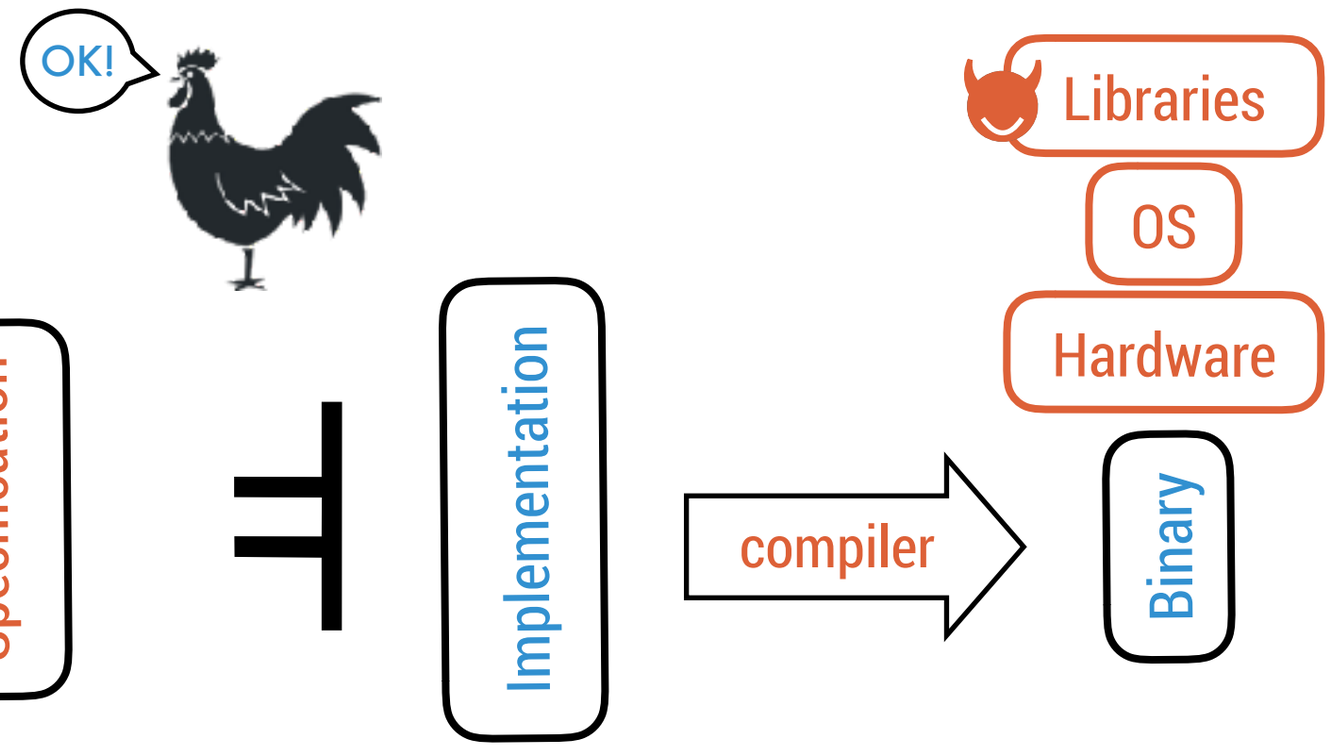
NARCISSUS: Correct-By-Construction Derivation of Decoders and Encoders from Binary Formats

Benjamin Delaware
Purdue University

Sorawit Suriyakarn[1], Clément Pit-Claudel[1], Qianchuan Ye[2], and Adam Chlipala[1]
[1]MIT, [2]Purdue University

The Larger Picture

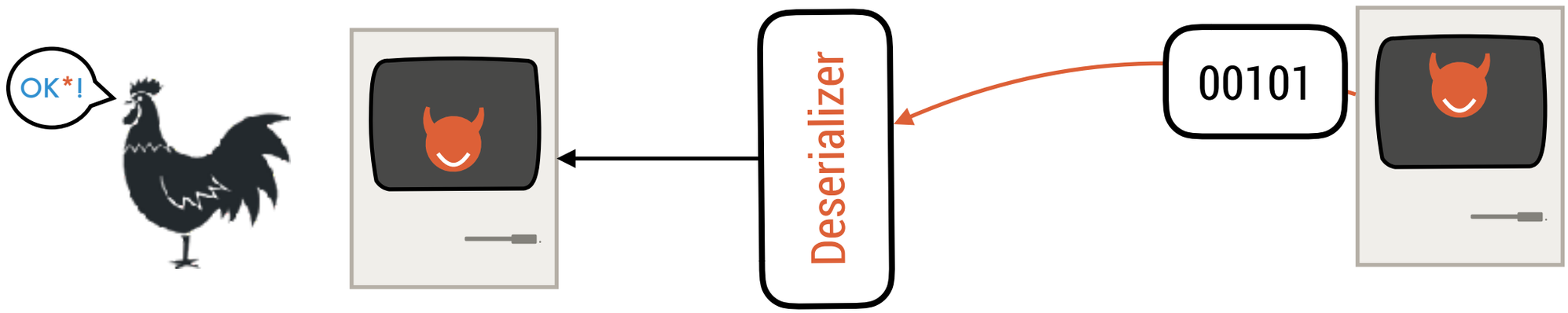
- Numerous developments of high-assurance software in proof assistants in the past five years:
  - CompCert C compiler
  - CertikOS
  - FSCQ file system
- Assurance comes from formal guarantees* provided by proof assistant:



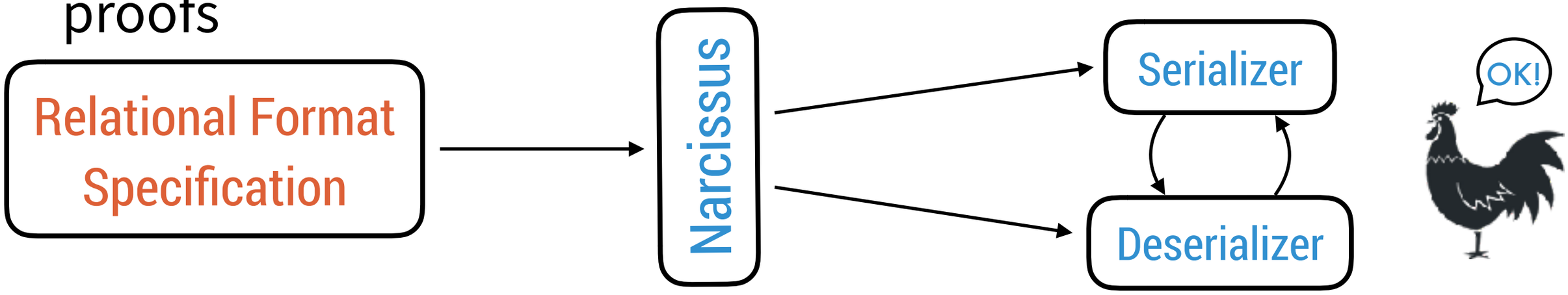
* w.r.t Trusted Base

Narcissus

- For networked systems, deserialization is important[1]
  - If these are in the TCB, bugs will break the assurance case!



- Enter Narcissus:
  - User-extensible framework for deriving encoders and decoders from format specifications, with machine-checked correctness proofs



[1] An Empirical Study on the Correctness of Formally Verified Distributed Systems. Pedro Fonseca, Kaiyuan Zhang, Xi Wang, and Arvind Krishnamurthy. 2017.

Today's Talk

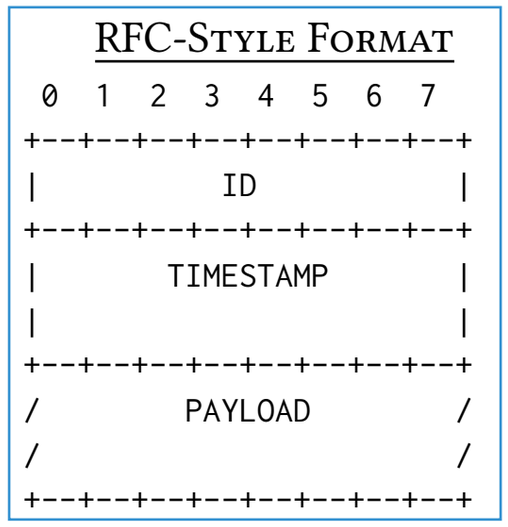
- Deriving correct-by-construction functions in Coq (encoders and decoders)
  - Formalizing standardized formats
  - Specifying correct decoders (and encoders) for these formats
  - Automating the derivation of encoders and decoders

A Simple Format

Source Type

```
Definition SimpleProd n :=
  Word (* id *)
  × DateTime (* Timestamp *)
  × Vector Word n. (* Payload *)
```

RFC-Style Format



Format

```
Definition SimpleProd_fmt n
  : FormatM (SimpleProd n) 𝕋 :=
    Pair_fmt
      Word_fmt
      (Pair_fmt
        DTime_fmt
        (Vector_fmt Word_fmt)).
```

Decoder

```
Definition SimpleProd_dec n
  : DecodeM (SimpleProd n) 𝕋 :=
    Pair_dec
      Word_dec
      (Pair_dec
        DTime_dec
        (Vector_dec Word_dec)).
```

- `SimpleProd_fmt` = composition of sub-formats
  - `Prod_fmt : ∀ S₁ S₂, Format S₁ T → Format S₂ T → FormatM (S₁×S₂) T`
- `SimpleProd_dec` = composition of decoders

Correct-by-Construction Decoders

- Format and decoder is so similar, let's define a combinator[2] packaging the two together with a proof of correctness:

```
Record CorrectFormatDecoder' (S T : Type) :=
{ format : FormatM S T; decode : DecodeM S T;
  decodeOK : format ⇄ decode }
```

"forall inputs, `decode` correctly inverts `format`"

- Correct decoder = inhabitant of this type
- Coq is quite good at finding inhabitants of types!
- **Deductive Synthesis**: "To construct a program whose output satisfies the conditions of the specification, we prove a theorem stating the existence of such an output. The proof is restricted to be sufficiently constructive so that a program computing the desired output can be extracted directly from the proof. The proof constitutes a demonstration of the correctness of this program[3]."

[2] FUNCTIONAL PEARL Pickler Combinators. Andrew J. Kennedy. 2004.
[3] Deductive Synthesis Of The Unification Algorithm. Richard J. Waldinger. 1981.

Decoder Combinators

- Proof-Carrying combinators are also compositional:

```
Definition ProdComb (S₁ S₂ T : Type)
 (fmt₁ : CorrectFormatDecoder' S₁ T) (fmt₂ : CorrectFormatDecoder' S₂ T)
: CorrectFormatDecoder' (S₁ * S₂) T :=
{ format := Pair_fmt fmt₁.format fmt₂.format;
  decode := Pair_dec fmt₁.decode fmt₂.decode;
  decodeOK := Pair_correct fmt₁.decodeOK fmt₂.decodeOK }
```

- With some combinators, we can define interactively:

```
Definition SimpleRec_Correct n :
 CorrectFormatDecoder' (SimpleRec n) T.
Proof.
apply ProdComb.
- apply WordComb.
- apply ProdComb.
  + apply DateTimeComb.
  + apply (VectorComb n).
Defined.
```

FORMAT

```
Definition SimpleProd_fmt n
 : FormatM (SimpleProd n) 𝕋 :=
   Pair_fmt
     Word_fmt
     (Pair_fmt
        DTime_fmt
        (Vector_fmt Word_fmt)).
```

Decoder Combinators

- Proof-Carrying combinators are also compositional:

```
Definition ProdComb (S₁ S₂ T : Type)
 (fmt₁ : CorrectFormatDecoder' S₁ T) (fmt₂ : CorrectFormatDecoder' S₂ T)
: CorrectFormatDecoder' (S₁ * S₂) T :=
{ format := Pair_fmt fmt₁.format fmt₂.format;
  decode := Pair_dec fmt₁.decode fmt₂.decode;
  decodeOK := Pair_correct fmt₁.decodeOK fmt₂.decodeOK }
```

- With some combinators, we can define interactively:

```
Definition SimpleRec_Correct n :
 CorrectFormatDecoder' (SimpleRec n) T.
Proof.
 auto.
Defined.
```

FORMAT

```
Definition SimpleProd_fmt n
 : FormatM (SimpleProd n) 𝕋 :=
   Pair_fmt
     Word_fmt
     (Pair_fmt
        DTime_fmt
        (Vector_fmt Word_fmt)).
```

Recap

- We're done*:
  - Define a package of formats + decoders together with proofs
  - Define compositional instances of these records
  - Use tactics to automate derive implementations automatically.

```
(•_•)
<)   )⌐ Dependent Types
/    \

\(•_•)
(   (> + Tactics
/    \

 (•_•)
<)   )>  FTW!!!
/    \
```

- Well, maybe not. This format made several simplifying assumptions that don't apply to many common formats.

