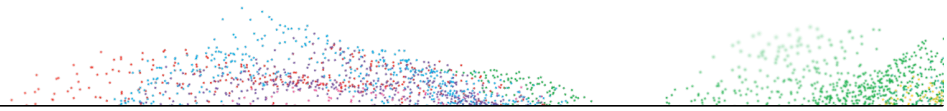
Hi everyone. I'm Joomy Korkut. I'm a researcher at Bloomberg's CTO Infrastructure and Security Research team.

Today I'm gonna talk about the flagship project of our certified programming research program, namely Crane. And particularly about how we are trying to sneak verified programs into Bloomberg software. But why do we care about verified programs so much? I'm sorry if this part is a bit like preaching to the choir.

Testing is great, but not enough!

```cpp
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.



Our main motivation is that **testing is great, but not enough**!

Here is a small JSON string parser in C++. I intentionally wrote it to look reasonable. It checks that the string starts with a quote. It bounds-checks before consuming an escape character. It handles the common JSON escape sequences. It returns nullopt if the string is unterminated.

There is no obvious crash here. This is exactly the sort of code that could pass a large suite of tests over valid JSON strings. But there is still a bug!

<click> The bug is the default case.

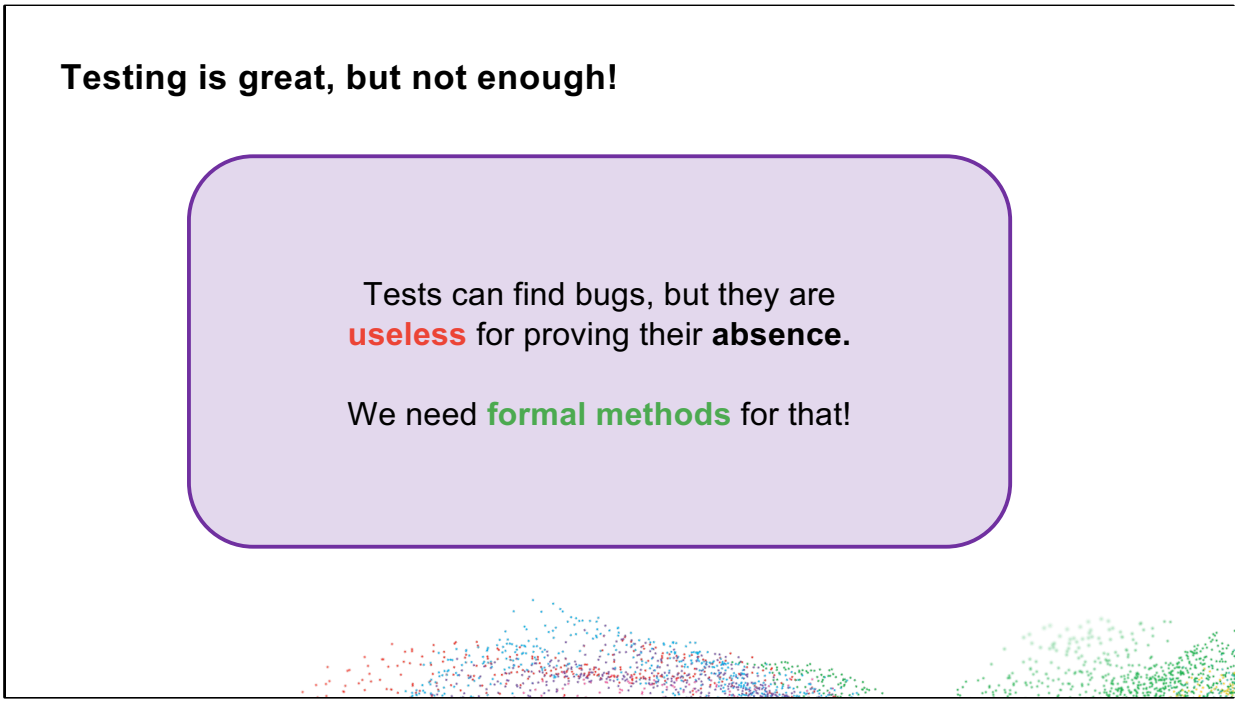
If the input contains backslash-q, or backslash-x, or backslash-a, this parser silently accepts it even though those are not valid JSON escape sequences.

This is not a memory safety bug, not a stack overflow, not a sanitizer finding. **It is a language-recognition bug!**

The implementation accepts a different grammar than the grammar defined in

the JSON standard.

And that matters because parser differentials usually come from exactly this kind of disagreement! One component validates according to one language; another component parses according to a slightly different language. An attacker aims at the gap between the two.



The deeper lesson is: tests over valid examples are not enough. If you do not have a formal definition of the language, your tests do not have a complete oracle for what should be rejected.

In other words, tests can find bugs, but they are **useless** for proving their **absence.** We need **formal methods** for that!

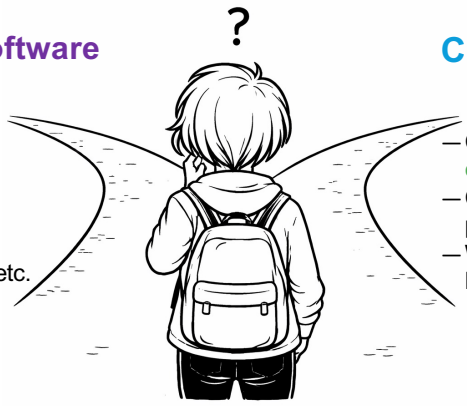
Here is another salient example from JSON land: duplicate keys.

Different JSON libraries and systems make different choices. Some reject duplicates. Some keep the first value. Some keep the last value. Some preserve all pairs.

The danger is not merely that one behavior is right and the others are wrong. The danger is disagreement. If one component validates a message under one interpretation and another component consumes it under another interpretation, an attacker can exploit the difference.

A formal specification forces us to choose. We should specify which kind of JSON we mean, and we should make sure that our parser implementation respects that specification. And by make sure, of course, I mean software verification.

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.

<but> which way?

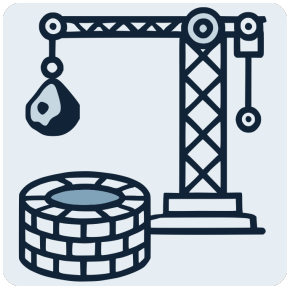
<click> One option is to verify existing software with various toolchains. I listed some here for C, Rust, and other languages. This approach is generally fine, but these tools are often clunky because the languages they operate on have complicated semantics, the proofs can be brittle, and they often sacrifice expressivity for automation.

<click> The other option is certified programming — create new software that is correct by construction! We implement our program along with its specification and correctness proofs, in a language that can express both. Then we compile or extract it to the implementation language. Languages like Rocq, Lean, Agda, and Isabelle support this. My choice here is Rocq, and I'll get into the reasons later.

But the main question for us was: how do we make certified programming 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*

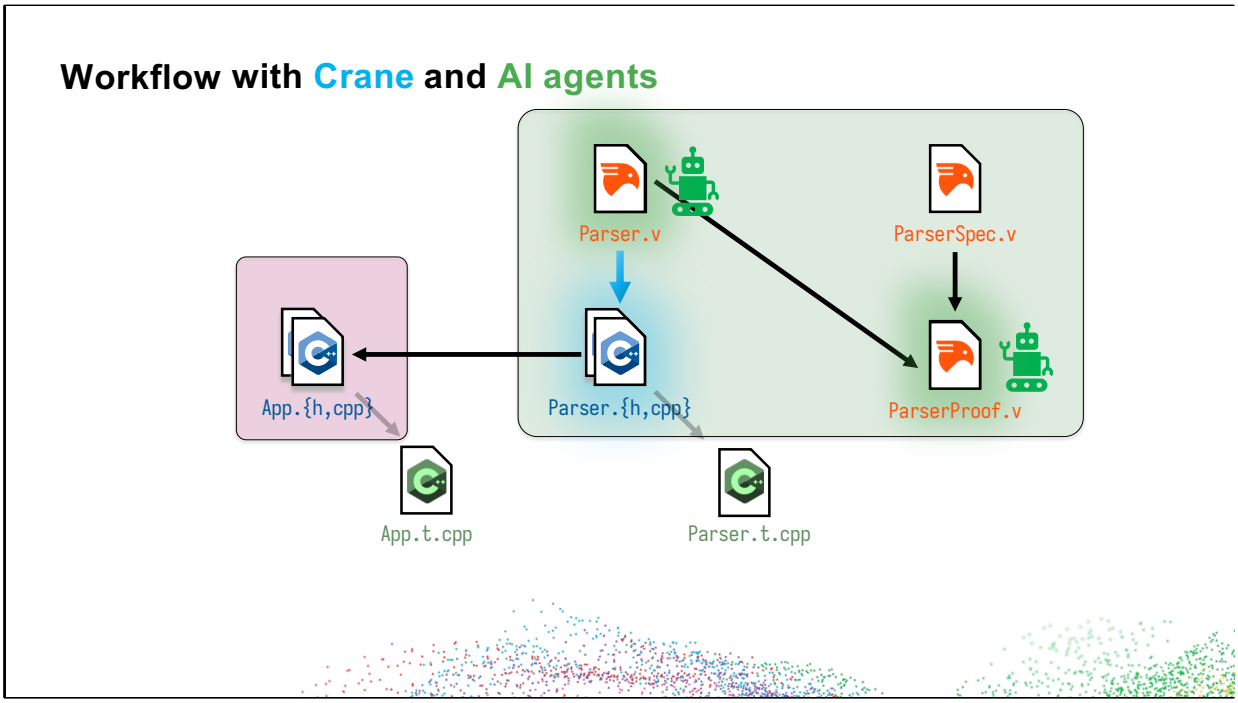
  C++ code.

- Still under active development!

