Extracting and Optimizing Formally Verified Code for Systems Programming

Extracting and optimizing formally verified code for systems programming

Eleftherios Ioannidis, Frans Kaashoek, and Nickolai Zeldovich

Massachusetts Institute of Technology
elefthei@mit.edu

Abstract. MCQC is 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 few memory and performance optimizations, MCQC combines verifiability with memory and runtime performance. By handling effectful and pure functions separately MCQC can generate executable verified code directly from Gallina, reducing the effort of implementing and executing verified systems.

Keywords: Formal Verification · Functional Compiler · Extraction · Systems

1 Introduction

The formal verification of computer systems has been a continuous subject of research over the last decade, with verified file systems [5][1], kernels [11][15], distributed systems [24] and cryptographic algorithms [9][4]. Formal proofs about programs are developed in a dependently-typed language [25], inside a mechanized proof-assistant, like Coq [25] [3]. Coq has its own programming language, Gallina which together with the proof-language Ltac enable the development of formally verified algorithms. The compilation and execution of formally verified software written in Gallina, for systems programming with side-effects and an emphasis on performance, is the focus of this paper.

1.1 The problem of code generation

The functional, dependent nature of Gallina makes it difficult to execute outside Coq. There are a few roadblocks to generating performant, effectful code from Gallina, which must be addressed:

1. Coq relies on a runtime system (RTS) and garbage collection (GC) for memory management, which makes it hard to execute verified code on bare hardware (OS, embedded systems, firmware etc).

2. Integral and bitfield types are inductively defined in Coq and they do not fit into CPU registers, making the performance overhead of executing Gallina prohibitive.
3. Gallina is completely pure and it cannot generate any observable effects.
4. The performance of dynamic memory datastructures such as lists, maps and trees, suffers during extraction. Coq passes arguments by value, which leads to excessive copying and a dependence on GC.

There are currently two approaches to generating formally verified, executable code and they each address a subset of the issues stated above; by verified compilation of deep embeddings and by extraction of shallow embeddings [2][18]. The first method requires advanced knowledge of programming language theory and involves defining, proving and compiling an embedded Domain Specific Language (eDSL) inside Coq, with varying degrees of proof automation available.

This paper focuses on the second approach of shallow embeddings and introduces the Monadic Coq Compiler (MCQC), a compiler for Gallina by means of extraction using C++17 as an intermediate representation. C++17 is a suitable intermediate language as it offers parametric polymorphism through templates, algebraic datatypes (ADTs) through variants and GC through smart pointers. The output C++17 can be compiled by any modern C compiler with no external dependencies. We chose the clang C compiler [16] for MCQC.

1.2 Previous work

The CertiCoq compiler [2] implements Coq's language inside the Coq proof-assistant, allowing for the verified compilation of Gallina. However, CertiCoq depends on a runtime GC and cannot generate static, stand-alone assembly. The Œuf verified extractor [21] reifies Gallina into an abstract syntax tree (AST) that it then translates to CompCert's intermediate representation [17] but does not target full Gallina, only a small subset of it relevant to reactive systems. The Fiat compiler does verified compilation of an eDSL down to static C but is only applicable to the domain of cryptographic algorithms [10].

1.3 Contributions

MCQC is a compiler, a library of native bitfield types and an `IO` library for interacting with the real world. The MCQC native library is modeled after the Coq standard library and obeys the same semantics, while offering fast, native computations. MCQC supports pure functional programming and effectful monadic `IO` operations, similar to the Haskell `IO` monad. Although side-effects cannot be executed inside Coq, they are compiled to real system calls by MCQC and executed by the underlying OS.

Using MCQC we have successfully compiled multiple types and functions from Coq's standard library. We have also written a proof-of-concept web application for online payments, with the web server written in Gallina and compiled to C++17 and the client written in Gallina and compiled to Webassembly. In

both cases, a minimal amount of boilerplate code and proofs was required, while MCQC made it possible to write and test verified client and server code without leaving the Coq proof-assistant.

