Crane Lowers Rocq Safely into C++

(Extended Abstract)

Matthew Z. Weaver
mweaver89@bloomberg.net
Bloomberg
New York, NY, USA

Joomy Korkut
jkorkut@bloomberg.net
Bloomberg
New York, NY, USA

Abstract

We report on our ongoing effort to extract verified programs from the Rocq Prover into production-grade C++. While existing compilers and extractors from Rocq target high-level functional languages (OCaml, Haskell) or lower-level imperative ones (C, Rust), none were well-suited for Bloomberg's requirements. We introduce Crane, a new extraction method to generate idiomatic, functional, memory- and thread-safe C++ code aligned with Bloomberg's coding practices. Our approach uses modern C++ features to represent Rocq's functional constructs in a way that preserves readability and maintainability. Our tool can work with mappings of Rocq data types to C++ standard library types, or to Bloomberg's core library types to facilitate integration with existing C++ code. Additionally, we provide concurrency primitives in Rocq which compile into software transactional memory (STM) constructs in C++, enabling safe concurrent execution. This extended abstract sketches out the design, implementation challenges, and early lessons learned in our quest to integrate verified functional programs into complex concurrent C++ systems.

1 Introduction

Testing remains a fundamental practice for building confidence in software, but it can only establish correctness over a finite set of inputs. It cannot rule out bugs across all possible executions. To obtain stronger guarantees, we turn to formal verification, and in particular to certified programming techniques that allow us to develop programs alongside mathematical proofs of their correctness. However, there is a significant gap between the languages used to write certified programs and those relied upon in production systems. Bridging this gap is crucial for bringing the benefits of formal verification into real-world software systems.

In the Infrastructure & Security Research group in Bloomberg's Office of the CTO, the Rocq Prover [20] (formerly known as Coq) is our principal tool for writing certified programs. Across Bloomberg, however, C++ remains the primary programming language and the lingua franca of our engineering teams. Many of our production systems, libraries, and tooling are written in C++, whose design patterns, idioms, and performance characteristics are familiar to our engineers. To that end, Bloomberg has engineered its BDE Development Environment (BDE) [3], a custom C++ development environment with comprehensive in-house coding standards [4, 12]. These conventions have shaped how Bloomberg engineers write, read, and reason about code over the past two and a half decades.

Introducing verified components into this landscape requires credibility as much as correctness. For extracted code to be adopted and maintained, it must be idiomatic, readable, and consistent with our established C++ practices. Engineers must be able to inspect and reason about the output without needing to learn unfamiliar languages or design patterns. To bridge this gap, we aim to meet our engineers where they are by lowering verified Rocq code into a familiar, battle-tested environment, where safety, performance, and readability can coexist. To this end, we introduce Crane[1], our new extraction method from Rocq to C++, for generating idiomatic, functional, memory- and thread-safe C++ code that adheres to Bloomberg's development standards.

Crane has a twofold role in our certified programming strategy at Bloomberg:

1. Crane enables Bloomberg engineers to implement high-assurance components in Rocq, which can be extracted into performant, idiomatic, and maintainable C++ via Crane, making the resulting code suitable for direct integration into production.
2. Crane allows a small team of formal verification engineers with deep expertise in formal methods to provide verified, opt-in libraries that application developers can adopt incrementally, without needing such expertise themselves.

Looking beyond these immediate applications, we see Crane as a step toward treating C++ as a portable high-level compilation target for functional languages. Modern C++ offers a functional subset rich enough to express higher-order code while still supporting low-level, hand-tuned optimizations, making it an attractive "portable assembly language" for compilers from higher-level languages.

2 Our philosophy

At Bloomberg, the scale and complexity of our production systems demand techniques that balance formal rigor with real-world constraints. With Crane, we take a pragmatic, lightweight approach: rather than prioritizing a fully verified extractor, we focus on generating code that integrates seamlessly into Bloomberg's infrastructure while remaining readable, idiomatic, and performant.

