot lk (∃ eff_cap l, own_chan chanref eff_cap closed ty l).
```

This version of `is_channel_alloc` now preserves the value of `closed` between acquiring and releasing the lock, which is true for all channel operations besides `close`. Since the make function returns an initially open channel, as long as no close functions are called, a channel will always remain open. This provides a good framework for making proofs about channels that are always open, but present some obvious problems with trying to close channels now. This will be discussed more in the limitations and future work section.

Putting it all together, we have the following three definitions:

```
Definition is_channel_alloc (chanref: loc) lk (closed: bool) ty: iProp Σ :=
    is_lock nroot lk (∃ eff_cap l, own_chan chanref eff_caf closed ty l).

Definition own_chan (chanref: loc) (eff_cap: Z) (closed: bool) ty (l: list val): iProp Σ :=
```

```
    (if closed then chanref ↦ ChannelClosedV (chan_contents (zero_val ty) l)
     else chanref ↦ ChannelOpenV #eff_cap (chan_contents (zero_val ty) l))%I ∗
    ⌜ has_zero ty ⌝ ∗ ([∗ list] _ ↦ elem ∈ l, ⌜ val_ty elem ty ⌝ ∗
    ⌜ length l < 2^64 ⌝ ∗ ⌜ 0 ≤ eff_cap < 2^64 ⌝).

Definition is_channel c closed ty: iProp Σ :=
    ⌜ c = InjLV #() ⌝ ∗ ⌜ has_zero ty ⌝ ∨ (∃ cap chanref lk, ⌜ c = InjRV (cap, #chanref, lk) ⌝
    ∗ is_channel_alloc chanref lk closed ty ∗ ⌜ 0 ≤ cap < 2^64⌝).
```

Our final version of these definitions makes one more addition. Currently, none of these invariants say very much about the contents of the channel, besides that they have the type `ty`. To make meaningful proofs, it will often be necessary to make claims about the elements in a channel, which should be incorporated into the invariants.

We introduce property `P: val -> iProp Σ`, which is a function that maps values to true/false expressions, and we add the following statement, which says that every element in the list has property `P`, to `own_chan`.

```
[∗ list] _ ↦ elem ∈ l, ⌜ val_ty elem ty ⌝ ∗ P elem
```

Here are our final invariant definitions:

```
Definition own_chan (chanref: loc) (eff_cap: Z) (closed: bool) ty (l: list val) (P: val → iProp Σ):
        iProp Σ :=
    (if closed then chanref ↦ ChannelClosedV (chan_contents (zero_val ty) l)
     else chanref ↦ ChannelOpenV #eff_cap (chan_contents (zero_val ty) l))%I ∗
    ⌜ has_zero ty ⌝ ∗ ([∗ list] _ ↦ elem ∈ l, ⌜ val_ty elem ty ⌝ ∗ P elem ∗
    ⌜ length l < 2^64 ⌝ ∗ ⌜ 0 ≤ eff_cap < 2^64 ⌝).

Definition is_channel_alloc (chanref: loc) lk (closed: bool) ty (P: val → iProp Σ): iProp Σ :=
    is_lock nroot lk (∃ eff_cap l, own_chan chanref eff_caf closed ty l P).
```

```
Definition is_channel c closed ty (P: val → iProp Σ): iProp Σ :=
  ⌜ c = InjLV #() ⌝ ∗ ⌜ has_zero ty ⌝ ∨ (∃ cap chanref lk, ⌜ c = InjRV (cap, #chanref, lk) ⌝
  ∗ is_channel_alloc chanref lk closed ty P ∗ ⌜ 0 ≤ cap < 2^64⌝).
```

Now let's move on to the proof specifications we provide for the various channel operations.

5.2 Make

The make function takes in a capacity and a type, and returns an open channel with that capacity and type. The postcondition of our proof should include the second half of the `is_channel` invariant associated with open channels: `is_channel_alloc chanref lk false ty P ∗ ⌜c = InjRV (cap, #chanref, lk)⌝`. As for the precondition, the make function has 2 restrictions: the type associated with the channel should have a defined zero value, and the capacity should be within the range of a uint64.