Our solution to that is Crane, a new extraction system from Rocq to C++. Crane generates mostly functional-style, memory-safe, thread-safe, and very importantly, **readable** C++ code from Rocq code. Not bytecode, not C with void pointers, we want to generate modern C++ that engineers can read and audit.

Though I should note
<click> It is still active development, so treat these as if there is an asterisk next to them.
The project is available open source.

Of course, I should note, that Rocq already has extraction to some languages like OCaml and Haskell. We started as a fork of that and diverged quite a bit at this point.



Here's how things work right now. Suppose we have a parser library, written in C++.
<click> We also tests for our library, in a separate file.
<click> Now we can use our library, suppose we have an app that uses our library.
<click> And surely, our app will also have its own tests. This is nice and all but we are not protected against all the pitfalls that we just talked about.

<click> Here's my vision for improving this with Crane.
<click> We write a specification for our parser in the Rocq proof assistant.
<click> We implement our parser in Rocq.
<click> We write a computer-checked proof in Rocq that our parser implementation satisfies its specification.
<click> Crane generates a new parser in C++ from our Rocq implementation. We replace our old parser with the new one.

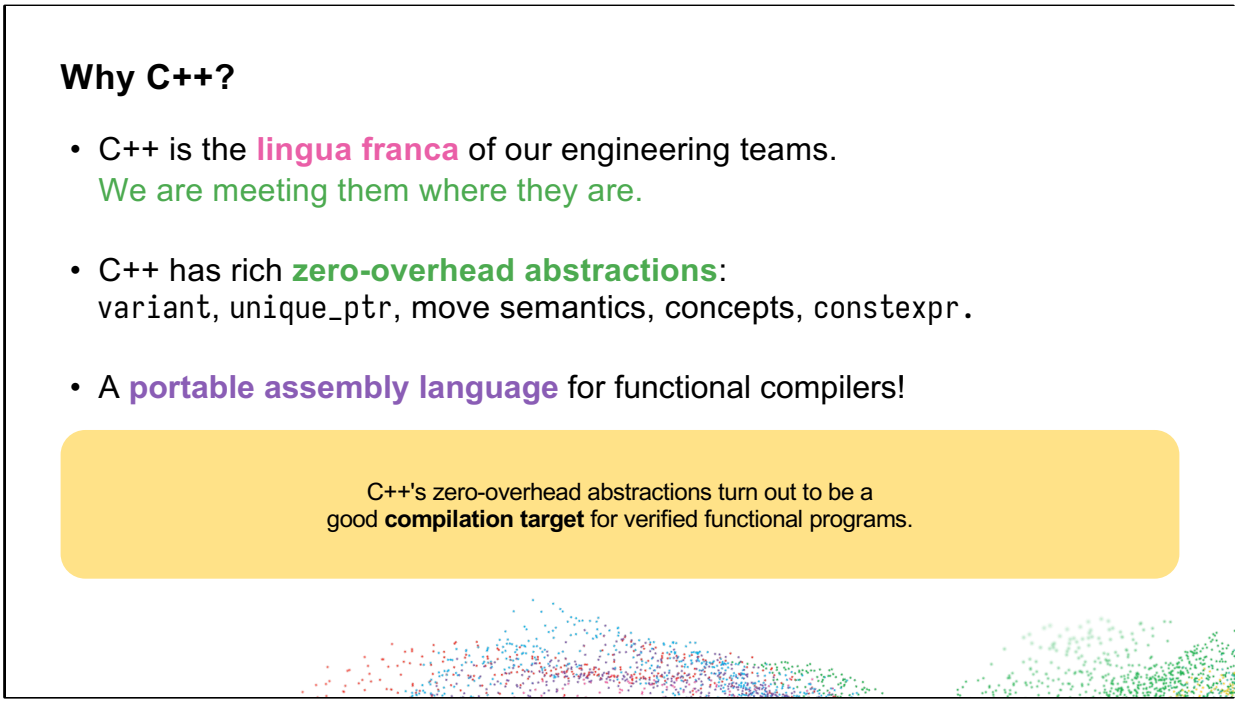
But, formal methods is famously expensive and time consuming. How are we going to get the resources for this?
<click> Well, formal methods is less expensive now, thanks to AI.

<click> We can have AI agents write the proof based on the spec and implementation that we have.
<click> If you're feeling more adventurous, you can even have AI agents write your initial implementation. You can later review it and iterate on it.

Could you also have an AI agent write your spec? Well, yes, but that's a deeper discussion about autoformalization and whether we can trust it, and when we can trust it. For now, I'll stick to writing specs by hand and using AI agents to write the proofs. I want to be more deliberate in what we want our programs to do. Writing the spec by hand helps with that quite a bit.

That being said, later today, Julien will talk about his work on inferring the specs, but that's for a different setting where you already have the code and tests.

I think this also connects well to the earlier keynote today: "Not everything has to be formally verified."
<click> We verify critical components, we verify what we can,
<click> but not everything is formally verified.
Though we're still increasing our assurance in our programs by verifying critical components that are shared by many applications.



Why C++? First, because C++ is the primary programming language and the lingua franca of our engineering teams. We are meeting them where they are. Our goal is to generate readable, verified libraries in C++ for engineers to integrate into production code.

But the deeper reason — the one we discovered along the way — is that modern C++ has rich zero-overhead abstractions. Variants give us tagged unions without virtual dispatch. Unique_ptr gives us ownership without garbage collection. Move semantics give us efficient transfer without copying. Concepts give us type constraints, so more compile-time safety. Constexpr gives us compile-time evaluation for pure functions.

These are nice features for hand-written code but they also make C++ almost a **portable assembly language** for functional compilers! They're expressive enough to generate readable code, and efficient enough for production.

<click> If you take away one thing from this talk, let it be this: the zero-overhead abstractions that C++ engineers already use — variant, unique_ptr, move semantics, concepts — are a good compilation target for verified

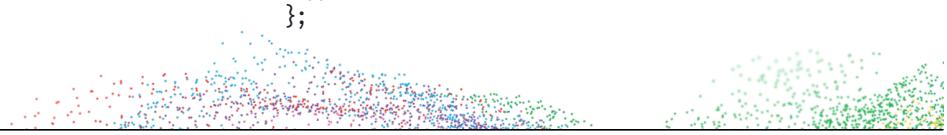
functional programs. The generated code uses exactly the constructs a C++ engineer would reach for by hand.

Let me show you what I mean.

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_;

  // ...
};
```



