Compiling Gallina to Go for the FSCQ File System

by

Daniel Ziegler

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

June 2017

Author . . . .
Department of Electrical Engineering and Computer Science
May 26, 2017

Certified by . . . .
Nickolai Zeldovich
Associate Professor
Thesis Supervisor

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

Accepted by . . . .
Christopher J. Terman
Chairman, Masters of Engineering Thesis Committee

Compiling Gallina to Go for the FSCQ File System

by

Daniel Ziegler

Submitted to the Department of Electrical Engineering and Computer Science
on May 26, 2017, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

Over the last decade, systems software verification has become increasingly practical. Many verified systems have been written in the language of a proof assistant, proved correct, and then made runnable using code extraction. However, due to the rigidity of extraction and the overhead of the target languages, the resulting code's CPU performance can suffer, with limited opportunity for optimization. This thesis contributes CoqGo, a proof-producing compiler from Coq's Gallina language to Go. We created Go$'$, a stylized semantics of Go which enforce linearity, and implemented proof-producing compilation tactics from Gallina to Go$'$ plus a straightforward translation from Go$'$ to Go. Applying a prototype of CoqGo, we compiled a system call in the FSCQ file system, with minimal changes to FSCQ's source code. Taking advantage of the increased control given by CoqGo, we implemented three optimizations, bringing the system call's CPU performance to 19% faster than the extracted version.

Thesis Supervisor: Nickolai Zeldovich
Title: Associate Professor

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

Acknowledgments

I am grateful to the many people who have supported and enabled the beginning of my research career at MIT. I'd like to thank my parents, who created the opportunities to learn which put me on this path. I am deeply indebted to my supervisors, Nickolai and Frans, who enabled me to jump into world-class research as sophomore, taught four of the classes I took, guided me through several different projects, and helped me write this thesis. Thanks to Frans for his encouragement and vision, and thanks to Nickolai for his incisive technical suggestions. Thanks also to Tej for frequent discussions on our journey into the world of formal verification. I'd like to thank Adam Chlipala and his students for creating the agenda which helped inspire our work and for their constant assistance with Coq. Particular thanks to Clément, whose thesis paved the way for this work. Finally, I'd like to thank my community at pika for its support over the years, and especially Andres and Jade for teaching me the value of my own judgment.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The last decade has seen the verification of increasingly practical pieces of software, including compilers [9, 11], microkernels [6, 8], web servers [3], distributed systems [7, 22], and file systems [2, 18]. Many verified systems, including CompCert [11], Verdi [22], and FSCQ [2], are written in Gallina, the specification language for the Coq proof assistant [4]. Using Gallina, implementors can write high-level functional programs and directly reason about their correctness in Coq, with verification made easier by the use of the same logic for the programs as well as the proofs.

To run these systems, many implementors employ Coq's extraction mechanism [12], directly translating Gallina to Haskell or OCaml while erasing proofs. Extraction requires minimal effort, but it has two key disadvantages. For one, the translation is unverified [1, 13, 14]: the proofs only apply to the Gallina source code, not the executable. In this work, we focus on a second disadvantage: extraction outputs code in a high-level language and offers few knobs for optimization, resulting in poor runtime performance of the output. In FSCQ's case, even after some optimization effort, the CPU performance can lag behind that of ext4 by a factor of five. Further improvements could be made, but the direct nature of extraction precludes deeper transformations, such as compilation to a lower-level language or the use of imperative rather than functional data structures. At some point, optimizing a program which is extracted runs out of steam.

To address extraction's lack of optimization opportunities, we present CoqGo, a compiler from Gallina to Go. By compiling to an imperative language and giving control over the compilation process, CoqGo makes it possible to apply more optimizations and beat the performance of extracted code. CoqGo's approach rests on two main ideas inspired by a stage of the Fiat pipeline [16]:

1. CoqGo defines a "stylized" semantics of Go, named Go′. By enforcing linearity and defining a specialized set of language constructs, Go′ makes for a much easier compilation target than Go, even though the translation from Go′ to Go is straightforward.

2. CoqGo compiles from Gallina to Go′ using proof-producing compilation tactics implemented in Coq's tactic language, Ltac. The compilation tactics break apart the source program into individual operations which can be translated directly, sequencing the execution and allocating Go′ variables to hold intermediate values.

To explore this approach, we built a prototype of CoqGo tailored specifically to FSCQ's source code, and used it to compile one of FSCQ's system calls to Go. (A significant portion of CoqGo was built as a joint effort with Alex Konradi.) Thanks to increased control over the compilation, we were able to implement three optimizations which bring the system call to around 24 times faster than the initial Go code and 19% faster than the extracted Haskell version, while requiring only 40 lines of changes to FSCQ's source code.

This thesis begins with some background on FSCQ's source code (Chapter 2) and an overview of CoqGo's pipeline (Chapter 3). Then, we explain the details of CoqGo's proof-producing compilation to Go′ (Chapter 4) and the implementation of the whole system, including its application to FSCQ (Chapter 5). To evaluate CoqGo, we investigate the changes necessary to FSCQ, the performance attained, and the optimizations enabled by increased control over compilation (Chapter 6). Finally, after discussing related work (Chapter 7) and future work (Chapter 8), we conclude (Chapter 9).

Chapter 2

Background on FSCQ

This chapter gives an overview of the source language (Section 2.1) and source code (Section 2.2) which serve as the input to CoqGo. Then, it explains the limitations of FSCQ's current code extraction approach (Section 2.3).

2.1 FSCQ source code: Monadic shallow embedding in Gallina

FSCQ's source code is written in Gallina, Coq's specification language, in a shallow embedding style on top of a simple monad [20], `prog T`, which defines the disk operations:

```
Inductive prog : Type → Type :=
| Ret (T: Type) (v: T) : prog T
| Bind (T T': Type) (p₁ : prog T) (p₂ : T → prog T') : prog T'
| Read (a: addr) : prog block
| Write (a: addr) (b: block) : prog unit
| Sync : prog unit.
```

`Ret` and `Bind` are the standard monadic combinators; `Bind p1 p2` is generally written

⟨`x ← p`$_1$`; p`$_2$` x`⟩. The other constructors are commands with side effects: reading or writing a block from disk at a particular address or synchronizing the disk by instructing it to flush write buffers.

The right-hand side of the `Bind` combinator (`p2`) is a raw Gallina function, making this a shallow embedding. Apart from the commands in `prog`, FSCQ source code is written as arbitrary Gallina code, using Coq standard library functions and data structures (such as lists and maps).

As an example, one could write the following program, which reads the disk block at address `a` and adds the block to the front of the list `vs`.

```
Definition read_into_list (a: addr) (vs: list block)
    : prog (list block) :=
  v ← Read a;
  Ret (v :: vs).
```

2.2 Constructs employed by FSCQ source code

Generic Gallina code can contain arbitrary expressions, but the constructs employed by the FSCQ source code are much more limited in practice.

FSCQ's code makes frequent use of a fairly small set of data types:

- **Natural numbers** ($\mathbb{N}$): Many parts of the code operate on arbitrarily sized natural numbers. Compared to using fixed-size integers, this simplifies proofs because it eliminates the need to prove bounds checks. However, a naive compilation must then use arbitrary-precision unsigned integers. In practice, the numbers which FSCQ manipulates (such as addresses) do not exceed 2^{64}.
  - **Operations**: The code primarily uses the standard arithmetic operations to manipulate numbers ($+$, $-$, $\times$, $/$, mod).
- **Lists** (`list T`): The code makes frequent use of the inductive list datatype, which can be thought of as a singly-linked functional list data structure.

  - **Operations**: The code often uses the `cons` constructor to prepend a new element onto the front of a list. It also iterates over lists, both in purely functional code as well as in monadic code that performs disk operations.

- **Fixed-size words** (`word n`): In some places, the code does use fixed-size bit strings rather than natural numbers. Disk blocks are 4096 bytes long, and on-disk records are defined with fixed sizes.

  - **Operations**: The most important operation which the code does on fixed-size words is to serialize and deserialize records. This involves reading and updating ranges of bytes, as well as serializing and deserializing integers.

- **Maps** (`FMapAVL`): Many parts of the code, such as the buffer cache, manipulate maps from numbers (e.g. addresses or inode numbers) to some other data structure. Everywhere maps are used, the keys are intended to be 64-bit integers, but they are expressed as $\mathbb{N}$s.

  - **Operations**: The code looks up, inserts, and removes elements from maps, as well as enumerating all entries.

- **Nested records**: Most functions in the code pass around global parameters and in-memory state using Coq record types. Generally, each module in the code defines a record type for its own in-memory state, which contains the state of the modules that it depends on.

  - **Operations**: The code constructs records out of their constituent values and gets and sets individual fields.

2.3 Existing compilation pipeline

FSCQ uses Coq's extraction feature [12] to directly translate its Gallina source code into Haskell, as outlined in Figure 2-1.

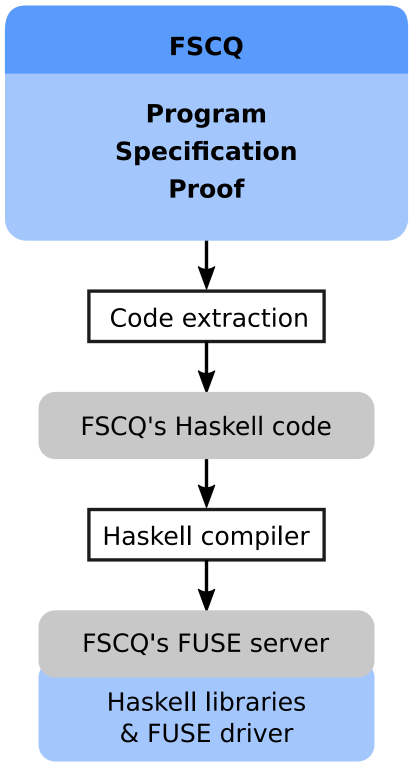

Figure 2-1: FSCQ's existing compilation pipeline

Using extraction to produce executable code enables running high-level dependently-typed functional code with minimal effort, but the directness of the translation prevents it from producing maximally performant code. Naively, extraction outputs data types that directly correspond to their structure in the Gallina source code. Lacking access to primitive machine words within Coq, programmers must define all their types as inductive algebraic data types, including integers or byte arrays; the resulting data types are extremely inefficient. To mitigate this, the extraction mechanism enables unsafe substitution of Coq data types and functions by manually-implemented Haskell code. For example, FSCQ's extraction replaces `word n` (which would natively require allocating one object per bit) with a shim around Haskell's built-in big integer type, `Integer`. The resulting performance is acceptable, but many operations are still inefficient. For one, extraction must replace types uniformly: even though `word 64` would be better to statically extract as `Word64` and `word 32768` would be better to statically extract as some variant of `ByteArray`, the extraction has to instead produce code which dynamically switches between `Word64` and `Integer`. Also, all updates to

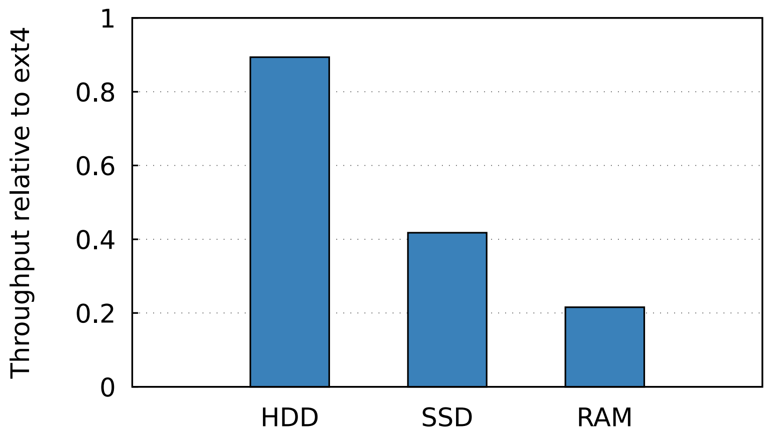

Figure 2-2: FSCQ's current performance on the small file write benchmark, as a fraction of the performance of ext4

data structures must still happen in a purely functional fashion, necessitating copying the contents of an entire disk block to mutate a single integer within it.

As a result of these problems, FSCQ's CPU performance still lags significantly behind that of ext4. Figure 2-2 shows a benchmark which repeatedly creates a new file, writes a small amount of data to it, and syncs it. On a hard drive (HDD), the high I/O latency enables FSCQ's throughput on the benchmark to be competitive with that of ext4. However, on a faster drive such as a fast SSD or a ramdisk, the slower CPU performance starts to dominate, resulting in throughput two-fifths or a fifth as high as what ext4 achieves. This benchmark shows that there exist operations where FSCQ would greatly benefit from better CPU performance.

