Narcissus: Correct-by-Construction Derivation of Decoders and Encoders from Binary Formats

BENJAMIN DELAWARE, Purdue University, USA
SORAWIT SURIYAKARN, Band Protocol, Thailand
CLÉMENT PIT-CLAUDEL, MIT CSAIL, USA
QIANCHUAN YE, Purdue University, USA
ADAM CHLIPALA, MIT CSAIL, USA

It is a neat result from functional programming that libraries of *parser combinators* can support rapid construction of decoders for quite a range of formats. With a little more work, the same combinator program can denote both a decoder and an encoder. Unfortunately, the real world is full of gnarly formats, as with the packet formats that make up the standard Internet protocol stack. Most past parser-combinator approaches cannot handle these formats, and the few exceptions require redundancy – one part of the natural grammar needs to be hand-translated into hints in multiple parts of a parser program. We show how to recover very natural and nonredundant format specifications, covering all popular network packet formats and generating both decoders and encoders automatically. The catch is that we use the Coq proof assistant to derive both kinds of artifacts using tactics, automatically, in a way that guarantees that they form inverses of each other. We used our approach to reimplement packet processing for a full Internet protocol stack, inserting our replacement into the OCaml-based MirageOS unikernel, resulting in minimal performance degradation.

CCS Concepts: • **Software and its engineering → Domain specific languages**.

Additional Key Words and Phrases: Deductive Synthesis, Parser Combinators, Program Synthesis, Serialization and Deserialization

ACM Reference Format:
Benjamin Delaware, Sorawit Suriyakarn, Clément Pit-Claudel, Qianchuan Ye, and Adam Chlipala. 2019. Narcissus: Correct-by-Construction Derivation of Decoders and Encoders from Binary Formats. *Proc. ACM Program. Lang.* 3, ICFP, Article 82 (August 2019), 29 pages. https://doi.org/10.1145/3341686

1 INTRODUCTION

Decoders and encoders are vital components of any software that communicates with the outside world. Accordingly, functions that process untrusted data represent a key attack surface for malicious actors. Failures to produce or interpret standard formats routinely result in data loss, privacy violations, and service outages in deployed systems [CVE 2013a,b, 2015]. In the case of formally verified systems, bugs in encoder and decoder functions that live in the unverified, trusted code have been shown to invalidate the entire assurance case [Fonseca et al. 2017]. There are no shortage of code-generation frameworks [Apache Software Foundation 2016; Back 2002; Bangert and Zeldovich 2014; Dubuisson 2001; Fisher and Gruber 2005; Fisher et al. 2006; Johnson 1979;

Authors' addresses: Benjamin Delaware, bendy@purdue.edu, Purdue University, West Lafayette, Indiana, USA; Sorawit Suriyakarn, swit@bandprotocol.com, Band Protocol, Bangkok, Thailand; Clément Pit-Claudel, cpitcla@csail.mit.edu, MIT CSAIL, Cambridge, Massachusetts, USA; Qianchuan Ye, ye202@purdue.edu, Purdue University, West Lafayette, Indiana, USA; Adam Chlipala, adamc@csail.mit, MIT CSAIL, Cambridge, Massachusetts, USA.

2475-1421/2019/8-ART82
https://doi.org/10.1145/3341686

McCann and Chandra 2000; Pang et al. 2006; Parr and Quong 1995; Srinivasan 1995; Varda 2008] that aim to reduce opportunities for user error in writing encoders and decoders, but these systems are quite tricky to get right and have themselves been sources of serious security bugs [CVE 2016].

Combinator libraries are an alternative approach to the rapid development of parsers which has proven particularly popular in the functional-programming community [Leijen and Meijer 2001]. This approach has been adapted to generate both parsers and pretty printers from single programs [Kennedy 2004; Rendel and Ostermann 2010]. Unfortunately, unverified combinator libraries suffer from the same potential for bugs as code-generation frameworks, with the additional possibility for users to introduce errors when extending the library with new combinators. This paper presents Narcissus, a combinator-style framework for the Coq proof assistant that eliminates the possibility of such bugs, enabling the derivation of encoders and decoders that are correct by construction. Each derived encoder and decoder is backed by a machine-checked functional-correctness proof, and Narcissus leverages Coq's proof automation to help automate both the construction of encoders and decoders and their correctness proofs. Key to our approach is how it propagates information through a derivation, in order to generate decoders and encoders for the sorts of non-context-free languages that often appear in standard networking protocols.

We begin by introducing the key features of Narcissus with a series of increasingly complex examples, leading to a hypothetical format of packets sent by a temperature sensor to a smart home controller. In order to build up the reader's intuition, we deliberately delay a discussion of the full details of our approach until Section 2. The code accompanying our tour is included in the Narcissus repository[1] in the `src/Narcissus/Examples/README.v` file, and can be run in Coq version 8.7.2 [The Coq Development Team 2018].

1.1 A Tour of Narcissus

Getting started. Our first format is extremely simple:

1.1: User input

```
Record sensor_msg :=
  { stationID: word 8; data: word 16 }.

Let format :=
     format_word ∘ stationID
  ++ format_word ∘ data.

Let invariant (msg: sensor_msg) := T.

Let enc_dec: EncDecPair format invariant :=
  ltac:(derive_encoder_decoder_pair).
```

All user input is contained in box 1.1. A `sensor_msg` value is a record with two fields; the Coq `Record` command defines accessor functions for these two fields. Our `format` specifies how instances of this record are serialized using two format *combinators*: `format_word` is a Narcissus primitive that serializes a word bit-by-bit, and ++ is a sequencing operator (write this, then that). Our `invariant` specifies additional constraints on well-formed packets, although this example does not have any. The `derive_encoder_decoder_pair` tactic is part of the framework and automatically generates encoder and decoder functions, as well as proofs that they are correct:

1.2: Encoder

```
Let encode := encoder_impl enc_dec.
  stationID ⋙ SetCurrentByte ≫
  data ⋙ (high_bits 8 ⋙ SetCurrentByte ≫
          low_bits 8 ⋙ SetCurrentByte)
```

[1] https://github.com/mit-plv/fiat/tree/narcissus-icfp2019

1.3: Decoder
```
Let decode := decoder_impl enc_dec.
  b ← GetCurrentByte;
  b₀ ← GetCurrentByte;
  b' ← GetCurrentByte;
  w ← ret b₀ ·b';
  ret {| stationID := b; data := w |}
```

Boxes 1.2 and 1.3 show the generated code. In box 1.2, the encoder operates on a data value and a fixed-size byte buffer (both implicit) and returns the encoded packet, or None if it did not fit in the supplied buffer. In box 1.3, the decoder takes a buffer and returns a packet, or None if the buffer did not contain a valid encoding. Both generated programs live in stateful error monads (← and ≫ are the usual binding and sequencing operators), offering primitives to read and write a single byte (GetCurrentByte, SetCurrentByte). The encoder uses the ⋙ reverse-composition operator (a ⋙ b ≡ b ∘ a) to pass record fields to SetCurrentByte. Since data is 16 bits long, the encoder also uses high_bits and low_bits to extract the first and last 8 bits, and the decoder reassembles them using the · concatenation operator: this byte-alignment transformation is part of the derive_encoder_decoder_pair logic.

Underspecification. We now consider a twist: to align data on a 16-bit boundary, we introduce 8 bits of padding after stationID; these bits will be reserved for future use:

2.1: User input
```
Record sensor_msg :=
  { stationID: word 8; data: word 16 }.
```
```
Let format :=
     format_word ∘ stationID
  ++ format_unused_word 8
  ++ format_word ∘ data.
```
```
Let invariant (msg: sensor_msg) := T.
```
```
Let enc_dec: EncDecPair format invariant :=
  ltac:(derive_encoder_decoder_pair).
```

2.2: Encoder
```
Let encode := encoder_impl enc_dec.
  stationID⋙SetCurrentByte ≫
  const 0b00000000⋙SetCurrentByte ≫
  data⋙(high_bits 8⋙SetCurrentByte ≫
          low_bits 8⋙SetCurrentByte)
```

2.3: Decoder
```
Let decode := decoder_impl enc_dec.
  b ← GetCurrentByte;
  _ ← GetCurrentByte;
  b₁ ← GetCurrentByte;
  b' ← GetCurrentByte;
  w ← ret b₁ ·b';
  ret {| stationID := b; data := w |}
```

These eight underspecified bits introduce an asymmetry: the encoder always writes 0x00, but the decoder accepts any value. The lax behavior is crucial because the format_unused_word specification allows conforming encoders to output *any* 8-bit value; as a result, a correct decoder for this format needs to accept *all* 8-bit values. In that sense, the encoder and decoder that Narcissus generates are not strict inverses of each other: the encoder is one among many functions permitted by the formatting specification, and the decoder is the inverse of the *entire family* described by the format, accepting packets serialized by *any* conforming encoder.

Constants and enums. Our next enhancements are to add a version number to our format and to tag each measurement with a `kind`, `"TEMP"` or `"HUMIDITY"`. To save space, we allocate 2 bits for the tag and 14 bits for the measurement:

3.1: User input

```
Let kind := EnumType ["TEMP"; "HUMIDITY"].

Record sensor_msg :=
  { stationID: word 8; data: (kind * word 14) }.

Let format :=
     format_word ∘ stationID
  ++ format_unused_word 8
  ++ format_const 0b0000011111100010
  ++ format_enum [0b00; 0b01] ∘ fst ∘ data
  ++ format_word ∘ snd ∘ data.

Let invariant (msg: sensor_msg) := ⊤.

Let enc_dec: EncDecPair format invariant :=
  ltac:(derive_encoder_decoder_pair).
```

3.2: Encoder

```
Let encode := encoder_impl enc_dec.
  stationID ⋙ SetCurrentByte ≫
  const 0b00000000 ⋙ SetCurrentByte ≫
  const 0b00000111 ⋙ SetCurrentByte ≫
  const 0b11100010 ⋙ SetCurrentByte ≫
  (λ r ⇒ (Vector.nth [0b00; 0b01] ∘ fst ∘ data) r · (snd ∘ data) r)
    ⋙ (high_bits 8 ⋙ SetCurrentByte ≫
       low_bits 8 ⋙ SetCurrentByte)
```

3.3: Decoder

```
Let decode := decoder_impl enc_dec.
  b ← GetCurrentByte;
  _ ← GetCurrentByte;
  b1 ← GetCurrentByte;
  b' ← GetCurrentByte;
  w ← ret b1 ·b';
  (if weq w 0b0000011111100010 then
     b2 ← GetCurrentByte;
     b'0 ← GetCurrentByte;
     w0 ← ret b2 ·b'0 ;
     match index (high_bits 2 w0) [0b00; 0b01] with
     | Some a' → ret {| stationID := b; data := (a', low_bits 14 w0) |}
     | None → fail end
   else fail)
```

The use of `format_const` in the specification forces conforming encoders to write out the value `0x7e2`, encoded over 16 bits. Any input that does not contain that exact sequence is malformed, which the generated decoder signals by throwing an exception. Narcissus also checks more subtle dependencies between subformats: for example, if a format were to encode the same value twice, the generated decoder will decode both values *and* check that they agree— the packet must be malformed if not. The argument passed to `format_enum` specifies which bit patterns to use to represent each tag (`0b00` for `"TEMP"`, `0b01` for `"HUMIDITY"`), and the decoder uses this mapping to reconstruct the appropriate enum member.

Lists and dependencies. Our penultimate example illustrates data dependencies and input restrictions. To do so, we replace our single data point with a list of measurements (for conciseness, we remove tags and use 16-bit words):

4.1: User input

```
Record sensor_msg :=
  { stationID: word 8; data: list (word 16) }.
Let format :=
     format_word ∘ stationID
  ++ format_unused_word 8
  ++ format_const 0b0000011111100010
  ++ format_list format_word ∘ data.
Let invariant (msg: sensor_msg) := ⊤.
Let enc_dec: EncDecPair format invariant :=
  ltac:(derive_encoder_decoder_pair).
```

The `format_list` combinator encodes a value by simply applying its argument combinator in sequence to each element of a list. We start a derivation as before, but we quickly run into an issue: the derivation fails, leaving multiple Coq goals unsolved. The first of these shows the portion of the format where `derive_encoder_decoder_pair` got stuck:

```
CorrectDecoder (format_list format_word ∘ data ++ ...) ?d
```

This goal indicates that the derivation got stuck trying to find a decoder for the list of measurements. Using an additional tactic takes us to the last unsolvable goal, which is equivalent to the following:

```
∀ msg: sensor_msg, msg.(stationID) = sid → length msg.(data) = ?Goal
```

The issue is that the built-in list decoder is only applicable if the number of elements to decode is known, but our format never encodes the length of the `data` list. An attempt to fix this problem by including the length of `data` does not completely solve the problem, unfortunately (`format_nat 8 ∘ length` specifies that the length should be truncated to 8 bits and written out):

5.1: User input

```
Let format :=
      format_word ∘ stationID
   ++ format_nat 8 ∘ length ∘ data
   ++ format_const 0b0000011111100010
   ++ format_list format_word ∘ data.
```

Indeed, the decoder derivation now gets stuck on the following goal:

```
CorrectDecoder (format_nat 8 ∘ length ∘ data ++ ...) ?d
```

Our debugging tactic now produces the following goal:

```
∀ msg: sensor_msg, invariant msg ∧ msg.(stationID) = proj → length msg.(data) < 2^8
```