On the left we have a Rocq definition of linked lists. nil and cons are the constructors. This is like any other functional language.

On the right side we have Crane's translation of it to C++.

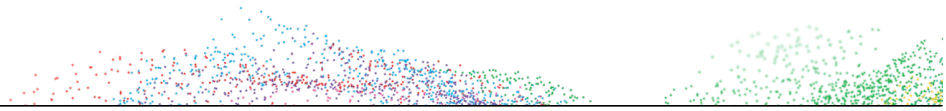
Here are the crucial ideas behind the translation:

1. Each constructor becomes a struct holding the constructor arguments, the type becomes a variant that can choose one of those structs.
2. Parametric polymorphism in Rocq becomes template polymorphism in C++.
3. No manual memory management, we leave that to smart pointers. This is probably our most controversial design choice. This allows us to stay away from maintaining separate Rocq runtime stuff in C++. We depend on zero overhead abstractions to do the memory management for us.

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)));
    }
  }


  // ...
};
```



On the left side we have the append function for lists. This should still be pretty readable to you to anyone familiar with functional programming.

On the right side, we have Crane's translation.

Notice how append is not a static function, it's a method it lives in the same struct as our List type definition.
<click> Notice how the Rocq function on the left takes 2 arguments,
<click> while the C++ function on the right takes one argument.

What would have been the first argument is
<click> just `this`, the object instance, so it doesn't need the first argument, it only needs the second. We also don't do template quantification separately for the method, we just use the one that's outside the struct.

Apart from that, we're just checking the tag of the variant with an if, and acting accordingly.

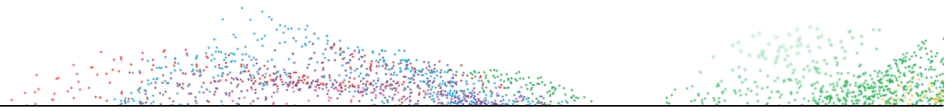
<click> When we have to create a new list, we have smart constructors that

handle the smart pointer business so the smart pointers are abstracted away.

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));
    }
  }

  // ...
};
```



Here is the map function for lists and the C++ Crane generates for it.