Chapter 3

Overview

This chapter outlines the approach behind CoqGo. It explains the goals which CoqGo attempts to achieve (Section 3.1). Then, it outlines the pipeline from Gallina to Go (Section 3.2) and describes the most significant parts: the intermediate language Go′ (Section 3.3), the proof-producing compilation tactics from Gallina to Go′ (Section 3.4), and the translator from Go′ to Go (Section 3.5).

3.1 Goals

CoqGo has two key goals:

1. It should produce reasonably performant output by enabling optimizations which are not possible in extraction.

2. It should support compiling FSCQ with minimal changes to its source code.

There are some goals we do not aim for. For now, CoqGo does not need to support other applications, so it can be tailored specifically to FSCQ. We also do not aim to prove that the compilation process always works. As long as compilation succeeds, the output should include a proof that it behaves identically to the source program, but the compilation is allowed to fail.

3.2 CoqGo Pipeline

Compiling FSCQ with CoqGo involves three distinct components:

1. The **CoqGo compiler** itself, including all four stages described in Section 3.2.
2. The **FSCQ source code**.
3. **FSCQ support code for CoqGo**, including performance tweaks (such as which buffers to make immutable) and some boilerplate (a small wrapper for each record type and each compiled function). See the top-right box in Figure 3.2.

Given these ingredients, CoqGo converts FSCQ's source code into a Go executable using four stages (see Figure 3.2):

1. A proof-producing compilation pass within Coq translates it into Go′, a "stylized" variant of Go tailored to this sort of application (see Section 4.1).
2. Coq's built-in extraction mechanism converts the Go′ AST in Coq into an equivalent AST in OCaml.
3. A small OCaml program prints the Go′ AST into Go source code.
4. The Go compiler produces an executable.

We chose Go as our target language because it enables moderately good performance and offers convenient features both in the language and in the surrounding tools. Go's built-in slices and lists are a good match for the functional lists and maps which we need to extract, and Go's good profiling tools enable easy performance analysis. However, the choice of target language is fairly unimportant; it would be straightforward to adapt the current work to a different language such as C or C++.

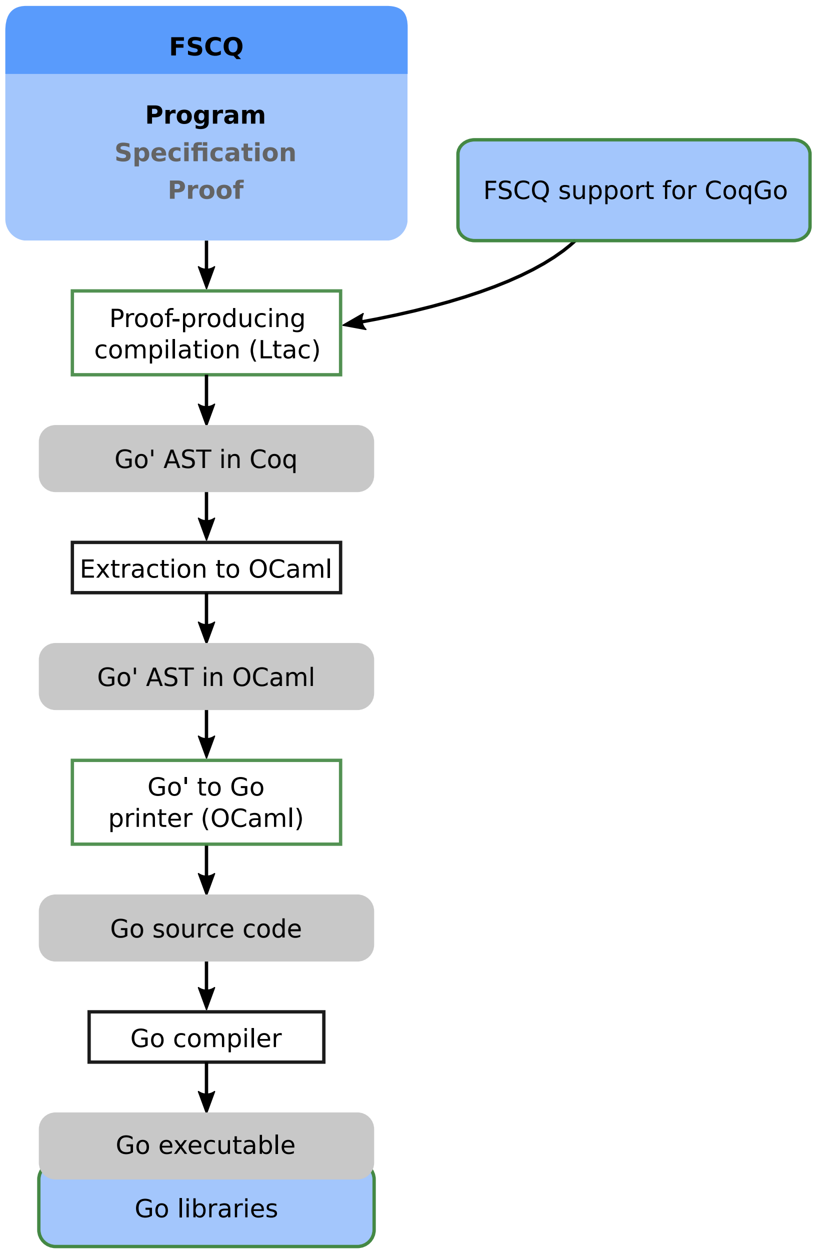

Figure 3-1: The CoqGo compilation pipeline. Blue-shaded boxes contain source code which will form part of the executable. Green borders indicate new contributions in CoqGo.

3.3 Go′

Instead of compiling directly into Go, CoqGo first compiles into an intermediate language, Go′. In a sense, Go′ is just a stylized semantics for Go: the translation from Go′ to Go is straightforward. However, Go′ adds several constraints and simplifications to Go to make compilation from functional code as easy as possible.

Go′ enforces linear use of memory. Rather than modeling a full-blown heap, Go′ ensures that mutable values can never be aliased. That way, the compiler can mutate values at will without worrying about other potential references, and we can model memory as consisting of nothing more than a lexically scoped set of variable values.

Go′ introduces additional type distinctions to keep track of invariants. For example, what will become a slice in Go could have three different types in Go′: a `Slice`, a `Buffer`, or an `ImmutableBuffer`. Each of them are treated differently: slices are kept in reverse order in Go so that a functional list cons can be turned into a slice append, mutable buffers must be copied when aliased, and immutable buffers must never be modified.

Go′ only defines a subset of Go. Since the current version of CoqGo is specific to FSCQ, Go′ only defines operations necessary for compiling FSCQ, and leaves out a number of language features, such as higher-order functions (even though they are supported by both Gallina and Go). Go′ also has no expressions, only operations, in order to uniformly handle mutation operations and immutable computation.

3.4 Proof-producing compilation tactics

The proof-producing compilation tactics convert the Gallina source code into a Go′ AST within Coq. Because the source is in a shallow embedding, it cannot be manipulated directly by Gallina functions. Instead, we implement the compilation tactics in Ltac, Coq's proof scripting language, which can manipulate Gallina terms at the syntax level. The tactics recurse through the source program, breaking it up into individual operations that can be translated directly. Monadic binds are converted into imperative sequencing (Section 4.3.1); expression evaluation is sequenced into individual steps (Section 4.4.3). The entire process is performed by repeated application of verified compilation lemmas (Section 4.3), producing a Go′ AST together with a proof that it corresponds to the source code.

Chapter 4 discusses the compilation tactics in more detail. Here, we introduce notation for stating that a particular Go′ program is a correct compilation of a Gallina program, and use it to walk through an example compilation.

We state the correctness of a particular compilation using Hoare pre- and postconditions plus a monadic source program on the side: ⟨`COMPILE p {{ P }} xp {{ Q }}`⟩. `p` denotes the source program (a Gallina term of type `prog T` for some result type `T`), `P` denotes the precondition on the Go variables, `xp` denotes the extracted Go program, and `Q` denotes the postcondition on the Go variables. The monadic source program evaluates to a single result value, but the result of a computation in Go could be stored in any variable. Thus, the postcondition `Q` is a function of the result value of `p`, and it indicates (among other things) which variable holds the Go′ value corresponding to the result. Roughly, the statement ⟨`COMPILE p {{ P }} xp {{ Q }}`⟩ means that if `P` is satisfied before `xp` is run, `xp` will have the same effects as `p`, and it will leave the Go variables in a state satisfying the postcondition `Q`, where `Q` is applied to the result of evaluating `p`.

Using the definition of correct compilation, we can frame the task of compilation as a synthesis problem: The compiler is given some source program `p` and pre- and postconditions `P` and `Q` (generally just specifying the inputs and outputs), and it must find some target program `xp` such that ⟨`COMPILE p {{ P }} xp {{ Q }}`⟩ holds. The compilation of the example program from Section 2.1 could start out as the following Coq goal, where _ indicates a hole which is yet to be filled in:

```
∀ a vs,
COMPILE v ← Read a;
        Ret (v :: vs)
{{ a_var ⇝ a * vs_var ⇝ vs }}
  _
{{ λ ret ⇒ a_var : ℕ * vs_var ⇝ ret }}
```

There are a few relevant notational details. The pre-and postconditions are expressed in terms of separation logic[1]: ⟨`a_var ⇝ a`⟩ means that the Go variable `a_var` contains a Go value which corresponds to the Gallina value `a`; ⟨`a_var : ℕ`⟩ means that `a_var` contains *some* value whose type corresponds to the Gallina type ℕ, representing natural numbers.

At this point, the compiler will see that we are extracting a monadic bind and do two things: First, it will put `var buf_var []byte` (or rather, the equivalent Go′ abstract syntax) in the top of the output code to hold the bound value. Then, it will split the remaining compilation task into two goals, one for `Read a` and one for `Ret (v :: vs)`.

First, we must read block `a` into the new variable:

```
∀ a vs,
COMPILE Read a
{{ buf_var : block * a_var ⇝ a * vs_var ⇝ vs }}
  _
{{ λ ret ⇒ buf_var ⇝ ret * _ }}
```

[1]Given that our Go semantics do not model the heap (see Section 4.1.3), we do not use separation logic in a very deep way; it was simply convenient to reuse FSCQ's existing separation logic infrastructure.

In this goal, part of the postcondition is unknown, since the automation does not know yet whether the program will need to modify `a_var` and `vs_var` to produce the right value in `buf_var` (in this case, it will not). This goal can be dispatched by doing a single disk read, filling in the Go code with `DiskRead(&buf_var, a_var)` and filling in the hole in the postcondition with ⟨ `a_var ⇝ a * vs_var ⇝ vs` ⟩ to match the precondition.

Now that the block value `v` is stored in `buf_var`, we must add it to our list:

```
∀ a vs v,
COMPILE Ret (v :: vs)
{{ buf_var ⇝ v * a_var ⇝ a * vs_var ⇝ vs }}
   _
{{ λ ret ⇒ buf_var : block * a_var : W * vs_var ⇝ ret }}
```

The compiler dispatches this goal by appending `buf_var` to `vs_var`, and our compilation is finished. The result is as follows:

```
∀ a vs,
COMPILE v ← Read a;
        Ret (v :: vs)
{{ a_var ⇝ a * vs_var ⇝ vs }}
   var buf_var []byte
   DiskRead(&buf_var, a_var)
   vs_var = append(vs_var, buf_var)
{{ λ ret ⇒ a_var : ℕ * vs_var ⇝ ret }}
```

Working inside Coq, this statement would look the same except that the Go would be in Go′ AST form rather than printed as Go.

3.5 Go′ to Go printer

CoqGo's proof-producing compilation produces a set of compiled Go′ functions in the form of a map from function names to function declarations, which include the parameter types and the body AST. Then CoqGo extracts the entire map to OCaml using Coq's built-in extraction mechanism and feeds it to the Go$'$ to Go printer written in OCaml. (Alex Konradi wrote the bulk of the Go$'$ to Go printer.)

The Go$'$ to Go printer outputs all the structs and functions into a single Go source file, which is combined with a small Go library to produce the final executable.

Chapter 4

Compilation to Go′

This chapter details the components of the proof-producing compilation from Gallina source code into Go′. The compilation of a typical Gallina construct requires implementation at five levels in the pipeline:

0. The Go′ to Go printer must convert the Go′ form of the construct into Go source code.
1. The behavior of the construct must be defined in the Go′ semantics.
2. A set of "`GoWrapper`s" must define how the relevant Gallina types map onto Go′ types.
3. A compilation lemma must state pre- and postconditions for the construct and prove them by relating the execution of the construct in Gallina to the execution of the corresponding construct in Go′.
4. An Ltac proof-producing compilation tactic must recognize when the operation should be compiled and apply the appropriate compilation lemma.