```
Theorem wp_NewChan E ty (cap : Z) P:
  {{{ ⌜ has_zero ty ⌝ ∗ ⌜ 0 <= cap < 2^64⌝}}}
    NewChan ty #(cap) @ E
  {{{ c chanref lk, RET (c); is_channel_alloc chanref lk false ty P ∗
        ⌜ c = InjRV (#cap, #chanref, lk) ⌝}}}.
```

5.3 Cap

The capacity function is a simple function that returns `0` if the channel is nil and the capacity otherwise. To capture both cases, we use `is_channel` as our precondition. The function returns an integer `cap`, and will fall into one of two cases: either the capacity is `0` and the channel is nil, or `cap` is equal to the first field in the channel.

```
Theorem wp_ChanCap c closed ty (P : val → iProp Σ):
  {{{ is_channel c closed ty P }}}
```

```
  ChanCap c
{{{ (cap: val), RET (cap); (⌜ #cap = #0 ⌝ ∗ ⌜ c = InjLV #() ⌝) ∨
  (∃ chanref lk, ⌜ c = InjRV (cap, #chanref, lk) ⌝)}}}.
```

5.4 Len

The `len` function is very similar to the `cap` function, which results in their proof specifications looking very similar as well. The `len` theorem uses the same `is_channel` precondition, and the postcondition says that either the returned `len` is `0` and the channel is nil, or the channel is not nil and $0 \leq \texttt{len} \leq \texttt{cap}$.

```
Theorem wp_ChanLen c closed ty (P : val → iProp ∑):
  {{{ is_channel c closed ty P }}}
    ChanLen c
  {{{ (len: Z), RET #(len); (⌜ #len = #0 ⌝ ∗ ⌜ c = InjLV #() ⌝) ∨
    (∃ (cap: Z) chanref lk, ⌜ c = InjRV (#cap, #chanref, lk) ⌝ ∗ ⌜ 0 ≤ len ≤ cap ⌝)}}}.
```

5.5 Send

The send function "can" be called on nil and closed channels, but sending on a nil channel causes an infinite loop, and sending on a closed channel causes a Panic. Neither is a particularly interesting or useful case, so our send theorem requires an open channel as part of the precondition. The send function has no return value, and no postcondition other than making sure the `is_channel_alloc` invariant is still true at the end of the send. As part of that proof, the element added to the channel will need to satisfy `P` and be of type `ty`, and so should be part of the precondition.

This gives us the following theorem:

```
Theorem wp_ChannelSend (cap: Z) (chanref : loc) (v : val) lk ty (P : val → iProp ∑):
```

```
{{{ is_channel_alloc chanref lk false ty P ∗ P v ∗ ⌜val_ty v ty⌝ }}}
    ChannelSend (InjRV(cap, #chanref, lk)) v
{{{RET #(); True}}}.
```

5.6 Receive

The last theorem we provide as part of our library is for the receive operation. Similar to send, we require the channel be non-nil, because again the nil case is not very interesting, but this time the channel is allowed to be closed. This gives us a precondition of `is_channel_alloc chanref lk closed ty P`.

The receive operation gives 2 return values. The first is the received element, and the second is a boolean that is true if the received element was from a non-empty list(channel is open or channel is closed, but not empty). If the boolean is false, which indicates that the channel is closed and empty, then the returned value should be the zero value of `ty`. In the non-empty case, since the returned element `a` was part of the channel, we can also add `P a` as part of the postcondition.

```
Theorem wp_ChannelReceive cap (chanref : loc) lk closed ty P:
    {{{ is_channel_alloc chanref lk closed ty P}}}
      ChannelReceive (InjRV(cap, #chanref, lk))
    {{{(a : val) (ok : bool),
      RET ((a, #(ok))); ⌜ val_ty a ty ⌝ ∗ if (ok) then P a else ⌜ a = zero_val ty ⌝}}}.
```

Chapter 6

Proofs

The full implementation of channels and channel operations, as well as the proofs for the theorems specified in Chapter 5 can be found at:

https://github.com/jzhang025/perennial/tree/final/src/goose_lang/lib/channel.

The file implementing channels and channel operations contains around 200 lines, while the proofs for all the theorems uses 600 lines of Coq. This chapter will go over some of the more interesting proofs and lemmas used.

6.1 Send

As a reminder, the send function and theorem are copied below:

