Programming TLS in Isabelle/HOL

Andreas Lochbihler and Marc Züst

Institute of Information Security, ETH Zurich, Switzerland

Abstract. Isabelle/HOL is not just a theorem prover, it has become a functional programming language. Algebraic datatypes and (pure) recursive functions are defined with various packages and compiled to executable code with the code generator. In this work, we explore whether and how this programming language is suitable for developing applications, which are stateful, interact with the environment, and use external libraries. To that end, we have implemented a prototype of the TLS network protocol as a case study. We present a model of interaction in HOL and its compilation, and discuss on the challenges in application development that the theorem prover/HOL Isabelle poses.

1 Introduction

Fourteen years ago, Berghofer and Nipkow implemented the first code generator for Isabelle/HOL [2]. Today, the code generator [16,17] translates an executable subset of higher-order logic to the target languages Standard ML, OCaml, Haskell, and Scala. It corresponds to a functional programming language with algebraic (co)datatypes [3,6,38], recursively defined functions [6,22,23], type classes [18], and Prolog-like inductive definitions [1]. Code generation has become a valuable part of Isabelle; it supports understanding of specifications [33,34] and debugging theorems [7] and enables proofs by reflection [9,37]. Several applications [13,21,29,32,36] have been implemented in Isabelle/HOL's functional language, verified with Isabelle and compiled to the target languages.

In these applications, the generated code consists of pure functions. Some wrapper code (written manually outside of Isabelle/HOL) triggers their evaluation and provides the user interface. The extracted functions are self-contained, i.e., everything they depend on is written in Isabelle/HOL. This makes sure that the generated code is partially correct by the meta-theory of the code generator.[1] These applications show that Isabelle/HOL supports the development of self-contained batch-style functional programs sufficiently well.

But what happens if we go beyond this kind of programs? Can an Isabelle/HOL application be stateful, use external libraries, and interact with the environment? What challenges does developing an application in Isabelle/HOL

[1] For efficiency reasons, a few types like `bool` and `integer` and their operations are replaced by target-language primitives. Nevertheless, these types are definitionally constructed in Isabelle—when proofs use evaluation, their correctness relies on the replacement being sound.

face? To find answers to these questions, we have conducted a case study with the Transport Layer Security (TLS) protocol [12] (see §2 for an introduction). We have implemented TLS in Isabelle2013-2 and generated an executable Haskell implementation. TLS is a good case study, because

relevant the TLS protocol is widely used in the real world and security-critical;
interactive the implementation interacts over the network with an unknown environment, possibly several times during one invocation;
stateful the connection state must be passed on between consecutive transmission requests of the application layer; and
comparable TLS has previously been implemented in functional languages [4], so we can discuss Isabelle/HOL's peculiarities (§7).

In this work, we discuss the challenges and present our approaches and design decisions. In particular,

– we present an executable HOL model of probabilistic interactive programs (§3);
– we show how to import library functions from the target languages into HOL and use them in the implementation (§4); and
– we discuss how HOL's restrictive type system prevents straightforward approaches to monadic programming (§5) and state passing (§6), and show how to circumvent the restrictions.

Large interactive and stateful applications face the same challenges. Thus, we hope that our approaches can be re-used in other contexts.

2 Overview of TLS and the case study

The TLS protocol ensures the security (confidentiality, integrity, and authenticity) of data transmitted over an open network. In the OSI network stack, TLS operates on the session and presentation layers. It is the basis of many application protocols such as HTTPS, SSH, and SMTPS. In a first phase called handshake, communication partners authenticate with certificates (if desired) and exchange a secret session key. Subsequent data from the application layer is encrypted with the session key to achieve confidentiality. Message authentication codes guarantee integrity and authenticity of the data.

Internally, TLS uses five sub-protocols arranged in two layers. On the lower layer, the record protocol encrypts, decrypts, authenticates, and verifies the data according to the current security parameters. The others (handshake, change cipher spec (CCS), alert, and application) on the higher layer communicate through the record protocol. The handshake protocol negotiates the cryptographic algorithms and generates and exchanges the session key. The CCS protocol signals a change in the security parameters to the peer. The application protocol takes care of transmitting the application's data. The alert protocol signals errors which typically abort the session immediately.

In our case study, we have implemented TLS in version 1.0 and a simple command-line client and a single-threaded server to exchange data. The TLS implementation supports fragmentation, renegotiation, session resumption, and the key exchange algorithms RSA and anonymous Diffie-Hellman, and the ciphers RC4 and DES (for the encryption of application data).[2] It uses a TCP socket for data transmission. Full details on the TLS implementation can be found in [41].

3 Interactive programs

In our case study, the TLS client and server interact with the user and the network. We program them in Isabelle/HOL, but we run them in some target language of the code generator. Therefore, they must meet the requirements of the code generator. In this section, we present a HOL model of interactive programs and how the code generator compiles them.

For a function $\mathsf{client} :: \alpha \Rightarrow \beta$ (for some types α and β), the code generator needs equational theorems of the form $\mathsf{client}\ \ldots = \ldots$ as the code equation. All HOL functions are pure, i.e., their result only depends on the inputs given as parameters. The client's output, however, depends on the input received from the network, and there is no way to know the input from the network already when we launch the client. Therefore, we model the interaction in HOL explicitly as a codatatype, whereas all computations between two interactions are ordinary HOL functions.

3.1 Probabilistic interactive values

Our model of interaction is inspired by Harrison's reactive resumption monad [19]. Basically, an interactive value either is a pure result α or performs IO, i.e., it sends some output or request o to the environment and then processes some input ι (the response) which yields another interactive value. Since interactive programs need not terminate, the following codatatype (rather than a datatype) models such values.[3]

codatatype (α, o, ι) $\mathsf{resumption}$ =
  Pure $(\mathsf{result}\colon \alpha)$ | IO $(\mathsf{output}\colon o)$ $(\mathsf{continuation}\colon \iota \Rightarrow (\alpha, o, \iota)\ \mathsf{resumption})$

Yet, our model is slightly more involved than this. The communication partners in security protocols consume randomness to create their own cryptographic material that protects the security of the transmitted data. In the above model,

[2] RC4 and DES are known to be vulnerable to attacks; stronger encryption schemes are left as future work. Our design of the implementation supports such extensions smoothly (§6.2) as we have addressed the challenges of uniformly handling stream and block ciphers already for RC4 and DES.

