From Proof to Practice: Extracting Verified Programs to Modern C++

Bloomberg

LangSec Workshop at the 47th IEEE Symposium on Security & Privacy
May 21, 2026

Joomy Korkut
jkorkut@bloomberg.net @joomy @joomy@functional.cafe

TechAtBloomberg.com

Testing is great, but not enough!

```
std::optional<std::string> parse_string(std::string_view s) {
  if (s.empty() || s[0] != '"') return std::nullopt;
  std::string out;
  for (size_t i = 1; i < s.size(); i++) {
    if (s[i] == '"') return out;
    if (s[i] == '\\') {
      if (++i >= s.size()) return std::nullopt;
      switch (s[i]) {
        case '"':  out += '"';  break;
        case '\\': out += '\\'; break;
        case '/':  out += '/';  break;
        case 'b':  out += '\b'; break;
        case 'f':  out += '\f'; break;
        case 'n':  out += '\n'; break;
        case 'r':  out += '\r'; break;
        case 't':  out += '\t'; break;
        default:   out += s[i]; break;
      }
    } else {
      out += s[i];
    }
  }
  return std::nullopt;
}
```

This parser accepts \q, \x, \a, ..., but those are not JSON escape sequences.

Testing is great, but not enough!

Tests can find bugs, but they are
useless for proving their **absence.**

We need **formal methods** for that!

Testing is great, but not enough!

```
{
  "role": "user",
  "role": "admin"
}
```

What do we do with duplicate keys?
Some parsers reject, some keep the first, some keep the last, some keep both.

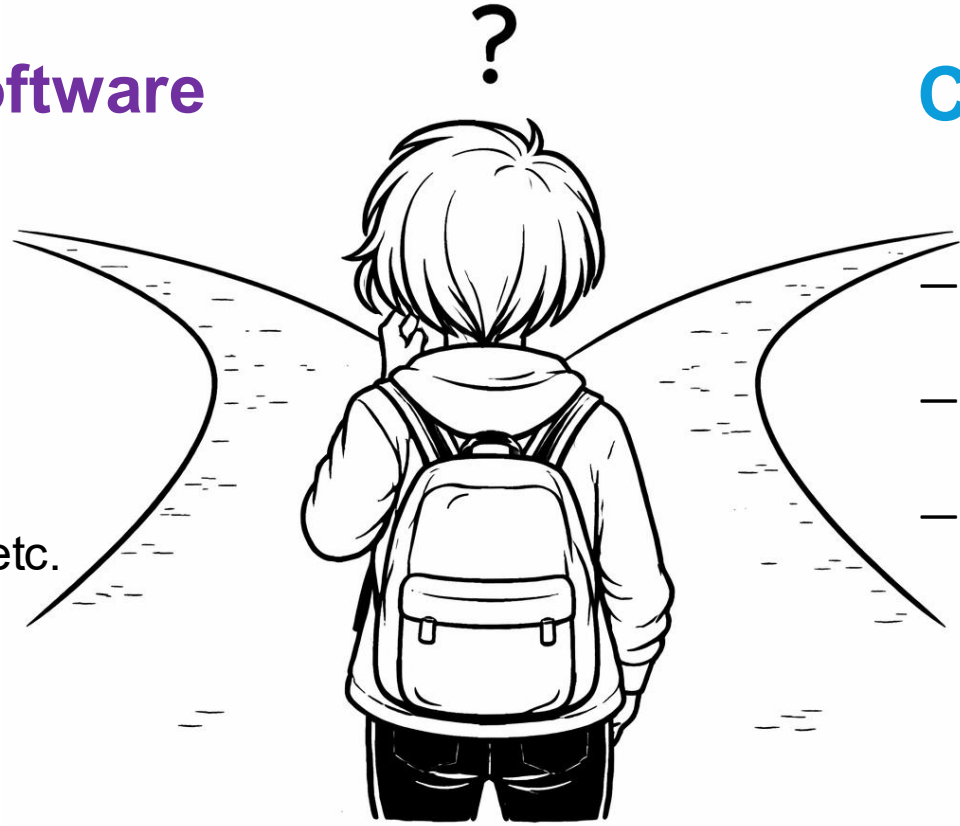
We should **specify** which.

Software verification, but which way?

Verifying existing software

- Frama-C, CBMC, VST for C
- Creusot, Verus, Aeneas, Kani for Rust
- Refinement / liquid types for Haskell, Rust, TS, Java, etc.
- Cameleer, CFML for OCaml

- Can be clunky
- Proofs are brittle
- Often not expressive enough.

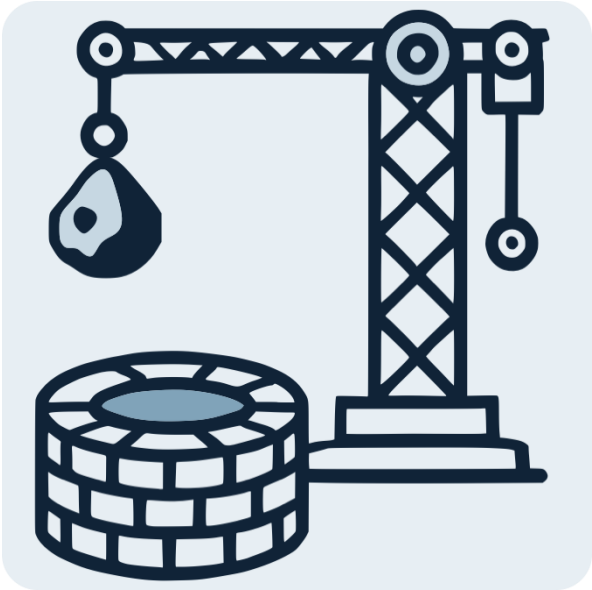


Certified programming

- Create new software that is **correct by construction!**
- Compile / extract the correct program into your favorite language.
- We need to make it work for Bloomberg's needs.

New Tool: Crane

- A new extraction system from Rocq to **C++**.