The problem is that, since we encode the list's length in 8 bits, the round-trip property that NARCISSUS enforces only holds if the list has fewer than 2^8 elements: larger lists have their lengths truncated, and it becomes impossible for the decoder to know for certain how many elements it should decode. What we need is an input restriction: a predicate defining which messages we may encode. To this end, we make one final adjustment:

6.1: User input

```
Let invariant (msg: sensor_msg) :=
  length (msg.(data)) < 2^8.
```

6.2: Encoder

```
Let encode := encoder_impl enc_dec.
  stationID ⋙ SetCurrentByte ≫
  data ⋙ Datatypes.length ⋙ natToWord 8 ⋙ SetCurrentByte ≫
  const 0b00000111 ⋙ SetCurrentByte ≫
  const 0b11100010 ⋙ SetCurrentByte ≫
  data ⋙ AlignedEncodeList (λ _ ⇒ high_bits 8 ⋙ SetCurrentByte ≫
                                  low_bits 8 ⋙ SetCurrentByte)
```

6.3: Decoder

```
Let decode := decoder_impl enc_dec.
  b ← GetCurrentByte;
  b0 ← GetCurrentByte;
  b1 ← GetCurrentByte;
  b' ← GetCurrentByte;
  w ← ret b1 ·b';
  (if weq w 0b0000011111100010 then
    l ← ListAlignedDecodeM (λ _ ⇒
            w0 ← GetCurrentByte;
            w' ← GetCurrentByte;
            ret w0 ·w') (wordToNat b0);
    ret {| stationID := b; data := l |}
   else fail)
```

User-defined formats. Our final example illustrates a key benefit of the combinator-based approach: integration of user-defined formats and decoders. The advantage here is that Narcissus does not sacrifice correctness for extensibility: every derived function must be correct. This example uses a custom type for sensor readings, `reading`. To integrate this type into Narcissus, the user also supplies the format specification for this type, corresponding encoders and decoders and proofs of their correctness, and a set of tactics explaining how to integrate these pieces into a derivation. Section 5 provides the complete details on these ingredients, but for now we note that the `format_reading` specification is nothing more exotic than a nondeterministic function in the style of the Fiat framework [Delaware et al. 2015], and that the two lemmas are normal interactive Coq proofs.

7.1: User input

```
Inductive reading :=
  | Temperature (_ : word 14)
  | Humidity (_ : word 14).
```

```
Let fmt_reading m s := match m with
  | Temperature t ⇒ ret (serialize (0b00 · t) s)
  | Humidity h ⇒ ret (serialize (0b01 · h) s) end.
```

```
Let enc_reading := ....
Lemma enc_readingCorrect: CorrectEncoder fmt_reading enc_reading. ...
```

```
Let dec_reading := ....
Lemma dec_readingCorrect: CorrectDecoder fmt_reading dec_reading. ...
```

```
Ltac new_encoder_rules ::= apply enc_readingCorrect.
Ltac new_decoder_rules ::= apply dec_readingCorrect.
```

```
Record sensor_msg :=
  { stationID: word 8; data: list reading }.
```

```
Let format :=
     format_word ∘ stationID
  ++ format_nat 8 ∘ length ∘ data
  ++ format_list fmt_reading ∘ data.
```

```
Let invariant (msg: sensor_msg) :=
  length (msg.(data)) < 2^8.
```

```
Let enc_dec: EncDecPair format invariant :=
  ltac:(derive_encoder_decoder_pair).
```

Wrapping up. In Narcissus, users specify formats using a library of combinators and use tactics to derive correct-by-construction encoder and decoder functions automatically from these specifications. Formats may be underspecified, in that a particular source value may be serialized in different ways, but decoders are guaranteed to interpret all of them correctly. Formats may induce dependencies between subformats; the derivation procedure is responsible for tracking these dependencies when generating a decoder. Finally, a user can extend Narcissus with new formats and datatypes by providing a few simple ingredients; extensions are guaranteed not to compromise the correctness of derived functions.

To more precisely summarize this paper's contributions:

- We develop specifications of correctness for encoders and decoders, keyed on a common nondeterministic format.
- We show how encoder and decoder combinators can be verified modularly, even when their correctness depends on the contexts in which they are used, in a way that enables compositional verification of composite encoders and decoders built from combinators.
- We show how to derive correct-by-construction encoders and decoders via interactive proof search using libraries of verified combinators, and we provide proof tactics to automate the process in a way that supports extension without compromising soundness.
- We demonstrate how a two-phase approach that iteratively refines bit-level specifications into byte-level functions can enable both clean specifications and efficient implementations.

We demonstrate the applicability of Narcissus by deriving packet processors for a full Internet protocol stack, which required the addition of a checksum combinator. Inserting our replacement into the OCaml-based MirageOS unikernel results in minimal performance degradation.

We pause briefly here to contrast the design choices made by Narcissus with other approaches to serializing and deserializing data, with a more detailed discussion deferred to Section 7. There has been a particular focus on formally verifying parsers and pretty printers for programming-language ASTs as parts of compiler frontends [Barthwal and Norrish 2009; Jourdan et al. 2012; Koprowski and Binsztok 2011] or to carry out binary analysis [Morrisett et al. 2012; Tan and Morrisett 2018]. One of the target applications of Narcissus is formally verified distributed systems, and the restriction to context-free languages (as found in those tools) disallows many of the standard network formats such applications require.

Narcissus has a similar motivation to bidirectional programming languages [Bohannon et al. 2008; Mu et al. 2004] in which programs can be run "in reverse" to map target values to the source values that produced them. The bidirectional programming language Boomerang adopts a similar combinator-based approach to deriving transformations between target and source values. Invertibility is an intrinsic property of a bidirectional language, so new combinators require extensions to its metatheory. In contrast, proofs of correctness are built alongside functions in Narcissus, allowing the framework to be augmented safely by including a proof justifying a new implementation strategy as part of an extension.

We now present the complete details of Narcissus in a more bottom-up fashion, before discussing our evaluation and a more detailed comparison with related work. For readability, some of the names below differ from those used in Narcissus's source code; we provide a correspondence table in the `README.md` file included in the distribution.

2 NARCISSUS, FORMALLY

We begin our ground-up explanation of Narcissus with the definition of the *formats* that capture relationships between structured source values and their serialized representations. The signature of a format from source type S to target type T with state type Σ is defined by a type alias:

$$\text{FormatM S T } \Sigma := \text{Set of } (S \times \Sigma \times T \times \Sigma)$$

That is, a format is a quaternary relation on source values, initial states, target values, and final states. Including states allows us to specify a rich set of formats, including DNS packets (Section 2.1). As hinted at by the M suffix, FormatM can be interpreted as a combination of the nondeterminism and state monads.

The format combinators showcased in Section 1.1 have straightforward definitions using standard set operations. The ++ combinator sequences its subformats using a monoid operation · provided by its target type.

$$\begin{aligned}(s, \sigma, t, \sigma') \in \text{format}_1 +\!\!+ \text{format}_2 \quad \equiv \\ \exists\, t_1\ t_2\ \sigma''.\ (s, \sigma, t_1, \sigma'') \in \text{format}_1 \wedge (s, \sigma'', t_2, \sigma') \in \text{format}_2 \wedge t = t_1 \cdot t_2\end{aligned}$$

The function-composition combinator ∘ is actually defined via the more elementary ⊙ combinator. This combinator uses a relation, R, to format a projection of the source domain:

$$(s, \sigma, t, \sigma') \in \text{format} \odot R \quad \equiv \quad \exists\, s'.\ (s', \sigma, t, \sigma') \in \text{format} \wedge R\ s\ s'$$

Underspecified formats can be built by combining ⊙ with a choice operator, as in the format for unused words:

$$\text{format\_unused\_word} \quad \equiv \quad \text{format\_word} \odot \{(\_, \_) \mid \top\}$$

In addition to ∘, ⊙ is used to define the ∩ combinator that restricts the source values included in a format:

$$\begin{aligned}\text{format} \circ f \quad &\equiv \quad \text{format} \odot \{(s, s') \mid s' = f\ s\} \\ P \cap \text{format} \quad &\equiv \quad \text{format} \odot \{(s, s') \mid P\ s \wedge s = s'\}\end{aligned}$$

Another helpful higher-order combinator is Union, which is useful for defining formats with variant encodings, e.g. Ethernet frames:

$$(s, \sigma, t, \sigma') \in \text{format}_1 \cup \text{format}_2 \quad \equiv \quad (s, \sigma, t, \sigma') \in \text{format}_1 \vee (s, \sigma, t, \sigma') \in \text{format}_2$$

While not very useful for user-defined formats, the empty format ϵ is helpful in the specifications of encoder and decoder combinators:

$$(s, \sigma, t, \sigma') \in \epsilon \quad \equiv \quad t = \iota \wedge \sigma = \sigma'$$

For clarity, we have presented these combinators in point-free style, but the monad formed by FormatM also admits definitions in a pointed style, which can be more convenient for defining base formats like format_reading. In addition to the standard **return** and **bind** (_ ← _; _) operators, this monad includes a set-comprehension operator { x | P x }, which specifies a set via a defining property P on possible return values. The three operators have straightforward interpretations as sets [Delaware et al. 2015]:

$$\begin{aligned}e \in \textbf{return}\ v \quad &\equiv \quad e = v \\ e \in \{x \mid P\ x\} \quad &\equiv \quad P\ e \\ e \in x \leftarrow y;\ k\ x \quad &\equiv \quad \exists\, e'.\ e' \in y \wedge e \in k\ e'\end{aligned}$$

As an example, we can specify the set of all possible locations of a period in a string s as:

$$s_1 \leftarrow \{\, s_1 : \text{String} \mid \exists\, s_2.\ s = s_1 +\!\!+ \text{"."} +\!\!+ s_2 \,\};\ \textbf{return}\ (\text{length}\ s_1)$$

In addition to enabling users to define their own formats in a familiar monadic style, the nondeterminism monad integrates nicely with Coq's rewriting machinery when deriving correct encoders.

Format	Higher-order?	Format	Higher-order?
Boolean	no	Fixed-Length Word	no
Peano Number	no	Unspecified BitString	no
Variable-Length List	yes	Fixed-Length List	yes
Variable-Length String	no	Fixed-Length String	no
Option Type	yes	ASCII Character	no
Enumerated Type	no	Variant Type	yes

Fig. 1. Formats for base types included in Narcissus.

Narcissus includes a library of formats for the standard types listed in Figure 1, most of which have pointed definitions.

We have left the definition of the target type of our formats underspecified until now. Either *bitstrings* or *bytestrings*, i.e. lists of bits or bytes, would be natural choices, each with its own advantages and disadvantages. Bitstrings have a conceptually cleaner interface that allows users to avoid byte-alignment considerations. As an example, the format in box 3.1 can simply sequence 14-bit and 2-bit words, while a byte-aligned specification would require splitting the first word into 8- and 6-bit words, combining the latter with the 2-bit word. Unfortunately, true bitstrings are quite removed from the byte buffers used in real systems, requiring bit-shifting to enqueue bits one at a time. Narcissus attempts to split the difference by using the bitstring abstract data type presented in Figure 2 for the target type of formats. Clients can treat a BitString value as a bitstring equipped with operations governed by algebraic laws for monoids and queues, while its actual representation type is closer to that of a bytestring. Section 4 details how our derivation procedures optimize away uses of this interface in order to produce better-performing implementations.

```
ADT BitString {
Definition ι : BitString;
Definition · : BitString → BitString → BitString;
Definition snoc : BitString → 𝔹 → BitString;
Definition unfold : BitString → option (𝔹 × BitString);

Axiom left_id : ∀ s1, ι · s1 = s1;
Axiom right_id : ∀ s1, s1 · ι = s1;
Axiom assoc : ∀ s1 s2 s3, s1 · (s2 · s3) = (s1 · s2) · s3;

Axiom unfold_app : ∀ b s1 s2 s3, unfold s1 = Some (b, s2)
        → unfold (s1 · s3) = Some (b, s2 · s3);
Axiom snoc_app : ∀ b s1 s2,
        snoc b (s1 · s2) = s1 · (snoc b s2);
Axiom unfd_snoc : ∀ b, unfold (snoc b ι) = Some (b, ι);
Axiom unfd_id : unfold ι = None;
Axiom unfd_inj : ∀ s1 s2, unfold s1 = unfold s2 → s1 = s2}
```

Fig. 2. The BitString interface, with some length operations elided.

2.1 Specifying Encoders and Decoders

These relational formats are not particularly useful by themselves— even checking whether a format permits specific source and target values may be undecidable. Instead we use them to specify the correctness of *both* encoders and decoders. So far, we have seen examples of relational formats that permit one or many target representations of a particular source value, but in their full generality, there might not be *any* valid encodings of some source value. As an extreme example, consider the following use of the ∩ combinator to define an empty relation: {s | False} ∩ format_word. More realistically, a format for domain names must disallow strings with runs of ".", e.g. "www.foo..bar.com". To account for formats that exclude some source values, Narcissus encoders are partial functions from source to target values: EncodeM S T Σ := S → Σ → Option (T × Σ). At a high level, a format describes a family of permissible encoders, where each must commit to a single target representation for each source value in the relation. More formally:

Definition 2.1 (Encoder Correctness). A correct encoder for a format format : FormatM S T Σ is a partial function, encode : EncodeM S T Σ, that only produces encodings of source values included in the format and signals an error on source values not included in the format:

$$\begin{aligned} &\forall s\ \sigma\ t\ \sigma'.\ \text{encode}\ s\ \sigma = \text{Some}\ (t, \sigma') \rightarrow && (s, \sigma, t, \sigma') \in \text{format} \\ \wedge\forall s\ \sigma.\quad &\text{encode}\ s\ \sigma = \bot && \rightarrow \forall t\ \sigma'.\ (s, \sigma, t, \sigma') \notin \text{format}. \end{aligned}$$

In other words, a valid encoder *refines* a format; we henceforth use the notation format ⊇ encode to denote that encode is a correct encoder for format.