[3] Harrison's reactive resumption monad differs from $\mathsf{resumption}$ in that the continuation returns a transformer in a state monad rather than an interactive value. See §6.1 for the disadvantages of the monad managing the state in Isabelle/HOL.

we could feed the randomness (an infinite bit stream) as an additional parameter to client and the client is then responsible for passing the unused tail to the continuation. However, this is unsatisfactory for two reasons. First, the explicit passing is error-prone and clutters the code of the functions. Second, it complicates developing generic reasoning techniques about the probabilities of executions. Hurd [20] has formalised a monad for pure probabilistic functions in this style, but he demands that all functions satisfy the well-formedness condition "strong independence". Instead, we explicitly model a consumer of randomness as a reader of coins (of type bool), whose structure encodes all necessary invariants. In particular, its semantics in (1) below will ensure that every bit is in fact used at most once. In §5, we introduce a monadic structure for the reader.

codatatype α coin-reader =
  Yield (yield: α) | Read (cont: bool $\Rightarrow \alpha$ coin-reader)

Finally, our model (type (α, o, ι) reactive) of probabilistic interactive values (PIV) folds the reader into the resumption. Thus, a PIV first consumes randomness and then yields either a pure result α or performs IO by outputting o and processing some input ι to return another PIV.

datatype (α, o, ι, χ) react =
  Pure (result: α) | IO (output: o) (continuation: $\iota \Rightarrow \chi$)
codatatype (α, o, ι) reactive =
  Reactive (reactive: $(\alpha, o, \iota, (\alpha, o, \iota)$ reactive) react coin-reader)

The special values Fail = Read (λ_. Fail) and RFail = Reactive Fail model computations that do not terminate.

3.2 Semantics of probabilistic interactive values

Reasoning about PIVs is based on a trace semantics. We start with the semantics of coin readers. The function consume :: α coin-reader $\Rightarrow$ bool stream $\Rightarrow$ $(\alpha \times$ bool stream) option executes a coin reader for a given bit stream (defined by (1) with Krauss' **partial-function** package [23]). It returns None iff the reader keeps reading forever the given bit stream.

$$\begin{aligned} &\text{consume (Yield } x)\ bs &&= \text{Some } (x, bs) \\ &\text{consume (Read } f)\ (b \cdot bs) &&= \text{consume } (f\ b)\ bs \end{aligned} \tag{1}$$

A trace of a PIV for a given bit stream is a possibly infinite list of events (output-input pairs) which is terminated by the result and the unconsumed bits or None if it keeps reading the bit stream.

type-synonym (o, ι) event = $o \times \iota$
codatatype (α, β) tllist = TNil β | TCons α $((\alpha, \beta)$ tllist)
type-synonym (α, o, ι) trace = $((o, \iota)$ event, $(\alpha \times$ bool stream) option) tllist

The set of all traces of a PIV for a given bit stream is defined as the greatest solution (in the complete lattice of sets with the subset ordering) of the following equation

$$
\begin{array}{l}
\mathsf{traces}\ (\mathsf{Reactive}\ r)\ bs = \\
(\mathsf{case\ consume}\ r\ bs\ \mathsf{of\ None} \Rightarrow \{\,\mathsf{TNil\ None}\,\} \\
\qquad\quad \mid \mathsf{Some}\ (\mathsf{Pure}\ x, bs') \Rightarrow \{\,\mathsf{TNil}\ (\mathsf{Some}\ (x, bs'))\,\} \\
\qquad \mid \mathsf{Some}\ (\mathsf{IO}\ out\ c, bs') \Rightarrow \displaystyle\bigcup_{in \in \mathsf{wf\text{-}responses}\ out} \mathsf{TCons}\ (out, in)\ `\ \mathsf{traces}\ (c\ in)\ bs')
\end{array}
$$

where the function $\mathsf{wf\text{-}responses} :: o \Rightarrow \iota\ \mathsf{set}$ over-approximates the possible responses by the environment to the given output, and $f\ `\ A = \{\, f\ x.\ x \in A\,\}$ denotes the image of f under A.

For illustration, consider two interactions: reading and printing a line of text. Each interaction has one representation for the output and one for the input. The output StdIn requests a line of input from the terminal and the response Receive s provides it. Conversely, StdOut s requests to print the line s, and Ack acknowledges this. Accordingly, the constants stdin and stdout perform a single interaction.

$$
\begin{array}{l}
\mathsf{stdin} = \mathsf{io\ StdIn}\ (\lambda in.\ \mathsf{case}\ in\ \mathsf{of\ Receive}\ s \Rightarrow \mathsf{Done}\ s \mid \_ \Rightarrow \mathsf{RFail}) \\
\mathsf{stdout}\ s = \mathsf{io}\ (\mathsf{StdOut}\ s)\ (\lambda in.\ \mathsf{case}\ in\ \mathsf{of\ Ack} \Rightarrow \mathsf{Done}\ () \mid \_ \Rightarrow \mathsf{RFail})
\end{array}
$$

Here, Done x stands for Reactive (Yield (Pure x)) and io out c abbreviates Reactive (Yield (IO out c)). The function wf-responses ensures that the semantics considers only traces in which inputs correspond to outputs, i.e., Receive answers StdIn and Ack answers StdOut.

$$
\mathsf{wf\text{-}responses\ StdIn} = \mathsf{range\ Receive} \qquad\qquad \mathsf{wf\text{-}responses}\ (\mathsf{StdOut}\ \_) = \{\,\mathsf{Ack}\,\}
$$

3.3 Compilation of probabilistic interactive values

In the generated code, stdin and stdout are implemented by their defining equation, i.e., as values of the (co)datatypes that we have defined so far. In particular, they do not call the target language API for reading from and writing to the terminal. One could instruct the code generator to bind them to this API. However, this will not work in our setting, because we have to make sure that the bit stream is correctly passed on. Instead, we implement an interpreter for PIVs and prove that its execution corresponds to one of the traces in traces, but we do not specify which. To that end, we define a type α IO as a partial transformer of the environment (of the unspecified type real-world) with the standard monad operations return and $\gg\!=$. During code generation, the IO type constructor is mapped to Haskell's `IO` and dropped in the other languages.

