Waddle: A proven interpreter and test framework for a subset of the Go semantics

by

Sydney Marie Gibson

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

May 2020

Author .....
Department of Electrical Engineering and Computer Science
May 12, 2020

Certified by .....
Tej Chajed
Doctoral Candidate
Thesis Supervisor

Certified by .....
M. Frans Kaashoek
Charles Piper Professor
Thesis Supervisor

Accepted by .....
Katrina LaCurts
Chair, Master of Engineering Thesis Committee

Waddle: A proven interpreter and test framework for a subset of the Go semantics

by

Sydney Marie Gibson

Submitted to the Department of Electrical Engineering and Computer Science
on May 12, 2020, 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 importing programs written in a subset of Go into Coq. Proving properties in Coq about a translated Go program implies that these properties hold for the original Go program, assuming that Goose is correct. Testing the correctness of Goose is challenging, however, because the Coq translation, called GooseLang, is not an executable semantics.

This thesis presents Waddle, a testing framework for Goose. Waddle consists of an interpreter for GooseLang, which makes GooseLang programs executable, a proof that the interpreter implements GooseLang correctly, and a framework for running GooseLang programs through the interpreter and checking that they return the same result as their corresponding Goose program. This thesis evaluates Waddle as a test-driven development framework and as a bug finding tool, and describes several bugs that Waddle has found.

Thesis Supervisor: Tej Chajed
Title: Doctoral Candidate

Thesis Supervisor: M. Frans Kaashoek
Title: Charles Piper Professor

Acknowledgments

This thesis would not have been possible without the assistance of many people along the way. I would like to thank Tej Chajed and Frans Kaashoek for guiding me through my research and helping me write my thesis, Nickolai Zeldovich and Joe Tassarotti for their technical advice and support, and Adam Chlipala for his technical design input. Also thank you to Jesse Selover for his mathmatical support, and for teaching me how to program in the first place. Thank you to everyone at MIT who has helped me along the way to here.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

One method for verifying a piece of software is to translate its implementation (e.g., a Go program) into an embedding of the execution language within a verification framework. This method provides the developers with a high degree of performance control since the software is implemented in its execution language, but requires a testing strategy that gives high confidence that the translation process preserves the semantics of the language in which the software is implemented. This thesis presents Waddle, a solution for testing a translation from Goose, a large subset of the Go programming language, into a nonexecutable model of this Go subset within the Coq Automated Theorem Prover [29].

The Waddle framework is based around a proven interpreter for the Coq embedding of Goose. Writing a proven interpreter in order to test a nonexecutable semantics is not new. A recent mechanization of the WebAssembly semantics relies on a proven interpreter, which is similar in structure to the one found in Waddle [30]. Waddle, however, applies this method to a high-level imperative language, Goose, and is the first system to use the interpreter as an iterative development tool for writing the

semantics of a language.

The following sections in this chapter provide a description of the standard workflow for verifying programs in Coq, a description of the workflow for verifying programs using Goose, high-level description of Waddle and the contributions of this thesis, and an outline for the remainder of this thesis.

1.1 Verification using the Standard Coq Workflow

Verified programs written in Coq typically rely on extraction to Haskell or OCaml, paired with a trusted interpreter to produce executable code [22]. A developer first writes a program in Gallina, the language used by the Coq framework,[1] then verifies that the program meets its defined specification. Next, they use Coq's native extraction tool to extract the program to Haskell, at which point they may compile and execute the verified program like any other Haskell program. We illustrate this process in Figure 1-1.

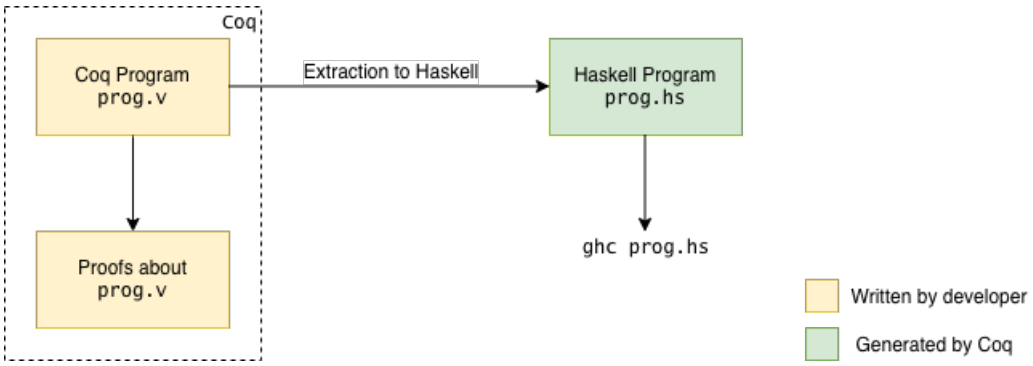

Figure 1-1: A developer first writes a Gallina program in Coq, `prog.v`, then uses the Coq framework to verify the code. Next, they extract the program to Haskell, at which point they may compile and execute the verified program.

[1]From this point forward, we refer to these programs as 'Coq programs,' for simplicity and by convention.

One problem with this approach is that developing formally verified programs directly in Coq can be tedious, especially without tailoring the code to be readily compatible with pre-existing low-level proof libraries [5, 7, 10, 14, 18, 20]. Another issue is that Coq extractions often result in performance overhead [8, 9, 19, 24]. Debugging Haskell code for performance is difficult in general, but even more so when the running code is auto-generated [16, 17, 23, 27, 28].

1.2 Verification using Goose

An alternative verification approach is to write programs in Goose. Goose is a subset of the Go programming language that can be translated into GooseLang, a semantic model of the Goose semantics embedded within Coq. Figure 1-2 visualizes the relationship between Go, Goose, and GooseLang.

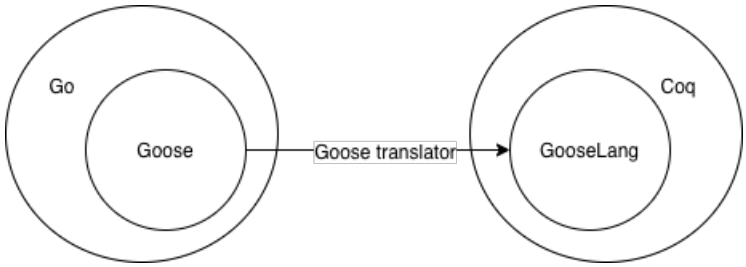

Figure 1-2: Goose is a subset of the Go programming language that can be translated into GooseLang. GooseLang is an embedding of Goose within Coq.

To verify a Goose program, a developer uses the Goose translator to translate the program into a GooseLang program, then uses Coq to verify the GooseLang program. After proving the desired theorems about the GooseLang program, the developer can then run the original Goose program using the Go runtime. Figure 1-3 shows the workflow of a developer using Goose.

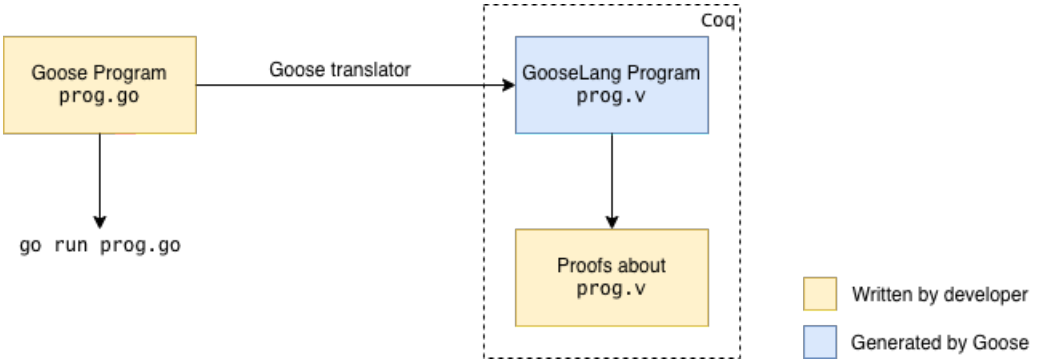

Figure 1-3: A developer first writes a Go program, `prog.go`, then uses Goose to convert `prog.go` to `prog.v`, a Coq program. The developer then uses Coq to prove that `prog.v` is correct. Once they have finished the proof, they can execute `prog.go`, knowing that it has been verified.

The benefit to using Goose is that developers can use the standard Go tools to compile and profile their code. The Go runtime is well-tuned to systems problems, and writing directly in Goose gives a developer more control over what runs, which allows for significantly easier performance analysis.

However, using Goose requires the developer to trust that proving properties about a GooseLang program truly proves the same properties about the corresponding Goose program. The Goose translator is not formally verified, and consequently, using Goose can result in bugs because the translator translates a Goose program into a GooseLang program that is not semantically equivalent to the original.

For example, consider the Goose program `bitwiseOps` shown in Figure 1-4, which returns the result of a logical OR and a logical AND operation between unsigned integers. Here are two ways one might expect this program to work:

```
1 func bitwiseOps() uint64 {
2         return 5|3&8
3 }
```

Figure 1-4: A Goose program that performs bitwise operations.

1. If AND and OR are given the same precedence, then we have $(5 \mid 3) \,\&\, 8$. Then, $5 \mid 3 = 7$, and then $7 \,\&\, 8 = 0$, so we return 0.

2. If AND is given higher precedence than OR, then we have $5 \mid (3 \,\&\, 8) = 5$.

In fact, GooseLang programs behave the first way, and Goose programs behave the second way. So, a straightforward transformation from Goose syntax to GooseLang syntax results in a bug.[2] In order to find such bugs, we need to compare the behavior of Goose programs to the behavior of their GooseLang counterparts. The approach one might expect is to run a Goose program, translate the Goose program into GooseLang, run the GooseLang program, and compare the outputs. Unfortunately, this approach does not work.

The GooseLang semantics are a nonexecutable transition system. This gives GooseLang a natural way of capturing the possible behavior of concurrent programs, but means that we cannot compare the result of running a Goose program and its translated counterpart using the transition system alone.

1.3 Waddle

This thesis presents Waddle, a test framework for Goose. Waddle interprets GooseLang programs into executable functions, enabling comparisons between return values of Goose and GooseLang programs. In order to ensure that the process of interpretation does not introduce any new bugs or change the semantic meaning of a GooseLang program, Waddle includes a proof that all interpreted programs follow the GooseLang semantics.

The contributions of this thesis are the following:

[2]This example is a real bug we found in Goose, which we discuss in further detail in Chapter 7.

1. Waddle, a test framework for Goose consisting of:

   (a) An executable interpreter for a deterministic subset of GooseLang, a Coq embedding of a subset of Go. The interpreter interprets GooseLang programs from their native relational semantics into an executable Coq function.

   (b) A proof that all interpreted programs correspond to a sequence of transitions within the GooseLang semantics, which ensures that the interpreter does not introduce any semantic differences when interpreting GooseLang programs.

   (c) A test framework, which provides infrastructure to interpret a set of GooseLang programs and check that these programs return the same result as their untranslated Goose counterparts.