Before stating the corresponding correctness definitions for decoders, consider the high-level properties a correct decoder should satisfy, ignoring for now the question of state. Clearly, it must be a *sound* left inverse of the format relation. That is, it should map every element t in the image of s in format back to s: $\forall s\ t.\ (s, t) \in \text{format} \rightarrow \text{decode}\ t = s$. Less clear is how much conformance checking a "correct" decoder should perform on target values that fall outside the image of format: should decode fail on such inputs, or should its behavior be unconstrained in these cases? If these decoders are being integrated into other formally verified systems that process decoded data further, it is desirable to provide the strongest assurance about the integrity of decoded data to downstream functions. On the other hand, there are valid reasons for looser standards, like sacrificing strict format validation for efficiency, e.g. by not verifying checksums. For now, we require correct decoders to flag strictly *all* malformed target values by signaling errors when applied to target values not included in the relation: $\forall s\ t,\ \text{decode}\ t = \text{Some}\ s \rightarrow (s, t) \in \text{format}$. As we shall see later (Definition 3.1), our formulation will also support deriving decoders with more lenient validation policies. However, we note that there are compelling security reasons for the top-level decoder to enforce strict input validation, in order to cut off potential side channels via demonic choice between legal alternatives. (E.g., consider a decoder integrated within an e-mail server, which decodes malformed packets into the contents of other users' inboxes.)

Looking at the signature of decoders in Narcissus,

$$\text{DecodeM S T Σ} \coloneqq \text{T} \rightarrow \Sigma \rightarrow \text{Option (S} \times \Sigma)$$

we see that we need to adapt these notions of correctness to account for the state used by a decoder. Whereas an encoder is a refinement of a format, and thus used identical types of state, we do not force compatible decoders and formats to share state types. To see why, consider a simplified version of the format for DNS domain names [Mockapetris 1987], which keeps track of the locations of previously formatted domains via its state argument:

Let format_domain (d: domain) (σ: domain → word) ≔ format_name d ∪ format_word (σ d)

This example uses an optional compression strategy in which a domain name can either be serialized or replaced with a pointer to the location of a previously formatted occurrence. A decoder for domains should also keep track of this information, in order to decode pointers:

Let decode_domain (t : T) (σ : word → domain) ≔ ...

In order for decode_domain to be correct, its state needs to "agree" with the state used to format its input. We do not want to require that these states be equal, so that decoders can have the freedom to use different data structures than the format (for example, decode_domain could be implemented using a binary search tree sorted on words, while encode_domain could use a prefix trie on domain names). Our notion of decoder correctness captures agreement between different state types via a binary relation defining when format and decoder states are consistent. Hence our full notion of decoder correctness accounts for both state and erroneous target values.

Definition 2.2 (Decoder Correctness). A correct decoder for a format, format : FormatM S T Σ_E, and relation on states, $\approx$: Set of ($\Sigma_E \times \Sigma_D$), is a function, decode : DecodeM S T Σ_D, that, when applied to a valid target value and initial state, produces a source value and final state similar to a pair included in format, signaling an error otherwise:

$$\begin{pmatrix} \forall(\sigma_E\ \sigma_E' : \Sigma_E)\ (\sigma_D : \Sigma_D)\ (\mathsf{s} : \mathsf{S})\ (\mathsf{t} : \mathsf{T}). \\ (\mathsf{s}, \sigma_E, \mathsf{t}, \sigma_E')\ \in \mathsf{format} \wedge \sigma_E \approx \sigma_D \\ \rightarrow \exists\ \sigma_D'.\ \mathsf{decode\ t}\ \sigma_D = \mathsf{Some}\ (\mathsf{s}, \sigma_D') \\ \wedge\ \sigma_E' \approx \sigma_D' \end{pmatrix} \wedge \begin{pmatrix} \forall(\sigma_E : \Sigma_E)\ (\sigma_D\ \sigma_D' : \Sigma_D)\ (\mathsf{s} : \mathsf{S})\ (\mathsf{t} : \mathsf{T}). \\ \sigma_E \approx \sigma_D \wedge \mathsf{decode\ t}\ \sigma_D = \mathsf{Some}\ (\mathsf{s}, \sigma_D') \\ \rightarrow \exists\ \sigma_E'.\ (\mathsf{s}, \sigma_E, \mathsf{t}, \sigma_E')\ \in \mathsf{format} \\ \wedge\ \sigma_E' \approx \sigma_D' \end{pmatrix}$$

We denote that decode is a correct decoder for format under a similarity relation on states $\approx$ as format $\overset{\approx}{\rightleftharpoons}$ decode.

By definition, it is impossible to find a correct decoder for a non-injective (i.e., ambiguous) format. While decoders and encoders have independent specifications of correctness, using a common format provides a logical glue that connects the two. We can, in fact, prove the expected round-trip properties between a correct encoder and correct decoder for a common format:

Theorem 2.3 (Decode Inverts Encode). *Given a correct decoder* format $\overset{\approx}{\rightleftharpoons}$ decode *and correct encoder* format $\supseteq$ encode *for a common format* format, decode *is an inverse for* encode *when restricted to source values in the format:*

$$\forall s\ \sigma_E\ t\ \sigma_E'\ \sigma_D.\ \mathsf{encode}\ s\ \sigma_E = \mathsf{Some}(t, \sigma_E')\ \wedge\ \sigma_E \approx \sigma_D \rightarrow \exists \sigma_D'.\ \mathsf{decode}\ t\ \sigma_D = \mathsf{Some}(s, \sigma_D')$$

Theorem 2.4 (Encode Inverts Decode). *Given a correct decoder* format $\overset{\approx}{\rightleftharpoons}$ decode *and correct encoder* format $\supseteq$ encode *for a common format* format, encode *is defined for all decoded source values produced by* decode*:*

$$\forall s\ \sigma_D\ t\ \sigma_D'\ \sigma_E.\ \mathsf{decode}\ t\ \sigma_D = \mathsf{Some}(s, \sigma_D') \wedge \sigma_E \approx \sigma_D \rightarrow \exists t'\ \sigma_E'.\ \mathsf{encode}\ s\ \sigma_E = \mathsf{Some}(t', \sigma_E')$$

That encode is an inverse of decode for source values with unique encodings is a direct corollary of Theorem 2.4.

3 DERIVING ENCODERS AND DECODERS

Equipped with precise notions of correctness, we can now explain how to derive provably correct encoders and decoders from a format. These functions will be byte-aligned in a subsequent derivation step presented in Section 3.2. We begin with encoders, since they often have similar structure to their corresponding formats. Intuitively, such a derivation is simply the search for a pair of an encoder function encode and a proof term witnessing that it is correct with respect to a format: format $\supseteq$ encode. As an example, a proof that a function that returns an empty bytestring correctly implements the empty format can also be read as evidence that it is safe to choose this implementation when searching for an encoder for ϵ. In this light, lemmas like enc_readingCorrect proving that encoder combinators are correct can be interpreted as *derivation rules* for constructing such proof trees from goal formats.

Leveraging this intuition, we denote these correctness lemmas using standard inference-rule notation:

$$\textbf{Lemma}\ \mathsf{EncA}\ (\mathsf{h}_1 : \mathsf{H}_1)\ (\mathsf{h}_2 : \mathsf{H}_2) : \mathsf{CorrectEncoder\ A\ T}\ \Sigma\ \mathsf{format_A\ encode_A}. \quad \equiv \quad \frac{\mathsf{H}_1 \qquad \mathsf{H}_2}{\mathsf{format_A} \supseteq \mathsf{encode_A}}\ (\textsc{EncA})$$

Figure 3 presents the encoder combinators for the formats from Section 2 using this inference-rule style. EncUnion is an example of an encoder that commits to a particular target value– given a correct encoder for each format in the union, it relies on an index function n on source values to select a particular encoding strategy, with the second hypothesis of the rule ensuring that this index

$$\frac{\begin{array}{c}\text{format}_1 \supseteq \text{encode}_1 \qquad \text{format}_2 \supseteq \text{encode}_2 \\ \forall\ (s, \sigma, t_1, \sigma')\ \in \text{format}_1. \forall\ t_2\ \sigma''.\ (s, \sigma', t_2, \sigma'')\ \in \text{format}_2 \rightarrow \\ \exists\ t_1'\ \sigma_2\ t_2'\ \sigma_3.\ \text{encode}_1\ s\ \sigma = \text{Some}\ (t_1', \sigma_2) \wedge \text{encode}_2\ s\ \sigma_2 = \text{Some}\ (t_2', \sigma_3)\end{array}}{\text{format}_1 +\!\!+ \text{format}_2 \supseteq \lambda s.\ t_1 \leftarrow \text{encode}_1\ s;\ t_2 \leftarrow \text{encode}_2\ s;\ \textbf{return}\ (t_1 \cdot t_2)}\ (\textsc{EncSeq})$$

$$\frac{\text{format} \supseteq \text{encode}}{\text{format} \circ g \supseteq \text{encode} \circ g}\ (\textsc{EncComp}) \qquad \frac{\text{format} \supseteq \text{encode} \qquad \forall\ s.\ p\ s = \text{true} \leftrightarrow s\ \in P}{P \cap \text{format} \supseteq \lambda s.\ \textbf{if}\ p\ s\ \textbf{then}\ \text{encode}\ s\ \textbf{else}\ \text{fail}}\ (\textsc{EncRest})$$

$$\frac{}{\epsilon \supseteq \lambda s.\ \textbf{return}\ \iota}\ (\textsc{EncEmpty}) \qquad \frac{\begin{array}{c}\text{format}_1 \supseteq \text{encode}_1 \qquad \text{format}_2 \supseteq \text{encode}_2 \\ \forall\ (s, \sigma, t, \sigma') \in \text{format}_i.\ n\ s = i\end{array}}{\text{format}_1 \cup \text{format}_2 \supseteq \lambda s.\ j \leftarrow n\ s;\ \text{encode}_j s}\ (\textsc{EncUnion})$$

Fig. 3. Correctness rules for encoder combinators.

function correctly picks a format that includes the source value. The rule for ++ establishes that a correct encoder for sequences can be built from encoders for its subformats. This rule features a wrinkle concerning state: in order to apply EncSeq correctly, if the format_1 produces *some* intermediate state that makes format_2 (and thus format_1 ++ format_2) nonempty, encode_1 must also produce a state on which encode_2 returns some value. The last hypothesis of EncSeq enforces that encode_1 does not "mislead" encode_2 in this manner.

By combining EncSeq and EncComp with the rules for base types, e.g. EncWord, we can iteratively derive correct encoders from a format; Figure 4 presents an example of such a derivation for one of the encoders from our introductory tour. Each step in the derivation corresponds to the encoder that results from applying EncSeq and an encoder-derivation rule for the topmost format. The proofs that none of the encoders pass on misleading states are elided. The hole □ at each step corresponds to the encoder that is built at the next step; recursively filling these in and simplifying the resulting expression with the monad laws yields the expected encoder for enc_data.

```
   format_word ∘ stationID
++ format_unused_word 8
++ format_nat 8 ∘ length ∘ data
++ format_list format_word ∘ data   ⊇   λ s ⇒ t1 ← encode_word s.stationID;
                                           t2 ← □;
                                           ret (t1 · t2)
```

⇑ EncSeq + EncUWord

```
   format_unused_word 8
++ format_nat 8 ∘ length ∘ data
++ format_list format_word ∘ data   ⊇   λ s ⇒ t2 ← encode_word 0b00000000;
                                           t3 ← □;
                                           ret (t2 · t3)
```

⇑ EncSeq + EncNat

```
   format_nat 8 ∘ length ∘ data
++ format_list format_word ∘ data   ⊇   λ s ⇒ t3 ← encode_nat (length s.data);
                                           t4 ← □;
                                           ret (t3 · t4)
```

⇑ EncSeq + EncList

```
   format_list format_word ∘ data   ⊇   λ s ⇒ ret (encode_list encode_word s.data)
```

Fig. 4. An example encoder derivation.

3.1 Decoders

Before defining similar correctness rules for decoder combinators, we pause to consider how they are used to build a top-level decoder. To concretize this discussion, consider how the signatures of the decoder components in the following top-level decoder correctness fact differ from the decoder built from them:

```
  format_word ∘ stationID                 id ← decode_word;
++ format_word ∘ data        ≈          d ← decode_word;
                                        ret { stationID := id; data := d }
```

Firstly, the two uses of `decode_word` naturally align with `format_word ∘ stationID` and `format_word ∘ data`. In contrast to top-level decoders, however, these components compute projections or *views* of the original sensor value, `stationID` and `data`, respectively. Secondly, the natural way to decode the source value resulting from sequencing these two formats with ++ is to have the first use of `decode_word` return any unconsumed portion of the target value via >>= for continued processing. We thus define the signature of a decoder combinator to be[2]:

$$\mathsf{DecodeCombM\ V\ T\ \Sigma} := \mathsf{T} \to \Sigma \to \mathsf{Option\ (V \times T \times \Sigma)}$$

Neither of these changes align with Definition 2.2, which required a correct decoder to recover the *full* source value by *completely* consuming a target value. To recover the desired top-level property, we must adapt our two correctness properties to account for these differences. To reconcile the discrepancy between the type of the source value used in the original format and the view computed by a combinator, we parameterize the correctness criteria over a relation between values of the two types and include an additional view format that relates view and target values.