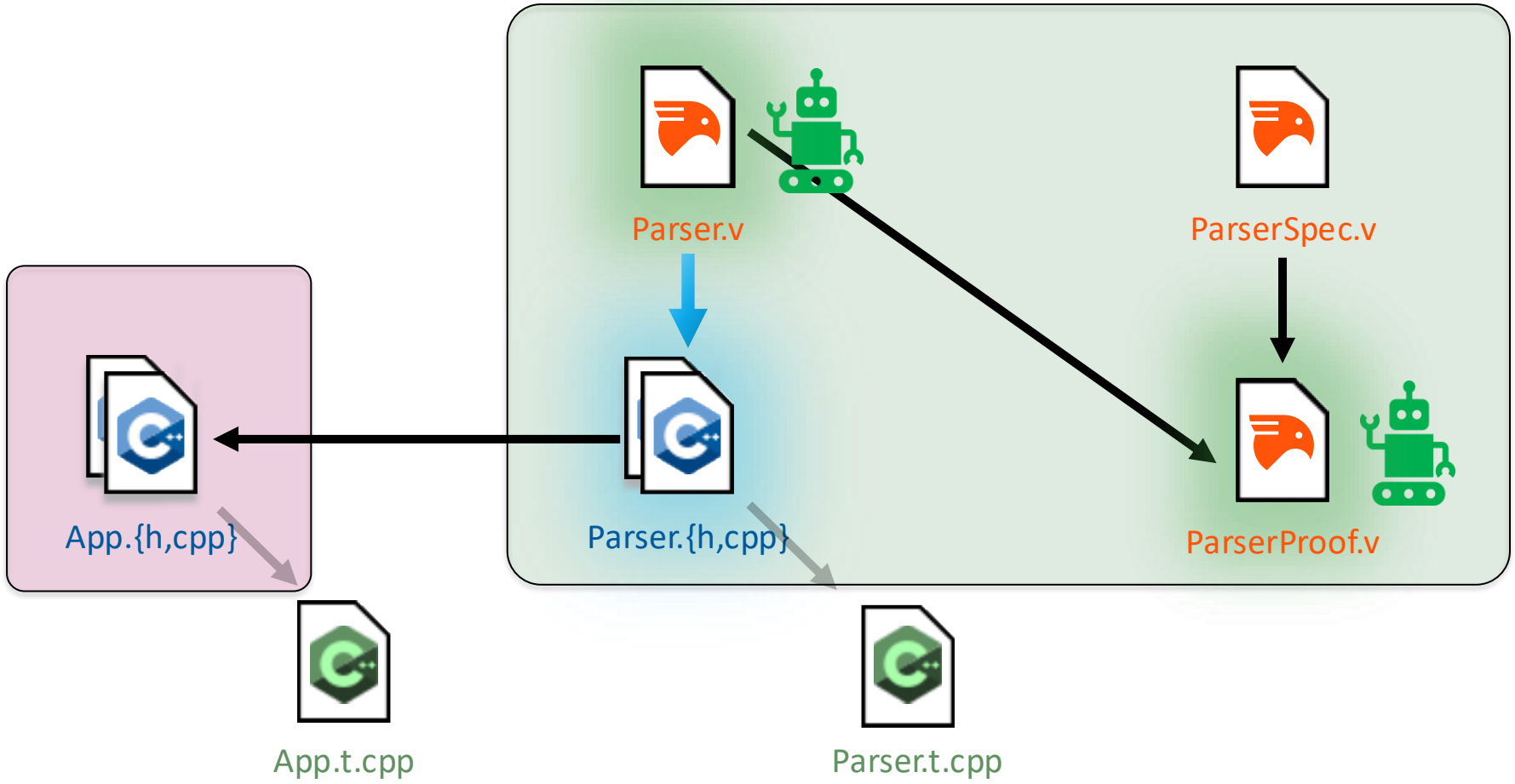


- Generates
  - **functional-style**
  - **memory-safe***
  - **thread-safe***
  - **readable***
  - UB-free*

  C++ code.

- Still under active development!

Workflow with Crane and AI agents



Why C++?

- C++ is the **lingua franca** of our engineering teams.
  We are meeting them where they are.

- C++ has rich **zero-overhead abstractions**:
  variant, unique_ptr, move semantics, concepts, constexpr.

- A **portable assembly language** for functional compilers!

C++'s zero-overhead abstractions turn out to be a
good **compilation target** for verified functional programs.

Inductive types in functional C++

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

```
template <typename A> struct List {
  // TYPES
  struct Nil {};

  struct Cons {
    A a;
    std::unique_ptr<List<A>> l;
  };

  using variant_t = std::variant<Nil, Cons>;

private:
  // DATA
  variant_t v_;

  // ...
};
```

Pattern matching in functional C++

```
Fixpoint append
      {A : Type}
      (l m : list A) : list A :=
 match l with
 | [] => m
 | x :: l1 => x :: append l1 m
 end.
```

```
template <typename A> struct List {
 // ...

 List<A> app(List<A> l2) const {
  if (std::holds_alternative<Nil>(this->v())) {
   return l2;
  } else {
   const auto &[a0, a1] = std::get<Cons>(this->v());
   return List<A>::cons(a0, a1->app(std::move(l2)));
  }
 }


 // ...
};
```

Higher-order functions in functional C++

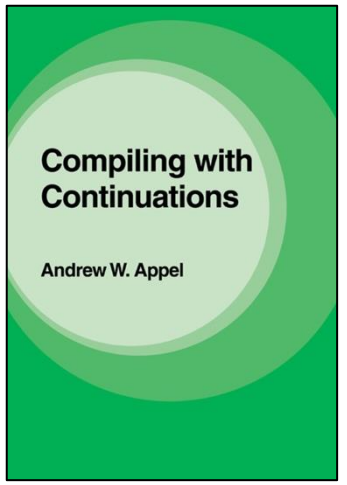
```
Fixpoint map
      {A B : Type}
      (f : A -> B)
      (l : list A) : list B :=
 match l with
 | [] => []
 | x :: l' => f x :: map f l'
 end.
```

```
template <typename A> struct List {
 // ...

 template <typename B, typename F0>
  requires std::is_invocable_r_v<B, F0 &, A &>
 List<B> map(F0 &&f) const {
  if (std::holds_alternative<Nil>(this->v())) {
   return List<B>::nil();
  } else {
   const auto &[a0, a1] = std::get<Cons>(this->v());
   return List<B>::cons(f(a0), a1->map<B>(f));
  }
 }

 // ...
};
```

Loopification

- A **composition** of different transformations:
  - Continuation-passing style
    - Selective: **only** at recursive call sites!
  - Defunctionalization
    - Put frames into an explicit **stack** data structure
  - Tail call optimization
    - No stack pushing and popping
  - Tail modulo cons
    - Special but **common** case in functional programming

- The main challenge is to keep the resulting code **typeable** and **readable**.