The sections below detail levels 1–4 above.

4.1 Go′ syntax and semantics

As described in the overview (Section 3.3), the primary goal of Go′ is to make proof-producing compilation from Gallina as easy as possible. It also needs to cover a sufficient subset of Go to compile FSCQ and enable the resulting code to be moderately performant. However, we did not aim to create a representation of Go in its full generality. Language features not needed by the compilation are simply omitted, and other features are represented in a specialized way in order to make them more suitable for compilation. Additionally, we did not place a high weight on minimizing the size of the semantics. Currently, the verified parts of the pipeline end at these Go′ semantics, so they are indeed part of the trusted code base, but the correct way to alleviate this would be to build a verified compiler for Go′, for instance by compiling to the Fiat pipeline's Cito language [21] or the CompCert compiler's Clight input language [11].

4.1.1 Abstract syntax

The syntax of Go′ is defined as an inductive data structure in Coq. Go′ is a simple imperative language which supports variable declarations (described in Section 4.1.4), control flow constructs `If`, `While`, and `Seq` (for sequencing statements—essentially, semicolons), function calls, disk operations to mirror those of `prog`, and a set of factored-out `Modify` operations which are used for all operations which only affect local variables. The abstract syntax also defines `Undeclare` and `InCall` constructs, which are disallowed in code and merely occur in intermediate execution states in the small-step semantics (see Section 4.1.2). Figure 4-1 shows the definition of the syntax in Coq.

Go′ supports a number of `Modify` operations which define operations on the various data types that only affect the values in variables; see Figure 4-2. Each has some fixed number of variables as arguments.

```
Inductive stmt :=
| Skip : stmt
| Seq : stmt → stmt → stmt
| If : cond → stmt → stmt → stmt
| While : cond → stmt → stmt
| Call (nparams: ℕ)
       (f: label) (* The function to call *)
       (argvars: n_tuple nparams var) (* The variables to pass in *)
| Declare : type → (var → stmt) → stmt
| Modify : ∀ op : modify_op, n_tuple (op_arity op) var → stmt
| DiskRead : var → var → stmt
| DiskWrite : var → var → stmt
| DiskSync : stmt
(* InCall and Undeclare only appear at runtime *)
| Undeclare : var → stmt
| InCall (s₀: locals) (* The stack frame inside the function *)
         (nparams: ℕ)
         (argvars: n_tuple nparams var) (* The variables jassed in *)
         (p: stmt) (* The remaining body *).
```

Figure 4-1: The definition of the abstract syntax of Go′

```
Inductive modify_op :=
| SetConst {t: type} (v: type_denote t) (* dst *)
| Duplicate (* dst, src *)
| Move (* dst, src *)
| Append (* list, element *)
| Uncons (* list, empty?, head, tail *)
| NumOp (nop: numop) (* dst, a, b *)
| SplitPair (* pair, fst, snd *)
| JoinPair (* pair, fst, snd *)
| MapAdd (* map, key, value *)
| MapRemove (* map, key *)
| MapFind (* map, key, value *)
| MapCardinality (* map, size *)
| MapElements (* map, list *)
| FreezeBuffer (* dst, src *)
| SliceBuffer (* dst, src, from, to *)
| InitSliceWithCapacity (cap: ℕ) (* dst *)
| DeserializeNum (* dst, src *).
```

Figure 4-2: The list of `Modify` operations which only affect variables.

4.1.2 Execution semantics

The Go′ execution semantics are expressed both as a big-step semantics and a small-step semantics, which are proved equivalent to each other. This enables switching between styles in proofs depending on which one is more convenient: for instance, we use the small-step semantics when reasoning about crashes which could happen at an arbitrary step in the program, but we use the big-step semantics to reason about the execution of a function call as a whole without needing to recurse through it. To represent intermediate execution states that can appear at runtime, the small-step semantics take advantage of the `InCall` construct (for when the program is inside a function call) and the `Undeclare` construct (for when the program is inside a declaration block, and needs to undeclare the variable at the end), as shown in Section 4.1.1.

An execution of a Go′ program is considered to have three possible outcomes:

- `Finished d l`: The program executes successfully, ending with the disk in state `d` and local variables with values `l`.

- `Crashed d`: The machine experienced a fail-stop crash (i.e. it lost power or kernel panicked), interrupting execution partway and leaving the disk in state `d`. Conceptually, a crash can occur at any point in a program's execution, but for simplicity, we model crashes as occurring only right before disk operations.

- `Failed`: The program attempted to perform an invalid operation (such as reading from a disk address which does not exist). We model all failures as runtime failures, even ones which would normally be compile-time failures (such as a type mismatch).

These outcomes correspond directly with the outcomes defined for the source programs in `prog`, except that `l` refers to the return value of the monadic program rather than the final state of the Go variables.

It would be possible to define a type system which guaranteed that well-typed Go$'$ programs do not go wrong (except by accessing the disk out of bounds), but this does not seem necessary in our system—if the compilation of a proven piece of source code succeeds, the correctness proof produced by the compilation already guarantees that the output program will not fail.

We now proceed to detail various aspects of the Go$'$ language.

4.1.3 Linear memory

In the most significant simplification compared to Go, Go$'$ disallows memory aliasing. As a result, they do not model pointers or the heap. Logically, all values follow a stack discipline, in which they are allocated upon variable declaration and deallocated when the variable goes out of scope; when values are used in multiple places, they are copied, at least semantically. Thanks to this simplification, the state of memory can be represented as a collection of values stored in lexically-scoped local variables, which are manipulated in an linear manner [19]. Since only one reference is allowed to any particular piece of mutable data, the semantics and the compilation process can mutate values without worrying about other references.

This simplification could plausibly have a performance impact, since it can introduce more copying than strictly necessary. However, a few optimizations mitigate the cost. For one, immutable values can be aliased without copying, since they are referentially transparent. In addition, in many cases, values can simply be moved rather than copied. For example, arguments can be passed by reference into functions, effectively transferring ownership to the callee and then transferring it back once the function returns.

To represent values which have been moved away or consumed in an operation, the semantics introduce the notion of "moving" values. For each Go$'$ type, we define how values of that type are affected when moved. Immutable values, such as integers or immutable buffers, are simply left in place; mutable types, such as slices or maps, are replaced with a special `Moved` value to mark that they cannot be used anymore. The `Moved` state does not necessarily correspond to anything at runtime: the final Go code does not need to clear the values at all since the semantics ensure that they will not be referenced.

4.1.4 Local variables

Go is a lexically scoped, statically typed language where all variables must be declared before use, so Go$'$ must allow for declarations. A straightforward encoding of variables would be fairly complicated, requiring a model of variable scope and shadowing. This would introduce unnecessary complexity to both the semantics and the compilation process. On the other hand, merely disallowing shadowing makes it impossible to write code fragments without knowing to avoid the entire set of variable names which are already in scope, which would also be an annoyance for compilation. Therefore, we use a different approach:

Rather than hardcoding variable identifiers in the code, they are allocated by the semantics. In Go$'$, a `Declare` command takes a Go$'$ type and a *function* from a variable identifier to a block of code which uses the variable. (The type signature of the `Declare` constructor is ⟨`type → (var → stmt) → stmt`⟩; see Figure 4-1.) This way, no matter which variables are already in scope, the semantics can always allocate fresh identifiers. This also ensures that variables are not used outside of their scope. For simplicity, natural numbers are used as the variable identifiers rather than strings. Later in the compilation pipeline, the Go$'$ to Go printer takes care of actually allocating unique variable names.

4.1.5 Function calls

In Go, functions can have any number of parameters and any number of return values. However, for simplicity, Go′ does not support return values and just passes all arguments by reference. The Go′ to Go printer adds an extra level of pointer indirection to parameters to implement this.

In addition to the program to execute and the runtime state, the execution semantics are parameterized by an environment containing a mapping from function names to function specifications comprising the function signature and the function body. When a `Call` instruction on some function name executes, the semantics look up the body of that function in the environment and create a new stack frame for it with the argument values filled into the parameter variables. Unlike variables created by `Declare` statements, the parameter variables are named by the hard-coded identifiers 0, 1, 2, and so forth.

4.1.6 Tuples: splitting and joining

Many pieces of FSCQ code manipulate tuples. As is typical in Coq, the code generally uses pair types, and builds up larger tuples out of nested pairs. Thus, CoqGo only needs to support pair types.

Instead of requiring the code to access the parts of a pair individually, Go′ has a `SplitPair` operation which takes a pair and moves its components into two other variables, where they can stay until the pair needs to be treated as one component again, when the `JoinPair` operation can be applied to construct a new pair.

4.1.7 Other constructs

The list cons operation, which is compiled into slice append in Go, can serve as an instructive example for the Modify operations (see Figure 4-2) which only affect the values stored in variables and do not affect control flow or the disk. We want to translate Gallina expressions like `x :: xs` into Go source code like `xs = append(xs, x)`.

The semantics define the behavior of an Append using the following definition:

```
Definition append_impl' t (l: list (type_denote t)) (a: type_denote t)
    : n_tuple 2 var_update :=
  ^(SetTo (Val (Slice t) (Here (a :: l))), Move).
```

In short, if the first argument contains list `l` and the second argument contains the value `a`, we update the variables by setting the first one to a Go′ value containing `a :: l`, and move the second one to ensure that, in case it is mutable, it is not referenced by both the list and the variable. ("Moving" immutable values simply leaves them in place.)

A wrapper around this definition completes the semantics of Append by letting it fail on type errors (the first argument must be a list of elements with the same type as the second argument).

4.2 Mapping Gallina types to Go′ types

The compilation tactics must associate Gallina types in the source code with Go′ types in the output.

The Go′ semantics define the following set of types, which are explained in more detail in the following sections:

```
Inductive type :=
| Num (* In Go: uint64 *)
| Bool (* In Go: bool *)
| String (* In Go: string *)
| Buffer (* In Go: []byte *)
| ImmutableBuffer (* In Go: []byte *)
| Slice : type → type (* In Go: []T *)
| Pair : type → type → type (* In Go: struct{fst T1; snd T2} *)
| AddrMap : type → type (* In Go: map[uint64]T *)
| Struct : list type → type (* In Go: struct *).
```

To simplify proofs about the execution of the semantics, we avoid the use of dependent types in the semantics as much as possible. Thus, even though in the source language, `word n` is a dependently typed bitstring with a fixed length, it is compiled as a `Buffer`, which can be of any (byte-aligned) length. As a result, the compiled Go′ code loses static information about the length of the bitstring, but since the compilation produces a proof of correctness anyway, this does not compromise correctness.

In the semantics, each variable holds a Go′ value, which consists of a Go′ type and a value of the corresponding Gallina type. In the semantics, the denotations for each Go′ type are defined as follows in terms of Gallina types:

```
Fixpoint type_denote (t: type) : Type :=
  match t with
  | Num ⇒ ℕ
  | Bool ⇒ 𝔹
  | String ⇒ string
  | Buffer ⇒ movable { n : ℕ & word n }
  | ImmutableBuffer ⇒ { n : ℕ & word n }
  | Slice t' ⇒ movable (list (type_denote t')) (* in reverse order *)
  | Pair t1 t2 ⇒ type_denote t1 * type_denote t2
  | AddrMap vt ⇒ movable (Map.t (type_denote vt))
  end.
```

Go′ types that need to have a distinct “moved” state in the semantics (e.g. mutable buffers) are defined using the following helper type, which adds a single `Moved` value to any type:

```
Inductive movable (T: Type) :=
| Here (v: T)
| Moved.
```

Now, we can define a single type to hold any Go′ value:

```
Inductive value :=
| Val {t: type} (v: type_denote t).
```

4.2.1 GoWrappers

When the compilation encounters a value in the source code of some Gallina type, it must know which Go′ type to pick for a variable holding that value, and it must know how to convert values of that type into Go′ values. The following typeclass encapsulates this functionality, "wrapping" a Gallina value into a Go value:

```
Class GoWrapper (WrappedType: Type) := {
  wrap_type: type;
  wrap':     WrappedType → type_denote wrap_type;
}.
```

We define a `GoWrapper` instance for each Gallina type which we would like to compile. Each `GoWrapper` instance must always produce values of the same Go type.