MCQC has some limitations compared to Gallina executed inside Coq. MCQC cannot generate code for Gallina typeclass instances, as typeclasses offer a model for ad-hoc polmorphism more general than C++ templates [23]. MCQC has limited multi-threading support. As part of `Proc` MCQC implements *spawn* : $\forall T, (T \to unit) \to T \to proc\ unit$ which can execute functions with no return values in parallel via `std::future`. To support parallel execution with return types, a promises interface would be more effective [19] in the future. Finally, the library of base types in C++17 is not formally verified. To ensure correctness with respect to the Coq standard library, a property-based testing suite is used [6].

2 Design

This section covers the most interesting part of the design; the full design is described in a master thesis [12]. MCQC is a compiler and a library of base types and system calls in C++17. The compiler 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-8.5.1). MCQC compiles the Gallina AST to C++17 which then clang compiles to assembly [16] and links with the library, as shown in Fig. 1.

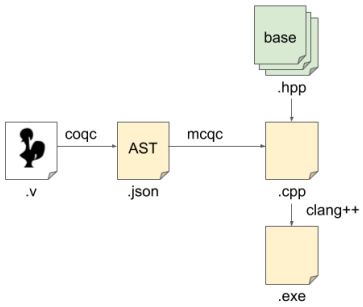


Fig. 1: MCQC block diagram. Coq files are the input, MCQC generates C++17 code and clang compiles it and links with the base type library to produce an executable. The white box is the input Gallina program, green boxes show imported libraries and yellow boxes show auto-generated files.

The input Gallina AST is described by an input grammar which is defined in the MCQC thesis [12] and is the starting point for MCQC. The top level structure is a `Module`, which contains multiple top-level `Declarations`. Gallina

declarations can be either inductive types, type aliases, named fixpoints or named expressions. MCQC breaks the compilation process into five stages; type inference, base semantics, algebraic datatypes, monadic effects and pretty-printing C++17.

2.1 Type inference

Coq extraction transforms dependently typed Gallina to a simpler Hindley-Milner (HM) language similar to ML [8]. Type inference starts at each function declaration, which is always guaranteed to be well-typed by Coq prior to extraction. Each binder is added to the local context as a constraint and those constraints are solved while traversing the AST by standard HM type inference [8].

The C++17 type system does not have support for function types. MCQC preserves function types until the pretty-printing stage, when they are transformed to C++ templates. We chose function templates over `std::function` as clang will inline functional arguments when they are passed as templates, offering better performance for higher-order functions. In addition, MCQC adds a type annotation in the return type with the `std::invoke_result_t` template function, to help clang type resolution [14].

2.2 Base semantics

Using Coq's standard library of base types can have a significant performance overhead as Coq defines base types inductively. MCQC substitutes slow Coq base types with their corresponding C++17 native, safe types. More details on safety of the base type library can be found in the MCQC thesis [12]. Base types are always passed by value in MCQC and conversely, ADTs are always passed by smart pointer.

Pattern matching in Coq corresponds to the polymorphic high-order function `match` in C++17, which is implemented differently for each type as seen in Fig. 2. As native types are succeptible to weak typing MCQC strengthens the C++17 type system with template metaprogramming (TMP) as seen is Fig. 2. A substitution failure at `std::enable_if_t` means the function will quietly disappear at clang compile-time without errors, a pattern known in C++ as SFINAE (Substituition Failure Is Not An Error) [14].