Our next adaptation is to require that a combinator consume *precisely* the portion of the bitstring corresponding to its view, so that sequencing works correctly. To see why, consider a simple format for card suits which uses unit for the state type (we elide the trivial state values below):

$$\mathsf{format\_suit} \equiv \{(\clubsuit, 0\mathrm{b}11), (\diamondsuit, 0\mathrm{b}0), (\heartsuit, 0\mathrm{b}1), (\spadesuit, 0\mathrm{b}10)\}$$

To format a pair of cards, we could format each card in sequence: format_suit ∘ fst ++ format_suit ∘ snd. This format is clearly not injective, as it is not possible to distinguish between the encodings of (♣, ♢) and (♡, ♠). Absent additional information in the surrounding format, e.g. a Boolean flag identifying the color of the suit being decoded, it is impossible for a combinator for format_suit to soundly identify how much of the target to process. Since this format lacks such information, it is impossible to find a correct *top-level* decoder for it.

Definition 3.1 (Decoder-Combinator Soundness). A sound decoder combinator for a source format, $\mathsf{format}_s : \mathsf{FormatM\ S\ T\ }\Sigma_E$, relation on states, $\approx : \mathsf{Set\ of\ }(\Sigma_E \times \Sigma_D)$, relation between source and view values $\rho : \mathsf{S} \to \mathsf{V} \to \mathsf{Prop}$, and view format, $\mathsf{format}_\mathsf{v} : \mathsf{FormatM\ V\ T\ }\Sigma_E$ is a function, $\mathsf{decode} : \mathsf{DecodeCombM\ V\ T\ }\Sigma_D$, that when applied to a valid target value appended to an arbitrary bitstring and initial state, produces a view of the source value that agrees with the view format while consuming exactly the portion of the target value in the source format:

$$\begin{aligned}&\forall(\sigma_E\ \sigma_E' : \Sigma_E)\ (\sigma_D{:}\Sigma_D)\ (\mathsf{s} : \mathsf{S})\ (\mathsf{t}\ \mathsf{t}' : \mathsf{T}).\ (\mathsf{s}, \sigma_E, \mathsf{t}, \sigma_E')\ \in \mathsf{format}_s \wedge \sigma_E \approx \sigma_D\\ &\to \exists\ \mathsf{v}\ \sigma_D'.\ \mathsf{decode}\ (\mathsf{t} \cdot \mathsf{t}')\ \sigma_D = \mathsf{Some}\ (\mathsf{v}, \mathsf{t}', \sigma_D') \wedge\\ &\qquad \rho\ \mathsf{s}\ \mathsf{v} \wedge (\mathsf{v}, \sigma_D, \mathsf{t}, \sigma_D')\ \in \mathsf{format}_\mathsf{v} \wedge \sigma_E' \approx \sigma_D'\end{aligned}$$

Adapting the consistency criterion for decoder combinators is more straightforward. We require that decoder combinators ensure that any computed value agrees with the provided view-format relation *and* is a consistent view of any source values in the original format with the same encoding:

[2]We change the name of the first type parameter from S to V to emphasize that combinators produce views of encoded source values.

$$\frac{Q \cap \text{format} \;\overset{\sim}{\approx}\; \text{decode} \qquad \forall s\ v.\ \rho\ s\ v \wedge P\ s \to Q\ v}{P \cap \text{format} \circledcirc \rho \;\overset{\sim}{\underset{\rho}{\approx}}\; \text{decode} \;\sim Q \cap \text{format}} \quad \text{(DecCompose)}$$

$$\frac{Q \cap \text{format} \;\overset{\sim}{\approx}\; \text{decode} \qquad \forall l.\ P\ l \to |l| = n \wedge \forall a \in l.\ Q\ a}{P \cap \text{format\_list format} \;\overset{\sim}{\approx}\; \text{decode\_list decode n}} \quad \text{(DecList)}$$

$$\frac{\forall s'.\ P\ s' \to s = s' \qquad b = \text{true} \leftrightarrow P\ s}{P \cap \epsilon \;\overset{\sim}{\approx}\; \textbf{if}\ b\ \textbf{then return}\ s\ \textbf{else}\ \text{fail}} \quad \text{(DecDone)}$$

$$\frac{P \cap \text{format}_s \;\overset{\sim}{\underset{\rho}{\approx}}\; \text{decode} \;\sim \text{format}_v \qquad \forall s\ v.\ \rho\ s\ v \wedge P\ s \to \rho'\ s\ (f\ v) \qquad \forall v\ \sigma\ t\ \sigma'.\ (v,\sigma,t,\sigma') \in \text{format}_v \to (f\ v,\sigma,t,\sigma') \in \text{format}'_v}{P \cap \text{format}_s \;\overset{\sim}{\underset{\rho'}{\approx}}\; \text{fmap f decode} \;\sim \text{format}'_v} \quad \text{(DecInject)}$$

$$\frac{P \cap \text{format}_1 \;\overset{\sim}{\underset{\rho_1}{\approx}}\; \text{decode}_1 \;\sim \text{format}_{v1} \qquad \forall s\ v_1\ v_2.\ \rho_3\ s\ (v_1,v_2) \leftrightarrow \rho_1\ s\ v_1 \wedge \rho_2\ s\ v_2 \qquad \forall v_1.\ \{s \mid P\ s \wedge \rho_1\ s\ v_1\} \cap \text{format}_2 \;\overset{\sim}{\underset{\rho_2}{\approx}}\; \text{decode}_2\ v_1 \;\sim \{v_2 \mid (v_1,v_2) \in \text{format}_{v2}\}}{P \cap \text{format}_1 +\!\!+ \text{format}_2 \;\overset{\sim}{\underset{\rho_3}{\approx}}\; v_1 \leftarrow \text{decode}_1;\ v_2 \leftarrow \text{decode}_2\ v_1;\ \textbf{return}\ (v_1, v_2) \;\sim \text{format}_{v1} \circ \pi_1 +\!\!+ \text{format}_{v2}} \quad \text{(DecSeq)}$$

$$\frac{P \cap \text{format}_T \;\overset{\sim}{\underset{\rho}{\approx}}\; \text{decode}_T \sim \text{format}_{vT} \qquad P \cap \text{format}_E \;\overset{\sim}{\underset{\rho}{\approx}}\; \text{decode}_E \sim \text{format}_{vE} \qquad P \cap \text{subformat} \;\overset{\sim}{\underset{\rho_B}{\approx}}\; \text{decode}_B \sim \left\{ \begin{array}{l} (b, \sigma, t, \sigma') \mid \forall s\ t'\ \sigma''. \\ (s, \sigma, t{+}{+}t', \sigma'') \in \text{format}_T \to b = \text{true} \\ \wedge\ (s, \sigma, t{+}{+}t', \sigma'') \in \text{format}_E \to b = \text{false} \end{array} \right\} \qquad \text{subformat} \le (\text{format}_1 \cup \text{format}_2)}{P \cap (\text{format}_T \cup \text{format}_E) \;\overset{\sim}{\underset{\rho}{\approx}}\; \lambda t.\ b \leftarrow \text{decode\_B};\ \textbf{if}\ b\ \textbf{then}\ \text{decode}_T\ \textbf{else}\ \text{decode\_E} \;\sim \text{format}_{vT} \cup \text{format}_{vE}} \quad \text{(DecUnion)}$$

Fig. 5. Selected correctness rules for decoder combinators.

Definition 3.2 (Decoder-Combinator Consistency). A consistent decoder combinator for a source format, format_s : FormatM S T Σ_E, view format, format_v : FormatM V T Σ_E, relation on states, $\approx$: Set of ($\Sigma_E \times \Sigma_D$), relation between source and view values ρ : S → V → Prop, is a function, decode : DecodeCombM V T Σ_D, that is guaranteed to produce a view and unconsumed bitstring in a manner consistent with the view and source formats:

$$\begin{aligned} &\forall (\sigma_E{:}\Sigma_E)\ (\sigma_D\ \sigma_D{'}{:}\Sigma_D)\ (v : V)\ (t\ t' {:}\ T).\ \sigma_E \approx \sigma_D \wedge \text{decode}\ t\ \sigma_D = \text{Some}\ (v, t', \sigma_D') \\ &\to \exists\ t''\ \sigma_E'.\ t = t'' \cdot t' \ \wedge (v, \sigma_E, t'', \sigma_E') \in \text{format}_v \wedge \sigma_E' \approx \sigma_D' \\ &\qquad \wedge \forall\ s.\ (s, \sigma_E, t'', \sigma_E') \in \text{format}_s \to \rho\ s\ v \end{aligned}$$

A correct decoder combinator is one that is both sound with respect to the source format and view relation and that is consistent with its view format. We denote this property as $\text{format}_s \;\overset{\sim}{\underset{\rho}{\approx}}\; \text{decode} \sim \text{format}_v$. We can now precisely capture how decode_word correctly recovers a word representing the stationID of the source value from our previous example:

$$\text{format\_word} \circ \text{stationID} \;\overset{\sim}{\underset{\{(s,\,v)\ \mid\ \text{stationID}\ s = v\}}{\approx}}\; \text{decode\_word} \sim \text{format\_word}$$

Note that choosing the equality relation as ρ and the original format as the view format yields a property equivalent to our original left-inverse criterion, which we continue to denote as $\overset{\sim}{\approx}$.

Figure 5 presents a selection of some decoder-combinator-correctness facts included in Narcissus as inference rules. Note that all the rules are parameterized over a predicate P used to restrict the source format P ∩ format. This predicate is key to our approach to the modular verification of decoder combinators: each of these rules uses this predicate to thread information about previously decoded data through a proof of correctness for a composite decoder. Such information is necessary

for a decoder combinator whose correctness *depends* on context in which it is used. As a concrete example, the decode_list combinator in DecList only correctly decodes lists of length n, a restriction enforced by the second assumption of the rule. In isolation this rule only justifies using decode_list to decode fixed-lengths lists. When used as part of a larger format that also includes the length of the original list, however, it can be applied to lists of variable length, as in box 7.1. At first glance, the DecDone rule seems even more limited, as it only applies to a format with a single, unique source value. In the context of a larger format, however, this rule becomes much more powerful, particularly when employing these rules to *derive* a decoder from a format specification.

To see how, consider how the source predicate evolves during the decoder derivation presented in Figure 6. This derivation makes use of the DecSeqProj rule, which is a consequence of the more elementary rules in Figure 5.

$$\frac{\begin{array}{c} Q \cap \text{format}_1 \overset{\approx}{\leftrightarrows} \text{decode}_1 \qquad \forall s.\ P\ s \rightarrow Q\ (f\ s) \\ \forall v.\ \{s \mid P\ s \wedge f\ s = v\} \cap \text{format}_2 \overset{\approx}{\leftrightarrows} \text{decode}_2\ v \end{array}}{P \cap \text{format}_1 \circ f \mathbin{+\!\!+} \text{format}_2 \overset{\approx}{\leftrightarrows} \text{decode}_1 \gg\!= \text{decode}_2} \quad \text{(DecSeqProj)}$$

Each intermediate node in this derivation corresponds to the format in the last premise of DecSeqProj. Note how each step introduces a variable for the newly parsed data, and how an additional constraint is added to P relating the original source value to this value. When the derivation reaches format_list, this constraint witnesses that the number of elements in that list is known. Similarly, although the format is empty at the topmost leaf of the derivation tree, P includes enough constraints to uniquely recover the original source value from previously decoded values, and DecDone can be applied to finish the derivation. The first premise of DecDone ensures that the restriction on source values is sufficient to prove the existence of some constant s that is equal to the original source value. The second premise of DecDone ensures that a derived decoder is not overly permissive in the case that P is too restrictive. As previously noted, a format could encode the same view of a source value twice, and the consistency of the corresponding decoded values should be validated during decoding. Thus, the function b in this premise acts as a decision procedure that validates the consistency of all the projections of the original source gathered during decoding. While this example is straightforward, similar dependencies can be found in many existing binary formats, in the form of tags for sum types, version numbers, and checksum fields. In each case, the correctness of a combinator for a particular subformat depends on a previously decoded value.

The rules in Figure 5 are mostly used to derive more specific rules which are more useful in derivations of top-level decoders. The DecInj and DecUnion rules demonstrate some interesting features of our formulation of decoder-combinator correctness:

- DecInj proves how to transform a projected value safely, updating the view format to reflect that a transformation has been applied. While not particularly helpful during derivation of top-level decoders, this rule is useful for proving the correctness of derivation rules like DecSeqProj.
- DecUnion is similar to UnionEnc, with the key difference being that it requires a Boolean value, b, indicating which format produced the current bitstring. The combinator uses a decoder, decode_B, to compute such a Boolean and uses the view format to ensure the Boolean flag is correct. In addition, the proof of correctness for decode_B only requires that it consume some prefix of the target value produced by the source format, giving it the freedom to return as soon as it can identify which subformat was used to generate the current source value. Framing the problem in this way allows Narcissus to leverage other derivation rules to build this function. We will see another example of this paradigm in the rule for IP checksums presented in Section 5.