Once an instance has been defined for some Gallina type, values of that type can be turned into Go′ values using the `wrap` function:

```
Definition wrap {T: Type} {Wr: GoWrapper T} (t: T) : value :=
  Val wrap_type (wrap' t).
```

For immutable primitive types, such as booleans, the denotations of the Go types are identical to the source Gallina types; the `wrap'` function in those `GoWrapper` instances is just the identity. The wrappers for mutable primitive types such as lists use the `Here` constructor to return a value of type `movable`. As a result, these wrappers are not surjective: there is no list value whose wrapped version is `Moved`. This is necessary but complicates the compilation tactics somewhat, as they sometimes have to deal with preconditions involving wrapped values as well as unwrapped values.

Record types in the source code must be compiled in terms of the primitive types in the semantics, so they are wrapped as nested pairs. To reuse the automation around pairs and increase convenience for the CoqGo user, we have the following typeclass, which wraps Gallina types into Coq types by converting them into a different wrappable Gallina type first:

```
Class WrapByTransforming (T: Type) := {
  T'        : Type;
  WrT'      : GoWrapper T';
  transform : T → T';
}.
```

For example, we define such an instance for FSCQ's `cachestate` record, which contains fields `CSMap`, `CSMaxCount`, and `CSEvict`:

```
Instance WrapByTransforming_cachestate
    : WrapByTransforming cachestate :=
  {| transform := λ (cs: cachestate) ⇒
      (cs.(CSMap), cs.(CSMaxCount), cs.(CSEvict)) |}.
```

Now that the `cachestate` has been expressed as a tuple (i.e. nested pairs), it can be wrapped using the existing instances for pairs. The programmer does need to write such an instance for every type in the source code, but this is likely unavoidable since Coq lacks metaprogramming functionality.

4.2.2 Details for various types

- **Numbers** ($\mathbb{N} \mapsto$ `Num` $\mapsto$ `uint64`): Since FSCQ uses `nat` to hold values not meant to exceed 2^{64}, the Go printer compiles them as `uint64` values in Go. We insert runtime bounds checks on all arithmetic operations, aborting the program on overflow. This is sound because the FSCQ specifications only guarantee partial correctness: if a system call finishes successfully, the postcondition must hold, but there is no guarantee that it must finish successfully.

- **Lists** (`list V` $\mapsto$ `Slice v` $\mapsto$ `[]T`): Lists are compiled as slices kept in reverse order so that prepending onto the head of a singly-linked list corresponds to appending onto the end of a slice.

  The zero value in Go for any slice is `nil`, so the semantics initialize values of type `Slice` to `Here []`.

- **Maps** (`FMapAVL.t V` $\mapsto$ `AddrMap v` $\mapsto$ `map[uint64]T`): The zero value of maps in Go is `nil`, which is an invalid value; operations on a `nil` map cause the program to panic. Thus, in the semantics, map variables start out with value `Moved`, so that no operations can be performed on them until they are properly initialized.

  Rather than leaving the iteration order of maps undefined, Go actually randomizes it at runtime – each iteration through a map happens in a different order. In Coq, on the other hand, `FMapAVL` guarantees that its iteration is ordered by key. Thus, to reflect the semantics of the source, the Go output needs to copy the map contents into a list and explicitly sort it before iteration. If the source language allowed for nondeterminism, this could be avoided.

- **Fixed-size words** (`word n` $\mapsto$ `Buffer` / `ImmutableBuffer` $\mapsto$ `[]byte`): Fixed-size bitstrings are compiled as byte slices. The zero value in Go is `nil` – equivalent to an empty slice – and the semantics reflects this by initializing variables of type `ImmutableBuffer` or `Buffer` to zero-length bitstrings.

Note that in the source, the length of a bitstring does not have to be divisible by eight, even though it always is in practice. However, we do not support bitstrings of non-byte-aligned length in Go. The semantics enforce this constraint by making all operations on bitstrings with invalid length fail; the compilation lemmas pass this constraint on to the compilation tactics, which prove the divisibility obligations automatically.

FSCQ frequently manipulates buffers containing file data or data on disk, many of which are fairly large (e.g. 4096 bytes for a disk block). A straightforward compilation might either choose to always extract a buffer as immutable, likely requiring copying it on each modification, or always extracting it as mutable, requiring copying when it (or a substring) is aliased. Instead of forcing a single choice to be taken globally which will be suboptimal for some pieces of code, we enable the source code to specify which buffers should be made mutable and which buffers should be made immutable.

By default, `word` values are made mutable, but if source code instead uses the type alias `immut_word`, it will be compiled using an immutable buffer. CoqGo accomplishes this by having two separate type classes, one of type `GoWrapper (word n)` and one of type `GoWrapper (immut_word n)`, and declaring `Typeclasses Opaque immut_word` so that the typeclass resolution mechanism can distinguish between the two.

- **Records** (`Record` $\mapsto$ `Pair Num (Pair Num ...)` $\mapsto$ `Pair_Num_Pair_Num_...`): Many FSCQ functions pass around global parameters and in-memory state using Coq record types. As described in Section 4.2.1, CoqGo compiles records by using instances of the `WrapByTransforming` typeclass, which require a small amount of boilerplate from the user. A record value is first `transform`ed into a set of nested pairs and then converted into the equivalent Go$'$ nested pair value. Then, the mechanisms for pairs in the semantics and the compilation tactics take over, accessing fields by splitting apart the pairs and constructing records by joining them.

4.3 Compilation lemmas

A "compilation lemma" proves pre- and postconditions about some Go′ construct. The compilation lemmas abstract away from direct reasoning about the Go′ semantics, enabling the compilation tactics to assemble a Go′ program, along with its proof of correctness, by applying the right sequence of compilation lemmas.

4.3.1 CompileBind

One compilation lemma used frequently by the compilation tactics is `CompileBind`, which breaks apart a compilation of a monadic bind into a pair of compilation tasks, one for the bound value and one for the monadic continuation. In its generic form, it has the following signature:

```
Lemma CompileBind : ∀ {T T': Type} {H: GoWrapper T} (env: Env)
                      (A B: pred) (C: T' → pred) (p: prog T)
                      (f: T → prog T') (xp xf: stmt) (var: Go.var),
  COMPILE p
  {{ var : T * A }}
    xp
  {{ λ ret ⇒ var ⇝ ret * B }} // env →
  (∀ (a : T),
    COMPILE f a
    {{ var ⇝ a * B }}
      xf
    {{ C }} // env) →
  COMPILE Bind p f
  {{ var : T * A }}
    xp; xf
  {{ C }} // env.
```

In words, if `p` can be compiled into `xp`, and `f a` can be compiled into `xf` for every value of `a`, then `Bind p f` can be compiled into `xp; xf`. The variable `var` to hold the bound value already has to be declared.

Note that when applying this lemma, the value of the intermediate predicate `B` is not determined by the goal, and can take any form as long as both hypotheses remain provable. This enables the compilation of `p` to modify some of the values in `A` if necessary.

It is possible to prove a stronger version of this lemma which accepts a weaker second hypothesis: the lemma could pass an additional hypothesis to the compilation of `f a` which states that `a` is a possible execution result of running `p`. In its general form, this is not necessary anywhere. For the case that `p` is of the form `Ret a` for some specific value `a`, we prove a compilation lemma which places that `a` into the precondition of `f` rather than quantifying over all possible `a` values.

4.3.2 CompileDeclare

The proof-producing compilation must also declare new variables. The following compilation lemma enables it to do that:

```
Lemma CompileDeclare : ∀ (env: Env) {R T: Type} {Wr: GoWrapper T}
                         (A B: pred) (p : prog R) (xp: Go.var → stmt),
  (∀ var,
     COMPILE p
     {{ var : T * A }}
       xp var
     {{ λ ret ⇒ var : T * B ret }} // env) →
  COMPILE p
  {{ A }}
    Declare wrap_type xp
  {{ λ ret ⇒ B ret }} // env.
```

This lemma exposes the local nature of variable declarations. At any point, extracting any program `p` with precondition `A` and postcondition `B`, the compilation can decide to declare a new variable, introducing it into the precondition and the postcondition. The body of the declaration scope must be parametrized over the new variable identifier, and its compilation must work no matter which variable identifier the semantics pick; by using separation logic, the precondition ⟨`var : T * A`⟩ captures the fact that the new variable identifier is guaranteed to be fresh. Also, since `p` could be a nondeterministic program involving disk operations, the postcondition cannot be required to specify the value which the variable takes at the end of the block. Instead, it uses a ⟨`var : T`⟩ predicate to indicate that the variable may take any value.

4.3.3 Other constructs

We proved a compilation lemma for each Go′ construct. For example, to be able to compile cons operations into `Append` operations, we proved the following compilation lemma:

```
Lemma CompileAppend : ∀ (env: Env) (F: pred) {T: Type}
                        {Wr: GoWrapper T} (lvar vvar: var)
                        (x: T) (xs: list T),
  COMPILE Ret (x :: xs)
  {{ vvar ⇝ x * lvar ⇝ xs * F }}
    Modify Append ^(lvar, vvar)
  {{ λ ret ⇒ vvar ↦ moved_value (wrap x) * lvar ⇝ ret * F }} // env.
```

Roughly, for any frame `F` containing other variables, if `vvar` contains some value and `lvar` contains a list of elements of the same type, we can compile the expression `x :: xs` using the `Append` operation, leaving the result in `lvar` and moving out of `vvar`. Note the use of the different arrow symbol ↦ in the postcondition, which is the basic separation logic points-to arrow: a use of the usual squiggly arrow ⟨`a_var ⇝ a`⟩ is shorthand for ⟨`a_var ↦ wrap a`⟩. Thus we see that `CompileAppend` simply changes `vvar`'s value from `wrap x` to `moved_value (wrap x)`.

The proof of this lemma is largely automated; it simply inverts an execution of the `Append` operation and proves that the result corresponds to the desired postcondition, where the return value of the program has been substituted in.

4.3.4 Record deserialization

In a number of places, FSCQ deserializes fixed-size record types from disk. For example, FSCQ encodes inodes on disk using the following `Rec` type:

```
Definition irectype : Rec.type := Rec.RecF ([
  ("len", Rec.WordF addrlen);      (* number of blocks *)
  ("attrs", iattrtype);            (* file attributes *)
  ("indptr", Rec.WordF addrlen);   (* indirect block pointer *)
  ("blocks", Rec.ArrayF (Rec.WordF addrlen) NDirect)]).
```

At first, we tried to compile the inode deserialization code by unfolding the generic deserialization function `Rec.of_word`, partially evaluating it for the specific case of `irectype`, and running the normal comilation tactics. However, since (including the file attributes and the direct blocks) inodes have 20 different fields, this resulted in a large mess of code which caused the compilation tactics to run unacceptably slowly. Rather than optimizing the tactics, we found it easier to create a single compilation lemma which can compile any `Rec.of_word` invocation. A recursive Gallina function `go_of_word` (Figure 4-3) takes the `Rec.type` definition of the record type to be deserialized and outputs Go′ code which performs the deserialization. The corresponding compilation lemma `CompileOfWord` references `go_of_word` and proves that the code is correct for any `Rec.type`. In a sense, this compilation lemma is the proof-by-reflection analogue of the usual proof-producing compilation tactics.

```
Fixpoint go_of_word (t: Rec.type) (vdst vsrc: var) (from: ℕ) : stmt :=
  match t with
  | Rec.WordF n ⇒
    Declare Num (λ vfrom ⇒
      Declare Num (λ vto ⇒
        (Modify (@SetConst Num from) ^(vfrom);
         Modify (@SetConst Num (from + n)) ^(vto);
         Modify SliceBuffer ^(vdst, vsrc, vfrom, vto))))
  | Rec.ArrayF t' n₀ ⇒
    (fix array_of_word n from :=
       match n with
       | 0 ⇒ Modify (InitSliceWithCapacity n₀) ^(vdst)
       | S n' ⇒
         Declare (go_rec_type t') (λ vt' ⇒
           (array_of_word n' (from + Rec.len t');
            go_of_word t' vt' vsrc from;
            Modify Append ^(vdst, vt')))
       end) n₀ from
  | Rec.RecF fs ⇒
    (fix rec_of_word fs vdst from :=
       match fs with
       | [] ⇒ Modify (@SetConst (Struct []) tt) ^(vdst)
       | (_, f) :: fs' ⇒
         Declare (go_rec_type f) (λ vf ⇒
           Declare (go_rec_type (Rec.RecF fs')) (λ vfs' ⇒
             (go_of_word f vf vsrc from;
              rec_of_word fs' vfs' (from + Rec.len f);
              Modify JoinPair ^(vdst, vf, vfs'))))
       end) fs vdst from
  end%go.
```

Figure 4-3: The function which computes the Go′ code for any record deserialization

```
Lemma CompileOfWord : ∀ (t: Rec.type) (vdst vsrc: var) (F: pred)
                        (buf: immut_word (Rec.len t)) (env: Env),
  byte_aligned t →
  COMPILE Ret (@Rec.of_word t buf)
  {{ vdst : Rec.data t * vsrc ⇝ buf * F }}
    go_of_word t vdst vsrc 0
  {{ λ ret ⇒ vdst ⇝ ret * vsrc ⇝ buf * F }} // env.
```

4.4 Proof-producing compilation tactics in Ltac

The proof-producing compilation tactics form the heart of CoqGo. They compile Gallina to Go′ by repeatedly applying compilation lemmas to break apart the source program and compiling the pieces individually.