Defunctionalization at Work *

Olivier Danvy and Lasse R. Nielsen
BRICS †
Department of Computer Science
University of Aarhus ‡

June, 2001

Abstract

Reynolds's defunctionalization technique is a whole-program transformation from higher-order to first-order functional programs. We study practical applications of this transformation and uncover new connections between seemingly unrelated higher-order and first-order specifications and between their correctness proofs. Defunctionalization therefore appears both as a springboard for revealing new connections and as a bridge for transferring existing results between the first-order world and the higher-order world.

Tail Modulo Cons

Frédéric Bour[1,2], Basile Clément[1], and Gabriel Scherer[1]

[1] INRIA
[2] Tarides

Abstract

OCaml function calls consume space on the system stack. Operating systems set default limits on the stack space which are much lower than the available memory. If a program runs out of stack space, they get the dreaded "Stack Overflow" exception – they crash. As a result, OCaml programmers have to be careful, when they write recursive functions, to remain in the so-called *tail-recursive* fragment, using *tail* calls that do not consume stack space.

Loopification with tail modulo cons

```
Fixpoint map
      {A B : Type}
      (f : A -> B)
      (l : list A) : list B :=
 match l with
 | [] => []
 | x :: l' => f x :: map f l'
 end.
```

```
template <typename A> struct List {

 template <typename B, typename F0>
  requires std::is_invocable_r_v<B, F0 &, A &>
 List<B> map(F0 &&f) const {
  std::unique_ptr<List<B>> _head{};
  std::unique_ptr<List<B>> *_write = &_head;
  const List *_loop_self = this;
  while (true) {
   auto &&_sv = *_loop_self;
   if (std::holds_alternative<Nil>(_sv.v())) {
    *_write = std::make_unique(List<T1>::nil());
    break;
   } else {
    const auto &[a0, a1] = std::get<Cons>(_sv.v());
    auto _cell =
      std::make_unique(List<B>::Cons(f(a0), nullptr));
    *_write = std::move(_cell);
    _write = &std::get<Cons>((*_write)->v_mut()).l;
    _loop_self = a1.get();
    continue;
   }
  }
  return std::move(*_head);
 }

 // ...
```

Loopification with tail call optimization

```
Fixpoint is_prefix_of
       (l1 l2 : list nat) : bool :=
 match l1, l2 with
 | [], _ => true
 | _ :: _, [] => false
 | x :: xs, y :: ys =>
  if Nat.eqb x y
  then is_prefix_of xs ys
  else false
 end.
```

```
bool is_prefix_of(const List<uint64_t> &l1,
                  const List<uint64_t> &l2) {
 const List<uint64_t> *_loop_l1 = &l1;
 const List<uint64_t> *_loop_l2 = &l2;
 while (true) {
  if (std::holds_alternative<Nil>(_loop_l1->v())) {
   return true;
  } else {
   const auto &[a0, a1] =
       std::get<Cons>(_loop_l1->v());
   if (std::holds_alternative<Nil>(_loop_l2->v())) {
    return false;
   } else {
    const auto &[a00, a10] =
        std::get<Cons>(_loop_l2->v());
    if (a0 == a00) {
     _loop_l1 = a1.get();
     _loop_l2 = a10.get();
    } else {
     return false;
    }
   }
  }
 }
}
```

Loopification with frames

```
Fixpoint length
      {A : Type} (l : list A) : nat :=
 match l with
 | [] => 0
 | x :: xs => 1 + length xs
 end.
```

```
template <typename A> struct List {

 uint64_t length() const {
  const List *_self = this;

  struct _Enter {
   const List *_self;
  };

  struct _Resume_Cons {
    uint64_t _s0;
  };

  using _Frame = std::variant<_Enter, _Resume_Cons>;
  uint64_t _result{};
  std::vector<_Frame> _stack;
  _stack.reserve(8);
  _stack.emplace_back(_Enter{_self});
  while (!_stack.empty()) {
    _Frame _frame = std::move(_stack.back());
    _stack.pop_back();
    if (std::holds_alternative<_Enter>(_frame)) {
      auto _f = std::move(std::get<_Enter>(_frame));
      const List *_self = _f._self;
      auto &&_sv = *_self;
      if (std::holds_alternative<Nil>(_sv.v())) {
        _result = UINT64_C(0);
      } else {
        const auto &[a0, a1] = std::get<Cons>(_sv.v());
        _stack.emplace_back(_Resume_Cons{UINT64_C(1)});
        _stack.emplace_back(_Enter{a1.get()});
      }
    } else {
      auto _f = std::move(std::get<_Resume_Cons>(_frame));
      _result = (_f._s0 + _result);
    }
  }
  return _result;
 }
```

Other interesting tidbits in translating Rocq to C++

Conceptual tidbits

- Rocq type classes and module types become C++ concepts (C++20).
- Rocq type class instances and modules become C++ structs.
- Dependent types become std::any with casts at boundaries