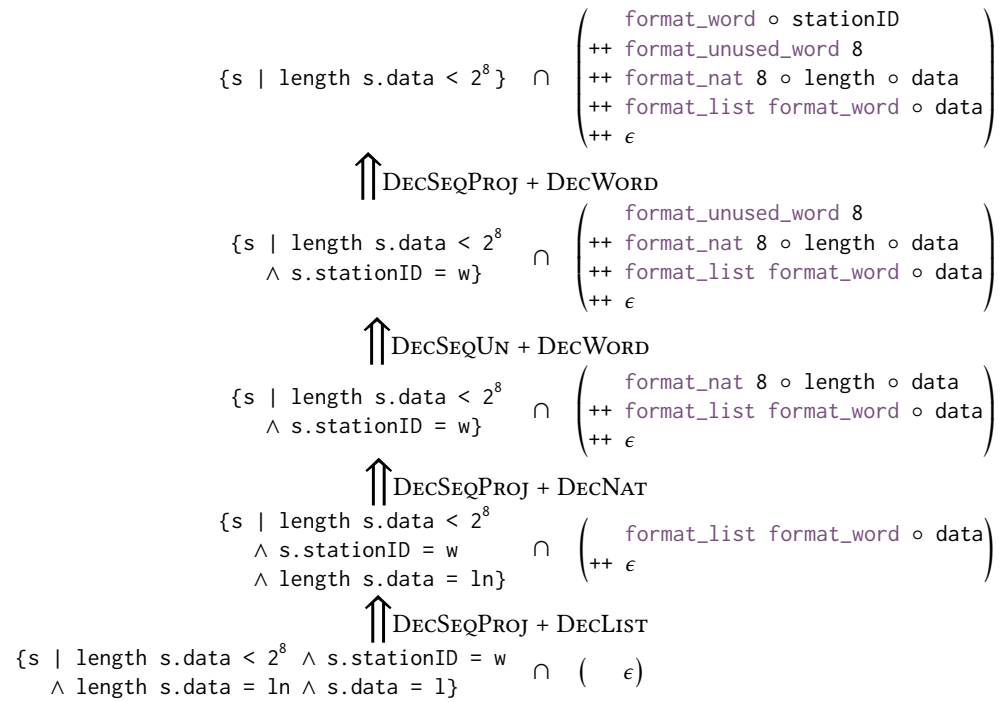

Fig. 6. An example of constraints added to a format during a decoder derivation.

3.2 Improving Performance of Encoders and Decoders

The encoders and decoders derived via our combinator rules utilize the same bitstring abstract data type as format specifications, employing the bitstring's snoc and unfold operations to enqueue and dequeue individual bits. Operating at the bit level imposes a large performance hit on these functions, since implementing these methods on the fixed-length byte buffers typically used for the target data type requires bitshifts. Converting encoders and decoders to use byte-level operations greatly improves the performance of these functions, to the point that they can be competitive with hand-implemented implementations, as our evaluation in Section 6 will show. In order to do so without compromising our correct-by-construction guarantee, we will justify this conversion using an equivalence between bit-aligned and byte-aligned functions.

The signatures of the byte-aligned functions instantiate the target type of their bit-aligned versions to a byte buffer of fixed length n:

AlignEncodeM S (n : nat) Σ := S → ByteBuff n → nat → Σ → Option (ByteBuff n × nat × Σ)
AlignDecodeM S (n : nat) Σ := ByteBuff n → nat → Σ → Option (S × nat × Σ)

In addition to fixing the target type, byte-aligned encoders now take the byte buffer they write to, and both functions now carry the index of the next byte to read/write. Both functions are instances of the state and error monads, although we force AlignDecodeM to be read-only by threading the byte buffer through the reader monad. We equip AlignEncodeM with a SetCurrentByte operation that sets the byte at the current index while updating that index, and AlignDecodeM with a corresponding GetCurrentByte operation for dequeuing bytes. We define the twin equivalences used to justify the correctness of byte-optimized functions as follows:

$$\frac{\text{decode\_bits}_1 \preceq \text{decode\_bytes}_1 \qquad \forall v.\ \text{decode\_bits}_2\ v \preceq \text{decode\_bytes}_2 v}{\text{decode\_bits}_1 \gg\!= \text{decode\_bits}_2 \preceq \text{decode\_bytes}_1 \gg\!= \text{decode\_bytes}_2}\ (\textsc{AlignDecSeq})$$

$$\frac{}{\text{throw} \preceq \text{throw}}\ (\textsc{AlignDecThrow}) \qquad \frac{}{\text{decode\_word}_8 \preceq \text{GetCurrentByte}}\ (\textsc{AlignDecByte}) \qquad \frac{}{\text{return a} \preceq \text{return a}}\ (\textsc{AlignDecReturn})$$

Fig. 7. A selection of byte-alignment rules for decoders.

Definition 3.3 (Correctness of Byte-Aligned Encoders). A byte-aligned encoder encode_bytes and bit-aligned encoder encode_bits are equivalent, encode_bits ≃ encode_bytes, iff:

- encode_bytes encodes the same bit sequence at the beginning of its byte buffer as encode_bits.
- encode_bytes fails when encode_bits would write past the end of the fixed-length byte buffer.
- encode_bytes fails whenever encode_bits does.

Definition 3.4 (Correctness of Byte-Aligned Decoders). A byte-aligned decoder decode_bytes and bit-aligned decoder decode_bits are equivalent, decode_bits ⪯ decode_bytes, iff:

- decode_bytes produces the same value as decode_bits, while consuming the same number of bits.
- decode_bytes fails when decode_bits would read past the end of the fixed-length byte buffer.
- decode_bytes fails whenever decode_bits does.

Armed with these definitions, we can build transformation rules for deriving correct byte-aligned implementations from bit-aligned functions in a similar manner to the previous section. Figure 7 gives examples of the rules for byte-aligning decoders. The most important of these is AlignDecSeq, which establishes that the byte-alignment transformation can be decomposed through sequences. The rules for byte-aligned encoders are similar. Note how AlignDecByte proves an equivalence between dequeuing an 8-bit word and AlignDecodeM's GetCurrentByte operation. A key part of automating derivations using these rules is reassociating sequences of bit-aligned decoders so that this rule applies, as the next section discusses in more detail.

4 AUTOMATING DERIVATIONS

As illustrated in Section 1.1, Narcissus provides a set of tactics to help automate the derivations described above. The tactic `derive_encoder_decoder_pair` presented in that tour is actually implemented via a pair of proof-automation tactics, DeriveEncoder and DeriveDecoder, that derive encoders and decoders, respectively. Algorithm 1 presents the pseudocode algorithm for DeriveDecoder; DeriveEncoder has a similar implementation. In addition to the top-level format, *fmt*, this tactic takes as input libraries of decoder-derivation and byte-alignment rules, *drules* and *arules*, which allow the tactic to be extended to support new formats. DeriveDecoder first converts the input format to a normal form by right associating sequences and collapsing nested applications of the format-composition operator ⊙. Next, the tactic attempts to derive a bit-aligned decoder for *fmt* via the ApplyRules subroutine that recursively applies the derivation rules in *drules*. If a bit-level decoder is found, the algorithm again normalizes the result using the monad laws and attempts to derive a byte-aligned decoder by calling the AlignDecoder subroutine. Before diving into the details of the ApplyRules and AlignDecoder tactics, we emphasize that DeriveDecoder is *interactive*: if it gets stuck on a goal it cannot solve with the current rule libraries, it presents that goal to the user to solve interactively, as in the derivation in box 5.1.

In the implementation of ApplyRules, derivation rules are implemented as tactics that apply correctness lemmas to decompose the current CorrectDecoder goal into a set of simpler subgoals in the standard interactive proof style. Conceptually, ApplyRules treats each derivation rule

Algorithm 1 Derive a byte-aligned decoder from a format

```
1:  function DeriveDecoder(fmt, drules, arules)
    Input: fmt: a format relation
           drules: set of decoder combinators derivation rules
           arules: set of byte-alignment transformation rules
    Output: dec: a byte-aligned decoder inverting fmt
2:      fmt_0 ← NormalizeFormat(fmt)
3:      dec ← ApplyRules(fmt_0, drules)
4:      dec_0 ← NormalizeDecoder(dec)
5:      AlignDecoder(dec_0, arules)

6:  function AlignDecoder(dec, arules)
7:      for rule ← arules do
8:          try
9:              dec_0 ← DecAssoc(dec)
10:             ⟨dec, K⟩ ← rule(dec)
11:             dec ← AlignDecoder(dec, rules)
12:             return K(dec)
```

```
1:  function ApplyRules(fmt, drules)
2:      try
3:          FinishDecoder(fmt)
4:      for rule ← drules do
5:          try
6:              ⟨fmt, P, K⟩ ← rule(fmt)
7:              dec ← ApplyRules(fmt, rules)
8:              if SolveSideConditions(P) then
9:                  return K(dec)

10: function FinishDecoder(fmt)
11:     try
12:         ⟨∅, (P_src P_dec), K⟩ ← DecDone(fmt)
13:         s ← ExtractView(P_src)
14:         b ← DecidePredicate(P_dec)
15:         return if b then return s else fail
```

rule ∈ FormatM S T ⇀ ⟨FormatM S_i T, Prop, Cont⟩ as a partial function, each mapping a format to a triple of a (possibly empty) set of subformats, a set of side conditions, and a continuation $\mathcal{K}$ ∈ Cont ≡ DecodeM S_i T → DecodeM S T that can construct a bit-aligned decoder from bit-aligned decoders for those subformats, when those side conditions are satisfied. The subformats represent the ≈ premises of each derivation rule, the side conditions capture its other premises, and the continuation is the decoder in its conclusion. Thus, DecSeqProj can be thought of as a function that returns the subformats Q ∩ fmt_1 and {s | f s = s′ ∧ P s} ∩ fmt_2, the side conditions ∀s. P s → Q (f s), and the continuation $\lambda d_1 d_2 . d_1 +_D d_2$, when applied to a format of the form P ∩ fmt_1 ∘ f+fmt_2. A rule can fail when the format in its conclusion does not match the current goal, when its CorrectDecoder subformats cannot be decoded, or when its side conditions are not satisfied, e.g. DecList fails when an appropriate length cannot be identified. ApplyRules first attempts to solve the goal completely via the FinishDecoder tactic, which we will discuss shortly.

If that tactic fails to find an appropriate decoder, the algorithm iteratively attempts to apply the available rules to the current format, starting with rules for base formats. If a rule is successfully applied, the algorithm recursively calls ApplyRules to derive decoders for any generated subformats. If those derivations are successful, the algorithm applies the continuation to the results and returns a finished decoder. If a recursive call fails to process a subformat completely, the tactic pauses and returns the corresponding CorrectDecoder subgoal, so that the user can see where the automation got stuck. AlignDecoder is algorithmically similar to ApplyRules, with the important modification that it attempts to reassociate the topmost decoder using the DecAssoc tactic before applying its transformation rules.

The FinishDecoder tactic warrants special discussion. FinishDecoder attempts to finish a derivation of a complete source value by finding instantiations of the s and b metavariables in the DecDone rule. Importantly, the original source value cannot be used for either, but must instead be instantiated with values that *only* use previously parsed data. Automatically finding an instance of s is particularly worrisome, as it is well-known that Ltac, Coq's proof-automation language, does not provide good support for introspecting into definitions of inductive types, and we would like to use Ltac to construct records of fairly arbitrary types, without relying on OCaml plugins.

Thankfully, a combination of standard tactics for case analysis and rewriting are up to the task. Let us see how the ExtractView tactic tries to discharge the first proof obligation of DecDone for the derivation from Section 3.1, which is presented in Figure 8. This figure denotes the unknown existential variable representing s as $\square_s$. ExtractView first uses Coq's standard destruct tactic to perform case analysis on s, generating the second subgoal presented in Figure 8, with occurrences of s replaced by its constructor applied to new variables x_1 and x_2. ExtractView then attempts to remove any variables that are not in the scope of the existential variable by rewriting the current goal using any equalities about the original source value available in the context. The resulting final goal equates $\square_s$ to previously decoded values and can be solved by unifying the two sides via the reflexivity tactic. Importantly, since s was not available when $\square_s$ was quantified, this final tactic *only* succeeds when the rewritten term depends solely on previously decoded data. FinishDecoder then attempts to solve a similar goal with a hole for b using the DecidePredicate tactic that employs known decision procedures and simplifies away any tautologies, relying on a special type class to resolve any user-defined predicates. We pause here to reemphasize that, while FinishDecoder relies on heuristic-based proof automation in a best effort attempt to solve the goal and is thus incomplete, the failure of a tactic does not necessarily spell the end of a derivation. By virtue of being implemented in an interactive proof assistant, Narcissus can loop users in when a derivation gets stuck: if FinishDecoder cannot find a decoder for the empty format, the user is presented a subgoal like the one at the top of Figure 8, so that they can attempt to solve the subgoal *interactively*.