4.4.1 Key invariants

Given a compilation task of the form ⟨`COMPILE p {{ P }} _ {{ λ ret ⇒ R ret * _ }}`⟩, the proof-producing compilation tactics decide their next move based on three pieces of state, in decreasing order of importance:

1. The monadic program `p` being compiled
2. The way that the return value `ret` appears in the postcondition
3. Which values are already present in the precondition `P`, in what form

Thus, `p`, `P`, and `R` must always be determined already (not existential variables). Since the rest of the postcondition can be an existential variable, it must not be necessary to compare the precondition and the postcondition except in places where the postcondition depends on `ret`; the existential variable must not depend on `ret`. This set of invariants is always maintained for the currently active compilation task (i.e. the current Coq goal).

4.4.2 compile_bind

Whenever the compilation tactics encounter a monadic bind `Bind p f`, they use the `compile_bind` tactic to apply the `CompileBind` compilation lemma, breaking the compilation task into two.

The precondition of the first compilation task is the same as the precondition of the original compilation task, and the postcondition of the second compilation task is the same as the postcondition of the original task. However, the intermediate predicate (`B` in Section 4.3.1) is not determined by the goal, letting the compilation of `p` modify some of the values in `A` if necessary. This flexibility is necessary, particularly to account for variables which might get moved out of, but it introduces a complication into the compilation: it may not be possible to determine `B` until `p` is compiled. To handle this, `compile_bind` simply introduces an existential variable for `B`, letting it be filled in as `p` is compiled. Then, when `f a` is compiled, the full precondition will be known. Since `B` does not depend on the return value of `p`, it can be an existential variable without violating the invariants above.

4.4.3 compile_decompose

Since the source code is written in a functional language, it consists almost entirely of deeply nested expressions, yet Go′ does not have expressions at all. Thus, in the compilation output, expressions must be evaluated one step at a time by Go′ statements using variables as temporaries. It would be possible to write specialized compilation lemmas for each operation which accept hypotheses which generate additional compilation tasks for evaluating the subexpressions. For instance, a compilation lemma applied to compiling the addition `(a * b) + c` could accept a compilation of `a * b` and a compilation of `c`, and then return a sequence of those two compilations plus the addition. However, this would require a significant amount of boilerplate for every operation.

Thus, we opt for a more general approach: a single `compile_decompose` tactic looks at programs of the form `Ret (f a0 a1 ... an)`, where some function (or operator) `f` is applied to some number of arguments. If any argument is missing from the precondition, `compile_decompose` creates an additional compilation task to first evaluate the argument before proceeding. To reuse the existing `compile_bind` functionality, it simply extracts out the evaluation of the argument using the left monad identity law: `Ret (f expr)` is equivalent to `Bind (Ret expr) f`, or, with notation, ⟨`x` ← `Ret expr; f x`⟩. For instance, `Ret ((a * b) + c)` would be replaced with ⟨`ab` ← `Ret (a * b); Ret (ab + c)`⟩. The compilation would then proceed by evaluating `a * b` first and then passing the result to the addition, as desired.

In a few cases, when the arguments to the Go construct do not match the arguments to the Gallina construct, this general approach does not work. For example, the Gallina function to slice out a substring of a bitstring has the following signature, for dependent typing reasons:

```
Definition middle (low mid high: ℕ) (w: word (low + (mid + high)))
  : word mid.
```

It accepts three arguments, `low`, `mid`, and `high`, which partition the original bitstring into three substrings, the middle of which will be chosen. They must add up to the length of the original bitstring. In Go, on the other hand, this should be compiled into a slice operation of the form `w[low : low + mid]`, so the arguments which need to be in place are `low` and `low + mid`. Naively applying `compile_decompose` would have the compilation produce values `low`, `mid`, and `high` instead, which would waste time calculating the unnecessary `high` value and in some situations miss out on the opportunity to use `low + mid` directly, which may already be present. Thus, the `compile_middle` tactic specializes the compilation logic in this case, choosing `low` and `low + mid` as the values to put into the precondition. `compile_decompose` is run with low priority, so for programs of the form `Ret (middle low mid high w)`, it will never run.

4.4.4 Functions

The compilation tactics proceed one function at a time, with one Coq theorem per function. Roughly, each compilation theorem states that there exists some Go program such that for any argument values and any environment containing all functions which are called, the Go′ program is a valid compilation of the source function.

For simplicity, functions currently use a simple "calling convention" in which the first argument is used for the return value and the remaining arguments are passed afterwards. A more sophisticated design could take advantage of passing arguments by reference.

4.4.5 Tuples

As described in Section 4.1.6, Go′ has a single `SplitJoin` operation to split tuples rather than accessing just the first member or just the second member. The compilation tactics greedily split and join tuples, leaving them in the last needed state for as long as possible.

To accomplish this, we use two compilation lemmas, `CompileFst` and `CompileSnd`, which compile programs such as `Ret (fst p)` into `SplitPair` operations that entirely consume the pair, moving the other component of the pair into a different variable. The compilation tactics also recognize when a desired value is bound up in a pair, and rewrite the compilation task to extract the value. For example, if the program is `Ret a` and the precondition contains ⟨`p` ⇝ `(a, b)`⟩, then the compilation finds the `a` in the pair and switches to compiling the equivalent program `Ret (fst (a, b))` instead. Then, the `CompileFst` rule trivially applies, finding the value `(a, b)` in the variable `p`. This works for nested pairs as well; the compilation tactics produce compilation tasks such as `fst (snd (a, (b, c)))` which are then compiled one step at a time.

However, this tuple splitting plan causes a complication: The declaration of the variable to hold the other component of the pair can be scoped too tightly, causing it to go out of scope too early. Consider the following compilation task, in which a pair of two lists is to be swapped:

```
COMPILE Ret (l₂ , l₁)
{{ pair ⇝ (l₁ , l₂) }}
  ?p
{{ λ ret ⇒ pair ⇝ ret }}
```

The compilation tactics could first split off a task to generate the first element (l2) of the pair to return:

```
COMPILE Ret l₂
{{ tmp : list ℕ * pair ⇝ (l₁ , l₂) }}
  ?p
{{ λ ret ⇒ tmp ⇝ ret * ?F }}
```

Here, the compilation tactics rewrite the task to pull l2 out of the pair in the precondition:

```
COMPILE Ret (snd (l₁ , l₂))
{{ tmp : list ℕ * pair ⇝ (l₁ , l₂) }}
  ?p
{{ λ ret ⇒ tmp ⇝ ret * ?F }}
```

At this point, the compilation tactics would like to apply `CompileSnd` to split the pair, but must declare another variable to hold the first component of the pair (`l1`). Unfortunately, if defined at this point, the declaration block for that variable will only surround the `SplitPair` operation, so it will go out of scope immediately afterwards, and the value `l1` will be lost. Thus, we must move the declaration farther up the syntax tree, but the code farther up is already fixed, and the compilation tactics cannot know how far up the declaration must go, since they do not know yet for how long the variable still needs to be referenced.

4.4.6 Putting declarations at the outside

To solve the complication above, we simply move all declarations to the very top of the syntax tree. Given that the compilation must produce a proof of correctness as it goes, this move cannot happen as a post-processing step; instead, the declarations must be inserted into the top of the syntax tree in the proof state even as the compilation tactics are filling in the leaves of the syntax tree. In order to do this, when they begin compiling a function, the compilation tactics create a Coq existential variable `?decls` which will be gradually filled in with all the top-level declarations. At the very top of the syntax tree, the tactics apply a `CompileDeclareMany` compilation lemma, which declares a variable for each declaration specified in `?decls`, obtaining a list `vars` of variable identifiers from the semantics. To refer to variables declared in this fashion, the code must write `nth_var 0 vars`, `nth_var 1 vars`, etc. (These expressions will be evaluated in OCaml as the Go′ to Go printer evaluates the extracted Go′ AST.) `CompileDeclareMany` puts a `decls_pre ?decls 0` term into the precondition for its body, which evaluates to ⟨`nth_var 0 vars : T`$_0$ `* nth_var 1 vars : T`$_1$ `* ...`⟩ if `[T0; T1; ...]` is the list of types in `?decls`. Now, whenever a new variable must be declared, instead of adding a `Declare` block inline, the compilation tactics fill in the head of the `?decls` list with the desired type. As a result, a term of the form `decls_pre ?decls 0` in the precondition will simplify to ⟨`nth_var 0 vars : T`$_0$ `* decls_pre ?decls’ 1`⟩ (if `?decls'` is a new existential variable representing the rest of the declarations), and the compilation tactics can simply manipulate `nth_var 0 vars` as if it had been declared inline.

4.4.7 Tuples: projection approach

Another approach to supporting pair types would add `PairFst` and `PairSnd` projection operations which move the value in the first or second component of the pair into a different variable. As a result, if the component was of a type that cannot be aliased , it will be replaced by the appropriate `Moved` value.

Because of the way it leaves variables in a partially moved state, this approach does not interact well with the compilation tactics. The issue is that `Moved` states do not correspond to the wrapped version of any source Gallina value, meaning that the separation logic predicate describing the pair variable has to specify its value in unwrapped form. For instance, if before a `PairSnd` operation the predicate contained the points-to relation ⟨`p` ⇝ `([1], [2; 3])`⟩, equivalent to ⟨`p` ↦ `wrap ([1], [2; 3])`⟩, then afterwards the list would be replaced with `Moved`, so the postcondition would have to be expressed as ⟨`p` ↦ `Val (Pair (Slice Num) (Slice Num)) (wrap' [1], Moved)`⟩; unfolding the `wrap'` would leave ⟨`p` ↦ `Val (Pair (Slice Num) (Slice Num)) (Here [1], Moved)`⟩, Later, if the code accesses the value `[1]`, the compilation tactics may need to identify that the first component of the pair still contains it. As a result, the compilation tactics would need to recognize values in both wrapped and unwrapped form, potentially in deeply nested pairs. Worse, the task of extracting a value from a nested pair must be compiled in one step and cannot be decomposed as described for the splitting approach above: it would not be possible to express a compilation task such as `Ret (fst (Here [1], Moved))` since `(Here [1], Moved)` is a Go′ value, not a source value. It would certainly be possible to enable the compilation tactics to solve these problems, but it would be a fairly different approach than the rest of the system.

4.4.8 Other constructs

Once again, slice append can serve to illustrate the compilation of normal operations. The Ltac tactic to compile list cons operations has several tasks. First, it must ensure that the element `x` and the list `xs` are already present in the precondition. If not, `compile_decompose` will cause those values to be computed first. Then, it checks to see whether the variable containing `xs` is the same variable which is intended to hold the result of the computation in the postcondition. If so, it can just apply `CompileAppend` directly and match up the pre- and postconditions. If not, it must first move `xs` into the return variable using a different compilation lemma, and then proceed.

Chapter 5

Implementation

This chapter gives various details of the implementation. It gives an overview of the components and their sizes in lines of code (Section 5.1), details one issue in the Go′ to Go printer (Section 5.2), explains difficulties in the proof-producing compilation caused by massive proof terms (Section 5.3), and finally shows an example of the output of CoqGo (Section 5.4).

Component	**Lines of code**	**Language(s)**
Go′ semantics (§ 4.1)	1610	Gallina
Compilation lemmas (§ 4.3)	3770	Gallina, Ltac
Proof-producing compilation (§ 4.4)	1480	Ltac
Go′ to Go printer (§ 3.5)	580	OCaml
Go support code	780	Go
Compilation of FSCQ (§ 8.2)	2260	Ltac
FSCQ source code changes (§ 6.1)	40	Gallina
Miscellaneous	930	Gallina
Total	11450	

Table 5.1: The size of CoqGo and its application to FSCQ

5.1 Overview

We implemented CoqGo in Coq and compiled 30 of FSCQ's functions with it (out of about 300), including the top-level functions `AsyncFS.file_get_attr` and `AsyncFS.file_get_sz` and all of the functions which they call. The two are effectively the same function except that `AsyncFS.file_get_attr` returns the entire attribute struct but `AsyncFS.file_get_sz` returns only the size field. In the Haskell FUSE driver, `AsyncFS.file_get_attr` implements the `stat` system call, and `AsyncFS.file_get_sz` helps implement the `read` and `write` system calls. We have not yet written a FUSE driver for Go because we did not compile a sufficient portion of the file system. The implementation comprises 11,450 lines of code added to FSCQ in the components described by Table 5.1. We worked using Proof General with the Company-Coq extension [17], which also formatted the code in this thesis.

