Draft presented at Workshop on Foundations of Computer Security 2017

NARCISSUS: Deriving Correct-By-Construction Decoders and Encoders from Binary Formats

Sorawit Suriyakarn
Clément Pit-Claudel
MIT CSAIL
{sorawit,cpitcla}@csail.mit.edu

Benjamin Delaware
Purdue University
bendy@purdue.edu

Adam Chlipala
MIT CSAIL
adamc@csail.mit.edu

Abstract

Every injective function has an inverse, although constructing the inverse for a particular injective function can be quite tricky. One common instance of inverse-function pairs is the binary encoders and decoders used to convert in-memory data into and out of a structured binary format for network communication. Ensuring that a given decoder is a proper inverse of the original encoder is particularly important, as any error has the potential to introduce security vulnerabilities or to corrupt or lose data in translation.

In this paper, we present a synthesis framework, NARCISSUS, that eliminates both the tedium and the potential for error in building the inverse of a binary encoder. The starting point of the process is a binary format, expressed as a functional program in the nondeterminism monad, that precisely captures all the valid binary encodings of an arbitrary datatype instance. From this specification, NARCISSUS synthesizes a decoder that is guaranteed to be the inverse of this relation, drawing on an extensible set of decoding strategies to construct the implementation. Each decoder is furthermore guaranteed to detect malformed encodings by failing on inputs not included in this relation. The derivation is carried out inside the Coq proof assistant and produces a proof trail certifying the correctness of the synthesized decoder. We demonstrate the utility of our framework by deriving and evaluating the performance of decoders for all packet formats used in a standard network stack.

1. Introduction

The code responsible for encoding and parsing external data is a vital component of any software that communicates with the outside world: failure to produce or correctly interpret encoded data that conform to a standard format can result in lost or corrupted data, while bugs in the decoder code that takes untrusted external data as input represents a key attack surface for malicious actors. Writing code that conforms to a high-level format description by hand is both tedious and error-prone. As one example, a bug in a seldom-used piece of the parser for IP headers used by the network stack of MirageOS [12] was only discovered and patched three months ago. Code-generation frameworks that synthesize encoders and decoders from high-level data-description languages [4, 7–9, 13] are a particularly appealing solution to this problem, offering both convenience and (potentially) higher-assurance code. While this approach removes the threat of programmer-introduced errors in the communication code, it forces the user to trust that the code-generation framework is correct. Frameworks are also constrained by their data-description language, potentially limiting their application to standardized formats. To address this latter problem, frameworks like Nail [4] allow for user-defined extensions to the code generator.

Unfortunately, extensibility comes at the cost of increasing the trusted code base: there is no guarantee that a user-defined extension does not introduce errors into either the decoder or encoder, violating conformance with the data format or introducing security vulnerabilities. This paper presents a framework, NARCISSUS, that solves both problems: it is both *extensible*, in that users can freely add new data formats to the specification language, and *sound*, in that every synthesized decoder and encoder is accompanied by a mechanized proof of correctness, checked by the Coq proof assistant [20], *regardless of any user-defined extensions*.

Input formats to NARCISSUS are specified as arbitrary injective binary relations between source data and its encoding in Coq's higher-order logic, allowing rich user-defined encoding strategies. From this specification, NARCISSUS synthesizes decoder functions that are guaranteed to both correctly invert the input specification and detect any malformed packet not conforming to the specified format. NARCISSUS consists of a small set of core definitions and provides a library of common binary formats and decoder implemen-

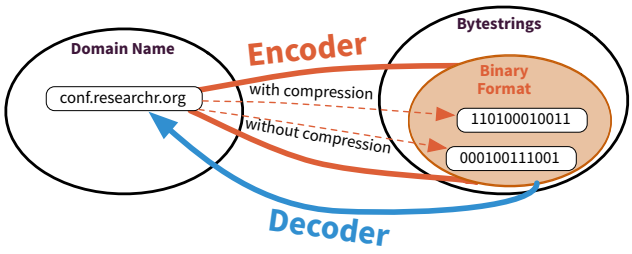


Figure 1. A decoder in NARCISSUS is guaranteed to be a proper inverse of the binary relation specifying a data format.

tation strategies that can be freely and safely extended by users, with nothing distinguishing user-provided components from those included in the library. The synthesis process is also extensible, in that users can add their own implementation strategies without breaking our correctness guarantees. This entire framework is realized in the Coq proof assistant, yielding a high degree of confidence in the derived code.

We evaluate the expressiveness of NARCISSUS by deriving decoders for several standard networking protocols, highlighting any protocol-specific extensions required. We furthermore evaluate the functionality and performance of the synthesized code by incorporating it into the networking stack of MirageOS [12].

1.1 Motivating Example

As a motivating example of NARCISSUS in action, consider the resource records used by hosts to respond to client queries in the domain name system, shown in Figure 2. Each field of this record has a different type: the name field holds the owner of the record as a list of labels, the class field is an enumerated type holding the protocol associated with the record (almost always Internet these days, as it is quite difficult to find CHAOS nodes), the ttl field holds a 32-bit word specifying the lifetime of the record for clients wanting to cache a response, and the rdata field is a variant type holding different kinds of information for the domain, e.g. an address, the authoritative server for a domain, or host information for the domain. This relatively simple example already illuminates some of the intricacies involved in both specifying and writing a correct decoder.

Importantly, the binary format for encoding these records, also shown in Figure 2, has been established for three decades [14], highlighting the need for NARCISSUS to be flexible enough to express existing protocols. The correctness criteria for the ttl and class fields are straightforward: our framework simply needs to ensure that the encoder and decoder agree on both the endianness used in ttl and the codes used for the enumerated type in the class field (and furthermore conform to the protocol specification), but the rdata and name fields require more subtle criteria. In order to properly decode the rdata field of a packet, the format specifies that two additional pieces of information be encoded alongside

```
Definition ResourceRecord :=
  {name : domain; (* Name of the data's owner *)
   class : enum; (* Protocol associated with record *)
   ttl : word; (* Maximum time record should be cached *)
   rdata : data (* Information specific to resource type *) }.
```

```
                                1  1  1  1  1  1
 0  1  2  3  4  5  6  7  8  9  0  1  2  3  4  5
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
/                      NAME                     /
/                                               /
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|                      TYPE                     |
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|                     CLASS                     |
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|                      TTL                      |
|                                               |
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|                    RDLENGTH                   |
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--|
/                      RDATA                    /
/                                               /
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
```

```
Definition encode_RRecord (r : resourceRecord) : Comp 𝕊:=
      encode_DomainName r!name
ThenC encode_Variant_Index RRTypeCodes r!rdata
ThenC encode_Enum RRClassCodes r!class
ThenC encode𝕎 r!ttl
ThenC encode_Length r!rdata
ThenC encode_Variant
          [encodeA; (* A; host address *)
           encodeNS; (* NS; authoritative name server *)
           encodeHINFO] r!rdata
DoneC.

Definition decode_RRecord
    : CorrectDecoderFor Valid_Cache Valid_RRecord
        encode_RRecord.
Proof.
  synthesize_decoder DomainName_Hints.
Defined.
```

Figure 2. Resource records and their encoded binary format used in the Domain Name System [14]

the resource records fields: a TYPE tag that dictates which variant type should be used when decoding the RDATA portion of the packet, and the RDLENGTH flag which is used to tell when to stop decoding certain types of resource data. Thus, the correctness of the decoder for rdata depends on previously decoded bits of information (which must have been decoded correctly themselves).