```
Definition ChannelSend: val :=
  λ: "channel" "v",
  match: "channel" with
    InjL "nullv" ⇒ InfiniteLoop #()
  | InjR "chan" ⇒
      let: "cap" := Fst (Fst "chan") in
      let: "chanref" := Snd (Fst "chan") in
      let: "lock" := Snd "chan" in
```

```
    (rec: "chanSend" "c" :=
      let: "r" := TrySend "c" "lock" "v" in
        if: "r" then #()
        else "chanSend" "c"
    ) ("chanref")
  end.

Theorem wp_ChannelSend (cap: Z) (chanref : loc) (v : val) lk ty (P : val → iProp Σ):
    {{{ is_channel_alloc chanref lk false ty P ∗ P v ∗ ⌜val_ty v ty⌝ }}}
        ChannelSend (InjRV(cap, #chanref, lk)) v
    {{{RET #(); True}}}.
```

The most difficult part of this proof is proving that the channel still satisfies `is_channel_alloc` after `v` is added, and that the function can terminate. The `ChannelSend` function repeatedly calls the `TrySend` function until `TrySend` eventually returns true. `TrySend` checks whether the effective capacity is greater than the channel's length. If it is, then it appends `v` to the end of the list using the helper function `ChanAppend` and returns true, otherwise, it does nothing and returns false.

```
Definition TrySend: val :=
  λ: "chanref" "lock" "v",
    lock.acquire "lock";;
    match: !"chanref" with
      InjL "empty" ⇒ Panic ("send on closed channel")
    | InjR "open" ⇒
        let: "eff_cap" := Fst "open" in
        let: "con" := Snd "open" in
        let: "len" := ChanLen' "con" in
        Assume ("eff_cap" < #(2^64 − 1));;
        if: "eff_cap" > "len" then
```

```
        "chanref" ← InjR ("eff_cap", ChanAppend "con" "v");;
        lock.release "lock";;#true
      else lock.release "lock";;#false
  end.

Definition ChanAppend: val :=
  λ: "con" "v",
  (rec: "chanAppend" "con" :=
    match: "con" with
      InjL "empty" ⇒ InjR ("v", InjL "empty")
    | InjR "elemCon" ⇒
      let: "elem" := Fst "elemCon" in
      let: "con2" := Snd "elemCon" in
        InjR ("elem", "chanAppend" "con2")
    end
  ) ("con").
```

Since the send function will call `TrySend` and `ChanAppend` many times, we made lemmas for these functions. For `ChanAppend`, which takes in a list of values `l`, formated as a `ChanCon<T>`, and a value `v`, it should return the `ChanCon<T>` of `l` + +[v].

```
Lemma wp_ChanAppend ty (l : list val) (v : val):
    {{{ True }}}
        ChanAppend (chan_contents (zero_val ty) l) v
    {{{RET (chan_contents (zero_val ty) (l ++ [v])); True}}}.
```

The proof itself of `wp_ChanAppend` is fairly straightforward and simple after inducting on the list `l`.

The theorem for `TrySend` should do most of the work for `wp_ChannelSend`, since the bulk of what `ChannelSend` does is call `TrySend` until it returns true. As such, the precondition for `wp_TrySend` is the same as the precondition for `wp_ChannelSend`. There is no postcondition

for `wp_ChannelSend`, which means if `TrySend` succeeds, it also has no postcondition. However, if `TrySend` fails, it should at least preserve the truthiness of the precondition, so that it can try again. `is_channel_alloc` unfolds to an `is_lock` statement, which will automatically force us to prove that the `is_channel_alloc` invariant holds, so it does not need to be added to the postcondition, but the rest of the precondition should be added.