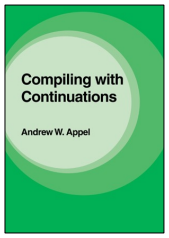
This is pretty similar to what we have for append, but now we have
1. more type parameters! We do another template quantification for the extra ones. Notice how we have a quantification outside the struct, and then another one inside.
2. higher-order functions! There are a few different way to do higher-order functions in C++, function pointers, std::function, etc. These have different upsides or downsides that I don't want to get into here. We get around that by using a forwarding reference (double ampersand), and then using a template constraint that says this thing behaves like a higher-order function when invoked.

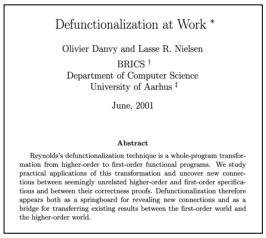
Ok, that's great. But, does anyone see any problem with these append and map functions we produced in C++?
<click> RECURSION is the problem. We have a limited call stack size, and a sufficiently deep chain of recursive calls will fill that and we'll get a stack overflow! So, what do we do?

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**.

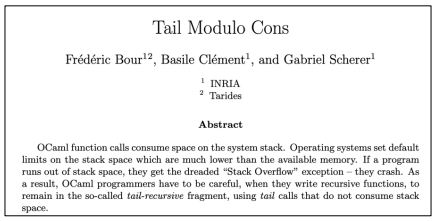



Defunctionalization at Work *

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.

We convert it into a loop. Crane has an extraction pass we call Loopification, which analyzes a Rocq function and decides what sort of optimization to do.

Loopification is a composition of many different ideas from the programming languages literature. If I had to summarize, depending on what sort of function we have:

- It does selective CPS transformation at the recursive sites.
- It does defunctionalization and replicates call frames as explicit objects in a stack data structure.
- It handles tail calls
- It handles tail modulo cons, which is pretty common in functional programming with lists, just like in map and append.

<click> The main challenge is to keep the resulting code **typeable** and **readable**. That is the main design constraint that drove our decisions.

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);
  }
  // ...
```

So, Crane translates the map function we had to a loop. In this case, Crane chooses to apply the **tail modulo cons** optimization. What does that mean?

<click? It means our recursive call is of the form: something CONS recursive call. This doesn't have to be cons literally, it can be any constructor, even multiple constructors.

The OCaml folks came up with an optimization for this form of recursive calls that allows you to do this in constant space for function calls. Here we use the same idea to generate C++ code that runs the same way.
<click> It builds the constructors in-place with a hole left for the next recursive call.

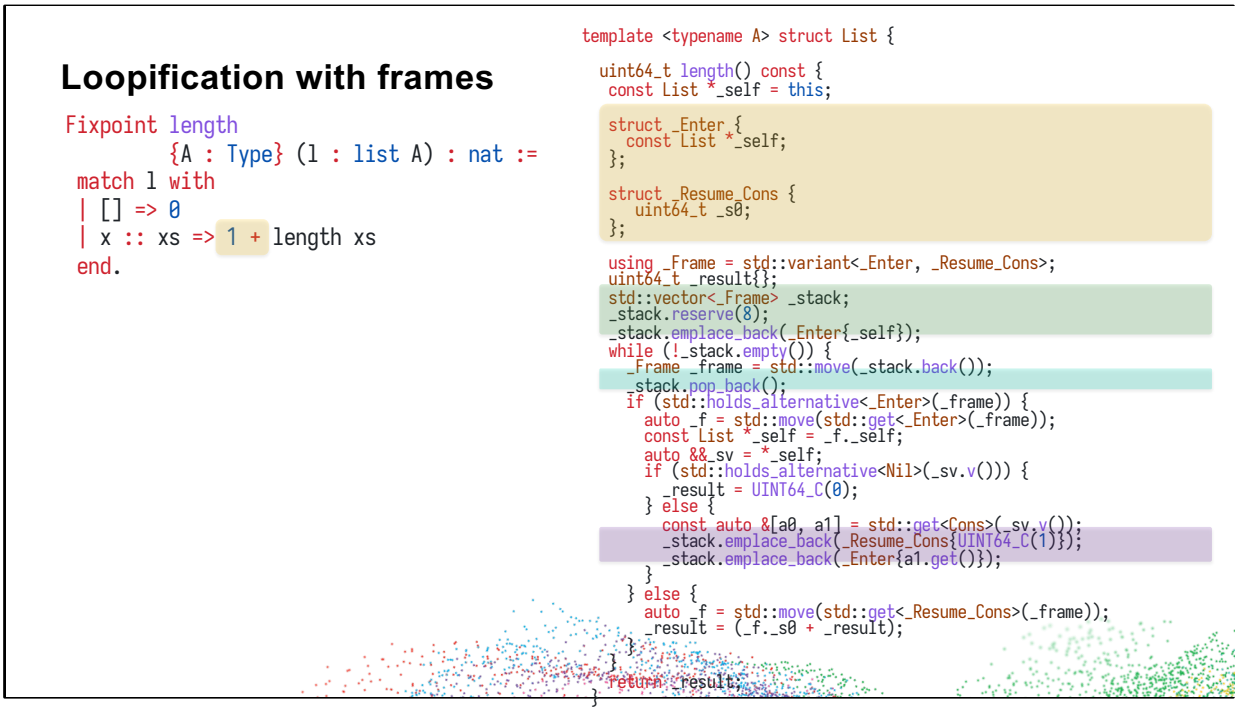
Loopification with tail call optimization