Furthermore, DNS packets use a novel form of compression for domain names, complicating the encoding of the name field: a domain name is either directly encoded as a list of labels terminated by a zero octet or as a (possibly empty) list of labels terminated by a pointer to the location of a previously encoded domain name. Thus, the domain

name `conf.researchr.org` could be encoded by either encoding the character strings `conf`, `researchr`, `org`, and the terminal character; or by encoding `conf` followed by a specialized pointer to the offset of a previous appearance of `researchr.org` in the packet. The pecularities of this compression strategy highlight the need of our framework to be not only extensible, but extensible in a *correctness-preserving* manner, that is, the framework should provide a strong guarantee that any user-provided extensions do not break our (as-of-yet unspecified) correctness properties. In order to do this, the framework may need to maintain invariants between the internal state of both the encoder and decoder (in the case of our example, that the cache used by both of them points to the correct location of previously seen domains). The DNS compression strategy points to another challenge, this time with respect to our specification of decoder correctness: since a domain name can be encoded in several different ways depending on if and where compression is used, there is no canonical encoding of a DNS packet according to the standard. Thus, a correct decoder is not the inverse of a single encoder function; rather it is the inverse of an encoding *relation* that specifies all the valid encodings of a particular data type.

This is precisely the intuition behind our approach to the derivation of correct-by-construction binary decoders, as illustrated in Figure 1. The first step is to capture the valid binary formats of a datatype as the possible outputs of a nondeterministic program, where different encodings are represented as a nondeterministic choice among available encoding strategies. encode_RRecord in Figure 2 is an example of such a nondeterministic encoder for DNS resource records. It is built up with a combination of formats for basic datatypes drawn from the core library, i.e. $\text{encode}_{\mathbb{W}}$ and encode_enum, and format combinators, i.e. **ThenC** and **DoneC**, for building up larger decoders from these building blocks. Users can freely include their own formats, written as nondeterministic programs, in this specification; encode_DomainName, the encoder for domain names used in encode_RRecord, is an example of such a user-provided extension.

A correct decoder is one that inverts this relation, precisely mapping any element in the set of bytestrings specified by such a format back to the member of the original datatype that generated it, and failing if the binary string does not fall in the set of valid encodings. The ValidDecoder type-level function used to define decoders is effectively the dependent pair of a decoder with a proof that it meets this specification; NARCISSUS frames the derivation of a correct-by-construction decoder as the interactive search for an inhabitant of this type à la decode_RRecord. The framework automates the search for an inhabitant of this type using the synthesize_decoder tactic, which can be extended with user-defined decoder implementation strategies. In the derivation of decode_RRecord in Figure 2, for example, these strategies are passed to this tactic as DomainName_Hints. The decoder functions synthesized by this tactic can then be extracted to OCaml via Coq's extraction mechanism.

We pause here to recap the features of NARCISSUS that distinguish it from existing frameworks:

- In contrast to interface generators, including XDR [19], ASN.1 [7], Apache Avro [1], and Protocol Buffers [21], where the tool decides the encoding format from a logical description of the source data, NARCISSUS provides *fine-grained control* over the precise shape of encoded data used in the binary relation specifying a format.
- The specification language in NARCISSUS is that of Coq's higher-order logic, and can be *extended* with new, user-defined formats, in contrast to traditional parser generators such as YACC [10], ANTLR [16], and binpac [15] with fixed input languages.
- NARCISSUS is *foundational*, in that it has a small core of trusted definitions: the base library of format specifications and decoder implementation strategies are components on top of this core. Other data-description frameworks, including PADS [8], PacketTypes [13], and Datascript [2], do not directly support user-defined language extensions and require modifying the large code base of the tools themselves.
- Finally, even those frameworks that easily support user-defined extensions, like Nail [4], provide no guarantees about the safety of those extensions. In contrast, NARCISSUS is *sound*, providing a machine-checked proof that even decoders built using user-defined extensions are correct by construction.

The rest of this paper is structured as follows: Section 2 describes the nondeterministic programs we use to specify datatype formats. Section 3 gives a precise specification for correct decoders. Subsection 3.1 describes how we package decoding strategies into composable building blocks. Section 4 describes the synthesize_decoder tactic for deriving correct-by-construction decoders, while Section 5 illustrates how an IP Checksum component is added to the library. Section 6 evaluates NARCISSUS via the synthesis of decoders for several standard binary formats, and Section 7 finishes with a discussion of related work.

2. Specifying Binary Formats

In NARCISSUS, a binary encoder is a function from an arbitrary data type T to an encoded representation in the type $\mathbb{S}$ of byte strings. Such a function is typed like $\text{encoder} : \mathsf{T} \to \mathbb{S}$, while a decoder is the inverse function that transforms the encoded representation back into the original type: $\text{decoder} : \mathbb{S} \to \mathsf{T}$. Section 3 describes precisely the desired relationship between the encoder and decoder. Our framework uses an abstract data type (ADT) for bit strings, $\mathbb{S}$, which is effectively a queue equipped with an append operator $+\!\!+$ to abstract the concrete details of the encoded representation from the

```
ADT 𝕊{
  Definition id : 𝕊;
  Definition ++ : 𝕊→ 𝕊→ 𝕊;
  Definition enqueue : 𝔹→ 𝕊→ 𝕊;
  Definition dequeue : 𝕊→ option (𝔹 × 𝕊);

  Axiom app_id_left : ∀ s1, id ++ s1= s1;
  Axiom app_id_right : ∀ s1, s1++ id = s1;
  Axiom app_assoc : ∀ s1s2s3, s1++ (s2 ++ s3) = (s1 ++ s2) ++ s3;
  Axiom dequeue_app : ∀ b s1s2s3, dequeue s1= Some (b, s2) →
                      dequeue (s1 ++ s3) = Some (b, s2++ s3);
  Axiom enqueue_app : ∀ b s1s2,
                      enqueue b (s1 ++ s2) = s1++ (enqueue b s2);
  Axiom dequeue_enqueue : ∀ b,
                      dequeue (enqueue b id) = Some (b, id);
  Axiom dequeue_id : dequeue id = None;
  Axiom dequeue_opt : ∀ b s1s2s3, dequeue s1= Some (b, s3) →
                      dequeue s2= Some (b, s3) → s1= s2
  ... }
```

Figure 3. The interface of the BitString ADT used for encoded data, with some length operations elided.

encoders, e.g. whether a bit is enqueued at the physical head or tail of a bit string. Figure 3 presents the interface of this ADT.

While a *particular* encoder is a deterministic function, there are many reasons that a binary format may allow several different encodings of a datatype instance. In addition to the optional domain-name compression given in Subsection 1.1, protocols can, for example, leave some parts of the bit string underspecified: the original specification of IPv4 [17] only used the lower six bits of the eight-bit type-of-service field, reserving the upper two bits for future use, while the TCP specification [18] leaves the contents of the urgent pointer byte unspecified when the urgent flag is set to zero. A user could also choose to throw away some encoded information when decoding a datatype by leaving it unspecified, for example only extracting the destination address from a UDP packet for use in a filter. In order to ensure the correctness of a synthesized decoder over the *entire* range of encodings allowed by a binary format, NARCISSUS captures all the possible encoders allowed by the format as a functional program in the nondeterminism monad, Comp.

The nondeterminism monad simply specifies a set of values, V, where each element represents a possible output of an implementation of that specification:

$$\text{Comp } V \triangleq \text{Set of } V$$

The monad includes a set-comprehension operator $\{x \mid P\ x\}$, which specifies a function via a defining property P on possible return values, as well as the standard **return** and **bind** ($\_ \leftarrow \_; \_$) operators. The three operators have straightforward interpretations as set operations:

$$\begin{aligned} \text{return } v &\triangleq \{v\} \\ \{x \mid P\ x\} &\triangleq \{x \mid P\ x\} \\ x \leftarrow y; k\ x &\triangleq \{v \mid \exists x.\ x \in y \ \wedge\ v \in k\ x\} \end{aligned}$$

As an example, we can specify the encoding of the type-of-service field in the original specification of IPv4 by appending two nondeterministically chosen bits onto the six-bit word in the type-of-service field used by a packet ip:

```
x ← { b : word2 | True }; return (x ++ ip!"ToS")
```

In addition to allowing NARCISSUS to capture formats with more than one valid encoding per source datatype, using the nondeterminism monad has the virtue of enabling more declarative specifications of encoders, which can in turn make it easier to prove the soundness of the corresponding decoder. As an example, consider how the following nondeterministic function separates the leftmost label in the domain name domain for encoding before processing the remaining labels in the domain[1]:

```
split domain := { (label, domain') | ValidLabel label ∧
                      (domain = label ++ "." ++ domain'
                    ∨ (label = domain ∧ domain' = "")) }
```