```
Theorem wp_TrySend (chanref : loc) (v : val) lk ty P:
    {{{ is_channel_alloc chanref lk false ty P ∗ P v ∗ ⌜ val_ty v ty ⌝ }}}
        TrySend #chanref lk v
    {{{(success : bool), RET (#(success)); if success then True else P v ∗ ⌜ val_ty v ty ⌝ }}}.
```

The proof for `wp_TrySend` is lengthy, but again fairly straightforward after casing on whether the if statement is taken or not, and applying the `wp_ChanAppend` lemma.

Finally, we can move on to the `wp_ChannelSend`. Since `wp_TrySend` already guarantees that the `is_channel_alloc` invariant is held, the only part left to prove is that `ChannelSend` can eventually terminate. For this, we use the iLöb induction tactic, which simplifies inducting on recursive functions.

6.2 Receive

The proof for `wp_ChannelReceive` will look very similar to the proof for `wp_ChannelSend`, since they follow the same format. Though, the proof for receive is a bit more complicated than the proof for send since we do not require that the channel is open for `wp_ChannelReceive`.

```
Definition ChannelReceive: val :=
  λ: "channel",
  match: "channel" with
    InjL "nullV" ⇒ InfiniteLoop #()
  | InjR "chan" ⇒
      let: "cap" := Fst (Fst "chan") in
      let: "chanref" := Snd (Fst "chan") in
```

```
    let: "lock" := Snd "chan" in
    (rec: "chanRec" "c" :=
      let: "r" := TryReceive "c" "lock" in
      let: "v" := Fst (Fst ("r")) in
      let: "non-empty" := Snd (Fst ("r")) in
      let: "valid" := Snd "r" in
        if: "valid" then ("v", "non-empty")
        else "chanRec" "c"
    ) ("chanref")
  end.

Theorem wp_ChannelReceive cap (chanref : loc) lk closed ty P:
    {{{ is_channel_alloc chanref lk closed ty P}}}
      ChannelReceive (InjRV(cap, #chanref, lk))
    {{{(a : val) (ok : bool),
      RET ((a, #(ok))); ⌜ val_ty a ty ⌝ ∗ if (ok) then P a else ⌜ a = zero_val ty ⌝}}}.
```

Again, we will want to make lemmas for all the helper functions of `ChannelReceive`, which include `IncCap`, `DecCap`, `InnerReceive`, and `TryReceive`.

6.2.1 IncCap and DecCap

The receive operation needs to increase/decrease the effective capacity, which is what the `IncCap` and `DecCap` functions do.

```
Definition IncCap: val :=
  λ: "chanref",
    match: !"chanref" with
      InjL "closed" ⇒ #()
    | InjR "open" ⇒
        let: "eff_cap" := Fst "open" in
```

```
    let: "con" := Snd "open" in
    Assume ("eff_cap" < #(2^64 − 1));;
    "chanref" ← InjR (#1 + "eff_cap", "con")
  end.

Definition DecCap: val :=
  λ: "chanref",
    match: !"chanref" with
      InjL "closed" ⇒ #()
    | InjR "open" ⇒
      let: "eff_cap" := Fst "open" in
      let: "con" := Snd "open" in
      Assume ("eff_cap" > #0);;
      "chanref" ← InjR("eff_cap" − #1, "con")
    end.
```

Since `IncCap` and `DecCap` work directly with the `ChanStruct<T>` pointer, and not the channel, they do not have access to the lock(they should only be called in sections of locked code), which means the invariants for `IncCap` and `DecCap` should be using `own_chan` instead of `is_channel_alloc`. Since `IncCap` and `DecCap` are only called in sections of locked code, we are also allowed to write a stronger postcondition, since the values of `eff_cap`, `closed`, and `l` cannot change unexpectantly.

```
Lemma wp_IncCap chanref (eff_cap: Z) closed ty l (P : val → iProp Σ):
  {{{ own_chan chanref eff_cap closed ty l P}}}
      IncCap #chanref
  {{{ RET #(); if closed then own_chan chanref eff_cap closed ty l P
               else own_chan chanref (int.Z (word.add 1 eff_cap)) closed ty l P}}}.