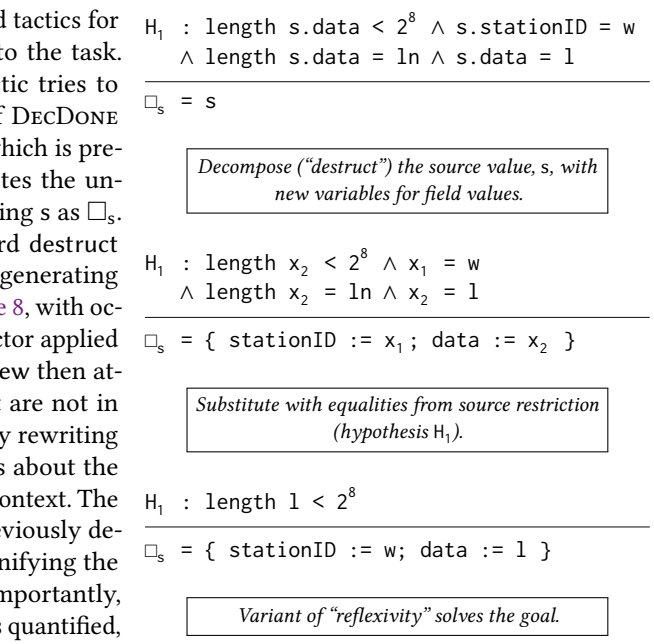

Fig. 8. Example Ltac reconstruction of the original source value at the end of the derivation in Figure 6.

5 EXTENDING THE FRAMEWORK

As outlined in Section 1.1, an extension to Narcissus consists of four pieces: a format, encoder and decoder combinators, derivation rules, and automation for incorporating these rules into DeriveEncoder and DeriveDecoder. As a concrete example, consider the format of the Internet Protocol (IP) checksum used in the IP headers, TCP segments, and UDP datagrams featured in our case studies. Figure 9 presents the format, decoder combinator, and decoder derivation rule needed for Narcissus to support IP checksums. IP_Checksum_format is a higher-order combinator in the spirit of ++; the key difference is that it uses the bitstrings produced by its subformat parameters to build the IP checksum (the one's complement of the bitstrings interpreted as lists of bytes), which it inserts between the two encoded values to produce the output bitstring. The IP_Checksum_decode combinator has two subdecoder parameters: it uses the first to calculate the number of bytes included in the checksum, and then it validates the checksum before decoding the rest of the string using its second parameter. The derivation rule for this format guarantees that, when given the correct number of bytes to include in the checksum, this test will always succeed for uncorrupted data and that it can avoid parsing the rest of the input otherwise. Figure 10 presents a complete example of this checksum combinator being used to derive encoders and decoders for IP headers.

```
Let IP_Checksum_format {S Σ} format1 format2 (s : S) (env : Σ) :=
  ‘(p, env) ← format1 s env;
  ‘(q, env) ← format2 s (addE env 16);
  c ← { c : word 16 | ∀ ext,
        IPChecksum_Valid (bin_measure (p ++ (encode_word c) ++ q))
                         (p ++ (encode_word c) ++ q ++ ext) };
  ret (p ++ (encode_word c) ++ q, env).
```

```
Let IP_Checksum_decode {S Σ} decM (decP : DecodeM S B Σ) (bin : B) (env : Σ) :=
  ‘(n, _, _) ← decM bin env;
  if checksum_Valid_dec (n * 8) bin then decP bin env
  else None
```

$$\frac{\begin{array}{c}\forall (s, \sigma_E, t, \sigma_E') \in \mathsf{fmt}_1.\ \mathsf{length}\ t = \mathsf{len}_1\ s \qquad \forall s.\ \mathsf{len}_1\ s \bmod 8 = 0 \\ \forall (s, \sigma_E, t, \sigma_E') \in \mathsf{fmt}_2.\ \mathsf{length}\ t = \mathsf{len}_2\ s \qquad \forall s.\ \mathsf{len}_2\ s \bmod 8 = 0 \\ \mathsf{P} \cap \mathsf{fmt}_1\ \mathsf{++}\ \mathsf{format\_unused\_word}\ 16\ \mathsf{++}\ \mathsf{fmt}_2 \approx \mathsf{dec}_P \\ \mathsf{P} \cap \mathsf{subformat}_{\{(s,n)\ |\ \overbrace{\overbrace{len_1 s}^{\mathsf{len}_1} + 16 + len_2 s}^{\mathsf{len}_2} = n \times 8\}} \mathsf{dec}_M \sim \mathsf{format}_M \\ \mathsf{subformat} \leq (\mathsf{fmt}_1\ \mathsf{++}\ \mathsf{format\_unused\_word16}\ \mathsf{++}\ \mathsf{fmt}_2)\end{array}}{\mathsf{P} \cap \mathsf{IP\_Checksum\_format}\ \mathsf{fmt}_1\ \mathsf{fmt}_2 \approx \mathsf{IP\_Checksum\_decode}\ \mathsf{dec}_M\ \mathsf{dec}_P}\ (\textsc{DecChkSum})$$

Fig. 9. Format, decoder, and decoder combinator for IP Checksums.

Figure 10 also includes a complete example of a new decoder-derivation tactic, which implements the DecChkSum rule presented in Figure 9. DecChkSum is similar to DecSeqProj, with a couple of key additional assumptions. The first four of these ensure the bytestrings produced by each subformat have constant length and are properly byte-aligned, which is needed to prove the validity of the initial checksum test. More interesting is the last assumption, which uses a decoder to calculate the number of bytes to include in the checksum. As with the DecUnion rule from Figure 5, this framing allows ApplyRules to discharge this condition recursively during a derivation. The new_decoder_rules tactic applies the DecChkSum rule and attempts to discharge the first four assumptions by using a database of facts about the lengths of encoded datatypes and the modulus operator, relying on ApplyRules to derive decoders for the subformats. Note that this tactic is a realization of the logic of the body of ApplyRules's loop, deriving subdecoders recursively while discharging other subgoals immediately. Other derivation rules included in Narcissus have similar implementations.

6 EVALUATION

To evaluate the expressiveness and real-world applicability of Narcissus, we wrote specifications and derived implementations of encoders and decoders for five of the most commonly used packet formats of the Internet protocol suite: Ethernet, ARP, IPv4, TCP, UDP. These formats were chosen to cover the full TCP/IP stack while offering a wide variety of interesting features and challenges:

Checksums. An IPv4 packet header contains a checksum equal to the one's-complement sum of the 16-bit words resulting from encoding all other fields of the header.

Pseudoheaders. TCP and UDP segments also contain checksums, but they are computed on a segment's payload prefixed by a *pseudoheader* that incorporates information from the IP layer. This pseudoheader is not present in the encoded packet.

```
Record IPv4_Packet :=
  { TotalLength: word 16; ID: word 16;
    DF: 𝔹; MF: 𝔹; FragmentOffset: word 13; TTL: word 8;
    Protocol: EnumType ["ICMP"; "TCP"; "UDP"];
    SourceAddress: word 32; DestAddress: word 32;
    Options: list (word 32) }.

Definition ProtocolTypeCodes := (* Protocol Numbers from RFC 5237 *)
  [0b00000001 (* ICMP: 1 *); 0b00000110 (* TCP: 6 *); 0b00010001 (* UDP: 17 *)].

Definition IPv4_Packet_Format : FormatM IPv4_Packet ByteString :=
    (format_nat 4 ∘ (constant 4)
  ++ format_nat 4 ∘ (plus 5) ∘ @length _ ∘ Options
  ++ format_unused_word 8 (* TOS Field! *)
  ++ format_word ∘ TotalLength
  ++ format_word ∘ ID
  ++ format_unused_word 1 (* Unused flag! *)
  ++ format_bool ∘ DF
  ++ format_bool ∘ MF
  ++ format_word ∘ FragmentOffset
  ++ format_word ∘ TTL
  ++ format_enum ProtocolTypeCodes ∘ Protocol)
  ThenChecksum IPChecksum_Valid OfSize 16 ThenCarryOn
    (format_word ∘ SourceAddress
  ++ format_word ∘ DestAddress
  ++ format_list format_word ∘ Options).

Definition IPv4_Packet_OK (ipv4 : IPv4_Packet) :=
  (length ipv4.(Options)) < 11 ∧
  20 + 4 * (length ipv4.(Options)) < wordToNat ipv4.(TotalLength).

Ltac new_encoder_rules ::=
  match goal with
    ⊢ CorrectAlignedEncoder (_ ThenChecksum _ OfSize _ ThenCarryOn _) _ ⇒
    eapply @CorrectAlignedEncoderForIPChecksumThenC;
    [ normalize_encoder_format| normalize_encoder_format | repeat calculate_length_ByteString
      | solve_mod_16 | repeat calculate_length_ByteString | solve_mod_8 ]
 end.

Ltac new_decoder_rules ::=
  match goal with
  | H : cache_inv_Property ?mnd _
    ⊢ CorrectDecoder _ _ _ _ (?fmt1 ThenChecksum _ OfSize _ ThenCarryOn ?fmt2) _ _ _ ⇒
    eapply compose_IPChecksum_format_correct' with (format1 := fmt1);
    [ exact H | repeat calculate_length_ByteString | repeat calculate_length_ByteString
    | solve_mod_8 | solve_mod_8 | intros; normalize_format; apply_rules
    | normalize_format; apply_rules | solve_Prefix_Format ]
end.

Let enc_dec : EncoderDecoderPair IPv4_Packet_Format IPv4_Packet_OK.
Proof. derive_encoder_decoder_pair. Defined.

Let IPv4_encoder := encoder_impl enc_dec.
Let IPv4_decoder := decoder_impl enc_dec.
```

Fig. 10. Format for IP version 4 headers, using the IP Checksum format.

Unions. An Ethernet frame header contains a 16-bit `EtherType` field, encoding either the length of the frame's payload (up to 1500 bytes) or a constant indicating which protocol the frame's payload encapsulates. The two interpretations were originally conflicting, but the ambiguity was resolved in IEEE 802.3x-1997 by requiring all EtherType constants to be above 1535. This dichotomy is easily expressed in Narcissus as a union format.

Constraints and underspecification. TCP, UDP, and IP headers include reserved-for-future-use or underspecified bits, as well as fields with interdependencies (for example, the 16-bit urgent-pointer field of a TCP packet is only meaningful if its `URG` flag is set, and the options of a TCP packet mush be zero-padded to a 32-bit boundary equal to that specified by the packet's data-offset field).

The specifications of these formats are short and readable: each format typically requires 10 to 20 lines of declarative serialization code and 10 to 20 lines of record-type, enumerated-type, and numeric-constant declarations. In addition to the base set of formats, these specifications leverage a few TCP/IP-specific extensions including checksums, pseudoheader checksums, and custom index functions for union types.

The decoders that our framework produces are reasonably efficient and sufficiently full-featured to be used as drop-in replacements for all encoding and decoding components of a typical TCP/IP stack. In the rest of this section, we describe our extraction methodology and support our claims by presenting performance benchmarks and reporting on a fork of the native-OCaml `mirage-tcpip` library used in the MirageOS unikernel, rewired to use our code to parse and decode network packets. We use Coq's extraction mechanism to obtain a standalone OCaml library, using OCaml's integers to represent machine words and natural numbers, a native-code checksum implementation, and custom array data structures for the bytestrings and vectors that encoders and decoders operate on. These custom data structures are unverified and thus part of our trusted base. Our source-code distribution contains detailed instructions to help readers reproduce our results, as well as pointers to patched version of `mirage-tcpip` and `mirage-www`.

6.1 Benchmarking

Figure 11 shows single-packet encoding and decoding times, estimated by regressing over the time needed to run batches of n packet serializations or deserializations for increasingly large values of n (experimental data were obtained using the `Core_bench` OCaml library [Hardin and James 2013]).

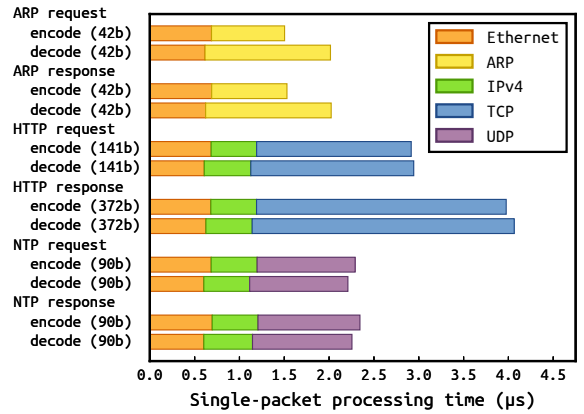

Fig. 11. Processing times for various network packets on an Intel Core i7-4810MQ CPU @ 2.80GHz. Each row shows how each layer of the network stack contributes to encoding and decoding times. TCP and UDP checksums are computed over the entirety of the packet, payload included, which explains the higher processing times. The HTTP and NTP payloads are a `GET` request to `http://nytimes.com` and a clock-synchronization request to `time.nist.gov`.

6.2 Mirage OS Integration

MirageOS [Madhavapeddy et al. 2013] 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 standalone kernel running on top of the Xen hypervisor. Security is a core feature of MirageOS, making it a natural target to demonstrate integration of our en-

coders and decoders. Concretely, this entails patching the `mirage-tcpip`[3] library to replace its serializers and deserializers by our own and evaluating the resulting code in a realistic network application. We chose the `mirage.io` website (`mirage-www` on OPAM), which shows that the overhead of using our decoders in a real-life application is very small.

Setup. After extracting the individual encoders and decoders to OCaml, we reprogrammed the TCP, UDP, IPv4, ARPv4, and Ethernet modules of the `mirage-tcpip` library to use our code optionally, and we recompiled everything. This whole process went smoothly: Mirage's test suite did not reveal issues with our proofs, though we did have to adjust or disable some of Mirage's tests (for example, one test expected packets with incorrect checksums to parse successfully, but our decoders reject them).