2. A test suite consisting of Goose programs designed to compare the behaviors of Goose and GooseLang.

3. An evaluation of the effectiveness of Waddle as a bug finding tool and as a framework for test-driven development of Goose. This evaluation describes several bugs found by Waddle, and several features added to Goose using Waddle as a development tool. It also includes a discussion of Waddle's performance.

The majority of Waddle is implemented within Perennial, found here: `https://github.com/mit-pdos/perennial`. The part of Waddle's test framework that generates test files from the test suite of Goose programs exists within the Goose repository, found here: `https://github.com/tchajed/goose`. The Goose repository also contains the test suite.

1.4 Roadmap for this Thesis

Chapter 2 of this thesis provides more background on Goose, including a full description of the GooseLang semantics. Chapter 3 discusses the design and implementation of Waddle. Chapter 4 explains the equivalence proof between Waddle's interpreter and the GooseLang semantics. Chapter 5 evaluates Waddle in terms of bugs found, its effectiveness as a regression test suite, and its limitations. Chapter 6 discusses related work, and Chapter 7 provides a conclusion and discussion of future work.

Chapter 2

Background and Challenge

Goose is a significant subset of the Go programming language that allows users to write idiomatic Go with only a few constraints. In particular:

1. Goose supports a limited number of control flow patterns that allow early returns and loops.
2. Goose does not support Go `channels`. Programmers must write concurrent programs in Goose using locks.
3. Goose does not support Go interfaces.

2.1 GooseLang Semantics

The Goose translator translates Goose programs into GooseLang, an effectful, untyped lambda calculus embedded within Coq. This translation defines the GooseLang semantics. We give the GooseLang syntax in Figure 2-1, and the semantics in Figure 2-2.

Syntax

$$
\begin{array}{lrcl}
 & x, y, f & \in & \mathit{Var} \\
 & \ell & \in & \mathit{Loc} \\
 & n & \in & U64 \cup U32 \cup U8 \\
 & s & \in & \mathit{String} \\
 & \circledcirc & ::= & + \mid - \mid * \mid = \mid < \mid \cdots \\
 & \ominus & ::= & - \mid \mathsf{ToString} \mid \mathsf{ToU64} \mid \cdots \\
\mathit{Val} & v & ::= & () \mid \mathsf{true} \mid \mathsf{false} \mid n \mid s \mid \ell \mid (v, v) \mid \mathsf{inj}_1\ v \mid \mathsf{inj}_2\ v \mid \mathsf{rec}\, f(x) = e \\
\mathit{Exp} & e & ::= & x \mid v \mid e \circledcirc e \mid \ominus e \mid \mathsf{if}\, e\, \mathsf{then}\, e\, \mathsf{else}\, e \mid (e, e) \mid \pi_1\, e \mid \pi_2\, e \\
 & & \mid & \mathsf{inj}_1\, e \mid \mathsf{inj}_2\, e \mid \mathsf{case}\, e\, \mathsf{of}\, \mathsf{inj}_1\, x \Rightarrow e\, \mathsf{or}\, \mathsf{inj}_2\, y \Rightarrow e\, \mathsf{end} \\
 & & \mid & \mathsf{rec} f(x) = e \mid e\, e \mid \mathsf{Panic}\, s \mid \mathsf{ArbitraryInt} \mid \mathsf{CmpXChg}(e, e, e) \\
 & & \mid & \mathsf{Fork} \mid \mathsf{AllocN}\, e\, e \mid \mathsf{PrepareWrite}\, e \mid \mathsf{FinishStore}\, e\, e \\
 & & \mid & \mathsf{StartRead}\, e \mid \mathsf{FinishRead}\, e \mid \mathsf{Load}\, e \mid \langle \mathit{External} \rangle \\
\mathit{ECtx} & E & ::= & - \mid E \circledcirc e \mid v \circledcirc E \mid \ominus E \mid \mathsf{if}\, E\, \mathsf{then}\, e\, \mathsf{else}\, e \mid E\, v \mid e\, E \\
 & & \mid & (E, e) \mid (v, E) \mid \pi_1\, E \mid \pi_2\, E \mid \mathsf{inj}_1\, E \mid \mathsf{inj}_2\, E \mid \cdots \\
\mathit{NonAtom} & z & ::= & \mathsf{Reading}\, n\, v \mid \mathsf{Writing}\, v \\
\mathit{Heap} & h & \in & \mathit{Loc} \xrightarrow{\text{fin}} \mathit{NonAtom} \\
\mathit{World} & w & & \langle \mathit{External} \rangle \\
\mathit{State} & \sigma & ::= & (h, w) \\
\mathit{TPool} & \mathcal{E} & \in & \mathit{List}\ \mathit{Exp} \\
\mathit{Config} & \rho & ::= & (\mathcal{E}, \sigma)
\end{array}
$$

Figure 2-1: Syntax of GooseLang

In order to keep the GooseLang semantics simple, certain Goose features are implemented in GooseLang as libraries built from simpler primitives. This includes slices, which are built from raw pointers; maps, built from general sums; and locks, built from compare-and-exchange operations.

2.1.1 External Operations

The GooseLang semantics contain a hole, which allows for extensions of the semantics to be defined as "external operations." Currently, GooseLang includes a set of external operations that model an external disk of a fixed size. The external semantics give

Pure reduction

$$
\begin{array}{rcll}
v \circledcirc v' & \overset{\text{pure}}{\rightsquigarrow} & v'' & \text{if } v'' = v \circledcirc v' \\
\ominus\, v & \overset{\text{pure}}{\rightsquigarrow} & v' & \text{if } v' = \ominus\, v \\
\mathsf{if\ true\ then}\ e_1\ \mathsf{else}\ e_2 & \overset{\text{pure}}{\rightsquigarrow} & e_1 & \\
\mathsf{if\ false\ then}\ e_1\ \mathsf{else}\ e_2 & \overset{\text{pure}}{\rightsquigarrow} & e_2 & \\
\pi_i(v_1, v_2) & \overset{\text{pure}}{\rightsquigarrow} & v_i & \\
\mathsf{case\ inj}_i\ v\ \mathsf{of\ inj}_1\ x_1 \Rightarrow e_1\ \mathsf{or\ inj}_2\ x_2 \Rightarrow e_2\ \mathsf{end} & \overset{\text{pure}}{\rightsquigarrow} & e_i[v/x_i] & \\
(\mathsf{rec}\ f(x) = e)v & \overset{\text{pure}}{\rightsquigarrow} & e\big[(\mathsf{rec}\ f(x) = e)/f, v/x\big] & \\
\mathsf{ArbitraryInt} & \overset{\text{pure}}{\rightsquigarrow} & n & \forall n \in U64
\end{array}
$$

Per-thread one-step reduction

$$
\begin{array}{rcll}
(\sigma, e) & \rightsquigarrow & (\sigma, e') & \text{if } e \overset{\text{pure}}{\rightsquigarrow} e' \\
((h, w), \mathsf{CmpXChg}(\ell, v_1, v_2)) & \rightsquigarrow & ((h\big[\ell \mapsto v\big], w), \mathsf{true}) & \text{if } h(\ell) = v_1 \\
((h, w), \mathsf{CmpXChg}(\ell, v_1, v_2)) & \rightsquigarrow & ((h, w), \mathsf{false}) & \text{if } h(\ell) \neq v_1 \\
((h, w), \mathsf{AllocN}\ n\ v) & \rightsquigarrow & ((h[\ell + i \mapsto v\ \forall\, 0 \leq i < n], w), \ell) & \text{if } \ell + i \notin \mathrm{dom}(h)\ \forall\, 0 \leq i < n \\
((h, w), \mathsf{PrepareWrite}\ \ell) & \rightsquigarrow & ((h[\ell \mapsto \mathsf{Writing}\ v], w), ()) & \text{if } h[\ell] = \mathsf{Reading}\ 0\ v \\
((h, w), \mathsf{FinishStore}\ \ell\ v) & \rightsquigarrow & ((h[\ell \mapsto \mathsf{Reading}\ 0\ v], w), ()) & \text{if } h[\ell] = \mathsf{Writing}\ v' \\
((h, w), \mathsf{StartRead}\ \ell) & \rightsquigarrow & ((h[\ell \mapsto \mathsf{Reading}\ (n+1)\ v], w), v) & \text{if } h[\ell] = \mathsf{Reading}\ n\ v \\
((h, w), \mathsf{FinishRead}\ \ell) & \rightsquigarrow & ((h[\ell \mapsto \mathsf{Reading}\ n\ v], w), ()) & \text{if } h[\ell] = \mathsf{Reading}\ (n+1)\ v \\
((h, w), \mathsf{Load}\ \ell) & \rightsquigarrow & ((h, w), v) & \text{if } h[\ell] = \mathsf{Reading}\ n\ v
\end{array}
$$

Context reduction

$$\frac{((h, w), e) \rightsquigarrow ((h', w), e)}{((h, w), \mathcal{E}[i \mapsto E[e]]) \to ((h', w), \mathcal{E}[i \mapsto E[e']])}$$

$$\frac{j \notin \mathrm{dom}(\mathcal{E}) \cup \{i\}}{(\sigma, \mathcal{E}[i \mapsto E[\mathsf{Fork}\ \{e\}]]) \to (\sigma, \mathcal{E}[i \mapsto E[()][j \mapsto e]])}$$

Figure 2-2: Operational Semantics of GooseLang

External Disk Syntax

$$\begin{array}{llll} \mathit{Block} & b & \in & \mathit{Vec}\ 4096\ \mathit{U8} \\ \mathit{World} & w & \in & \mathit{Loc} \xrightarrow{\text{fin}} \mathit{Block} \end{array}$$

External disk per-thread one-step reduction

$$\frac{\forall\, 0 \le i < 4096,\ \ell + i \notin \mathrm{dom}(h) \qquad w[n] = b}{((h, w),\ \mathsf{Read}\, n) \rightsquigarrow ((h[\ell + i \mapsto \mathsf{Reading}\, 0\, b[i]\ \forall\, 0 \le i < 4096], w), \ell)}$$

$$\frac{\forall\, 0 \le i < 4096\ \exists\, k,\ (\mathsf{Reading}\, k\, b[i]) = h[\ell + i]}{((h, w),\ \mathsf{Write}\, n\, \ell) \rightsquigarrow ((h, w[n] \mapsto b), \ell)}$$

$$\frac{}{((h, w), \mathsf{Size}) \rightsquigarrow ((h, w), |\mathrm{dom}(w)|)}$$

Figure 2-3: Syntax and Operational Semantics for an External Disk

rules for a reading from the disk, writing to the disk, and retrieving the size of the disk. The rules for these disk operations are given in Figure 2-3.

2.2 The GooseLang Semantics by Example

To give an example of how the GooseLang semantics work, consider the function shown in Figure 2-4. This function creates a byte slice `x`, and assigns the value 1 to index 3 of the slice. It then makes a second byte slice `y`, and copies the entries from `x` to `y`. It then returns the value in index 3 of `y`, which is 1. Running this function through the Goose translator results in the GooseLang program shown in 2-5.

Now consider the expression `App testCopySimple #()` and the default starting machine state (with an empty heap), which we will call Σ.[1] These together comprise a configuration ρ. We can ask what configurations ρ can transition to under the GooseLang semantics given in Figure 2-2.