Lemma wp_DecCap chanref (eff_cap: Z) closed ty l (P : val → iProp Σ):
```

```
{{{ own_chan chanref eff_cap closed ty l P}}}
    DecCap #chanref
{{{ RET #(); if closed then own_chan chanref eff_cap closed ty l P
            else own_chan chanref (int.Z (word.sub eff_cap 1)) closed ty l P}}}.
```

The proofs for both lemmas is a trivial walkthrough of the `IncCap`/`DecCap` code.

6.2.2 InnerReceive

Similar to `IncCap` and `DecCap`, `InnerReceive` works directly with the `ChanStruct<T>` pointer, and will only be called in sections of locked code.

```
Definition InnerReceive: val :=
  λ: "chanref",
    match: !"chanref" with
      InjL "closed" ⇒
        match: "closed" with
          InjL "empty" ⇒ ("empty", #false, #true)
        | InjR "con" ⇒
            let: "elem" := Fst "con" in
            let: "rest" := Snd "con" in
              "chanref" ← InjL("rest");;
              ("elem", #true, #true)
        end
    | InjR "open" ⇒
        let: "eff_cap" := Fst "open" in
        let: "con" := Snd "open" in
        match: "con" with
          InjL "empty" ⇒ ("empty", #false, #false)
        | InjR "con" ⇒
            let: "elem" := Fst "con" in
```

```
        let: "rest" := Snd "con" in
          "chanref" ← InjR("eff_cap", "rest");;
          ("elem", #true, #true)
      end
  end.
```

Since `InnerReceive` can only be called in locked code, the postcondition for `wp_InnerReceive` should be made as strong as possible.

```
Lemma wp_InnerReceive chanref eff_cap closed ty l (P : val → iProp Σ):
  {{{ own_chan chanref eff_cap closed ty l P}}}
    InnerReceive #chanref
  {{{RET ((peek (zero_val ty) l, #(non_empty l), #(valid_return closed l)));
    own_chan chanref eff_cap closed ty (tail l) P ∗ ⌜ val_ty (peek (zero_val ty) l) ty ⌝ ∗
    if (andb (non_empty l) (valid_return closed l)) then P (peek (zero_val ty) l)
    else ⌜ (peek (zero_val ty) l) = zero_val ty ⌝}}}.
```

Since there are a lot of components to this postcondition, let's break it down one at a time. The first part is the `own_chan`. The precondition states `own_chan chanref eff_cap closed ty l P`. `InnerReceieve` removes the first element from `l` and changes nothing else, so at the end of `InnerReceieve`, `own_chan chanref eff_cap closed (tail l) l P` should be true. Where `tail` is defined as:

```
Definition tail (l : list val): list val :=
  match l with
  | nil ⇒ l
  | cons x l' ⇒ l'
  end.
```

The rest of `wp_InnerReceieve`'s postcondition pertains to the 3 three return values of `InnerReceieve`. The first return value is `peek (zero_val ty) l`, where `peek` is defined as:

```
Definition peek (zero: val) (l : list val): val :=
```

```
  match l with
  | nil ⇒ zero
  | cons x l' ⇒ x
  end.
```

This returns the first element of the list `l` is `l` is not empty, and otherwise returns the zero value of type `ty`. The first return value should always be of type `ty`, which can be expressed by the condition `⌜val_ty (peek (zero_val ty) l) ty⌝`.

The second return value of `InnerReceieve` is true when the list `l` is not empty, which can be written as `#(non_empty l)`, where `non_empty` is defined as:

```
Definition non_empty (l : list val): bool :=
  match l with
  | nil ⇒ false
  | cons x l' ⇒ true
  end.
```

Finally, the third return value represents whether the received succeeded, which happens if the channel is closed(receives finish immediately), or if the channel is open and the list is not empty. This can be written as `#(valid_return closed l)`, where `valid_return` is defined as:

```
Definition valid_return (closed : bool) (l : list val): bool :=
  orb closed (non_empty l).
```

Now, the final part of `wp_InnerReceive`'s postcondition is

```
  if (andb (non_empty l) (valid_return closed l)) then P (peek (zero_val ty) l)
  else ⌜ (peek (zero_val ty) l) = zero_val ty ⌝}}}.
```

which says if the receive suceeds and the list is not empty, then the returned element should satisfy property `P`. Otherwise, the returned value should be the zero value of `ty`.

The proof for `wp_InnerReceieve` is long, but boils down to casing on whether the channel

is open or closed, and then inducting on the list `l` and verifying that the return values match the spec, and the first return still retains property `P` even after it leaves the channel.

6.2.3 TryReceive

The last helper function for `ChannelReceive` is `TryReceive`, which puts together the `IncCap`, `InnerReceive`, and `DecCap` steps and wraps them in sections of locked code appropriately.