There is no nondeterminism in this set, in the sense that it includes (at most) a single element, but it allows an encoder specification to capture the essential functionality of a split function. A particular encoder can guarantee that this set will be inhabited by limiting domain to be a valid domain name, as we will see in subsequent sections. The nondeterminism monad can also conveniently express recursive formats without worrying about termination via a fixpoint combinator, FixComp, using the standard definition of the intersection of all sets closed under a generator function:

$$\text{FixComp } f \ = \{x \mid x \in f\ x\}$$

Combining these two definitions enables a succinct specification of the format of (uncompressed) domain names:

```
encodeDN domain ≜
  FixComp (λ rec. if domain = ""then return id
                  else (label, domain') ← split domain;
                       s1← encodeLabel label;
                       s2← rec domain';
                       return (s1 ++ s2) ).
```

In order to enable encoders to utilize internal state, we layer the state monad on top of the nondeterminism monad to create the full signature for binary formats used by NARCISSUS:

```
encoderSpec : A → Σ → Comp (𝕊× Σ )
```

The type Σ is a parameter of the framework, corresponding to whichever type of state a particular format requires. Figure 4 gives the specifications of encoders for fixed-length

[1] The ValidLabel predicate ensures that label does not contain invalid characters, such as ".".

```
Def encode_𝕎 (w : 𝕎_n) (σ : Σ) :=
  FixComp (λ rec. match w with
                  | WO ⇒ return (σ , id)
                  | WS b w' ⇒ rec w' (enqueue b s) σ
                  end).

Def encode_ℕ (n : Nat) (σ : Σ) :=
  encode_𝕎 (ℕToW_16 n) σ .

Def encode_List {T} (encode_T : T → Σ → Comp (Σ × 𝕊))
    (l : List_T) (σ : Σ) :=
  FixComp (λ rec. match l with
                  | [ ] ⇒ return (σ , id)
                  | a :: l' ⇒ (s_1, σ) ← encode_T a σ ;
                              (s_2, σ) ← rec l' σ ;
                              return (s_1 ++ s_2, σ)
                  end).

Def encode_ListTerm {T} t (encode_T : T → Σ → Comp (Σ × 𝕊))
    (l : List_T) (σ : Σ) :=
  FixComp (λ rec. match l with
                  | [ ] ⇒ encode_T t σ
                  | a :: l' ⇒ (s_1, σ) ← encode_T a σ ;
                              (s_2, σ) ← rec l' σ ;
                              return (s_1 ++ s_2, σ)
                  end).
```

Figure 4. Encoder specifications for fixed-length words and variable-length polymorphic lists with and without a terminal code.

words and polymorphic lists, pulled from the base library as nondeterministic, stateful functions. Note that the higher-order encoders for lists, encode_List and encode_ListTerm, are parameterized over an encoder for list elements, encode_T .

In addition to formats for base datatypes, the base library of NARCISSUS includes two encoder combinators for encoding more complex data, presented in Figure 5. The first of these, **ThenC**, sequences the results of two encoders, while **DoneC** syntactically signifies the end of the encoded bit string. The specification of an encoder for a list of the resource records from Figure 2 can be built up using these combinators and a function $\mathbb{N}\text{To}\mathbb{W}_{16}$ from natural numbers to 16-bit words:

```
Def encodeRRs (l : List ResourceRecord) :=
      encode_𝕎 ℕToW_16 | l |
ThenC encode_List encode_RRecord l
DoneC.
```

Intuitively, the formats for base datatypes and the format combinators make up the core of an embedded domain-specific language for building up formats representing binary relations between input data and bit strings. Users can, of course, incorporate their own syntax for base formats or combinators into this EDSL by providing denotations for their extensions.

Deriving Binary Encoders A correct encoder is one that maps every instance of the source datatype to a bit string falling into the set described by a format:

```
Def EncodeCorrect {T} (encoderSpec :T→ Σ → Comp (Σ × 𝕊))
```

```
Def compose (encode_1 : Σ → Comp (Σ × 𝕊))
            (encode_2 : Σ → Comp (Σ × 𝕊)) (σ : Σ) :=
  (s_1, σ) ← encode_1 σ ;
  (s_2, σ) ← encode_2 σ ;
  return (s_1 ++ s_2, σ).

encode_1 ThenC encode_2 ≜ compose encode_1 encode_2 .

encode_1 DoneC ≜ compose encode_1 (λ σ . return (id, σ )).
```

Figure 5. Encoder combinators included in NARCISSUS

```
                                (encoderImpl :T→ Σ → Σ × 𝕊) :=
∀ t σ . encoderImpl t σ ∈ encoderSpec t σ .
```

Given the specification of a format, it is possible to automatically derive a correct-by-construction implementation by iteratively refining the specification using a set of refinement rules à la the Fiat framework [5].

3. Specifying Binary Decoders

Before diving into the precise definition of decoder correctness used by NARCISSUS, we consider the high-level properties that such a "correct" decoder should satisfy. Given the specification of a binary format as a nondeterministic function encodeSpec, a correct decoder decode must clearly be a left inverse of encodeSpec, i.e. decode must map every element s in the image of t of encodeSpec back to t:

$$\text{decodeCorrect}_1\ \text{encodeSpec}\ \text{decode} \triangleq \forall\ t\ s, s \in \text{encodeSpec}\ t \rightarrow \text{decode}\ s = t$$

Less clear is what decode should do with bit strings that do not fall into the set allowed by the format: should the decoder fail on such inputs, or should its behavior be unconstrained? In the interest of detecting malformed and potentially malicious inputs, NARCISSUS takes the former approach, requiring that a correct decoder be a partial function that *only* translates correctly encoded data:

$$\text{decodeCorrect}_2\ \text{encodeSpec}\ \text{decode} \triangleq \forall\ t\ s, \text{decode}\ s = t \rightarrow s \in \text{encodeSpec}\ t$$

While the rest of this paper focuses on deriving decoders satisfying both criteria, our approach also straightforwardly applies to the synthesis of decoders only satisfying the first criterion, in the case that users choose to sacrifice strict format validation for efficiency.

We can now identify the precise signature of decoders in NARCISSUS. An important design element of the framework is to treat decoders as consumers of the queue provided by the bit string ADT. In other words, a decoder decodes some portion of the head of the bit string and returns the rest alongside a decoded value, as shown in the decoder corresponding to $\text{encode}_{\mathbb{W}}$ from Figure 4:

```
Def decode_𝕎 n (s : 𝕊) (σ :Σ) :=
  match n with
  | 0 ⇒ return (WO, s, σ)
  | S n' ⇒ (b, s') ← dequeue s;
```

```
Def decode_N (s : S) (σ :Σ ) :=
    (w, s', σ ') ← decode_W 16 s σ ;
    return (W_16ToN w, s', σ ')

Def decode_List {T}
  (decode_T : S→ Σ → Comp (T × Σ × S)) n (s : S) (σ  : Σ ) :=
  match n with
    | O ⇒ return (nil, s, σ )
    | S n' ⇒ (t, s', σ ') ← decode_T s σ ;
             (l, s', σ ') ← decode_List s' σ ';
             return (t :: l, s', σ ')
  end.

Def decode_ListTerm {T}
  (decode_T : S→ Σ → Comp (T × Σ × S)) t (s : S) (σ  : Σ ) :=
  (t', s', σ ') ← decode_T s σ ;
  if (t' = t) then
     return ([ ], s', σ ')
  else
     (l, s', σ ') ← decode_ListTerm t s' σ ';
     return (t' :: l, s', σ ').
```

Figure 6. Decoders for the binary formats from Figure 4.

```
            (w, s'') ← decode_W n' s' σ ;
            return (WS b w, s'', σ )
  end.
```

As shown above, decoders are also monadic functions, although in contrast to encoder functions, they combine the state monad with the option (failure) monad in order to model the partiality of decoders. In $\mathsf{decode}_{\mathbb{W}}$, for example, should a dequeue fail because s is empty, the entire decoder will terminiate with an error value. Thus, the complete signature for a decoder for a datatype T in our framework is:

$$\mathsf{decode}_{\mathsf{T}} : \mathbb{S} \to \Sigma \to \mathsf{Option}\ (\mathsf{T} \times \mathbb{S} \times \Sigma)$$