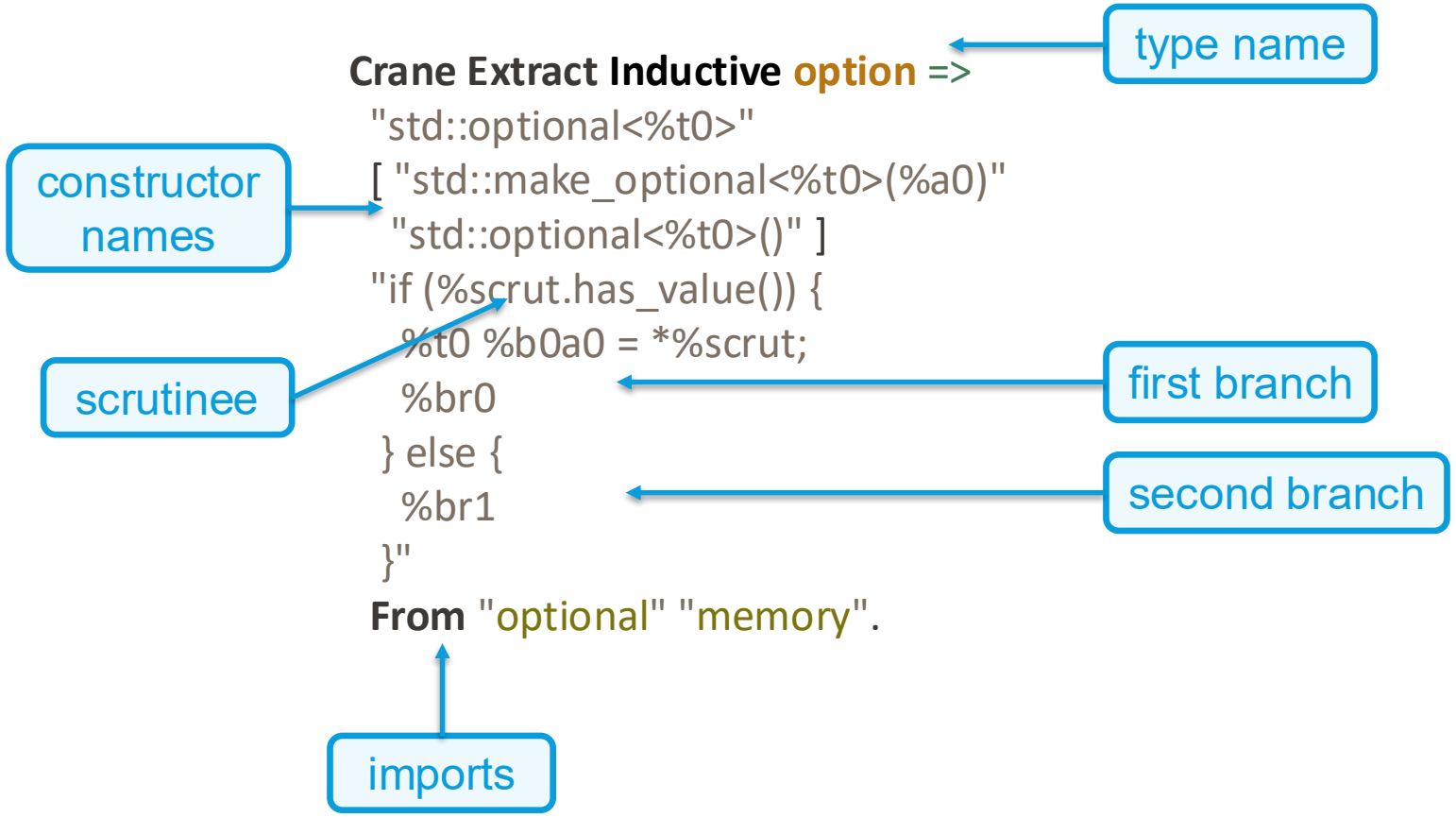
Optimization tidbits

- std::move at last use.
- All-nullary inductives become enum class.
- Single-constructor inductives become flat structs.

Custom mappings from Rocq to C++

```
Inductive option (A : Type) : Type :=
| Some : A -> option A
| None : option A.
```

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



Expressing effectful programs

- Rocq is a **purely** functional language:
  no IO, no state, no concurrency, just functions manipulating values.

- But… purely functional languages can represent effectful computation using **monads**!

- The Crane-preferred way of doing this is through **interaction trees**.

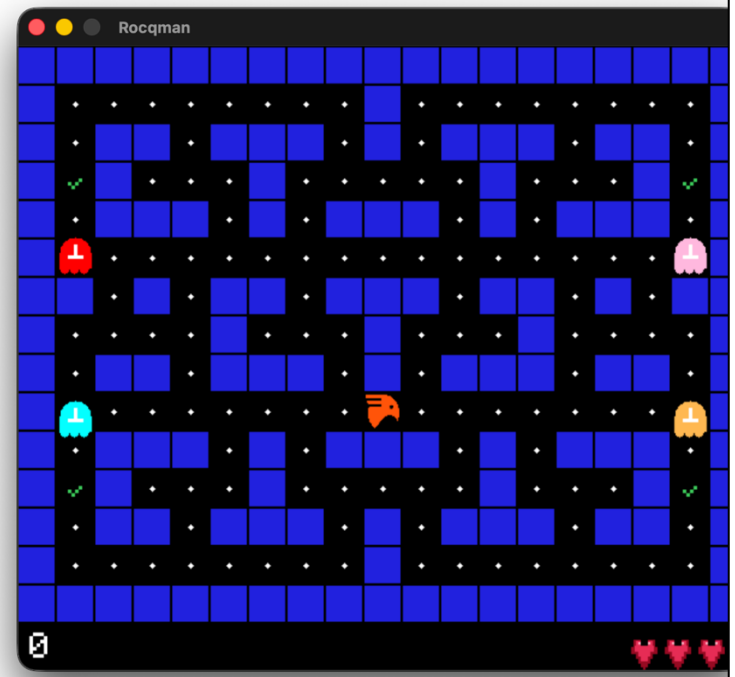
Expressing effectful programs

- The users can now write **effectful** programs in Rocq…

```
Definition test (s : string) : itree ioE unit :=
  print ("printing " ++ s) ;;
  ret tt.
```

- and extract them to C++!

```
void test(const std::string s) {
    std::cout << "printing " + s;
    return;
}
```



https://github.com/joom/rocqman

A Rose Tree Is Blooming (Proof Pearl)

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

Abstract

Game trees are a fundamental mathematical abstraction for analyzing games, often implemented as rose trees in functional programming. We can construct rose trees by starting from an initial state and iteratively applying the function that provides the states one move away, an approach known as an *anamorphism* (or colloquially, an *unfold*).

In ML or Haskell, finite and infinite rose trees share the same type, but proof assistants typically distinguish them: finite trees as inductive, infinite ones as coinductive. With the inductive approach, defining the unfold function is tricky but we obtain a finite tree that is easy to compute with and easy to reason about. With the coinductive approach, defining the unfold function is easy but we obtain a potentially infinite tree, which is harder to reason about since we must resort to proof techniques like bisimulation. In this paper, we compare the inductive and coinductive approaches to game trees, implement unfold functions for both, and prove the soundness and completeness of both unfold functions (with respect to the game) in the Rocq Prover. We illustrate these techniques with implementations of a tic-tac-toe game and a simple SAT solver.

CCS Concepts: • **Software and its engineering** → **Formal software verification**; **Functional languages**; • **Computing methodologies** → Game tree search.