typedecl real-world
typedef α IO = UNIV :: (real-world $\rightharpoonup$ $\alpha \times$ real-world) set

The interpreter $\mathsf{interp\text{-}reactive} :: (\alpha, o, \iota)\ \mathsf{reactive} \Rightarrow \mathsf{bool\ stream} \Rightarrow \alpha\ \mathsf{IO}$ depends on an interpretation function $\mathsf{interp} :: o \Rightarrow \iota\ \mathsf{IO}$ that produces for a given output o the response ι of the environment.

$$\begin{array}{l}
\textsf{interp-reactive } (\textsf{Reactive } r)\ bs = \\
(\textsf{case consume } r\ bs \textsf{ of} \\
\quad\ \ \textsf{Some } (\textsf{Pure } x, bs') \Rightarrow \textsf{return } (x, bs') \\
\ \ \mid \textsf{Some } (\textsf{IO } out\ c, bs') \Rightarrow \textsf{interp } out \gg\!= (\lambda in.\ \textsf{interp-reactive } (c\ in)\ bs'))
\end{array}$$

We show that this interpreter is sound with respect to the semantics traces. Formally, the interpreted run of a PIV corresponds to one of its traces provided that interp in fact satisfies the specification wf-responses and returns total transformers.

In the example above, we declare two unspecified functions stdin-impl :: String.literal IO and stdout-impl :: String.literal ⇒ unit IO. We tell the code generator to bind them to the target language functions for reading and writing a line on the terminal. The interpretation function interp dispatches StdIn and StdOut to stdin-impl and stdout-impl, resp.

$$\begin{array}{lll}
\textsf{interp StdIn} & = \textsf{stdin-impl} & \gg\!= \textsf{return} \circ \textsf{Receive} \\
\textsf{interp } (\textsf{StdOut } s) & = \textsf{stdout-impl } s \gg & \textsf{return Ack}
\end{array}$$

The generated code contains the PIVs as values of the codatatype reactive and the interpreter interp-reactive to run them (it also takes a parameter for the bit stream). This separation has two disadvantages. First, the code is harder to read, because the PIVs do not use the sequencing notation of the target language (e.g. `;` in SML, `>>=` in Haskell)—only the interpreter does. Second, interpretation has some run-time overhead, as the datatype values have to be constructed and destructed. Fortunately, the interpreter only acts when evaluation consumes randomness or interacts with the environment; data processing is not interpreted, as ordinary HOL functions do this. Thus, we expect little impact on performance.

In return, modeling and reasoning benefits from the separation in two ways. First, the gap between the interpreted function and the generated code is small (the approach is similar to the mapping by Bulwahn et al. [8]). This is important because the correctness of the mapping to the target language is unverified and thus trusted. Second, we can explicitly feed the source of randomness to the semantics and therefore reason about probabilities inside the logic.

4 Foreign function interface

In this section, we demonstrate that the adaptation facility of the code generator provides a minimalistic foreign function interface (FFI) for Isabelle. Our TLS implementation uses the FFI for two purposes. First, to call functions in cryptographic libraries. Second, to map the actual interaction operations to their target language operations. This way, PIVs in our HOL model interact with the real world during execution. In detail, we have imported line-based terminal IO (stdin-impl and stdout-impl) and a network API based on TCP sockets.

Adaptation maps constants specified in HOL to target language constants such that the code becomes more readable and more efficient [15]. By default, Isabelle/HOL maps the types bool, unit, $\alpha \times \beta$, α option, and α list to their counterparts in the target languages, and a library theory does so for char. Moreover, there are the special types integer and String.literal that reflect arbitrary-precision integers and strings from the target language. They are used to interface with target-language operations, especially for arithmetic and error printing. All these types and their operations are modelled in HOL.

In contrast, a FFI provides access to libraries without a detailed model—the type signatures suffice. This is crucial for application development, because reimplementing the library in HOL would defeat the point of using a library. Unfortunately, this sometimes cannot be avoided (see §5.2). Moreover, the FFI supports data exchange between Isabelle/HOL programs and the libraries.

Our approach to FFI in Isabelle works as follows. To import a type and a function in Isabelle/HOL, one declares them with **typedecl** and **consts**, resp., and thus leave them unspecified. Using adaptation, one instructs the code generator to serialise them to the imported function. Declaring and not specifying the types and functions has the advantage that the adaptation is automatically correct whenever the imported functions obey referential transparency,[4] because we can only prove trivial statements like non-emptiness and reflexivity about unspecified types and constants.

This works well in practice, but the details can be tricky. In particular, two issues are worth pointing out, which we illustrate with examples below. First, exchanging data between Isabelle/HOL and the imported functions can only be done via types of the target language, because the code generator does not offer a reliable way to access generated types and functions from adaptations. Second, the code generator supports four target languages each of which offers a different API. As the code equations are the same for all target language, one has to find a common base that can be mapped to all the APIs with as little glue code as possible.

In the case study, we used the FFI heavily to access the cryptographic libraries of the target language. We did not want to reimplement these algorithms to avoid that security problems slip in. Currently, we only use Haskell as target language, because Haskell's crypto platform[5] provides a nice functional API with type classes. However, we plan to design an Isabelle crypto library with serialisations to the available libraries in all target languages.

For example, consider the FFI import of the SHA1 hash implementation `Crypto.Hash.SHA1.hash`. In Haskell, this function digests a `ByteString` rather than a string. Hence, we import both `ByteString` and the hash function with the following declarations.[6]

---

[4] Referential transparency is crucial. For example, we can prove serial () = serial () by reflexivity. However, if we serialise serial to a function that returns the number of times it has been called (like the `serial` function in Isabelle/ML), the generated code for the equality test statement evaluates to False. That is, the generated code does not adhere to the specification.