Figure 6 shows the decoders for the other binary formats. Even these simple types demonstrate that the decoders are more complicated than the corresponding encoders, precisely because they do not have a full view of the original data. In the case of decode_List, for example, the decoder takes the number of iterations as input; in order for a compound decoder using this function to be correct, it needs to be able to recover this parameter prior to decoding the list. A decoder for the binary format specified by encodeRRs, for example, does precisely this:

```
Def decodeRRs (s : S) (σ :Σ ) :=
    (n, s', σ ') ← decode_N s σ ;
    decode_List decode_RRecord n s' σ .
```

Before giving the complete definition of decoder correctness, we note that it is often convenient to condition it on two additional assumptions. Firstly, the statement restricts the data being encoded via a predicate in order to only consider well-formed input, i.e. DNS packets that only have valid domain names or trees that have been normalized. A similar result could be achieved by using subset types as the source datatype in lieu of an extra assumption, but externalizing the well-formedness condition frees NARCISSUS from having to worry about subtleties of dependent typing. Furthermore, this predicate is a natural means for retaining information about previously parsed data when verifying decoder combinators, as we will see in the next section. Secondly, the statement requires that decoders are run in appropriate states via relations between the state used to encode the data and the state used by the decoder. Finally, NARCISSUS defines the judgment of decoder correctness with respect to a binary format as a quaternary relation on a state relation Q, a data predicate P, a binary format encode, and a decoder function decode:

$$\mathsf{Q}, \mathsf{P} \vdash \mathsf{encode} \circlearrowright \mathsf{decode} :=$$

$$\begin{aligned}
&(\forall\ (\sigma_E\ \ \sigma_E' : \Sigma_E)\ (\sigma_D : \Sigma_D)\ (\mathsf{data} : \mathsf{A})\ (\mathsf{s}\ \mathsf{s}' : \mathbb{S}),\\
&\quad \mathsf{Q}\ \sigma_E\ \sigma_D \to \mathsf{P}\ \mathsf{data} \to\\
&\quad (\mathsf{s}, \sigma_E') \in \mathsf{encode}\ \mathsf{data}\ \sigma_E \to\\
&\quad \exists\ \sigma_D',\ \mathsf{decode}\ (\mathsf{s} +\!\!+ \mathsf{s}')\ \mathsf{env}' = \mathsf{Some}\ (\mathsf{data}, \mathsf{s}', \sigma_D')\\
&\qquad \wedge\ \mathsf{Q}\ \sigma_E'\ \sigma_D') \wedge\\
&(\forall\ (\sigma_E : \Sigma_E)\ (\sigma_D\ \ \sigma_D' : \Sigma_D)\ (\mathsf{data} : \mathsf{A})\ (\mathsf{s}\ \mathsf{s}' : \mathbb{S}),\\
&\quad \mathsf{Q}\ \sigma_E\ \sigma_D \to\\
&\quad \mathsf{decode}\ \mathsf{s}\ \sigma_D = \mathsf{Some}\ (\mathsf{data}, \mathsf{s}', \sigma_D') \to\\
&\quad \exists\ \mathsf{s}''\ \sigma_E',\ (\mathsf{s}'', \sigma_E') \in \mathsf{encode}\ \mathsf{data}\ \sigma_E\\
&\qquad \wedge\ \mathsf{s} = \mathsf{s}'' +\!\!+ \mathsf{s}' \wedge \mathsf{P}\ \mathsf{data} \wedge \mathsf{Q}\ \sigma_E'\ \sigma_D').
\end{aligned}$$

3.1 Building Decoding Strategies

Figure 7 presents the correctness rules for $\mathsf{decode}_{\mathbb{W}}$ and the decoders in Figure 6. NATCORRECT has the first example of a data predicate, constraining the natural numbers that can be safely encoded and decoded using $\mathsf{encode}_{\mathbb{N}}$ and $\mathsf{decode}_{\mathbb{N}}$ to those that can be stored in a byte. While both $\mathsf{decode}_{\mathbb{W}}$ and decode_List recurse over the lengths of the datatype they are decoding, they are distinguished by how this parameter is encoded in the binary format. NARCISSUS uses length-indexed words, that is, the number of bits in a word is encoded in its type, so the length parameter used by $\mathsf{decode}_{\mathbb{W}}$ is completely determined by the type of data being decoded. Since most formats use fixed-length words, this length parameter is embedded in the format itself. Contrast this with decode_List, where the length is not fixed by the data type: here we rely on a predicate relating the list l being encoded to the parameter n passed to decode_List: $|l| = n$. Note that the variable l is the argument of the data predicate and stands in for *any* list used in the correctness judgment. Thus, we are not able to pass $|l|$ as an argument to decode_List, rather this length must be recovered by, for example, encoding it in the format as well. This may in turn limit the length of the list itself, depending on the encoder used for this value. decode_ListTerm does not require such a parameter, as its correctness rule hinges on another condition: the data predicate stipulates that the terminal character must not be in the encoded list, allowing the decoder to determine when the end of the list has been reached. Finally, we note that the correctness of both decode_List and decode_ListTerm depend on the correctness of the decoder used for list elements $\mathsf{decode}_{\mathsf{T}}$ *and* require that each element of the list satisfies the predicate P_{T} used in the proof of correctness.

State Predicates The encoders and decoders referenced in Figure 7 are pure, so each corresponding correctness rule is

$$\frac{}{\mathsf{Q}, \mathbb{T} \vdash \mathsf{encode}_{\mathbb{W}} \rightleftarrows \mathsf{decode}_{\mathbb{W}}\ \mathsf{n}} \text{ (WordCorrect)}$$

$$\frac{}{\mathsf{Q}, \lambda m.\ m < 2^n \vdash \mathsf{encode}_{\mathbb{N}} \rightleftarrows \mathsf{decode}_{\mathbb{N}}\ \mathsf{n}} \text{ (NatCorrect)}$$

$$\frac{\mathsf{Q}, \mathsf{P_T} \vdash \mathsf{encode_T} \rightleftarrows \mathsf{decode_T}}{\begin{array}{c}\mathsf{Q}, \lambda l.\ |l| = n\ \wedge\ \forall a \in l.\ \mathsf{P_T}\ a\ \vdash \\ \mathsf{encode\_List}\ \mathsf{encode_T} \rightleftarrows \mathsf{decode\_List}\ \mathsf{decode_T}\ n\end{array}} \text{ (ListCorrect)}$$

$$\frac{\mathsf{Q}, \mathsf{P_T} \vdash \mathsf{encode_T} \rightleftarrows \mathsf{decode_T}}{\begin{array}{c}\mathsf{Q}, \lambda l.\ t \notin l\ \wedge\ \forall a \in l.\ \mathsf{P_T}\ a\ \vdash \\ \mathsf{encode\_ListTerm}\ \mathsf{encode_T} \rightleftarrows \mathsf{decode\_ListTerm}\ \mathsf{decode_T}\ t\end{array}} \text{ (ListTCorrect)}$$

Figure 7. Correctness rules for the decoders for the binary formats from Figure 6

parameterized over an arbitrary relation Q between encoder and decoder states. For an example of a more constrained rule, recall the format for domain names, encode_DomainName, discussed in Subsection 1.1. encode_DomainName compresses domain names by using a map from previously seen domain names to their locations in the bit string in order to (potentially) replace repeated domain names with a pointer to that location. The corresponding decoder decode_DomainName maintains a map from previously parsed locations to domain names; this map needs to be aligned with the one used in the format in order for this decoder to be correct. This relationship is embodied in the correctness rule for decode_DomainName:

$$\frac{}{\begin{array}{c}\lambda\sigma_{\mathsf{E}}\ \sigma_{\mathsf{D}}.\forall \mathsf{p}\ \mathsf{n}.\ \sigma_{\mathsf{E}}\ \mathsf{p} = \mathsf{n} \leftrightarrow \sigma_{\mathsf{D}}\ \mathsf{n} = \mathsf{p}, \mathsf{ValidDomain} \vdash \\ \mathsf{encode\_DomainName} \rightleftarrows \mathsf{decode\_DomainName}\end{array}} \text{ (DomainNameCorrect)}$$