We strove to integrate into `mirage-tcpip` with minimal code changes: the vast majority of our changes affect the five files concerned with marshaling and unmarshaling our supported formats. This yields a good estimate of the amount of modification required (roughly 15 to 30 lines of glue code for each format), but it leaves lots of optimization opportunities unexplored: we incur significant costs doing extra work and lining up mismatched representations. Additionally, because we are strict about rejecting nonconforming packets, we perform new work that Mirage was not performing, such as computing checksums at parsing time or validating consistency constraints (Mirage's packet decoders are a combination of hand-written bounds checks and direct reads at automatically computed offsets into the packets).

Benchmarking. To evaluate the performance of the resulting application, we ran the `mirage-www` server atop our modified `mirage-tcpip` and measured the time needed to load pages from the `mirage.io` website as we replaced each component by its verified counterpart (we repeated each measurement 250 times, using the `window.performance.timing` counters in Firefox to measure page load times). The incremental overhead of our verified decoders and encoders is minimal, ranging from less than 1% on small pages to 0.5-4% on large pages such as the `blog/` section of the MirageOS website (Figure 12).

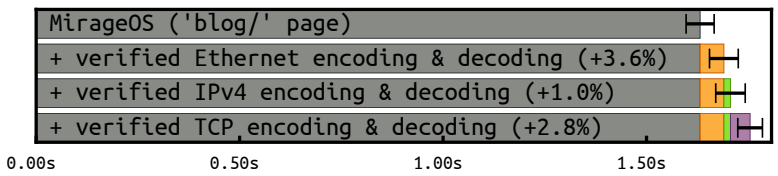

Fig. 12. Incremental overhead on page load times for `mirage.io/blog` incurred when replacing each encoder and decoder in `mirage-tcpip` by an implementation derived using Narcissus. Accessing this page causes the client to fetch about 4.2 MB of data, obtained through 36 HTTP requests spread across 1040 TCP segments. Each row shows one added encoder/decoder pair. Error bars indicate the 95% confidence intervals.

7 RELATED WORK

Parsers for Context-Free Languages. There is a long tradition of generating parsers for context-*free* languages from declarative Backus-Naur-form specifications [Johnson 1979; Parr and Quong 1995] automatically. Such generators may themselves have errors in them, so in order to reduce the trusted code base of formally verified compilers, there have been a number of efforts in verifying

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

standalone parsers for a variety of context-free languages [Barthwal and Norrish 2009; Bernardy and Jansson 2016; Jourdan et al. 2012; Koprowski and Binsztok 2011; Ridge 2011]. In closely related work, the authors of RockSalt [Morrisett et al. 2012] developed a regular-expressions DSL, equipped with a relational denotational semantics, in order to specify and generate verified parsers from bitstrings into various instruction sets. In subsequent work, Tan and Morrisett [2018] extended this DSL to support bidirectional grammars in order to provide a uniform language for specifying and generating both decoders and encoders, proving a similar notion of consistency to what we present here. Danielsson [2013], on the other hand, shows how to extend traditional approaches to derive correct-by-construction pretty-printers, whose output is guaranteed to be parsable into the original value by a user-supplied context-free grammar. In the FliPpr language [Matsuda and Wang 2018], instead of supplying the grammar, users annotate a pretty-printer with additional directives specifying how to parse non-pretty input; the annotated program is then automatically inverted to generate a context-free grammar. Importantly, all of these works focus on languages that are insufficient for many network protocols. Additionally, parsers generated from BNF specifications produce ASTs for types defined by their input grammars; these ASTs may need to be processed further (possibly using *semantic actions*) to recover original source values. This processing phase must itself be verified to guarantee correctness of the entire decoder.

Verification of Parsers for Network Protocol Formats. A wide range of tools have been used to verify generated parsers for binary network protocol formats [Amin and Rompf 2017; Protzenko et al. 2017; Simmons 2016; Swamy et al. 2016; Tullsen et al. 2018], including the SAW symbolic-analysis engine [Dockins et al. 2016], the Frama-C analyzer [Cuoq et al. 2012], F* [Swamy et al. 2016], Agda [van Geest and Swierstra 2017], and Coq. The correctness properties of each project differ from Narcissus's: Amin and Rompf [2017] focus on memory safety. While Protzenko et al. [2017] and Tullsen et al. [2018] prove that a pair of encoder and decoder functions satisfy a round-trip property similar to ours, relying on deterministic functions rules out many common formats, including DNS packets, Google Protocol Buffers, or formats using ASN.1's BER encoding. In addition, some of these approaches only support constrained sets of formats: Tullsen et al. [2018] are restricted to ASN.1 formats, while Simmons [2016] requires the format to align with the source type. Van Geest and Swierstra [2017] also use a library of parsers and pretty printers for a fixed set of data types to build implementations, but they rely on verified datatype transformations to support a more flexible set of formats, including IPv4 headers. More closely related is the verified protocol-buffer compiler of Ye and Delaware [2019], whose development adopts Narcissus's definition of correctness for top-level decoders and reuses its format for fixed-length words. That effort did not adopt Narcissus's combinator-based philosophy, as the compiler was hand-written and manually verified in Coq, and is limited to formats in the fixed data-description language of the Protocol Buffer standard.

Deductive Synthesis. The idea of deriving correct-by-construction implementations from specifications using deductive rules has existed for at least half a century [Dijkstra 1967; Manna and Waldinger 1979]. Kestrel's Specware [Srinivas and Jüllig 1995] system was a seminal realization of this idea and has been used to implement correct-by-construction SAT solvers [Smith and Westfold 2008], garbage collectors [Pavlovic et al. 2010], and network protocols. Deductive approaches have been employed more recently for interactive derivation of verified recursive functions in a general-purpose programming language [Kneuss et al. 2013] and cache-efficient implementations of divide-and-conquer algorithms [Itzhaky et al. 2016]. Very closely related is the Fiat framework [Chlipala et al. 2017; Delaware et al. 2015] for interactively deriving abstract data types inside of Coq with domain-specific specifications and proof automation. Narcissus builds upon Fiat, reusing its implementation of the aforementioned nondeterminism monad, in addition to some of its datatype

definitions and general proof-automation tactics. These dependencies represent a small portion of the Fiat library; the remaining aspects of Narcissus presented in this paper are novel, including the problem domain, the specifications of decoder/encoder correctness and the formulation of formats used in those specifications, and the derivation tactics for encoders and decoders.

Parser-Combinator Libraries. There is a long history in the functional-programming community of using combinators [Leijen and Meijer 2001] to eliminate the burden of writing parsers by hand, but less attention has been paid to the question of how to generate both encoders and decoders. Kennedy [Kennedy 2004] presents a library of combinators that package serializers and deserializers for data types to/from bytestrings (these functions are also called picklers and unpicklers) into a common type class. A similar project extended Haskell's Arrow class [Hughes 2000] with a reverse arrow in order to represent invertible functions [Alimarine et al. 2005]. In more closely related work, Rendel and Ostermann developed a combinator library for writing pairs of what they term partial isomorphisms [Rendel and Ostermann 2010]; that is, partial functions each of which correctly inverts all values in the other's range. The authors give a denotational semantics for their EDSL using a relational interpretation that closely mirrors Narcissus's. Importantly, proofs of correctness for these libraries, where they exist, are strictly informal.

While the focus of Narcissus is on implementing encoders and decoders for existing formats, Vytiniotis and Kennedy [2010] propose a way to create binary formats using question-answer games and generate verified encoders and decoders for these formats. This technique constructs compact, elegant, and deterministic serialization algorithms for typed data, but it does not allow users to ensure that the encoding matches an existing specification.

Bidirectional / Invertible Programming Languages. Mu et al. [2004] present a functional language in which only injective functions can be defined, allowing users to invert every program automatically. The authors give a relational semantics to this language, although every program in the language is a function. The authors show how to embed noninjective programs in their language automatically by augmenting them with sufficient information to invert each computation. They prove that this additional information can be dropped given a user-provided inversion function.

BiGUL [Ko et al. 2016] is a core language for putback-based bidirectional programming, written and verified in Agda. Unlike Narcissus, the original release of BiGUL relies on dynamic checks to guarantee that user-written lenses are well-behaved: programs that do not satisfy these informally stated constraints may fail at runtime. In a follow-up paper, Ko and Hu [2017] strengthened this foundation by developing a Hoare-style logic for BiGUL programs, potentially allowing programmers to eliminate runtime checks from correctly written lenses statically by proving that the checks always succeed (and are therefore unnecessary).

Boomerang [Bohannon et al. 2008] is a bidirectional programming language for projecting transformations on a data view back to the original source data; in contrast to Narcissus, Boomerang does not require that the original source values can be recovered from a target view. Boomerang programs are built using a collection of lens combinators, which include get, put, and create operations for transporting modifications between source and target representations. While Boomerang originally synthesized functions that assumed that every source value had a canonical target representation, it has since been extended with quotient lenses that relax this restriction [Foster et al. 2008]. The recently developed Optician tool [Miltner et al. 2017] synthesizes Boomerang programs that implement bijective string transformations from regular expressions describing source and target formats and sets of user-provided disambiguating examples. The format-decoding problem differs from the lens setting in that lenses consider how to recover a new source value from an updated target value given full knowledge of the *old* source value, while decoding must work given only a single target value.

Extensible Format-Description Languages. Interface generators such as XDR [Srinivasan 1995], ASN.1 [Dubuisson 2001], Protocol Buffers [Varda 2008], and Apache Avro [Apache Software Foundation 2016] generate encoders and decoders from user-defined data schemes. The underlying data format for these frameworks can be context-sensitive, but this format is defined by the system, however, preventing data exchange between programs using different frameworks. The lack of fine-grained control over the target representation prevents users from extending the format, which could bring benefits in dimensions like compactness, even ignoring the need for compatibility with widely used standards.

The binpac compiler [Pang et al. 2006] supports a data-format-specification language specifically developed for network protocols but does not support extending the language beyond the built-in constructs. More recent frameworks, like PADS [Fisher and Gruber 2005], PacketTypes [McCann and Chandra 2000], and Datascript [Back 2002], feature sophisticated data-description languages with support for complex data dependencies and constraints for specific data schemes but also lack support for extensions. Nail [Bangert and Zeldovich 2014] is a tool for synthesizing parsers and generators from formats in a high-level declarative language. 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, 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.

8 CONCLUSION

We have presented Narcissus, a framework for specifying and deriving correct-by-construction encoders and decoders for non-context-free formats in the style of parser-combinator libraries. 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 and encoders. We evaluated the expressiveness of Narcissus by deriving decoders and encoders for several standardized formats and demonstrated the utility and reasonable performance of the derived functions by incorporating them into the OCaml-based Mirage operating system.

ACKNOWLEDGMENTS

The authors wish to thank the anonymous reviewers for their careful reviews and detailed suggestions. In addition, the authors are thankful to KC Sivaramakrishnan for proofreading and suggestions about MirageOS, to Josh Ko for clarifications and pointers regarding BiGUL, and to Thomas Bourgeat for speedy yet careful proofreading.

This work has been supported in part by NSF grants CCF-1512611 and CCF-1521584, and by DARPA under agreement number FA8750-16-C-0007. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of DARPA or the U.S. Government.

REFERENCES

2013a. CVE-2012-5965: Stack-based buffer overflow in the `unique_service_name` function in `ssdp/ssdp_server.c` in the SSDP parser in the portable SDK for UPnP Devices 1.3.1 allows remote attackers to execute arbitrary code via a long DeviceType field in a UDP packet. (Jan. 2013). https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2012-5965

2013b. CVE-2013-1203: Cisco ASA CX Context-Aware Security Software allows remote attackers to cause a denial of service (device reload) via crafted TCP packets that appear to have been forwarded by a Cisco Adaptive Security Appliances device. (May 2013). https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2013-1203

2015. CVE-2015-0618: Cisco IOS XR 5.0.1 and 5.2.1 on Network Convergence System 6000 devices and 5.1.3 and 5.1.4 on Carrier Routing System X devices allows remote attackers to cause a denial of service via malformed IPv6 packets with extension headers. (Feb. 2015). https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2015-0618

2016. CVE-2016-5080: Integer overflow in the `rtxMemHeapAlloc` function in `asn1rt_a.lib` in Objective Systems ASN1C for C/C++ before 7.0.2 allows context-dependent attackers to execute arbitrary code or cause a denial of service, on a system running an application compiled by ASN1C, via crafted ASN.1 data. (July 2016). https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2016-5080

Artem Alimarine, Sjaak Smetsers, Arjen van Weelden, Marko van Eekelen, and Rinus Plasmeijer. 2005. There and Back Again: Arrows for Invertible Programming. In *Proceedings of the 2005 ACM SIGPLAN Workshop on Haskell (Haskell '05)*. ACM, New York, NY, USA, 86–97. https://doi.org/10.1145/1088348.1088357

Nada Amin and Tiark Rompf. 2017. LMS-Verify: Abstraction Without Regret for Verified Systems Programming. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL 2017)*. ACM, New York, NY, USA, 859–873. https://doi.org/10.1145/3009837.3009867

Apache Software Foundation. 2016. Apache Avro 1.8.0 Documentation. (2016). http://avro.apache.org/docs/current/ [Accessed May 04, 2016].