*modulo some elided definitions

A Not as Simple Format

- What about this example?
  - Source Type:
    - Record Type
    - Variable-length payload
  - Format:
    - Encodes payload length
    - Out of sync with type:
      - ++ : ∀ S T, FormatM S T → FormatM S T → FormatM S T
      - ○ : ∀ S S' T, FormatM S' T → (S → S') → FormatM S T
  - Decoder:
    - Uses decoded length to parse payload
    - Needs to reconstruct source value

Source Type

```
Record SimpleRec :=
  { id : Word;
    Payload : List Word;
    Timestamp : DateTime }.
```

Format

```
Definition SimpleRec_fmt
  : FormatM SimpleRec 𝕋 :=
    Word_fmt ∘ id
  ++ Nat_fmt 8 ∘ length ∘ Payload
  ++ DTime_fmt ∘ Timestamp
  ++ List_fmt Word_fmt ∘ Payload.
```

Decoder

```
Definition SimpleRec_dec
  : DecodeM SimpleRec 𝕋 :=
    w ← Word_dec;
    ln ← Nat_dec;
    dt ← DTime_dec;
    l ← List_dec Word_dec ln;
    return { id := w; Payload := l;
             Timestamp := ln }.
```

Decoding Lists

- How to define proof-carrying combinator for Lists?

```
Definition List_Correct' {S T} (len : nat) fmtS
(fmtSOK : CorrectFormatDecoder' S T fmtS)
  : CorrectFormatDecoder' (List S) T (List_fmt fmtS) :=
{ decode := List_dec fmtS.decode len;
  decodeOK := ?? len }
```

- `?? : List_fmt fmtS` ⇄ `List_dec fmtS.decode n`
- `List_dec fmtS.decode len` builds lists of length `len`, but the intuitive spec talks about all source values!
- Correctness of the decoder depends on the larger format
- Goal: Compositional, context-aware decoder combinators

Constraining the Context

- Proposal: Restrict the source values considered:

$$\texttt{format}|_{\texttt{P}} \rightleftarrows \texttt{decode}$$

- "For all source values `s` such that `P s`, `decode` correctly inverts `format`"

- Can now define a combinator for lists:

```
Definition List_Correct' {S T} (len : nat) fmtS
(fmtSOK : CorrectFormatDecoder' S T fmtS)
  : CorrectFormatDecoder' (List S) T (List_fmt fmtS)|{l | length l = len}
 :={decode := List_dec fmtS.decode len;
  decodeOK := list_correct len }
```

- Works for other dependencies: e.g. tags, version numbers, etc.

Threading Context

- How to ensure that proof-relevant context gets passed down?

```
Definition Seq_Word_Correct {S T} P fmtS (f : S -> Word)
  (fmtSOK : ∀ w, CorrectFormatDecoder'' S T fmtS|(P ∩ { s | f s = w }))
: CorrectFormatDecoder'' S T (Word_fmt ○ f ++ fmtS)|P :=
{ decode := Word_dec >>= fmtS.decode;
  decodeOK := seq_word_OK fmtS f fmtSOK.decodeOK
```

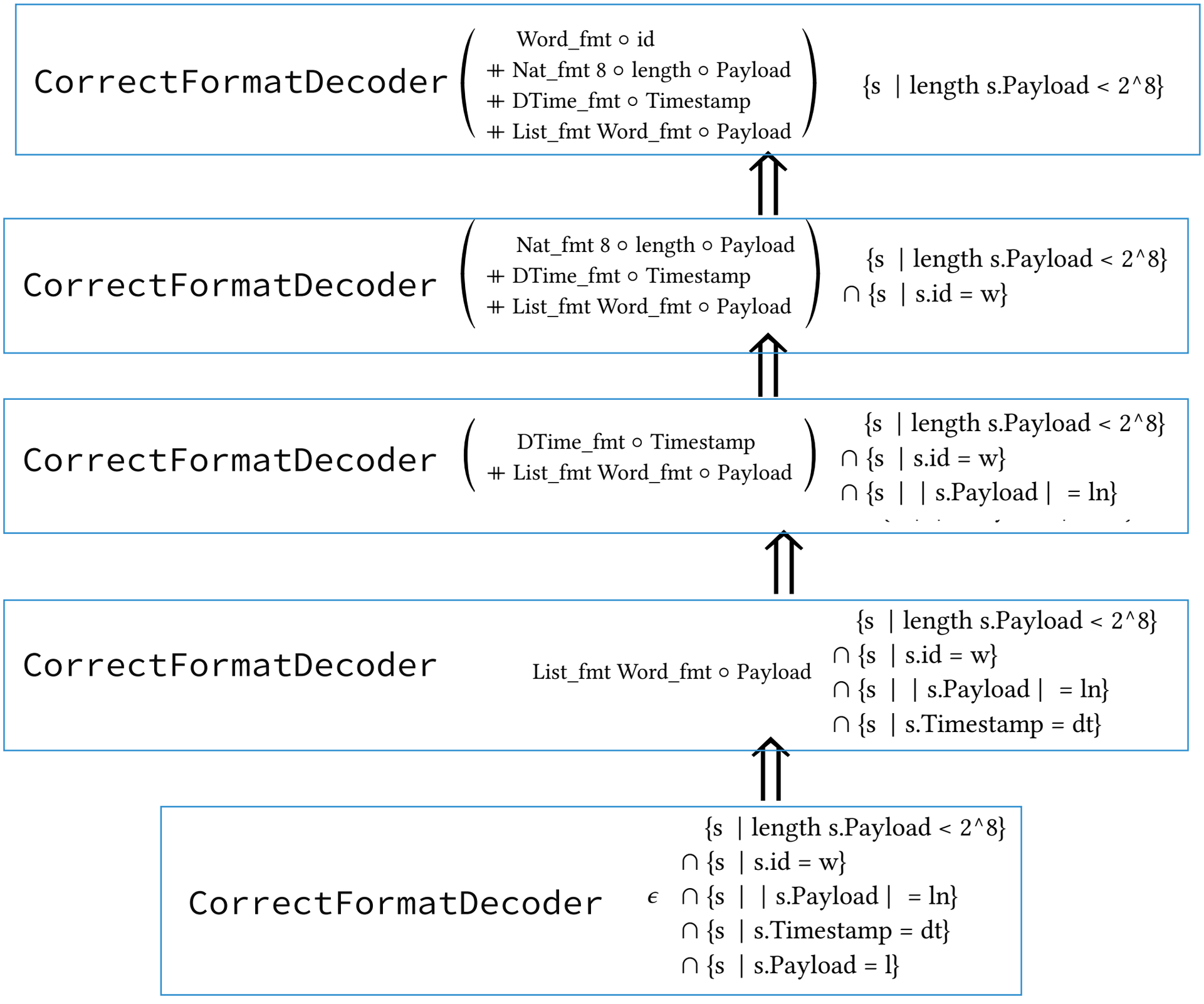
- Finding a correct decoder for a sequence of formats only needs to consider the case where the first format was decoded correctly

```
P ∩ { s | f s = w }
```

