# Extracting and Optimizing low-level bytecode from High-level verified Coq

by

Eleftherios Ioannidis

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of
Masters of Engineering in Computer Science and Engineering
at the
MASSACHUSETTS INSTITUTE OF TECHNOLOGY

February 2019

© Eleftherios Ioannidis, MMXIX. All rights reserved.

The author hereby grants to MIT permission to reproduce and to distribute publicly paper and electronic copies of this thesis document in whole or in part in any medium now known or hereafter created.

Author . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
Department of Electrical Engineering and Computer Science
February 1, 2019

Certified by . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
Frans Kaashoek
Professor of EECS
Thesis Supervisor

Certified by . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
Nickolai Zeldovich
Professor of EECS
Thesis Supervisor

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

# Extracting and Optimizing low-level bytecode from High-level verified Coq

by

Eleftherios Ioannidis

Submitted to the Department of Electrical Engineering and Computer Science
on February 1, 2019, in partial fulfillment of the
requirements for the degree of
Masters of Engineering in Computer Science and Engineering


## Abstract

This document is an MEng thesis presenting MCQC, a compiler for extracting verified systems programs to low-level assembly, with no Runtime or Garbage Collection requirements and an emphasis on performance. MCQC targets the Gallina functional language used in the Coq proof assistant. MCQC translates pure and recursive functions into C++17, while compiling monadic effectful functions to imperative C++ system calls. With a series of memory and performance optimizations, MCQC combines verifiability with memory and runtime performance. By handling effectful and pure functions MCQC can generate executable code directly from Gallina and link it with trusted code, reducing the effort of implementing and executing verified systems.


Thesis Supervisor: Frans Kaashoek
Title: Professor of EECS

Thesis Supervisor: Nickolai Zeldovich
Title: Professor of EECS

# Acknowledgments

I would like to extend my thanks to professors Frans Kaashoek, Nickolai Zeldovich and Adam Chlipala for their guidance, Tej Chajed for constructive feedback and MIT's CSAIL for bringing everyone together.

# Contents


**1 Introduction** **13**

1.1 Code generation . . . . . . . . . . . . . . . . . . . . . . . . . . . . 13

1.2 Existing approaches to code generation . . . . . . . . . . . . . . . . 15

1.2.1 Verified Compilation . . . . . . . . . . . . . . . . . . . . . . 15

1.2.2 Extraction . . . . . . . . . . . . . . . . . . . . . . . . . . . 16

1.3 Problem with extraction . . . . . . . . . . . . . . . . . . . . . . . 16

1.4 Previous work . . . . . . . . . . . . . . . . . . . . . . . . . . . . 17

1.5 Approach . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 17

1.6 Contributions of this thesis . . . . . . . . . . . . . . . . . . . . . 18

**2 Design** **21**

2.1 Compilation stages . . . . . . . . . . . . . . . . . . . . . . . . . 21

2.1.1 Parse JSON . . . . . . . . . . . . . . . . . . . . . . . . . . 22

2.1.2 Code Generation . . . . . . . . . . . . . . . . . . . . . . . 24

2.1.3 Algebraic Data types . . . . . . . . . . . . . . . . . . . . . 24

2.1.4 Base typess . . . . . . . . . . . . . . . . . . . . . . . . . . 26

2.1.5 Monadic effects (`Proc`) . . . . . . . . . . . . . . . . . . . . 29

2.1.6 Generate Type context . . . . . . . . . . . . . . . . . . . . . 29

2.1.7 Type unification . . . . . . . . . . . . . . . . . . . . . . . . 31

2.1.8 Generate `main` . . . . . . . . . . . . . . . . . . . . . . . . 33

2.1.9 Pretty-print C++17 . . . . . . . . . . . . . . . . . . . . . . 33

2.2 Garbage collection . . . . . . . . . . . . . . . . . . . . . . . . . 34

2.3 Algebraic datatype compile-time reflection . . . . . . . . . . . . . . 35

2.4 Tail-Call Optimization . . . . . . . . . . . . . . . . . . . . . . . . 35
2.5 Currying . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 36

**3 Implementation and evaluation** **37**
3.1 Implementation metrics . . . . . . . . . . . . . . . . . . . . . . . 37
3.2 Zoobar web server . . . . . . . . . . . . . . . . . . . . . . . . . . 38
3.2.1 Server . . . . . . . . . . . . . . . . . . . . . . . . . . . . 38
3.2.2 Client . . . . . . . . . . . . . . . . . . . . . . . . . . . . 39
3.2.3 Linking verified applications . . . . . . . . . . . . . . . . 40
3.3 Performance Evaluation . . . . . . . . . . . . . . . . . . . . . . . 41

**4 Conclusion** **45**

**Appendices** **47**

# List of Figures


1-1 Peano `nat` definition used in Coq . . . . . . . . . . . . . . . . . . . . . 14

1-2 Inductive `string` definition used in Coq . . . . . . . . . . . . . . . . . 15

2-1 MCQC block diagram. A Coq file is the input, then MCQC generates C++17 code, Clang compiles it and links with the base type library. The white box is the input Gallina program, green boxes show imported libraries and yellow boxes show auto-generated C++17 and executables. 22

2-2 MCQC input Gallina grammar. . . . . . . . . . . . . . . . . . . . . . . 23

2-3 Compiling the `fibonacci` function on the left in C++17, on the right. The shaded box surrounds Coq and C++17 boilerplate code for natural numbers. The definitions are almost isomorphic, except for overflow exceptions in native types which are safely detected and propagated to the caller. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 25

2-4 The `app` function that appends two lists. MCQC extracts the list definition from the Coq standard library and translates it to the pointered data structure on the right, not unlike the linked-list implementation in the C++ standard library. . . . . . . . . . . . . . . . . . . . . . . 27

2-5 The `cat` unix utility that reads and displays a text file. The shaded box shows Coq and C++17 boilerplate code, the `proc` monad for `IO` operations. Instances of `proc` are translated to imperative C++ system calls, as shown at the bottom-right C++ `cat` definition. . . . . . . . . 30

2-6 The high-order function `map` applies a function `f:A->B` to every element of a list `list A`, returning a `list B`. Clang requires some help for the type of F, by hinting on its return type `B` with `std::invoke_result_t`. 33
2-7 MCQC output C++17 grammar. . . . . . . . . . . . . . . . . . . . . 34

3-1 Lines of code count for MCQC . . . . . . . . . . . . . . . . . . . . . 37
3-2 A block diagram of the zoobar web application compiled to run in the browser with MCQC, Go and Webassembly. The gray box shows code executing inside the Coq proof-assistant, dotted arrows indicate compilation steps and solid arrows show a request/response channel. . 39
3-3 Zoobar web application for verified payments . . . . . . . . . . . . . . 40
3-4 Performance and memory benchmarks for four Coq programs, compiled with MCQC versus GHC. Increasing values for $N$ were used for calculating Fig. 3-4a and only the highest value $N$ was used for memory benchmarks in Fig. 3-4b. The corresponding Coq code is in Appendix A. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 42
3-5 Comparative memory profiling of four programs with MCQC and Haskell, over time. The left column is MCQC, clang-7.0 with debug symbols, valgrind and `massif`. The right column is GHC with profiling enabled. 43

A.1 Function that computes the n-th Fibonacci number, uses `Nat` from Coq.Init. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 48
A.2 Factorial function, an example of a tail-recursive function. Uses `Nat` from Coq.Init . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 48
A.3 List reverse function, not tail-recursive and requires quadratic memory allocations. Uses `List` from Coq.Lists . . . . . . . . . . . . . . . . . 48
A.4 Mergesort with merge stacks from Coq.Sorting.Mergesort. Uses `List` from Coq.Lists. . . . . . . . . . . . . . . . . . . . . . . . . . . . . 49

# List of Tables

# Chapter 1

# Introduction

Computer systems are becoming more complex every day. To ensure their correctness, the programmer must verify all execution paths return the expected result. In most cases this is impossible due to path explosion; paths increase exponentially with the size of the program. Unit tests can verify some inputs are mapped to the expected outputs, but give no guarantees about the program behavior on untested inputs.

Formal verification addresses the testing problem in a systematic way. Instead of generating inputs and mapping them against expected outputs, a proof evaluates the formal semantics of the language to show the implementation matches a specification. Formal proofs about programs are developed in a dependently-typed language [24], inside a mechanized proof-assistant, like Coq [24] [2]. Coq is a proof-assistant for writing and formally proving the correctness of programs written in Coq's programming language, Gallina. The execution of formally verified programs written in Gallina is the focus of this thesis.

## 1.1 Code generation

Gallina is a functional language with dependent types [24] that executes inside the Coq proof-assistant. It is the implementation language of Coq, which is to say it describes programs, not proofs. There is a separate language called Ltac for writing proofs in Coq, which is less relevant for this thesis. The functional nature of Gallina

```
Inductive nat : Set :=
| O : nat
| S : nat -> nat.
```

Figure 1-1: Peano `nat` definition used in Coq

makes it cumbersome to run and execute outside Coq. To understand why, one must first understand how base datatypes such as numbers and strings are represented algebraically in Gallina.