3.2 Decoder Combinators

We now turn to the rules for building correct decoders for the **ThenC** combinator used to sequence two encoders. These rules, shown in Figure 8, allow framework users to derive a valid decoder for a complex format by finding correct decoders for each subformat. Of the two rules for **ThenC**, ComposeCorrect handles the case when the second decoder depends on the output of the first, while ComposeCorrectNoDep handles the case when the two decoders are independent. The latter rule is useful for decoding fixed formats which do not depend on the object being encoded, e.g. the version numbers in an IP packet. The first format in ComposeCorrect uses a projection function, π_B, to encode some partial view of the full datatype a, e.g. a particular field of a record or the length of a list, before encoding the rest of a. The decoder for the compound format first uses a decoder for the domain of π_B, $\mathsf{decode_B}$, to recover $\pi_B\ a$ before

$$\frac{\begin{array}{c}\mathsf{Q}, \mathsf{P_B} \vdash \mathsf{encode_B} \rightleftarrows \mathsf{decode_B} \qquad \forall a.\ \mathsf{P_A}\ a \rightarrow \mathsf{P_B}\ (\pi_B\ a) \\ \forall b.\ \mathsf{P_B}\ b \rightarrow \\ \mathsf{Q},\ \lambda a.\ \mathsf{P_A}\ a \wedge \pi_B\ a = b\ \vdash \mathsf{encode_A} \rightleftarrows \mathsf{decode_A}\ b\end{array}}{\begin{array}{ll}\mathsf{Q}, \mathsf{P_A} \vdash & \mathsf{encode_B}\ (\pi_B\ a) \\ & \textbf{ThenC}\ \mathsf{encode_A}\ a\end{array} \rightleftarrows \begin{array}{l}(b, s', \sigma_D') \leftarrow \mathsf{decode_B}\ s\ \sigma_D; \\ \mathsf{decode_A}\ b\ s'\ \sigma_D'\end{array}} \text{ (ComposeCorrect)}$$

$$\frac{\begin{array}{c}\mathsf{Q}, \mathsf{P_B} \vdash \mathsf{encode_B} \rightleftarrows \mathsf{decode_B} \qquad \mathsf{P_B}\ b \\ \mathsf{Q}, \mathsf{P_A} \vdash \mathsf{encode_A} \rightleftarrows \mathsf{decode_A}\end{array}}{\begin{array}{ll}\mathsf{Q}, \mathsf{P_A} \vdash & \mathsf{encode_B}\ b \\ & \textbf{ThenC}\ \mathsf{encode_A}\ a\end{array} \rightleftarrows \begin{array}{l}(b', s', \sigma_D') \leftarrow \mathsf{decode_B}\ s\ \sigma_D; \\ \textbf{if}\ (b = b')\ \textbf{then}\ \mathsf{decode_A}\ a'\ s'\ \sigma_D' \\ \qquad \textbf{else}\ \mathsf{None}\end{array}} \text{ (ComposeCorrectNoDep)}$$

$$\frac{\forall \mathsf{a}.\ \mathsf{P_A}\ \mathsf{a}\ \rightarrow \mathsf{a} = \mathsf{a}'}{\begin{array}{ll}\mathsf{Q}, \mathsf{P_A} \vdash & \\ \lambda\ \mathsf{a}\ \sigma_{\mathsf{E}}.\ \textbf{return}(\mathsf{id}, \sigma_{\mathsf{E}}) & \rightleftarrows\end{array} \begin{array}{l}\textbf{if}\ \mathsf{P_A}\ a'\ \textbf{then return}(a', s, \sigma_D) \\ \qquad \textbf{else}\ \mathsf{None}\end{array}} \text{ (DoneCorrect)}$$

Figure 8. Rules for decoder combinators

processing the rest of the bit string using a decoder, $\mathsf{decode_A}$, which takes the decoded $\pi_B\ a$ as input.

The validity of ComposeCorrect depends on how it threads information about the projection of the encoded data through its three hypotheses. The first hypothesis establishes that $\mathsf{decode_B}$ is a correct decoder for $\mathsf{encode_B}$ under some data predicate $\mathsf{P_B}$; the second states that this predicate holds for any projection from data a which satisfies the predicate used by the composite decoder $\mathsf{P_A}$. Taken together, these two hypotheses establish that $\mathsf{decode_B}$ safely recovers the data encoded in the first piece of the format, which is then available to $\mathsf{decode_A}$ when parsing the rest of the string. The proof that the rest of the string is correctly decoded is captured by the final hypothesis of ComposeCorrect, which is quantified over an arbitrary value of this parameter, b. The hypothesis augments the proof of correctness for $\mathsf{decode_A}$ with two pieces of information relating b to the data obtained from the first part of the format. Firstly, the correctness of $\mathsf{decode_B}$ ensures that its data predicate holds for b: $\mathsf{P_B}\ \mathsf{b}$. Secondly, recall that the statement of $\rightleftarrows$ is quantified over an *arbitrary* piece of data, a, constrained by a predicate $\mathsf{P_A}$; we include the knowledge that b was derived from $\pi_B\ a$ into the proof of $\mathsf{encode_A} \rightleftarrows \mathsf{decode_A}$ by augmenting its data predicate with this fact: $\lambda a.\ \mathsf{P_A} a \wedge \pi_B\ a = b$.

In contrast, the identifier b used in ComposeCorrectNoDep is a *constant* that is independent of the original data. This rule avoids polluting the data predicate in the proof of correctness for $\mathsf{decode_A}$ with an extraneous clause, at the cost of proving that the data predicate $\mathsf{P_B}$ used by $\mathsf{encode_B}$ holds for this constant. In order to ensure that the encoded bit string

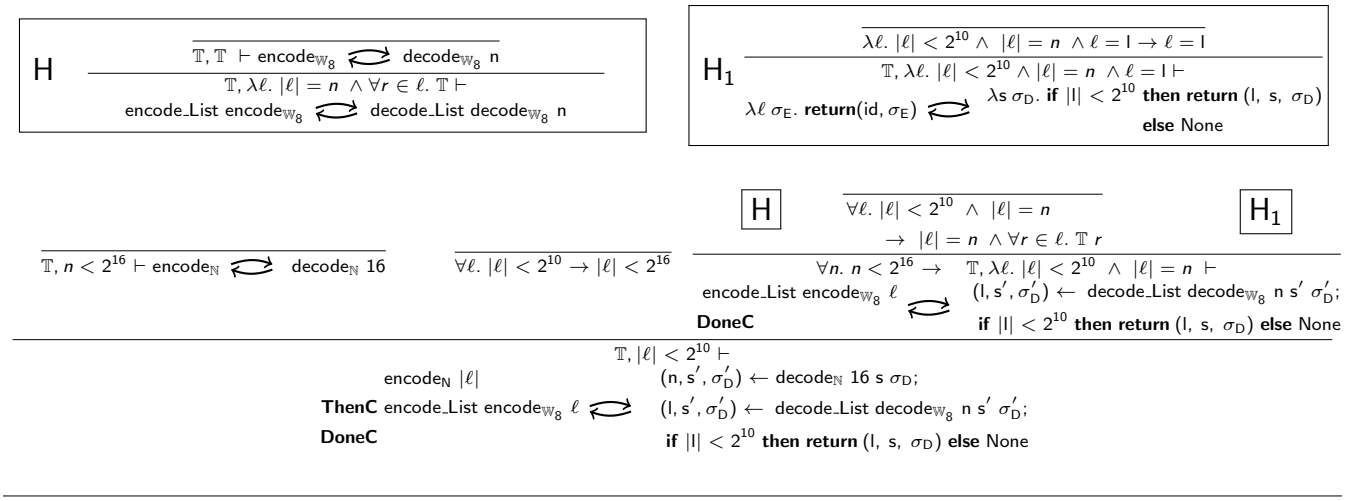


Figure 9. Proof of correctness of a decoder for a list of bytes.

