Certifying Derivation of State Machines from Coroutines

MIRAI IKEBUCHI, National Institute of Informatics, Japan
ANDRES ERBSEN, MIT CSAIL, USA
ADAM CHLIPALA, MIT CSAIL, USA

One of the biggest implementation challenges in security-critical network protocols is nested state machines. In practice today, state machines are either implemented manually at a low level, risking bugs easily missed in audits; or are written using higher-level abstractions like threads, depending on runtime systems that may sacrifice performance or compatibility with the ABIs of important platforms (e.g., resource-constrained IoT systems). We present a compiler-based technique allowing the best of both worlds, coding protocols in a natural high-level form, using freer monads to represent nested *coroutines*, which are then compiled automatically to lower-level code with explicit state. In fact, our compiler is implemented as a tactic in the Coq proof assistant, structuring compilation as search for an equivalence proof for source and target programs. As such, it is straightforwardly (and soundly) extensible with new hints, for instance regarding new data structures that may be used for efficient lookup of coroutines. As a case study, we implemented a core of TLS sufficient for use with popular Web browsers, and our experiments show that the extracted Haskell code achieves reasonable performance.

CCS Concepts: • **Theory of computation** → **Logic and verification**; • **Security and privacy** → *Cryptography*.

Additional Key Words and Phrases: coroutines, interaction trees, proof assistants, program derivation, nested state machines, cryptographic protocols

ACM Reference Format:
Mirai Ikebuchi, Andres Erbsen, and Adam Chlipala. 2022. Certifying Derivation of State Machines from Coroutines. *Proc. ACM Program. Lang.* 6, POPL, Article 24 (January 2022), 31 pages. https://doi.org/10.1145/3498685

1 INTRODUCTION

Many popular implementations of security-critical network protocols such as TLS are prone to "state-machine vulnerabilities" [Beurdouche et al. 2015; de Ruiter and Poll 2015; Yadav and Sadhukhan 2019] that occur due to human error in translating the desired interaction flow into a state type and state-transition function. This section will review the key programming task during which these errors occur and lead to our proposed alternative: a compiler that translates code with I/O operations into a self-contained state type and a step function that accepts inputs and returns outputs. We were surprised to discover that ideas with the flavor of algebraic effects [Plotkin and Pretnar 2008], specifically with programs in the style of freer monads [Kiselyov and Ishii 2015] (also associated lately with the terminology "interaction trees" [Xia et al. 2020]), facilitate a pleasant new source syntax with elegant theory of correct compilation. We prove (using the Coq proof assistant) that our translation preserves the behavior of the program and use it to write a short and elegant

Authors' addresses: Mirai Ikebuchi, ikebuchi@nii.ac.jp, National Institute of Informatics, Tokyo, Japan; Andres Erbsen, andreser@mit.edu, MIT CSAIL, Cambridge, Massachusetts, USA; Adam Chlipala, adamc@csail.mit, MIT CSAIL, Cambridge, Massachusetts, USA.

2475-1421/2022/1-ART24
https://doi.org/10.1145/3498685

```
             Can send         |  o o o
             app data         | Send Finished message
             after here  -->  | Now use "application" keys to encrypt sent data
                              | Now use "handshake" keys to decrypt received data
                   +----------+--------+
            No auth |                  | Requested client auth earlier
                    |              WAIT_CERT
                    |   Recv empty |       | Receive Certificate message
                    |  Certificate |    WAIT_CV
                    |              v       | Receive CertVerify message
                    +--> WAIT_FINISHED <---+
                              | Receive Finished message
                              v Now use "application" keys to decrypt received data
                         CONNECTED
```

Fig. 1. TLS 1.3 server-state-transition diagram. Simplified, based on a similar diagram in RFC 8446, Appendix A. State labels (all caps) have no meaning; other text describes actions from the server perspective.

TLS implementation. The main new technical challenge is handling of *nested coroutines* (to our knowledge, freer monads had not previously been applied to implement any kind of concurrency) and their proved compilation to nested state machines. However, we found that careful definition of the source language led the compilation details to fall out rather directly and pleasingly.

Direct-style protocol implementation. The IETF RFC specifying TLS 1.3, the latest version of the protocol underlying cryptographically secure Web browsing and more, includes a diagram like Fig. 1. It appears reasonably straightforward to translate it into the following pseudocode:

```
SendFinished(); KeysAppHandshake()
if RecvCertificate():
    RecvCertificateVerify()
else:
    FailIfCertificateRequired()
RecvFinished(); KeysAppApp()
```

The previous pseudocode takes advantage of familiar conveniences of "direct-style" programming, where we ask a compiler to handle representing control state with call stacks, continuations, or whatever else. There is no *explicit* programmer effort to keep track of "where we are" in the code. However, conventional implementation strategies for concurrent and nested direct-style code (interpreters, compilers, runtime systems) are difficult to support as C libraries that can be linked with unrelated C libraries – it does not scale to allow each library to introduce a new concurrency idiom and required runtime support. In practice, each network-protocol library written in C has some high-priority use case whose compatibility and performance constraints are believed to require use of "idiomatic" C code where protocol control state is explicit. However, that sort of explicit control management is precisely where state-machine bugs can arise.

State-machine challenges. For instance, NSS, Mozilla's cryptographic library, includes the code in Fig. 2 as part of its handling of TLS handshaking. The argument `ss` holds connection-specific state. Note how the `switch` statement on the left is dispatching based, in some sense, on "where the program counter is" in the more natural, direct-style version of this code. The more convoluted control flow of explicit state machines makes it harder to keep track of what is happening in the protocol. We must take care to consult the proper fields of persistent state and update them and others based on new protocol messages. Understanding the code on the right in Fig. 2 is suddenly far from obvious: while the fields consulted for conditional branches and written based on these decisions seem still to correspond to "where in the protocol" the server is, only one of them matches

```
HandlePostHelloHandshakeMsg(Socket*ss,u8buf*b){
  /* TODO(REDACTED): Would it be better to
                 check all the states here? */
  switch (ss→ssl3.hs.msg_type) {
    // [many other cases omitted]
    case ssl_hs_finished:
      return ServerHandleFinished(ss, b);
    default:
      FATAL_ERROR(ss,..., unexpected_message);
      return SECFailure;
```

```
ServerHandleFinished(Socket *ss, u8buf *b) {
    if (ss→ssl3.hs.ws != wait_finished) {
        return SECFailure;
    }
    ss→ssl3.hs.endOfFlight = PR_TRUE;
    ss→handshake = NULL;
    ss→ssl3.hs.ws = idle_handshake;
    // [cryptographic computation omitted]
    if (ss→sec.authType != ssl_auth_psk) {
        SendNewSessionTicket(ss, NULL, 0);
```

Fig. 2. Excerpts from NSS 3.59 server state machine code. Comments with // are ours, and /* .. */ are not.

```
match parse_handshake b with Ok handshake → (match cs, handshake with
 | AwaitClientCert(d,f,c,s,l),  Cert(x)              → answer_client_cert hs x d f c s b l
 | AwaitClientCertVerify(d,f,c,s,l), CertVerify(x) → answer_client_cert_verify hs x d f c s b l
 | AwaitClientFinished(f,c,s,l), Finished(x)        → answer_client_finished hs x f c s buf log
 |  (* ... *) | _, h                                  → fail(`Fatal(`UnexpectedHandshake h)))
```

Fig. 3. OCaml-TLS code for handling Finished message or alternatives, simplified and shortened under the same assumptions as NSS code earlier. 20 lines in the original.

a label from Fig. 1. The only reliable way to review this kind of code is to trace all reads and writes of each state variable exhaustively, mentally reconstructing a state diagram akin to Fig. 1 with many fewer branches than in the implementation. This exercise is time-consuming and error-prone, as evidenced by the severe bugs we recount in section 1.1.

The same complexity inherent to explicit nested state machines appears in the clean-slate OCaml implementation of TLS titled “Not-quite-so-broken TLS” [Kaloper-Meršinjak et al. 2015]: just compare Fig. 3 to our pseudocode sketch and then Fig. 2. While the OCaml TLS code is shorter and easier to read than the NSS code, we believe that this improvement is due to notational changes (primarily variant types) and omission of obscure functionality, making it orthogonal to the challenges we are discussing here.

Nested state machines. The challenge so far is similar to classic challenges of compiling direct-style code, but real-world network protocols are distinguished by the use of nested state machines. Consider the much more mundane task of reading a specified number of bytes from a stream: every time a chunk of bytes arrives, NSS appends it to the previously received buffer and checks whether the specified threshold has been reached (Fig. 11, pushed to the end of the paper). If yes, the code for handling incoming messages (and eventually calling code in Fig. 2) is invoked. The call stack handling the last byte of a message crosses all abstraction levels (substatemachines) of the TLS stack, again illustrating the difference of control flow between NSS (and other widely used implementations of TLS or similar protocols) and the way the same protocols are described in the standards or rigged up in toy/reference implementations (such as our pseudocode sketch above). For a server willing to communicate with multiple clients at once, it is natural to consider at least three levels of nesting of communicating state machines: top-level server, per-client message-level logic, per-client bytestream chunker.

We propose thinking of the “blocking” and “wakeup” code paths that appear between protocol-level actions as an inlined userland implementation of cooperative multithreading, such that NSS’s reads and writes of ss correspond to saving and restoring the state of a lightweight thread. The same *could* have been written using a couple of threads per connection, each of which would have straight-line control flow through actions at a single abstraction layer. However, the style with explicit state machines has three important benefits: it is deterministic, it does not depend on a particular implementation of concurrency, and it is believed to achieve better performance. We will focus on the first two.

Implementing network-protocol concurrency. The implementation of TLS in the standard library of the Go programming language expects the user to create multiple Go threads to handle simultaneous connections, which allows the library to use blocking I/O for receiving and sending messages, removing the need to reify the state between interactions and allowing for significantly shorter code. OCaml-TLS provides a similar interface using the LWT library to wrap the implementation that uses hand-reified state. As a result, a typical application would experience arbitrarily complicated interleavings between the handling of its multiple connections, complicating analysis and troubleshooting of potential issues. Further, both LWT and Go rely on intricate dynamic user-space implementations of multithreading which an application using the library would need to adopt and the deployment platform would need to support. We are not aware of any production code that uses a userspace multithreading implementation for some but not all I/O inside a process, or that uses two libraries with different implementations of multithreading, whereas NSS is one of 536 libraries in Firefox that use POSIX-style nonblocking I/O and C-style single-stack function calls[1]. Finally, libraries with hand-reified state can be used on embedded devices for which no general-purpose multithreading implementation is available. One could also use operating-system threads to the same effect, but this choice is rare because most libraries seek portability across operating systems and have some use cases for which the overhead of creating and switching operating-system threads is prohibitive.

Our contribution. We propose a programming style and a compiler to generate code with reified state machines (in the style that is written by-hand for implementations that are widely used today) from source code that uses blocking I/O and allows for successive steps in each layer of the protocol to be written in straightline code. The source language is realized as a Coq library for structuring these computations, with the ability to include arbitrary code in Coq's Gallina language in the steps between explicit state-machine operations. We encode interthread communication in the style of freer monads, as elements of an inductive datatype that sequences stateful operations explicitly. Our implementation can be seen as an example of an approach to compiling monadic programs with coroutines to explicit state machines. The compiler is written in Coq's Ltac tactic language, and it generates proofs of semantics preservation, according to a natural relation between monadic programs and state machines. In fact, compilation is structured as a search for a proof of "there exists a related state-machine program."