```
Definition TryReceive: val :=
  λ: "chanref" "lock",
    lock.acquire "lock";;
    IncCap "chanref";;
    lock.release "lock";;
    lock.acquire "lock";;
    let: "r" := InnerReceive "chanref" in
      DecCap "chanref";;
      lock.release "lock";;
      "r".
```

Similar to `wp_TrySend`, the `TryReceive` theorem should handle most of the work of `wp_ChannelReceive`. As such, it has a nearly identical precondition and postcondition. The only difference is since `TryReceive` has 3 return values instead of 2, its if condition relies on both `valid` and `ok`.

```
Theorem wp_TryReceive (chanref : loc) lk closed ty P:
    {{{ is_channel_alloc chanref lk closed ty P}}}
        TryReceive #chanref lk
    {{{(a : val) (ok : bool) (valid : bool),
        RET ((a, #(ok), #(valid))); ⌜ val_ty a ty ⌝ ∗
        if (andb valid ok) then P a else ⌜ a = zero_val ty ⌝}}}.
```

The proof for `wp_TryReceive` is relatively simple, as it follows similar casework to the proof

for `wp_InnerReceive` and mostly consists of applying the lemmas for `IncCap`, `InnerReceive`, and `DecCap` at the appropriate times.

6.2.4 ChannelReceive

Finally, we return to `wp_ChannelReceive`. Similar to the proof of `wp_ChannelSend`, `wp_TryReceive` already handles the postcondition for `wp_ChannelReceive`, so the only part left to prove is that `ChannelReceive` can terminate. Again, we use the iLöb induction tactic, and the rest of the proof follows from there.

Chapter 7

Evaluation

To evaluate how effective our model and library of separation logic specifications is as a tool for verifying Go code with channels, we tested its usefulness in verifying some basic Go code.

7.1 Zero capacity channel operations completes

Let's start with a simple test case. The following Go program creates a zero capacity channel and performs a single concurrent send and receive.

```
func example1() {
    var c chan bool = make(chan bool, 0)
    go func() {
        c <- true
    }()
    var ret = <-c
}
```

It translates into the following GooseLang program:

```
Definition example1: val :=
  rec: "example1" <> :=
```

```
    let: "c" := ref_to (chanT boolT) (NewChan boolT #0) in
    Fork (ChannelSend (![chanT boolT] "c") #true);;
    let: "ret" := ref_to boolT (Fst (ChannelReceive (![chanT boolT] "c"))) in
    #().
```

Let's prove that the above code executes correctly. If our only goal is to prove that the program can finish, there is no postcondition, and it requires no preconditions, which translates into the following theorem:

```
Theorem can_complete :
  {{{ True }}} example1 #() {{{ RET #(); True }}}.
Proof.
```

The proof for `can_complete` takes a total of 40 lines and features using `wp_NewChan`, `wp_ChannelSend`, and `wp_ChannelReceive`. `wp_NewChan` introduces a `is_channel_alloc` invariant that is necessary to be able to apply `wp_ChannelSend` and `wp_ChannelReceive`. In order to share access to channel `c` across both paths of the fork, it is necessary to declare channel `c` to be read-only using the tactic `readonly_alloc_1`. This guarantees the variable `c` will not be overwritten during either path, so both halves of the fork can access channel `c`.

Since our only goal is to prove termination, the values being sent over the channel do not matter, which means the property `P` field of `is_channel_alloc` is not needed. This allows us to use the function `fun x => True`, which just returns true for every value, for `P`. A similar strategy can be used on any Go program using channel operations on open channels to prove that they can complete.

7.2 Proving information about channel values

Next let's look at an example that wants to keep track of the values sent across the channel. The following code makes a channel of `uint64`s of capacity 2. Then it concurrently sends the values 1 and 2 to the channel. Finally, it receives 2 values from the channel and stores them

in variables `a` and `b`.