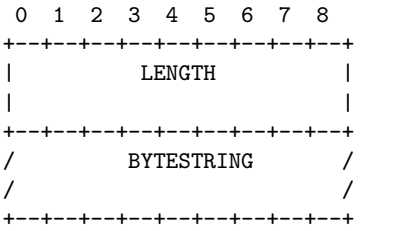
conforms to the format, per the second clause of ⇄ , the decoder *must* check that the decoded constant is equal to b, via the **if** (b' = b)**then** ... **else** ... expression in the decoder.

The final rule presented in Figure 8, DONECORRECT, finishes the proof for a decoder for a format terminated by **DoneC**. It may appear that this rule can only be used for decoders of a single fixed value, since its sole hypothesis stipulates that the data predicate $\mathsf{P_A}$ requires that the encoded data must be some constant a'. As discussed above, however, $\mathsf{P_A}$ accumulates additional constraints when parsing formats built up using **ThenC**. Thus, a proof terminated with DONECORRECT will include constraints relating the encoded data to all of the previously parsed views of that data. Assuming that the binary format includes enough information to uniquely identify the original data, the constraints in $\mathsf{P_A}$ will force the decoded data to be *exactly* that used to generate the original bytestring, ensuring the correctness of the full decoder.

One important feature of the decoder in DONECORRECT is the conditional validating the data predicate $\mathsf{P_A}$, as required by ⇄ **if** $\mathsf{P_A}$ a' **then**...**else**...[2]. When $\mathsf{P_A}$ follows immediately from the binary format, e.g. the length of any list decoded by decode_List must be less than the maximum value that can be encoded by the format used for its length, this conditional can be discarded via partial evaluation (indeed, our automation removes trivial conditionals in just this manner). If the predicate is more restrictive, for example requiring that a list hold fewer than five elements while using a byte to encode the length, the decoder must check that the data is well-formed.

Figure 9 gives an example application of COMPOSECORRECT and DONECORRECT in a proof of correctness for a binary format that formats a list of at most 2^{10} bytes as a sixteen-bit word encoding its length followed by the sequence of bytes in the list:

[2] We overload our syntax here to treat $\mathsf{P_A}$ as a decision procedure for the predicate $\mathsf{P_A}$ when used in an if statement.



The leaves of the proof trees consist of rules for basic data types (natural numbers and lists) and the DONECORRECT rule which finishes the proof. The leftmost leaf uses NatCorrect, requiring that the encoded length fit into a sixteen-bit word. Although this is different from the predicate used by the overall format, the second clause of COMPOSECORRECT ensures that this condition holds. A similar weakening step occurs in the other application of COMPOSECORRECT, where the trivial condition on the elements of the encoding needed by LISTCORRECT, $\forall r \in \ell, \mathbb{T}\ r$, is dropped from the data predicate. Observe that the data predicate used in the proof trees for the right branch of **ThenC** is augmented with information about the parsed string as the proof proceeds, so that when the proof reaches the end of the format, the predicate dictates that the length of the encoded string be the result of the initial decoder function, $|\ell| = \mathsf{n}$, and the encoded list be the result of the second decoder $\ell = \mathsf{l}$. This predicate constrains the encoded datatype to be exactly l, allowing the decoder to reconstitute the initial list from the data it has decoded. While the last two clauses in the predicate are guaranteed by the format, the decoder still needs to check that l meets the original length requirements via the **if** $|\mathsf{l}| < 2^{10}$ **then** ... **else** ... statement. Of course, if this check is too restrictive, the user could relax the initial data predicate to $|l| < 2^{16}$, as guaranteed by the format, allowing this statement to be simplified away.

Figure 10 includes a complete list of the components currently included in NARCISSUS. As may be expected, the lines of proof needed to verify ⇄ typically outnumber the

Format	LoC	LoP	Higher-order
Sequencing (**ThenC**)	7	164	✓
Termination (**DoneC**)	1	28	✓
Conditionals (IfC)	25	204	✓
Booleans	4	24	✗
Fixed-length Words	65	130	✗
Unspecified Field	30	60	✗
List with Encoded Length	40	90	✗
String with Encoded Length	31	47	✗
Option Type	5	79	✗
Ascii Character	10	53	✗
Enumerated Types	35	82	✗
Variant Types	43	87	✗
Domain Names	86	671	✗
IP Checksums	15	1064	✓

Figure 10. Formats included in NARCISSUS with lines of code (LoC) and lines of proof (LoP).

lines of code needed to specify the format and implement the decoding functions. Additionally, each row indicates whether the format is a higher-order combinator taking a bit string produced by another format as input, like **ThenC**. Included at the bottom are the two format-specific components we built for the network protocols used in our case study: domain names with compression used in DNS packets, and IP checksums used in IP, TCP, and UDP packets.

4. Deriving Correct-By-Construction Decoders

NARCISSUS frames the decoder-synthesis problem as a user-guided search for both a decoder function for the encoded type *and* a proof that it is correct with respect to a specified format. We define a type alias, CorrectDecoderFor, previously seen in Figure 2, that packages these two terms together as a dependent pair. CorrectDecoderFor is parameterized over the state predicate StateOK, the data predicate DataOK, and the format specification FormatSpec:

```
Definition CorrectDecoderFor StateOK DataOK FormatSpec :=
  Σ decode. StateOK, DataOK ⊢ FormatSpec ⟲ decode.
```

NARCISSUS includes a tactic synthesize_decoder for automatically finding inhabitants of CorrectDecoderFor. The behavior of this tactic is broken into three phases:

Format Normalization This initial phase unfolds the identifiers of the parameters to CorrectDecoderFor, uses the monad laws to normalize the format specification, simplifies the resulting term, and caches any string constants via an OCaml plugin to minimize the size of intermediate proof terms in order to speed up the rest of the derivation.

Syntax-Directed Search The main body of the tactic searches for an initial decoder implementation by repeatedly applying ⟲ lemmas. This search consists of a series of rules, triggered on the syntax of the current format in the goal. In addition to specifying which lemma to apply, these rules also attempt to discharge any hypotheses dealing with data invariants via a hint database. The rule for COMPOSECORRECT, for example, tries to automatically show that the data invariant for the compound decoder entails the data invariant used by the first decoder by using this database.

The most interesting of these rules is the one that tries to finish a derivation by applying DONECORRECT. This rule produces two proof obligations: showing that the original data can be completely reconstituted using previously parsed data, and finding a decision procedure for the current data predicate. The first obligation takes the form of an equality between the original data a and some unknown existential variable: $a = ?_a$. The finishing tactic tries to solve this goal by using the equations built up in the data predicate during the correctness proof to rewrite the original encoded datatype into a term built entirely from parsed data, before unifying the resulting term with the $?_a$. Since a is not in the scope of the $?_a$, this latter step only succeeds when the rewritten term only depends on parsed data. The second obligation also uses an existential variable, this time to represent the decision procedure for the data predicate: $\mathsf{P_A}\, a' \leftrightarrow ?_{\mathsf{dec}}\, a' = \mathsf{true}$. The tactic tries to discharge this goal by composing known decision procedures for predicates in $\mathsf{P_A}$ and simplifying away any tautologies, including any predicates built up during the correctness proof.

Decoder Optimization The final phase of synthesize_decoder tries to further simplify the synthesized decoder by rewriting with the monad laws and removing any redundant tests in the decision procedure synthesized in the previous step.