[1] A `#` symbol is used to signify a GooseLang `Val`.

```
func testCopySimple() byte {
        x := make([]byte, 10)
        x[3] = 1
        y := make([]byte, 10)
        copy(y, x)
        return y[3]
}
```

Figure 2-4: A Go program called `testCopySimple`, which creates and partially populates a slice, copies this slice to another slice, and returns one populated index from the copy.

```
Definition testCopySimple: val :=
  rec: "testCopySimple" <> :=
    let: "x" := NewSlice byteT #10 in
    SliceSet byteT "x" #3 (#(U8 1));;
    let: "y" := NewSlice byteT #10 in
    SliceCopy byteT "y" "x";;
    SliceGet byteT "y" #3.
```

Figure 2-5: The GooseLang translation of `testCopySimple`.

Since the expression in ρ is an `App` of a `Rec`, we are allowed to perform a β-reduction to replace the expression with the body of the `Rec` after performing substitution. This gives us a configuration that has the expression

```
let: "x" := NewSlice byteT #10 in
    SliceSet byteT "x" #3 (#(U8 1));;
    let: "y" := NewSlice byteT #10 in
    SliceCopy byteT "y" "x";;
    SliceGet byteT "y" #3.
```

and the same state (the starting one). Call this configuration ρ_1.

A `let` is syntactic sugar for another `App` of a `Rec`, but we cannot perform another β-reduction because the argument of the `App` is `NewSlice byteT #10`, which is not a `Val`. However, a context reduction does apply, using the context `v E`.

First, we will figure out what the configuration `(NewSlice byteT #10, Σ)` would transition to. Then, the first configuration reduction rule will tell us what transition ρ_1 will take. The definition for `NewSlice` is shown in Figure 2-6.

```
Definition make_cap: val :=
  λ: "sz",
  let: "extra" := ArbitraryInt in
  if: "sz" + "extra" > "sz"
  then "sz" + "extra" else "sz".

Definition NewSlice (t: ty): val :=
  rec: "NewSlice" "sz" :=
  if: "sz" = #0 then slice.nil
  else let: "cap" := make_cap "sz" in
       let: "p" := AllocN "cap" (zero_val t) in
       (Var "p", Var "sz", Var "cap").
```

Figure 2-6: Implementation of the `NewSlice` function.

We find that we keep applying the same reduction rules for several more steps (in order to reduce `let`s and function calls). The first new transition we encounter while reducing the `NewSlice` call is inside the `make_cap` procedure shown in Figure 2-6. In order to evaluate the expression `ArbitraryInt`, we apply the pure rule for reducing an `ArbitraryInt`.

The next new transition we apply is the per-thread one-step rule for reducing an `AllocN` instruction. This is also the first transition which has an effect on the state. We transition to a `Val` holding the location of the allocated slice, and a heap with the zero byte in every slot of the allocated slice. Finally, `NewSlice` is able to reduce to a list of three values (implemented as nested pairs), which together form the GooseLang representation of a slice.

One legal value for the arbitrary integer is 7, and a legal allocation location is location 1. So for instance, ρ_1 transitions to ρ_2, where the expression is:

```
SliceSet byteT (#(loc 1), #10, #17) #3 (#(U8 1));;
    let: "y" := NewSlice byteT #10 in
    SliceCopy byteT "y" (#(loc 1), #10, #17);;
    SliceGet byteT "y" #3.
```

and the state contains a heap with seventeen zero bytes allocated contiguously starting at location 1.

```
1 Definition SliceSet t: val :=
2   λ: "s" "i" "v",
3   (slice.ptr "s" +ℓ [t] "i") ← [t] "v".
```

Figure 2-7: Implementation of the `sliceSet` function.

Now the expression in ρ_2 is a bind, which means it is another `App` of a `Rec`. During the process of reducing `SliceSet` we encounter the new expression `(slice.ptr "s" +ℓ[t] "i") ← [t] "v"`, shown in Figure 2-7. The ← is notation for a `PrepareWrite` and then a `FinishStore` operation, which we'll perform in sequence. Note that writes to the heap (and reads from the heap) are split into two operations in the GooseLang semantics. Splitting these operations allows GooseLang to accurately model Goose writes and reads, which are nonatomic.

Substituting the variables from the other reductions, the expression that we need to reduce for the `PrepareWrite` is `(PrepareWrite #(loc: 4))`, which begins a write to the third index of our slice. We apply the one-step reduction rule for `PrepareWrite`, and because the heap in Σ at location 4 is in state `Reading: 0`, we are able to transition to the expression `LitV LitUnit` and a heap where location 4 is in the `Writing` state.

Again substituting the variables from the other reductions, the expression that we need to reduce for `FinishStore` is `(FinishStore #(loc: 4) #1)`. We apply the

one-step reduction rule for `FinishStore`, and because the heap in Σ at location 4 is in state `Writing`, we are able to transition to the expression `LitV LitUnit` and a heap where location 4 is in the `Reading: 0` state with the value 1.

When we fully finish the `SliceSet` operation, we get ρ_3 with the expression

```
let: "y" := NewSlice byteT #10 in
SliceCopy byteT "y" (#(loc: 1), #10, #17);;
SliceGet byteT "y" #3.
```

and a state with a heap containing the value 1 at location 4.

We next perform a series of reductions to allocate another slice. Suppose that we allocated this slice at location 20, and that `ArbitraryInt` returns `#9`. Then, we end this series of reductions with ρ_4, which has the expression

```
SliceCopy byteT (#(loc: 20), #10, #19) (#(loc: 1), #10, #17);;
SliceGet byteT (#(loc: 20), #10, #19) #3.
```

and a state with a heap that now also contains nineteen zero bytes allocated contiguously starting at location 20.

The `SliceCopy` implementation in Figure 2-8 shows is that we have yet another function call and `let`. Transitioning through these steps leads to the `if` statement in `SliceCopy`. We only have reduction rules for "if true" and "if false", so there is no immediate pure reduction available. Instead, we perform a context reduction using the context `if E then e else e`. The expression

```
slice.len (#(loc: 20), #10, #19) < slice.len (#(loc: 0), #10, #17)
```

reduces to `#false`, and does not affect the state. Our expression now matches the

pure reduction rule for an expression beginning with "if false", so we process the expression `slice.len (#(loc: 1), #10, #17)`, and find that it reduces to `#10`.

```
Definition MemCpy_rec t: val :=
  rec: "memcpy" "dst" "src" "n" :=
    if: "n" = #0
    then #()
    else "dst" ←[t] (![t] "src");;
         "memcpy" ("dst" +ℓ[t] #1) ("src" +1[t] #1) ("n" - #1).

Definition SliceCopy t : val :=
  λ: "dst" "src",
  let: "n" :=
     (if: slice.len "dst" < slice.len "src"
     then slice.len "dst" else slice.len "src") in
  MemCpy_rec t (slice.ptr "dst") (slice.ptr "src") "n";;
  "n".
```

Figure 2-8: The implementation of the `sliceCopy` function.

We now look at the expression `MemCpy_rec t (slice.ptr #(loc: 20), #10, #19)) (slice.ptr (#(loc: 1), #10, #17)) #10;; #10`. The transitions for the `MemCpy_rec` call are primarily ones we have already discussed, with the exception of a `Load`, which is hidden by the notation `"! e" := (Load e)`. We transition from this load using the corresponding one-step reduction rule. We end with ρ_5, which contains the expression

```
SliceGet byteT (#(loc: 20), #10, #19) #3.
```

and a heap which now contains `#1` at location 23.

Finally, we find that `SliceGet` is an `App` of a `Load` at location `#23`. This location contains the value `#1`. So, the entire configuration ρ transitions to the value `#1` (and the state from ρ_5), and the program is finished.

2.3 Challenge

The process of analyzing a possible execution of ρ is quite different than running the original Goose program, which runs natively using the Go runtime. It is unclear whether the GooseLang semantics, which are abstracted from the realities of hardware, accurately model the semantics of Goose. Further, errors in the implementation of the GooseLang library functions (e.g., the slice library) could lead to discrepancies in the behavior of the two systems. Waddle makes the GooseLang semantics executable, which allows us to empirically confirm that the GooseLang model is accurate. In a sense, Waddle automates the process of analyzing a possible execution shown in Section 2.2, allowing us to compare complete executions on many programs.

Chapter 3

Waddle

This chapter describes the design of Waddle. A diagram showing the components of Waddle is shown in 3-1. Waddle consists of two main components:

1. A verified interpreter, which implements a deterministic, single-threaded subset of the GooseLang semantics as an executable function.

2. A test framework consisting of a generator, which automatically generates Waddle tests of the GooseLang semantics from a directory of Goose test programs, as well as the necessary machinery to run Waddle tests.

The input to Waddle consists of a library of test programs written in Goose. These test programs take no arguments, and are always expected to return `true`. We discuss the structure of these test functions in greater depth in Section 3.2.

Before using Waddle, this library must be run through the Goose translator to produce the corresponding set of GooseLang functions. The Waddle test generator then produces two files. The first file consists of Go test functions that use the Go `testing` package. This allows us to verify that our Goose tests produce the

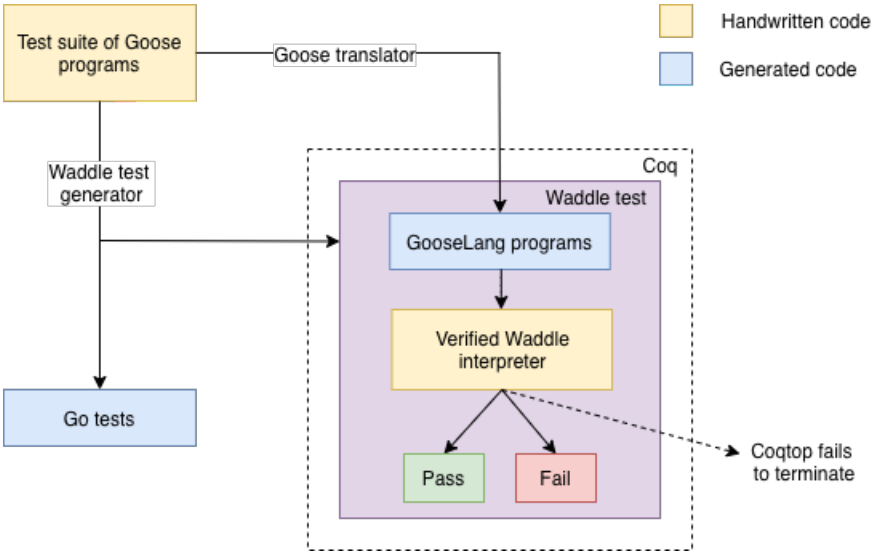

Figure 3-1: The figure above illustrates Waddle. The Waddle test generator produces a Waddle test for each test program in the test suite, as well as a Go `test` to confirm the expected output. Each Waddle test runs its corresponding GooseLang program through the interpreter, and checks the interpreter output against the expected value.

expected result when run with the standard GoLang compiler. The second file is a Coq file consisting of Waddle tests. Each Waddle test corresponds to a function from the library of test functions. Executing `coqc` on one of these tests takes the translated GooseLang function, runs it through the interpreter, and checks whether the interpretation of the GooseLang function returned `#true`. If a test fails (i.e., the interpreted programs returns `#false`), Coq fails to run and the Waddle test file will not compile, calling the failed test to attention. Another possibility is that the call to `coqc` will not terminate in a reasonable time frame. This case alerts us to a bug caused by a GooseLang program containing a subexpression which cannot be fully evaluated, such as a library function missing an argument (Section 5.1).