Keywords: formal verification, termination, recursion, corecursion, game trees, Rocq

ACM Reference Format:
Joomy Korkut. 2026. A Rose Tree Is Blooming (Proof Pearl). In *Proceedings of the 15th ACM SIGPLAN International Conference on Certified Programs and Proofs (CPP '26), January 12–13, 2026, Rennes, France.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3779031.3779091

CPP '26, Rennes, France

ACM ISBN 979-8-4007-2341-4/2026/01
https://doi.org/10.1145/3779031.3779091

1 Introduction

Game trees are a useful mathematical abstraction for reasoning about pathfinding, constraint satisfaction, and games [1]. A game tree consists of the states of a game organized as a tree, where each branch leads to a state after a move. A complete game tree is one that contains all possible game states. Such trees can be used for global reasoning about future moves, such as deterministically computing the best possible move at a given point in the game.

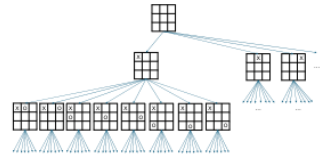
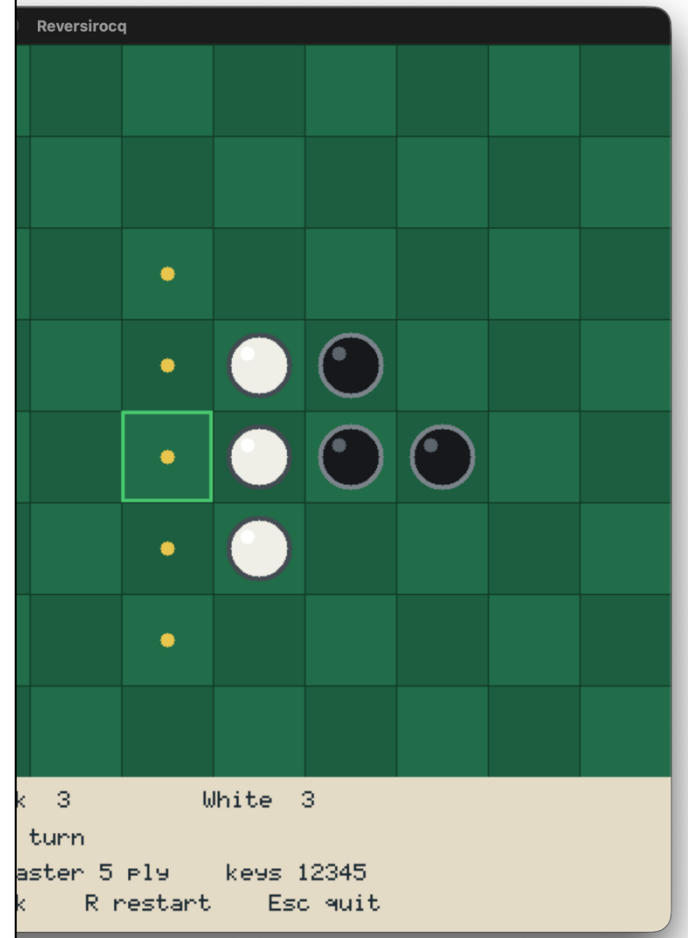


Figure 1. An example game tree for tic-tac-toe.

An example game tree for tic-tac-toe can be seen in Figure 1. At the root node, we start with the empty board, which is the initial game state. First it is **x**'s turn, so we consider all possible moves **x** can make. There are 9 empty cells on the board **x** could choose, so the root node has 9 immediate subnodes. At each of the subnodes, it is now **o**'s turn. There are 8 empty cells left on the board at any of these nodes, therefore they all should have 8 immediate subnodes.

We omit most branches of the tic-tac-toe game tree here for brevity. The complete tree would have 549,946 nodes if we expand naively, and that is if we were to stop building the game tree when a player wins before all cells are filled. The lesson to learn here is that game trees can grow exponentially, and that computing game trees is hard! For games with a large number of possible game states, it may even be infeasible. Famously, the chess games of IBM's Deep Blue against grandmaster and then long-time world champion Garry Kasparov exemplified the difficulty of computing game trees and searching for the best move [27, 46].

The straightforward implementation of game trees as a functional data structure coincides with what the functional programming community calls *rose trees*: trees where nodes can have a list of subtrees. This type has been the subject of considerable research, both in the functional programming community [35, 21, 23, 22, 6, 25], due to its challenges

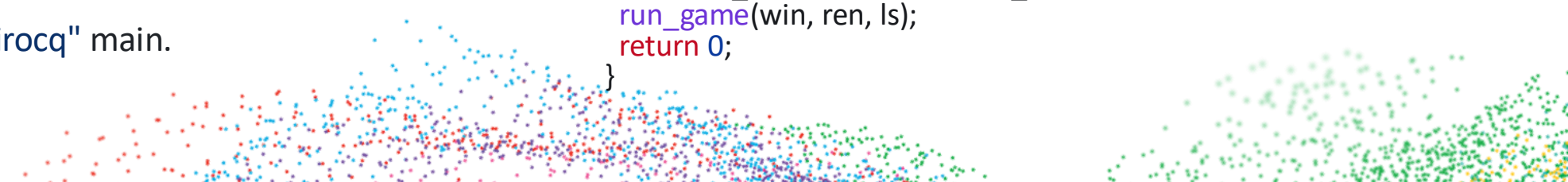


https://github.com/joom/reversirocq

Writing programs with Crane

```
(** Main recursive SDL game loop. *)
CoFixpoint run_game
        (win : sdl_window) (ren : sdl_renderer)
         (ls : loop_state) : itree sdlE unit :=
 res <- process_frame ren ls ;;
 let '(quit, ls') := res in
 if quit
   then exit_game win ren
   else Tau (run_game win ren ls').

(** Program entry point used by extraction. *)
Definition main : itree sdlE c_int :=
 init <- init_game ;;
 let '(win_ren, ls) := init in
 let '(win, ren) := win_ren in
 run_game win ren ls ;;
 Ret c_zero.

Crane Extraction "reversirocq" main.
```

```
void run_game(const sdl_window win, const sdl_renderer ren,
          const Reversirocq::loop_state &ls) {
  std::pair<bool, Reversirocq::loop_state> res =
     Reversirocq::process_frame(ren, ls);
  const bool &quit = res.first;
  const Reversirocq::loop_state &ls_ = res.second;
  if (quit) {
    exit_game(win, ren);
    return;
  } else {
    run_game(win, ren, ls_);
    return;
  }
}


int main() {
  std::pair<std::pair<sdl_window, sdl_renderer>, Reversirocq::loop_state> i
=
     Reversirocq::init_game();
  const std::pair<sdl_window, sdl_renderer> &win_ren =
    init.first;
  const Reversirocq::loop_state &ls = init.second;
  const sdl_window &win = win_ren.first;
  const sdl_renderer &ren = win_ren.second;
  run_game(win, ren, ls);
  return 0;
}
```