2.3 Algebraic Data types (ADTs)

MCQC transforms Coq ADT definitions, like lists, trees etc, to a reference-counted, pointer datastructure in C++17. Sum types are transformed to tagged-unions implemented by `std::variant` [7] and product types are implemented by C structs. The combination of sums and products allows MCQC to define any algebraic data type in C++17 [20]. Finally, pattern-matching for those types is auto-generated as the polymorphic, high-order `match` function. ADTs are passed

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

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

// Pattern matching on nat
template<typename F0, typename FS,
  typename = enable_if_t<CallableWith<F0>>,
  typename = enable_if_t<CallableWith<FS, nat>>>
constexpr auto match(nat a, F0 f, FS g) {
  switch(a) {
  case 0:  return f();    // Call 0 clause
  default: return g(a-1); // Call S clause
  }
}
```

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

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

Fig. 2: 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.

by smart pointer, a reference counted pointer that requires no GC, implemented via `std::shared_ptr`. An example of generating a pointer list from the ADT list definition in Coq can be seen in Fig. 3 and more details on ADT generation can be found in the MCQC thesis [12].

2.4 Monadic effects (`Proc`)

Coq is so pure it 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 with the `Proc` monad, similar to the Haskell `IO` monad [13]. Effectful monads in Gallina elaborate to imperative-style C++ statements, as shown in Fig. 4. An example of generating an implementation for the `cat` utility is shown in Fig. 4.

2.5 Pretty-print C++17

In order to apply transformations and finally pretty-print C++17, MCQC transforms the input Coq AST to an intermediate representation closer to C++17. Going from that representation to a `.cpp` file is a matter of implementing a Wadler/Leijen prettyprinter [22].

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

```
template<class T>
struct Coq_nil {};
// Forward declarations
template<class T>
struct Coq_cons;
template<class T>
// Reference counted tagged-union
using list = std::shared_ptr<
    std::variant<Coq_nil<T>, Coq_cons<T>>>;

template<class T>
struct Coq_cons {
  T a;
  list<T> b;
};

// Pattern match
template<class T, class U, class V>
auto match(list<T> self, U f, V g) {
  return std::visit(*self, overloaded {
    [=](Coq_nil<T> _) { return f(); },
    [=](Coq_cons<T> _) { return g(_.a, _.b); }
  });
}
```

Fig. 3: Polymorphic list definition in Coq, MCQC generates the pointer data structure on the right, as well as `match` to deconstruct it.

```
(** 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;

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");
}
static proc<string> read(fd f, nat size) {
  auto dp = string(size, '\0' );
  sys::read(f, &(dp[0]), sizeof(char)*size);
  return dp;
}
static proc<void> close(fd f) {
  if(sys::close(f)) {
    throw IOException("Could not close file");
  }
}
static proc<void> print(string s) {
  std::cout << s << std::endl;
}
```