“previous context”

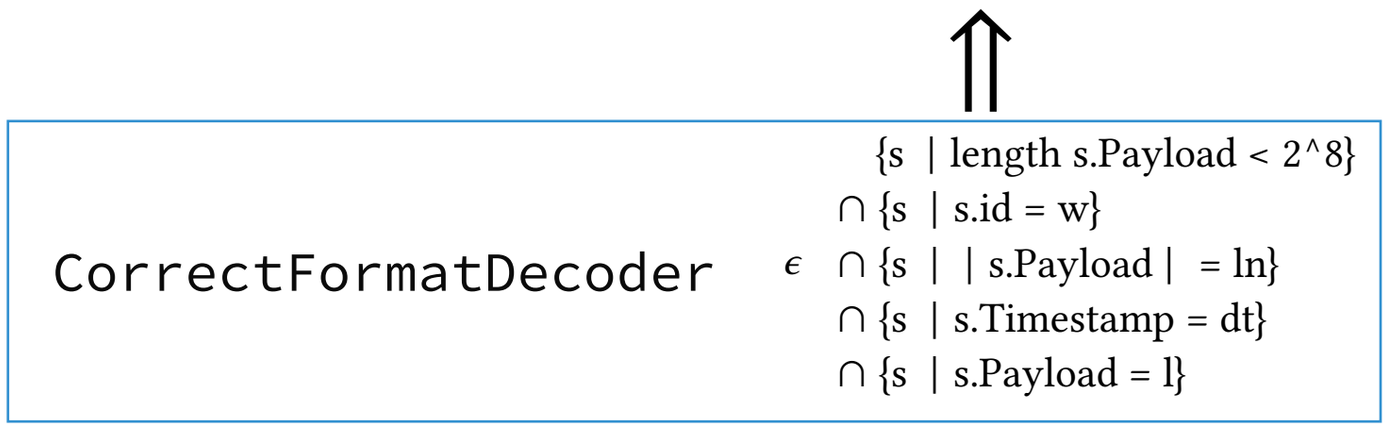
“newly decoded context”

Threading Context



Threading Context

- The last step is constrained to a single value
- It's obvious how to rebuild the original source value



DECODER

```
Definition SimpleRec_dec
  : DecodeM SimpleRec 𝕋 :=
    w ← Word_dec;
    ln ← Nat_dec;
    dt ← DTime_dec;
    l ← List_dec Word_dec ln;
    return { id := w; Payload := l;
                Timestamp := dt }.
```

Base Combinators

- Our basic building blocks:
  - Functional programs in monadic style
  - Target type is a monoid

```
-snoc t x = t ++ [x]
-unfold [x] ++ t = (x, t)
```

- Narcissus-specific details hidden in bind operation
- Proofs components are proofs

Format

```
Fixpoint Word_fmt {n}
  : FormatM (Word n) 𝕋 :=
  λw. match w with
        | ϵ ⇒ ι
        | b · w' ⇒ t ← Word_fmt w';
                   return (snoc b t)
  end.
```

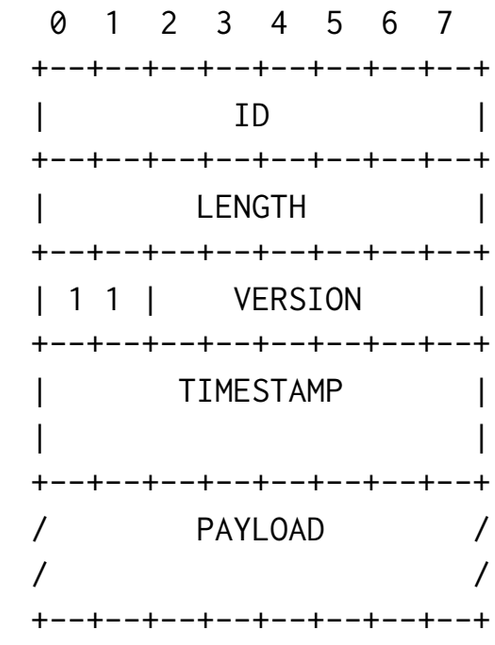
Decoder

```
Fixpoint Word_dec {n}
  : DecodeM (Word n) 𝕋 :=
λt. match n with
    | O ⇒ return ϵ
    | S n' ⇒ (b, t') ← unfold t;
             w' ← word_dec n' t';
             return (w' · b)
  end.
```

Nondeterministic Formats

- Many formats do not prescribe a canonical target
  - Version = 8-bit word whose two lowest-order bits are set
  - 2^6 different encodings for each source value
- Many Examples: Ethernet, ASN.1, DNS
- Narcissus' solution: relational interpretation of formats
  - Every source value is related to a set of target values
  - Admits new choice operator: {x | P x}

RFC-Style Format



Format

Definition NDRec_fmt
: FormatM SimpleRec 𝕋 :=
  Word_fmt ∘ id
⧺ Word_fmt ∘ length ∘ Payload
  (* New: Version Number *)
⧺ Word_fmt ∘ {w | w ≥11000000 }
⧺ DTime_fmt ∘ Timestamp
⧺ Vector_fmt Word_fmt ∘ Payload.