The step taken in this paper is just the first toward trustworthy compilation of natural protocol definitions to efficient and portable code. We will show how to compile natural *higher-order, modular* descriptions into less-natural *first-order, flat* descriptions of state machines, and we will do it with generation of Coq proofs per-compilation. The resulting functional programs may very well achieve superior performance to the originals on certain language implementations, but our longer-term goal is to pick up with trustworthy compilation of first-order functional programs to C and related languages. In that way, we can maximize control over performance-relevant behavior, and we can generate idiomatic code free of dependencies on garbage collectors, thread libraries, and the like. For now, though, our experiments do depend on standard functional-language implementations.

Starting the project, it was not at all obvious to us that there was a close connection between freer monads and nested state machines in multithreaded code. We eventually arrived at what we hope is a pleasingly simple formulation, of original programs and their certifying compilation to state machines. It allows use of arbitrary container data structures to keep track of nested state

[1]On Debian Linux on x86_64, including vendored code in `third_party`

machines, via type classes. Before moving on to specifics, we will review some of the evidence that a new programming style is sorely needed.

1.1 Past High-Impact Bugs Motivating This Work

Perhaps the best-known TLS state-machine vulnerability is that OpenSSL 1.01g and earlier[2] accepted a `ChangeCipherSpec` (`CCS`) message before the connection master secret was established and generated session keys without any entropy. A network attacker intercepting a connection between vulnerable implementations could decrypt and modify the traffic by injecting one CCS message in each direction, deterministically generating the same keys and hijacking the connection from there on. Operationally, a buggy execution interprets a Boolean state variable `new_cipher` as "is it valid to change to a new cipher now?", but that variable is written as "is a new cipher type known?", with the crucial difference being whether the key is known [Langley 2014].

The history of the bug suggests it was a conceptual issue, not a fat-finger error. TLS client code in all public versions of OpenSSL dating back to 1998 is affected. Similar issues related to mishandling of early CCS messages had been patched in 2004 and 2009 before the famous one was discovered in 2014. The discoverer of the vulnerability also highlights language in the SSLv3 specification that could have led to misunderstanding of the requirements for handling CCS [Kikuchi 2014], but since encryption without a key is glaringly useless, we prefer to look for the root cause in code structure that obscured this possibility.

The following systematic investigation uncovered numerous bugs by automatically tampering with the sequence of messages [Beurdouche et al. 2015]; the authors caution that the search was far from complete due to the amount of effort required to tell serious bugs apart from harmless state-machine imprecision. Arguably the most broadly severe finding was that almost all tested TLS client code allowed an unexpected `ServerKeyExchange` message to replace a properly negotiated public key with a weaker alternative also offered by the server. Due to legacy support for intentionally weak "export" ciphers from before 2000, this allowed for impersonation of many web servers at the cost of 512-bit factoring. Affected implementations included OpenSSL, BoringSSL (Chrome), SecureTransport (Safari), SChannel (Internet Explorer), LibReSSL, Mono, and Java. Again, an implementation with straightforward control flow could not exhibit this behavior: an incoming message of an unexpected type would be handled naturally as an error.

1.2 Formal-Methods Context

We decided not just to design a language (extension) and compiler to automate this kind of error-prone programming. We also want to craft the language and tooling for eventual connection with end-to-end machine-checked proof of cryptographic systems.

Formal methods and cryptography have long been connected fruitfully. There is a natural division of labor across subcommunities. For instance, one might verify cryptographic security of a protocol like TLS using a tool like CryptoVerif [Blanchet 2006] while relying on number-crunching cryptographic primitives implemented and proved in the F* language as in the Ever-Crypt library [Protzenko et al. 2020]. It is tempting to try to fit all relevant proofs within a single framework, as with a proof assistant like Coq. Indeed, alternatives for the ingredients already exist, e.g. for security verification, CertiCrypt [Barthe et al. 2009] and the Foundational Cryptography Framework [Petcher and Morrisett 2015]; and for functional-correctness verification, Fiat Cryptography [Erbsen et al. 2019]. The team behind the Verified Software Toolchain (VST), a Coq library for verifying C programs, has even done the integration for two important primitives [Beringer et al.

[2]CVE-2014-0224, https://www.openssl.org/news/secadv/20140605.txt

2015; Ye et al. 2017], connecting to verified C compiler CompCert [Leroy 2009] to push guarantees down to assembly.

However, an important gap in this work has been natural specification or coding of protocols. It is not enough to treat a cryptographic security condition as the "public interface" of a protocol. Implementations need to interoperate with each other, so they must agree on all protocol details. Past cryptographic work with formal methods has also implemented nested state machines directly, with the accompanying opportunities for bugs. Proof of security properties is, of course, valuable and effective, but proving two implementations secure on their own does not imply that they will interoperate securely (or at all!). Two teams implementing the same protocol from an informal description may translate it into code bases that are subtly insecure when used together, even if each individually is proven against the same security property. It seems we have a case truly demanding that complex protocol descriptions be auditable by humans.

Thus, we have framed our work in terms of a useful building block for end-to-end crypto proof: certifying compilation of natural higher-level code to nested state machines. As a result, we were able to write our executable Coq version of a sufficient TLS subset to interact with real browsers, finding only two bugs through testing rather than type-checking and manual inspection (and one bug was in a library dependency we used for crypto primitives, which we hope eventually to replace with a verified alternative). We did not prove our version against a security property, though our source notation is designed to be more congenial to that purpose than alternatives.

Our work fits in the tradition of program derivation through mechanized proof, where a program is found by constructively proving its existence against a spec, using tactic scripts. For instance, the Fiat project has demonstrated such derivation in a common framework, for the examples of relational-database-style querying [Delaware et al. 2015] and binary-format parsing [Delaware et al. 2019]. Our new compiler can be viewed as adding another domain, coroutine management in network protocols, to that repertoire. We also only consider features relevant to how engineers in industry write down protocols for easy manual translation to many languages, thus not complicating the story with features like higher-order state, general recursion, or exceptions, which often create headaches for semantics designers.

As in the work with VST, our eventual goal is to generate proven C code and connect with verified compilers, though we did not implement that translation yet, stopping instead at functional code that can be extracted to Haskell. As a result, we do not yet realize our performance goals, though we come surprisingly close to a prior Haskell implementation at low concurrency levels (our current tooling does not take advantage of multicore execution). Taking inspiration from Fiat, we hope to use tactic-based program derivation to produce C code equivalent to ours, perhaps adopting their technique [Pit-Claudel et al. 2020] literally.

1.3 Outline of This Paper

Having set the stage for the problem we solve, we switch, in the next section, to illustrating an alternative programming style (based on coroutines) and how it can be compiled to maintain state machines explicitly. Afterward, we introduce our languages and compilation in detail, also considering how the latter can generate machine-checked (Coq) proofs of behavior preservation. Then we turn to our case study implementing TLS 1.3 as required to serve Web requests from popular browsers. We discuss the code, its performance, and our experience debugging it, before wrapping up with related and future work.

The contributions we mean to highlight are:

- We identify the important problem of translating higher-level, coroutine-based protocol implementations into explicit nested state machines.

- We present a simple convention on top of freer monads that serves as a convenient embedded source language for protocols, within a general-purpose functional language.
- We present an effective proof-generating compilation method, including a multiphase way of handling nested coroutines and a type-class-based approach to handling different container data structures.
- We evaluate a prototype tool on a large enough subset of TLS 1.3 to interoperate with all popular browsers.

The code is available under an open-source license at:

https://github.com/mit-plv/certifying-derivation-of-state-machines-from-coroutines

2 BLOCKING I/O, COROUTINES, AND STATE MACHINES

Let us now consider what it might be like to program these protocol implementations more directly, while relying on a compiler to produce traditional implementations without library dependencies. We start with simplified syntax for functional programming, before moving to Coq specifics later. For now, we rely on reader intuitions about the notations before we explain their formal desugarings in the next section.

To get a sense for our compilation strategy, it is illustrative to begin with single-threaded code. Here is our first example program, which reads two inputs (e.g., from a terminal) and then perhaps prints a derived value, before returning. It is a useful starting intuition to imagine we are programming in the style of Haskell's IO monad.

```
Definition get_put :=
    w1 ← input;
    w2 ← input;
    if w1 == "" then
        return "done"
    else
        print (w1+w2);
        return w2
```

It is now well-known how to implement such direct-style code in terms of lower-level primitives in continuation-passing style; see, for instance, Reynolds [1972]. It will be instructive to review the outcome of such a process, before we move on to the new subtleties of our work. Our compilation strategy produces the following alternative code without implicit blocking for `input`, where we omit the definition of an algebraic datatype being used to represent state (otherwise known as defunctionalized continuations). We have a step function that, given a state and a result from a prior system call (primitive I/O operation), returns a new state and an optional next system call.

```
Definition get_put' (st : state) (syscall_result : string) : state * option syscall :=
  match st with
  | Init ⇒ (AfterInput1 syscall_result, Some Input)
  | AfterInput1 w1 ⇒ (AfterInput2 w1 syscall_result, Some Input)
  | Afterinput2 w1 w2 ⇒
    if w1 == "" then
      (Terminate "done", None)
    else
      (AfterPrint w2, Some (Print (w1 + w2)))
  | AfterPrint w2 ⇒ (Terminate w2, None)
  end
```

Each constructor of the state (continuation) type receives free variables as arguments. The function `get_put'` itself is a kind of automaton step function. Note how it handles a state constructor per effectful line of source code, returning values built with the constructors `Input` or `Print` to request next system calls. Those calls' results flow back in through the `syscall_result` argument in subsequent steps. We simplify these examples by allowing ourselves to skip handling `Terminate` states, which signify that no more steps are needed.

2.1 Coroutines

Now let us introduce the twist vs. past work: multithreading, which is actually presented with *coroutines* in our source language. Adopting Python's coroutines terminology in functional code, we end up with examples like the following.

```
Definition ex_coro :=
    s0 ⟵ yield "";
    s1 ⟵ yield s0;
    yield (s0 + s1)

Definition parent :=
    let c := ex_coro in
    send c "";
    r1 ⟵ send c "Hello, ";
    print r1; (* Output: Hello, *)
    r2 ⟵ send c "world!";
    print r2 (* Output: Hello, world! *)
```

Note how control is ping-ponging back-and-forth between parent code and its child coroutine, where the argument of a parent `send` call becomes the result of a pending child `yield`, and where, symmetrically, the argument of a child `yield` becomes the result of a parent `send`. The compiled state-machine version explains it all in terms of conventional functional programming. First, here is how we compile the child coroutine, just like for our last example but presuming that `Yield` is the only available syscall.

```
Definition ex_coro' (st : child_state) (syscall_result : string)
  : state * option syscall :=
  match st with
  | Init ⇒ (AfterYield1, Some (Yield ""))
  | AfterYield1 ⇒ (AfterYield2 syscall_result, Some (Yield syscall_result))
  | AfterYield2 s0 ⇒ (Terminate, Some (Yield (s0 + syscall_result)))
  end
```

For the parent, before transforming it into a state machine, we transform it into an intermediate form that creates the state machine of `ex_coro` and calls its step function instead of `ex_coro` itself:

```
Definition parent_inter :=
    let c := (Init, ex_coro') in
    r1 ⟵ step c "Hello, ";
    print r1; (* Output: Hello, *)
    r2 ⟵ step c "world!";
    print r2 (* Output: Hello, world! *)
```

Then, the state machine for `parent_inter` is obtained in the same way. That is, we need not invent new compilation techniques for handling coroutines, once we inline state machines in the code of their parents. We maintain a small veil of mystery until the next section, in terms of how we make some of these coroutine notations work in a purely functional setting.