Verifying a compiler or extractor is a herculean undertaking, as evidenced by projects such as CompCert, whose verified C compiler required *many* person-years of formalization and proof effort [13]. Achieving that level of assurance demands simplifying the code generation algorithm and minimizing language features to keep the verification tractable, which would force us into tradeoffs that make the resulting code generation impractical for industrial use. Our immediate priority, instead, is to generate output that engineers can read, reason about, and maintain effectively.

To further strengthen this foundation, we are developing tools for random Rocq program generation and differential fuzzing, comparing Crane's output against that of other extractors and compilers. Together with static analysis tools for memory and thread safety, these experiments will empirically validate Crane's correctness and guide improvements to its extraction system. This work, inspired by the state-of-the-art on random program generation [7, 9, 19], is ongoing, and we expect it to evolve alongside Crane's feature coverage.

[1] Available at https://github.com/bloomberg/crane.

The emphasis on readability also ensures that the generated code remains compatible with the lightweight verification methods already common in Bloomberg's development practices, such as structured code review, property-based testing, fuzzing, and static analysis for memory and thread safety. These practices, while *not* formally verified, provide strong practical assurance when combined with certified Rocq source code.

Moreover, we have designed Crane as a general-purpose tool for any Rocq programmer, independent of Bloomberg-specific infrastructure. Integrations such as BDE support are modular and optional, allowing users to target the C++ standard library or other stacks. This design keeps Crane flexible, portable, and broadly applicable to any Rocq programmer's needs.

3 Novel features

Macros for custom extraction of Rocq definitions. As a practical feature, Rocq's built-in OCaml extractor allows users to specify custom mappings for extracting any constant definition or inductive type to corresponding terms defined in OCaml. These mappings work by replacing each occurrence of the Rocq definition with the string provided when defining the mapping; as OCaml is also a functional language, this simple approach is sufficient.

The situation is not as straightforward in C++; to replicate this feature, we introduce a small macro language that lets users specify where each relevant argument or component appears in the generated string. Consider the following instruction, which describes how to extract the Rocq `option` type to `std::optional` in C++:

```
Crane Extract Inductive option =>
  "std::optional<%t0>"
  [ "std::make_optional<%t0>(%a0)" "std::nullopt" ]
  "if (%scrut.has_value())
     {%t0 %b0a0 = *%scrut; %br0 }
   else { %br1 }"
  From "optional" "memory".
```

In the above syntax, **`%t0`** indicates where to place the first type argument in the mapping of `option`, the `Some` constructor, and within the pattern matching statement. In the mapping for `Some`, the placeholder **`%a0`** indicates where to insert the extracted form of the constructor's first argument. When extracting a pattern match over an `option` type, we generate the `if`-statement shown above, where **`%scrut`** is replaced with the scrutinee of the match statement, **`%b0a0`** is the first argument bound in the first branch, and **`%b0`** and **`%b1`** correspond to the bodies of the first and second branches respectively. Lastly, by indicating that these mappings are "from" `optional` and `memory`, we signal Crane to import both C++ libraries at the top of any generated file that uses this mapping.

Extensible, user-defined monadic effects. To maximize both Crane's flexibility and the readability of the generated C++ code, we allow users to specify their own monadic effects by providing the monad's interface and defining the C++ syntax for each operation. This lets experienced developers build effect interfaces tailored to specific C++ libraries, while general users can rely on standard effects provided in Crane's library.

Crane represents effectful programs using interaction trees [23] by default, to allow composition of different effects and integration with existing verification efforts. However, users can supply their own monads by introducing a type (e.g., `IO : Type → Type`) and its associated operations, such as `print_line : string → IO unit`. On the Crane side, users specify how to interpret types like `IO A` and terms like `print_line s` in C++, after which any Rocq term of type `IO A` extracts to C++ code of the corresponding type.

Leveraging this infrastructure, we have written interfaces for several monadic effects, including one for software transactional memory [8, 21]. STM allows us to define data structures guaranteeing race- and deadlock-free reads and writes in concurrent settings. While starvation is possible, we want to build an interface in which a user can prove the absence of starvation in specific use cases.

4 Future work

Future work includes expanding Crane's coverage of Rocq's syntax and features to currently unsupported constructs, developing a differential random testing tool to generate Rocq programs to detect discrepancies between code produced by Crane and other extractors, and deploying verified BDE library components at scale within Bloomberg's C++ infrastructure.