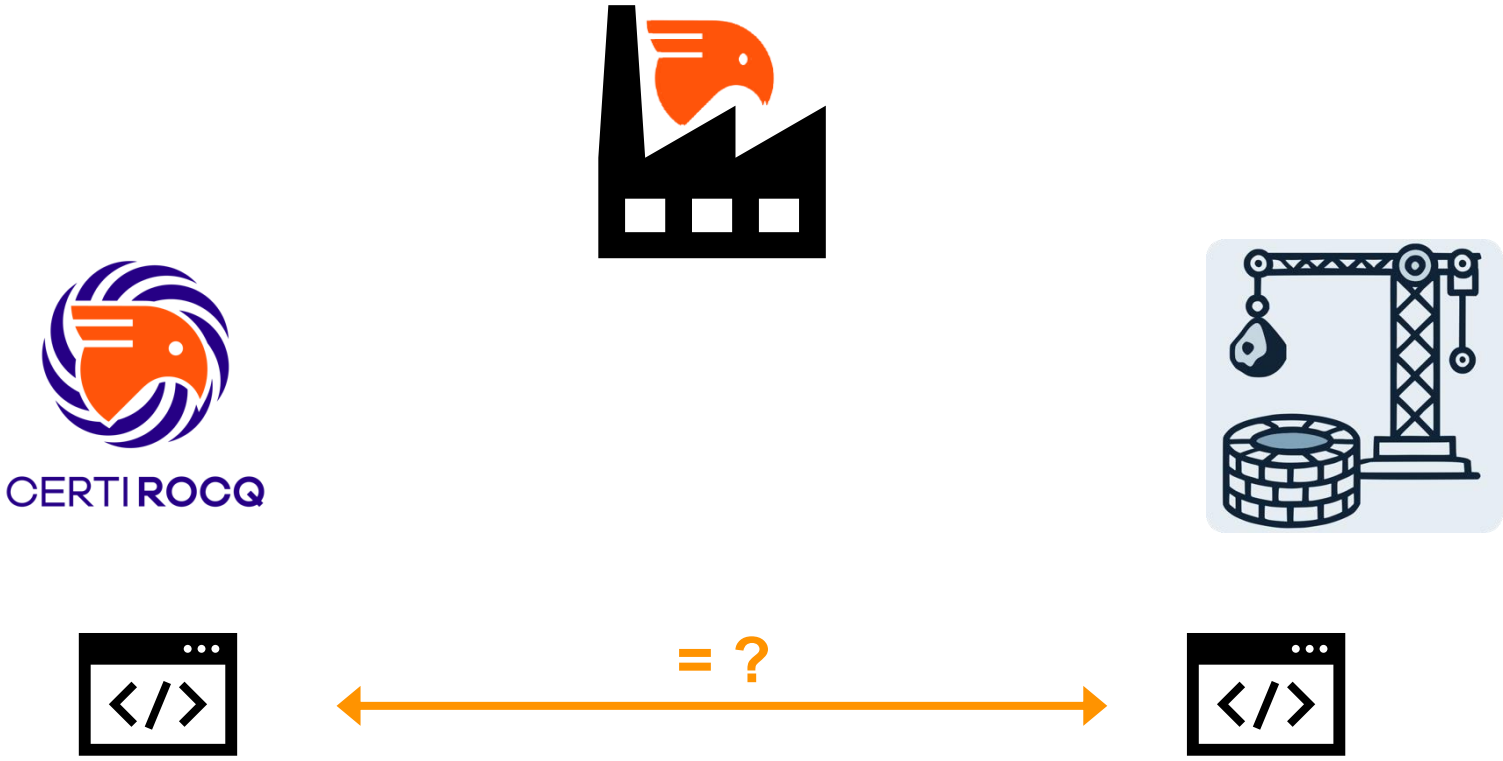


Is Crane itself verified?

- This is **lightweight formal methods**: our extractor is **not** verified.
- Generated code will also undergo the same extensive **testing** and **static analysis** all handwritten code at Bloomberg is subject to.
- We also **attack** it with AI agents and try to exploit Crane-generated code to find safety issues. We have already **found** and **fixed** some.

Is Crane itself verified?

- We are developing a **differential testing** framework to compare Crane and other compilers and extractors like CertiRocq.



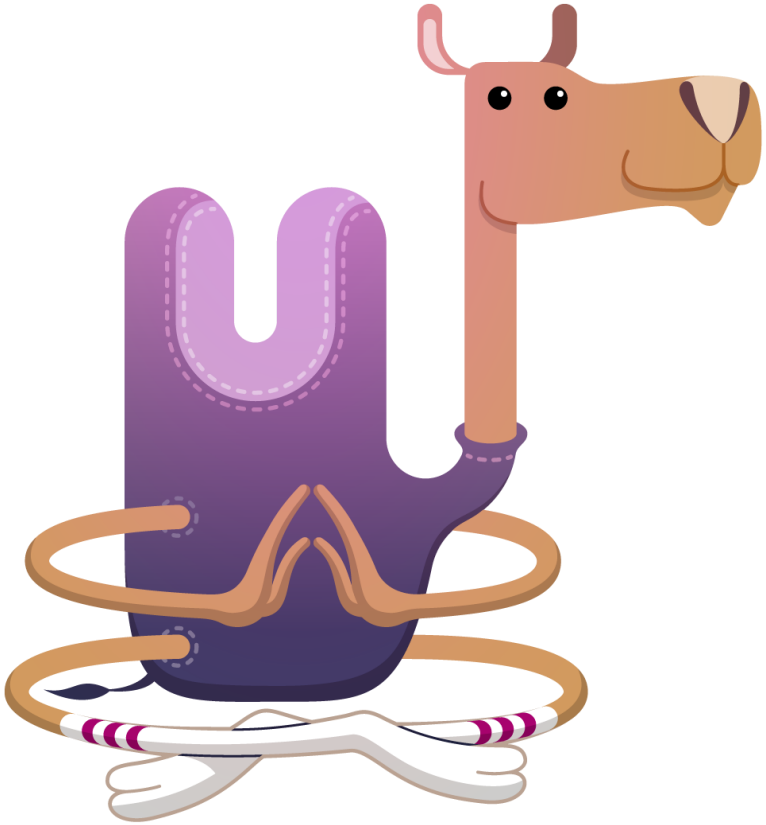
Future work

- Improve the **efficiency** of generated code without sacrificing readability *too much.*
- Extract C++ code that runs **faster** than extracted OCaml code! (partially there!)
- Get Crane-generated code into **production**!