5. Extending the Framework

An extension to NARCISSUS consists of four pieces: a format, a decoder function, a proof of a ⟲ rule relating the two, and automation for incorporating this rule into synthesize_decoder. As an example, consider the format of the Internet Protocol (IP) checksum used in the IP headers, TCP packets, and UDP packets featured in our case studies. Figure 11 presents the first three pieces of this extension. Note that encode_IPChecksum is a higher-order format combinator in the spirit of **ThenC**; the key difference is that it uses the bytestrings produced by the two format parameters to build the IP checksum (the one's complement of the bytestrings interpreted as lists of bytes), which it then inserts between the two encoded bytestrings in the output bytestring. decode_IPChecksum first validates the checksum before decoding the rest of the string; the ⟲ rule for this decoder guarantees this test will always succeed for uncorrupted data, and can safely avoid parsing the rest of the string otherwise.

```
Definition encode_IPChecksum encode1 encode2 σ_E :=
  (s, σ_E ') ← encode1 σ_E ;
  (s', σ_E ') ← encode2 σ_E ;
  c ← { c | onesComplement (𝕊2CharList (s ++ c ++ s'))
            = 1111 1111 1111 1111 };
  return (s ++ c ++ s', σ_E ')
```

```
Definition decode_IPChecksum decode1 decode2 s σ_D :=
  if onesComplement (𝕊2CharList s) = 1111 1111 1111 1111
  then (a', s', σ_D ') ← decode1 s σ_D ;
       (_, s', σ_D ') ← decode_𝕎 s' σ_D ';
       decode2 a' s' σ_D '
  else None
```

$$\frac{(\forall b\ s\ \sigma_E\ \sigma_E'.\ \mathsf{P_B}\ b \to (s, \sigma_E') \in \mathsf{encode_1}\ b\ \sigma_E \to |s| = \mathsf{n}) \quad \mathsf{n} \bmod 8 = 0 \quad (\forall a\ s\ \sigma_E\ \sigma_E'.\ \mathsf{P_A}\ a \to (s', \sigma_E') \in \mathsf{encode_2}\ a\ \sigma_E \to |s'| = \mathsf{n}') \quad \mathsf{n}' \bmod 8 = 0}{\mathsf{Q}, \mathsf{P_B} \vdash \mathsf{encode_2} \rightleftarrows \mathsf{decode_2} \quad \forall a.\ \mathsf{P_A}\ a \to \mathsf{P_B}\ (\pi_B\ a) \quad \forall b.\ \mathsf{P_B}\ b \to \mathsf{Q},\ \lambda a.\ \mathsf{P_A}\ a \wedge \pi_B\ a = b\ \vdash \mathsf{encode_1} \rightleftarrows \mathsf{decode_1}\ b}$$

$$\mathbb{Q}, \mathsf{P_A} \vdash \mathsf{encode\_IPChecksum}\ (\mathsf{encode_1}(\pi_B a))\ (\mathsf{encode_2}\ a) \rightleftarrows \mathsf{decode\_IPChecksum}\ \mathsf{decode_1}\ \mathsf{decode_2}$$

(IPCORRECT)

Figure 11. Binary format, decoder, and correctness rule for IP Checksums.

```
Definition encode_IPv4_Packet_Spec (ip4 : IPv4_Packet) :=
  encode_IPChecksum
    (encode_𝕎  (natToWord 4 4)
      ThenC encode_ℕ 4 (IPv4_Packet_Header_Len ip4)
      ThenC encode_unused_W 8
      ThenC encode_𝕎 ip4!TotalLength
      ThenC encode_𝕎 ip4!ID
      ThenC encode_𝕎 1
      ThenC encode_𝕎 ip4!DF
      ThenC encode_𝕎 ip4!MF
      ThenC encode_𝕎 ip4!FragmentOffset
      ThenC encode_𝕎 ip4!TTL
      ThenC encode_enum ProtocolTypeCodes ip4!Protocol
      DoneC)
    (encode_𝕎  ip4!SourceAddress
      ThenC encode_𝕎 ip4!DestAddress
      ThenC encode_List encode_𝕎 ip4!Options
      DoneC).
```

Figure 12. Format for IP version 4, utilizing the IP Checksum format.

The correctness rule for decode_IPChecksum, IPCORRECT, closely mirrors COMPOSECORRECT, adding additional assumptions that the bytestrings produced by each subformat have constant length and are properly byte-aligned; these assumptions are needed to prove the validity of the initial checksum test. Integrating this combinator into synthesize_decoder requires adding a new rule to the main loop of the tactic, keyed on the encode_IPChecksum combinator. After applying IPCORRECT, this rule attempts to discharge the assumptions by rewriting with a database of facts about the lengths of encoded datatypes and the modulus operator, relying on the body of the main loop to synthesize decoders for the subformats. Figure 12 presents an example of the checksum combinator being used in the format for the IP headers.

6. Evaluation

To evaluate the expressiveness of NARCISSUS, we implemented parsers for the set of network stack protocols shown in Figure 13. We note that the specifications of these formats are relatively short, consisting of on average roughly 180 lines of code. The specifications for DNS, IP, UDP, and TCP packets require adding the additional combinators to the framework noted in Figure 10 but do not require modifications to the core set of definitions. The source code for these parsers is included in the accompanying supplementary material.

Protocol	LoC	Interesting Features
Ethernet	150	Multiple format versions
ARP	41	
IP	141	IP Checksum; underspecified fields
UDP	115	IP Checksum with pseudoheader
TCP	181	IP Checksum with pseudoheader; underspecified fields
DNS	474	DNS compression; variant types

Figure 13. Formats included in case studies, with lines of code and interesting features highlighted.

The decoders that our framework produces are reasonably efficient and sufficiently full-featured to be used as a drop-in replacements for all parsing components of a typical TCP/IP stack. We support this claim by extracting our decoders to OCaml and integrating the resulting code into the pure-OCaml TCP/IP stack of MirageOS.

MirageOS [12] is a "*library operating system that constructs unikernels for secure, high-performance network applications*": a collection of OCaml libraries that can be assembled into a stand-alone kernel running on top of the Xen hypervisor. Security is a core feature of MirageOS, making it a natural target to integrate our decoders into. Concretely, this entails patching the `mirage-tcpip`[3] library to replace its packet deserializers by our own verified decoders and evaluating the performance and functionality of the resulting library. This allowed us to benchmark a realistic network application, the `mirage.io` website (`mirage-www`[4]), off a networking stack enhanced with verified decoders. This shows that the overhead of using our decoders in real-life applications is very small.

[3] https://github.com/mirage/mirage-tcpip

[4] https://github.com/mirage/mirage-www

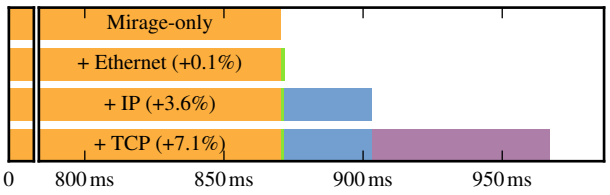


Figure 14. Average page load times with 4 TCP/IP stacks.

By themselves, our decoders are about fifty times slower than Mirage's built-in decoders. The comparison is not entirely fair, however: we do significantly more validation (including checksum computations that require re-encoding a pseudoheader after decoding a TCP or UDP packet, where Mirage simply reads a few fields from an in-memory buffer), we rely on intermediate structures to ease integration with Mirage, and some parts of our framework are simply not heavily optimized (for example, our decoders fetch data bit-by-bit from the underlying data buffers; we could instead fetch whole bytes when the decoder performs byte-aligned reads).

Set up Starting with the derived encoders, we use Coq's extraction mechanism to obtain OCaml libraries compatible with Mirage (extraction is mostly straightforward, though we customize it to use OCaml's `Int64` type for machine words, regular OCaml integers for numbers, and an efficient custom-built bit-string library to represent the bit buffers used for encoding and decoding). We then replace the TCP, UDP, IPv4, ARPv4, and Ethernet unmarshalling modules of the `mirage-tcpip` library with simple wrappers around our own extracted code and recompile the library and the Mirage website (patches against Mirage's public repository[5] and setup instructions are included as supplementary materials). This whole process went very smoothly: Mirage's test suite did not reveal issues with our decoders (we did have to adjust two tests: one of them used an IP packet with an EtherType that we do not support, and the other one unmarshalled a packet with an incorrect checksum, causing our decoders to reject it).

Mirage's website We recompile the Mirage website to serve `mirage.io` atop a full user-space TCP/IP stack and measure page load times as we replace each decoder in the stack by its verified counterpart. We focus on the "blog" page of the website: loading it causes the client to fetch about 4 MB of data, obtained through 36 HTTP requests spread across 1040 TCP packets (covering page content, images, scripts, and stylesheets). We used the debugging API of the Chromium browser to accurately measure page load time. Figure 14 shows the average page load time across ten pages loads: overall, transitioning all decoding to Fiat incurs a 7% page load overhead.