5 Related work

Verified C++. Numerous projects target the formal verification of C++ code. The CBMC project offers a bounded model checker for C and C++ [5], and Monteiro et al. describe a workflow for model checking C++ programs. In the Rocq literature, Malecha et al. present the BRiCk program logic for reasoning about C++ code within Rocq and Iris.

Rocq extractors. Several extractors and compilers exist for Rocq, targeting a range of functional and imperative languages—including OCaml, Haskell, and Scheme [6, 14]; C [1, 18, 22]; WebAssembly [16]; and more recently, Rust and Elm [2]. Some of these tools, such as CertiCoq [1], aim for full formal verification of the compilation pipeline. Others prioritize practical correctness through empirical validation and robust engineering.

A noteworthy extractor most similar to our approach is MCQC [10, 11], a tool that converts Rocq's JSON extraction output to a memory-safe, functional subset of C++. MCQC possesses many desirable qualities for our use case:

+ It uses modern C++ features such as smart pointers, variants, and lambdas, to write code in a functional style;
+ It translates certain Rocq standard library types to C++ standard library types instead of redefining them;
+ It translates monadic control structures for I/O into sequential (`;`) statements in C++.

However, MCQC also falls short of many of our requirements:

– It hard-codes mappings from Rocq standard library types to C++ standard library types, without allowing for user-defined mappings;
– It does not handle higher-order functions adequately;
– It does not attempt concurrency (let alone user-defined mappings of arbitrary monadic effects);
– Its code generation style differs from BDE's requirements and best practices.

Our project builds directly on MCQC's core ideas but extends them to generate memory-safe C++ that aligns with Bloomberg's libraries and coding standards.

References

[1] Abhishek Anand, Andrew W. Appel, Greg Morrisett, Zoe Paraskevopoulou, Randy Pollack, Olivier Savary Bélanger, Matthieu Sozeau, and Matthew Weaver. 2017. CertiCoq: A Verified Compiler for Coq. *the 3rd International Workshop on Coq for Programming Languages (CoqPL)* (2017). https://web.archive.org/web/20221117172213/https://www.cs.princeton.edu/~appel/papers/certicoq-coqpl.pdf

[2] Danil Annenkov, Mikkel Milo, Jakob Botsch Nielsen, and Bas Spitters. 2022. Extracting functional programs from Coq, in Coq. *Journal of Functional Programming* 32 (2022), e11. doi:10.1017/S0956796822000077

[3] Bloomberg L.P. 2014. BDE. http://github.com/bloomberg/bde

[4] Bloomberg L.P. 2024. BDE C++ Coding Standards. https://web.archive.org/web/20250220223754/https://bloomberg.github.io/bde/knowledge_base/coding_standards.html

[5] Edmund Clarke, Daniel Kroening, and Flavio Lerda. 2004. A Tool for Checking ANSI-C Programs. In *Tools and Algorithms for the Construction and Analysis of Systems*, Kurt Jensen and Andreas Podelski (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 168–176.

[6] Yannick Forster, Matthieu Sozeau, and Nicolas Tabareau. 2024. Verified Extraction from Coq to OCaml. *Proc. ACM Program. Lang.* 8, PLDI, Article 149 (June 2024), 24 pages. doi:10.1145/3656379

[7] Justine Frank, Benjamin Quiring, and Leonidas Lampropoulos. 2024. Generating Well-Typed Terms That Are Not "Useless". *Proc. ACM Program. Lang.* 8, POPL, Article 77 (Jan. 2024), 22 pages. doi:10.1145/3632919

[8] Tim Harris, Simon Marlow, Simon Peyton-Jones, and Maurice Herlihy. 2005. Composable memory transactions. In *Proceedings of the Tenth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (Chicago, IL, USA) *(PPoPP '05)*. Association for Computing Machinery, New York, NY, USA, 48–60. doi:10.1145/1065944.1065952

[9] Tram Hoang, Anton Trunov, Leonidas Lampropoulos, and Ilya Sergey. 2022. Random testing of a higher-order blockchain language (experience report). *Proc. ACM Program. Lang.* 6, ICFP, Article 122 (Aug. 2022), 16 pages. doi:10.1145/3547653