[5] `https://github.com/vincenthz/hs-crypto-platform`

[6] The declarations show an inconvenience of adaptation. To import a library function, one must write a boilerplate Haskell module (`Isa_Crypto_Hash` in the lower part), which contains the import statement and exports the function of interest.

typedecl byte-string
consts hash-sha1 :: byte-string ⇒ byte-string
code-printing constant hash-sha1 ⇀ (Haskell) `Isa_Crypto_Hash.hash_sha1`

code-printing code-module `Isa_Crypto_Hash` ⇀ (Haskell) `{*`
```
  import qualified Crypto.Hash.SHA1;
  hash_sha1 = Crypto.Hash.SHA1.hash;
*}
```
code-reserved Haskell `Isa_Crypto_Hash`

However, hash-sha1 cannot yet be used, as there is no way to get hold of a byte-string (in the executable fragment of HOL). So, we additionally import Haskell's conversion functions pack and unpack between lists of unsigned bytes and `ByteString` with similar declarations. Fortunately, the first author has previously modelled unsigned bytes in Isabelle/HOL [28]. Thus, we can end the chain of importing external types and their conversion functions here.

Next, we illustrate the intricacies of supporting four target languages simultaneously using the type uint8 of unsigned bytes from [28]. The HOL-Word library [11] already formalises the type 8 word of unsigned bytes as an instance of the general type α word. However, adaptation only works for type constructors, not type expressions like 8 word. That is why the type (constructor) uint8 copies 8 word and its operations. Unfortunately, unsigned bytes are available only in the SML Basis library (`Word8.word`) and Haskell (`Word8`); Scala only provides signed bytes (`Byte`), and we have not found anything in the OCaml standard library. Sign-sensitive operations like division therefore need different implementations for different target languages, but recall that all languages use the same code equations. We circumvent this restriction by specifying adaptations for different constants as shown in Fig. 1. Division div on uint8 is implemented in terms of uint8-div, which is unspecified if the divisor is 0; this accounts for the fact that division by 0 is not specified uniformly across the target languages. The code equation for uint8-div implements unsigned division in terms of signed division uint8-sdiv and shift operations using an algorithm from [39]. Finally, signed division on uint8 uses signed division sdiv on 8 word, which itself is implemented via unbounded integers following [11]. For SML and Haskell, uint8-div is serialised directly to the provided operations, i.e., the equation for uint8-div is ignored. For Scala, adaptation kicks in only for uint8-sdiv, i.e., uint8-div is implemented according to the formally proven code equation. As there are no adaptations for OCaml, unsigned division follows the chain of code equations down to unbounded integers (starting with those in Fig. 1). This is not particularly efficient, but one could write a library of unsigned bytes for OCaml directly and import it analogously.

5 Monadic Programming

Most of our TLS implementation is written in monadic style because sequencing is ubiquitous, e.g., checking and propagating errors, parsing input, consuming

$$
\begin{aligned}
x \;\mathsf{div}\; y \quad &= (\mathsf{if}\ y = 0\ \mathsf{then}\ 0\ \mathsf{else}\ \mathsf{uint8\text{-}div}\ x\ y) \\
\mathsf{uint8\text{-}div}\ x\ y \quad &= (\mathsf{if}\ 128 \le y\ \mathsf{then}\ \mathsf{if}\ x < y\ \mathsf{then}\ 0\ \mathsf{else}\ 1 \\
&\quad\ \mathsf{else}\ \mathsf{if}\ y = 0\ \mathsf{then}\ \mathsf{uint8\text{-}div0}\ x \\
&\quad\ \mathsf{else}\ \mathsf{let}\ q = (\mathsf{uint8\text{-}sdiv}\ (x >> 1)\ y) << 1 \\
&\qquad\quad \mathsf{in}\ \mathsf{if}\ x - q * y \ge y\ \mathsf{then}\ q + 1\ \mathsf{else}\ q) \\
\mathsf{uint8\text{-}sdiv}\ x\ y &= (\mathsf{if}\ y = 0\ \mathsf{then}\ \mathsf{uint8\text{-}div0}\ x\ \mathsf{else}\ x\ \mathsf{sdiv}\ y)
\end{aligned}
$$

code-printing constant uint8-div ⇀ (SML) `Word8.div ((_), (_))`
and (Haskell) `Prelude.div`
code-printing constant uint8-sdiv ⇀ (Scala) `(_ / _).toByte`
code-abort uint8-div0

Fig. 1. Code equations and adaptations for division on unsigned bytes uint8

randomness, and exchanging messages. In this section, we present the monads for PIVs (§5.1) and parsing (§5.2) and discuss how Isabelle's restrictions influenced our design decisions. In particular, unspecified imports via the FFI (as discussed in the previous section) are not always feasible (§5.2).

5.1 A monad for interactive programs

A monad consists of a unit operation and a sequencing operation $\gg=$, which obey the three monad laws: the unit is neutral for $\gg=$ on both sides and $\gg=$ is associative. We define monad operations on coin readers (type _ coin-reader), resumptions (type $(\_, o, \iota)$ resumption for fixed o and ι) and PIVs (type $(\_, o, \iota)$ reactive) and prove the monad laws. The units are given by Yield, Pure, and Done = Reactive ∘ Yield ∘ Pure, resp. The sequencing operations are defined by primitive corecursion as follows.

$$
\begin{aligned}
&\mathsf{Yield}\ x \gg= g = g\ x &&\mathsf{Read}\ f \gg= g = \mathsf{Read}\ (\lambda b.\ f\ b \gg= g) \\
&\mathsf{Pure}\ x \gg= g = g\ x &&\mathsf{IO}\ o\ c \gg= g = \mathsf{IO}\ o\ (\lambda i.\ c\ i \gg= g)
\end{aligned}
$$

$$
\begin{aligned}
&\mathsf{Reactive}\ r \gg= g = \\
&\quad \mathsf{Reactive}\ (r \gg= (\lambda r'.\ \mathsf{case}\ r'\ \mathsf{of}\ \mathsf{Pure}\ x \Rightarrow \mathsf{reactive}\ (g\ x) \\
&\qquad\qquad\qquad\qquad\qquad \mid \mathsf{IO}\ o\ c \Rightarrow \mathsf{Yield}\ (\mathsf{IO}\ o\ (\lambda i.\ c\ i \gg= g))))
\end{aligned}
$$

Additionally, we equip both types with a chain-complete partial order (ccpo) with least elements Fail and RFail, resp. This makes it possible to define monadic PIVs with Krauss' **partial-function** package [23].