```
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);
}
```

Fig. 4: The `cat` UNIX utility that displays a text file. Instances of `proc` are translated to imperative C++ system calls. The shaded box surrounds Coq and C++17 boilerplate code, part of the MCQC library.

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 are; can we link verified and unverified code to create end-to-end applications, can we get get better memory performance than extracted Haskell and can we get runtime performance comparable to Haskell compiled with GHC.

MCQC is open source under an MIT license and can be found here `https://github.com/mit-pdos/mcqc`. MCQC is implemented in 1800 lines of Haskell and 600 lines of C++17 code for the base type and `proc` library.

3.1 Linking verified applications

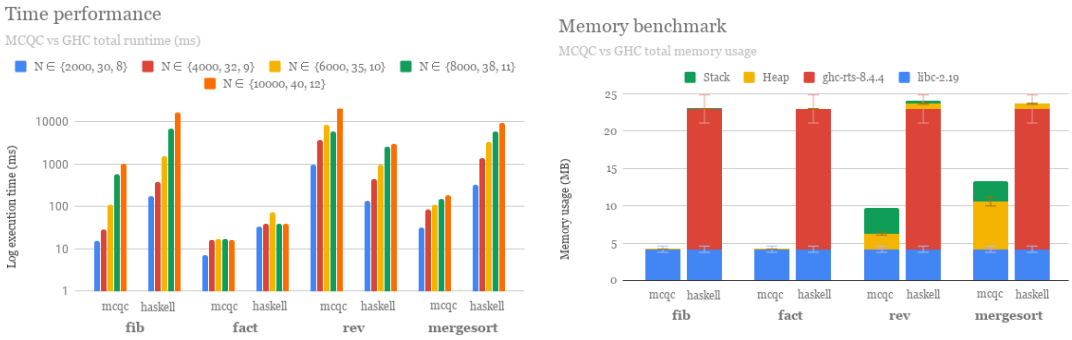
In order to demonstrate MCQC's capabilities we have developed a demo web application for payments, the verified *Zoobar* server. The design and implementation details of the *Zoobar* server is presented in full detail in the mcqc thesis [12]. The *Zoobar* server demonstrates the ease of linking code compiled with MCQC, as both the server and client were built and proven in Coq and extracted to C++17 before linking with the HTTP libraries. The proof effort required for proving the transaction logic is minimal and focuses on the code that is most important. With the *Zoobar* demo we demonstrate a hybrid approach to verification, by combining verified logic with unverified trusted code.

3.2 Benchmarks

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. 5a. MCQC reduces the memory footprint of executing verified programs by 66.25% on average compared to GHC, as seen in Fig. 5b.

We compare the performance of C++17 code generated with MCQC against Haskell code extracted from Coq with native types to ensure the comparison is fair. The clang-7.0 compiler compiles generated C++17 and `GHC-8.4.4` compiles extracted Haskell. More details on the hardware and profiling tools used can be found in the MCQC thesis [12].

The results in Fig. 5 show MCQC extracted code performs with considerably less memory compared to Haskell and at comparable run-time. Tail-call optimization is supproted in clang so it is supported in MCQC, even across pattern matching. For `fact`, we see no heap or stack usage which confirms TCO has optimized recursion away. Finally, in algorithms that rely on GC we show that MCQC uses less memory compared to Haskell and in most cases, MCQC is faster.



(a) Run-time in logarithmic scale.

(b) Memory; shared libraries, heap and stack.

Fig. 5: Performance and memory benchmarks for four Coq programs compiled with MCQC versus GHC. Increasing values for N were used for calculating Fig. 5a and only the highest value N was used for memory benchmarks.

4 Conclusion

We have presented the MCQC compiler, a novel approach to generating executable formally verified code directly from the Gallina functional specification. Code compiled with MCQC has a TCB comparable to standard Coq extraction mechanisms [18]. The MCQC TCB includes the clang compiler and MCQC itself, as well as the base types library. Coq extraction to Haskell and Ocaml includes the compiler and runtime in the TCB, which MCQC does not. We hope to see MCQC used as part of the Coq ecosystem, for the execution of formally verified code without scraping the full stack.

References

1. Amani, S., Hixon, A., Chen, Z., Rizkallah, C., Chubb, P., O'Connor, L., Beeren, J., Nagashima, Y., Lim, J., Sewell, T., et al.: Cogent: Verifying high-assurance file system implementations. ACM SIGOPS Operating Systems Review **50**(2), 175–188 (2016)
2. Anand, A., Appel, A., Morrisett, G., Paraskevopoulou, Z., Pollack, R., Belanger, O.S., Sozeau, M., Weaver, M.: Certicoq: A verified compiler for coq. In: The Third International Workshop on Coq for Programming Languages (CoqPL) (2017)