2.2 Coroutines in Containers

Our compiler also handles parents with container data structures storing coroutines, as in the following example.

```
Definition parent2 :=
    let l := map (fun _ ⇒ ex_coro) (numeric_range 5) in
    l ← mapM (fun c ⇒ send c "") l;
    l ← foldlM (fun _ l ⇒
      n ← input;
      let n := as_int(n) in
      if 0 <= n < 5 then
        with_list_elementM l n (fun c ⇒
          s ← input;
          send c s)
      else
        return l) l (numeric_range 3);
    return "done"
```

This example makes good use of higher-order functions for monadic code. For instance, `mapM` is used to run a side-effecting computation on each element of a list. We start by initializing a list `l` of five child coroutines, and then we use `mapM` to send each coroutine an initial message. Then we use `foldlM` to run through integers 0, 1, and 2 (the elements of `numeric_range 3`), in each case reading a numeric input and using it to choose a coroutine in `l` that should be passed another fresh input. It is important that the coroutines maintain their local state throughout these interactions.

We are playing a bit fast and loose with the plumbing of coroutine state above; we very shortly reveal the details, which do require small modifications to this example. In general, realistic protocol implementations may nest coroutines like these to arbitrary depth, and they may use different container structures to support efficient lookup of the coroutine implicated in a particular step.

The compiled program for this example is obtained as in the prior example: we first transform this parent code to store explicit state machines instead of coroutines, and we then compile the transformed parent into a state machine itself. Imagine that process proceeding recursively, for examples with deeper nesting.

Taking a step back, what we have accomplished with this style is good modularity between aspects of a communication protocol (e.g., the code of parents vs. children in these last two examples), which communicate via messages internal to the protocol implementation. Each aspect can be audited in terms of its own vocabulary of incoming requests and outgoing responses. Unfortunately, standard implementations of coroutines add significant performance overhead, with dynamic allocation of coroutine objects, nested inside each other's closures. Furthermore, the very need to identify and allocate closures implicitly forces us toward a "managed language" that may be difficult to bring up on a resource-constrained embedded system, or even as part of a new library added to a program written in a language with a different runtime system. The outputs of our compiler are "more first-order" and easier to compile directly to, say, C code (which we hope to do, in a proof-producing way, in future work). Further, we generate Coq proofs of correct compilation, in a sense taking a step toward certifying compilation for features familiar from Python and other languages.

3 COMPILATION STRATEGY

Our compiler produces a state machine from an effectful program and proves the equivalence between the input and output programs. From this point forward in the paper, we more rigorously present the actual code representation and other Coq-level details in our prototype library.

3.1 Source Programs: Freer Monads

To represent effectful programs in our system, we use the following variant of the structure called a *freer monad* [Kiselyov and Ishii 2015].

```
Inductive t {eff : Set} (args rets : eff → Set) (A : Set) :=
| Eff : forall (e : eff), args e →
    (rets e → t args rets A) → t args rets A
| Return : A → t args rets A.
```

The type `eff` is the type of primitive effectful operations (and curly braces ask for it to be inferred at use sites), and functions `args` and `rets` respectively assign argument and return types to operations. For example, consider the following:

```
Inductive effect := getStr | putStr.

Definition args_effect (e : effect) :=
    match e with
    | getStr ⇒ unit
    | putStr ⇒ string
    end.

Definition rets_effect (e : effect) :=
    match e with
    | getStr ⇒ string
    | putStr ⇒ unit
    end.
```

Then we introduce convenient notations, following the convention of writing e.g. `c(x)` as a metavariable standing for a term where variable `x` can occur free, so that we can write `c(e)` for substitution of term `e` for `x`. Also note that `tt` is the one value of type `unit`.

```
x ← getStr; c(x) ≡ Eff getStr tt (fun x ⇒ c(x))
putStr x; c ≡ Eff putStr x (fun _ ⇒ c)
```

Now we can write effectful programs in an imperative style. Coroutines are represented as programs inhabiting the `t` type family, with `yield`s as their effects. So, the type of effects is

```
Inductive yield_eff := yield.
```

The argument and return types of `yield` will vary across use cases, as they indicate which sorts of data are passed between parent and child coroutines.

Our focus in the rest of the paper will be on correct compilation of these programs to a lower-level form. Of course, it is also valuable to establish correctness and security properties of the source code. As that source language is more or less interaction trees, and as there has already been substantial work on functional verification of interaction trees [Koh et al. 2019; Xia et al. 2020; Zhang et al. 2021], we have high hopes that follow-on work can connect such proofs to our certifying compilation.

3.2 Target Programs: State Machines

Let us consider an example (Fig. 4) of how such an effectful program can be compiled to a state machine. Recall that our state machines are just pure Coq (Gallina) programs paired with state types and initial states. As a result, there is no need to delineate what the "target language" is. Instead, think of arbitrary pure functions of type `state → syscall_response → state * option syscall` as in the examples of section 2. Our implementation actually uses dependent types to model the varying argument and return types of different effects, but the core ideas can be understood without appeal to dependent types.

get_put_state is the type of states of the generated state machine. It has five states (some taking arguments) that are constructed by q_0, q_1, $q_2\ x$, $q_3\ x$, and $q_4\ x$; where x in each state is a tuple of saved original-program-variable values needed to proceed from that state. The function get_put_step is another component of the state machine. It takes a state as its first argument and the value returned by the previous effect as the third argument. The second argument is expected to be the previous effect performed, which is needed to make the step function well-typed. If the given first argument is the initial state, then get_put_step does not use the remaining arguments. It returns Done r if the program is finished outputting the value r and returns $\mathrm{GoNext}(s, o)$ otherwise. Variable s is the next state to visit, and o is $\mathrm{Some}(e, a)$ if the effect operation e with argument a is expected to be performed. Variable o is None if no more effects will be performed (i.e., execution is complete).

To run this state machine, we extract the step function to another language and write a loop to drive I/O. An example of a Haskell implementation of the loop is given in Fig. 5. It relies on a few "unsafe" type casts to work around the expressivity gaps between the type systems of Haskell and Coq, though one can imagine extending our Coq code derivation to produce (proved) lower-level code where these concerns no longer exist.

Here we pause to emphasize how lightweight this encoding is, as a layer on top of Coq's functional language Gallina. We are adding effects in a very similar style to how I/O was added to Haskell via monads [Peyton Jones and Wadler 1993]. The crucial difference is that we materialize explicit abstract syntax trees of effectful programs in our `t` type family, making it easy to deconstruct such values both in Coq semantics and proofs and in interpreters like the one in Fig. 5. As when Haskell was extended with monadic I/O, all familiar pure constructs of the language remain usable, usually requiring no special effort by our compiler to preserve. The compiler is designed so that, if a source program is higher-order only in its calls to constructors of the `t` type family, then the compiled state-machine version is first-order, in the sense of using no higher-order functions.

3.3 Compilation, Informally

We shall see how we obtain a type of states and a step function from a source program. First, we focus on programs that interleave I/O actions with computation, postponing to the next subsection a modularizing of code across coroutines. Recall that our encoding of stateful programs is very lightweight and relies on built-in pure-Gallina features for the plumbing between effects. As a result, there is no more detailed syntax-tree type beyond the inductive definition of `t` shown earlier in this section. However, implicit in the compiler is a limited grammar of "statement" constructs, or syntactic forms used to build values inhabiting `t`, while we impose no restrictions on "expression" constructs, for building e.g. the parameters passed to system calls.

Here is a grammar in familiar notation, meant to convey that idea. Most important to remember is that the nonterminal v encompasses *all terms of Coq's functional language Gallina*. The nonterminal p of programs includes cases for the constructors of `t`, plus a handful of control-flow features that we found useful to support, each of which needs its own case in our compiler. Note that, while the inductive definition of `t` includes no constructors for special control flow, the use of an argument

Source :

```
get_put =
  $w_1 \leftarrow$ getStr;
  $w_2 \leftarrow$ getStr;
  if $w_1$ = "" then
    Return None
  else
    putStr ($w_1$++$w_2$);
    Return (Some $w_2$)
```

Compiled :

$$\text{get\_put\_state} = q_0 \mid q_1 \mid q_2\ \text{string} \mid q_3\ (\text{string} \times \text{string}) \mid q_4\ (\text{option string})$$

$$\text{get\_put\_step}\ s\ e\ a =$$

$$\begin{cases} \text{GoNext}(q_1, \text{Some}(\text{getStr}, ())), & \text{if } s = q_0 \\ \text{GoNext}(q_2\ a, \text{Some}(\text{getStr}, ())), & \text{if } s = q_1,\ e = \text{getStr} \\ \text{GoNext}(q_4\ \text{None}, \text{None}), & \text{if } s = q_2\ w_1,\ w_1 = \text{""},\ e = \text{getStr} \\ \text{GoNext}(q_3\ (w_1, a), \text{Some}(\text{putStr}, w_1 +\!\!+ a)), & \text{if } s = q_2\ w_1,\ w_1 \neq \text{""},\ e = \text{getStr} \\ \text{GoNext}(q_4\ (\text{Some}(w_2)), \text{None}), & \text{if } s = q_3\ (w_1, w_2),\ e = \text{putStr} \\ \text{Done r}, & \text{if } s = q_4\ r \\ \text{Done None}, & \text{otherwise} \end{cases}$$

Fig. 4. An effectful source program and its compiled version

```
loop :: Get_put_state → Effect → Any →  IO (Maybe String)
loop st0 e r =
    case get_put_step st0 e r of
        Done v → return v
        GoNext (st, Nothing) → loop st dummyEffect dummyArg
        GoNext (st, Just (ExistT e' a)) →
            case e' of
                GetStr → do
                    str ← getLine
                    loop st GetStr (unsafeCoerce str)
                PutStr → do
                    putStrLn (unsafeCoerce a)
                    loop st PutStr (unsafeCoerce ())
    where dummyEffect = GetStr
          dummyArg = unsafeCoerce ()
```

Fig. 5. Haskell program to run the state machine get_put_step

of type `rets e` $\rightarrow$ `t args rets A` naturally allows use of native Gallina features to deconstruct system-call return values in `rets e`. (For a gentler introduction to the consequences of this encoding style, see a recent functional pearl by one of us [Chlipala 2021].)

Identifier	i
Program (without effects)	v
Effect	e
Effectful program	$p ::= \mathsf{Return}\ v$
	$\mid i \leftarrow e\ v; p(i)$
	$\mid \mathsf{let}\ i := v\ \mathsf{in}\ p(i)$
	$\mid \mathsf{if}\ v\ \mathsf{then}\ p\ \mathsf{else}\ p$
	$\mid (\mathsf{fix}\ i_1\ i_2\ i_3 :=$
	$\quad \mathsf{if}\ i_2 = 0\ \mathsf{then}\ p(i_3)\ \mathsf{else}\ p(i_2, i_3, i_1(i_2 - 1)))\ i_2\ i_3$

We can easily extend to support pattern matching for an arbitrary algebraic datatype, but our current compiler only supports a few selected algebraic datatypes (currently lists, options, and variants). Here we include recursive-function definition specialized to the natural numbers with a fix keyword. Our implementation also supports recursion on lists.

We present the compilation approach first in a relatively informal way, circling back in the next section to fill in the gaps. Fig. 6 shows an implementation in some least-common-denominator purely functional language. Function C_0 compiles an effectful program (inhabiting type family t) to a type of states and a pure step function. It relies mostly on helper function C_0', which takes as an additional argument a typing context Γ, mapping free variables of the effectful program to their types (which will not change during compilation).

The outputs of compilation are accumulated in two mutable lists state_type and step_cases, which respectively build up the constructors of the algebraic data type for states and the pattern-matching cases of the step function. An entry in the former gives a constructor name and its argument types; while an entry in the latter gives a state-type constructor with variable binders for its arguments, an effect constructor with a variable binder for its parameter, and a body expression to evaluate under all those binders. Compilation kicks off by allocating a fresh name q_0 for the initial state constructor.