The following sections provide detailed descriptions of each of the Waddle components.

3.1 The Interpreter

The interpreter implements a deterministic, single-threaded, executable version of the GooseLang semantics, as described in Chapter 2. Running the interpreter on a GooseLang program results in a fully-evaluated expression (i.e., a GooseLang value) or it results in an interpreter error. In order to ensure that the interpreter does not introduce any semantic differences from the input GooseLang program, we prove that every successfully interpreted program corresponds to a a valid sequence of steps in the GooseLang semantics. The details of this proof are discussed in Chapter 4.

3.1.1 Deterministic Semantics

The majority of the GooseLang semantics is already sequential and deterministic. The only instruction in the GooseLang semantics that the interpreter does not support is `Fork`. Support for forking would require the interpreter to handle concurrent programs. Theoretically, this could be implemented by overhauling the design of the interpreter to track a thread pool, with some deterministic scheduling procedure. However, doing this would make the interpreter and its proof significantly more complicated. Implementing a single scheduler would not match the scheduler in the Go runtime, and attempting to evaluate all schedules would quickly result in performance problems for even the smallest test programs.

The interpreter implements the remainder of the instructions in the GooseLang semantics in their entirety, in some cases making deterministic choices about instructions which allow nondeterministic implementations. In particular, the interpreter always returns `#1` for `ArbitraryInt`, and always allocates at the next available location on the heap for `AllocN`. The interpreter handles external operations on a case-by-case basis. We discuss integrating external operations into the interpreter in

Section 3.1.3.

3.1.2 Implementation

The interpreter is implemented as a single loop which incrementally constructs an executable program from an input GooseLang expression. In order to guarantee termination, the interpreter also takes in an integer "fuel" value. This value decrements at each recursive call to the interpreter, and the entire evaluation fails if this fuel value reaches zero.

When evaluating an expression, we want to track the state of the system, such as data allocated on the heap or on an external disk. We also want to halt the interpreter when we discover that an expression is not semantically valid, so we must fail on invalid programs. We want to prove that if the interpreter returns a particular configuration ρ, consisting of a machine state Σ and a return value v, then GooseLang semantics could return the same result.

These goals are easy to achieve using a type that returns the system state and allows computations to fail with an error message. The interpreter returns a value of type `StateT Σ Error val`, where `Σ` is the machine state and `Error X` is a tagged union returning either a "success" tag and an `X` or a "failed" tag and an error message.[1] The type signature and high-level structure for the interpreter is shown in Figure 3-2.

The `Σ` in our return type is shared with the transition semantics for GooseLang. The implementation for `Σ` is shown in Figure 3-3. Unlike the GooseLang transition system, the interpreter does not use every component of the state. The relevant components of `Σ` for the interpreter are the `heap` and `world` fields. The `world` models

[1]The interpreter's return type implementation is essentially the same as the `StateT state (Either String)` monad in Haskell.

```
Fixpoint interpret (fuel: nat) (e: expr) : StateT Σ Error val :=
    match fuel with
    | O => mfail "Fuel depleted"
    | S n =>
      match e with
      | Val v => mret v
      | Var y ...
```

Figure 3-2: The type signature for the Waddle interpreter.

the state for external operations; in our case, this is usually an external disk with persistent memory, which we discuss in the following section.

```
Record Σ : Type := {
  heap: gmap loc (nonAtomic val);
  world: ffi_state;
  trace: Trace;
}.
```

Figure 3-3: The implementation of the GooseLang `Σ` type.

3.1.3 External Operations

Just like the GooseLang semantics, the interpreter is extensible over a set of external operations. External operations are added to the GooseLang semantics by defining an instance of the `ext_semantics` class. This class is a foreign function interface that requires a transition system defining the operations of the external semantics.

```
Class ext_interpretable :=
    {
      ext_interpret_step: external → val → StateT Σ Error expr;
      ext_interpret_ok: ...;
    }.
```

Figure 3-4: Skeleton of the `ext_interpretable` class, which gives the conditions for a set of external operations to be interpretable.

In order to support external operations in Waddle, we define a class named `ext_interpretable`, shown in Figure 3-4. The `ext_interpret_step` field contains is an interpreter for a single transition within an external semantics, and takes an external operation and a value as arguments. This single-step interpreter steps to an expression rather than a value, allowing its result to be returned to the main interpreter. This design avoids the need to duplicate the code for reducing non-external expressions into every implementation of `ext_interpret_step`. In other words, after interpreting an `ExternalOp`, we just recursively interpret the result using the GooseLang interpreter. The implementation for handling `ExternalOp`s in the interpreter is shown in Figure 3-5.

The `ext_interpret_ok` field contains a proof of semantic equivalence between the external transition system and its corresponding `ext_interpret_step` instance.

```
| ExternalOp op e =>
        v <- interpret n e;
          e' <- ext_interpret_step op v;
          interpret n e'
```

Figure 3-5: The Waddle interpreter implementation for external operations, which is dependent on an external interpreter function.

In order to implement the external disk semantics specifically, we construct an instance of `ext_interpretable`, which we call `disk_interpretable`. This instance contains `disk_interpret_step` of type `ext_interpret_step`, which interprets the per-thread one-step reductions for external disk operations shown in Figure 2-3 from Chapter 2 as an executable function. It also contains `disk_interpret_ok`, a proof that any expression interpreted by `disk_interpret_step` is equivalent to a valid transition in the external disk semantics. We discuss the details of `ext_interpret_ok` and `disk_interpret_ok` in Section 4.2.

3.1.4 Error Handling

The interpreter provides a variety of error messages when it fails to interpret a program, depending on the reason for the error. If the interpreter runs out of fuel, it simply returns a "fuel depleted" message.

Failures resulting from interpreting a program that does not match any cases in the interpreter print error messages with relevant information about the machine state at time of failure. For example, if a program attempts to perform a `Load` from a location ℓ that is not allocated on the heap, the interpreter will fail with the message "`Load` failed at location ℓ." We provide a small pretty printing library for GooseLang instructions and GooseLang values in order to print relevant debugging information.

In addition to instruction-specific error messages, the interpreter also prints a backtrace of functions called in the GooseLang program prior to its error, allowing the developer to get a better understanding of the point at which the GooseLang program failed to be interpretable. We implement backtrace support by creating an augmented state type. We replace `Σ` in the interpreter with a record type containing two fields, `Σ` and `backtrace`, where the backtrace type is a list of strings. Every time the interpreter encounters an `App` instruction with a named function, the interpreter prepends the function name to the backtrace list. Since the backtrace is stored in the state, and the interpreter does not return the state when it fails, we provide a wrapper function which serializes the backtrace to a string and returns it in the error message for the failure.

3.2 Test Framework

In order to streamline the process of testing using the interpreter, Waddle provides a test framework, which generates a formatted file of Waddle tests from a directory of Goose test programs, runs the corresponding GooseLang test programs through the interpreter, and checks the result against an expected value.

```
1 func testBitwiseOps() bool {
2         return 5|3&8 == 5
3 }
```

Figure 3-6: A test for bitwise operations, structured for compatibility with the Waddle test generator.

The first component of this framework is a test generator. The test generator is written in Go, and takes in a directory of Goose programs as input. The test programs in this directory are required to be Goose functions that have names beginning in `test`, and that take no inputs and produce Boolean outputs. All test programs are expected to return true. Other functions within the directory can be called as helper functions within the test programs, and have no name, input, or return value restrictions. Figure 3-6 shows the `bitwiseOps` program from Section 1.2 reformatted to comply with the test generator requirements.

```
1 func (suite *GoTestSuite) TestBitwiseOps() {
2         suite.Equal(true, testBitwiseOps())
3 }
```

Figure 3-7: A test for `testBitwiseOps` using the Go `testing` package, which ensures that the test program behaves as desired under the Go runtime.

In addition, tests that perform reads and writes from an external disk using the external disk semantics must initialize an empty logical disk. This disk is implemented as an array of blocks, where each block is an array of bytes.

The generator produces the two required test files for the Waddle workflow. The first test file is a Go file that uses the Go `testing` package. This file runs the test programs using the Go runtime, allowing the Waddle user to check that their programs run as expected. We show the test corresponding to `testBitwiseOps` in Figure 3-7.

```
Example testBitwiseOps_ok : testBitwiseOps #() ⤳ #true := t.
```

Figure 3-8: A Waddle test, corresponding to the `testBitwiseOps` Goose program.

The second test file is a Coq file consisting of a Coq `Example` for each test. We call these `Example`s Waddle tests. The Waddle test corresponding to `testBitwiseOps` is shown in Figure 3-8.

The second component of the Waddle framework is the Coq code required to run Waddle tests. Waddle tests assume that the programs they were generated from have also been run through the Goose translator to produce their GooseLang equivalents. The framework runs the GooseLang program called in the Waddle test through the interpreter with an initial starting `Σ` and a default fuel value. The framework then checks whether the test successfully runs to `#true`. The `coqc` process will fail on failing tests, as discussed in the following section.

3.3 Using Waddle

The step-by-step instructions for developing tests as follows:

1. Write a set of test programs that comply with the requirements of the Waddle test generator.

2. Generate the Go `test` file using the test generator, and run `go test` to ensure the tests successfully return `true` using the Go runtime.

3. Run the Goose translator on the test programs to generate a Coq file with the GooseLang translation of the test programs.

4. Generate the Coq test file, and compile the Waddle tests for the GooseLang programs using `coqc`.

5. If the test file compiles, the tests have passed. The file might fail to compile for four reasons:

   (i) A Waddle test returned `#false`, indicating that a Boolean operation in the test program evaluated oppositely of the behavior in its original GooseLang program.

   (ii) A Waddle test returned an error message indicating that the translated program did not comply with the subset of GooseLang implemented by the Waddle interpreter. This could be the result of a type error in the GooseLang program, or the result of incompleteness in the interpreter.

   (iii) A Waddle test returned an error indicating that the interpreter ran out of fuel when interpreting the program. This could indicate that more fuel should be added to interpret a specific GooseLang program, or that the program contained an infinite loop not present in the original Goose program.

   (iv) A Waddle test failed to return in a reasonable time frame, requiring manual cancellation of the compilation. This indicates that a subexpression of a GooseLang program failed to fully evaluate, for example, because it relied on a Coq `Axiom`, or on a library function missing one or more of its arguments.

The Goose developer must identify the cause of the compilation failure, and then make alternations to Goose, to a GooseLang library, or to the test framework (to adjust fuel values) to fix the error. The developer then rebuilds the adjusted files, and returns to step (iii).