```
import (
    "github.com/tchajed/goose/machine"
)

func example2() {
    var c chan uint64
    c = make(chan uint64, 2)
    go func() {c <- 1}()
    c <- 2
    var a uint64
    var b uint64
    var ok bool
    a, ok = <-c
    if ok {machine.Assert(a == 1 || a == 2)}
    b, ok = <-c
    if ok {machine.Assert(b == 1 || b == 2)}
}
```

To ensure that the code behaves as expected, we add 2 assert statements. The first asserts that the variable `a` has either the value 1 or 2 after the receive finishes, and the second asserts the same for `b`. The `machine.Assert` line will add an assertion to be proved in the translated Coq code, which can be seen below.

```
Definition example2: val :=
  rec: "example2" <> :=
    let: "c" := ref (zero_val (chanT uint64T)) in
```

```
  "c" ← [chanT uint64T] NewChan uint64T #2;;
  Fork (ChannelSend (![chanT uint64T] "c") #1);;
  ChannelSend (![chanT uint64T] "c") #2;;
  let: "a" := ref (zero_val uint64T) in
  let: "b" := ref (zero_val uint64T) in
  let: "ok" := ref (zero_val boolT) in
  let: ("0_ret", "1_ret") := ChannelReceive (![chanT uint64T] "c") in
  "a" ← [uint64T] "0_ret";;
  "ok" ← [boolT] "1_ret";;
  (if: ![ boolT] "ok"
  then control.impl.Assert ((![uint64T] "a" = #1) || (![uint64T] "a" = #2))
  else #());;
  let: ("0_ret", "1_ret") := ChannelReceive (![chanT uint64T] "c") in
  "b" ← [uint64T] "0_ret";;
  "ok" ← [boolT] "1_ret";;
  (if: ![ boolT] "ok"
  then
    control.impl.Assert ((![uint64T] "b" = #1) || (![uint64T] "b" = #2));;
    #()
  else #()).
```

The assertions added by `machine.Assert` will force the proof to ensure that `a` and `b` become the expected values, so the only step left to prove is completion. This results in a similar theorem to the previous example.