[10] Eleftherios Ioannidis, Frans Kaashoek, and Nickolai Zeldovich. 2019. Extracting and optimizing formally verified code for systems programming. In *NASA Formal Methods: 11th International Symposium, NFM 2019, Houston, TX, USA, May 7–9, 2019, Proceedings 11*. Springer, 228–236. doi:10.1007/978-3-030-20652-9_15

[11] Eleftherios Ioannis Ioannidis. 2019. *Extracting and Optimizing low-level bytecode from High-level verified Coq.* Master's thesis. Massachusetts Institute of Technology.

[12] John Lakos. 1996. *Large-Scale C++ Software Design.* Addison-Wesley Professional, Reading, Massachusetts.

[13] Xavier Leroy. 2009. Formal verification of a realistic compiler. *Commun. ACM* 52, 7 (July 2009), 107–115. doi:10.1145/1538788.1538814

[14] Pierre Letouzey. 2008. Extraction in Coq: An Overview. In *Logic and Theory of Algorithms*, Arnold Beckmann, Costas Dimitracopoulos, and Benedikt Löwe (Eds.). Springer, Berlin, Heidelberg, 359–369. doi:10.1007/978-3-540-69407-6_39

[15] Gregory Malecha, Gordon Stewart, František Farka, Jasper Haag, and Yoichi Hirai. 2022. Developing With Formal Methods at BedRock Systems, Inc. *IEEE Security & Privacy* 20, 3 (2022), 33–42. doi:10.1109/MSEC.2022.3158196

[16] Wolfgang Meier, Martin Jensen, Jean Pichon-Pharabod, and Bas Spitters. 2025. CertiCoq-Wasm: A Verified WebAssembly Backend for CertiCoq. *Proceedings of the 14th ACM SIGPLAN International Conference on Certified Programs and Proofs* (2025), 127–139. doi:10.1145/3703595.3705879

[17] Felipe R. Monteiro, Mikhail R. Gadelha, and Lucas C. Cordeiro. 2022. Model checking C++ programs. *Software Testing, Verification and Reliability* 32, 1 (2022), e1793. doi:10.1002/stvr.1793

[18] Eric Mullen, Stuart Pernsteiner, James R. Wilcox, Zachary Tatlock, and Dan Grossman. 2018. Œuf: Minimizing the Coq Extraction TCB. In *Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs* (Los Angeles, CA, USA) *(CPP 2018)*. Association for Computing Machinery, New York, NY, USA, 172–185. doi:10.1145/3167089

[19] Michał H. Pałka, Koen Claessen, Alejandro Russo, and John Hughes. 2011. Testing an optimising compiler by generating random lambda terms. In *Proceedings of the 6th International Workshop on Automation of Software Test* (Waikiki, Honolulu, HI, USA) *(AST '11)*. Association for Computing Machinery, New York, NY, USA, 91–97. doi:10.1145/1982595.1982615

[20] The Rocq Team. 2025. The Rocq Prover. http://rocq-prover.org

[21] Nir Shavit and Dan Touitou. 1995. Software transactional memory. In *Proceedings of the Fourteenth Annual ACM Symposium on Principles of Distributed Computing* (Ottowa, Ontario, Canada) *(PODC '95)*. Association for Computing Machinery, New York, NY, USA, 204–213. doi:10.1145/224964.224987

[22] Akira Tanaka. 2021. Coq to C translation with partial evaluation. In *Proceedings of the 2021 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation* (Virtual, Denmark) *(PEPM 2021)*. Association for Computing Machinery, New York, NY, USA, 14–31. doi:10.1145/3441296.3441394

[23] Li-yao Xia, Yannick Zakowski, Paul He, Chung-Kil Hur, Gregory Malecha, Benjamin C. Pierce, and Steve Zdancewic. 2019. Interaction Trees: Representing Recursive and Impure Programs in Coq. *Proc. ACM Program. Lang.* 4, POPL, Article 51 (dec 2019), 32 pages. doi:10.1145/3371119