Unfortunately, the construction above is not as elegant as it could be. In Isabelle's type system, only individual monads can be expressed in HOL; monads in general would require type operators. Therefore, we have to settle with the syntactic illusion of monads and do notation by Krauss and Sternagel [24]. Moreover, monad transformers [27] cannot be expressed either. However, the type reactive merely combines the coin reader monad with the resumption monad. With monad transformers, the manual construction of reactive and the proofs of the monad laws and possibly of ccpo structure would not be needed, it would follow by composing coin-reader and resumption. The proofs about reactive reflect this redundancy, too. Following the recursion through a codatatype (coin-reader) and a datatype (react), they nest a case analysis inside a coinduction inside another coinduction. We have not been able to parametrise the sequencing operation on coin-reader such that the coin-reader lemmas can be reused in the proofs about reactive. Instead, we essentially reprove them in the coinductions about reactive, which leads to large and complicated proofs that are hard to automate.

5.2 Parsing

Parsing and validating the received messages constitutes a considerable part of our TLS implementation. At first, we thought about importing an efficient, well-tested parsing library like Parsec [26] from Haskell using the FFI interface. However, this turned out to be impractical, because one cannot prove anything useful about unspecified functions. Recall that we are interested only in implementing TLS, not in proving. Nevertheless, the packages for defining recursive functions rely on definitional principles which an unspecified type of parsers cannot provide (**function** requires a termination proof [22], **partial-function** a ccpo and monotonicity [23], and **primcorec** a corecursor [6]).

Therefore, we reimplemented the Parsec library [26] in Isabelle/HOL.[7] This made it possible to prove that the monadic $\gg\!=$ operation on parsers satisfies the following congruence rule (where parsec-range q returns all possible results of a successful run of the parser q).

$$\frac{p = q \qquad \bigwedge x.\ x \in \mathsf{parsec\text{-}range}\ q \Longrightarrow f\ x = g\ x}{p \gg\!= f = q \gg\!= g} \tag{2}$$

The **function** package uses this rule to enable reasoning about the results of sub-parsers in termination proofs of recursive parsers. For example, consider the parser certsp for a list of X509 certificates in Fig. 2. It takes the expected length n (an unsigned 32-bit integer) of the (remaining) string of certificates[8] and returns the list of certificates it parses. As it is defined with the **function** package, recursion must be shown to terminate. Fortunately, the number of remaining bytes n decreases in the recursive call.[9] However, this only holds after the checks of the assertion parser assertp, because subtraction might otherwise

[7] We briefly tried to delegate this task to Haskabelle [14], but failed due to bugs in Haskabelle and strictness annotations being unsupported. We have not yet adapted the code generator for the parser. So, it does not serialise the parser functions to the Haskell library, because soundness would be hard to achieve. Consequently, the generated parser is probably less efficient than the original, because boxing and strictness annotations are lost in the round trip.

[8] As usual in network protocols, TLS message fields of variable length start with the length of the fields. Thus, we know in advance how many bytes the parser is supposed to process.

[9] In principle, recursion must terminate anyway because the input to the parser is a finite list and some input is consumed before the recursive call. However, the

```
certsp $n$ = (if $n = 0$ then parsec-return [] else do {
    assertp ($n \geq 3$) ("certificate list is too short: " @ show $n$);
    $len$ ← uint24p;
    assertp ($len > 0$) "certificate length is 0";
    assertp ($len \leq n - 3$) "certificate exceeds certificate string";
    $cert$ ← parse-bytes $len$;
    $cs$ ← certsp ($n - 3 - len$);
    parsec-return (X509v3 $cert \cdot cs$)
  })
```

Fig. 2. Monadic parser for a list of X509 certificates

underflow. It is the above congruence rule that makes these checks available in the termination proof, because assertp *b msg* succeeds only if *b* holds.

In principle, we could code the assertions as normal ifs and do without (2). However, the code quickly becomes hard to read, as if statements break the do block apart. Hence, the FFI import of Parsec is not a viable option.

6 State passing

The TLS protocol is inherently stateful, as it must maintain the connection state across several calls to its API. Since the implementation already lives in a monad, it might be tempting to delegate the state management to the monad. However, this causes several problems (§6.1). Therefore, we decided to treat the state as an ordinary value that is passed from one function to the other. Between two TLS calls, the application must make sure that the next call receives the TLS state that the previous call has returned. That is, we impose the burden of state passing on the programmer.

In some cases, one can avoid these problems by storing the operations on the state rather than the state itself. The implementation uses this for the state of the cipher suites (§6.2).

6.1 Problems with hiding the states in the monad

Initially, we tried to let the monad take care of the state, like Harrison does in [19]. Unfortunately, we ran into several problems. In this section, we discuss three of them.

First, HOL's type system severely restricts how the state can be used. Basically, there are two options. Either, the state type shows up in the monad's type

input is not a direct argument of the function, it is hidden in the parser type. Consequently, the **function** package cannot exploit this in the termination proof. Danielsson [10] designed a library of total parser combinators in Agda, which might solve this problem. However, it is not yet clear how to express his constructions without dependent types and to obtain efficient parsers with good error reporting capabilities.

as an additional type parameter σ, which applications instantiate as needed. This makes it difficult to combine applications with different state types, as Bulwahn et al. have already noted [8] in the context of Imperative-HOL. Or, the state is represented as a universal domain in which all storable types can be encoded. In [8], the domain nat covers all first-order values of countable types. This excludes, in particular, all string-processing functions and many codatatypes like bit streams. Even if the domain can be enlarged to cover functions up to some fixed order, state transformers never can be stored for cardinality reasons.