Coq formalizes natural numbers using Peano arithmetic. There are two `Nat` constructors as seen in Fig 1-1; the `O` constructor instantiates a `nat` with a numerical value of 0, while the `S` constructor defines a *successor* to any given `nat`. For example, `S O` $= 1$, `S (S (S O))` $= 3$ etc. The set of all natural numbers is inductively defined this way.

Low-level programming languages such as C and assembly represent natural numbers as bitfields that fit into CPU registers. Bitfield arithmetic is not isomorphic to the Peano arithmetic in Coq, as bitfields can overflow. For example in a system with a word size of 32-bits, assigning a value of $2^{32}$ to an unsigned int will overflow. In Peano arithmetic, $2^{32}$ is no different than any other number represented as $2^{32}$ applications of the successor constructor to a to a single zero constructor. Computers have finite memory though, so at some point the number of successor applications will overcome the total RAM of the computer running Coq and the program will be killed. Peano nats in Coq are not infinite in practice, and the observed behavior when running out of memory is a segmentation error.

Coq can prove propositions about natural numbers without the need to enumerate them. The key idea is mathematical induction: proofs by induction on naturals prove the base case and inductive case given an inductive hypothesis, which leads to a proof for all naturals.

Similarly, Coq defines strings inductively as seen in Fig. 1-2. A string can either be empty, or a single character prefix in front of a given string. A faithful execution of the Gallina inductive semantics creates a prohibitive performance overhead, due to the amount of memory and pointer references needed to do even the most basic

```
Inductive string : Set :=
| EmptyString : string
| String : ascii -> string -> string.
```

Figure 1-2: Inductive `string` definition used in Coq

computations. At the same time, subtracting two natural numbers or splitting a string will result in unreferenced memory regions which Coq reclaims by Garbage Collection (GC) in Gallina.

## 1.2 Existing approaches to code generation

Running Gallina programs can be cumbersome and can have a significant performance overhead, while also depending on a big Runtime System (RTS) for GC, higher-order functions, lazy evaluation and more. There are two approaches to generating formally verified, executable code: by verified compilation of deep embeddings and by extraction of shallow embeddings [1][14].

### 1.2.1 Verified Compilation

Verified compilation is the state-of-the-art of getting executable low-level code, from verified high-level code. It is a long and involved process requiring knowledge of advanced programming language theory and proofs in every compilation stage. The high-level overview is this; The programmer defines A high-level eDSL with formal semantics inside Coq. Then she writes the target program in this high-level eDSL and proves it correct against a functional specification written in Gallina. Then, a low-level programming language like C, assembly or LLVM must be formally defined inside Coq. An equivallence relation between the high-level and low-level implementations, or alternatively, a series of verified compilation passes will generate the low-level representation from the high-level representation in a proven way. In the end, the low-level implementation can be executed outside Coq and a trail of proofs connects it to the original high-level implementation and correctness specification. It is our goal to make formal verification accessible to a wider crowd with little background

in programming language theory, which is why we'll be focusing on the alternative method.

### 1.2.2 Extraction

Coq offers extraction plugins that take a Gallina program and extract it to Haskell, OCaml or an Abstract Syntax Tree (AST) serialized to a JSON format [14]. The process is straightforward for Haskell and OCaml, as their expressive type systems can implement the Gallina language directly, with algebraic datatypes, pattern matching and GC provided by their respective RTS. GHC and OCaml then compile the extracted program [9] [12] to generate an executable. The Coq extraction plugins offer an option to syntactically replace a Coq type with a native bitfield type for better performance, ie: `Nat` substituted with a Haskell `Int` is a feature we used for generating pragmatic benchmarks.

## 1.3 Problem with extraction

Standard Coq extraction mechanisms were not built with embeddability or performance in mind. Coq extracts verified code to OCaml and Haskell, both of which require a Runtime System (RTS) to execute programs. Running extracted code directly onto hardware with no high-level memory abstractions is hard, as it requires porting the language's RTS first, then running the extracted programs on top of it. In situations with low memory requirements, this task may be impossible.

Additionally, the performance of algorithms operating on dynamic memory datastructures, such as lists, maps and trees, suffers in extracted code. Gallina passes arguments to functions by value, which leads to excessive copying and a dependence on GC. This is a compromise that functional languages make in order to gain immutability.

Another caveat is that Coq extraction plugins are part of the unverified part of Coq and have to be trusted as Trusted Code Base (TCB). Having a large TCB can defeat the purpose of Formal Verification as bugs can creep in. For extraction to

Haskell for example, not only is the Coq Haskell extraction plugin part of the TCB but also the GHC compiler and the GHC RTS [9].

Finally, Gallina is pure in its implementation, so it cannot generate any observable side-effects. Simple `IO` operations that are straightforward in other languages are not supported at all in Coq. Prior to extraction, the programmer must model side-effects using monadic composition and interpret the monads post-extraction using the `IO` mechanism of the target language.

## 1.4 Previous work

The CertiCoq compiler implements Coq's language inside the Coq proof-assistant, allowing for the verified compilation of shallow embeddings. However, CertiCoq still depends on a runtime GC and cannot generate static, stand-alone assembly. The Œuf verified extractor [17] reifies Gallina into an AST that it then translates to CompCert's intermediate representation [13] but does not target the full Gallina, only a small subset of it relevant to reactive systems. The Fiat crypto compiler does verified compilation of an embedded domain-specific language (eDSL) down to static C, but is only applicable to the domain of cryptographic algorithms [7].

## 1.5 Approach

MCQC attempts to solve the extraction problem in a more general way, in order to make Gallina a system's programming language with `IO` capabilities similar to Haskell but without the GHC runtime [9]. MCQC uses C++17 as an intermediate representation, an extensible, compiled language, with strong compile-time facilities. C++17 offers expressive polymorphism through templates, algebraic datatypes through variants and GC through smart pointers. To implement Gallina's strict Hindley-Milner (HM) type system [5] MCQC makes use of Template meta-programming (TMP), a Turing-complete, compile-time language on top of C++ that operates on types.

TMP has been used before to create eDSLs with dependent type-systems [16] and is used in MCQC to implement strict typing, polymorphism and reflection on algebraic types [20][6].

Additionally, MCQC relies on smart pointers for GC. As part of the standard library `std::unique_ptr` and `std::shared_ptr` require no RTS to perform reference counting. MCQC uses `std::shared_ptr` to allocate and deallocate datatype instances, which by default can have multiple owners.

Finally, the introduction of sum types as tagged unions by `std::variant` in C++17 and product types as structs, make the extraction of algebraic data types in C++17 possible. MCQC can generate any C++17 algebraic data type with pattern matching and generic support from their Gallina inductive definition. By using TMP for runtime reflection, MCQC serializes and prints arbitrary algebraic datatypes, a useful feature for inter-process communication and debugging.

## 1.6 Contributions of this thesis

This thesis introduces MCQC, a compiler from Gallina to assembly through C++17 as an intermediate representation, using the Clang compiler as a back-end. MCQC focuses on performance and usability for systems programming, while making extensive use of the compile-time facilities of C++17 and the LLVM optimization passes in Clang. MCQC provides a library of bitfield base types to achieve better performance and uses smart pointers to enable reference-counting in algebraic data structures, without a need for a RTS.

With respect to `IO` operations, MCQC provides an `IO` C++17 library for basic POSIX functionality, modeled in Coq by the `Proc` monad library. The `Proc` monad is a construction for modeling sequential `IO` side-effects, similar to Haskell's `IO` monad. By defining a `main` function of type `Proc unit` in Gallina and extracting with MCQC, we can directly compile Coq programs to an executable with a `main` entry point.

Using MCQC we have successfully written and proved a demo web application for online payments. The web server was written in Gallina and compiled to C++17,

while the client was written in Gallina and compiled to Webassembly through MCQC and emscripten [8]. In both cases, a minimal amount of boilerplate code and proofs was required, while the use of MCQC made it possible to write and execute verified client and server code without leaving the Coq proof-assistant.

MCQC has some limitations compared to Gallina executed in Coq. MCQC cannot generate code for Gallina typeclass instances [22]. Typeclasses offer a model for ad-hoc polmorphism which is more general than C++ templates, In addition, MCQC has limited multi-threading support. As part of `Proc` MCQC implements $spawn : \forall T, (T \rightarrow unit) \rightarrow T \rightarrow proc\ unit$ which can execute closures with no return values in parallel with `std::future`. To support parallel execution with return types, a promises interface would be more effective [15] in the future.