[5] https://github.com/mirage/mirage

7. Related Work

Existing Frameworks There exist a number of frameworks for generating parsers and encoders. Traditional parser generaters like YACC (Yet Another Compiler-Compiler) [10] and ANTLR [16] automatically generate parser implementations from declarative Backus–Naur form specifications. These frameworks decode inputs into the abstract syntax tree (AST) defined by the input grammar; users must post-process this AST (possibly using *semantic actions*) to convert it into the desired representation. Semantic actions are effectively user-defined programs that can easily introduce errors into the synthesized decoder.

An alternative approach is taken by interface generators like XDR [19], ASN.1 [7], Apache Avro [1], and Protocol Buffers [21], which generate encoders and decoders from user-defined data schemes. The underlying data format is defined by the system, however, preventing data exchange between programs using different frameworks. More importantly, the lack of fine-grained control over the physical data representation prevents users from extending the format in anyway, disallowing the use of interface generators with standardized formats like those used by network protocols.

The binpac compiler [15] supports a data-format specification language specifically developed for network protocols, including various general-purpose language constructs. Although binpac generates efficent protocol parsers for existing network protocols, these is no support for extending the specification language outside of the builtin constructs, and the system provides no formal guarantees about the generated code. More recent frameworks, like PADS [8], PacketTypes [13], and Datascript [2], feature sophisticated data-description languages with support for complex data dependencies and constraints for specific data schemes. Extending these frameworks with new encoding strategies requires directly modifying each system's code generator, again with no guarantees about the correctness of the extended system.

Nail [4] is a tool for synthesizing parsers and generators from a high-level format. Nail unifies the data-description format and internal data layout into a single specification and allows users to specify and automatically check dependencies between encoded fields. More importantly, Nail natively supports extensions to its parsers and generators via user-defined *stream transformations* on the encoded data, allowing it to capture protocol features that other frameworks cannot. However, Nail provides no formal guarantees, and these transformations can introduce bugs violating the framework's safety properties. We also note two other differences between Nail and NARCISSUS. First, Nail has many more orthogonal primitives than NARCISSUS has, as our primitives may be considered to be little more than the definition of decoder correctness. Second, while Nail provides flexibility in describing binary formats, it maps each format to a fixed C struct type, where NARCISSUS is compatible with arbitrary Coq types.

In contrast to these other frameworks, NARCISSUS provides a strong, machine-checked guarantee that every synthesized decoder properly inverts the data format and detects any malformed inputs. Using Coq's logic as the specification language allows NARCISSUS to have a small trusted core of built-in features while still being both highly extensible and sound.

Nondeterminism Monad The nondeterminism monad [3] has a long history as a vehicle for deriving correct-by-construction programs by iterative refinement [6]. Recently frameworks supporting this approach to program derivation have been developed for both the Coq [5] and Isabelle [11] proof assistants, with both systems providing machine-checked refinement proofs for derived software. While NARCISSUS does support the derivation of correct encoders via refinement of the nondeterministic function specifying a format, its approach to decoder synthesis does not. Rather, NARCISSUS builds the inversion of this relation specified by a format, ensuring that a synthesized decoder is sound with respect to *every possible* encoder function.

8. Future Work and Conclusion

We can imagine extending the library of NARCISSUS with several additional components. One natural set of additions would be compression schemes, allowing users to derive decoders for both archive formats like ZIP and image formats like PNG. The nondeterminism monad used in NARCISSUS specifications also provides a natural way to state nondetermistic perturbations to encoded data, allowing us potentially to characterize the resiliency of decoders to transmission errors; extending the library with components for error-correcting codes is another promising direction. Another future direction we plan to explore is integrating NARCISSUS-derived decoders and encoders into the development of certified Internet servers, like DNS and SMTP servers, integrating the NARCISSUS-derived proofs to build strong end-to-end functional correctness guarantees. We would also like to improve the performance of encoders and decoders by pushing the derivation process further, to low-level imperative rather than just pure-functional code.

In conclusion, we have presented NARCISSUS, a framework for specifying and synthesizing correct-by-construction decoders from formats. This framework provides fine-grained control over the shape of encoded data, is extensible with user-defined formats and implementation strategies, has a small set of core definitions augmented with a library of common formats, and produces machine-checked proofs of soundness for derived decoders. We evaluated the expressiveness of NARCISSUS by deriving decoders for several standard networking protocols and evaluated the performance of the synthesized code by incorporating it into the networking stack of MirageOS.

References

[1] Apache Software Foundation. Apache Avro 1.8.0 documentation, 2016. [Accessed May 04, 2016].

[2] Godmar Back. Datascript - a specification and scripting language for binary data. In *Proceedings of the 1st ACM SIGPLAN/SIGSOFT Conference on Generative Programming and Component Engineering*, GPCE '02, pages 66–77, London, UK, UK, 2002. Springer-Verlag.

[3] Ralph-JR Back. A calculus of refinements for program derivations. *Acta Informatica*, 25(6):593–624, 1988.

[4] Julian Bangert and Nickolai Zeldovich. Nail: A Practical Tool for Parsing and Generating Data Formats. In *OSDI*, pages 615–628. USENIX Association, 2014.

[5] Benjamin Delaware, Clément Pit-Claudel, Jason Gross, and Adam Chlipala. Fiat: Deductive synthesis of abstract data types in a proof assistant. In *Proc. POPL*, 2015.

[6] Edsger W. Dijkstra. A constructive approach to the problem of program correctness. Circulated privately, August 1967.

[7] Olivier Dubuisson. *ASN. 1: communication between heterogeneous systems*. Morgan Kaufmann, 2001.

[8] Kathleen Fisher and Robert Gruber. Pads: a domain-specific language for processing ad hoc data. *ACM Sigplan Notices*, 40(6):295–304, 2005.

[9] Kathleen Fisher, Yitzhak Mandelbaum, and David Walker. The next 700 data description languages. *SIGPLAN Not.*, 41(1):2–15, January 2006.

[10] Stephen C. Johnson. Yacc: Yet another compiler-compiler. Technical report, 1979.

[11] Peter Lammich and Thomas Tuerk. Applying data refinement for monadic programs to hopcroft's algorithm. In Lennart Beringer and Amy Felty, editors, *Interactive Theorem Proving*, volume 7406 of *Lecture Notes in Computer Science*, pages 166–182. Springer Berlin Heidelberg, 2012.

[12] Anil Madhavapeddy, Richard Mortier, Charalampos Rotsos, David Scott, Balraj Singh, Thomas Gazagnaire, Steven Smith, Steven Hand, and Jon Crowcroft. Unikernels: Library operating systems for the cloud. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '13, pages 461–472, New York, NY, USA, 2013. ACM.

[13] Peter J. McCann and Satish Chandra. Packet types: Abstract specification of network protocol messages. In *Proceedings of the Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, SIGCOMM '00, pages 321–333, New York, NY, USA, 2000. ACM.

[14] P. Mockapetris. Domain names - implementation and specification. RFC 1035, November 1987.

[15] Ruoming Pang, Vern Paxson, Robin Sommer, and Larry Peterson. binpac: A yacc for writing application protocol parsers. In *Proceedings of the 6th ACM SIGCOMM conference on Internet measurement*, pages 289–300. ACM, 2006.

[16] T. J. Parr and R. W. Quong. Antlr: A predicated-ll(k) parser generator. *Softw. Pract. Exper.*, 25(7):789–810, July 1995.

[17] Jon Postel. Internet protocol. RFC 791, September 1981.

[18] Jon Postel. Transmission control protocol. RFC 793, September 1981.

[19] Raj Srinivasan. Xdr: External data representation standard. Technical report, 1995.

[20] The Coq Development Team. The Coq proof assistant reference manual, version 8.4. 2012.

[21] Kenton Varda. Protocol buffers. https://developers.google.com/protocol-buffers/.