3. Barras, B., Boutin, S., Cornes, C., Courant, J., Filliatre, J.C., Gimenez, E., Herbelin, H., Huet, G., Munoz, C., Murthy, C., et al.: The Coq proof assistant reference manual: Version 6.1. Ph.D. thesis, Inria (1997)
4. Bhargavan, K., Bond, B., Delignat-Lavaud, A., Fournet, C., Hawblitzel, C., Hritcu, C., Ishtiaq, S., Kohlweiss, M., Leino, R., Lorch, J., et al.: Everest: Towards a verified, drop-in replacement of https. In: LIPIcs-Leibniz International Proceedings in Informatics. vol. 71. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik (2017)

5. Chen, H., Ziegler, D., Chajed, T., Chlipala, A., Kaashoek, M.F., Zeldovich, N.: Using crash hoare logic for certifying the fscq file system. In: Proceedings of the 25th Symposium on Operating Systems Principles. pp. 18–37. ACM (2015)
6. Claessen, K., Hughes, J.: Quickcheck: a lightweight tool for random testing of haskell programs. Acm sigplan notices **46**(4), 53–64 (2011)
7. Cock, D.: Bitfields and tagged unions in c: Verification through automatic generation. VERIFY **8**, 44–55 (2008)
8. Damas, L., Milner, R.: Principal type-schemes for functional programs. In: Proceedings of the 9th ACM SIGPLAN-SIGACT symposium on Principles of programming languages. pp. 207–212. ACM (1982)
9. Erbsen, A., Philipoom, J., Gross, J., Sloan, R., Chlipala, A.: Simple high-level code for cryptographic arithmetic-with proofs, without compromises. In: Simple High-Level Code for Cryptographic Arithmetic-With Proofs, Without Compromises. p. 0. IEEE
10. Erbsen, A., Philipoom, J., Gross, J., Sloan, R., Chlipala, A.: Simple high-level code for cryptographic arithmetic-with proofs, without compromises. In: Simple High-Level Code for Cryptographic Arithmetic-With Proofs, Without Compromises. p. 0. IEEE
11. Gu, L., Vaynberg, A., Ford, B., Shao, Z., Costanzo, D.: Certikos: a certified kernel for secure cloud computing. In: Proceedings of the Second Asia-Pacific Workshop on Systems. p. 3. ACM (2011)
12. Ioannidis, E.: Extracting and optimizing low-level bytecode from high-level verified coq (2019)
13. Jones, S.P., Hall, C., Hammond, K., Partain, W., Wadler, P.: The glasgow haskell compiler: a technical overview. In: Proc. UK Joint Framework for Information Technology (JFIT) Technical Conference. vol. 93 (1993)
14. Josuttis, N.M.: C++ Templates: The Complete Guide. Addison-Wesley Professional (2003)
15. Klein, G., Elphinstone, K., Heiser, G., Andronick, J., Cock, D., Derrin, P., Elkaduwe, D., Engelhardt, K., Kolanski, R., Norrish, M., et al.: sel4: Formal verification of an os kernel. In: Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles. pp. 207–220. ACM (2009)
16. Lattner, C.: Llvm and clang: Next generation compiler technology. In: The BSD Conference. pp. 1–2 (2008)
17. Leroy, X., et al.: The compcert verified compiler. Documentation and user's manual. INRIA Paris-Rocquencourt (2012)
18. Letouzey, P.: Extraction in coq: An overview. In: Conference on Computability in Europe. pp. 359–369. Springer (2008)
19. Liskov, B., Shrira, L.: Promises: linguistic support for efficient asynchronous procedure calls in distributed systems, vol. 23. ACM (1988)
20. Magalhães, J.P., Dijkstra, A., Jeuring, J., Löh, A.: A generic deriving mechanism for haskell. ACM Sigplan Notices **45**(11), 37–48 (2010)
21. Mullen, E., Pernsteiner, S., Wilcox, J.R., Tatlock, Z., Grossman, D.: Œuf: minimizing the coq extraction tcb. In: Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs. pp. 172–185. ACM (2018)
22. Wadler, P.: A prettier printer. The Fun of Programming, Cornerstones of Computing pp. 223–243 (2003)
23. Wadler, P., Blott, S.: How to make ad-hoc polymorphism less ad hoc. In: Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages. pp. 60–76. ACM (1989)

24. Wilcox, J.R., Woos, D., Panchekha, P., Tatlock, Z., Wang, X., Ernst, M.D., Anderson, T.: Verdi: a framework for implementing and formally verifying distributed systems. In: ACM SIGPLAN Notices. vol. 50, pp. 357–368. ACM (2015)
25. Xi, H., Pfenning, F.: Dependent types in practical programming. In: Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages. pp. 214–227. ACM (1999)