# Chapter 2

# Design

MCQC is written in Haskell and accepts as input Gallina abstract syntax trees (AST) in a JSON format, extracted by the Coq JSON extraction plugin (Coq v.8.5.1). MCQC compiles the Gallina AST to C++17 which then Clang compiles to assembly [11] as shown in Fig. 2-1.

## 2.1 Compilation stages

MCQC breaks the compilation of Coq to C++17 into several stages. For each stage there is a dependency on all previous stages.

1. Parse JSON
2. Load module dependencies
3. Code generation
4. Base types
5. Algebraic types
6. Proc monad
7. Generate type context
8. Substitute native functions

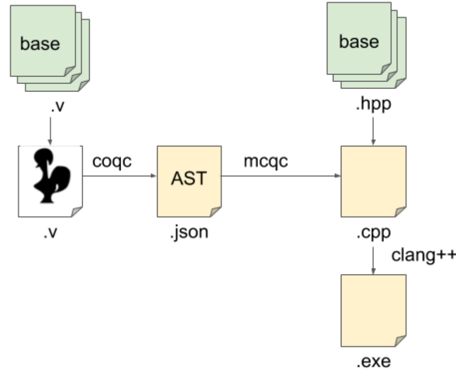



Figure 2-1: MCQC block diagram. A Coq file is the input, then MCQC generates C++17 code, Clang compiles it and links with the base type library. The white box is the input Gallina program, green boxes show imported libraries and yellow boxes show auto-generated C++17 and executables.

9. Type unification

10. Generate `main`

11. Pretty-print C++17

### 2.1.1 Parse JSON

The input JSON is an AST of a Gallina program, described by the grammar in Fig. 2-2. The top level structure is a `Module`, which represents a Coq module. Every module can have dependencies expressed through `used_modules` and multiple `Declarations`. A `Declaration` can be an algebraic type represented by `IndDecl`, a type alias `TypeDecl`, a Fixpoint function `FixDecl` or a non-fixpoint named expression `TermDecl` also exported as a function.

Expressions (`Expr`) are the computational part of Gallina and they represent in the order which they appear in Fig. 2-2: Lambda expressions, pattern matching, constructors, function applications, `let` statements, safe coercions, relative and global references and dummy expressions.

The first step of loading module dependencies is reading used modules from disk

```
-- Top level Coq module
data Module = Module { name :: Text, used_modules :: [Text], declarations :: [Declaration] }

-- High level declarations
data Declaration =
    IndDecl { name :: Text, iargs :: [Text], constructors :: [Expr] }
    | TypeDecl { name :: Text, targs :: [Text], tval :: Type }
    | FixDecl { fixlist :: [Fix] }
    | TermDecl { name :: Text, typ :: Type, val :: Expr }

-- Patterns
data Pattern = PatCtor { name :: Text,  argnames :: [Text] }
    | PatTuple { items :: [Pattern] }
    | PatRel  { name :: Text }
    | PatWild {}

-- Pattern match case
data Case = Case { pat :: Pattern, body :: Expr}

-- Fixpoint declaration
data Fix = Fix { name :: Maybe Text, ftype :: Type, value :: Expr }

-- Types
data Type =
    TypeArrow { left :: Type, right :: Type }
    | TypeVar { name :: Text, args :: [Expr] }
    | TypeGlob { name :: Text, targs :: [Type] }
    | TypeVaridx { idx :: Int }
    | TypeUnknown {}
    | TypeDummy {}

-- Expressions
data Expr = ExprLambda { argnames :: [Text], body :: Expr }
          | ExprCase { expr :: Expr, cases :: [Case] }
          | IndConstructor { name :: Text, argtypes :: [Type] }
          | ExprConstructor { name :: Text, args :: [Expr] }
          | ExprApply { func :: Expr , args :: [Expr]}
          | ExprLet { name :: Text, nameval :: Expr, body :: Expr }
          | ExprCoerce { value :: Expr }
          | ExprRel { name :: Text }
          | ExprGlobal { name :: Text }
          | ExprDummy {}
```

Figure 2-2: MCQC input Gallina grammar.

into multiple `Module` objects. Those modules are then concatenated to make a monolithic, stand-alone module with no dependencies. MCQC outputs a single C++17 file, which helps avoid incompatibilities between the Coq and C++ module systems and simplifies dependency resolution.

### 2.1.2 Code Generation

This step does most of the initial fitting of Coq into the C++17 language standard. All safe coercions, relative and global references elaborate to global references, since all modules are parts of the same lexical scope. Pattern matching in Coq is translated to the polymorphic C++17 function `match`, seen in Fig. 2-3 and Fig. 2-4. In MCQC `match`, `show` and `proc` are reserved words.

Another transformation that takes place early is `ExprLet` to statement inlining. Instead of substituting with a lambda as is common in functional compilers, MCQC takes advantage of C++17 sequential semantics by making it an assignment statement in the function body. The type of this statement cannot be inferred by MCQC right now so it will be set to `auto`, a C++17 type keyword that delegates type resolution to Clang. MCQC handles type resolution in a more general way during the *Type unification* phase during which it can explicitly type `auto` types to help Clang type inference.

### 2.1.3 Algebraic Data types

MCQC transforms algebraic type definitions in Coq to an `std::variant` in C++17. Sum types via variants is a new addition to C++17 that MCQC relies on. A variant is a tagged union between multiple types [4], only one of which inhabits it at a time. To keep track of the variant inhabitant, `std::variant` holds an enum which acts as the tag. Product types exist in C as structs. C structs represent the cartesian product between types and can be inhabited by all types at the same time. Sums and products constitute the entirety of Coq algebraic datatypes and their combination allows MCQC to define any algebraic data type in C++17 as a sum-of-products.

```coq
Inductive nat : Set :=
| O : nat
| S : nat -> nat.
```

```cpp
// Nat type alias for bitvector type
using nat = unsigned int;

// Pattern matching on nat
template<typename FO, typename FS,
  typename = enable_if_t<CallableWith<FO>>,
  typename = enable_if_t<CallableWith<FS, nat>>>
constexpr auto match(nat a, FO f, FS g) {
  switch(a) {
  case 0:  return f();    // Call O match clause
  default: return g(a-1); // Call S match clause
  }
}
```

```coq
Fixpoint fib(n: nat) :=
  match n with
    | O => 1
    | S sm =>
      match sm with
        | O => 1
        | S m =>
          (fib m) + (fib sm)
      end
  end.
```

```cpp
nat fib(nat n) {
  return match(n,
    [=]() { return 1; },
    [=](nat sm) { return match(sm,
      [=]() { return 1; },
      [=](nat m) {
        return add(fib(m), fib(sm));
      });
    });
}
```

Figure 2-3: Compiling the `fibonacci` function on the left in C++17, on the right. The shaded box surrounds Coq and C++17 boilerplate code for natural numbers. The definitions are almost isomorphic, except for overflow exceptions in native types which are safely detected and propagated to the caller.

An example of a list definition can be seen in Fig. 2-4. Algebraic type declarations are the only case where MCQC generates multiple C++ declarations for one Coq declaration. MCQC generates a struct declaration for each constructor, a type alias declaration for the variant, the constructor function declarations that create variant inhabitants and finally a polymorphic `match` function for deconstructing the variant.

As constructors allocate memory for the new object and `match` takes objects apart into their constructors, possibly deallocating memory, some memory management is necessary. Constructors and `match` declarations wrap their return values and arguments in a `std::shared_ptr` which handles memory allocation and deallocation implicitly, by by keeping track of the object scope.

MCQC implements pattern matching with the parametric, variadic function `gmatch` seen in Fig. 2-4. The generic match `gmatch` function accepts an arbitrary algebraic object and a series of lamdas, one lambda for each constructor. Then, `gmatch` calls the standard library function `std::visit` which will query the variant tag and apply

the appropriate lambda to it depending on the inhabitant's type and the lambda type signature. Since lambdas in C++17 are always `constexpr`, Clang will inline them in the body of the caller when the variant tag is known at compile time, creating a zero-cost pattern-matching abstraction.

### 2.1.4 Base typess

Using Coq's library of base types, such as numbers, strings and booleans can have a significant performance impact, making MCQC unusable in practice. Coq defines every base type algebraically and even the simplest bitwise and arithmetic computations must be defined recursively. MCQC substitutes the slow Coq base types with a library of fast, safe, native types written in C++17. MCQC base types are implemented as `unsigned int`, `char`, `bool` etc. in C++, while operations such as addition and multiplication safely detect overflows and throw exceptions at runtime. To show the native base type library obeys the Gallina semantics, MCQC tests base types against a specification by using property-based tests.

Base types are always passed by value. Conversely, non-base algebraic datatypes that are auto-generated are always passed by smart pointer.