5.2 Go′ to Go printer: Working around Go's lack of generics

Go does not have user-defined polymorphic types, so we cannot write a generic pair struct that will work once and for all without type casts. Thus, as the Go′ to Go printer traverses the ASTs to convert them into code, it collects a list of all the pair types which are used in order to declare them all as structs at the top of the output file. It also implements a `DeepCopy` method on every pair struct, which is used to implement the `Duplicate` operation in Go′ (see Section 4.1.1).

Go's built-in maps and slices are polymorphic, but the printer still collects all the map and slice types referenced in order to define type aliases for them as well. That way, `DeepCopy` can also be defined on those types.

5.3 Challenge: Massive proof terms

The most significant implementation challenge was to deal with occasionally massive proof terms which came out of the compilation tactics, leading to memory exhaustion, stack overflows, and slow compilation times. The compilation tactics manipulate large Coq goals stating typed separation logic predicates over dozens of variables, including deeply nested pairs, over a significant number of compilation steps. Often, the separation logic cancellation tactics produced particularly large terms; we solved this issue with an alternative implementation solving most separation logic implications using proof by reflection.

Even after some optimization effort, compiling the subset of FSCQ with CoqGo still requires increasing the allowed stack size with `ulimit -s unlimited` and using at least 16 GB of RAM. Without the stack size increase, the compilation can overflow during the type checking performed once the compilation for a function concludes with a `Defined`. In addition, the OCaml compiler overflows its stack trying to parse the literals for 4-kilobyte disk blocks.

5.4 Example compilation output

Figure 5-1 shows the source code for the writeback function in FSCQ's cache. It takes an address `a` and a cache state `cs`. If the block with that address is present in the cache and has its dirty bit set to `true`, then the cache writes the block out to disk and creates a new cache state which is identical except that the block's dirty bit is set to `false`.

Figure 5-2 shows the CoqGo output for this function, annotated with a few additional comments. Parameter `val0` is the return parameter (holding the modified cachestate); parameter `val1` holds `a` and `val2` holds the input cachestate. The code shows significant low-hanging fruit remaining for optimization: ideally, only a single parameter would

```
Definition writeback (a: addr) (cs: cachestate) :=
  match (Map.find a (CSMap cs)) with
  | Some (v, true) ⇒
    Write a v;;
    Ret (mk_cs (Map.add a (v, false) (CSMap cs))
               (CSMaxCount cs)
               (CSEvict cs))
  | _ ⇒
    Ret cs
  end.
```

Figure 5-1: The Coq source code for `Cache.writeback`

be used for the cachestate, which would be properly passed by reference, and the unmodified `CSMaxCount` and `CSEvict` fields would not need to be touched at all.

```
func Cache_writeback(val0 *Pair_Pair_AddrMap_Pair_ImmutableBuffer_Bool_Num_Struct_,
                     val1 *Num,
                     val2 *Pair_Pair_AddrMap_Pair_ImmutableBuffer_Bool_Num_Struct_) {
  var val3 Pair_Bool_Pair_ImmutableBuffer_Bool
  _ = val3 // prevent 'unused' error
  var val4 AddrMap_Pair_ImmutableBuffer_Bool
  _ = val4
  var val5 Pair_AddrMap_Pair_ImmutableBuffer_Bool_Num
  _ = val5
  var val6 Struct_
  _ = val6
  var val7 Num
  _ = val7
  var val8 Bool
  _ = val8
  var val9 Pair_ImmutableBuffer_Bool
  _ = val9
  var val10 Bool
  _ = val10
  var val11 ImmutableBuffer
  _ = val11
  var val12 Pair_AddrMap_Pair_ImmutableBuffer_Bool_Num
  _ = val12
  var val13 AddrMap_Pair_ImmutableBuffer_Bool
  _ = val13
  var val14 Pair_ImmutableBuffer_Bool
  _ = val14
  var val15 Bool
  _ = val15
  var val16 Pair_AddrMap_Pair_ImmutableBuffer_Bool_Num
  _ = val16
  var val17 Pair_AddrMap_Pair_ImmutableBuffer_Bool_Num
  _ = val17
  // Split cachestate to access CSMap
  val5, val6 = (*val2).Fst, (*val2).Snd
  val4, val7 = val5.Fst, val5.Snd
  { // Map.find
    in_map, val := AddrMap(val4).Find((*val1))
    _ = val
    val3.Fst = Bool(in_map)
    if in_map {
      val.(Pair_ImmutableBuffer_Bool).DeepCopy(&val3.Snd)
    }
  }
  val8, val9 = val3.Fst, val3.Snd
  if bool(val8) {
    val11, val10 = val9.Fst, val9.Snd
    if bool(val10) {
      DiskWrite((*val1), val11)
      val13 = val4
      val15 = false
      // Reconstruct modified cachestate
      val14.Fst, val14.Snd = val11, val15
      AddrMap(val13).Insert(*val1, val14)
      val12.Fst, val12.Snd = val13, val7
      (*val0).Fst, (*val0).Snd = val12, val6
    } else {
      val16.Fst, val16.Snd = val4, val7
      (*val0).Fst, (*val0).Snd = val16, val6
    }
  } else {
    val17.Fst, val17.Snd = val4, val7
    (*val0).Fst, (*val0).Snd = val17, val6
  }
}
```

Figure 5-2: The Go code produced for `Cache.writeback`

Chapter 6

Evaluation

We evaluate CoqGo to answer three questions:

1. How much did we need to change FSCQ's source code to compile it with CoqGo? (Section 6.1)

2. What CPU performance does the compiled code achieve? (Section 6.3)

3. How much control over performance does CoqGo enable? (Section 6.4)

6.1 Changes to FSCQ source code

Applying CoqGo required minimal changes to FSCQ's source code: only 40 lines needed to be modified for the subset of FSCQ which we compiled. The changes came in two flavors:

- To reduce copying (see Section 6.4.1), we marked certain buffers as immutable by using `immut_word` in the source code rather than `word`. This changed about 15 lines.

- FSCQ's code makes use of higher order functions in a few places: it has generalpurpose functions to scan through blocks or scan through directory entries to find one satisfying a predicate, which is passed in as a bool-valued function. Go does support higher-order functions, but the Go′ semantics do not, so we had to specialize the scanning function for each predicate. (We did not have to manually copy the code—effectively, we just used a small amount of Ltac to perform the specialization.) This changed about 25 lines.

6.2 Benchmarking methodology

We evaluate the performance of CoqGo's output using microbenchmarks of the two top-level FSCQ functions we compiled, `file_get_attr` and `file_get_sz`, and comparing the performance between CoqGo's output and Haskell.

6.2.1 Haskell measurement

Thanks to Haskell's lazy evaluation, the Haskell functions proved fairly difficult to measure. There are several pitfalls. If the output of the benchmarked function is not used in some fashion, it will not be evaluated at all, potentially speeding up the execution by a significant factor. On the other hand, if the benchmark code naively chains the outputs of each iteration together (e.g. by summing them), then it can build up a large tree of lazy "thunks" and cause more garbage collection expense than under a realistic use of the top-level function, where its output would be evaluated immediately in order to pass it back to FUSE. Precisely which level of performance should be counted is not clear; to mimic the behavior of running under FUSE, our benchmark code uses the `deepseq` package to force the entire evaluation of the previous iteration's output before the current iteration is started. The benchmark measures performance using the x86 `rdtsc` cycle counting instruction.

6.2.2 Go measurement

As a strict imperative language, Go is much more straightforward to benchmark. The Go benchmark code uses `rdtsc` to count cycles as well as using Go's built-in benchmarking tools. We also used Go's profiling support to determine where the code spends its time.

6.2.3 Benchmarking hardware

We ran the benchmarks on a Dell XPS 15 9550 with an Intel® Core™ i7-6700 HQ processor. The disk drive is a Toshiba 256GB NVMe drive with model number THNSN5256GPU7, although after the initial iteration, the benchmarks run entirely off of FSCQ's buffer cache. We used Arch Linux with kernel version 4.10.11 and the "performance" CPU frequency governor.

6.3 Performance

Figure 6-1 shows the benchmark results. On `file_get_attr`, CoqGo brings the performance from 3530 cycles to 2850 cycles, a 19% speedup. Since the benchmark repeatedly reads the attributes of a small set of inodes, all disk blocks accessed will be resident in FSCQ's buffer cache, so the improvement is entirely in CPU performance. However, significant optimization opportunities remain. Using the Go profiler to measure the execution of `file_get_attr` shows the impact of the remaining inefficiences of the code—see Figure 6-2. A combined 54% of CPU time is spent in `runtime.duffzero` and `runtime.duffcopy`, which are used zero-initialize variables upon allocation and copy large fields between variables, respectively.

Since `file_get_attr` and `file_get_sz` are effectively the same function except that `file_get_sz` reads the size field, they perform identically in Go. However, in Haskell,

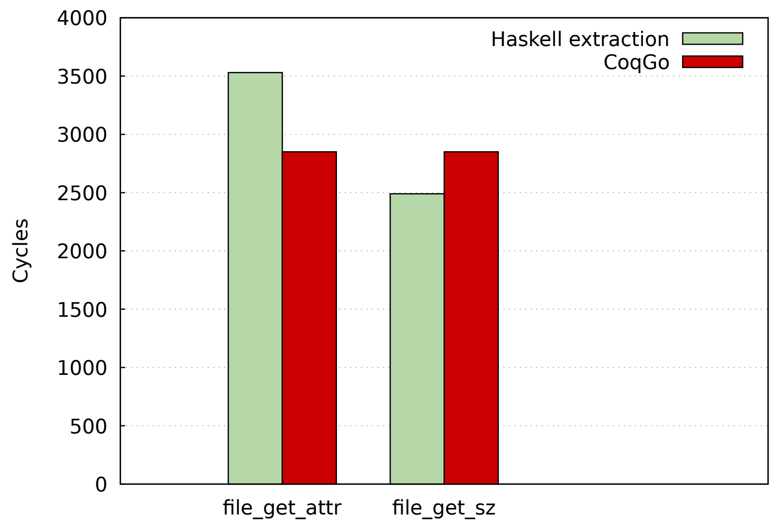

Figure 6-1: The performance of two top-level functions under the two compilation pipelines

`file_get_sz` can exploit laziness in order to only perform the deserialization necessary to read the size field, omitting the rest of the computation. Since the deserialization is costly in Haskell, this enables the Haskell pipeline to beat CoqGo's performance on `file_get_sz`.

6.4 Control over performance: optimizations

We investigate three optimizations, finding that CoqGo gives enough control to enable significant optimizations with only a small effort.

Without any of the optimizations, `file_get_attr` takes 68,000 cycles to run, or about 26 microseconds. With a total of only +138 / −143 lines of code changed, the three optimizations take this down to 2850 cycles.