```
Theorem ok :
  {{{ True }}} example2 #() {{{ RET #(); True }}}.
Proof.
```

The proof for `example2` took 150 lines, significantly more than the proof for `example1` due to doubling the number of channel operations, as well as adding logic to prove the assertions.

The overall structure is very similar to the proof for `example1`, and the proof makes heavy use of the `is_channel_alloc` invariant in `wp_ChannelSend` and `wp_ChannelReceive`.

To prove the assertions that `a` and `b` become either 1 or 2, we use the function `(fun` $x =>$ $\ulcorner x = \#1 \lor x = \#2 \urcorner$) for property `P` in our `is_channel_alloc` invariant. This requires us to prove that every element added to the channel satisfies this function, which is trivial in this example, and guarantees that every element received from the channel satisfies this property, which makes proving the assertion an easy task.

7.3 Conclusion

The full proofs for both examples can be found at:

https://github.com/jzhang025/perennial/tree/final/src/examples/simple_chan.

Overall, `wp_NewChan`, `wp_ChannelSend`, and `wp_ChannelReceive` are quite effective at proving that Go code using open channels behave as expected. While there a few limitations, which will be discussed in the next chapter, the `is_channel_alloc` invariant is very useful for verifying that all values sent and received over a channel satisfy a given property.

Chapter 8

Conclusion and Future Work

8.1 Conclusion

In this thesis, we presented a model to represent channels in GooseLang, a translator for channel operations, and a library of separation logic specifications for the base channel operations. The translator and library simplify the process of verifying Go code in the future, and adding support for channels opens up many new synchronization techniques.

In the following sections, we will discuss some of the limitations of our model and how these could be corrected in future work.

8.2 Limitations and Future Work

8.2.1 Proofs on Closed Channels

The most glaring limitation of our model is its lack of support for closed channels in our proofs. While the GooseLang functions contain code that can close and gracefully handle closed channels, the setup of the proof specifications do not handle closed channels well. Since our invariant states that channel operations will not change the closed vs open nature of the channel, a channel cannot be closed while it is holding the invariant. However, the invariant

is necessary for some of our proofs that require the channel to be open to function.

This limitation can be remedied using a more sophisticated invariant definition and the use of ghost states to keep track of whether a channel is closed or open.

8.2.2 Select Statements

While not directly a channel operation, Go's select statements have a unique interaction with channel operations. A select statement contains multiple cases, but the select statement terminates when any one case finishes executing. Even if multiple cases could proceed, only one of them does. Select statements are allowed to contain channel operations like send and receive. A channel operation in a case that does not proceed will not finish and is treated as if it never occurred.

Our model does not support channels in select statements, but the code could be modified to. The `TrySend` and `TryReceive` functions already function as one time attempts to complete a send or receive, and could easily be used to support select statements.

References

[1] *The Go Programming Language Specification*. Version 1.23. Golang, June 13, 2024. URL: https://go.dev/ref/spec (visited on 01/16/2025).

[2] G. G. Stephen F. Siegel. "Formal Analysis of Message Passing". In: *Lecture Notes in Computer Science*. Ed. by D. S. Ranjit Jhala. Vol. 6538. Berlin, Heidelberg: Springer, 2011. DOI: 10.1007/978-3-642-18275-4_2.

[3] M. Sälzer and M. Lange. *Fundamental Limits in Formal Verification of Message-Passing Neural Networks*. 2022. arXiv: 2206.05070 [cs.LG].

[4] H. Yu, Z. Chen, X. Fu, J. Wang, Z. Su, J. Sun, C. Huang, and W. Dong. "Symbolic Verification of Message Passing Interface Programs". In: *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. ICSE '20. Seoul, South Korea: Association for Computing Machinery, 2020, pp. 1248–1260. ISBN: 9781450371216. DOI: 10.1145/3377811.3380419. URL: https://doi.org/10.1145/3377811.3380419.

[5] H. A. López, E. R. B. Marques, F. Martins, N. Ng, C. Santos, V. T. Vasconcelos, and N. Yoshida. "Protocol-Based Verification of Message-Passing Parallel Programs". In: *SIGPLAN Not.* 50.10 (Oct. 2015), pp. 280–298. ISSN: 0362-1340. DOI: 10.1145/2858965.2814302. URL: https://doi.org/10.1145/2858965.2814302.

[6] S. M. Gibson. "Waddle: A proven interpreter and test framework for a subset of the Go semantics". Master's thesis. Cambridge, MA: Massachusetts Institute of Technology, 2020.

[7] T. C. D. Team. *The Coq proof assistant*. Version 8.20.0. 2025. URL: https://coq.inria.fr/doc/V8.20.0/refman/index.html.

[8] R. JUNG, R. KREBBERS, J.-H. JOURDAN, A. BIZJAK, L. BIRKEDAL, and D. DREYER. "Iris from the ground up: A modular foundation for higher-order concurrent separation logic". In: *Journal of Functional Programming* 28 (2018), e20. DOI: 10.1017/S0956796818000151.

[9] T. Chajed, J. Tassarotti, M. F. Kaashoek, and N. Zeldovich. "Verifying Concurrent, Crash-Safe Systems with Perennial". In: *Proceedings of the 27th ACM Symposium on Operating Systems Principles*. SOSP '19. Huntsville, Ontario, Canada: Association for Computing Machinery, 2019, pp. 243–258. ISBN: 9781450368735. DOI: 10.1145/3341301.3359632. URL: https://doi.org/10.1145/3341301.3359632.

[10] T. Chajed. "Verifying a concurrent, crash-safe file system with sequential reasoning". Ph.D. dissertation. Cambridge, MA: Massachusetts Institute of Technology, 2022.