#### Base type library

MCQC has native support for the following base types in C++17:

1. Nats
2. Chars
3. Bools
4. Strings
5. Pairs
6. Options

```cpp
// Overload functions to a single overloaded definition
template<class... Ts>
struct overloaded : Ts... {
    using Ts::operator()...;
};
template<class... Ts>
overloaded(Ts...) -> overloaded<Ts...>;

// Generic polymorphic match
template<typename T, typename ...Args>
auto gmatch(std::shared_ptr<T> m, Args... args){
    return std::visit(overloaded { args... }, *m);
}
```

```coq
Inductive list (T:Type) : Type :=
  | nil : list T
  | cons : T -> list T -> list T.

Fixpoint app {T} (l m : list T) :
    (list T) :=
  match l with
  | nil => m
  | cons h t => cons h (app t m)
  end.
```

```cpp
template<class T>
struct Coq_nil {};
template<class T>
struct Coq_cons {
  T a;
  std::shared_ptr<std::variant<
      Coq_nil<T>,
      Coq_cons<T>>> b;
  Coq_cons(T a,
      std::shared_ptr<std::variant<
        Coq_nil<T>,
        Coq_cons<T>>> b) {
    this->a = a;
    this->b = b;
  };
};

template<class T>
using list = std::variant<Coq_nil<T>, Coq_cons<T>>;

template<class T>
std::shared_ptr<list<T>> coq_nil() {
  return std::make_shared<list<T>>(Coq_nil<T>());
}

template<class T>
std::shared_ptr<list<T>> coq_cons(T a,
    std::shared_ptr<list<T>> b) {
  return std::make_shared<list<T>>(Coq_cons<T>(a, b));
}

template<class T, class U, class V>
auto match(std::shared_ptr<list<T>> self, U f, V g) {
  return gmatch(self,
    [=](Coq_nil<T> _) { return f(); },
    [=](Coq_cons<T> _) { return g(_.a, _.b); });
}

template<class T>
std::shared_ptr<list<T>> app(std::shared_ptr<list<T>> l,
    std::shared_ptr<list<T>> m) {
  return match(l,
    [=]() { return m; },
    [=](auto a, auto l1) {
      return coq_cons<T>(a, app<T>(l1, m));
    });
}
```

Figure 2-4: The app function that appends two lists. MCQC extracts the list definition from the Coq standard library and translates it to the pointered data structure on the right, not unlike the linked-list implementation in the C++ standard library.

MCQC represents the shared interface between base types in Coq and C++17 with Coq typeclasses. Let's take the interface for natural numbers, defined in the `MNat.v` library as an example. `MNat.v` contains the `NativeNat` typeclass for which Coq's standard `Nat` is an instance. `NativeNat` declares functions like `add`, `sub`, `mul` etc. which MCQC implements with native bitwise arithmetic in `nat.hpp`.

MCQC has compile-time semantics for all base types and does constant propagation and compile time evaluation of expressions. An important note is that MCQC implements semantics for base types twice; once in Haskell for MCQC compile-time evaluation and once in C++17 for runtime execution. The MCQC compile time semantics are not necessary strictly speaking, as syntactically replacing Coq constructors with their C++17 function definitions is enough for Clang to do constant propagation. However, constant propagation helps with the readability of the produced C++17 code, as for example the number 5 is easier to read than its Coq representation; `S (S (S (S (S 0))))`.

### Base type soundness

Because the soundness of the MCQC native base type library is not verified, some reassurances are necessary. MCQC does property-based testing of native base types in order to confirm they have the same properties as their Gallina counterparts. Rapidcheck, a property-based testing library and clone of quickcheck [3] autogenerates tests for the native base types based on a specification. The same specification is shared between Coq and Rapidcheck tests and ensures the consistency of the shared interface. While property-based tests are not exhaustive, they are randomly generated and thus are more general in testing program behavior than unit tests.

### Weak to strict typing

Weak typing is the default behavior in C++17, which means arbitrary types are accepted as function arguments as long as they fit in the allocated stack frame. The common example of weak typing is the implicit cast of `char` to `int`, when passed to functions accepting `int` arguments. Weak typing can cause undefined behavior

when haphazardly calling into extracted C++17 programs. MCQC strengthens the C++17 type system by using TMP to enforce strict typing.

Programs extracted with MCQC enforce a HM type-system similar to the one used in OCaml[5]. An example of strict types enforced in C++17 by TMP is in Fig. 2-3 in the definition of `match`. If template substitution fails at `std::enable_if_t`, the function will quietly disappear at Clang compile-time, a pattern known as SFINAE in the C++ world (Substituition Failure Is Not An Error) [23]. Failed template substitution does not throw an error but hides the templated C++ function completely. If Clang finds no other compatible definition with that name, an undefined reference error will be thrown.

### 2.1.5 Monadic effects (`Proc`)

Coq has no way of interacting with the underlying OS in an effectful way. MCQC offers an interface for effectful computations by means of monadic composition. Effectful monads in Gallina elaborate to imperative-style C++ code as shown in Fig. 2-5. The elaboration to imperative style commands happens at MCQC compile time, by evaluation of the monadic laws for `Proc`. This behavior is special to the `Proc` monad, defined in `MProc.v` and implemented in `proc.hpp` in C++17. An example of deriving an imperative implementation for the `cat` utility, that reads a file from disk and prints it, is shown in Fig. 2-5.

### 2.1.6 Generate Type context

Code extracted from Coq uses the HM type system [5]. While Clang can infer some `auto` types it is not capable of full HM type inference and requires additional template annotations. To that end, MCQC implements full HM type inference [5] in order to fill in missing C++ template arguments. MCQC builds the type context needed for HM in two steps; it adds all base types and function signatures from the base types library first, then it recursively adds all function signatures and types from all input

```
(** Filedescriptor type *)
Definition fd := nat.

(** Effect composition *)
Inductive proc: Type -> Type :=
| open : string -> proc fd
| read: fd -> proc string
| close : fd -> proc unit
| print : string -> proc unit
(** Monad *)
| ret: forall T, T -> proc T
| bind: forall T T',
  proc T
  -> (T -> proc T')
  -> proc T'.

Notation "p1 >>= p2" :=
  (bind p1 p2).
```

```
// Filedescriptor type
using fd = nat;

// open file
static proc<fd> open(string s) {
  if (int o = sys::open(FWD(s).c_str(), O_RDWR) {
    return static_cast<fd>(o);
  }
  throw IOException("File not found");
}
// read file
static proc<string> read(fd f, nat size) {
  auto dp = string(size, '\0' );
  sys::read(f, &(dp[0]), sizeof(char)*size);
  return dp;
}
// close file
static proc<void> close(fd f) {
  if(sys::close(f)) {
    throw IOException("Could not close file");
  }
}
```

```
Definition cat (path fn: string):=
  open (path ++ "/" ++ fn) >>=
    (fun f => read f >>=
      (fun data => close f >>=
        (fun _ => print data >>=
          (fun _ => ret unit)))).
```

```
proc<void> cat(string path, string fn) {
  fd f = open(append(path, append("/", fn)));
  string data = read(f);
  close(f);
  print(data);
}
```

Figure 2-5: The `cat` unix utility that reads and displays a text file. The shaded box shows Coq and C++17 boilerplate code, the `proc` monad for `IO` operations. Instances of `proc` are translated to imperative C++ system calls, as shown at the bottom-right C++ `cat` definition.

modules.

Substituting native functions in the place of Coq functions is straightforward. As long as the function name and type signatures match, MCQC replaces every Coq function call with a call to its static C++17 implementation. For example, the binary `add` function that accepts two `nat` arguments and returns a `nat` is defined recursively in Coq but with native arithmetic in C++17. MCQC will pick the C++17 version of `add` by linking the `nat.hpp` native library.

### 2.1.7 Type unification

While every declaration has a type, that's not the case for every expression. Looking at Fig. 2-4 in the definition of the `app` function that appends two lists, it is clear the type signature of `app` is $\forall\ T,\ list\ T \rightarrow list\ T \rightarrow list\ T$ but there is some work involved to go from the type signature to C++ template arguments for `coq_cons<T>` and `app<T>`. There are two steps to type unification, type inference and template resolution.

#### Type inference

In a HM type system there are base types like `nat` and `bool`, type expressions like `list<nat>`, function types like `nat` → `nat` → `nat` and free types like the $T$ in $\forall\ T,\ list\ T$. We extend this type system for C++17 with a pointer type which MCQC outputs as `std::shared_ptr` and an `auto` type, for which inference will be delegated to Clang. MCQC represents free types with De Bruijn indices [10] which translate to C++ templates during the pretty-printing stage.

Type inference begins at the type declaration for each function, which is always guaranteed to be well-typed from Coq. First, MCQC unifies the return type of a function with the expression on the right-hand side of its `return` call. Every free type that can be filled in on the right-hand side will be filled in at this point. Then, each argument type in the function declaration is added to the local context as a constraint. Those constraints are solved while traversing the AST of the function

body and the resulting unified types are substituted in-place of `auto` in the AST. Type inference in MCQC annotates every expression in the program with a type and fills-in free types with template arguments.