IPv4

- Extensions = User-Defined Components!

```
Definition IPv4_Packet_Format : FormatM IPv4_Packet T :=
   format_word ○ (fun _ => natToWord 4 4)
++ format_nat 4 ○ IPv4_Packet_Header_Len
++ 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
++ IPChecksum ++
++ format_word ○ SourceAddress
++ format_word ○ DestAddress
++ format_list format_word ○ Options.
```

Bits	0–3	4–7	8–11	12–15	16–18	19–23	24–27	28–31
0	Version	Head Length	Type of Service		Total length (Packet)			
32	Identification				Flags	Fragment Spacing		
64	Lifespan (TTL)		Protocol		Header checksum			
96	Source Address							
128	Destination Address							
160	Options							

The Emperor's Old Clothes

Charles Antony Richard Hoare
Oxford University, England

I gave desperate warnings against the obscurity, the complexity, and overambition of the new design, but my warnings went unheeded. I conclude that there are two ways of constructing a software design: One way is to make it so simple that there are obviously no deficiencies and the other way is to make it so complicated that there are no obvious deficiencies.

A Constructive Approach to the Problem of Program Correctness.

Personally I cannot refrain from feeling that many debugging aids that are en vogue now are invented as a compensation for the shortcomings of a programming technique that will be denounced as obsolete within the near future.

Design Decisions

- Nondeterministic, relational formats, with Narcissus-specific details hidden in monads
- Combinator libraries for decoders (and encoders) keyed on a common format.
- Automate derivations when possible using tactics
- Dependencies between subformats are reflected in the proofs of correctness for individual combinators.

On to the details!

Specifying Formats

- Type of Formats:
  - `FormatM S T Σ := Set of(S×Σ×T×Σ)`
- Define relations via Nondeterminism monad:
  - `e ∈ return v ↔ e = v`
  - `e ∈ { x | P x } ↔ P e`
  - `e ∈ x ←y; k x ↔ ∃ e'. e' ∈ y`
    `Λ e ∈ k e'`
- Or using higher-order formats

Sequence Formats

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

Compose Formats

$(s, \sigma, t, \sigma') \in \text{format} \circledcirc f \leftrightarrow$
$\exists\, s'.\ (s', \sigma, t, \sigma') \in \text{format} \wedge f\ s\ s'$

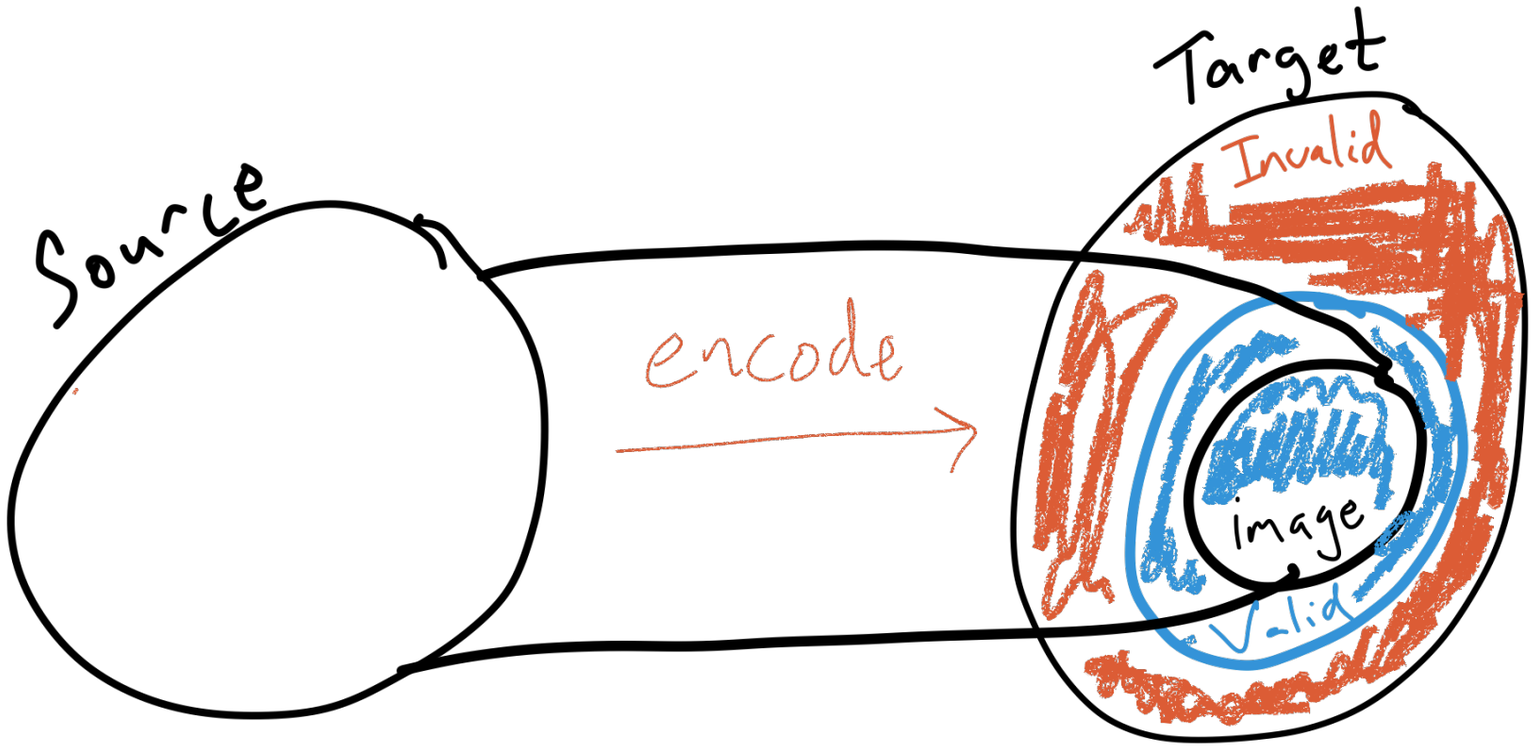
Union Format

$(s, \sigma, t, \sigma') \in \bigcup_i \text{format}_i \leftrightarrow$
$\exists\, j.\ (s, \sigma, t, \sigma') \in \text{format}_j$