Godmar Back. 2002. 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)*. Springer-Verlag, London, UK, UK, 66–77. http://dl.acm.org/citation.cfm?id=645435.652647

Julian Bangert and Nickolai Zeldovich. 2014. Nail: A Practical Tool for Parsing and Generating Data Formats. In *11th USENIX Symposium on Operating Systems Design and Implementation, OSDI '14, Broomfield, CO, USA, October 6-8, 2014*. 615–628. https://www.usenix.org/conference/osdi14/technical-sessions/presentation/bangert

Aditi Barthwal and Michael Norrish. 2009. Verified, Executable Parsing. In *Programming Languages and Systems*, Giuseppe Castagna (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 160–174. https://doi.org/10.1007/978-3-642-00590-9_12

Jean-Philippe Bernardy and Patrik Jansson. 2016. Certified Context-Free Parsing: A formalisation of Valiant's Algorithm in Agda. *Logical Methods in Computer Science* Volume 12, Issue 2 (June 2016). https://doi.org/10.2168/LMCS-12(2:6)2016

Aaron Bohannon, J. Nathan Foster, Benjamin C. Pierce, Alexandre Pilkiewicz, and Alan Schmitt. 2008. Boomerang: Resourceful Lenses for String Data. In *Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '08)*. ACM, 407–419. https://doi.org/10.1145/1328438.1328487

Adam Chlipala, Benjamin Delaware, Samuel Duchovni, Jason Gross, Clément Pit-Claudel, Sorawit Suriyakarn, Peng Wang, and Katherine Ye. 2017. The End of History? Using a Proof Assistant to Replace Language Design with Library Design. In *2nd Summit on Advances in Programming Languages (SNAPL 2017) (Leibniz International Proceedings in Informatics (LIPIcs))*, Benjamin S. Lerner, Rastislav Bodík, and Shriram Krishnamurthi (Eds.), Vol. 71. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 3:1–3:15. https://doi.org/10.4230/LIPIcs.SNAPL.2017.3

Pascal Cuoq, Florent Kirchner, Nikolai Kosmatov, Virgile Prevosto, Julien Signoles, and Boris Yakobowski. 2012. Frama-C: A Software Analysis Perspective. In *Proceedings of the 10th International Conference on Software Engineering and Formal Methods (SEFM'12)*. Springer-Verlag, Berlin, Heidelberg, 233–247. https://doi.org/10.1007/978-3-642-33826-7_16

Nils Anders Danielsson. 2013. Correct-by-construction Pretty-printing. In *Proceedings of the 2013 ACM SIGPLAN workshop on Dependently-typed programming, DTP@ICFP 2013*. 1–12. https://doi.org/10.1145/2502409.2502410

Benjamin Delaware, Clément Pit-Claudel, Jason Gross, and Adam Chlipala. 2015. Fiat: Deductive Synthesis of Abstract Data Types in a Proof Assistant. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages - POPL '15*. ACM Press, 689–700. https://doi.org/10.1145/2676726.2677006

Edsger W. Dijkstra. 1967. A constructive approach to the problem of program correctness. (Aug. 1967). http://www.cs.utexas.edu/users/EWD/ewd02xx/EWD209.PDF Circulated privately.

Robert Dockins, Adam Foltzer, Joe Hendrix, Brian Huffman, Dylan McNamee, and Aaron Tomb. 2016. Constructing Semantic Models of Programs with the Software Analysis Workbench. In *Verified Software. Theories, Tools, and Experiments*, Sandrine Blazy and Marsha Chechik (Eds.). Springer International Publishing, Cham, 56–72. https://doi.org/10.1007/978-3-319-48869-1_5

Olivier Dubuisson. 2001. *ASN. 1: communication between heterogeneous systems*. Morgan Kaufmann.

Kathleen Fisher and Robert Gruber. 2005. PADS: A Domain-Specific Language for Processing Ad Hoc Data. In *Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation, Chicago, IL, USA, June 12-15,*

2005. 295–304. https://doi.org/10.1145/1065010.1065046

Kathleen Fisher, Yitzhak Mandelbaum, and David Walker. 2006. The Next 700 Data Description Languages. In *Proceedings of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2006, Charleston, South Carolina, USA, January 11-13, 2006*. 2–15. https://doi.org/10.1145/1111037.1111039

Pedro Fonseca, Kaiyuan Zhang, Xi Wang, and Arvind Krishnamurthy. 2017. An Empirical Study on the Correctness of Formally Verified Distributed Systems. In *Proceedings of the Twelfth European Conference on Computer Systems (EuroSys '17)*. ACM, New York, NY, USA, 328–343. https://doi.org/10.1145/3064176.3064183

J. Nathan Foster, Alexandre Pilkiewicz, and Benjamin C. Pierce. 2008. Quotient Lenses. In *Proceedings of the 13th ACM SIGPLAN International Conference on Functional Programming (ICFP '08)*. ACM, 383–396. https://doi.org/10.1145/1411204.1411257

Christopher S. Hardin and Roshan P. James. 2013. Core_bench: micro-benchmarking for OCaml. (2013). https://github.com/janestreet/core_bench

John Hughes. 2000. Generalising Monads to Arrows. *Sci. Comput. Program.* 37, 1-3 (May 2000), 67–111. https://doi.org/10.1016/S0167-6423(99)00023-4

Shachar Itzhaky, Rohit Singh, Armando Solar-Lezama, Kuat Yessenov, Yongquan Lu, Charles Leiserson, and Rezaul Chowdhury. 2016. Deriving divide-and-conquer dynamic programming algorithms using solver-aided transformations. *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications - OOPSLA 2016* (2016). https://doi.org/10.1145/2983990.2983993

Stephen C. Johnson. 1979. *Yacc: Yet Another Compiler-Compiler*. Technical Report.

Jacques-Henri Jourdan, François Pottier, and Xavier Leroy. 2012. Validating LR(1) Parsers. In *Programming Languages and Systems*, Helmut Seidl (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 397–416. https://doi.org/10.1007/978-3-642-28869-2_20

Andrew J. Kennedy. 2004. Functional Pearl: Pickler Combinators. *J. Funct. Program.* 14, 6 (Nov. 2004), 727–739. https://doi.org/10.1017/S0956796804005209

Etienne Kneuss, Ivan Kuraj, Viktor Kuncak, and Philippe Suter. 2013. Synthesis modulo recursive functions. In *Proc. OOPSLA*. 407–426. https://doi.org/10.1145/2509136.2509555

Hsiang-Shang Ko and Zhenjiang Hu. 2017. An Axiomatic Basis for Bidirectional Programming. *Proceedings of the ACM on Programming Languages* 2, POPL, Article 41 (Dec. 2017), 29 pages. https://doi.org/10.1145/3158129

Hsiang-Shang Ko, Tao Zan, and Zhenjiang Hu. 2016. BiGUL: a formally verified core language for putback-based bidirectional programming. *Proceedings of the 2016 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation, PEPM 2016* (2016), 61–72. https://doi.org/10.1145/2847538.2847544

Adam Koprowski and Henri Binsztok. 2011. TRX: A Formally Verified Parser Interpreter. *Logical Methods in Computer Science* 7, 2 (2011). https://doi.org/10.2168/LMCS-7(2:18)2011

Daan Leijen and Erik Meijer. 2001. Parsec: Direct style monadic parser combinators for the real world. (2001).

Anil Madhavapeddy, Richard Mortier, Charalampos Rotsos, David Scott, Balraj Singh, Thomas Gazagnaire, Steven Smith, Steven Hand, and Jon Crowcroft. 2013. 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)*. ACM, New York, NY, USA, 461–472. https://doi.org/10.1145/2499368.2451167

Z. Manna and R. Waldinger. 1979. Synthesis: Dreams ⇒ Programs. *IEEE Trans. Softw. Eng.* 5, 4 (July 1979), 294–328. https://doi.org/10.1109/TSE.1979.234198

Kazutaka Matsuda and Meng Wang. 2018. FliPpr: A System for Deriving Parsers from Pretty-Printers. *New Generation Computing* 36, 3 (01 Jul 2018), 173–202. https://doi.org/10.1007/s00354-018-0033-7

Peter J. McCann and Satish Chandra. 2000. Packet Types: Abstract Specification of Network Protocol Messages. In *Proceedings of the Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM '00)*. ACM, New York, NY, USA, 321–333. https://doi.org/10.1145/347057.347563

Anders Miltner, Kathleen Fisher, Benjamin C. Pierce, David Walker, and Steve Zdancewic. 2017. Synthesizing Bijective Lenses. *Proceedings of the ACM on Programming Languages* 2, POPL (Dec 2017), 1–30. https://doi.org/10.1145/3158089

P. Mockapetris. 1987. *Domain names - implementation and specification*. RFC 1035.

Greg Morrisett, Gang Tan, Joseph Tassarotti, Jean-Baptiste Tristan, and Edward Gan. 2012. RockSalt: Better, Faster, Stronger SFI for the x86. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '12, Beijing, China - June 11 - 16, 2012*. 395–404. https://doi.org/10.1145/2254064.2254111

Shin-Cheng Mu, Zhenjiang Hu, and Masato Takeichi. 2004. An Injective Language for Reversible Computation. In *Mathematics of Program Construction*, Dexter Kozen (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 289–313. https://doi.org/10.1007/978-3-540-27764-4_16

Ruoming Pang, Vern Paxson, Robin Sommer, and Larry Peterson. 2006. binpac: A yacc for writing application protocol parsers. In *Proceedings of the 6th ACM SIGCOMM conference on Internet measurement*. ACM, 289–300. https://doi.org/10.1145/1177080.1177119

T. J. Parr and R. W. Quong. 1995. ANTLR: A Predicated-LL(k) Parser Generator. *Software: Practice and Experience* 25, 7 (July 1995), 789–810. https://doi.org/10.1002/spe.4380250705

Dusko Pavlovic, Peter Pepper, and Douglas R. Smith. 2010. Formal Derivation of Concurrent Garbage Collectors. In *Mathematics of Program Construction*. Springer Berlin Heidelberg, 353–376. https://doi.org/10.1007/978-3-642-13321-3_20

Jonathan Protzenko, Jean-Karim Zinzindohoué, Aseem Rastogi, Tahina Ramananandro, Peng Wang, Santiago Zanella-Béguelin, Antoine Delignat-Lavaud, Catalin Hritcu, Karthikeyan Bhargavan, Cédric Fournet, and Nikhil Swamy. 2017. Verified Low-Level Programming Embedded in F*. *PACMPL* 1, ICFP (Sept. 2017), 17:1–17:29. https://doi.org/10.1145/3110261

Tillmann Rendel and Klaus Ostermann. 2010. Invertible Syntax Descriptions: Unifying Parsing and Pretty Printing. In *Proceedings of the Third ACM Haskell Symposium on Haskell (Haskell '10)*. ACM, New York, NY, USA, 1–12. https://doi.org/10.1145/1863523.1863525

Tom Ridge. 2011. Simple, Functional, Sound and Complete Parsing for All Context-Free Grammars. In *Certified Programs and Proofs*, Jean-Pierre Jouannaud and Zhong Shao (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 103–118. https://doi.org/10.1007/978-3-642-25379-9_10

Keith Simmons. 2016. Cheerios. (2016). https://courses.cs.washington.edu/courses/cse599w/16sp/projects/cheerios.pdf.

Douglas R. Smith and Stephen J. Westfold. 2008. Synthesis of Propositional Satisfiability Solvers. (2008).

Yellamraju V. Srinivas and Richard Jüllig. 1995. Specware: Formal support for composing software. In *Mathematics of Program Construction*, Bernhard Möller (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 399–422.

Raj Srinivasan. 1995. *XDR: External data representation standard.* Technical Report.

Nikhil Swamy, Cătălin Hriţcu, Chantal Keller, Aseem Rastogi, Antoine Delignat-Lavaud, Simon Forest, Karthikeyan Bhargavan, Cédric Fournet, Pierre-Yves Strub, Markulf Kohlweiss, Jean-Karim Zinzindohoue, and Santiago Zanella-Béguelin. 2016. Dependent Types and Multi-monadic Effects in F*. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '16)*. ACM, New York, NY, USA, 256–270. https://doi.org/10.1145/2837614.2837655

Gang Tan and Greg Morrisett. 2018. Bidirectional Grammars for Machine-Code Decoding and Encoding. *Journal of Automated Reasoning* 60, 3 (01 Mar 2018), 257–277. https://doi.org/10.1007/s10817-017-9429-1

The Coq Development Team. 2018. The Coq Proof Assistant, version 8.7.2. (Feb. 2018). https://doi.org/10.5281/zenodo.1174360

Mark Tullsen, Lee Pike, Nathan Collins, and Aaron Tomb. 2018. Formal Verification of a Vehicle-to-Vehicle (V2V) Messaging System. In *Computer Aided Verification*, Hana Chockler and Georg Weissenbacher (Eds.). Springer International Publishing, Cham, 413–429. https://doi.org/10.1007/978-3-319-96142-2_25

Marcell van Geest and Wouter Swierstra. 2017. Generic Packet Descriptions: Verified Parsing and Pretty Printing of Low-level Data. In *Proceedings of the 2Nd ACM SIGPLAN International Workshop on Type-Driven Development (TyDe 2017)*. ACM, New York, NY, USA, 30–40. https://doi.org/10.1145/3122975.3122979

Kenton Varda. 2008. Protocol Buffers. https://developers.google.com/protocol-buffers/. (2008).

Dimitrios Vytiniotis and Andrew J. Kennedy. 2010. Functional Pearl: Every bit counts. *Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming, ICFP 2010* (2010), 15–26. https://doi.org/10.1145/1863543.1863548

Qianchuan Ye and Benjamin Delaware. 2019. A verified protocol buffer compiler. In *Proceedings of the 8th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2019, Cascais, Portugal, January 14-15, 2019*. 222–233. https://doi.org/10.1145/3293880.3294105