### Template resolution

Having a type for every expression is helpful, but the absolute type is not what Clang expects. It expects a template argument relative to the function or type declaration. For example in Fig. 2-4 in the `app` body, `app` is recursively called as `app<T>` and not `app<std::shared_ptr<list<T»` which is the absolute type of the expression. In this example, MCQC plugged-in `T` during template resolution which is what Clang expects.

Template resolution uses the Type context. MCQC matches every expression's type against the absolute type in the context, then minimizes the two types to their common type subexpressions, until the two become different. What is left is the template argument plug. This is how for example, MCQC minimizes a type `std::shared_ptr<list<T»` from the context and inferred `std::shared_ptr<list<nat»` to `nat`, the template argument Clang expects.

The C++ type system does not have support for function types. At the same time, function types are vital in an HM type system. Using the `std::function` class from the C++17 standard library is a poor choice, as Clang cannot introspect or inline `std::function` objects at compile time. Templates come to the rescue; Clang will introspect functional arguments when they are passed as templates, inline them and optimize them, contrary to `std::function` objects which are opaque.

The transformation of function types to templates is lossy, as it is impossible to go back and find the function return and argument types afterwards. In the case of high-order functions like `map` in Fig. 2-6, overgeneralizing the argument function type will certainly result in the Clang type-checker failing to resolve it. To help the Clang type checker while not losing the benefits of inlining for functional arguments, the `std::invoke_result_t` template function shown in Fig. 2-6 indicates that `R` is the result type of applying function `V` to an argument of type `T`. Now the Clang type

```
Fixpoint map (f: A->B) (l: list A):
    list B :=
match l with
  | [] => []
  | a :: t => (f a) :: (map f t)
end.
```

```
template<class A, class F,
         class B = std::invoke_result_t<V,T>>
std::shared_ptr<list<B>> map(F f,
    std::shared_ptr<list<A>> l) {
  return match(l,
    [=]() { return coq_nil<B>(); },
    [=](auto a, auto t) {
      return coq_cons<B>(f(a), map<A>(f, t));
    });
}
```

Figure 2-6: The high-order function `map` applies a function `f:A->B` to every element of a list `list A`, returning a `list B`. Clang requires some help for the type of F, by hinting on its return type `B` with `std::invoke_result_t`.

checker knows about the type of function `f` from its relation to `T` and `R` and can successfully compile the code in Fig. 2-6.

### 2.1.8 Generate `main`

The `main` function expected by MCQC's `Proc` monad in Coq and the `main` function expected by Clang are typed differently typed in each language. In Coq, a main function cannot take or return anything, so its type is `proc unit`. C++17 inherits the rules for `main` from C and it must return `int` which becomes the return value for the whole process. For MCQC, command line arguments to main passed by `argc` and `argv` are not supported and `main` always returns 0.

The use of `proc unit` is equivalent to Haskell's `main` definition as `IO ()`, as it interacts with the underlying system by using the `IO` monad and returns nothing. Type substitution for `main` is the last pass that occurs before pretty-printing the C++17 AST to C++17 code.

### 2.1.9 Pretty-print C++17

In order to pretty-print C++17, MCQC transformed the input Coq grammar in Fig. 2-2 to an intermediate representation more close to C++17. MCQC uses the intermediate grammar in Fig. 2-7 for all optimization passes. Now going from this grammar to a `.cpp` file is a matter of implementing a Wadler/Leijen based pret-

```haskell
-- C++ typed name
data CDef = CDef { _nm :: Text, _ty :: CType }

-- C++ file lexical scope
data CFile = CFile { _includes :: [Text], _decl :: CDecl }

-- Global scope C++ definitions,
data CDecl =
    CDFunc     { _fd :: CDef, _fargs :: [CDef], _fbody :: CExpr }
    | CDType   { _td :: CDef }
    | CDStruct { _sn :: Text, _fields :: [CDef], _nfree :: Int }
    | CDSeq    { _left :: CDecl, _right :: CDecl }
    | CDEmpty  {}

-- C++ Types
data CType =
    CTFunc    { _fret :: CType, _fins :: [CType] }
    | CTExpr  { _tbase :: Text, _tins :: [CType] }
    | CTVar   { _vname :: Text, _vargs :: [CExpr] }
    | CTBase  { _base :: Text }
    | CTPtr   { _inner :: CType }
    | CTFree  { _idx :: Int }
    | CTAuto  {}

-- C++ Expressions
data CExpr =
          -- High level C++ expressions
            CExprLambda { _lds :: [CDef], _lbody :: CExpr }
          | CExprCall   { _cd :: CDef, _cparams :: [CExpr] }
          -- Continuations
          | CExprSeq    { _left :: CExpr, _right :: CExpr }
          -- C++ statament
          | CExprStmt   { _sd :: CDef, _sbody :: CExpr }
          -- Reduced forms
          | CExprVar    { _var :: Text }
          | CExprStr    { _str :: Text }
          | CExprNat    { _nat :: Int }
          | CExprBool   { _bool :: Bool }
          | CExprPair   { _fst :: CExpr, _snd :: CExpr }
```

Figure 2-7: MCQC output C++17 grammar.

typrinter [21]. MCQC uses the `prettyprinter` package to implement pretty-printing to C++17.

## 2.2 Garbage collection

A common C++17 technique for the garbage collection of data structures is smart pointers. They eliminate the need for manual memory management by calls to `new` and `delete`. Instead, when a smart pointer goes out of scope, the object it points to loses a reference. When it reaches zero references the smart pointer calls the object's destructor. This matches Coq's memory model, where data structures are passed by

value and Coq deallocates them when they lose all references.

The C++17 `std::shared_ptr` class is used throughout MCQC for reference counting in pointered data structures. Reference ownership can be shared across multiple callers for `std::shared_ptr` which incurs some overhead. This small overhead is a reasonable trade-off for getting a generic dynamic memory allocation interface applicable to all Gallina programs. The reference counter used by `std::shared_ptr` is stored near allocated memory for the object, which allows for better locality and cache performance.

## 2.3 Algebraic datatype compile-time reflection

MCQC can print arbitrary algebraic datatypes by using TMP. An overloaded implementation of `show` for all types implements the following reflection scheme; Clang will check the type of the argument to `show`; if it is a base type it will use one of the predefined `show` functions in the MCQC base type library. Otherwise, it will have to be either a variant or a pointer to a variant. Then `show` will recurse inside variants and pointers, until it finds a struct (product type). Then, it will decompose the product into its parts and recurse on each one. Since all base types have a corresponding `show` function and all other types are pointered algebraic types, it is easy to prove by structural induction that `show` always terminates in a base type, while it works as expected for all algebraic datatypes.

Since this kind of parametric polymorphism cannot be encoded in Coq without a function $\forall T,\ Show:\ T \rightarrow string$, an axiomatic definition of `Show` is included in the `MShow.v` library.

## 2.4 Tail-Call Optimization

The Clang compiler has support for Tail-Call Optimization (TCO) so MCQC does as well. In recursive C++ functions that call themselves as the last step, the stack is reused. TCO is equivalent to unrolling recursive calls to for-loops, and thus generates

faster code. MCQC makes use of TCO when possible in recursive pure functions. Since `match` is a `constexpr` it will be inlined early in the Clang compilation process and if TCO is possible Clang will perform the optimization across the scope of `match` statements.

## 2.5 Currying

A key characteristic of functional languages is partial evaluation, or currying. Partial evaluation of a function with a value binds the value to an argument position and returns an anonymous function with reduced arity. MCQC uses `constexpr` lambdas to compose functions, providing a zero-cost currying abstraction. The number of arguments passed to a function is known at Clang compile time, and `constexpr` will elaborate one lambda for every argument given, up to $n$ arguments. If all $n$ arguments are given, then Clang will evaluate all $n$ `constexpr` lambdas and substitute a value. If $n - 1$ arguments are given, the result will be the last unevaluated lambda, which will take a single argument and return the result.

# Chapter 3

# Implementation and evaluation

In this section we present the runtime properties and performance of programs compiled with MCQC. The three questions we try to answer is; can we link verified and unverified code to create end-to-end applications, can we get better memory performance than extracted Haskell compiled with GHC and can we get comparable runtime performance to GHC.

## 3.1 Implementation metrics

MCQC is open source under an MIT license and can be found here `https://github.com/mit-pdos/mcqc`. MCQC is written in Haskell and comes with the C++17 base type library and the corresponding Coq ADT typeclasses. The lines-of-code count for the various parts of MCQC is shown in Fig. 3-1.

| Language | LoC |
|---|---|
| MCQC in Haskell | 1800 |
| C++17 native types | 630 |
| Coq typeclasses | 200 |

Figure 3-1: Lines of code count for MCQC

## 3.2 Zoobar web server

In order to demonstrate MCQC's capabilities we have developed a demo web application for payments, the verified *Zoobar* server. It is influenced by the zoobar server used in 6.858 Computer Systems Security class in MIT. It serves as a case-study of linking verified transaction logic from Coq, with unverified, trusted HTTP libraries, in order to implement an end-to-end web application. In Fig. 3-2 we can see the block diagram of the zoobar compilation process and in Fig. 3-3 a screenshot of the zoobar server running in the browser.