The C_0' cases for Return and Eff are responsible for generating step-function expressions using Done and GoNext directly. The latter generates a fresh state name and makes a recursive call to compile the code coming *after* the indicated effect. Crucially, the same state name is used both to extend the pattern-matching cases and to generate a reference to the same state in the arguments to GoNext.

The control-flow cases for let and if just make recursive calls inside the bodies of the expressions, while handling of fix is more involved. For termination of the algorithm, we memoize compilation results in a mutable variable recursive_call_cache. The idea is to compile each of the two cases within the fix under appropriately extended environments Γ, with the fix itself substituted for the function name in the latter. When we do eventually hit another recursive call, it will (almost; see below) hit in the cache and avoid any more deep traversal.

The code of Fig. 6 is cutting corners in a few dimensions that would already be problematic for writing an executable compiler, let alone one amenable to mechanized proof.

- We deal with the usual bureaucracy of manipulating terms with variable binders, including substitution and fresh-name generation, which are not so trivial to formalize.

```
$C_0(p)$ =
  $q_0 \leftarrow$ gensym;
  state_type $\leftarrow$ ref $[q_0]$;
  step_cases $\leftarrow$ ref [];
  recursive_call_cache $\leftarrow$ ref [];
  let rec $C_0'(\Gamma, p)$ = case $p$ of
    | Return($v$) $\Rightarrow$ return (Done $v$)
    | Eff($e, a, k$) $\Rightarrow$
      $q \leftarrow$ gensym;
      append state_type ($q(\Gamma)$);
      $x \leftarrow$ gensym;
      $p' \leftarrow C_0'(\Gamma + x : \text{args}(e), k(x))$;
      append step_cases ($q(\Gamma), e(x), p'$);
      return (GoNext ($q(\text{vars}(\Gamma)), (e, a)$))
    | (let $x : \tau = e$ in $k(x)$) $\Rightarrow$
      $p' \leftarrow C_0'(\Gamma + x : \tau, k(x))$;
      return (let $x : \tau = e$ in $p'(x)$)
    | (if $b$ then $p_1$ else $p_2$) $\Rightarrow$
      $p_1' \leftarrow C_0'(\Gamma, p_1)$;
      $p_2' \leftarrow C_0'(\Gamma, p_2)$;
      return (if $b$ then $p_1'$ else $p_2'$)
    | (fix $f\ n\ (x : \tau)$ := if $n = 0$ then $p_1(x)$ else $p_2(n, x, f(n-1))$) $a\ b$ $\Rightarrow$
      cached $\leftarrow$ lookup recursive_call_cache $p$;
      case cached of
      | Some($v$) $\Rightarrow$ return $v$
      | None $\Rightarrow$
        $p_1' \leftarrow C_0'(\Gamma + x : \tau, p_1(x))$;
        $p_2' \leftarrow C_0'(\Gamma + n : \mathbb{N} + x : \tau,$
          $p_2(n, x,$ (fix $f\ n\ x$ := if $n = 0$ then $p_1(x)$ else $p_2(n, x, f(n-1))$)));
        let $r$ = (if $a = 0$ then $p_1'(b)$ else $p_2'(a-1, b)$) in
        append recursive_call_cache ($p, r$);
        return $r$ in
  $p' \leftarrow C_0'([], p)$;
  append step_cases ($q_0$, _, $p'$);
  $T \leftarrow$ read state_type;
  $S \leftarrow$ read step_cases;
  return ($T, S$)
```

Fig. 6. Informal version of core compiler

- We play fast-and-loose with syntax of the metalanguage vs. the object language, overloading e.g. let for compile-time and quoted run-time versions.
- The handling of recursive fix functions seems to beg the question, actually leading to an infinite loop at compile time. It could probably be made to work out, though, if the cache associated recursive functions with mutable references that are filled in as we compile.
- The code is rather imperative, which tends to create headaches for mechanized proofs, compared to more purely functional implementations.

Since we want a Coq proof of correct compilation on each run of this algorithm, it is natural to expect that Coq's tactic language Ltac will be involved somehow. However, it also turns out that Ltac's features provide for elegant solutions to the rough edges above. Tactic languages are designed for recursive analysis and construction of logical terms. Management of variables and environments is built into the proof engine, and there is integrated quoting of terms as needed, connected smoothly with type checking. Finally, the same feature addresses our worries about fix compilation and excessive use of imperative features: *unification variables*, which are selectively filled in during proof search. We simply fill in recursive_call_cache with fresh unification variables *before* proceeding to compile fix bodies, and all three imperatively growing lists become unification variables in Ltac.

Now for the details, after we lay the foundations for how we state compilation as a proof goal compatible with Ltac.

3.4 Compilation, Formally

To state compilation-correctness theorems, we define an equivalence between an effectful program and a state machine. We write equiv step q_0 p if an effectful program p and a pair of step function step and initial state q_0 are equivalent, and it is defined by the following inference rules, which also define a further relation $\simeq$ connecting states for target and source programs. Here, in equations between function calls and specific shapes of results, we use underscores to require that the function result is independent of those argument values.

$$\frac{\mathsf{step}\ q_0\ \_\ \_ = \mathsf{GoNext}\ (s, o) \quad (\mathsf{step}, s, o) \simeq p}{\mathsf{equiv}\ \mathsf{step}\ q_0\ p}$$

This top-level relation begins by assuming the state machine's initial state is a kind of dummy state that ignores its next input, returning the same pair of a new state and next effect regardless. Then the main task of equivalence checking is pushed off to relation $\simeq$. Note that the lefthand operand to $\simeq$ characterizes the state machine's condition with a step function, state, and intended next effect (o). Also note that these rules are defined inductively, not coinductively, requiring that every execution terminate, even if executions can run arbitrarily long depending on input values.

$$\frac{\mathsf{step}\ s\ \_\ \_ = \mathsf{Done}\ v}{(\mathsf{step}, s, \mathsf{None}) \simeq \mathsf{Return}\ v} \qquad \frac{\forall r, s', o'.\ \mathsf{step}\ s\ e\ r = \mathsf{GoNext}\ (s', o') \to (\mathsf{step}, s', o') \simeq p'(r)}{(\mathsf{step}, s, \mathsf{Some}\ (e, a)) \simeq (i \leftarrow e\ a; p'(i))}$$

These rules are essentially a simplification of the classic technique of bisimulation for our setting. In fact, we proved equivalence[3] with the more standard definition of bisimulation, after expressing our source and target semantics in labeled-transition-system style. We show that the two versions of a program evolve in lockstep, regardless of how the environment responds to effect requests. If the low-level step function indicates execution is done regardless of the next effect response, then the high-level program had better be done with the same return value. If the low-level and high-level programs agree on an effect to run next, then for any state transition indicated by the low-level step function for some possible effect response (r), the relation is preserved thereafter.

Note that this relation does not need rules for the other language constructs we explained so far, because they are encoded in shallow-embedding style, using native Coq constructs, the possibility for which is a significant advantage of the monadic style. We also point out that, like standard bisimulation, this relation enforces consideration of all possible effect responses, even the bizarre corner-case responses that make the most trouble for authors of protocol implementations. Also, all concurrency behavior of the state machine is inherited literally from the source program; there are no new concerns of fairness, etc.

[3]In our implementation, see theorem `equiv_is_bisimulate` in file `src/ClConv.v`.

Our system automatically proves equivalence between a source program and its compiled version. Moreover, the proof is constructed as we compile a program. That is, we state compilation as the process of proving that *there exists* a state-machine program related to the source program by $\simeq$. As we carry out the proof, we fill in more and more of the structure of the state-machine program.

If the source program is $i \leftarrow e\ a; p(i)$, then the inductive definition of $\simeq$ determines what the step function should be at the state for $i \leftarrow e\ a$, and it matches the form of the step function described in derive_core that will be explained later in this subsection. If the program is if b then p_1 else p_2, we apply the following lemma.

Lemma 1. If $(\mathrm{step}, s_1, o_1) \simeq p_1$ and $(\mathrm{step}, s_2, o_2) \simeq p_2$, then $(\mathrm{step}, s, o) \simeq p$ holds where

$$\begin{aligned} s &= \text{if } b \text{ then } s_1 \text{ else } s_2,\\ p &= \text{if } b \text{ then } p_1 \text{ else } p_2,\\ o &= \text{if } b \text{ then } o_1 \text{ else } o_2. \end{aligned}$$

We have similar lemmas for let and fix, and an entire proof is a sequence of applications of these lemmas and rules of $\simeq$.

Now we are ready to present the Ltac realization of compilation procedure C_0 from Fig. 6. Key complications from that figure, principally quoting of syntax trees and management of fresh variable names and environments, are built into the Coq proof engine and need not be dealt with explicitly anymore. The three imperative lists that we built up by appending are replaced by unification variables. Probably least intuitively, rather than returning target programs, the compiler fills them into unification variables passed as arguments. They will be written as $?s$ for the state part of a target program and $?o$ for the next-side-effect part. This commitment to unification variables turns out to solve our challenge with compilation of recursive functions.

A type of states and a step function are determined in the following procedure. First, let the type of states be $q_0 \mid ?X$, and let q_0 be the initial state. The notation $q_0 \mid ?X$ means that the available states include q_0 (with no arguments) plus a currently unknown sequence of additional states $?X$. Then, we partially define a step function step as step q_0 _ _ = GoNext$(?s, ?o)$ for new unification variables $?s, ?o$. By the end of our procedure, step will be fully defined, filling in one state's case at a time, adding new cases as the compiler discovers new states.

Next, let prog be the input program and run the procedure derive_core below, which we implement in Ltac.

derive_core prog $?s$ $?o$ =

- Case prog = $(r \leftarrow e\ a;\ p(r))$: If the type of states is of the form $q_0 \mid \cdots \mid q_n x \mid ?X$, unify $?o$ with Some(e, a) and $?s$ with q_{n+1} fv, where fv is the (possibly nested) tuple of free variables in prog. (By this process, $?X$ is unified with $q_{n+1}\ T \mid ?Y$ for the type T of fv, with $?Y$ a fresh unification variable.) Then, let step $?s\ e\ r$ = GoNext$(?s', ?o')$ for new unification variables $?s'$ and $?o'$, then run derive_core $p(r)$ $?s'$ $?o'$ recursively, in a context with r introduced as a new local variable.
- Case prog = let $x := y$ in $p(x)$: Unify $?s$ with let $x := y$ in $?s'$ and $?o$ with let $x := y$ in $?o'$ for new unification variables $?s'$ and $?o'$. Run derive_core $p(x)$ $?s'$ $?o'$, in a context with x introduced as a new local variable.
- Case prog = if b then p_1 else p_2 for some Boolean b: Unify $?s$ with if b then $?s_1$ else $?s_2$ and $?o$ with if b then $?o_1$ else $?o_2$ for new unification variables $?s_1, ?s_2, ?o_1, ?o_2$. Make recursive calls to derive_core p_1 $?s_1$ $?o_1$ and derive_core p_2 $?s_2$ $?o_2$.
- Case prog = (fix $f\ n\ x$:= if $n = 0$ then $p_1(x)$ else $p_2(n, x, f(n-1))$) $n\ x$ for some x and a natural number n: If derive_core prog $?s'$ $?o'$ was run for some $?s'$, $?o'$ before, unify $?s$ with $?s'$ and $?o$ with $?o'$. Otherwise, unify $?s$ with if $n = 0$ then $?s_1$ else $?s_2$ and

$?o$ with if $n = 0$ then $?o_1$ else $?o_2$ for new unification variables $?s_1, ?s_2, ?o_1, ?o_2$ and run derive_core $p_1(x)$ $?s_1$ $?o_1$, derive_core $p_2(n, x$, (fix f n x := if $n = 0$ then $p_1(x)$ else $p_2(n, x, f(n-1)))$ $(n-1)$ $x)$ $?s_2$ $?o_2$, in a context with n and x introduced as new local variables.