Correct Encoders

- A correct encoder is a function wholly contained in the relation defined by a format

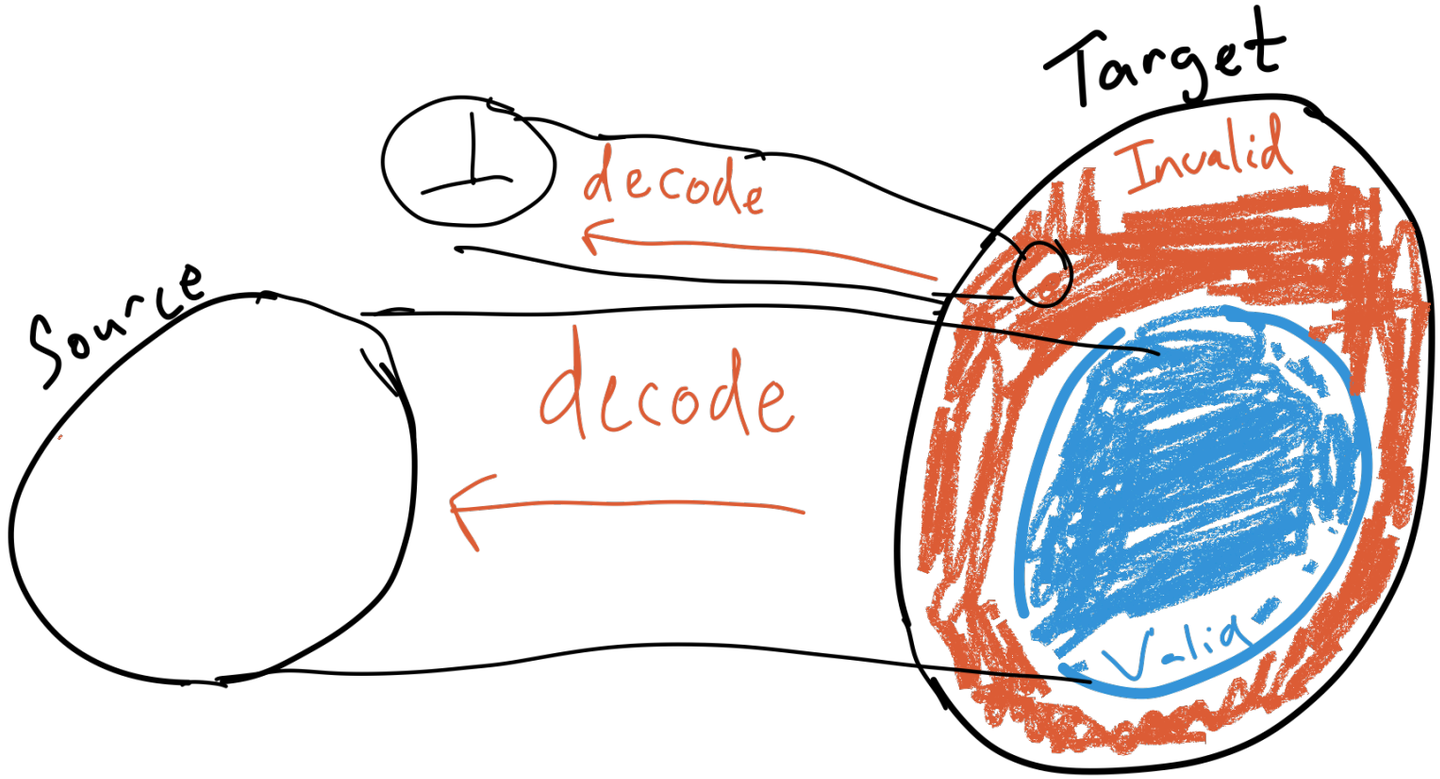
```
EncodeM S T Σ := S → Σ → Option T×Σ
```




```
encode ⊆ format ≔
∀s σ t σ'. encode s σ = Some (t,σ') → (s,σ,t,σ') ∈ format
∧∀s σ. encode s σ = None → ∀t' σ'. (s,σ,t',σ') ∉ format
```

Correct Decoders

- A correct decoder maps values in the image of the format back to the original source value, *and* signals an error for other values



$$\text{format} \approx \text{decode}.$$

$$\Updownarrow$$

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

Relating Encoders + Decoders

- Encoders and decoder are specified by a common format:

$$\text{encode} \subseteq \text{format} \lessapprox \text{decode}.$$

- Correctness of both yields the expected roundtrip properties:

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

$$\forall s\ \sigma_E\ t\ \sigma'_E\ \sigma_D.\ \text{encode}\ s\ \sigma_E = \text{Some}(t, \sigma'_E) \land \sigma_E \approx \sigma_D \to \exists \sigma'_D.\ \text{decode}\ t\ \sigma_D = \text{Some}(s, \sigma'_D)$$

Theorem 2.4 (Encode Inverts Decode). *Given a correct decoder* format $\lessapprox$ decode *and correct encoder* encode $\subseteq$ format *for a common format format, encode is defined for all decoded source values produced by decode,*

$$\forall s\ \sigma_D\ t\ \sigma'_D\ \sigma_E.\ \text{decode}\ t\ \sigma_D = \text{Some}(s, \sigma'_D) \land \sigma_E \approx \sigma_D \to \exists t'\ \sigma'_E.\ \text{encode}\ s\ \sigma_E = \text{Some}(t', \sigma'_E)$$