```
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;
      }
    }
  }
 }
}
```

Crane can turn tail calls into loops as well.
<click> the main idea is to hold temporary variables for what the arguments for the recursive calls would be.
<click> where you would have recursive calls, you would update those temporary variable and continue the loop.

Even without this optimization, Clang can **sometimes** optimize tail recursion, but we want to optimize it without having to depend on what Clang chooses. Not to mention, we don't want to force our users to pick higher optimization levels for their code, so we loopify it ourselves. And the resulting loop is pretty readable, in my opinion.



If none of the other optimizations apply, we still have a way of loopifying.

Here we have a list length function written in a less-than-ideal way.
<click> This is why. Ideally you would keep an accumulator as an argument and do it tail recursively. A smarter compiler may be able to optimize this but we're not there yet. So I'll use this function as the simplest example of a frame-based Loopification.

so, imagine we have a function for which we really have to keep things in the call stack. In that case,
<click> we create actual stack frame objects that hold intermediate data
<click> and we store them in a stack data structure. We start with an entry point frame in the stack,
<click> and in every loop, we pop a frame, use the frame content,
<click> and push new frames as we need. This is the same thing that the call stack, but now it is reified into a stack data structure. It is reified. Also, since the stack size is dynamic now, so we won't run into stack overflow issues.

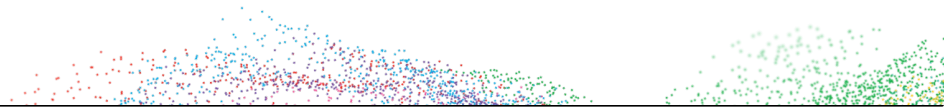
You may ask, what happened to readability? Well, we already write functions

like this (maybe with a bit less administrative noise) in real life. If you were implementing depth-first search in C++, would you write it with recursion? I'll guess no, you'll use a stack data structure and loop until the stack is empty. Well, this is essentially the same thing.
Some of the noise here is unavoidable C++: variants, move semantics, but I'll argue that even in this case this is quite readable. (though we'll keep working on it!)

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.



Instead of showing more C++ code that I have trouble fitting in a slide, I'll just summarize some other ideas in Crane that are worth mentioning.

Conceptually:
We translate Rocq type classes and module types into C++ concepts, and instances or modules that satisfy those classes and types as C++ structs that satisfy those concepts. Usually you can even check this with a static_assert.

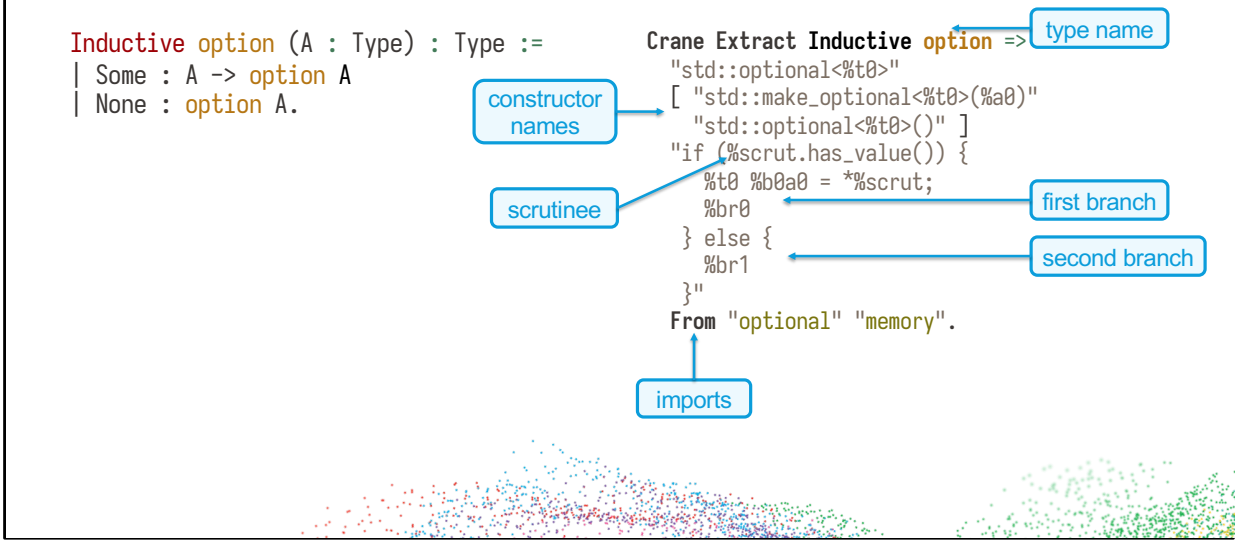
Rocq has dependent types, and C++ doesn't. Now, the preferred way of writing programs in Rocq is to avoid dependent types in programs. You use dependent types in your proofs that are separate from the programs. But still, if you have to use them in your programs, they become the any type in C++, with runtime casts at boundaries.

Optimization-wise:
We take advantage of move semantics whenever we can, particularly by moving values at their last use.