Second, the semantics and implementation of PIVs get more complicated. Interaction with the world can always result in unexpected responses, so it is reasonable to not specify functions like stdin-impl and stdout-impl. In contrast, data stored in the state should provably remain the same while, say, a message is being sent over a socket. Thus, the state must not be stored in real-world. Yet, it would be desirable to update it destructively, i.e., store it in the IO monad. Therefore, we would have to thread another state through the interaction model. Moreover, if the state is polymorphic, its type variable has to show up in the IO type, but they must not in the generated code (Haskell's `IO` type, e.g., takes only one type argument for the result of the computation, and none for its actions). Yet, HOL does not allow to hide type variables.

Third, functions for data processing are given the complete state and can thus manipulate it at their will. Modularity mechanisms like information hiding do not work in HOL, because HOL has no notion of computation or computability. For example, description operators can be used to scrutinize and change values even of polymorphic type. Clearly, this is no issue for code generation, as such functions are outside of the executable fragment. Reasoning, however, has to deal with this. For example, suppose that the state stores the bit string of randomness (this would allow to get rid of the coin reader monad). Then, functions can peek at randomness without consuming it. Hence, to reason about probabilities, we would need additional well-formedness conditions on the processing functions similar to Hurd [20].

6.2 Storing the cipher suite

The TLS connection state stores the security parameters, among others. They specify the encryption algorithm and the cipher state (whose format depends on the chosen algorithms). Recall that the CCS protocol dynamically changes the encryption algorithm; and the cipher state is updated after every message transmission. Here, we discuss three options for modelling the symmetric cipher and its state; the other cryptographic algorithms are done similarly.

In the simplest approach, the different states of the encryption suites are distinguished with different constructors of a datatype; the cryptographic functions dispatch to the corresponding algorithm by pattern-matching. The following sketch illustrates the idea (where rc4 and des model the state of the cipher and des-iv the DES initialisation vector).

datatype cipher = RC4 rc4 | DES des des-iv | ...

encrypt :: cipher $\Rightarrow$ string $\Rightarrow$ (cipher $\times$ string)
encrypt (RC4 s) m = apfst RC4 (encrypt-rc4 s m)
encrypt (DES s iv) m = apfst (uncurry DES) (encrypt-des s iv m)

This approach hard-codes the available ciphers in the datatype. When adding a new cipher, one therefore has to change the datatype definition and adapt all pattern-matching functions. Due to this restriction, we do not consider this a good solution.

Haskell's crypto library achieves this kind of extensibility by introducing type classes for block and stream ciphers with the cryptographic operations as parameters. Initially, we tried to mimic this in Isabelle/HOL and import it with the FFI. However, existential type quantification would be needed to make this work, which is not sound for HOL. The following explains the problem. Suppose there are two classes with instances for the cipher states of RC4 and DES.

class block-cipher = ...
class stream-cipher = ...

typedecl des **instance** des :: block-cipher ..
typedecl rc4 **instance** rc4 :: stream-cipher ..

Clearly, to add a new cipher, it suffices to declare an instance of the appropriate class. The difficulty lies in combining the type classes in one type, e.g., as follows (where β iv is the type of initialisation vectors for the block cipher β):

datatype cipher = Stream (α :: stream-cipher) | Block (β :: block-cipher) (β iv)

This declaration is not valid, because the datatype cipher must mention all type variables on which it depends. If one does so, the cipher suite shows up statically in the type. Hence, the CCS protocol cannot change it dynamically any more. We would need existential types to make the representation abstract [31].[10]

The third approach inlines the type class as a dictionary and eliminates the existential quantification by representing the existentially quantified type by its possible observations (as discussed on the OCaml mailing list in post 7fb6359bea374fe7#1d9c3931ec3e72ab). Now, state and operations changes roles. There is only a small fixed set of operations which TLS needs, namely the operations of the type classes stream-cipher and block-cipher. Hence, instead of modeling the state, the cipher state type models the behaviour of the cipher suites as a codatatype. The codatatype makes it possible to apply the cipher any number of times, because each application returns an updated cipher.[11]

[10] The type variables α and β represent the state of the encryption algorithms, which cipher is supposed to hide. They are not just type tokens to select the algorithm—otherwise, we could encode the type with typerep. As cipher states are typically first-order values, a universal domain similar to [8] could work in this case.

[11] TLS processes outgoing and incoming messages with distinct ciphers, because the CCS changes ciphers asynchronously. Thus, each cipher is used only for encryption or only for decryption. Therefore, cipher has just one operation apply, which either encrypts or decrypts.

```
codatatype cipher = Cipher (apply: string ⇒ cipher × string)
primcorec mk-rc4-enc :: rc4 ⇒ cipher where
  apply (mk-rc4-enc s) = map-pair mk-rc4-enc id ∘ encrypt-rc4 s
definition init-rc4-enc :: string ⇒ cipher where
  init-rc4-enc k = mk-rc4-enc (mk-key-rc4 k)
```

With this design, adding a new cipher suite becomes easy; one just defines another wrapping function function like mk-rc4-enc. Conversely, new operations on cipher suites are hard to add, as the constructor cipher would need another field for the operation. (This is easy with the state model above.) However, TLS requires only a fixed set of cryptographic operations, so new operations are probably not necessary.

7 Comparison with functional protocol implementations

TLS has previously been implemented in F# [4] and complete network stacks in SML [5] and OCaml [30]. They all use imperative features to store the mutable state. In contrast, we have implemented TLS in a pure language, in which state passing becomes explicit (see §6).

The network protocol stack implementations [5,30] demonstrate that functional languages are adequate for such applications. Our case study shares this goal, but does not go as far. It is much smaller, and efficiency played a minor role in its design. To reach a comparable level, a lot more work and extensions of Isabelle would be needed.

The TLS implementation in F# [4] has been proven secure with the F7 type checker. Security is expressed with assertions and dependent types, and the verification relies on information hiding through interfaces and parametricity of functions. We have not yet analysed the security of our implementation, but the F7 approach seems unfeasible in HOL, as parametricity does not come for free and hiding mechanisms are limited (see §6.1). Conversely, it is (in theory) possible to prove the whole implementation correct in Isabelle/HOL, whereas F7 analyses only the stubs of the cryptographic algorithms.