Linking unverified trusted libraries with verified code allows the MCQC user and proof writer to focus on the parts of an application for which verification is most important. At the same time, MCQC users have the choice of trusting the well-maintained, performant libraries that they use already and can complement them with some verified core logic from Coq, compiled with MCQC.

### 3.2.1 Server

We built the zoobar web server by linking an HTTP RPC server written in Go with the transaction logic written in Gallina and MCQC. The server has no persistent state, all users and associated balances are stored in-memory. The database is a list of pairs, the username and the total zoobars amount of that user. Initially everyone has the same balance, as in Fig. 3-2.

When an HTTP request arrives the proxy server will shed the HTTP metadata and proxy the request to the transaction logic. In the transaction logic, a parser written in Gallina will deserialize and execute the request. A request initiates a transfer between users by subtracting $n$ zoobars from the first user and adding it to the second. If either the source or the target user does not exist, nothing will happen. If the source user has less balance than the transaction needs, nothing will happen. If the source user has sufficient balance, then the amount will de deducted from his balance and added to the target user. This is the core transaction logic for the server

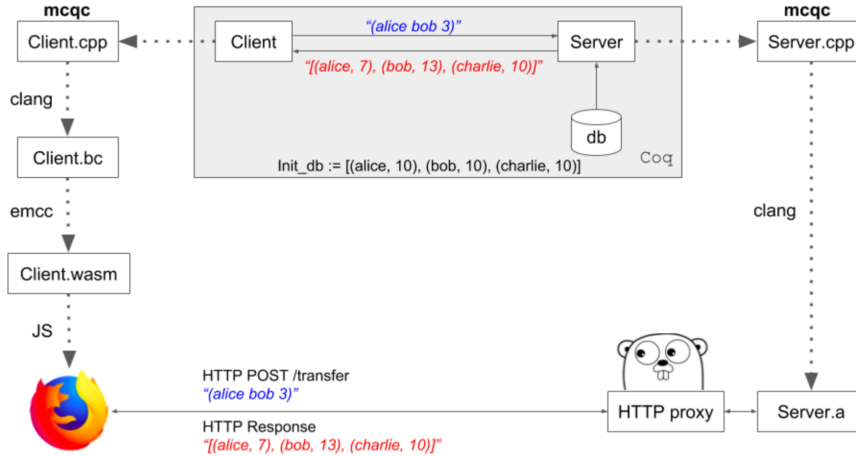



Figure 3-2: A block diagram of the zoobar web application compiled to run in the browser with MCQC, Go and Webassembly. The gray box shows code executing inside the Coq proof-assistant, dotted arrows indicate compilation steps and solid arrows show a request/response channel.

and the following lemma applies:

$$\forall\ DB,\ req:\ let\ DB' := transfer(req, DB)\ in \sum_{u \in users}^{zoobars(u)} DB = \sum_{u \in users}^{zoobars(u)} DB'$$

The lemma states that after any transaction the total zoobars in the system will remain constant. Or phrased another way, no value can be lost or created by making transactions. Coq makes it straightforward to prove the lemma above with respect to the transaction logic, and requires no boilerplate proofs, except for the theorems included in the Coq standard library.

The core logic was compiled with MCQC and linked against a Go RPC web server acting as an HTTP proxy. The total Gallina code and proofs require about 600 lines of Coq and one work-day for the author to write, verify and compile it. Setting up the Go web proxy took another work-day.

### 3.2.2 Client

We also wrote the client part of the Zoobar server in Coq and compiled it with MCQC. The client is straightforward; it accepts two users and an amount of zoobars from the

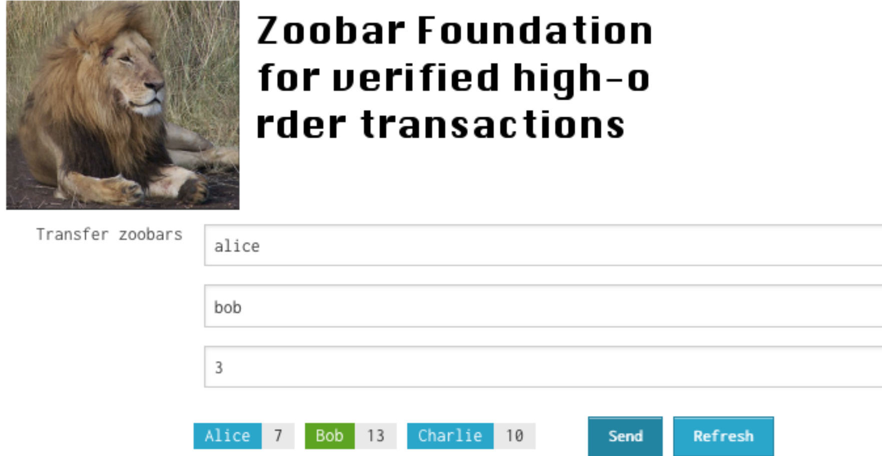



Figure 3-3: Zoobar web application for verified payments

Javascript UI, serializes it into a request and sends it to the server. We compiled the client with MCQC and Emscripten [8] to Webassembly and embedded it into a Javascript HTTP client, completing the client part of the *zoobar* server.

### 3.2.3 Linking verified applications

The zoobar server demonstrates the ease of linking code compiled with MCQC. Both the server and client were built and proven in Coq and extracted to C++17 before linking with the HTTP libraries in Go and Javascript. The proof effort for the transaction logic is minimal and focuses on the code that is most important for the overall correctness. The same code that executes inside Coq in Fig. 3-2 can now execute in the real world, by linking against standard trusted libraries, such as Javascript and the Go HTTP server. This zoobar demo demonstrates a hybrid approach to verification, by combining verified logic with unverified, trusted code and can greatly save development and proof effort for systems programming.

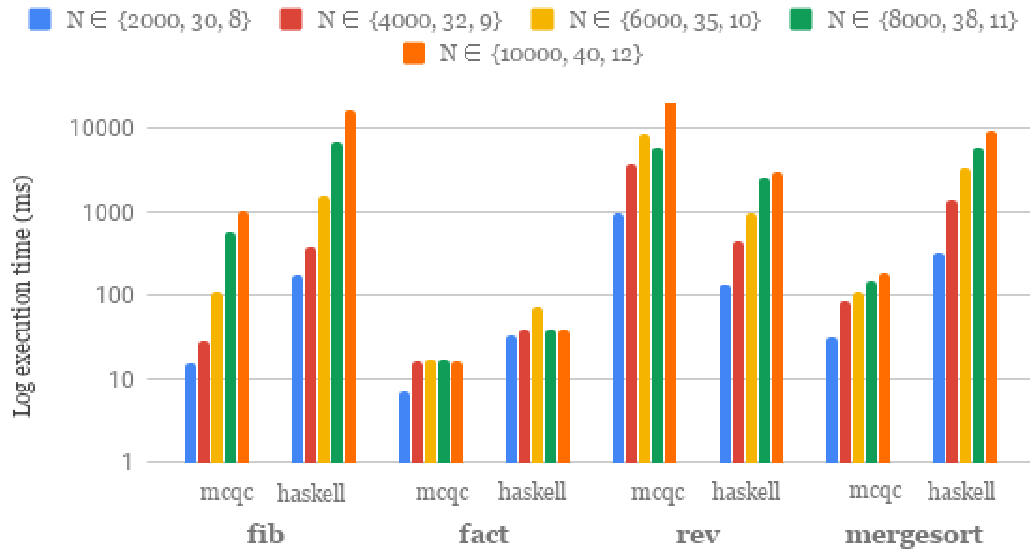
## 3.3 Performance Evaluation

MCQC compares fairly well against GHC in terms of run-time performance and total memory used. The execution time of MCQC programs is on average 14.8% faster than GHC programs, as seen in Fig. 3-4a. MCQC reduces the memory footprint of executing verified programs by 66.25% on average compared to GHC, as seen in Fig. 3-4b. While the reduction in reserved memory was one of the goals of MCQC, achieved by removing the GHC RTS, the reduction in program run-time is encouraging for using Clang as a back-end to functional compilers in the future. Especially on large inputs in Fig. 3-4a MCQC outperforms Haskell every time.