We have some other, simpler optimizations, like inductive types with all nullary

constructors become an **enum class**. Single constructor inductives, and records become flat structs, etc.

Custom mappings from Rocq to C++



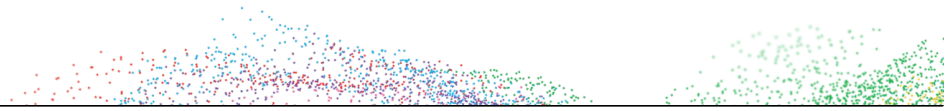
Ok we covered a lot of material already, but how do we interact with the real world? I mean that in more than one way.

In fact, it would have been fair to interrupt me and say “but why would I want to use this list type in C++, I want to use standard vectors, I want to use this and that”. I hear you, and you can!
Just like OCaml extraction, Crane allows users to customize how specific definitions are translated by extraction. You can provide custom mappings for Rocq types and Rocq terms into C++ types and C++ terms. We have a macro language that lets you do that.

Here we see an example for the option type in Rocq.
It is a bit more complicated for types like vector, but this will suffice to show an example mapping.
<click> Here we see the different components of a Rocq type that is mapped into a C++ type.

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**.



In the other sense of interacting with the real world, how do we write effectful programs?

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

<click> But… keep in mind that purely functional languages can represent effectful computation using **monads!**

<click> The Crane-preferred way of doing this is through **interaction trees**. Interaction trees are a variant of something called free monads, in which you can specify different kinds of effects that appear in your program with more granularity. This is a rabbit hole I should stay out of, but they are essential to organizing effects and reasoning about effects.

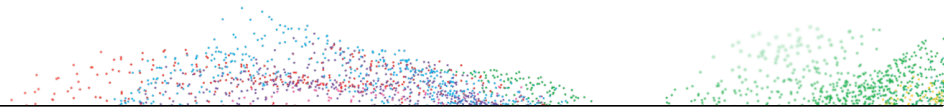
Expressing effectful programs

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

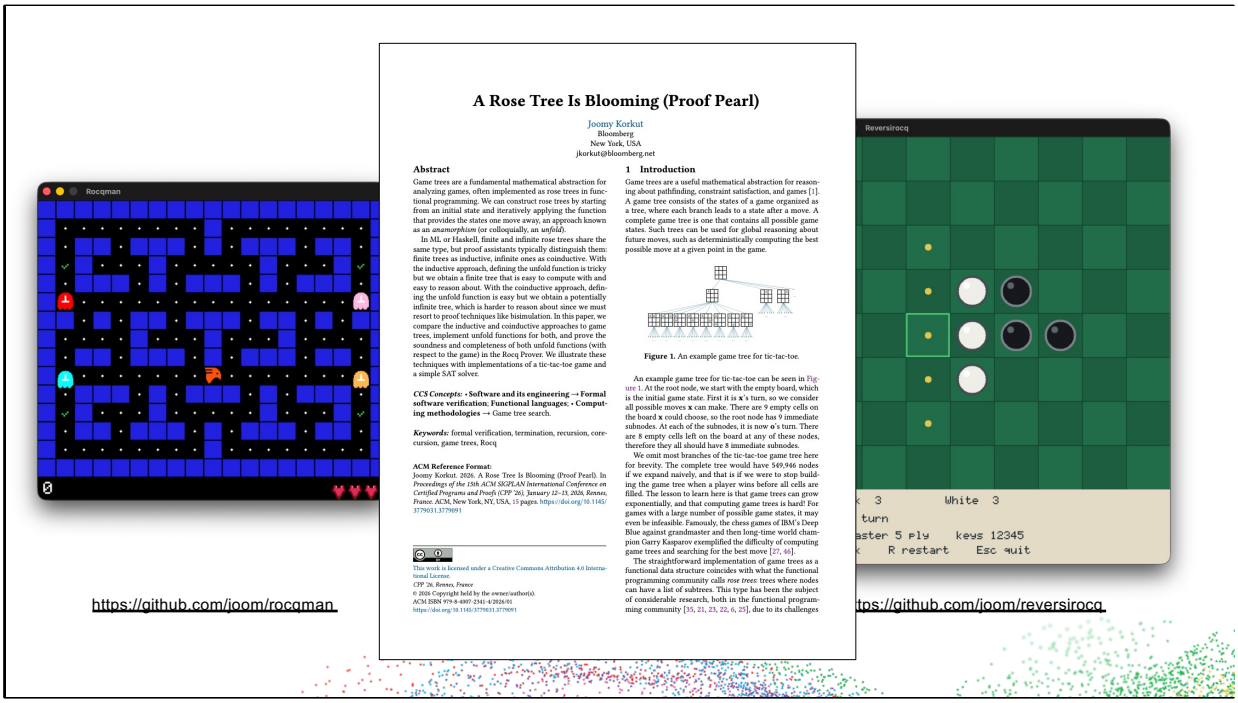
```
Definition test (s : string) : itree ioE unit :=
  print ("printing " ++ s) ;;
  ret tt.
```

- and extract them to C++!

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



Using interaction trees, the users can now write effectful programs in Rocq, as you see here.
<click> and Crane makes sure they are extracted into idiomatic C++.



https://github.com/joom/rocqman

https://github.com/joom/reversirocq

And using this method, I have implemented actual programs! (And by I mean "Claude, Codex and I")

I have written SDL bindings for Crane, and written 3 games with it. A Pacman clone, a Rocqsweeper clone, and a Reversi clone. I have various correctness proofs for these games.

The reversi clone is especially dear to my heart because
<click> it uses some of my prior work on game trees. You play the game against the computer, and the computer uses a coinductive game tree that it builds as needed, and prunes according to the difficulty level. In the paper, I talk about writing functions that build the game tree in a sound and complete way. That is, a game state is in the game tree if and only if it's a valid move in the game.

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.
```