8 Insights and Future Directions

Our case study shows that it is feasible to implement a stateful, interactive application in Isabelle/HOL, although Isabelle's restrictive type system and its nature as a theorem prover pose additional challenges over other (impure) functional languages. Still, our TLS implementation is just a prototype. Strong cryptographic algorithms are missing and certificates are not checked, so it cannot yet provide the desired security guarantees. Also, some implementation bugs prevent interoperability of our client and server with third-party TLS servers and clients. Moreover, efficiency was of little concern during the implementation. There are a lot of conversions, especially between string, String.literal and byte-string. A larger set of operations on the latter (modelled in Isabelle/HOL) might make many obsolete. Also, the server accepts only one connection at a time, as it is single-threaded.

Below, we summarise our experience during the development and comment on future directions for improvement.

Development support We implemented the prototype in the Isabelle/jedit editor, which provides many conveniences of a modern IDE for programming. Navigation with Ctrl-Click was particularly useful.

We found it crucial to regularly test the implementation; as we do not prove theorems about our functions, this is the only way to find typos and bugs. However, Isabelle does not support test automation natively. The **value** [code] command can compile and execute test cases, but it does not check whether the output matches the expectation. Moreover, **value** breaks down with FFI imports from non-ML libraries, because it evaluates the functions in Isabelle's run-time in the special code target Eval, which does not know about the imports. To automate the regression tests, we wrote test scripts that generate Haskell code, compile and run it, and check the output. We plan to develop this further into a test harness for the code generator, because it seems useful also for testing adaptations, e.g. for types like native machine words [28].

Foreign functions Adaptation in the code generator provides a minimalistic foreign function interface. Exchanging data with imported functions, however, is complicated, as the Isabelle/HOL library formalises only few types for that. First steps in this directions have been done (machine words [28], IEEE floating point numbers [40], and arrays to some extent [25]). Yet, it is still hard to exchange structured data or strings. More work in this area is needed.

Interactive model Our model of interaction is still under development. The IO type as the set of partial transformers of the environment appears as a sound implementation model for PIVs which interact only by exchanging messages. However, it is not adequate for modelling shared-memory concurrency as would be needed for a multi-threaded TLS server. The semantics of Haskell's IO monad [35] looks like an interesting alternative. Moreover, we plan to develop a reasoning infrastructure such that properties of PIVs can be stated and proved formally. In particular, the present semantics fails to capture the interaction part, as it executes every PIV in isolation. The challenge here is that the environment determines the ways of interaction and its formalisation therefore has to be tailored to the application.

Acknowledgements. Johannes Hölzl suggested to model probabilistic function with the coin reader monad and Joachim Breitner suggested improvements to the presentation. This work was supported by the SNF.

References

1. Berghofer, S., Bulwahn, L., Haftmann, F.: Turning inductive into equational specifications. In Berghofer, S., Nipkow, T., Urban, C., Wenzel, M. (eds.) Theorem Proving in Higher Order Logics (TPHOLs 2009). LNCS, vol. 5674, pp. 131–146. Springer (2009)
2. Berghofer, S., Nipkow, T.: Executing higher order logic. In Callaghan, P., Luo, Z., McKinna, J., Pollack, R., Pollack, R. (eds.) Types for Proofs and Programs (TYPES 2000). LNCS, vol. 2277, pp. 24–40. Springer (2002)
3. Berghofer, S., Wenzel, M.: Inductive datatypes in HOL – lessons learned in formal-logic engineering. In Bertot, Y., Dowek, G., Théry, L., Hirschowitz, A., Paulin, C. (eds.) Theorem Proving in Higher Order Logics (TPHOLs 1999). LNCS, vol. 1690, pp. 19–36. Springer (1999)
4. Bhargavan, K., Fournet, C., Kohlweiss, M., Pironti, A., Strub, P.Y.: Implementing TLS with verified cryptographic security. In: Security and Privacy (S&P 2013), pp. 445–459 (2013)
5. Biagioni, E., Harper, R., Lee, P.: A network protocol stack in Standard ML. Higher Order and Symbolic Computation 14(4), 309–356 (2001)
6. Blanchette, J.C., Hölzl, J., Lochbihler, A., Panny, L., Popescu, A., Traytel, D.: Truly modular (co)datatypes for Isabelle/HOL. In Klein, G. Gamboa, R. (eds.) Interactive Theorem Proving (ITP 2014). LNCS. Springer (2014)
7. Bulwahn, L.: The new quickcheck for Isabelle – random, exhaustive and symbolic testing under one roof. In Hawblitzel, C. Miller, D. (eds.) Certified Programs and Proofs (CPP 2012). LNCS, vol. 7679. Springer 92–108 (2012)
8. Bulwahn, L., Krauss, A., Haftmann, F., Erkök, L., Matthews, J.: Imperative functional programming in Isabelle/HOL. In Mohamed, O. A., Muñoz, C., Tahar, S. (eds.) Theorem Proving in Higher Order Logics (TPHOLs 2008). LNCS, vol. 5170, pp. 134–149. Springer (2008)
9. Chaieb, A.: Integration of local and reflective proof-procedures. In Dixon, L. Johansson, M. (eds.) Isabelle Workshop (Isabelle 2007), pp. 68 (2007)
10. Danielsson, N.A.: Total parser combinators. In: Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming (ICFP 2010), pp. 285–296. ACM (2010)
11. Dawson, J.: Isabelle theories for machine words. In Goldsmith, M. Roscoe, B. (eds.) Automated Verification of Critical Systems (AVOCS 2007). ENTCS, vol. 250(1), pp. 55–70. Elsevier (2009)
12. Dierks, T., Allen, C.: The TLS protocol, version 1.0. RFC 2246 (1999)
13. Esparza, J., Lammich, P., Neumann, R., Nipkow, T., Schimpf, A., Smaus, J.G.: A fully verified executable LTL model checker. In Sharygina, N. Veith, H. (eds.) Computer Aided Verification (CAV 2013). LNCS, vol. 8044, pp. 463–478. Springer (2013)
14. Haftmann, F.: From higher-order logic to Haskell: There and back again. In: Proceedings of the 2010 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation (PEPM 2010), pp. 155–158. ACM (2010)
15. Haftmann, F., Bulwahn, L.: Code generation from Isabelle/HOL theories. `http://isabelle.in.tum.de/dist/Isabelle2013-2/doc/codegen.pdf` (2013)
16. Haftmann, F., Krauss, A., Kunčar, O., Nipkow, T.: Data refinement in Isabelle/HOL. In Blazy, S., Paulin-Mohring, C., Pichardie, D. (eds.) Interactive Theorem Proving (ITP 2013). LNCS, vol. 7998, pp. 100–115. Springer (2013)