If a failing test in category (i), (ii), or (iii) cannot be fixed right away, the developer can instead update the failing test program's name to have `failing_` as a prefix. Re-running the test generator on tests with this pattern will cause the corresponding Waddle tests to expect failure, allowing the test file to compile. Tests which expect failure will then throw an error if the test begins to succeed, allowing for an alternative mechanism for the developer to track when a bug is fixed. For failing tests in category (iv), the developer can change the test program to begin with `disabled_`. This will prevent the corresponding Waddle test from being generated, but will allow the test program to remain alongside the other test programs (ideally, with documentation about why the test is disabled).

We discuss the ways in which the debugging process could be improved in Section 7.1.

Chapter 4

Interpreter Verification

In order to ensure that the interpreter doesn't produce any additional bugs or doesn't mutate the meaning of GooseLang programs, we want to prove that the interpreter does exactly the same thing as the transition system that defines the semantics of GooseLang.

Ideally, we would prove the statement "machine configuration A transitions to a final machine configuration B if and only if running the interpreter with machine configuration A returns machine configuration B". However, the interpreter does not implement concurrency, and uses a deterministic subset of the GooseLang's nondeterministic semantic rules. These differences between Waddle's GooseLang interpreter and the GooseLang semantics prevent us from proving the theorem above. We can guarantee that any interpreted program corresponds to a valid series of GooseLang transition rules, but we cannot guarantee that every GooseLang program can be interpreted.

We work around the differences between the GooseLang semantics and the interpreter by proving a weaker theorem. This theorem states that if the interpreter in

a machine configuration A produces a result in a configuration B, then A must be allowed to step to B in the GooseLang semantics (although B may not be the only possible result of the computation represented by A). The code for this theorem is shown in Figure 4-1. The `runStateT` function is the "run" function for the interpreter (i.e., it executes the interpret function on a starting configuration and returns the ending configuration). This theorem ensures that all programs that are successfully interpreted by the Waddle interpreter are guaranteed to be equivalent to a valid series of transitions in the GooseLang semantics.

```
1 Theorem: interpret_ok : forall (n: nat) (e: expr) (σ:
2 Σ) (v: val) (σ': Σ),
3   (((runStateT (interpret n e) σ)
4     = Works _ (v, σ')) → exists m,
5        @nsteps goose_lang m ([e], σ) ([Val v], σ')).
```

Figure 4-1: The `interpret_ok` theorem, which states that for any natural number n, GooseLang expression e, and starting state σ that interprets to a value v and ending state σ', there is some set of GooseLang transitions that take the starting configuration (e, σ) to the ending configuration (v, σ').

However, there are certain correctness guarantees that this theorem does not provide. An interpreter that always fails would satisfy the `interpret_ok` property. The Waddle interpreter could fail to interpret a specific program because the program is wrong, but it could also fail because the interpreter runs out of fuel, or because the interpreter is incomplete. In order to get confidence that our interpreter implements all of the features we want without failing, we have a suite of example programs and assert that they interpret to the correct results. There is no formal guarantee that we have implemented every feature we want to test except for these examples.

4.1 Proof Structure

At a high level, `interpret_ok` is an induction proof over the fuel value, which proceeds by cases depending on the top-level constructor of the expression being interpreted. The base case of the induction of the induction is when we have 0 fuel, which causes the interpreter to fail and therefore the interpreter satisfies the `interpret_ok` property.

For the inductive case, we assume that the interpreter satisfies the `interpret_ok` property when it has n fuel or less. Then we proceed in cases based on the instruction at the head of the `expr` we are interpreting, assuming we have $n+1$ fuel, and proving that interpreting that expression is equivalent to performing one or more steps in the transition semantics.

For each `expr` constructor, we show that an expression containing this constructor as the head can transition to the expression to which it interprets. This requires some machinery for manipulating the interpreter code so that we can deduce needed properties about configurations from a hypothesis that the code interprets to a particular result. The main thing that this machinery does is work backwards from a hypothesis that an interpretation succeeded (or failed), and prove all of the properties about the configuration needed to create a certificate that the configuration can take some series of transition steps to reach the the ending expression.

4.1.1 Example: Proving Equivalence for Binary Operations

The proof starts with the goal

```
(((runStateT (interpret n e) σ) =
  Works _ (v, σ')) →
    ∃ m, @nsteps goose_lang m ([e], σ) ([Val v], σ'))
```

where `runStateT` is the "run" function of the interpreter (i.e., it executes the interpret function on a starting configuration and returns the ending configuration). The `nsteps` relation must be witnessed by a sequence of steps that takes the starting configuration to the ending configuration. Here, `goose_lang` refers to the type of the individual steps (which are the GooseLang transitions).

We pick a configuration `(e, σ)`, interpret that configuration by doing `runStateT (interpret n e) σ`, and we get an output configuration `(Val v, σ')`. Now we have the hypothesis that the interpretation succeeded within our proof state, and we need to show that there exists some number of steps m such that the configuration `(e, σ)` transitions to `(v, σ')`. To illustrate the details of this proof, we walk through the proof for the case where the head of the expression `e` is a binary operation. In this case, we begin with the goal

```
∃ (m : nat), nsteps m ([BinOp op e1 e2], σ) ([v], σ'),
```

which says that there exists some number of steps m such that a configuration containing an expression with binary operation instruction at its head and a state σ can transition to a configuration `(v, σ')`.

Our inductive hypothesis says that if we interpreted any configuration `(e, σ)` with n fuel and stepped to an output configuration `(v, σ')`, then `(e, σ)` and `(v, σ')` are related under the `nsteps` relation. Given this inductive hypothesis, and the fact that we know we are interpreting a `BinOp` instruction with $n + 1$ fuel, we unfold the definition of `interpret` to reveal the implementation for handling `BinOp` instructions in the interpreter. Specifically, this takes us from the hypothesis

```
runStateT (interpret (S n) (BinOp op e1 e2)) σ
        = Works (val * Σ) (v, σ')
```

to the unfolded hypothesis H:

```
H : runStateT
        (v1 ← interpret n e1;
         v2 ← interpret n e2;
         mlift (bin_op_eval op v1 v2)
         ("BinOp eval on invalid type:
           " ++ pretty op ++ "(" ++ pretty v1 ++ ", "
             ++ pretty v2 ++ ")"))
        σ = Works (val * Σ) (v, σ').
```

This unfolded hypothesis tells us that in order to evaluate a `BinOp`, we first interpret the left expression `e1`, then the right expression `e2`. Then we call the partial function `bin_op_eval`, which calculates the result of the binary operation `op` on input values `v1` and `v2`. The function `mlift` lifts `bin_op_eval` into a monadic action which makes the interpreter fail if `bin_op_eval` returns `None`.

Notice that every interpret call in H is a call to interpret with `n` fuel, meaning that our inductive hypothesis will apply to each of these recursive calls. Now, we need to take the sequence of computations in H, and for each one, apply our inductive hypothesis to find a corresponding `nsteps` relation. For example, the first line of H will lead to the hypothesis

```
runStateT (interpret n e1) σ = Works (val * Σ) (v1, s),
```

where `(v1, s)` is some intermediate configuration. Applying the inductive hypothesis will give us the corresponding relation `nsteps m ([e1], σ) ([v1], s)`. We will produce such a relation for each call to `interpret`, and then chain these relations together to show we can reach `(v, σ')`, the ending configuration from running the interpreter on `e`, by repeatedly applying transitions from the constructor that encodes the GooseLang semantics.

To begin this chaining process, we rely on an `Ltac` tactic called `run_next_interpret`,

```
Ltac run_next_interpret IHn :=
    match goal with
    | [H : runStateT (mbind (fun x => @?F x)
                            (interpret ?n ?e)) ?σ = _ |- _] =>
      let interp_e := fresh "interp_e" in
      let IHe := fresh "IHe" in
      let e_to_v := fresh "nsteps_interp" in
      let v0 := fresh "v" in
      let m := fresh "m" in
      let v := fresh "v" in
      let s := fresh "s" in
      destruct (runStateT (interpret n e) σ) as
                        [v0|] eqn:interp_e; [|runStateT_bind];
      destruct v0 as (v & s);
      pose (IHn e σ v s interp_e) as IHe;
      destruct IHe as (m & e_to_v);
      runStateT_bind
    | _ => fail
    end.
```

Figure 4-2: The `run_next_interpret` tactic handles hypotheses where the first component of a running computation is an interpret call. The tactic destructs the `runStateT` of a call to `interpret` to determine whether the call works or fails. If it works, the tactic applies the inductive hypothesis, which is passed to the tactic to avoid having to pattern-match on the context to find the correct hypothesis, to produce an `nsteps` nsteps relation.

shown in Figure 4-2. This tactic handles hypotheses like H, where the first component of a running computation is an `interpret` call. The tactic destructs the `runStateT` of the call to `interpret` to determine whether the call works or fails. If it works, the tactic applies the inductive hypothesis (passed to the tactic to avoid having to pattern-match on the context to find the correct hypothesis) to produce an `nsteps` relation.

We call `run_next_interpret` twice in the `BinOp` case, which runs the computations for the first two lines in H. Each time we call `run_next_interpret`, we get a new hypothesis that relates our previous configuration to a new configuration. Note

that this is where evaluation order can matter, because the second call to `interpret` takes the state from the ending configuration of the first `interpret` call, not the starting state σ.

As previously stated, the first call to `run_next_interpret` gives us the relation r_1:

```
nsteps m ([e1], σ) l ([v1], s),
```

and the hypothesis H becomes H':

```
H' : runStateT
       (v2 ← interpret n e2;
          mlift (bin_op_eval op v1 v2)
            ("BinOp eval on invalid type: " ++ pretty op
              ++ "(" ++ pretty v1 ++ ", " ++ pretty v2 ++ ")"))
       s = Works (val * Σ) (v, σ').
```

The second call to `run_next_interpret` gives the relation r_2:

```
nsteps m0 ([e2], s) ([v2], s0),
```

and transforms H' into H'':

```
H'' : runStateT
       (mlift (bin_op_eval op v2 v1)
          ("BinOp eval on invalid type: " ++
           pretty op ++ "(" ++ pretty v2 ++ ", " ++ pretty v1 ++ ")"))
       s0 = Works (val * Σ) (v, σ').
```

Next, we have to extract information from the rest of our hypothesis, which no longer contains any recursive calls to `interpret`. We call the tactic `runStateT_inv`, which is how we handle plain Gallina that isn't recursive. Specifically, we apply inversion to H'' to extract information about the pure Gallina functions called, such as `bin_op_eval`. This gives the hypothesis `bin_op_eval op v2 v1 = Some v`, which

tells us that the `bin_op_eval` must have returned a particular value for the intermediate step under consideration to have succeeded. It also tells us that `s0 = σ'` because none of the operations in H'' change the state.

We now introduce an existential variable using the Coq `eexists` tactic, to avoid having to specify the the exact number of steps we're going to have to do (in this case, the actual number is `m + m0 + 1`).

We next need to prove that there is a GooseLang transition corresponding to r_1 which takes the starting configuration to some intermediate configuration, that r_2 can take this intermediate configuration to yet another configuration, and so forth until we reach our goal configuration `([v], σ')`.