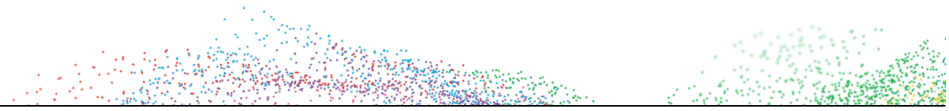
```
void run_game(const sdl_window win, const sdl_renderer ren,
              const Reversirocq::loop_state &ls) {
  std::pair<bool, Reversirocq::loop_state> res =
      Reversirocq::process_frame(ren, ls);
  const bool &quit = res.first;
  const Reversirocq::loop_state &ls_ = res.second;
  if (quit)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

Just for you to get a taste of what the generated code looks like, here's the Rocq code for the Reversi game loop, on the left.

On the right side, you see the version Crane generates for it. This is exactly the generated code, I have not change anything in it to make it pretty. I'm pretty happy with its readability, there is a lot of work behind this to flatten the monadic structure of itrees, and to treat the unit type in Rocq as a void type. But all of that is invisible to the C++ programmers who might want to read the code!

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.



At this point, it is fair to ask me if Crane itself is verified. After all, we spend all this effort making sure that our program is correct. What if there is a bug in Crane? Unfortunately, that is a fair point. But…

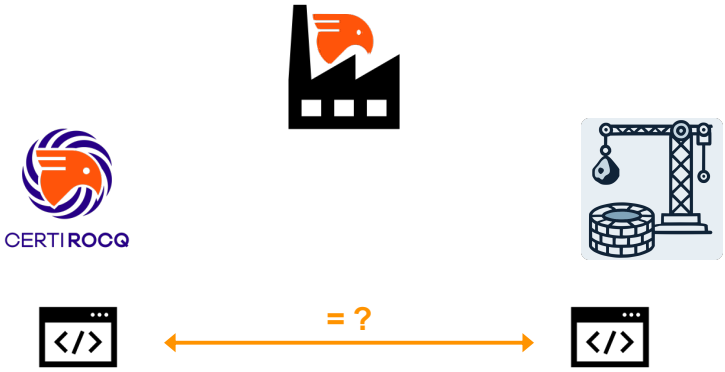
This is **lightweight formal methods**: our extractor is **not** verified. Extraction / compiler verification efforts are feasible especially when you can deal with a simple core language. Our readability concerns force us to generate code in a larger surface language.

However, for further trust in our generated code, we put Crane-generated code under the same extensive **testing** and **static analysis** all handwritten code at Bloomberg is subject to.

We have also found that AI agents are very good at exploiting bugs. They have already revealed many issues with our code generation, and we have fixed most of these issues.

Is Crane itself verified?

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



We are also actively developing a **differential testing** framework to compare (outputs of)
<click> randomly generated Rocq programs compiled with Crane and other compilers and extractors (e.g. CertiRocq).

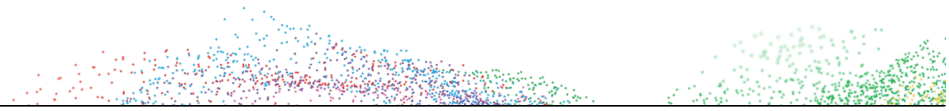
CertiRocq is a verified compiler from Rocq to C. Therefore if we catch any <click> discrepancies with CertiRocq, then we have found a bug in Crane.

My colleague Matthew is driving the effort behind this these days.

There are other extraction methods for Rocq as well. Once this technique is developed, we might also be able to find bugs in those other extraction methods.

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**!



While the basics are done, we have a lot to do, including (but not limited to):

- One aspect of our design that drove our decisions is the trade-off between readability and performance. Optimized code is often less readable. While we want to improve the **efficiency** of generated code, we also want to maintain readability. It's important for us that Bloomberg engineers can read the generated code and convince themselves that the code is correct.
    - One of our intermediate goals is to generate C++ code that runs **faster** than extracted OCaml code! We have varying benchmarks about this. We've had some benchmarks in which we beat the OCaml extraction by 20%, some other benchmarks in which we are behind.

      There is a verified lexer and parser project in Rocq, namely Verbatim and CoStar. I spent the last couple weeks modernizing them and optimizing them. Crane can currently extract and run the parser, but the performance isn't that great. But we'll get there!

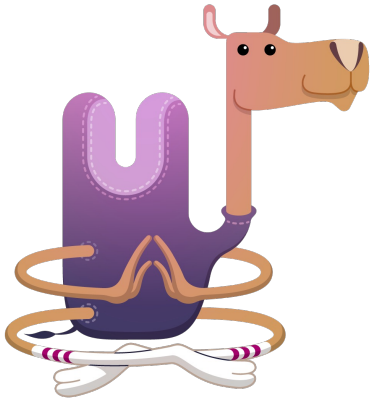
So, I don’t have any numbers for you today, but the performance of Crane generated code is something I care about a lot. Most of our efforts so far have been on covering all of Rocq, and the safety of the generated code. Now we can focus our efforts on performance.

- Our end goal, of course, is to get Crane-generated code into **production**!

Our first target is critical points in the

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!



Before I finish, I should saw a few words about my experience with developing Crane. I've been coding primarily in typed functional languages for the last 13 years. I believe languages with rich type systems will become more and more popular in the next few years, as the type errors provide a natural harness for AI agents to write code. But my language of choice wouldn't have been OCaml; it would have been Haskell.

We didn't exactly pick OCaml. We knew pretty early on, that if we wanted to be able to integrate with Rocq well, we had to do a Rocq plugin. OCaml was the only choice there.