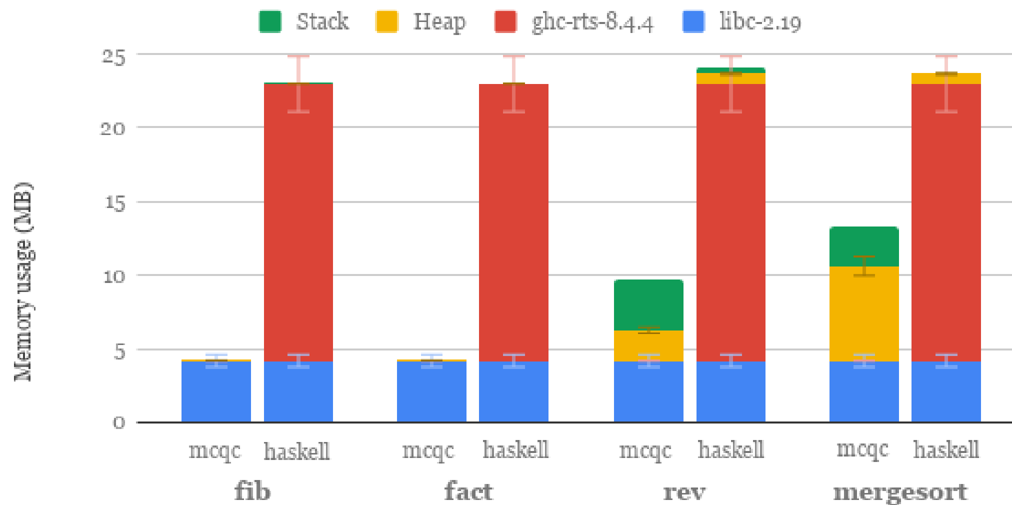
We compare the performance of code generated with MCQC against Haskell extracted from Coq. During Haskell extraction, we tell Coq to substitutes bitvector types (`Int`) for native arithmetic as the performance of peano arithmetic is multiple orders of magnitude slower making the comparison unfair. The Clang-7.0 compiler is used to compile C++17 and `GHC-8.4.4` to compile extracted Haskell. All benchmarks were run on a MacBook Air 2014, 1.4 GHz Intel Core i5, 4 GB 1600 MHz DDR3.

On the C++17 side, `valgrind` and the `massif` tool perform memory profiling by instrumenting all stack and heap memory accesses [18] [19] as seen in Fig. 3-5. On the Haskell side, the GHC profiler performs both stack and heap instrumentation through the GHC RTS. By querying the OS for the number of pages mapped to Haskell executables is how we determined the GHC RTS size to be about *23MB*. MCQC and Haskell share `libc` so we subtract it from the GHC RTS size as seen in Fig. 3-4b.

The results in Fig. 3-4 show MCQC extracted code performs with considerably less memory compared to Haskell and at comparable run-time. For `fib`, using native integers explains the lack of heap memory allocated and a linear rise in the stack due to loading all the `fib` stack frames. For `fact`, we get no heap or stack usage, which indicates TCO has optimized recursion successfully as seen in Fig. 3-4. Finally, in algorithms that rely on GC we show that MCQC uses less memory compared to Haskell and in most cases, MCQC is faster, as seen in Fig. 3-4a.

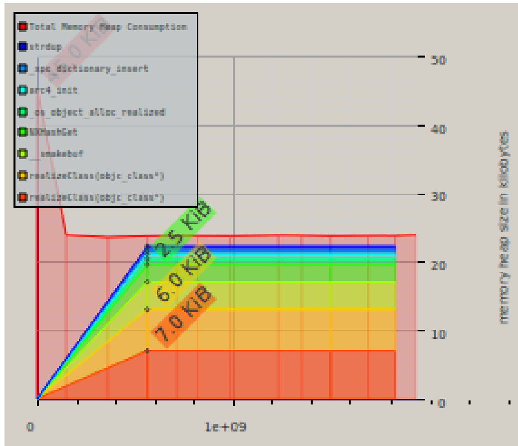



(a) Program run-time in logarithmic scale.

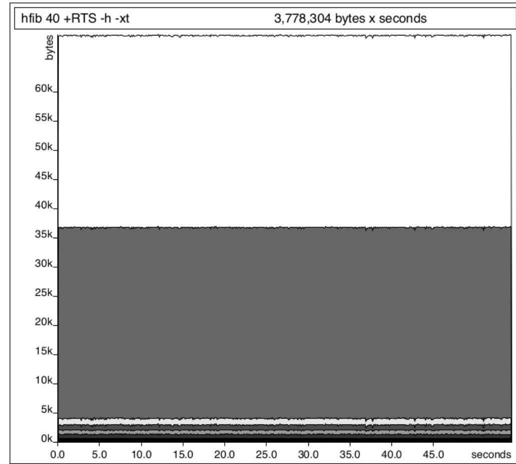



(b) Shared libraries, heap and stack use in *MB*.

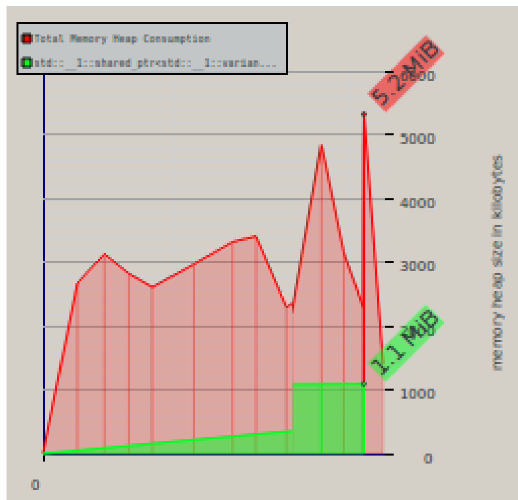
Figure 3-4: Performance and memory benchmarks for four Coq programs, compiled with MCQC versus GHC. Increasing values for $N$ were used for calculating Fig. 3-4a and only the highest value $N$ was used for memory benchmarks in Fig. 3-4b. The corresponding Coq code is in Appendix A.



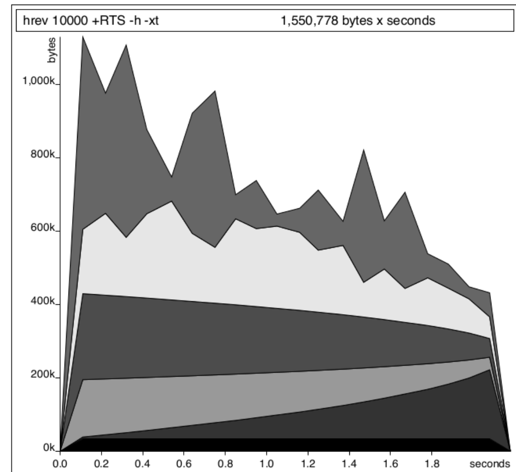

(a) `Fib.v` extraction with MCQC



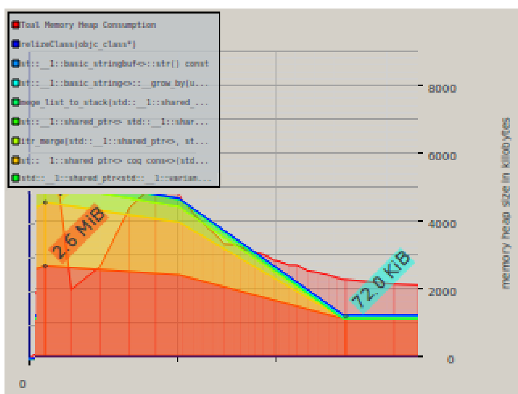

(b) `Fib.v` extraction with GHC



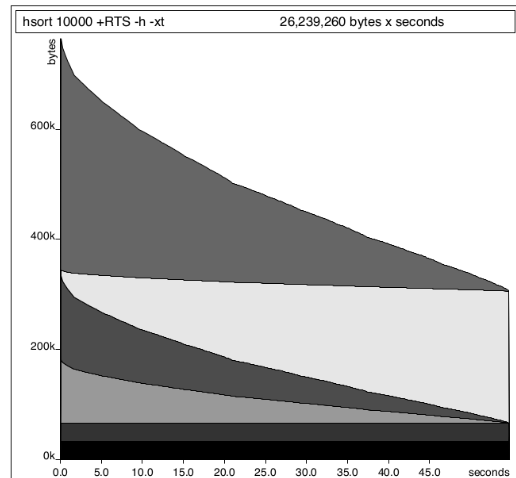

(c) `Rev.v` extraction with MCQC




(d) `Rev.v` extraction with GHC




(e) `Sort.v` extraction with MCQC




(f) `Sort.v` extraction with GHC

Figure 3-5: Comparative memory profiling of four programs with MCQC and Haskell, over time. The left column is MCQC, clang-7.0 with debug symbols, valgrind and `massif`. The right column is GHC with profiling enabled.