A note of appreciation

- Implementing Crane in OCaml was a **necessity** rather than a choice:
  - Rocq plugins can only be written in OCaml.

- Over the last year, I got convinced that OCaml is actually a **great** choice!
  - AI agents work **faster** with a fast feedback loop!



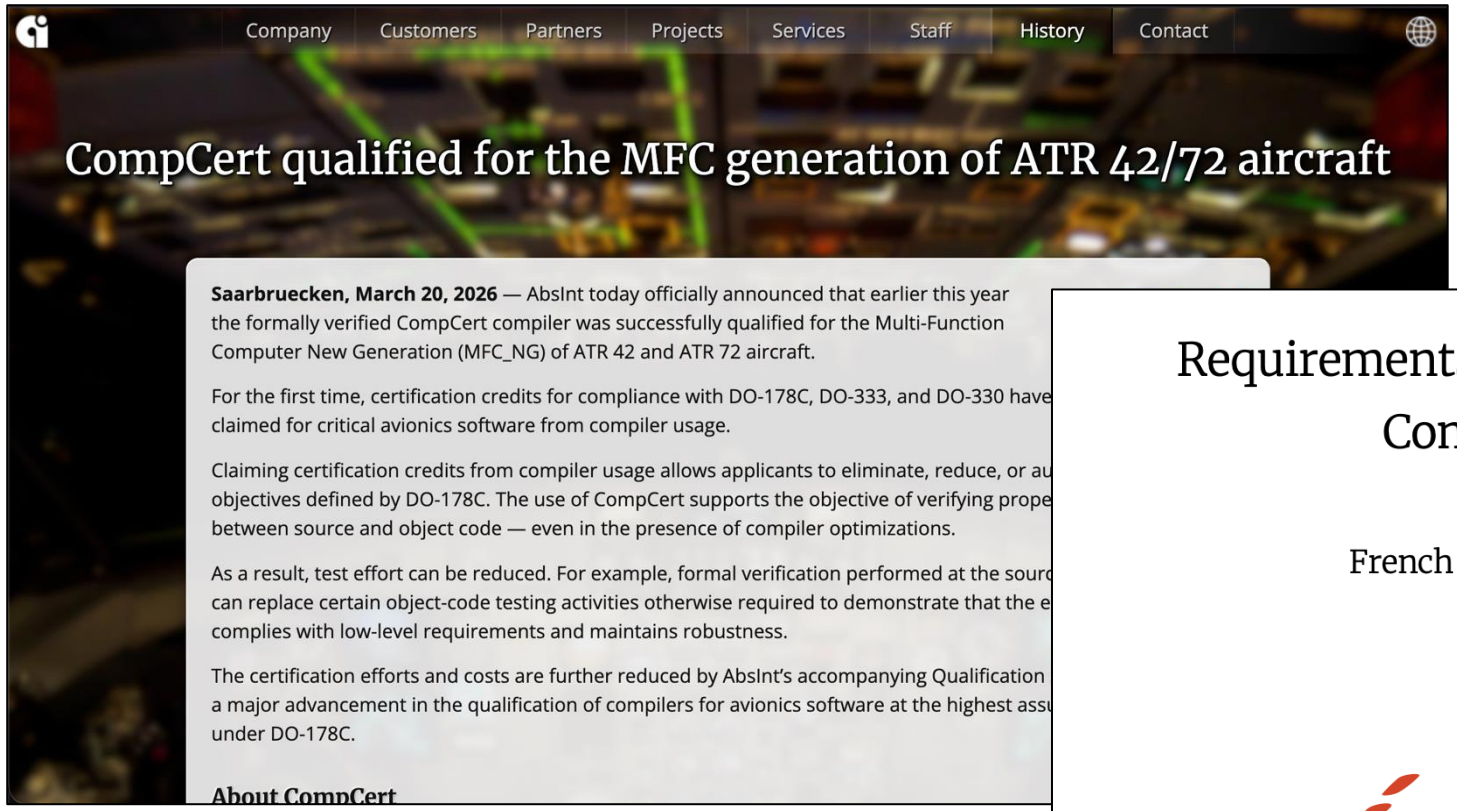
Why Rocq is better than Lean for program verification