- Case prog = Return v: Unify $?o$ with None. Insert $(?s, v)$ in a database db.

If derive_core finishes with the type of states of the form $q_0 \mid \cdots \mid q_n\ x \mid ?X$, for each $(?s, v)$ in the database db, unify $?s$ with $q_{n+1}\ v$ and let step $?s$ _ _ = Done v. Finally, for any pair of s and e such that step s e has not been defined yet, let step s e _ = Done v_0 for some v_0 to represent failure.

Procedure derive_core can easily be extended to the case that prog is a pattern-matching expression for an arbitrary algebraic type in a similar way with if-expressions. However, as mentioned earlier, our compiler does not uniformly support pattern-matching expressions for all algebraic types, because of difficulty in Coq of replacing `v1`, ..., `vm` in the term

```
match x with
| c1 a11 ... a1n ⇒ v1
...
| cm am1 ... amk ⇒ vm
end
```

with unification variables.

3.5 Coroutines

In this subsection, we consider programs with coroutines, finally revealing how we desugar coroutine constructs into standard functional code. Fig. 7 shows a coroutine and a parent. let_coro c :=

```
ex_coro s₀ =
  s₁ ← yield s₀;
  _ ← yield (s₀++s₁);
  Return tt

example =
  let_coro c := ex_coro in
  r₁ ← resume c ”Hello, ”;
  putStr r₁; (*Output : Hello, *)
  r₂ ← resume c ”world!”;
  putStr r₂; (*Output : Hello, world!*)
  Return tt
```

```
example′ =
  let_coro c := ex_coro in
  match c ”Hello, ” with
  | Eff yield a q ⇒
    (fun r₁ c ⇒
      putStr r₁;
      match c ”world!” with
      | Eff yield a q ⇒
        (fun r₂ c ⇒
          putStr r₂;
          Return tt) a q
      | _ ⇒ Return tt
      end) a q
  | _ ⇒ Return tt
  end
```

Fig. 7. A coroutine (ex_coro), a parent (example), and the parent without resume notation (example′)

... in ... is equivalent to let c := ... in ..., and the resume syntax desugars as follows:

```
r ← resume c a; p(c, r) ≡
match c a with
| Eff yield b q ⇒
  (fun r c ⇒ p(c, r)) b q
| Return _ ⇒ Return v₀
end
```

where v_0 is a value of the parent's return type that represents failure. Note that the identifier c is rebound in the fun-expression. Note also that the continuation q of the coroutine is explicitly captured and passed off as the new value of c. Therefore, in the example, even though we use the same identifier c for two resumes, these c's refer to different terms. The second refers to

```
fun $s_1$ ⇒   (_ ← yield ("Hello, "++$s_1$);
              Return tt)
```

Overall compilation proceeds as a bottom-up traversal through the tree of coroutines. Leaf coroutines can be compiled in the way explained in the last subsection. To compile each parent coroutine, we need a further procedure. Since resume in a parent is a pattern-matching on an effectful program, we first change each resume into a case analysis on the output of the state machine obtained from the coroutine. Concretely, we transform example from Fig. 7 into the intermediate form shown in Fig. 8. Here, init_coro and step_coro are the initial state and the step function of ex_coro. An algorithm for this transformation (again implemented in Ltac to generate not just target code but proof of its equivalence to source code) works as follows.

to_state prog =

- Case prog = let_coro c := coro in $p(c)$:
  Return let c := coro_init in to_state $p(c)$
- Case prog = $r \leftarrow$ resume c a; $p(c, r)$: Return

```
match step_coro $c$ yield $a$ with
| GoNext($c$, Some (_, $r$)) ⇒ to_state ($p(c, r)$)
| _ ⇒ Return $v_0$
end
```

- Case prog = if b then p_1 else p_2:
  Return if b then to_state p_1 else to_state p_2
- Case prog = (fix f n x := $p(f, n, x)$) n x:
  Return (fix f n x := to_state ($p(f, n, x)$)) n x
- Case prog = Return v: Return Return v

For the case prog = $r \leftarrow$ resume c; p, we need to associate the right step function step_coro to the coroutine variable c, as there may be multiple different kinds of coroutines in the program,

```
let c := init_coro in
match step_coro c "Hello, " with
| GoNext(s, Some (_, r1)) ⇒
   putStr r1;
   match step_coro s "world!" with
   | GoNext(_, Some (_, r2)) ⇒
      putStr r2;
      Return tt
   | _ ⇒ Return tt
   end
| _ ⇒ Return tt
end
```

Fig. 8. Intermediate form of the parent example

each originating from one explicit code body and then propagated across chains of yields. In our implementation, we put step_coro into the type of c as a dummy argument:

```
Definition coro_type A B C step :=
  A → t yield_eff (fun _ ⇒ A) (fun _ ⇒ B) C.
```

where `A, B, C` are types. Then, the step function is obtained by type inference. If the compiler needs to know the step function of a coroutine variable c, it infers the type of c, which is expected to be of the form `coro_type A B C step`, so then it recognizes `step` as the step function. Different coroutine variables in the source program are effectively grouped together union-find-style, as an implicit consequence of type inference, finding those clusters associated with the same pieces of coroutine source code. The choice of step function for a cluster is seeded for coroutine variables that refer to coroutine code directly, rather than using `resume`.

The proof of equality between a parent before and after transformation mostly appeals to the following lemma:

Lemma 2. If a coroutine c, a step function step, and a state s satisfy equiv step s $(c\ r)$ for any choice of r, then $r \leftarrow$ resume c a; $p(r, c)$ is equal to

$$\begin{array}{l}\text{match step } c \text{ yield } a \text{ with}\\ \mid\ \text{GoNext}(c, \text{Some }(\_, r)) \Rightarrow p(r, c)\\ \mid\ \_ \Rightarrow \text{Return } v_0\\ \text{end.}\end{array}$$

3.6 Coroutines Inside Data Structures

However, if the parent uses a container data structure holding child coroutines (e.g., a map from file descriptor to coroutine for that connected client), we need more steps. For example, suppose that a parent has the subterm

$$\begin{array}{l}\text{match nth\_error } l\ n \text{ with}\\ \mid\ \text{Some } c \Rightarrow r \leftarrow \text{resume } c\ a;\ p\\ \mid\ \text{None} \Rightarrow q\\ \text{end}\end{array}$$

where l is a list of child coroutines, and each coroutine in l has the same step function. nth_error l n returns Some c if the nth element of l is c and None if n is greater than or equal to the length of l. After the transformation, l will be a list l' of states corresponding to the coroutines. To apply Lemma 2, we must have the condition equiv step $(c\ r)$ s for the nth element c of l, the nth element s of l', and any r. Our proof strategy is to maintain the invariant that, whenever an input program to our compiler produces a list of coroutines, the compiled program generates a list of states related to the original list (and indeed we make sure that Coq's hint databases are configured to allow that proof to be found automatically when needed later), so that we can apply Lemma 2 as required. If we handle various data structures rather than just lists, we need to generalize this type of condition. For that purpose, we use a type class that is defined as follows, for F a container type family, parameterized on the type of data value stored within.

$$\begin{array}{l}\text{Class HasGenForall2 } F :=\\ \quad \{\text{GenForall2 : forall } A_1\ A_2,\\ \qquad (A_1 \rightarrow A_2 \rightarrow \text{bool}) \rightarrow F\ A_1 \rightarrow F\ A_2 \rightarrow \text{bool}\}\end{array}$$

For lists, GenForall2 p l_1 l_2 is defined as the proposition "p a_1 a_2 holds for any n, the nth element a_1 of l_1, and the nth element a_2 of l_2." Also, we have proven lemmas for operations on some data structures. For example, we have the next lemma.

Lemma 3. For any state s_0 and coroutine c_0, we have equiv step s_0 c_0 if each of the following three conditions are satisfied:

- GenForall2 (equiv_coro step) l_1 l_2
- nth_error l_1 n = Some s_0
- nth_error l_2 n = Some c_0

where l_1 is a list of states, l_2 is a list of coroutines, and n is a natural number.

Using this lemma, we teach our compiler how to handle nth_error lookups in lists of coroutines. When our system proves equivalence between source and compiled code, it automatically attempts to apply each preproven lemma about containers holding coroutines, discharging lemmas' side conditions. Recall again that we perform compilation as a kind of side effect of proof search, so this process of finding applicable lemmas also guides selection of code, making the lemma database a kind of extensible database of compilation hints, as well.

4 TLS IMPLEMENTATION

We evaluated our Coq library by using it to derive a state-machine implementation for TLS, the protocol behind secure Web browsing, implementing just a large enough subset of TLS that we can test with standard Web browsers. We relied on an existing Haskell library[4] for most of the pure parts of this subset, e.g. particular cryptographic ciphers and interchange formats. As our compiler does not need to modify pure code, having only the type signatures of these functions in Coq is enough, and the Haskell implementations are linked in during Haskell compilation, after the code produced by our compiler is extracted from Coq.

Nevertheless, our implementation includes everything needed for use in a conventional web server. Concretely, it generates session tickets and supports session resumption, but it refuses 0RTT and client authentication as allowed by the specification. The conservative selection of cryptographic algorithms currently supported (X25519, chacha20, poly1305) is not enough to claim full TLS1.3 support as per the RFC, but it is sufficient to interoperate with recent versions of all major browsers. To safely achieve acceptable performance using the remaining algorithms, the software would need to delegate cryptographic computations to platform-specific dedicated hardware and low-level implementations. This practical hurdle is orthogonal to the network-protocol-implementation challenges we tackle in this work.

4.1 Structure of the Implementation

Our TLS server implementation has two coroutines for each connection, and a parent (node 1 in Fig. 9) accepts new connections and spawns and resumes coroutines for them. One of the two coroutines for a connection handles parsing, decrypting, and encrypting of data (node 2 in Fig. 9). The other implements the TLS handshake protocol and yields primitive operations such as receiving a specific type of message (node 3 in Fig. 9). Coroutine 2 is the parent of coroutine 3. For example, if coroutine 3 yields `RecvClientHello`, coroutine 2 attempts to receive data from the peer, parse it as the requested message type, and resume coroutine 3 with the result of parsing. If some bytes are remaining at the end of parsing a `client_hello` message, they are stored in a buffer and will be consumed if coroutine 3 asks to receive another message. Also, if coroutine 3 yields `SetSecret (hash, cipher, secret, false)`, coroutine 2 will decrypt later messages with the hash algorithm `hash`, the cipher suite `cipher`, and the secret key `secret`. If the last argument of `SetSecret` is `true`, coroutine 2 will encrypt messages to send instead.

Coroutine 3 yields the following effectful operations.