The heap profiles of MCQC and GHC executables show some similarities in terms of GC, shown in Fig. 3-5. For `Sort.v`, an initial quadratic curve of allocations for `mergesort` stack frames is seen in Fig. 3-5e and then the `merge` function linearly concatenates all the leafs in the recursion tree while simultaneously releasing them. This leads to the asymptotic drop in memory usage, as seen in Fig. 3-5e, Fig. 3-5f.

# Chapter 4

# Conclusion

We have presented the MCQC compiler, a novel approach to generating executable formally verified code directly from the Gallina functional specification. MCQC compiles pure and effectful Gallina programs to C++17, with no RTS or GC requirements. The auto-generated C++17 code can then be compiled for multiple architectures, by using the Clang compiler [11].

Code compiled with MCQC has a similar size trusted code base (TCB) as standard Coq extraction mechanisms [14]. The MCQC TCB includes the Clang compiler and MCQC itself, as well as the base types library. Traditional Coq extraction to Haskell and Ocaml includes both the compiler (GHC or Ocaml) and RTS in the TCB, MCQC does not need a trusted RTS, which makes the MCQC TCB smaller than standard Coq execution through Haskell and OCaml.

We have demonstrated MCQC, a compiler that extracts performant, runnable C++17 code from verified Gallina. To demonstrate MCQC we wrote a web application for online transactions in Gallina that we compiled to C++17 with MCQC. A minimal amount of boilerplate code and proofs was required, while the use of MCQC made it possible to write a verified web application without leaving the Coq proof-assistant.

Existing approaches to formally verified software have been labor intensive. They require rewritting the whole application stack, even OS, hardware and CPU features, to be formally verified. MCQC allows a pragmatic approach to verification where the

application developer has the final judgment of what requires verification effort and what does not.

By linking formally verified logic with well-established trusted software, we plan to write more applications in Gallina as well as proofs for their correctness. We also hope to see MCQC used as part of the Coq ecosystem for the execution of formally verified code across platforms and architectures, making formally verified programs possible without scraping the whole stack.

# Appendices

# Appendix A: Benchmark code

Here is the Coq code for the benchmarks in Fig. 3-4 and Fig, 3-5. The `main` definitions in Fig. A.3 and Fig. A.4 compile to C making it easy to execute the benchmarks with MCQC.

```
Fixpoint fib(n: nat) :=
  match n with
    | 0 => 1
    | S sm =>
      match sm with
        | 0 => 1
        | S m => (fib m) + (fib sm)
      end
  end.
```

Figure A.1: Function that computes the n-th Fibonacci number, uses `Nat` from Coq.Init.

```
Require Import Coq.Lists.List.

Fixpoint fact(n: nat) :=
  match n with
    | 0 => 1
    | S sm => n * (fact sm)
  end.
```

Figure A.2: Factorial function, an example of a tail-recursive function. Uses `Nat` from Coq.Init

```
Fixpoint rev {T} (l : list T) : list T :=
  match l with
    | [] => []
    | h :: ts => rev(ts) ++ [h]
  end.

(** Generate an arithmetic series, list of [1..n] *)
Fixpoint series (n: nat) :=
  match n with
    | 0 => []
    | S m => n :: series m
  end.

Definition main :=
  print (show (rev (series 10000))).
```

Figure A.3: List reverse function, not tail-recursive and requires quadratic memory allocations. Uses `List` from Coq.Lists

```
Definition  merge l1 l2 :=
  match l1, l2 with
  | [], _ => l2
  | _, [] => l1
  | a1::l1', a2::l2' =>
      if a1 <=? a2 then a1 :: merge l1' l2 else a2 :: merge l1 l2'
  end.

Fixpoint merge_list_to_stack stack l :=
  match stack with
  | [] => [Some l]
  | None :: stack' => Some l :: stack'
  | Some l' :: stack' => None :: merge_list_to_stack stack' (merge l' l)
  end.

Fixpoint merge_stack stack :=
  match stack with
  | [] => []
  | None :: stack' => merge_stack stack'
  | Some l :: stack' => merge l (merge_stack stack')
  end.

Fixpoint iter_merge stack l :=
  match l with
  | [] => merge_stack stack
  | a::l' => iter_merge (merge_list_to_stack stack [a]) l'
  end.

Definition sort := iter_merge [].

Definition main :=
  let test := series 10000 in
  print (show (sort test)).
```

Figure A.4: Mergesort with merge stacks from Coq.Sorting.Mergesort. Uses `List` from Coq.Lists.

# Bibliography


[1] Abhishek Anand, Andrew Appel, Greg Morrisett, Zoe Paraskevopoulou, Randy Pollack, Olivier Savary Belanger, Matthieu Sozeau, and Matthew Weaver. Certicoq: A verified compiler for coq. In *The Third International Workshop on Coq for Programming Languages (CoqPL)*, 2017.

[2] Bruno Barras, Samuel Boutin, Cristina Cornes, Judicaël Courant, Jean-Christophe Filliatre, Eduardo Gimenez, Hugo Herbelin, Gerard Huet, Cesar Munoz, Chetan Murthy, et al. *The Coq proof assistant reference manual: Version 6.1*. PhD thesis, Inria, 1997.

[3] Koen Claessen and John Hughes. Quickcheck: a lightweight tool for random testing of haskell programs. *Acm sigplan notices*, 46(4):53–64, 2011.

[4] David Cock. Bitfields and tagged unions in c: Verification through automatic generation. *VERIFY*, 8:44–55, 2008.

[5] Luis Damas and Robin Milner. Principal type-schemes for functional programs. In *Proceedings of the 9th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 207–212. ACM, 1982.

[6] François-Nicola Demers and Jacques Malenfant. Reflection in logic, functional and object-oriented programming: a short comparative study. In *Proceedings of the IJCAI*, volume 95, pages 29–38, 1995.

[7] Andres Erbsen, Jade Philipoom, Jason Gross, Robert Sloan, and Adam Chlipala. Simple high-level code for cryptographic arithmetic-with proofs, without compromises. In *Simple High-Level Code for Cryptographic Arithmetic-With Proofs, Without Compromises*, page 0. IEEE.

[8] Andreas Haas, Andreas Rossberg, Derek L Schuff, Ben L Titzer, Michael Holman, Dan Gohman, Luke Wagner, Alon Zakai, and JF Bastien. Bringing the web up to speed with webassembly. In *ACM SIGPLAN Notices*, volume 52, pages 185–200. ACM, 2017.

[9] SL Peyton Jones, Cordy Hall, Kevin Hammond, Will Partain, and Philip Wadler. The glasgow haskell compiler: a technical overview. In *Proc. UK Joint Framework for Information Technology (JFIT) Technical Conference*, volume 93, 1993.

[10] Fairouz Kamareddine and Alejandro Ríos. A $\lambda$-calculus à la de bruijn with explicit substitutions. In *International Symposium on Programming Language Implementation and Logic Programming*, pages 45–62. Springer, 1995.

[11] Chris Lattner. Llvm and clang: Next generation compiler technology. In *The BSD Conference*, pages 1–2, 2008.

[12] Xavier Leroy, Damien Doligez, Alain Frisch, Jacques Garrigue, Didier Rémy, and Jérôme Vouillon. The ocaml system release 4.02. *Institut National de Recherche en Informatique et en Automatique*, 54, 2014.

[13] Xavier Leroy et al. The compcert verified compiler. *Documentation and user's manual. INRIA Paris-Rocquencourt*, 2012.

[14] Pierre Letouzey. Extraction in coq: An overview. In *Conference on Computability in Europe*, pages 359–369. Springer, 2008.

[15] Barbara Liskov and Liuba Shrira. *Promises: linguistic support for efficient asynchronous procedure calls in distributed systems*, volume 23. ACM, 1988.

[16] Matthew Milano and Andrew C Myers. Mixt: a language for mixing consistency in geodistributed transactions. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 226–241. ACM, 2018.

[17] Eric Mullen, Stuart Pernsteiner, James R Wilcox, Zachary Tatlock, and Dan Grossman. Œuf: minimizing the coq extraction tcb. In *Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs*, pages 172–185. ACM, 2018.

[18] Nicholas Nethercote and Julian Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *ACM Sigplan notices*, volume 42, pages 89–100. ACM, 2007.

[19] Julian Seward, Nicholas Nethercote, and Josef Weidendorfer. *Valgrind 3.3-advanced debugging and profiling for gnu/linux applications*. Network Theory Ltd., 2008.

[20] Steffen Van Bakel. Principal type schemes for the strict type assignment system. *Journal of Logic and Computation*, 3(6):643–670, 1993.

[21] Philip Wadler. A prettier printer. *The Fun of Programming, Cornerstones of Computing*, pages 223–243, 2003.

[22] Philip Wadler and Stephen Blott. How to make ad-hoc polymorphism less ad hoc. In *Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 60–76. ACM, 1989.

[23] Werner Welsch. Templates in depth. 2011.

[24] Hongwei Xi and Frank Pfenning. Dependent types in practical programming. In *Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 214–227. ACM, 1999.