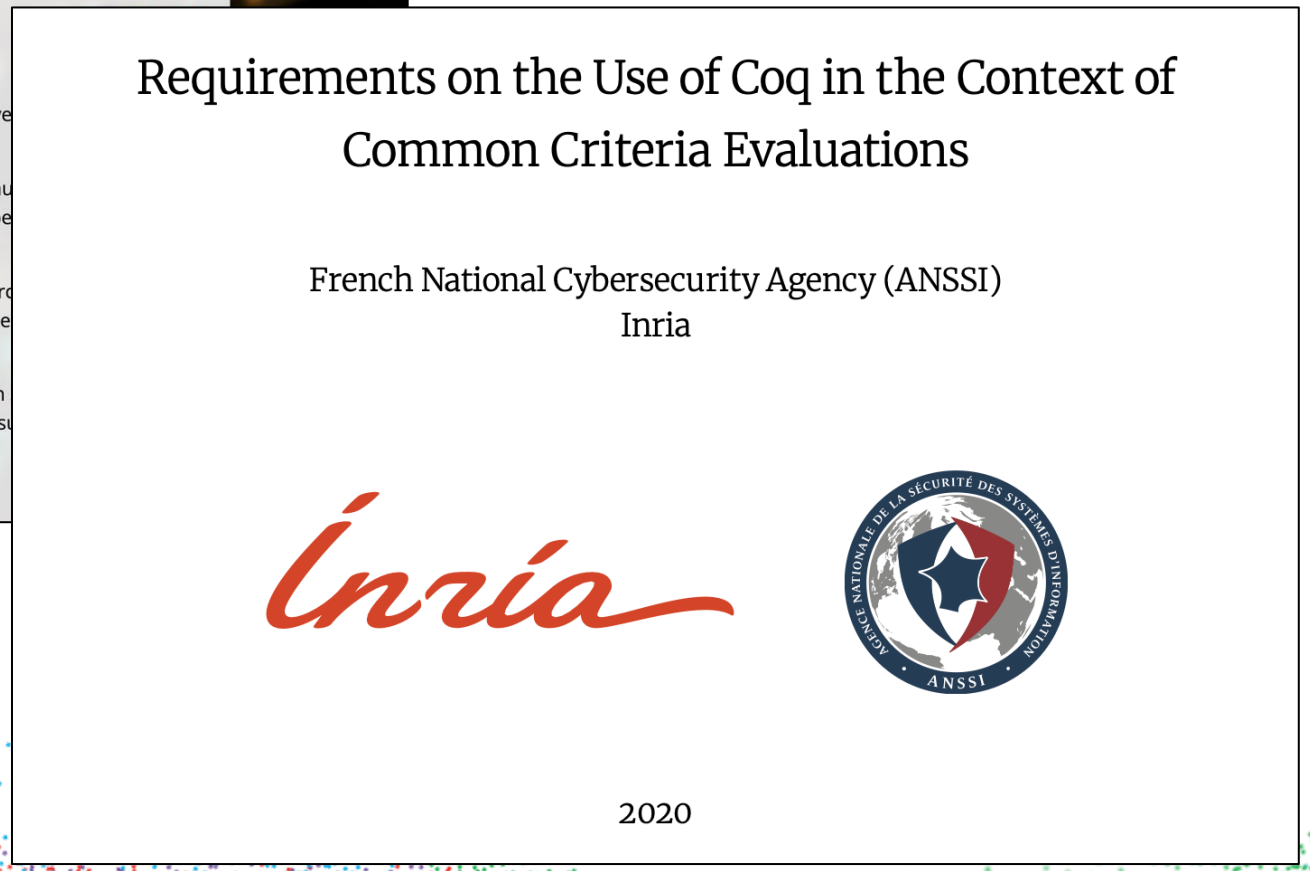
- Language-level issues:
  - —Coinductive types and cofixpoints exist as a library in Lean, but not ready for use.
    Rocq coinductives are not perfect, but there are solutions.
  - —Nested inductive types and predicates are less expressive in Lean.

- Extraction:
  - —Lean is opinionated in its extraction.
  - —Lean's extraction is unverified and unspecified.
  - —Rocq has **many** extraction / compilation options!

- Missing parts in the Lean ecosystem:
  - —Abstractions: *Interaction trees, choice trees*
  - —Program verification frameworks: *Iris, VST, BRiCk, Frama-C, Why3*
  - —Formal definitions of semantics: *CompCert, Cerberus, WasmCert, JSCert*
  - —Lighter-weight program verification: *hs-to-coq, rocq-of-ocaml, rocq-of-python, rocq-of-rust*
  - —Program synthesis: *FiatCrypto*
  - —Verified lexing and parsing: *Narcissus, Verbatim, CoStar*

Why Rocq is better than Lean for program verification





(with thanks to Meven Lennon-Bertrand)

Why Rocq is better than Lean for program verification

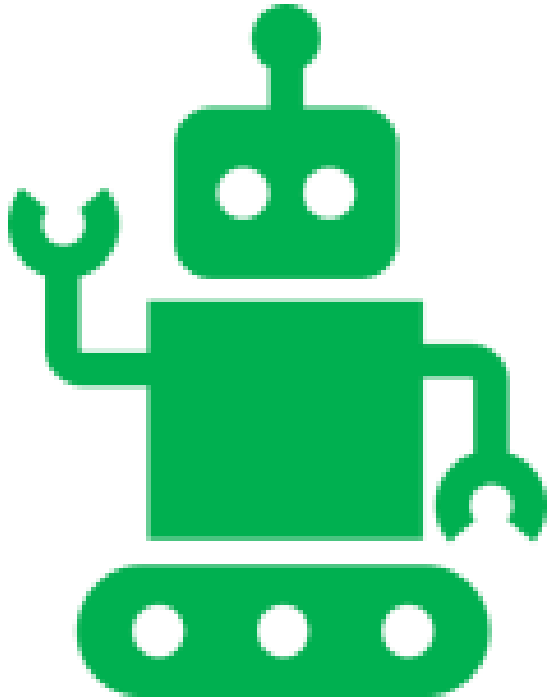
as good as

- AI agents are **as good as** writing Rocq as they are writing Lean.

  —There is a lot of Rocq out there, written since late 80s.

  —AI models do **well** now with languages they have **not** seen before.

  —Feel free to try new languages, DSLs, libraries!
  AI agents can learn them and get up to speed with them.

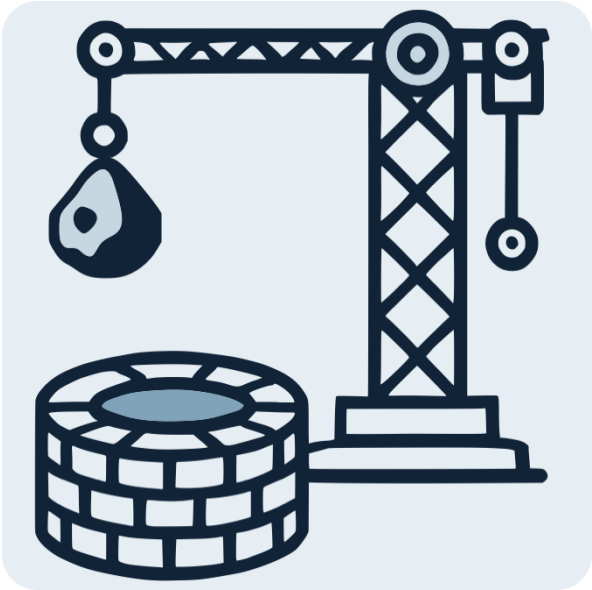


Summary

- Crane is a new extraction system from Rocq to C++, that generates **functional-style**, **memory-safe**, **thread-safe**, **readable** C++ code.

- Using Crane, we are implementing verified libraries in C++ that will be used by Bloomberg engineers in the future.

- Formal verification now is **much cheaper** than it used to be, but we still don’t have a good way of bringing verified programs to production. **We’re fixing that!**



Thank you!

Check out our project at:
https://github.com/bloomberg/crane

Bloomberg

TechAtBloomberg.com