Applying the context `BinOpLCtx op e2` to r_1 transforms the relation into

```
nsteps m ([BinOp op e1 e2], σ) ([BinOp op v1 e2], s),
```

which starts at exactly our starting configuration. We do this context application using the tactic `ctx_step`, shown in Figure 4-3. This tactic takes a context and searches the hypotheses in our proof state for `nsteps` relations, which, when we apply the context, would start at the current configuration.

The `ctx_step` tactic is able to prove goals like

```
nsteps m ([BinOp op e1 e2], σ) ([BinOp op v1 e2], s).
```

Our goal in the proof at this point, however, is

```
nsteps ?m ([BinOp op e1 e2], σ) ([v], σ').
```

They do not have the same end configuration, but we want to say that the context

```
Ltac ctx_step ctx_expr :=
    repeat match goal with
    | [ nsteps_interp : nsteps _ ([?e], ?s) _ (_, _) |- _ ] =>
      let r := eval simpl in (ctx_expr e) in
          match goal with
          | [ |- nsteps _ ([r], s) _ _ ] =>
            let H := fresh "nsteps_interp_ctx" in
            pose proof (@nsteps_ctx _ ctx_expr
                                    _ _ _ _ _ _ _ nsteps_interp) as H;
            simpl in H;
            exact H
          | _ => fail
          end
    | _ => fail
    end.
```

Figure 4-3: Given a `ctx_expr`, the fill of some context, this tactic looks for hypotheses that an expression e steps to some value v, and a goal which states that e steps to v within `ctx_expr`. This tactic relies on the `nsteps_ctx` lemma, which says that if we know a step is valid outside of a context, then it is also a valid step within a context.

step is the first part of a chain of relations, which will eventually prove the goal. To do this, we first apply the `nsteps_transitive` tactic, which takes our goal and splits it apart into two goals. The first goal is to show that we can take our starting configuration to some intermediate configuration, and the second goal is to show that this intermediate configuration is related to the ending configuration. Then, applying `ctx_step` with the context `BinOpLCtx op e2` dispatches the first goal, and we are left with the goal

`nsteps ?m ([BinOp op v1 e2], s) ([v], σ').`

We now repeat the same process of splitting the goal using `nsteps_transitive` and applying `ctx_step` with the appropriate context, which is now `BinOpRCtx op`

v1. This time, the context application transforms r_2 into

```
nsteps ?m ([BinOp op v1 e2], s) ([BinOp op v1 v2], σ'),
```

which starts at exactly our previous intermediate configuration.

Our final goal is

```
nsteps ?m ([BinOp op v1 v2], σ') ([v], σ'),
```

which we dispatch using the tactic `single_step`. This tactic allows us to directly apply the transition rules encoded in the GooseLang semantics. In this case, we have `bin_op_eval op v1 v2 = Some v` as a hypothesis in our proof context, so `single_step` automatically applies the pure reduction rule for binary operations, completing our example.

4.2 Proving External Operations

The proof of the `ExternalOp` case of the interpreter structurally resembles the other cases, except that we must rely on an external interpreter with its own proof rather than just our inductive hypothesis. As mentioned in Section 3.1.3, we define the class `ext_interpretable`. We now show the complete implementation of this class in Figure 4-4.

The class requires a single-step interpreter, which takes an external operation (`external`) and a `Val`. It returns a GooseLang expression representing the state changes caused by the evaluation, and the expression to which the starting `ExternalOp` expression transitions. The class also requires a proof for this single-step interpreter,

```
Class ext_interpretable :=
    {
      ext_interpret_step : external -> val -> StateT Σ Error expr;
      ext_interpret_ok : ∀ (op : external) (v : val) (e' : expr) (σ σ': Σ),
          (runStateT (ext_interpret_step eop v) σ = Works _ (e', σ')) →
          ∃ m, @nsteps goose_lang m ([ExternalOp op (Val v)], σ) ([e'], σ');
    }.
```

Figure 4-4: The `ext_interpretable` class defines the requirements for defining an external semantics for integration with the Waddle interpreter. Similar to `interpret_ok`, `ext_interpret_ok` requires a proof that if the interpreter successfully interprets some configuration A to some configuration B, then there must be a step in the external semantics that reduces A to B.

`ext_interpret_ok`. This proof says that, for any external operation `op` which all take single value `v` as an argument, if running the external single-step interpreter `ext_interpret_step` on a configuration `(op v, σ)` results in a successful interpretation to a configuration `(e', σ')`, then there exists a series of transitions in the GooseLang semantics which takes the configuration `(ExternalOp op v, σ)` to the configuration `(e', σ')`. In other words, taking a single step in an interpreter of type `ext_interpret_step` is equivalent to a transition in the GooseLang semantics which starts at a configuration that contains an external operation at its head.

In order to implement the external disk semantics, we define and implement `disk_interpret_step` of type `ext_interpret_step` and `disk_interpret_ok` of type `ext_interpret_ok`, and use them to define an instance of the `ext_interpretable` class. We give the signatures for these types in Figure 4-5.

The main interpreter uses `ext_interpretable` to prove the case for external operations by letting `ext_interpret_ok` fill the role of the inductive hypothesis in order to show the existence of a valid transition from a configuration with an expression beginning with an external operation. More explicitly, we prove the transition `(ExternalOp op e, σ)` $\rightsquigarrow$ `(v, σ')` by first showing

```
Fixpoint disk_interpret_step (op: DiskOp) (v: val) : StateT Σ Error expr.

Lemma disk_interpret_ok : ∀ (op : DiskOp) (v : val) (e' : expr) (σ σ': Σ),
    (runStateT (disk_interpret_step op v) σ = Works _ (e', σ')) →
    ∃ m, @nsteps goose_lang m ([ExternalOp op (Val v)], σ) ([e'], σ').
```

Figure 4-5: The `disk_interpret_step` function is an interpreter which interprets the equivalent of a single step in the external disk semantics. The lemma `disk_interpret_ok` is equivalent to `interpret_ok`, in that it proves that any successful interpreted step from one configuration to another is also a valid transition in the external disk semantics.

$$\texttt{(ExternalOp op e, } \sigma\texttt{)} \rightsquigarrow \texttt{(ExternalOp op v', s)}$$

using the inductive hypothesis, then showing that

$$\texttt{(ExternalOp op v', s)} \rightsquigarrow \texttt{(e', s')}$$

using the hypothesis from `ext_interpret_ok`, and finally that

$$\texttt{(e', s')} \rightsquigarrow \texttt{(v, } \sigma\texttt{')}$$

using the inductive hypothesis.

Chapter 5

Evaluation

This chapter gives an evaluation of Waddle in terms of its effectiveness in finding bugs, its value as a regression test suite, its ability to scale, and its ability to adapt to future changes to Goose.

5.1 Bugs Found

In this section, we give a brief summary of some of the bugs found using Waddle and our suite of Goose tests.

Encoding and Decoding Operations

Goose uses Go's built-in support for handling conversions between integers and bytes. GooseLang implements these encoding and decoding operations using library functions. The encoding operation converts an integer to a slice of bytes, and stores each byte of the slice on the heap in adjacent locations. To decode, the bytes are loaded from the heap, converted to integers, left-shifted by the appropriate amounts, and

then summed. Our Waddle test discovered that the implementation for decoding bytes into unsigned 64-bit integers was missing a left-shift operation.

Infinite-precision Arithmetic

We discovered a bug in the way GooseLang handles untyped arithmetic. The Go compiler handles an untyped expression like `1«32-1` at infinite precision, and then appropriately truncates the resulting value. However, the Goose translator was translating arithmetic expressions with an element larger than 32 bits as an unsigned 64-bit integer, resulting in different arithmetic precision.

GooseLang also failed to include a cast back to a 32-bit integer around the arithmetic expression. This caused a type error when an expression like `1«32-1` was used in an operation which expected an unsigned 32-bit value.

Comparison and Equality Operations

Testing the behavior of integer comparison operations revealed that the GooseLang implementation for $<$ and $\leq$ operations were identically implemented as $<$. We also found that our equality comparison, rather than comparing a value v_1 to another value v_2, was checking v_2 against itself and always returning true.

Slice Copy

The implementation of the `SliceCopy` function, shown in Figure 2-8 in Chapter 2, originally made a call to the helper function `MemCpy_rec` (also shown in Figure 2-8) without its final argument.

Overloaded Operation Notation

We found several Goose unit tests (visually-inspected tests for whether the translator was syntactically correct, which were used prior to Waddle), used ‖ (Unicode character U+2225), also known as the parallel symbol, instead of two consecutive `|` (vertical pipe) characters. These characters display identically under the Emacs `prettify-symbols-mode`, which is a common extension used by Emacs users, but mean different things. The parallel symbol is not a valid token in Goose or in GooseLang, so neither the original program nor the translation held any semantic meaning.

After fixing this bug, we discovered that bitwise and logical AND operations were translated into the same GooseLang token, `&&`, resulting in code ambiguity. The same was true for OR operations, which translated into `||`. We fixed this bug by adding in separate GooseLang tokens for bitwise and logical operations, with `&` used for bitwise AND operations and `|` used for bitwise OR operations.

Function Call Ordering

Function evaluation within `struct` fields was found to be out of order. In Go, a `struct` can be initialized with its fields listed in any order, and the initial values of those fields can be given as functions. The evaluation of functions within an initialized struct follows the order listed in the initialization. In GooseLang, the evaluation occurs in the order the fields are listed in the `struct` definition. For example, if we have a struct

```
type Pair struct {
      x uint64
      y uint64
}
```

and initialize an instance `p := Pair{y: f(0), x: g(0)}` where `f` and `g` are func-

tions which take and return a single `uint64`, Goose evaluates `f(0)` before `g(0)`. GooseLang did the opposite.

Shortcircuiting

Waddle was able to confirm suspected errors in GooseLang's shortcircuiting behavior. While Goose performs shortcircuiting evaluation of logical expressions, the original implementation of GooseLang did not.

Precedence Bugs

Waddle uncovered several bugs related to precedence ordering discrepancies between Goose and GooseLang. In these bugs, Goose expressions which relied on Go operator precedence rules would be translated directly into GooseLang expressions which looked identical, but parsed under Coq operator precedence rules which were different. These bugs are fixed by adding appropriate parenthesization rules to the translator.

1. As discussed in Section 1.2, GooseLang bitwise operators for AND and OR were given the same precedence, but Goose gives higher precedence to AND operations.

2. Precedence for division and modular arithmetic was incorrect. An expression like `x := 513 + 12%8` would be translated by Goose to

   ```
   let: "x1" := (#513 + #12) ‘rem’ #8 in ...
   ```

   instead of

   ```
   let: "x1" := #513 + (#12 ‘rem’ #8) in ...,
   ```

resulting in incorrect arithmetic evaluation.

3. The arguments of logical AND and OR operations were incorrectly parenthesized, resulting in comparison operations having the same precedence as level as logical operations. This caused statements like `4 > 3 || 2 < 3` to be translated to `#4 > #3 || #3 > #2` and evaluated left-to-right, as opposed to the correct translation, `(#4 > #3) || (#3 > #2)`.

5.2 Other Bugs

Goose was under active development at the time Waddle was created, and during this time several bugs were found during development without a Waddle test. In around half of these cases, we were able to write a Waddle test which demonstrated the bug, highlighting the importance of a comprehensive test suite for Waddle's success. We describe a few examples of these bugs below.