I write a lot of Haskell in my free time, always have. I've spent a lot of my personal time at home on a Haskell project. I barely write any code by hand anymore, I mostly write tests, and I write detailed prompts. Haskell is neat and convenient for writing code, but the cost of this cleverness is that the compile times are long. In comparison, OCaml's compile times are so much shorter! This makes working with AI agents a better experience.

Therefore, I must say, over the last year, I got converted, I'm a reformed man.

OCaml is actually a great choice.

Not to mention, Crane's code generator has pretty decent performance! I've had to spend a lot of time optimizing my Haskell program at home, and I'm still not happy with it, but that's just not an issue with Crane. Our only bottleneck is clang-format, but we don't have a solution to that at the moment.

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*

Especially now with AI and its well publicized success in advances in math, I get a lot of questions on why I use Rocq, why don't I give in to the hype and switch to Lean? So I want to summarize my stance on this, which I think might interest some folks here too.

The slide is provocatively titled "Why Rocq is better than Lean for program verification". I'm not trying to start a flame war. If you're a more mature person than me, you can feel free to say "better fit for use today" instead when you retell these arguments!

Language-level issues:

- Coinductive types and cofixpoints exist as a library, but not ready for use. It has issues with mutually coinductive types.
  Rocq coinductives are not perfect, but there are solutions like Paco, the coinduction plugin ,etc.
- Nested inductive types and predicates are less expressive in Lean. This makes the engineering of programs that have to use them messier. And this is something that I encounter as well. The game tree paper I mentioned earlier, that code would be much uglier in Lean.

Extraction:

- Lean is opinionated in its extraction; it has its own runtime. This is a plus for Lean libraries but if Lean's runtime design doesn't work for you, you're out of luck.
- Lean's extraction is unverified and unspecified.
- In comparison, Rocq has many extraction / compilation options: OCaml (and verified extraction to Malfunction), Haskell, Scheme, Rust, Elm, C (and CertiRocq: a verified compiler to C and WebAssembly), and now C++

Ecosystem:

Missing abstractions: Interaction trees, choice trees

Program verification frameworks: Iris (Iris-Lean is in progress), VST, BRiCk, Frama-C, Why3

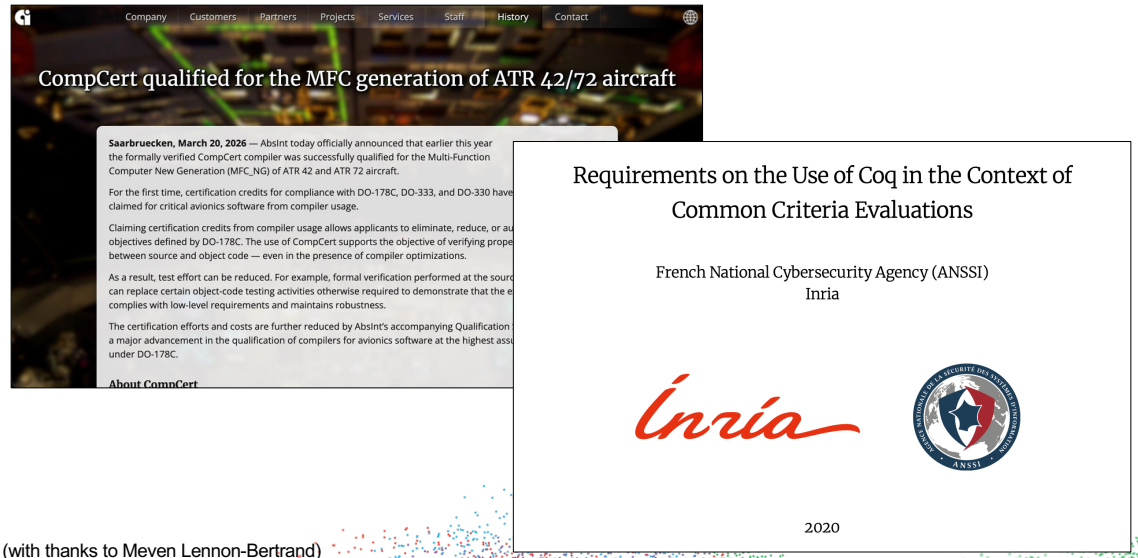
Formal definitions of semantics: CompCert, Cerberus, WasmCert, JSCert

Lighter-weight methods of program verification: hs-to-coq, rocq-of-ocaml, rocq-of-python, rocq-of-rust (Aeneas works with Lean as well)

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)

This may be more important for the European colleagues, but there has been a lot of work in talking to regulating bodies and explaining to them what Rocq is.

The French National Cybersecurity Agency has criteria on how to write Rocq, CompCert folks got the aircraft related certifications from European regulatory bodies so that it can be used on Airbus software. Gaining this sort of trust takes time and it's hard. Convincing them to add another proof assistant or verified tool will not be easy. It'll be easier than explaining to them what verification is for the first time, but it will not be easy. If you look at the list of missing parts of the ecosystem from the earlier slide and think "oh, that's easy, I can vibe-port that to Lean in a weekend", that may not fly in a regulatory setting.

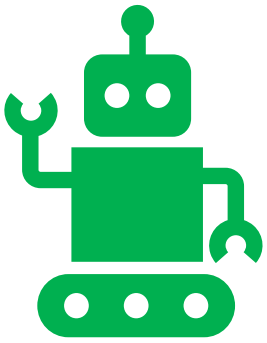
Why Rocq is ~~better~~ as good as than Lean for program verification

- 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.



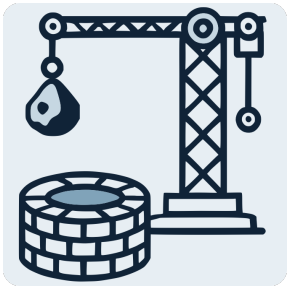
And finally, on using AI agents to write Rocq. I know all the hype is on making AI agents write Lean, but the agents are good at writing Rocq too.

<click> Here I should get some humility and change my title to “why Rocq is **as good as** Lean for program verification”

There is a lot of Rocq out there, the language has been around since late 80s.
AI models do pretty well now with languages they have not seen before. This is no longer 2024.
So feel free to try new languages, DSLs. Your AI tools can learn them and get up to speed with them pretty quickly.
Make the AI agent read your docs and it will get good at 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!**

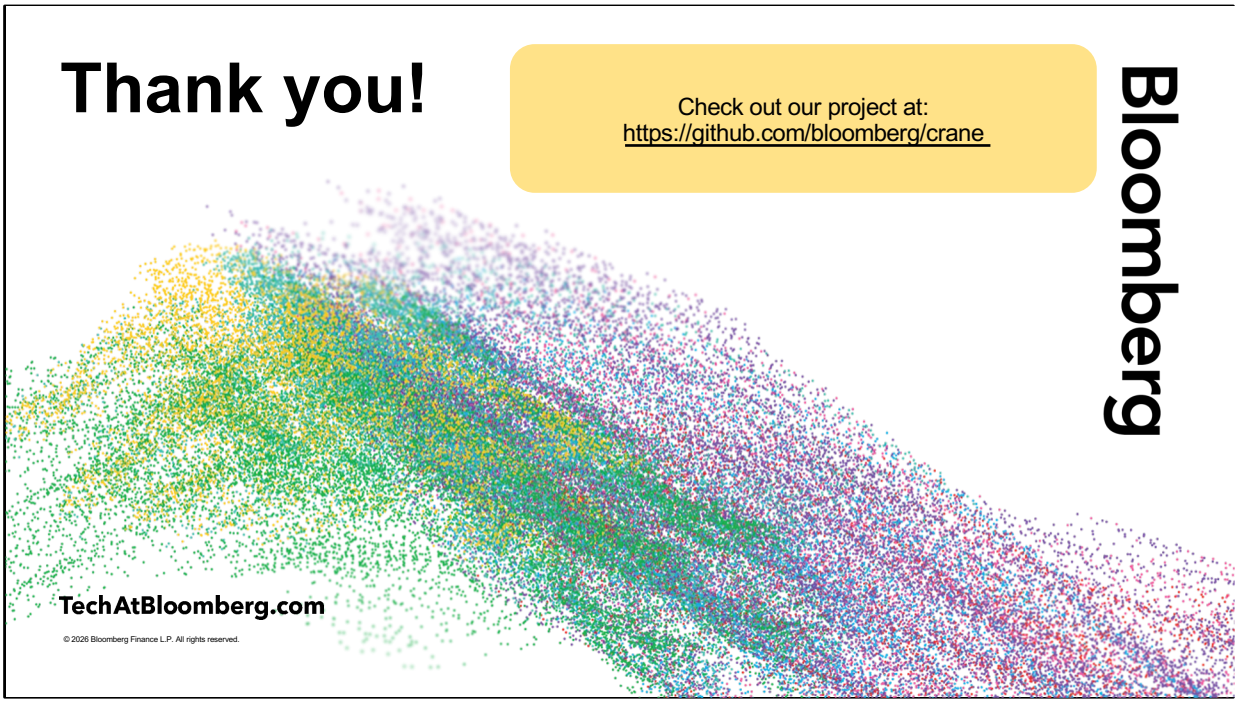


Ok, let's wrap this up. In summary:

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 C++ libraries such as concurrent hash tables, that will eventually be used by Bloomberg engineers.

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!**



If you have any suggestions about how we can get better performance, or if you've spotted a mistake somewhere, please find me or contact me. And with that, thank you for listening!

And the project is available on GitHub!