[4]https://github.com/vincenthz/hs-tls

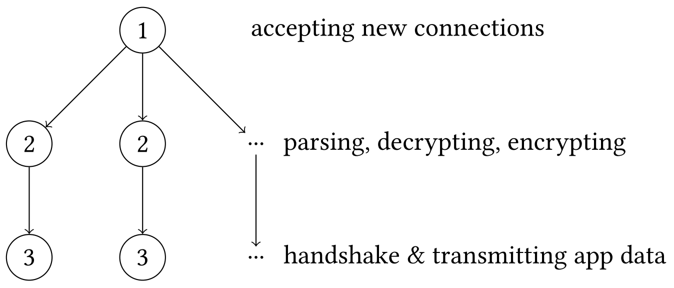
1
accepting new connections
2
2
···
parsing, decrypting, encrypting
3
3
···
handshake & transmitting app data

Fig. 9. Overview of our TLS implementation

- `RecvClientHello`
- `RecvFinished`
- `RecvCCS`
- `RecvAppData`
- `SendPacket packet`
- `GetRandomBytes n`
- `SetSecret (hash, cipher, secret, b)`
- `GetCurrentTime`
- `Close`: close this connection.
- `GroupGetPubShared g`: given a group `g`, generate a pair of a public key and a shared key.
- `MakeCertVerify (pub, priv, hashSig, tran)`: given a public key `pub`, private key `priv`, hash and signature algorithm `hashSig`, and transcript hash `tran`, generate a certificate verify.
- `SetPSK (sessID, sessInfo)`
- `SessionResume sessID`

Here is the key control flow of coroutine 3 that implements the last steps of Fig. 1, with some pure computations omitted.

```
  finEncoded ⟵ SendPacket ([HFinished fin]);
  sfSentTime ⟵ GetCurrentTime;
  _ ⟵ SetSecret (usedHash, cipher, ..., false);
  _ ⟵ SetSecret (usedHash, cipher, ..., true);
  fin ⟵ RecvFinished;
  cfRecvTime ⟵ GetCurrentTime;
  let resumptionMasterSecret := ... in
  _ ⟵ SetPSK ...;
  _ ⟵ SendPacket [HNewSessionTicket ...];
```

Concurrency between different sessions is handled by coroutine 1, which also coordinates access to shared state such as the session cache (`m`). The parent interacts with the environment using standard socket operations performed by coroutine 2 along with `accept`ing new connections. Each network-I/O operation specifies the OS-level socket as one of the arguments, and coroutine 1 is responsible for correlating this with TLS sessions: it keeps a map from OS sockets to coroutines (`coros`) and associates operations it passes through with socket identifiers. It is an invariant that every coroutine stored in the session map is waiting for network input; coroutine 1 performs local actions in an inner loop without reinserting and looking up the relevant session again and again:

```
outer m coros =
  event ⟵ yield ();
  match event with
  | Accept ⇒ ...
  | Receive (sa,r) ⇒
    let coro := find sa coros in
    inner r m sa coro
  end
inner r m sa coro =
  let ef := resume coro r in
  match ef with
  | SetPSK sid sess ⇒
      let m' := insert sid sess m in
      inner () m' sa coro
  | SessionResume sid ⇒
      let (sess, m') := remove sid m in
      inner (FromSessionResume sess) m' sa coro
  | _ ⇒
      yield (wrap sa ef);
      let coros' := replace sa coro coros in
      outer m coros'
  end
```

This example also illustrates the (small) extent to which writing an inherently stateful server is complicated by the use of a pure source language. From the perspective of the connection-specific coroutines, accessing shared state is an explicit effect – syntactically indistinguishable from e.g., generating a random number. The interleaving of operations by different child coroutines is decided by the control flow of the parent coroutines, in this case eagerly executing all actions of the last coroutine that received input until it becomes blocked. The handlers for `SetPSK` and `SessionResume` are similar in concept to getters and setters in languages such as Java, but here the child coroutine does *not* have a reference to the parent coroutine – it just yields operations.

4.2 Code Size

We find the coroutine style to offer at least a moderate advantage in raw code size, though we think the biggest payoff is in code *complexity*. Still, the former is easier to measure, so here are some figures. Our file `TLS.v` with the main protocol implementation, up to and including the main loop, runs to 1377 lines. The roughly comparable files in ocaml-tls[5] total 3379 lines. That library does implement more versions and features, but our analysis excludes files dedicated to them, so feature complexity should differ by less than 2x. Further, we classify 445 lines of our code as declarations, workarounds, and boilerplate within reach of Haskell's `deriving`, whereas ocaml-tls's code lacks these.

4.3 Bug Finding and Fixing

We have tested our implementation manually and by fuzzing. The errors occurring during testing were divided into two main cases: (1) a peer failed to decrypt messages from our implementation, and (2) our implementation did not abort the handshake when a peer sent some type of illegal message. Case (1) is more difficult to debug, since there are many different ways to encrypt incorrectly. For our particular bug, the problem was that our implementation did not handle sequence numbers[6]

[5] `reader.ml`, `engine.ml`, `config.ml`, `handshake_common.ml`, `handshake_server13.ml`, `core.ml`, and `state.ml`
[6] https://tools.ietf.org/html/rfc8446#section-5.3

and computed the transcript hash including change_cipher_spec, which should not actually be included[7]. The latter required a small change, and the former was resolved by adding one more state field in the loop of coroutine 2 so that it can count messages received or sent. Case (2) was easily fixed by adding case analyses to detect illegal parameters in the peer's messages, except that one bug was caused by the hs–tls function we used for decryption[8]. We see this hs-tls issue as a result of trying to compensate for other state-machine imprecision in the decryption function: the code was looking at the message header to see whether it should be decrypted or not, passing through plaintext messages, whereas in actuality the TLS handshake dictates that the next message must be authenticated and decrypted. We believe that if we had restricted reuse of pure code from hs-tls to single-purpose functional units (cryptography or serialization, not both) we would have not made the same mistake because our code structure is driven by protocol tasks such as RecvFinished.

4.4 Performance

We show the result of benchmarking our code against OpenSSL (with Nginx[9] 1.19.4) and the Haskell library we reused pure functions from (with Warp[10]). We use wrk[11] as a benchmarking tool and compared our server with Nginx and a simple combination of Warp and hs–tls shown below:

```
header = [("Content-Type", "text/plain"),
          ("Content-Length", "6")]
app _ respond = respond $ respondLBS status200 header "Hello!"
let tls = tlsSettings "cert.pem" "key.pem" in
runTLS tls (setPort 4433 defaultSettings) app
```

We exercised each server with the following commands.

```
wrk -t1 -c1 -d30s https://localhost:4433
wrk -t1 -c40 -d30s https://localhost:4433
```

This means that 1 and 40 HTTP connections are established using 1 thread with duration 30 seconds. The results are in Fig. 10.

First, let us consider the results with one client thread. In both latency and throughput, our derived implementation comes within 50% of the performance of either of the more established alternatives. We conjecture that the gap comes mostly from inefficient use of data structures, e.g. for simplicity our compiler represents state not in one algebraic datatype but rather with nested sum types. Also, our longer-term goal is to extend our code-derivation process in Coq to produce C syntax trees rather than native functional programs, allowing us to remove more overheads thanks to fine-grained control over data representation. Still, it seems we do have a practical implementation of server-side TLS for security-critical applications.

Next, we consider the results with 40 client threads. We see this experiment as more of a worst-case preview driving fruitful future work, since our implementation does not take advantage of multiple cores, while our competitors do. Such a benchmark is also useful as a stress test of concurrency logic in our code, and we do exhibit functionally correct behavior under this load. Unsurprisingly, the other implementations with their multicore execution perform several times better than we do, though again it seems we are within the window where an especially paranoid user might prefer our proved server under moderate load.

[7]https://tools.ietf.org/html/rfc8446#section-4.4.1
[8]https://github.com/vincenthz/hs-tls/issues/438. Maintainers have classified this issue as "bug", "security", and "moderate".
[9]https://www.nginx.com/
[10]https://hackage.haskell.org/package/warp
[11]https://github.com/wg/wrk

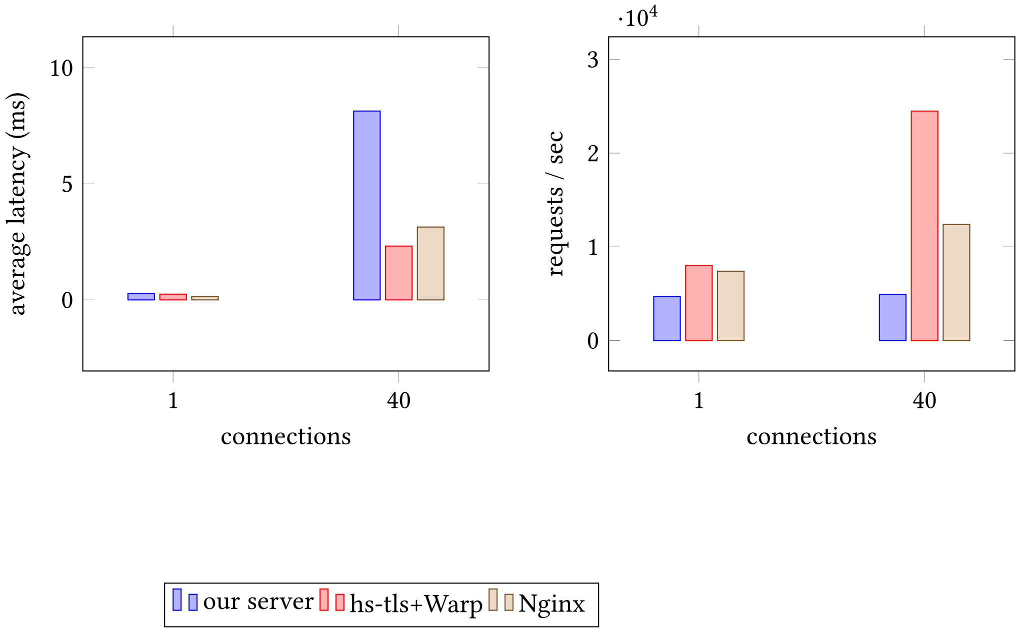

Fig. 10. Performance-experiment results

Not shown in the figure is compile time, which is a significant hassle with our current implementation. Though the algorithm sketched earlier is simple enough to pose no performance challenges in a freestanding implementation, we ran into bottlenecks with Coq's proof engine. Our compiler makes heavy use of unification variables (evars), which were not implemented with large-scale automated use in mind. To give some intuition for why general-purpose evars are not trivial to implement in a performant manner, consider that taking a proof step in a goal that contains some evars requires the previous context of each evar to be expressed in terms of the new context – even if the overall context of the goal did not change, the evars may appear under different binders than before. This inherent challenge is further compounded by non-performance-conscious engineering choices such as allowing evars to remain in the proof engine even when an instantiation is known and representing evar context substitutions as immutable arrays. As a consequence, even straightforward tactic-programming patterns are surprisingly slow. For example, a proof that introduces n variables from the goal (which is just another evar) into the proof-engine context takes $O(n^3)$ time.[12] Our compiler is bound by the same underlying inefficiencies: it takes hours and tens of gigabytes of memory to compile our full TLS example (or, more accurately, to compile its proof), but we are not discouraged, because (1) our implementation can be seen more as a proof of an algorithm, inspiring future freestanding implementations, and (2) we hope for future performance improvements to Coq's proof engine, useful not just for this application.

5 RELATED WORK

5.1 TLS State Machines

Our case study was inspired by the TLS implementation in the standard library of the Go programming language. The Go language runtime provides a blocking I/O interface using an efficient

[12] https://github.com/coq/coq/issues/8244

user-space implementation of multithreading on top of event-based I/O facilities provided by the operating system. This approach allows for intuitive control flow in TLS code and high performance even with many concurrent connections, but it creates an obstacle for embedding of Go code in larger (non-Go) applications or microcontroller systems.

Widely deployed TLS implementations rely on handwritten explicit state machines. The programming errors associated with this approach have been identified as the root cause of numerous severe security issues and studied in several papers: Beurdouche et al. [2015] identify "state machine bugs and several critical security vulnerabilities" due to "incorrect compositions of individually correct state machines" using systematic testing. de Ruiter and Poll [2015] report that "new security flaws were found (in GnuTLS, the Java Secure Socket Extension, and OpenSSL)" using protocol state fuzzing. This kind of issue can be discovered even without access to the source code, as demonstrated on Windows SChannel [Yadav and Sadhukhan 2019].