Optimization	**Cycles**	**Lines added / removed**
Zero-initialization	68,000 $\mapsto$ 6200 (−91%)	+19 / −49
Immutable disk blocks in cache	6200 $\mapsto$ 4150 (−33%)	+71 / −80
Hint slice capacity	4150 $\mapsto$ 2850 (−31%)	+47 / −8

Table 6.1: The three optimizations performed and their effect on `file_get_attr`

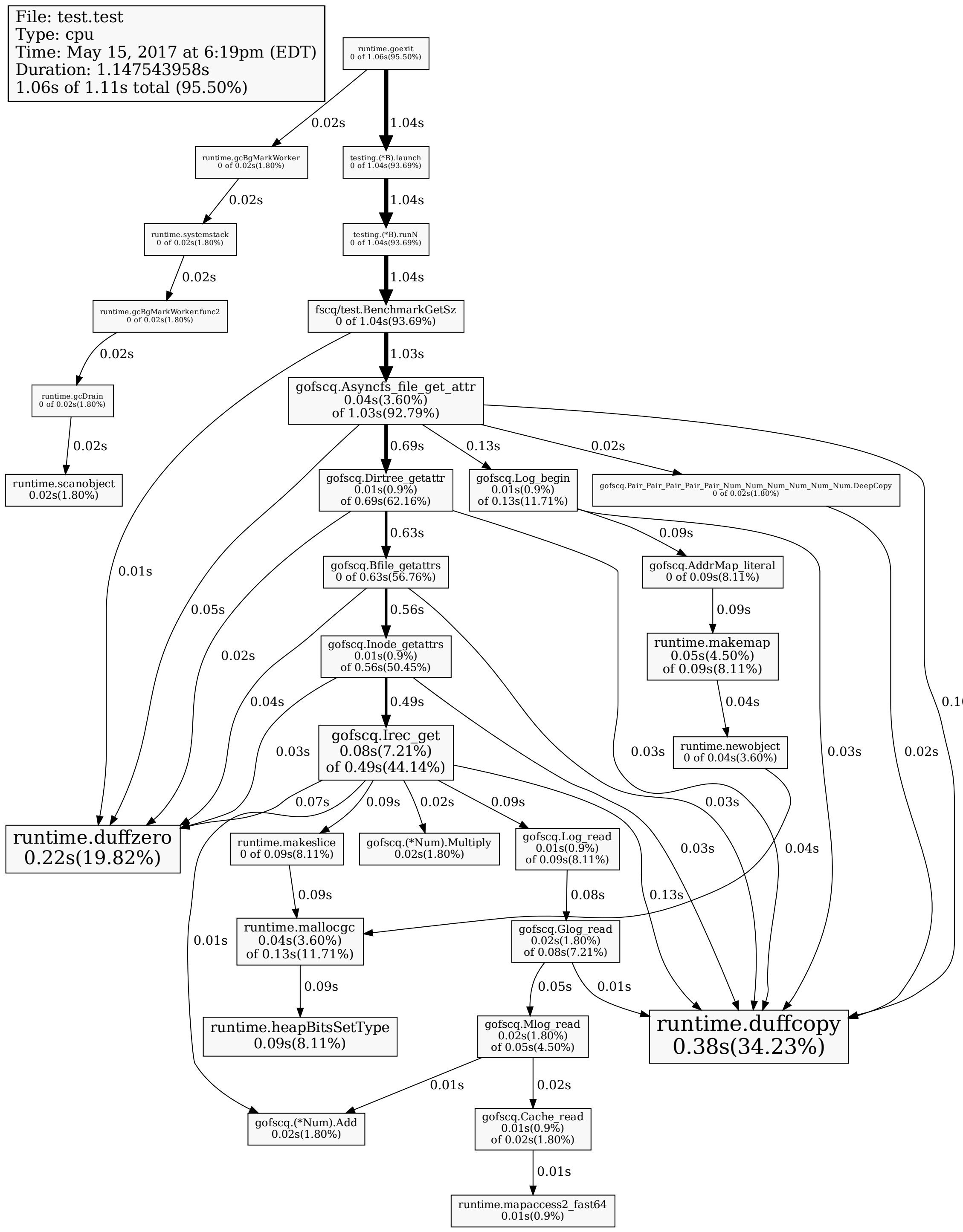
File: test.test
Type: cpu
Time: May 15, 2017 at 6:19pm (EDT)
Duration: 1.147543958s
1.06s of 1.11s total (95.50%)
runtime.goexit 0 of 1.06s(95.50%)
0.02s
1.04s
runtime.gcBgMarkWorker 0 of 0.02s(1.80%)
testing.(*B).launch 0 of 1.04s(93.69%)
0.02s
1.04s
runtime.systemstack 0 of 0.02s(1.80%)
testing.(*B).runN 0 of 1.04s(93.69%)
0.02s
1.04s
runtime.gcBgMarkWorker.func2 0 of 0.02s(1.80%)
fscq/test.BenchmarkGetSz 0 of 1.04s(93.69%)
0.02s
1.03s
runtime.gcDrain 0 of 0.02s(1.80%)
gofscq.Asyncfs_file_get_attr 0.04s(3.60%) of 1.03s(92.79%)
0.02s
0.69s
0.13s
0.02s
runtime.scanobject 0.02s(1.80%)
gofscq.Dirtree_getattr 0.01s(0.9%) of 0.69s(62.16%)
gofscq.Log_begin 0.01s(0.9%) of 0.13s(11.71%)
gofscq.Pair_Pair_Pair_Pair_Pair_Num_Num_Num_Num_Num_Num.DeepCopy 0 of 0.02s(1.80%)
0.63s
0.09s
0.01s
gofscq.Bfile_getattrs 0 of 0.63s(56.76%)
gofscq.AddrMap_literal 0 of 0.09s(8.11%)
0.05s
0.56s
0.09s
0.02s
gofscq.Inode_getattrs 0.01s(0.9%) of 0.56s(50.45%)
runtime.makemap 0.05s(4.50%) of 0.09s(8.11%)
0.04s
0.49s
0.04s
0.10s
0.03s
0.03s
0.03s
0.02s
gofscq.Irec_get 0.08s(7.21%) of 0.49s(44.14%)
runtime.newobject 0 of 0.04s(3.60%)
0.07s
0.09s
0.02s
0.09s
0.03s
runtime.duffzero 0.22s(19.82%)
runtime.makeslice 0 of 0.09s(8.11%)
gofscq.(*Num).Multiply 0.02s(1.80%)
gofscq.Log_read 0.01s(0.9%) of 0.09s(8.11%)
0.03s
0.04s
0.09s
0.08s
0.13s
0.01s
runtime.mallocgc 0.04s(3.60%) of 0.13s(11.71%)
gofscq.Glog_read 0.02s(1.80%) of 0.08s(7.21%)
0.09s
0.05s
0.01s
runtime.heapBitsSetType 0.09s(8.11%)
gofscq.Mlog_read 0.02s(1.80%) of 0.05s(4.50%)
runtime.duffcopy 0.38s(34.23%)
0.01s
0.02s
gofscq.(*Num).Add 0.02s(1.80%)
gofscq.Cache_read 0.01s(0.9%) of 0.02s(1.80%)
0.01s
runtime.mapaccess2_fast64 0.01s(0.9%)

Figure 6-2: Go's profiling output for `file_get_attr`

6.4.1 Enabling zero-initialization of all data types

A large portion of the baseline runtime is spent running type initialization functions such as `gofscq.New_Pair_Buffer_Pair_AddrMap_Buffer_Struct__`, which are called on each variable declaration to initialize the type. This is only necessary in order to initialize maps to a valid empty state, since a `nil` slice acts as an empty slice in Go but a `nil` map panics if accessed. However, these initialization functions are unnecessary almost everywhere they are called, since the fields are generally overwritten anyway. To eliminate them, we changed the Go′ semantics to initialize maps in the `Moved` state, and removed the generation and invocation of the `New_***` functions in the Go′ to Go printer; now Go's zero value works for all types. No changes were necessary to the compilation tactics or the compilations themselves—in the few places where it was actually required, the compilation tactics were already capable of initializing map values with an empty literal. In total, 49 lines were deleted and 19 lines were inserted. This optimization had a dramatic impact, speeding up `file_get_attr` by a factor of 10.

6.4.2 Making the cache hold immutable disk blocks

By default, CoqGo compiles FSCQ's buffer cache to hold mutable disk blocks. This means that they have to be copied on every access, even for read-only operations, since it's not possible to have multiple references to a mutable value. We changed the Gallina type of the values held in the cache from `word` to `immut_word`, making no semantic difference in the source but indicating to CoqGo that those values should be compiled as immutable buffers. This optimization caused a 33% speedup when applied after the first optimization. It required changing the type in a few places for consistency and adapting some compilations slightly, for a total of 71 insertions and 80 deletions.

The small number of lines changed does underestimate the effort the change required, however: since the type signatures of a number of functions were affected and the compilation takes minutes to compile every function, each iteration that we performed to fix the compilation required waiting a significant amount of time. Moreover, it was not always obvious what had gone wrong—typically, some predicate implication failed because a variable was of Go$'$ type `Buffer` on one side and `ImmutableBuffer` on the other side, and then we had to track down the right location to change another `word` to `immut_word` to make everything line up. Future improvements to the automation could eliminate this hassle.

6.4.3 Hinting slice capacity in deserialization

Arrays in on-disk fixed-size records have statically known length, but they are still compiled to Go using variable-length slices. As a result, when the compiled Go code for inode deserialization repeatedly appends to the slice for direct blocks, the slice has to be reallocated several times to account for the increased length. To mitigate this, we introduced a new Go$'$ operation which initializes a slice with a given capacity, applying it in the compilation lemma for record deserialization (see Section 4.3.4). This optimization cut a third off the runtime again. It required 47 insertions and 8 deletions; since no changes were required to compilations, the change went quickly.

Chapter 7

Related Work

7.1 Fiat to Facade compilation

Our approach draws heavily on the proof-producing compilation approach used as part of the verified pipeline from the Fiat language [5] to assembly. This Fiat-to-Facade stage, written by Pit-Claudel [16], sits in the middle of the pipeline and, using proof-producing tactics implemented within Coq, translates non-deterministic functional programs into Facade, a simple imperative language which enforces linear use of heap-allocated data. The setting has significant similarities to that of CoqGo's proof-producing compilation: both source languages are monadic shallow embeddings in Gallina; both target languages (Facade and Go$'$) restrict a more complete imperative language by enforcing linearity. However, due to some differences in the setting and some differences in the goals, many details of our approach ended up diverging significantly.

Go requires typed variable declarations. In Facade, to make a new variable, programs can simply assign to any previously unused variable name. Programs can also change the type of the value stored in a variable. Both of these do not work in

Go, so we needed to make Go′ support scoped variable declarations, as described in Section 4.1.4.

The monad in our source language has effects. Fiat source programs are written in a monad that allows for nondeterminism but not side effects, so execution order does not matter. The Fiat to Facade compiler takes advantage of this by making each variable in its pre- and postconditions contain the result of a monadic computation, rather than just a simple value; it has no separate monadic program on the side. It would be possible to use a similar approach for CoqGo, but since monadic computations in `prog` have side effects (disk operations), this would introduce some compilations. The currently unordered pre- and postconditions would have to become ordered. Then, the statement of a compilation ⟨`COMPILE {{ P }} xp {{ Q }}`⟩ would have to mean something like "If executed from left to right, the computations in the precondition `P`, when followed by the execution of the Go program `xp`, have the same effects as the computations in the postcondition `Q` executed from left to right"; mixing variable values and effects in this manner would be rather cumbersome.

We do not aim for extensibility without modification of CoqGo itself. The Fiat to Facade compiler goes to significant lengths to support adding new operations or data types without modifying the semantics or the core compilation tactics themselves. Since we are perfectly happy to make changes to CoqGo to support new source language constructs or applications, we were able to simplify Go′ and the compilation tactics somewhat.

7.2 Alternative proof-producing compilers

A number of projects explore related approaches to proof-producing compilation.

Lammich [10] has developed a similar system within Isabelle/HOL which refines from monadic programs in the Isabelle Refinement Framework to the Imperative/HOL language. Somewhat like CoqGo, the tool works by largely automated proof-producing refinement within a proof assistant. However, there are a few key differences. For one, the sort of source programs which are supported is very different—Imperative/HOL aims primarily at classic data structures and algorithms, whereas CoqGo aims to compile a file system. Also, the outputs are still monadic functional programs, just in an effectful monad that has access to a heap; the refinement doesn't have to completely transform into an imperative language.

Recently, a team at the University of Washington has started working on Œuf [13], which also aims to extract systems code written in Gallina to efficient imperative code. Instead of using compilation tactics written in Ltac, Œuf uses an approach analogous to proof by reflection: the compiler first uses the smallest amount of Ltac possible to reify the Gallina code into a deeply embedded AST, and then proceeds using verified compilation, producing input for the CompCert verified C compiler [11]. This approach seems like an appealing way to avoid the Ltac difficulties described in Section 8.2, but proving the entire compiler correct is likely significantly more challenging than proving individual compilation lemmas.

7.3 Alternatives to proof-producing compilation

Proof-producing compilation is not the only way to turn a verified program into an executable.

As described in Section 2.3, the current version of FSCQ compiles using Coq's extraction mechanism, enabling easy production of running code but sacrificing performance and verification.

Many projects employ verified compilation from variety of source languages. The recently launched CertiCoq project [1] is working on a verified compiler for Gallina; Œuf [13] can also be described as a verified compiler for Gallina. Other verified compilers use different source languages: CompCert [11] compiles C to assembly; Cogent [15], a linearly typed language with semantics like those of Go′, has a verified compiler to C; CakeML [9] is a functional programming language with a verified compiler to x86-64 machine code. Using a different source language has the potential to make compilation more straightforward but can sometimes make reasoning about the source language more difficult.