Encoder Combinators

- Combinators for correct-by-construction encoders are similar to decoders:

```
Record CorrectFormatEncoder (S T Σ : Type) :=
{ format : FormatM S T Σ ; encode : EncodeM S T Σ ;
   encodeOK : encode ⊆ format }
```

- Formats define a family of valid encoding strategies;
- A correct encoder picks one:

```
Fixpoint Word_enc {n} {σ}
 : EncodeM (Word n) 𝕋 σ :=
 λw. match w with
      | ϵ ⇒ ι
      | b · w' ⇒ t ← Word_enc w';
                 return (snoc b t)
 end.
```

⊆

```
Fixpoint Word_fmt {n} {σ}
 : FormatM (Word n) 𝕋 σ :=
 λw. match w with
      | ϵ ⇒ ι
      | b · w' ⇒ t ← Word_fmt w';
                 return (snoc b t)
 end.
```

(WordEnc)

Deriving Encoders

- Encoders derivations pick a particular family member

λs. $t_1 \leftarrow$ Word_enc s.id;
$t_2 \leftarrow \square$;
return ($t_1 \cdot t_2$)
$\subseteq$
Word_fmt $\circ$ id
$\dagger$ Nat_fmt 8 $\circ$ length $\circ$ Payload
$\dagger$ DTime_fmt $\circ$ Timestamp
$\dagger$ List_fmt Word_fmt $\circ$ Payload

$\Uparrow$ WordSeqEnc

λs. $t_2 \leftarrow$ Nat_enc
(length s.Payload);
$t_3 \leftarrow \square$;
return ($t_2 \cdot t_3$)
$\subseteq$
Nat_fmt 8 $\circ$ length $\circ$ Payload
$\dagger$ DTime_fmt $\circ$ Timestamp
$\dagger$ List_fmt Word_fmt $\circ$ Payload

$\Uparrow$ NatSeqEnc

λs. $t_3 \leftarrow$ DTime_enc
s.Timestamp;
$t_4 \leftarrow \square$;
return ($t_3 \cdot t_4$)
$\subseteq$
DTime_fmt $\circ$ Timestamp
$\dagger$ List_fmt Word_fmt $\circ$ Payload

$\Uparrow$ DTSeqEnc

λs. **return** (List_enc
Word_enc
s.Payload)
$\subseteq$
List_fmt Word_fmt $\circ$ Payload

λs. $t_1 \leftarrow$ Word_enc s.id;
$t_2 \leftarrow$ Nat_enc (length s.Payload);
$t_3 \leftarrow$ DTime_enc s.Timestamp;
$t_4 \leftarrow$ List_enc Word_enc s.Payload;
return ($t_1 \cdot t_2 \cdot t_3 \cdot t_4$)

Decoder Combinators

- Decoders for subformats have a particular shape:
  - They're partial: return rest of target value for further parsing
  - Restricted source values
- Bake this into the type of combinators:

```
Record CorrectFormatDecoder {S T Σ_E Σ_D}
(sourcePred : S -> Prop) ≈ (fmt_S : FormatM S T Σ_E):=
{ decode : DecodeM (S * T) T Σ_D;
  decodeOK : fmt_S |_sourcePred ≈ decode }
```

Decoder Combinators

- Can define combinators for our higher-order formats:

$$\frac{\mathrm{P}' \cap \mathrm{format}_1 \approx \mathrm{decode}_1 \qquad \forall s.\ \mathrm{P}\ s \to \mathrm{P}'(\mathrm{f}\ s) \qquad \forall s'.\ \mathrm{P}'\ s' \to \{s \mid \mathrm{f}\ s = s' \wedge \mathrm{P}\ s\} \cap \mathrm{format}_2 \approx \mathrm{decode}_2\ s'}{\mathrm{P} \cap \mathrm{format}_1 \circ \mathrm{f} \mathbin{+\!\!+} \mathrm{format}_2 \approx \mathrm{decode}_1 \mathbin{+\!\!+}_{\mathrm{D}} \mathrm{decode}_2} \quad (\textsc{SeqDec})$$

- The assumptions characterize how to correctly decode a sequence of two formats:
  - Decoder for first format
  - Restrictions on first format are satisfied
  - Decoder for second format, assuming the argument is a projection of original source value

Decoder Combinators

- Can define combinators for our higher-order formats:

$$\frac{}{\mathsf{P} \cap \mathsf{format}_i \approx \mathsf{decode}_i}$$

$$\frac{\forall\, (s, \sigma, t, \sigma') \in \bigcup_i \mathsf{format}_i .\ (s, \sigma, t, \sigma') \in \mathsf{format}_{n(t)}}{\bigcup_i \mathsf{format}_i \approx \lambda t.\ j \leftarrow n\ t;\ \mathsf{decode}_j\, t}\ (\textsc{UnionDec})$$

- The assumptions characterize how to correctly decode the union of several formats:
  - Decoder for each variant
  - Index function n, which correctly determines format used

Decoder Combinators

- Can define combinators for our higher-order formats:

$$\frac{\begin{array}{c} P' \cap \text{format} \approx \text{decode} \\ \forall s.\ P\ s \to P'(f\ s) \qquad \forall s.\ P\ s \to (g \circ f)s = s \\ \forall s.P'\ (f\ s) \to \Big((b \circ g \circ f)\ s = \text{true} \leftrightarrow P\ s\Big) \end{array}}{\begin{array}{l} P \cap \text{format} \circ f \approx \lambda t.\ (s, t') \leftarrow \text{decode } t; \\ \qquad \textbf{if } b \circ g\ s \textbf{ then} \\ \qquad\quad \textbf{return } (g\ s) \\ \qquad \textbf{else fail} \end{array}} \text{(TrmDec)}$$

- The assumptions characterize how to decode a “singleton” format:
  - Decoder for first format
  - Restrictions on first format are satisfied
  - Restriction uniquely determines source value
  - Decision procedure for source restriction