We discussed this state of affairs with David Benjamin, maintainer of BoringSSL and frequent contributor of OpenSSL and Go TLS. What we heard confirmed that implementers are also worried about these issues. Further, the current best practice to avoid them (applied in BoringSSL) is to make sure control flow of state-transition functions is driven primarily by the state-machine state and only when necessary by network input. We believe the branches on the state correspond directly to our own reification of inter-input control flow into a variant type of possible suspension points during coroutine compilation.

5.2 Fully Compile-Time Coroutine Implementations

There has been considerable exploration of compile-time implementation of coroutines in the context of Rust, Scala, and C++, but we are not aware of any widely deployed code base using these features. The Rust async/await feature is the most mature of the three and was recently enabled in the stable release of the compiler. The main differences from our implementation are that (1) the mechanism is primarily used for wakeup only (I/O is handled through side effects instead of yield/resume arguments), and (2) saving and restoring of mutable and destructor-carrying local variables significantly complicates the implementation [Mandry 2019]. The implementation of coroutines through Scala metaprogramming does not seem to have gained adoption, but the corresponding paper [Prokopec and Liu 2018] comprehensively describes what we believe are the same challenges that Rust faced earlier (with slightly different design choices for nested coroutines). The C++ standard has not yet converged on a single proposal for coroutine implementation or semantics, but the Coroutines TS [Nishanov 2018] proposal that is currently implemented most widely differs from Rust's (and ours) further in that coroutine compilation is performed after most compiler optimizations as opposed to in the frontend [Smith et al. 2019].

There is a wide variety of platform-specific or otherwise limited implementations of coroutines relying on an unmodified compiler for an existing language, usually C. Even though these implementations target the same applications and systems as we hope our work will eventually be used on, the key distinguisher from our work is prioritization of portability and compatibility with existing codebases – the code our approach produces does not rely on any platform-specific features a handwritten state machine would not need. Example techniques include using `longjmp` or assembly code to change the stack pointer, using the POSIX signal stack to store stacks of coroutines, and leaving storage of local variables up to the programmer and using Duff's device for control flow. We recommend the paper describing the Gnu Portable Threads library [Engelschall 2000] for a comparison of implementation strategies along with the more recent systematic review [Belson et al. 2019] for its discussion of how application context and requirements seem to have formed these approaches.

5.3 Runtime-Implemented Coroutines

Coroutines and similar control-flow constructs are much more commonly implemented using dedicated features of the language runtime. As the maintainers of an application would be unable or unwilling to add a runtime system to their program to use a TLS library, these options are less attractive for our use case. However, these implementations have played an important role in popularization of these facilities. For example, the earliest-dated occurrence we found for the catchphrase "coroutines are to state machines what recursion is to stacks" is in a 2009 blog post [Bendersky 2009] explaining how Python generators (lacking delegation at the time) could be used elegantly to implement a simple wire protocol, contrasting it with a handwritten state machine. The resemblance to our usage and work on algebraic effects is striking. Shortly thereafter (2011), C# 2.0 introduced `async`/`await` syntax and type-distinguished coroutines we are now used to. Similar features are now being added to Java, Scala, JavaScript, etc. See James and Sabry [2011] for an earnest review of mainstream languages' adoption of "small" coroutine-like features.

5.4 Coroutines, Continuations, and Algebraic Effects

First-class coroutines correspond closely to one-shot continuations (`call1cc` and similarly restricted algebraic effects [Kawahara and Kameyama 2020; Moura and Ierusalimschy 2009]). Further, coroutines whose state can be duplicated correspond to general continuations (`callcc`) [Prokopec and Liu 2018] and presumably general algebraic effects. The above equivalences are originally stated for stackful and delimited coroutines and continuations, but unfortunately without presenting a precise definition of these qualifiers. We will briefly revisit the key constructions behind the equivalences, restate the requirements, and discuss how this applies to our system.

Whether a coroutine is stackful seems to be used to refer to multiple related but nonidentical notions.

(1) Same stack: In common use, coroutines are considered stackful iff the activation records of functions and coroutines are stored in the same stack. This criterion is applied to representation of coroutines when they are suspended, i.e. sharing a stack during execution is not sufficient. For example, the `cppreference.com` entry on coroutines[13] describes the C++ implementation as follows: "Coroutines are stackless: they suspend execution by returning to the caller and the data that is required to resume execution is stored separately from the stack."
(2) Delegation forced, all functions are coroutines: Moura and Ierusalimschy [2009] reject Python's `yield` operator as an implementation of coroutines on the basis of the following behavior: given a coroutine calling a function calling a coroutine, the inner coroutine yielding would transfer control to the intermediate function, not further out. This choice is essential for the unconditional statement of the equivalence between coroutines and continuations in an untyped setting, as `call1cc` can act from inside any function, whereas Python uses *different* constructs for calling a function and delegating to a coroutine.
(3) Delegation supported, coroutines may be distinguished from functions: Prokopec and Liu [2018] accept Python's `yield` as an implementation of coroutines while referencing the `yield from` operator for one coroutine delegating to another, even though `yield from` is just syntactic sugar for calling `yield` in a loop. The implementation of coroutines contributed in the work itself is in a typed setting where coroutines are distinguished from functions. We interpret the extended claim of equivalence between copyable coroutines and multi-shot continuations as applying to the coroutine-typed fragment of the language and find it satisfying enough.

[13] https://en.cppreference.com/w/cpp/language/coroutines

Our compile-time implementation of coroutines is stackless in the first sense (we do not use the function call stack for representing suspended coroutines), stackless in the second sense (coroutines and functions have disjoint types), but stackful in the last sense (our TLS implementation includes examples of one coroutine delegating to another). As our compiler represents suspended coroutine state using a simple nonfunction type in a purely functional language, our coroutines are trivially copyable.

We believe all implementations discussed here are delimited, even though the delimiter is often implicit. For example, in Python `for` loops act as delimiters to `yield`. In our language, `resume` acts as an implicit delimiter, and delegation is a derived notion similarly to Python's `yield from`. A truly undelimited system (e.g. Cilk) would be unable directly to simulate the `handle` operator of algebraic effects, as every yield hands control to the runtime system, not user code.

The usage of coroutines in our TLS library resembles the pattern of algebraic effects with handlers in that a value a coroutine yields requests some action to be performed, and the response sent back to the coroutine is the result of that action. However, the (handwritten) code encoding these requests as coroutines is substantially different from outputs of the generic translation of effect-handler code into coroutines presented by Kawahara and Kameyama [2020]. In particular, we seem to have somehow avoided having to introduce a continuation-carrying `Rehandle` effect wrapper when yielding unhandled effects further out from an effect handler. While we believe our compiler would work on the coroutine code generated by their systematic translation, we are concerned that a handler translating `Rehandle (Printf ..) k` into more fine-grained effects (`putchar`) would be compiled into a state machine whose state type includes the continuation `k`, thus complicating further efficient compilation. It would be interesting to investigate whether the encoding of algebraic effects as coroutines used in our case study can be generalized into a systematic translation that does not introduce explicit continuations.

6 FUTURE WORK

While we believe that the compilation algorithm presented in this paper does everything necessary to convert coroutines to state machines, there are several optimizations that we expect would greatly benefit the code thus generated (as well as similar handwritten code). We will sketch a couple that we believe would be the most impactful.

Switching-Optimized Variant Layout. The memory layout of the cases of the variant type representing a coroutine state should be chosen to minimize data movement when switching from one case of the variant to another. In C code for TLS, the state variant is (partially or fully) flattened into a struct, preferably reusing the same field for semantically related values of two variant cases. Minimizing state-record size is important to support more concurrent connections using a fixed amount of RAM or to enable use of TLS on memory-constrained devices. To achieve the same benefit automatically, the code would need to be analyzed for frequency of switches between each pair of variant cases; an implementation specialized to coroutines could also just remember the successor cases for each case of the variant during state-machine generation. Then a small integer linear program could be solved for each type, where the variables are the offsets of fields in each case, the constraints enforce nonoverlap of fields that appear in the same case, and the objective is to minimize a linear combination of state size and data-movement cost estimated based on recorded transitions and the size of each field. This is especially important if large structures (e.g., state of subcoroutines) are stored inline in the struct (without pointer indirection) as is common in hand-optimized network programming.

Memory-Management-Oriented Compilation. Further, inlining fields that are themselves records (as opposed to keeping pointers to them) can reduce the number of cache misses, improving

```
GatherData(Socket *ss, Gather *gs, int flags) { //[152 lines]
  while (gs→remainder) {
    nb = nonblocking_receive(ss, bp, gs→remainder, flags);
    if (!nb) {
      return 0;
    //[caller should call GatherData again when more bytes are available]
    }
    if (gs→remainder -= nb > 0) {
      continue;
    }
    /* have received entire record header, or entire record. */
    switch (gs→state) {
      case GS_HEADER:
        //[remainder length parsing and potential SendAlert omitted]
      case GS_DATA:
        return 1; }
  }
}
GatherCompleteHandshake(Socket *ss, int flags) { //[175 lines]
  do {
    rv = GatherData(ss, &ss→gs, flags);
    if (rv ≤ 0) {
      return rv;
    }
    rv = HandleRecord(ss, &ss→gs);
  } while (!ss→ssl3.hs.ws == idle_handshake || rv != SECSuccess);
  return rv; }
```

Fig. 11. NSS code for GatherData

performance on workloads with many connections. Finally, it is also desirable to eliminate redundant memory allocation and deallocation operations; some libraries take this to the extreme and can be used without even implementing `malloc` and `free`. The Glasgow Haskell Compiler performs some struct unboxing based on simple heuristics, but in our experience it is significantly behind the level of optimization commonly found in network-protocol implementations written in C. Compiling functional code into imperative code that mutates values in-place is challenging, even in the case of immutable data and nonrecursive functions. We direct the reader to Reinking et al. [2021]; Ullrich and de Moura [2020] for a literature review and an implementation.

Handler Inlining. Our TLS library uses coroutine yield and resume as algebraic effects and handlers to give coroutines access to program state stored outside the coroutines. For example, the handshake coroutine interacts with the connection-resumption database and record-layer buffering and encryption state. This means that our coroutines frequently yield just to be resumed immediately after accessing or binding a local variable in another coroutine, and that the generated state-transition functions return just to be called again with new inputs. We expect these indirections could be eliminated by an optimization that moves code that according to constant propagation always follows a function into that function (passing in a pointer to relevant data if necessary), when combined with standard tail-call optimization and applied recursively.

7 CONCLUSION

Writing network-protocol code directly in nested-state-machine style has challenged developers and led to several high-profile security vulnerabilities. Higher-level implementations with threads have been easier to debug but also harder to integrate with vanilla C code, due to assumptions about runtime systems. We demonstrated achieving the best of both worlds while adding mechanized proof of correct compilation. We noted how freer monads provide an effective setting for encoding nested coroutines within a purely functional language, and we prototyped a proof-generating compiler from such programs to flat state machines. Though our implementation has run afoul of some performance-scaling challenges in Coq, it was sufficient to generate a correct TLS implementation whose performance is comparable to an idiomatic Haskell alternative's, at least at low concurrency levels. We plan to close that performance gap by extending proof-generating derivation to C code.

ACKNOWLEDGMENTS

This work was supported in part within the National Science Foundation Expedition on the Science of Deep Specification (award CCF-1521584), in addition to gifts from Amazon Web Services, Google, and the Tezos Foundation.