Chapter 8

Future Work

8.1 Further optimizations

Section 6.3 yields numerous opportunities for further optimizations which could enable FSCQ to become competitive with ext4 in CPU performance.

8.1.1 Compile to structs rather than nested pairs

CoqGo's current output wastes CPU time unpacking deeply nested pairs one step at a time, copying out ever smaller portions of the whole struct. Directly compiling Coq record types to Go structs would avoid this inefficiency. This would require updating Go′ and the compilation tactics, as well as requiring a small amount of additional boilerplate per record type from the user.

8.1.2 Make the tuple projection plan work

The unimplemented tuple projection plan described in Section 4.4.7, in which tuple components can be accessed individually rather than by splitting the whole tuple apart, could increase performance. Components which are not needed would not have to be copied out. Then, as discussed in Section 5.4, the tactics could recognize when certain components are left unchanged across a functional record update, entirely eliminating the cost of copying the components in and out. This optimization would work together with the previous optimization (Section 8.1.1).

8.1.3 Take advantage of passing arguments by reference

The current simplified calling convention (see Section 4.4.4) does not take advantage of the by-reference parameters in Go′; each parameter is either only an input parameter or only an output parameter. Often, the Gallina return value is actually a functional update of one of the parameters, in which case the compilation tactics will already have turned the functional update into an imperative update; returning it in the same parameter could eliminate a copy. In most cases, this would have to be combined with the projection plan to provide a benefit.

8.1.4 Pass tuple components / struct fields by reference

A common pattern in C or in Go is to pass a field of a larger struct into a function by address, enabling the function to mutate the field without requiring any copying. It would be fairly straightforward to let Go′ support this pattern as well. This would be particularly useful for code fragments like the following—the `MSLL` field of `fms` is passed to `INODE.getattrs`, and when the mutated version is returned, it is recombined with the unmodified `al`. The whole record splitting and recombining could be replaced with simply passing `fms.(MSLL)` to `INODE.getattrs` by reference.

```
Definition getattrs lxp ixp inum fms :=
  let '(al, ms) := (fms.(MSAlloc), fms.(MSLL)) in
  let^ (ms, n) ← INODE.getattrs lxp ixp inum ms;
  Ret ^(mk_memstate al ms, n).
```

However, it's not clear how the automation should recognize this situation. Here, rather than writing convoluted source code which is then simplified by the compiler, it may be better to change the source language to have native support for this pattern in the first place, simplifying both the source code and the compilation process.

8.2 More usable proof-producing compilation tactics

A key bottleneck for the continued development and use of CoqGo is the unreliable and slow nature of the proof-producing compilation tactics. Ltac is a notoriously nasty language to work with, and it requires a high level of discipline in addition to some trial and error to produce tactics which consistently work and consistently run quickly. As it stands, CoqGo's tactics run fairly slowly and do not cover quite all of the cases that appear in the code that we have compiled so far; sometimes, making a small change to any part of the system (the source code, Go$'$, or the tactics themselves) can result in a previously working tactic falling apart in an unexpected way. This has several unfortunate effects:

- The compilations of many functions are only semi-automated, resulting in a significant amount of proof code. (See Table 5.1.)

- Developing the tactics is a very cumbersome process. Adding new functionality or extending a tactic to cover a previously unsupported situation often causes some part of FSCQ to not compile anymore. However, it's only possible to tell that after letting the whole compilation process run; thanks to the poor performance of Ltac and our tactics themselves, this takes upwards of 15 minutes on 8 logical cores even for the small fraction of FSCQ we have compiled. Then, debugging the issue requires manually inlining the compilation tactics in the proof assistant and stepping through them until it's obvious what has gone wrong, waiting minutes for some sequences. Once a potential fix has been made, the cycle repeats.

- Compiling new pieces of code is also very cumbersome. Generally, a new piece of code will hit some case which the compilation tactics do not yet support (possibly due to a bug), and fixing the problem requires a similarly tedious debugging loop as the one just described.

Increasing the reliability and performance of the compilation tactics would significantly increase the productivity of working with CoqGo. There are a few clear places to start:

- The compilation tactics greedily mutate values if they can even when the old value is actually still needed later. Thus, when compiling a function where that is the case, the user has to manually break the compilation process into steps and insert a copy at the appropriate point, which is very cumbersome. However, making the tactics recognize where a copy is necessary would be somewhat difficult. A simpler plan would let the user annotate where copies should happen in the source.

- The compilation tactics use rather slow separation logic cancellation tactics, even though this application of separation logic is completely trivial. It's worth considering moving away from separation logic entirely. Since the compilation tactics put all the variables in a list anyway (see Section 4.4.6), the pre- and postconditions could simply be lists of predicates on the variable values, where predicate n applies to variable n.

- It may also be possible to bypass the limitations of working with Ltac and increase performance by applying proof by reflection to various parts of the compilation tactics.

8.3 Complete compilation of FSCQ

Currently, we have only compiled roughly a tenth of FSCQ's source code. Given improvements to the compilation tactics, it could become fairly easy to compile the entire thing. As more source constructs are reliably supported, more parts of FSCQ should compile fully automatically on the first try, meaning that progress should accelerate as it continues.

Once FSCQ is compiled completely, one could implement a FUSE driver in Go that calls into FSCQ, creating a properly usable file system. Additionally, if compiled into C (or integrated into an operating system written in Go), FSCQ could be placed into the kernel, removing the overhead of FUSE.

8.4 Verified compilation down to assembly

CoqGo is designed to improve performance, but it does not meaningfully reduce the trusted code base, which still includes Coq extraction, the Go′ to Go printer, the Go library, and the Go compiler and runtime. In future work, a verified compilation pass could compile Go′ to a different language which already has a verified compiler, such as CompCert's encoding of C [11], or the Fiat pipeline's Cito language [21]. Then, applications could be proved correct all the way down to the assembly level.

8.5 Compiling other systems

CoqGo could be applied to other systems as well. The currently supported source language constructs are fairly tailored to FSCQ, but with some extensions, CoqGo could be useful for many applications which are written in Gallina.

Chapter 9

Conclusion

This thesis contributes CoqGo, a compiler from Gallina to Go which is applied to the FSCQ file system. By employing Go′, a stylized semantics of Go which enforce linearity, CoqGo simplifies the task of implementing proof-producing compilation from Gallina, and produces the final output with a straightforward translation from Go′ to Go.

We built prototype of CoqGo and used it to compile one of FSCQ's system calls, replacing FSCQ's existing extraction pipeline with minimal changes to FSCQ's source code. Taking advantage of the increased control given by CoqGo, we implemented three optimizations which bring the system call's CPU performance to 19% faster than the extracted version. With further work, CoqGo has the potential to make whole verified systems fast.

Bibliography

[1] A. Anand, A. W. Appel, G. Morrisett, Z. Paraskevopoulou, R. Pollack, O. S. Bélanger, M. Sozeau, and M. Weaver. CertiCoq: A verified compiler for Coq. In *Proceedings of the 3rd International Workshop on Coq for Programming Languages*, Paris, France, Jan. 2017. URL https://www.cs.princeton.edu/~appel/papers/certicoq-coqpl.pdf.

[2] H. Chen, D. Ziegler, T. Chajed, A. Chlipala, M. F. Kaashoek, and N. Zeldovich. Using Crash Hoare Logic for certifying the FSCQ file system. In *Proceedings of the 25th ACM Symposium on Operating Systems Principles (SOSP)*, Monterey, CA, Oct. 2015.

[3] A. Chlipala. From network interface to multithreaded web applications: A case study in modular program verification. In *Proceedings of the 42nd ACM Symposium on Principles of Programming Languages (POPL)*, Mumbai, India, Jan. 2015.

[4] Coq development team. *The Coq Proof Assistant Reference Manual, Version 8.6*. INRIA, Apr. 2016. http://coq.inria.fr/distrib/current/refman/.

[5] B. Delaware, C. Pit-Claudel, J. Gross, and A. Chlipala. Fiat: Deductive synthesis of abstract data types in a proof assistant. In *Proceedings of the 42nd ACM Symposium on Principles of Programming Languages (POPL)*, Mumbai, India, Jan. 2015.

[6] R. Gu, Z. Shao, H. Chen, X. N. Wu, J. Kim, V. Sjöberg, and D. Costanzo. CertiKOS: An extensible architecture for building certified concurrent OS kernels. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 653–669, GA, 2016. USENIX Association. ISBN 978-1-931971-33-1. URL https://www.usenix.org/conference/osdi16/technical-sessions/presentation/gu.

[7] C. Hawblitzel, J. Howell, M. Kapritsos, J. R. Lorch, B. Parno, M. L. Roberts, S. Setty, and B. Zill. IronFleet: Proving practical distributed systems correct. In *Proceedings of the 25th ACM Symposium on Operating Systems Principles (SOSP)*, Monterey, CA, Oct. 2015.

[8] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, M. Norrish, R. Kolanski, T. Sewell, H. Tuch, and S. Winwood. seL4: Formal verification of an OS kernel. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles (SOSP)*, pages 207–220, Big Sky, MT, Oct. 2009.

[9] R. Kumar, M. O. Myreen, M. Norrish, and S. Owens. Cakeml: A verified implementation of ml. In *Proceedings of the 41st ACM Symposium on Principles of Programming Languages (POPL)*, pages 179–191, New York, NY, USA, Jan. 2014. ACM. ISBN 978-1-4503-2544-8. doi: 10.1145/2535838.2535841. URL http://doi.acm.org/10.1145/2535838.2535841.

[10] P. Lammich. Refinement based verification of imperative data structures. In *Proceedings of the 5th ACM SIGPLAN Conference on Certified Programs and Proofs*, CPP 2016, pages 27–36, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-4127-1. doi: 10.1145/2854065.2854067. URL http://doi.acm.org/10.1145/2854065.2854067.

[11] X. Leroy. Formal verification of a realistic compiler. *Communications of the ACM*, 52(7):107–115, July 2009.

[12] P. Letouzey. Extraction in Coq: An overview. In *Conference on Computability in Europe*, pages 359–369. Springer, 2008.

[13] E. Mullen, S. Pernsteiner, J. R. Wilcox, Z. Tatlock, and D. Grossman. Œuf: Verified Coq extraction in Coq, 2016. URL http://oeuf.uwplse.org/.

[14] M. O. Myreen and S. Owens. Proof-producing synthesis of ML from higher-order logic. In *Proceedings of the 12th ACM SIGPLAN International Conference on Functional Programming (ICFP)*, pages 115–126, New York, NY, USA, Sept. 2012. ACM. ISBN 978-1-4503-1054-3. doi: 10.1145/2364527.2364545. URL http://doi.acm.org/10.1145/2364527.2364545.

[15] L. O'Connor, C. Rizkallah, Z. Chen, S. Amani, J. Lim, Y. Nagashima, T. Sewell, A. Hixon, G. Keller, T. C. Murray, and G. Klein. COGENT: certified compilation for a functional systems language. *CoRR*, abs/1601.05520, 2016. URL http://arxiv.org/abs/1601.05520.

[16] C. Pit-Claudel. Compilation using correct-by-construction program synthesis. Master's thesis, Massachusetts Institute of Technology, Sept. 2016.

[17] C. Pit-Claudel and P. Courtieu. Company-Coq: Taking Proof General one step closer to a real IDE. In *Proceedings of the 2nd International Workshop on Coq for Programming Languages*, St. Petersburg, FL, Jan. 2016.

[18] H. Sigurbjarnarson, J. Bornholt, E. Torlak, and X. Wang. Push-button verification of file systems via crash refinement. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 1–16, GA, 2016.

USENIX Association. ISBN 978-1-931971-33-1. URL https://www.usenix.org/conference/osdi16/technical-sessions/presentation/sigurbjarnarson.

[19] P. Wadler. Linear types can change the world! In *Programming Concepts and Methods*, 1990.

[20] P. Wadler. Monads for functional programming. In *Advanced Functional Programming, First International Spring School on Advanced Functional Programming Techniques-Tutorial Text*, pages 24–52, London, UK, UK, 1995. Springer-Verlag. ISBN 3-540-59451-5. URL http://dl.acm.org/citation.cfm?id=647698.734146.

[21] P. Wang, S. Cuellar, and A. Chlipala. Compiler verification meets cross-language linking via data abstraction. In *Proceedings of the 2014 Annual ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 675–690, Portland, OR, Oct. 2014.

[22] J. R. Wilcox, D. Woos, P. Panchekha, Z. Tatlock, X. Wang, M. D. Ernst, and T. Anderson. Verdi: A framework for implementing and formally verifying distributed systems. In *Proceedings of the 2015 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 357–368, Portland, OR, June 2015.