Loop Patterns

GooseLang was found to incorrectly handle loops that break within a conditional expression unless the loop contains a `continue` statement after the end of the conditional expression. For example,

```
for {
  if true {
    break
  }
}
```

failed to break unless a `continue` is added after the final closing bracket. A test for this behavior is now included in the test suite.

Comparisons to Nil

We found that all instances of `nil` were being incorrectly translated to `slice.nil` in GooseLang (i.e., the value representing the zero value of of a slice). In Goose, `nil` also represents the zero value of multiple other types, such as maps and pointers. This bug was fixed by adding a Waddle test for comparisons to `nil`, and using the Waddle debugging process.

Pointer Type Annotations

The Goose translator was found to be annotating all stored pointers with the type of the pointer, rather than the type of the element. For example, `*x := y` was translated to `x <-[t] y`, where `t` was the type of `x` (a pointer) rather than the type of `y` (the element). While this bug was not initially detected using a Waddle test, it was debugged by adding a minimal Waddle test that captured the incorrect behavior.

Missing Dereferences when Storing Pointers

We found that GooseLang failed to dereference pointers when storing to a field of `[var ptr *T]`.

Argument Numbering in Printing Functions

Goose contains support for debugging level-based logging, which includes a printing function that prints logged messages when a non-zero debugging flag is set. We show the function below.

The function takes a variadic number of arguments, but the GooseLang implementation expects a single slice of arguments. This resulted in undefined behavior in

```
1       func DPrintf(level uint64, format string, a ...interface{}) {
2         if level <= Debug {
3           log.Printf(format, a...)
4         }
5       }
```

Figure 5-1: Implementation of the `util.DPrintf` function.

GooseLang. This issue was fixed by opting to drop the `[]interface{}` arguments in GooseLang.

Pointer Capacity

GooseLang originally modeled pointers as a tuple containing the base pointer and length, without any capacity tracking. However, a subslice in Goose may share capacity with its original slice, allowing for the original slice to be modified by appending to a subslice that shares its capacity.

5.3 Waddle as a Regression Test Suite

Waddle provides a method for ensuring that future changes to the Goose translator or to the GooseLang internals do not break previously-working test cases. We recommend that the Waddle tests be generated and run after each change make to Goose.

In addition to catching when tests fail, the Waddle tests will not compile if a previously-failing test begins to pass. This brings to attention not only when a previously-passing test begins to fail, but also when a previously-failing test begins to pass unexpectedly. This behavior mandates that any changes in the behavior of a Waddle test be acknowledged, and therefore prevents unintentional behavioral changes in GooseLang code from going unnoticed.

5.4 Test-Driven Development with Waddle

Waddle is most useful for test-driven development of Goose. To add a new feature to Goose, the developer writes a semantic unit test demonstrating the behavior, and adds a corresponding (failing) Waddle test. The developer can then check the result of the Waddle test as they develop. Examples of features that have been added to Goose using test-driven development with Waddle include casting functionality from bytes to strings, fixing the bug for comparing pointers to `nil` described above, adding support for additional control flow patterns, and adding support for nested `struct` manipulation through pointers.

5.5 Performance

Our test suite currently contains just over 50 tests, which range in length from 8 to over 100 lines of Goose. Running this test suite through the Goose translator, the Waddle test generation script, and the Go test suite takes around 3 seconds in total. Compiling the GooseLang programs and running the Waddle tests takes an additional 3 seconds. This means that executing the entire Waddle pipeline takes under 10 seconds. This performance makes Waddle worthwhile as an iterative development tool for Goose.

Compiling the interpreter and the external disk interpreter each take just under one minute. This compilation is only required if changes are made to the Coq-implemented components of Waddle or one of its dependencies. It also is only needed to re-check proofs, not to re-run the interpreter.

Component	Source lines of code
Interpreter and proof	870
Disk interpreter and proof	269
Testing framework	44
Coq code total	1,183
Test generator (Go)	136
Waddle total	1,319

Table 5.1: Source Lines of Code in Waddle Components

5.6 Implementation and Verification Effort

The implementation and verification of Waddle took place over the course of four months, with an additional two months devoted to test writing and bug fixing. Table 5.1 gives a breakdown of the code line and proof line counts for the components of Waddle. The interpreter and its proof, including all custom types, supporting lemmas, and tactics, was implemented in around 900 lines of Coq. The disk interpreter and its proof consists of around an additional 250 lines. The framework for running Waddle tests is under 50 lines. In addition, the test generator is implemented in under 150 lines of Go.

The components of Waddle implemented in Coq exist within Perennial, an Iris-based framework for verifying systems code [8, 18]. Perennial contains the implementation of GooseLang, as well as supporting libraries for proving properties about GooseLang programs. The test generator and test suite are located within the Goose project.

5.7 Limitations

There are a few notable limitations to Waddle, which we discuss in the following sections.

5.7.1 Concurrent Programs

Waddle can only test sequential programs. Waddle compares Goose programs with their GooseLang counterparts by checking that they produce the same return value, limiting Waddle to test functions with deterministic results. Further, the Waddle interpreter only implements a deterministic subset of the GooseLang semantics.

5.7.2 Debugging Support

Although Waddle does support reporting of limited error messages and a backtrace of called functions, Waddle provides little additional support for identifying the cause of a bug. Debugging a Waddle test requires manual inspection of the GooseLang program which caused the error. Further, a Waddle test which does not terminate (in a reasonable amount of time) must be manually halted, and in this case Waddle will not display any error messages.

5.7.3 Test Scalability

Waddle suffers from extreme performance hits as the number of allocations to the heap and the size of the external disk increases. This means that Waddle does not scale to tests with large numbers of loads, stores, reads, or writes. The heap and the external disk for Waddle programs are each implemented with a Coq `gmap`.

Operations on a `gmap` are increasingly costly as the number of entries in a `gmap` grows.

As mentioned in Section 5.5, the existing Waddle test suite runs in fewer than 3 seconds in its entirety. Yet, a test which performs a single write to an external disk with 30 blocks takes around 2 minutes to run. We address possible solutions to this problem in Section 7.1.

Chapter 6

Related Work

The following sections discuss prior work involving topics or methods similar to those found in Waddle.

6.1 Verification Strategies

Constructing an interpreter for testing Goose reinforces our trust in the connection between the code that we are verifying and the code that we are executing. All verification projects need some method of converting between verification code in a proof framework like Coq and executable code written in some other language. There are many methods for doing so, each with some justification for why the conversion is trusted to be correct.

A common approach, as explained in Section 1.1, and the standard approach used in Coq, is extraction. Extraction translates code from a proof theorem's native language into a similarly functional language such as Haskell or OCaml, and erases the proofs that were mixed in with the original code. Side effects are implemented in

extracted code using an interpreter in the target language that runs the verified part, and using the target language to handle interactions with the console, networking stack, and other systems in the outside world.

Another option is to deeply embed a model of an efficient, imperative language such as C or Go into a verification framework. The embedding must accurately model the chosen syntax and semantics at a low-level. Variable bindings, pointers, function calls, concurrency behaviors, and side effects in the model must all match the behavior of the real language. There are many ways to do this: Cogent [26] allows for coding in such a language directly, Fiat Cryptography [12] produces it as part of proof-producing synthesis, and clightgen for VST [3, 4] imports it directly from C source code.

Goose takes the approach of translating a stylized subset of Go into a deep embedding of Go within Coq, and trusting that the translation process is correct. Directly modeling a low-level, imperative language (and especially writing in it) gives a great deal of control, which is good for performance. It also allows developers to rapidly iterate a system in a familiar programming language, and to limit their interaction with Coq to writing theorems and proofs.

Interpreting formal semantics of programming languages into an executable interpreter has been done before, although not for an embedding of Go. The CompCert compiler includes such an interpreter for its formal model of C, and uses this interpreter to catch when programs exhibit behaviors not covered by CompCert's model of C [21].

There has previous been some work on developing a model of (a subset of) Go in Gallina, such as CoqGo [31]. However, the goal of CoqGo is to compile the Gallina source code for the verified file system FSCQ into Go, and the subset of Go that it models is tailored to the FSCQ code.

6.2 Testing Strategies

There are multiple strategies for interpreting transition system-based models into executable code. One approach is to use Coq's tactic language to automatically reify the steps of the model into an inductive structure, where the semantics of each transition are captured in a corresponding constructor, and to then write an interpreter for this reified syntax [15]. This approach was the subject of prior research, where we attempted to use it to write an interpreter for a previous iteration of Goose implemented as a shallow embedding [6]. While this approach provides simple proofs of equivalence between model transitions and their reified counterparts, we found that limitations in Coq's pattern matching system prevents this approach from working on code which uses certain dependent typing design patterns. Unfortunately, those patterns were prevalent in Goose.

As the design of Goose was in flux at the time, and this testing approach would have constrained it, we moved to a more independent interpreter design. By building a separate interpreter, the only coupling between Goose and the interpreter is in the proof that the interpreter accurately models the system. This interpreter directly translates each transition into an executable function. While this strategy adds complexity to proving semantic equivalence between modeled programs and their interpretation, it is the most suitable approach for a system that was not necessarily built with an interpreter in mind.

Building a verified, executable interpreter to test a relational semantics is not a unique idea. A mechanization of the core WebAssembly in Isabelle relies on interpreting the WebAssembly semantics to an executable Isabelle function [30]. Similar to Waddle, the proof of their interpreter requires manually applying evaluation contexts to construct a proof that the interpreter builds a function corresponding to a set of

reduction steps. However, their interpreter does not model WebAssembly program instantiation, and instead requires extraction to OCaml so that it can interface with a trusted external instantiation procedure.

One alternative to writing an interpreter for a relational semantics is to write a semantics that is natively executable. For example, the *Netsem* project used HOL4 to develop specifications for the TCP/IP protocols and Sockets API which were executable as test oracles [2]. SibylFS produces executable test oracles for POSIX, OS X, Linux, and FreeBSD file system behavior [25]. Sail supports executable semantic models for the sequential behavior of ARMv8-A, RISC-V, MIPS, and CHERI-MIPS architectures [1].

Chapter 7

Conclusion

This thesis contributes Waddle, a testing framework for Goose. By implementing a verified semantic interpreter for a deterministic subset of GooseLang, Waddle enables direct comparisons between Goose and GooseLang programs. Waddle also implements a test generator and framework for running test programs through the interpreter. Waddle has successfully found multiple bugs within Goose, and has been successfully used as a tool for test-driven development of Goose.

7.1 Future Work

There are several improvements to Waddle that would allow it to be an even more effective testing framework. We describe some of these improvements below.

7.1.1 Proof Efficiency

The proof for the interpreter and for the disk interpreter could be rewritten for efficiency. The current interpreter was implemented before the GooseLang semantics

used its current transitions library, which is much more natively executable than its predecessor. Some of the cases for various operations within the interpreter proof can likely be collapsed into a single case that relies on the new transitions library, which would likely decrease the proof's compilation time.

Additionally, the current implementation of the external disk interpreter does not have the same level of tactic support for automating repetitive components of the proof as the main Waddle interpreter. Adding this automation would significantly reduce the length of the `disk_interpret_ok` proof, and make it more robust to future changes or additions to the external disk semantics.