REFERENCES

Gilles Barthe, Benjamin Grégoire, and Santiago Zanella-Béguelin. 2009. Formal Certification of Code-Based Cryptographic Proofs. In *36th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2009*. ACM, 90–101. http://software.imdea.org/~szanella/Zanella.2009.POPL.pdf

Bruce Belson, Jason Holdsworth, Wei Xiang, and Bronson Philippa. 2019. A Survey of Asynchronous Programming Using Coroutines in the Internet of Things and Embedded Systems. *ACM Trans. Embed. Comput. Syst.* 18, 3, Article 21 (June 2019), 21 pages. https://doi.org/10.1145/3319618 https://arxiv.org/pdf/1906.00367.pdf

Eli Bendersky. 2009. Co-routines as an alternative to state machines. https://eli.thegreenplace.net/2009/08/29/co-routines-as-an-alternative-to-state-machines/ Retrieved 2020-08-20.

Lennart Beringer, Adam Petcher, Katherine Q. Ye, and Andrew W. Appel. 2015. Verified Correctness and Security of OpenSSL HMAC. In *24th USENIX Security Symposium*. 207–221. https://www.usenix.org/system/files/conference/usenixsecurity15/sec15-paper-beringer.pdf

B. Beurdouche, K. Bhargavan, A. Delignat-Lavaud, C. Fournet, M. Kohlweiss, A. Pironti, P. Strub, and J. K. Zinzindohoue. 2015. A Messy State of the Union: Taming the Composite State Machines of TLS. In *2015 IEEE Symposium on Security and Privacy*. 535–552. https://smacktls.com/smack.pdf

B. Blanchet. 2006. A computationally sound mechanized prover for security protocols. In *2006 IEEE Symposium on Security and Privacy (S&P'06)*. 15 pp.–154. https://doi.org/10.1109/SP.2006.1

Adam Chlipala. 2021. Skipping the Binder Bureaucracy with Mixed Embeddings in a Semantics Course (Functional Pearl). In *Proc. ICFP*. http://adam.chlipala.net/papers/FrapICFP21/

Joeri de Ruiter and Erik Poll. 2015. Protocol State Fuzzing of TLS Implementations. In *24th USENIX Security Symposium (USENIX Security 15)*. USENIX Association, Washington, D.C., 193–206. https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/de-ruiter

Benjamin Delaware, Clément Pit-Claudel, Jason Gross, and Adam Chlipala. 2015. Fiat: Deductive Synthesis of Abstract Data Types in a Proof Assistant. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, January 15-17, 2015*. 689–700. https://doi.org/10.1145/2676726.2677006

Benjamin Delaware, Sorawit Suriyakarn, Clément Pit-Claudel, Qianchuan Ye, and Adam Chlipala. 2019. Narcissus: Correct-By-Construction Derivation of Decoders and Encoders from Binary Formats. In *Proc. ICFP* (Berlin, Germany). http://adam.chlipala.net/papers/NarcissusICFP19/

Ralf S. Engelschall. 2000. Portable Multithreading: The Signal Stack Trick for User-Space Thread Creation. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference* (San Diego, California) *(ATC '00)*. USENIX Association, USA, 20. https://www.usenix.org/legacy/publications/library/proceedings/usenix2000/general/full_papers/engelschall/engelschall.pdf

Andres Erbsen, Jade Philipoom, Jason Gross, Robert Sloan, and Adam Chlipala. 2019. Simple High-Level Code For Cryptographic Arithmetic – With Proofs, Without Compromises. In *IEEE Security & Privacy* (San Francisco, CA, USA). http://adam.chlipala.net/papers/FiatCryptoSP19/

Roshan P James and Amr Sabry. 2011. Yield: Mainstream delimited continuations. In *First International Workshop on the Theory and Practice of Delimited Continuations (TPDC 2011)*. 20 pages. https://legacy.cs.indiana.edu/~sabry/papers/yield.pdf

David Kaloper-Meršinjak, Hannes Mehnert, Anil Madhavapeddy, and Peter Sewell. 2015. Not-Quite-so-Broken TLS: Lessons in Re-Engineering a Security Protocol Specification and Implementation. In *Proceedings of the 24th USENIX Conference on Security Symposium* (Washington, D.C.) *(SEC'15)*. USENIX Association, USA, 223–238.

Satoru Kawahara and Yukiyoshi Kameyama. 2020. One-Shot Algebraic Effects as Coroutines. In *Trends in Functional Programming*, Aleksander Byrski and John Hughes (Eds.). Springer International Publishing, Cham, 159–179. https://link.springer.com/content/pdf/10.1007%2F978-3-030-57761-2_8.pdf

Masashi Kikuchi. 2014. How I discovered CCS Injection Vulnerability. http://ccsinjection.lepidum.co.jp/blog/2014-06-05/CCS-Injection-en/index.html

Oleg Kiselyov and Hiromi Ishii. 2015. Freer Monads, More Extensible Effects. In *Proceedings of the 2015 ACM SIGPLAN Symposium on Haskell* (Vancouver, BC, Canada) *(Haskell '15)*. Association for Computing Machinery, New York, NY, USA, 94–105. https://doi.org/10.1145/2804302.2804319

Nicolas Koh, Yao Li, Yishuai Li, Li-yao Xia, Lennart Beringer, Wolf Honoré, William Mansky, Benjamin C. Pierce, and Steve Zdancewic. 2019. From C to Interaction Trees: Specifying, Verifying, and Testing a Networked Server. In *Proceedings of the 8th ACM SIGPLAN International Conference on Certified Programs and Proofs* (Cascais, Portugal) *(CPP 2019)*. Association for Computing Machinery, New York, NY, USA, 234–248. https://doi.org/10.1145/3293880.3294106

Adam Langley. 2014. Early ChangeCipherSpec Attack. https://www.imperialviolet.org/2014/06/05/earlyccs.html

Xavier Leroy. 2009. A Formally Verified Compiler Back-end. *J. Autom. Reason.* 43, 4 (Dec. 2009), 363–446. http://gallium.inria.fr/~xleroy/publi/compcert-backend.pdf

Tyler Mandry. 2019. How Rust optimizes async/await. https://tmandry.gitlab.io/blog/posts/optimizing-await-1/ Retrieved 2020-08-20.

Ana Lúcia De Moura and Roberto Ierusalimschy. 2009. Revisiting Coroutines. *ACM Trans. Program. Lang. Syst.* 31, 2, Article 6 (Feb. 2009), 31 pages. https://doi.org/10.1145/1462166.1462167 http://www.cs.tufts.edu/comp/250RTS/archive/roberto-ierusalimschy/revisiting-coroutines.pdf

Gor Nishanov. 2018. N4760: Working Draft, C++ Extensions for Coroutines.

Adam Petcher and Greg Morrisett. 2015. The Foundational Cryptography Framework. In *Proceedings of the 4th International Conference on Principles of Security and Trust - Volume 9036*. Springer-Verlag New York, Inc., New York, NY, USA, 53–72. https://doi.org/10.1007/978-3-662-46666-7_4 http://adam.petcher.net/papers/FCF.pdf

Simon L. Peyton Jones and Philip Wadler. 1993. Imperative Functional Programming. In *Proceedings of the 20th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Charleston, South Carolina, USA) *(POPL '93)*. Association for Computing Machinery, New York, NY, USA, 71–84. https://doi.org/10.1145/158511.158524

Clément Pit-Claudel, Peng Wang, Benjamin Delaware, Jason Gross, and Adam Chlipala. 2020. Extensible Extraction of Efficient Imperative Programs with Foreign Functions, Manually Managed Memory, and Proofs. In *IJCAR'20: Proceedings of the 9th International Joint Conference on Automated Reasoning* (Paris, France).

G. Plotkin and M. Pretnar. 2008. A Logic for Algebraic Effects. In *2008 23rd Annual IEEE Symposium on Logic in Computer Science*. 118–129. https://doi.org/10.1109/LICS.2008.45

Aleksandar Prokopec and Fengyun Liu. 2018. Theory and Practice of Coroutines with Snapshots. In *32nd European Conference on Object-Oriented Programming, ECOOP 2018, July 16-21, 2018, Amsterdam, The Netherlands.* 3:1–3:32. https://doi.org/10.4230/LIPIcs.ECOOP.2018.3 https://drops.dagstuhl.de/opus/volltexte/2018/9208/pdf/LIPIcs-ECOOP-2018-3.pdf

Jonathan Protzenko, Bryan Parno, Aymeric Fromherz, Chris Hawblitzel, Marina Polubelova, Karthikeyan Bhargavan, Benjamin Beurdouche, Joonwon Choi, Antoine Delignat-Lavaud, Cédric Fournet, Natalia Kulatova, Tahina Ramananandro, Aseem Rastogi, Nikhil Swamy, Christoph M. Wintersteiger, and Santiago Zanella-Béguelin. 2020. EverCrypt: A Fast, Verified, Cross-Platform Cryptographic Provider. In *IEEE Symposium on Security and Privacy*. IEEE. https://www.microsoft.com/en-us/research/publication/evercrypt-a-fast-veri%ef%ac%81ed-cross-platform-cryptographic-provider/

Alex Reinking, Ningning Xie, Leonardo de Moura, and Daan Leijen. 2021. Perceus: Garbage Free Reference Counting with Reuse. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (Virtual, Canada) *(PLDI 2021)*. Association for Computing Machinery, New York, NY, USA, 96–111. https://doi.org/10.1145/3453483.3454032

John C. Reynolds. 1972. Definitional Interpreters for Higher-Order Programming Languages. In *Proceedings of the ACM Annual Conference - Volume 2* (Boston, Massachusetts, USA) *(ACM '72)*. Association for Computing Machinery, New York, NY, USA, 717–740. https://doi.org/10.1145/800194.805852

Richard Smith, Daveed Vandevoorde, Geoffrey Romer, Gor Nishanov, Nathan Sidwell, Iain Sandoe, and Lewis Baker. 2019. Coroutines: Language and Implementation Impact. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2019/p1492r0.pdf P1492 R0.

Sebastian Ullrich and Leonardo de Moura. 2020. Counting Immutable Beans: Reference Counting Optimized for Purely Functional Programming. In *International Symposium on Implementation and Applicationof Functional Languages (IFL'19)*. New York, NY, USA, 12 pages. https://arxiv.org/abs/1908.05647

Li-yao Xia, Yannick Zakowski, Paul He, Chung-Kil Hur, Gregory Malecha, Benjamin C. Pierce, and Steve Zdancewic. 2020. Interaction trees: representing recursive and impure programs in Coq. *Proc. ACM Program. Lang.* 4, POPL (2020), 51:1–51:32. https://doi.org/10.1145/3371119 Distinguished paper award.

Tarun Yadav and Koustav Sadhukhan. 2019. Identification of Bugs and Vulnerabilities in TLS Implementation for Windows Operating System Using State Machine Learning. In *Security in Computing and Communications*, Sabu M. Thampi, Sanjay Madria, Guojun Wang, Danda B. Rawat, and Jose M. Alcaraz Calero (Eds.). Springer Singapore, Singapore, 348–362. https://arxiv.org/pdf/1902.07471.pdf

Katherine Q. Ye, Matthew Green, Naphat Sanguansin, Lennart Beringer, Adam Petcher, and Andrew W. Appel. 2017. Verified Correctness and Security of MbedTLS HMAC-DRBG. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security* (Dallas, Texas, USA) *(CCS '17)*. Association for Computing Machinery, New York, NY, USA, 2007–2020. https://doi.org/10.1145/3133956.3133974

Hengchu Zhang, Wolf Honoré, Nicolas Koh, Yao Li, Yishuai Li, Li-yao Xia, Lennart Beringer, William Mansky, Benjamin C. Pierce, and Steve Zdancewic. 2021. Verifying an HTTP Key-Value Server with Interaction Trees and VST. In *12th International Conference on Interactive Theorem Proving, ITP 2021, June 29 to July 1, 2021, Rome, Italy (Virtual Conference) (LIPIcs, Vol. 193)*, Liron Cohen and Cezary Kaliszyk (Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 32:1–32:19. https://doi.org/10.4230/LIPIcs.ITP.2021.32