17. Haftmann, F., Nipkow, T.: Code generation via higher-order rewrite systems. In Blume, M., Kobayashi, N., Vidal, G. (eds.) Functional and Logic Programming (FLOPS 2010). LNCS, vol. 6009, pp. 103–117. Springer (2010)
18. Haftmann, F., Wenzel, M.: Constructive type classes in Isabelle. In Altenkirch, T. McBride, C. (eds.) Types for Proofs and Programs (TYPES 2006). LNCS, vol. 4502, pp. 160–174. Springer (2007)
19. Harrison, W.L.: The essence of multitasking. In Johnson, M. Vene, V. (eds.) Algebraic Methodology and Software Technology (AMAST 2006). LNCS, vol. 4019, pp. 158–172. Springer (2006)
20. Hurd, J.: A formal approach to probabilistic termination. In Carreño, V. A., Muñoz, C. A., Tahar, S. (eds.) Theorem Proving in Higher Order Logics (TPHOLs 2002). LNCS, vol. 2410, pp. 230–245. Springer (2002)
21. Kanav, S., Lammich, P., Popescu, A.: A conference management system with verified document confidentiality. In: Computer Aided Verification (CAV 2014). LNCS. Springer (2014)
22. Krauss, A.: Partial and nested recursive function definitions in higher-order logic. J. Autom. Reasoning 44(4), 303–336 (2010)
23. Krauss, A.: Recursive definitions of monadic functions. In Bove, A., Komendantskaya, E., Niqui, M. (eds.) Workshop on Partiality and Recursion in Interactive Theorem Proving (PAR 2010). EPTCS, vol. 43, pp. 1–13 (2010)
24. Krauss, A., Sternagel, C.: Monad syntax. Isabelle Developer Workshop (IDW 2010), (2010)
25. Lammich, P.: Collections framework. Archive of Formal Proofs (2009) `http://afp.sf.net/entries/Collections.shtml`, Formal proof development.
26. Leijen, D., Meijer, E.: Parsec: Direct style monadic parser combinators for the real world. Technical Report UU-CS-2001-27, Department of Computer Science, Universiteit Utrecht (2001)
27. Liang, S., Hudak, P., Jones, M.: Monad transformers and modular interpreters. In: Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 1995), pp. 333–343. ACM (1995)
28. Lochbihler, A.: Native word. Archive of Formal Proofs (2013) `http://afp.sf.net/entries/Native_Word.shtml`, Formal proof development.
29. Lochbihler, A., Bulwahn, L.: Animating the formalised semantics of a Java-like language. In van Eekelen, M., Geuvers, H., Schmalz, J., Wiedijk, F. (eds.) Interactive Theorem Proving (ITP 2011). LNCS, vol. 6898, pp. 216–232. Springer (2011)
30. Madhavapeddy, A., Ho, A., Deegan, T., Scott, D., Sohan, R.: Melange: Creating a "functional" internet. In: Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007 (EuroSys 2007), pp. 101–114. ACM (2007)
31. Mitchell, J.C., Plotkin, G.D.: Abstract types have existential type. ACM Trans. Program. Lang. Syst. 10(3), 470–502 (1988)
32. Nipkow, T.: Verified efficient enumeration of plane graphs modulo isomorphism. In van Eekelen, M., Geuvers, H., Schmalz, J., Wiedijk, F. (eds.) Interactive Theorem Proving (ITP 2011). LNCS, vol. 6898, pp. 281–296. Springer (2011)
33. Nipkow, T.: Teaching semantics with a proof assistant: No more LSD trip proofs. In Kuncak, V. Rybalchenko, A. (eds.) Verification, Model Checking, and Abstract Interpretation (VMCAI 2012). LNCS, vol. 7148, pp. 24–38. Springer (2012)
34. Nipkow, T., Klein, G.: Conrete Semantics – A Proof Assistant Approach. `http://www.in.tum.de/~nipkow/Concrete-Semantics/` (2014)

35. Peyton Jones, S.: Tackling the awkward squad: monadic input/output, concurrency, exceptions, and foreign-language calls in Haskell. In Hoare, T., Broy, M., Steinbruggen, R. (eds.) Engineering theories of software construction. IOS Press 47–96 (2001)
36. Thiemann, R., Sternagel, C.: Certification of termination proofs using CeTA. In Berghofer, S., Nipkow, T., Urban, C., Wenzel, M. (eds.) Theorem Proving in Higher Order Logics (TPHOLs 2009). LNCS, vol. 5674, pp. 452–468. Springer (2009)
37. Traytel, D., Nipkow, T.: A verified decision procedure for MSO on words based on derivatives of regular expressions. In: Proc. ACM SIGPLAN International Conference on Functional Programming (ICFP 2013), pp. 3–12 (2013)
38. Traytel, D., Popescu, A., Blanchette, J.C.: Foundational, compositional (co)datatypes for higher-order logic: Category theory applied to theorem proving. In: Proceedings of the 2012 27th Annual IEEE/ACM Symposium on Logic in Computer Science (LICS 2012), pp. 596–605. IEEE Computer Society (2012)
39. Warren, H.S.: Hacker's Delight. 2nd edn. Addison-Wesley (2012)
40. Yu, L.: A formal model of IEEE floating point arithmetic. Archive of Formal Proofs (2013) `http://afp.sf.net/entries/IEEE_Floating_Point.shtml`, Formal proof development.
41. Züst, M.: A functional protocol implementation. Bachelor's thesis, Institute of Information Security, ETH Zurich (2014)