7.1.2 Proof Robustness

The proof of the Waddle interpreter currently relies on tactics that are specific to implementation details of the proof. The robustness of the interpreter proof could be significantly improved through tactics that abstracted away from the structure of the proof's implementation and that could solve a wider range of goal types.

Additionally, the proof of the disk interpreter currently does not have the same level of tactic support as the main interpreter. The robustness of the proof for external operations (as well as its length and efficiency) could be improved by adding tactics similar to the ones that currently exist for the main interpreter.

7.1.3 Additional Semantic Tests

The semantic test suite for Waddle is currently limited to a subset of Goose's semantics. The test suite could be expanded to include additional tests that stress the Goose semantics, as well as test which use features that may be implemented in the near future.

7.1.4 Improvements to the Testing Infrastructure

Waddle currently requires Goose test programs to have a rigid structure: they must take no inputs, and produce a Boolean output value that is expected to be true. While most properties can be tested using a program with this structure, the testing infrastructure could be expanded to allow for more flexibility when writing tests. In particular, automating the process of executing Goose test programs parallel to running their associated Waddle tests would allow for comparisons between any type of return value.

Additionally, the test generation infrastructure could also be expanded to include property-based random testing by employing techniques from QuickCheck or QuickChick [11, 13].

7.1.5 Concurrency and Nondeterminism Support

The Waddle interpreter could theoretically be extended to support multithreaded programs. This would require implementing a deterministic scheduler (since the interpreter cannot support non-deterministic behavior) and a thread pool. Proving semantic validity of multithreaded programs would require changes to supporting interpreter lemmas that assume an empty thread pool, as well as proving the `Fork` case of the interpreter itself. While implementing this feature would allow the Waddle interpreter to interpret multithreaded programs, it is unclear how multithreaded programs could be meaningfully used to test Goose using the rest of the Waddle framework.

Support for other nondeterministic features of the GooseLang semantics (such as allocation or arbitrary integer retrieval) could be could be supported by using a pseudo-random number generator instead of Waddle's current deterministic process.

7.1.6 Debugging Support

Running Waddle tests currently relies on Coq's `vm_compute` evaluation mechanism. This mechanism does not recognize opaque names, such as names declared using `Axiom` or `Parameter`. It also does not quickly recognize when an evaluation will be incomplete, such as when an argument is incorrectly missing from a function. Running a Waddle test for a program which relies on an opaque name or which cannot fully evaluate will seemingly run forever, eventually requiring manual intervention to terminate the test.

Waddle could benefit from added support that caught these situations. This would prevent Waddle tests that failed to terminate in a reasonable time, and would enable Waddle to distinguish more easily between tests which interpreted to an incorrect value, tests which interpreted to a semantically invalid program, and tests which failed to reach either a complete interpretation or a failure case in the interpreter.

Bibliography

[1] Alasdair Armstrong, Thomas Bauereiss, Brian Campbell, Alastair Reid, Kathryn E. Gray, Robert M. Norton, Prashanth Mundkur, Mark Wassell, Jon French, Christopher Pulte, Shaked Flur, Ian Stark, Neel Krishnaswami, and Peter Sewell. ISA semantics for ARMv8-a, RISC-v, and CHERI-MIPS. *Proc. ACM Program. Lang.*, 3(POPL), January 2019.

[2] Steve Bishop, Matthew Fairbairn, Hannes Mehnert, Michael Norrish, Tom Ridge, Peter Sewell, Michael Smith, and Keith Wansbrough. Engineering with logic: Rigorous test-oracle specification and validation for TCP/IP and the Sockets API. *J. ACM*, 66(1), December 2018.

[3] Sandrine Blazy and Xavier Leroy. Mechanized semantics for the Clight subset of the C language. *Journal of Automated Reasoning*, 43(3):263–288, Oct 2009.

[4] Qinxiang Cao, Lennart Beringer, Samuel Gruetter, Josiah Dodds, and Andrew Appel. VST-Floyd: A separation logic tool to verify correctness of C programs. *Journal of Automated Reasoning*, 61, 02 2018.

[5] Tej Chajed, Frans Kaashoek, Butler Lampson, and Nickolai Zeldovich. Verifying concurrent software using movers in CSPEC. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 306–322, Carlsbad, CA, October 2018. USENIX Association.

[6] Tej Chajed, Joseph Tassarotti, Frans Kaashoek, and Nickolai Zeldovich. Verifying concurrent Go code in coq with Goose. In *The Sixth International Workshop on Coq for Programming Languages*, CoqPL 2020, 2020.

[7] Tej Chajed, Joseph Tassarotti, M. Frans Kaashoek, and Nickolai Zeldovich. Argosy: Verifying layered storage systems with recovery refinement. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and*

Implementation, PLDI 2019, pages 1054âĂŞ–1068, New York, NY, USA, 2019. Association for Computing Machinery.

[8] Tej Chajed, Joseph Tassarotti, M. Frans Kaashoek, and Nickolai Zeldovich. Verifying concurrent, crash-safe systems with Perennial. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP 2019, pages 243–258, New York, NY, USA, 2019. Association for Computing Machinery.

[9] Haogang Chen, Tej Chajed, Alex Konradi, Stephanie Wang, Atalay undefinedleri, Adam Chlipala, M. Frans Kaashoek, and Nickolai Zeldovich. Verifying a high-performance crash-safe file system using a tree specification. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP 2017, pages 270–286, New York, NY, USA, 2017. Association for Computing Machinery.

[10] Haogang Chen, Daniel Ziegler, Tej Chajed, Adam Chlipala, M. Frans Kaashoek, and Nickolai Zeldovich. Using crash Hoare logic for certifying the FSCQ file system. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)*, Denver, CO, June 2016. USENIX Association.

[11] Koen Claessen and John Hughes. QuickCheck: A lightweight tool for random testing of Haskell programs. In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming*, ICFP 2000, pages 268–279, New York, NY, USA, 2000. Association for Computing Machinery.

[12] Benjamin Delaware, Clément Pit-Claudel, Jason Gross, and Adam Chlipala. Fiat: Deductive synthesis of abstract data types in a proof assistant. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL 2015, pages 689–700, New York, NY, USA, 2015. Association for Computing Machinery.

[13] Maxime Dénès, Catalin Hritcu, Leonidas Lampropoulos, Zoe Paraskevopoulou, and Benjamin C. Pierce. QuickChick : Property-based testing for Coq. 2014.

[14] A. Erbsen, J. Philipoom, J. Gross, R. Sloan, and A. Chlipala. Simple High-Level Code for Cryptographic Arithmetic - With Proofs, Without Compromises. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 1202–1219, 2019.

[15] Jason Gross, Andres Erbsen, and Adam Chlipala. Reification by parametricity. In Jeremy Avigad and Assia Mahboubi, editors, *Interactive Theorem Proving*, pages 289–305, Cham, 2018. Springer International Publishing.

[16] S. L. Peyton Jones, K. Hammond, W. D. Partain, P. L. Wadler, and C. V. Hall. The Glasgow Haskell Compiler: a technical overview. In *Proceedings of Joint Framework for Information Technology Technical Conference, Keele*, pages 249–257. DTI/SERC, March 1993.

[17] Don Jones Jr., Simon Marlow, and Satnam Singh. Parallel performance tuning for Haskell. In *ACM SIGPLAN 2009 Haskell Symposium*. Association for Computing Machinery, Inc., September 2009.

[18] Ralf Jung, Robbert Krebbers, Jacques-Henri Jourdan, Ales Bizjak, Lars Birkedal, and Derek Dreyer. Iris from the ground up: A modular foundation for higher-order concurrent separation logic. *Journal of Functional Programming*, 28:e20, 2018.

[19] Gerwin Klein, June Andronick, Kevin Elphinstone, Toby Murray, Thomas Sewell, Rafal Kolanski, and Gernot Heiser. Comprehensive formal verification of an OS microkernel. *ACM Trans. Comput. Syst.*, 32(1), February 2014.

[20] Robbert Krebbers, Jacques-Henri Jourdan, Ralf Jung, Joseph Tassarotti, Jan-Oliver Kaiser, Amin Timany, Arthur Charguéraud, and Derek Dreyer. MoSeL: A general, extensible modal framework for interactive proofs in separation logic. *Proc. ACM Program. Lang.*, 2(ICFP), July 2018.

[21] Xavier Leroy, Sandrine Blazy, Daniel Kästner, Bernhard Schommer, Markus Pister, and Christian Ferdinand. CompCert – a formally verified optimizing compiler. In *ERTS 2016: Embedded Real Time Software and Systems*. SEE, 2016.

[22] Pierre Letouzey. A new extraction for Coq. In Herman Geuvers and Freek Wiedijk, editors, *Types for Proofs and Programs*, pages 200–219, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg.

[23] Simon Marlow, Simon Peyton Jones, and Satnam Singh. Runtime support for multicore Haskell. In *Proceedings of the 14th ACM SIGPLAN International Conference on Functional Programming*, ICFP 2009, pages 65–78, New York, NY, USA, 2009. Association for Computing Machinery.

[24] Eric Mullen, Stuart Pernsteiner, James R. Wilcox, Zachary Tatlock, and Dan Grossman. Œuf: Minimizing the Coq extraction TCB. In *Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs*,

CPP 2018, pages 172–185, New York, NY, USA, 2018. Association for Computing Machinery.

[25] Tom Ridge, David Sheets, Thomas Tuerk, Andrea Giugliano, Anil Madhavapeddy, and Peter Sewell. SibylFS: Formal specification and oracle-based testing for POSIX and real-world file systems. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP 2015, pages 38–53, New York, NY, USA, 2015. Association for Computing Machinery.

[26] Christine Rizkallah, Japheth Lim, Yutaka Nagashima, Thomas Sewell, Zilin Chen, Liam O'Connor, Toby Murray, Gabriele Keller, and Gerwin Klein. A framework for the automatic formal verification of refinement from Cogent to C. In Jasmin Christian Blanchette and Stephan Merz, editors, *Interactive Theorem Proving*, pages 323–340, Cham, 2016. Springer International Publishing.

[27] Colin Runciman and David Wakeling. Profiling parallel functional computations (without parallel machines). In John T. O'Donnell and Kevin Hammond, editors, *Functional Programming, Glasgow 1993: Proceedings of the 1993 Glasgow Workshop on Functional Programming, Ayr, Scotland, 5–7 July 1993*, pages 236–251, London, 1994. Springer London.

[28] Patrick M. Sansom and Simon L. Peyton Jones. Profiling lazy functional programs. In John Launchbury and Patrick Sansom, editors, *Functional Programming, Glasgow 1992*, pages 227–239, London, 1993. Springer London.

[29] The Coq Development Team. The Coq proof assistant, version 8.9.0, January 2019.

[30] Conrad Watt. Mechanising and verifying the WebAssembly specification. In *Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs*, CPP 2018, pages 53–65, New York, NY, USA, 2018. Association for Computing Machinery.

[31] Daniel Ziegler. Compiling Gallina to Go for the FSCQ file system. Master's thesis, Massachusetts Institute of Technology, 2017.