Decoder Combinators

- Consider the final proof state for our simple record format:

$$\frac{\begin{array}{c} P' \cap \text{format} \approx \text{decode} \\ \forall s.\ P\ s \to P'(f\ s) \qquad \forall s.\ P\ s \to (g \circ f)s = s \\ \forall s.P'\ (f\ s) \to \Big((b \circ g \circ f)\ s = \text{true} \leftrightarrow P\ s\Big) \end{array}}{P \cap \text{format} \circ f \approx \begin{array}{l} \lambda t.\ (s, t') \leftarrow \text{decode } t; \\ \quad \textbf{if } b \circ g\ s \textbf{ then} \\ \quad \textbf{return } (g\ s) \\ \quad \textbf{else fail} \end{array}} \text{(TrmDec)}$$

$$\begin{array}{l} \forall\ w\ ln\ dt.\ ln < 2^8 \to \\ \{s \mid s.\text{Timestamp} = dt \wedge \text{length } s.\text{Payload} = ln \wedge s.id = w \\ \qquad \wedge \text{length } s.\text{Payload} < 2^8\} \cap \text{List\_fmt Word\_fmt} \circ \text{Payload} \end{array} \approx \begin{array}{l} \lambda t.\ (l, t') \leftarrow \text{List\_dec Word\_dec } ln\ t; \\ \quad \textbf{return } \{id := w;\ \text{Timestamp} := dt; \\ \qquad \text{Payload} := l\} \end{array}$$

Automating Derivations

- Tricky bit is finding appropriate g and b
- Tactics to the rescue!
- $\square_g$ is an evar for a function which reconstructs the original source value.
- Why not use reflexivity at start?

$H_1 : \ln < 2^8$
$H_2 : s.\text{Timestamp} = dt \wedge \text{length } s.\text{Payload} = \ln$
$\wedge\ s.id = w \wedge \text{length } s.\text{Payload} < 2^8$
==========
$\square_g\ s.\text{Payload} = s$

Destruct the source value, s.

$H_1 : \ln < 2^8$
$H_2 : x_2 = dt \wedge \text{length}(x_3) = \ln \wedge x_1 = w$
$\wedge\ \text{length}(x_3) < 2^8$
==========
$\square_g\ x_3 = \{ id := x_1;$
$Timestamp := x_2;$
$Payload := x_3 \}$

Substitute with equalities in source restriction.

$H_1 : \ln < 2^8$
$H_2 : \ln < 2^8$
==========
$\square_g\ x_3 = \{ id := w;$
$Timestamp := dt;$
$Payload := x_3 \}$

Variant of reflexivity solves the goal. □

Simplifying Specifications

- Narcissus includes a library of common formats
  - Base formats for single data types
  - Higher-order formats for compositional format definitions

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

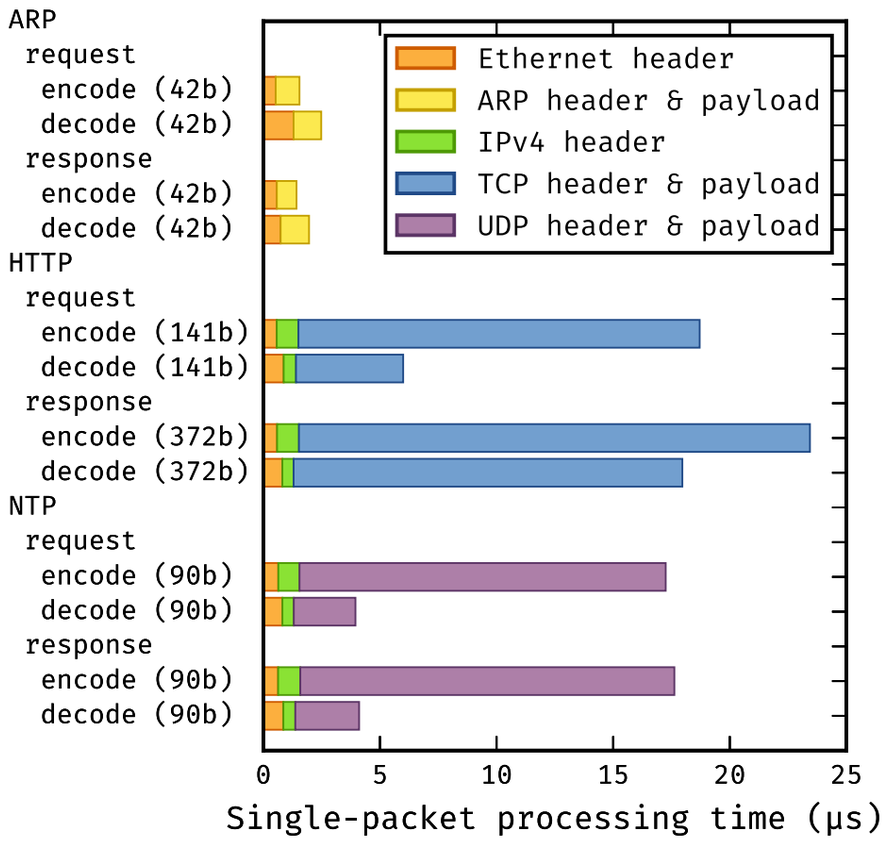
Component Library

Narcissus in Action

- MirageOS is a library operating system for secure, high-performance network applications written in OCaml
- Replaced IP stack of MirageOS with extracted OCaml implementations of synthesized decoders.

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

Derived Decoders



Shameless Plug

- PL @Purdue:
  - 10 faculty across 2 departments
  - Many interests Formal Methods, Synthesis, Compilers, Metaprogramming, PL Theory, Static Analysis, PL+Security, PL+ML …

- Undergrads+Master's students: Please applying!
- Follow us on twitter + facebook:
  - Twitter: @purdue_pl
  - https://www.facebook.com/PurduePL/

Conclusion

- Coq is a powerful tool for automatically deriving correct software
  - Powerful specification language (formats + correctness)
  - Gallina is a pretty good functional language (implementations)
  - Tactics are useful for automating construction of well-understood terms (derivations)
- If you're working on verified distributed systems; let's talk!
  (Happy to talk if you're not working on distributed systems too)

Questions?