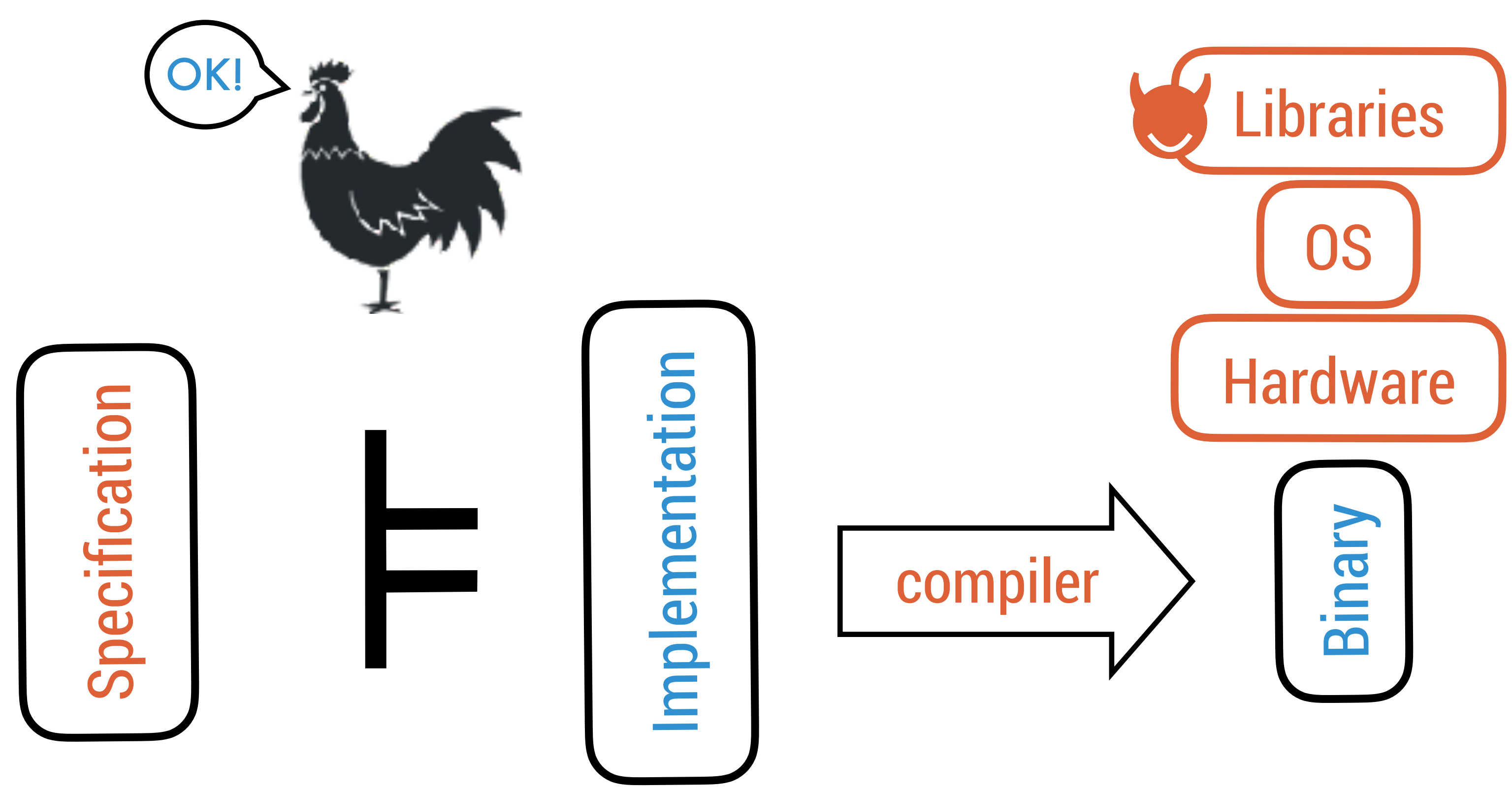
# From Verified Parsers and Serializers to Format-Aware Fuzzers

Benjamin Delaware

Purdue Computer Science

# Formal Verification

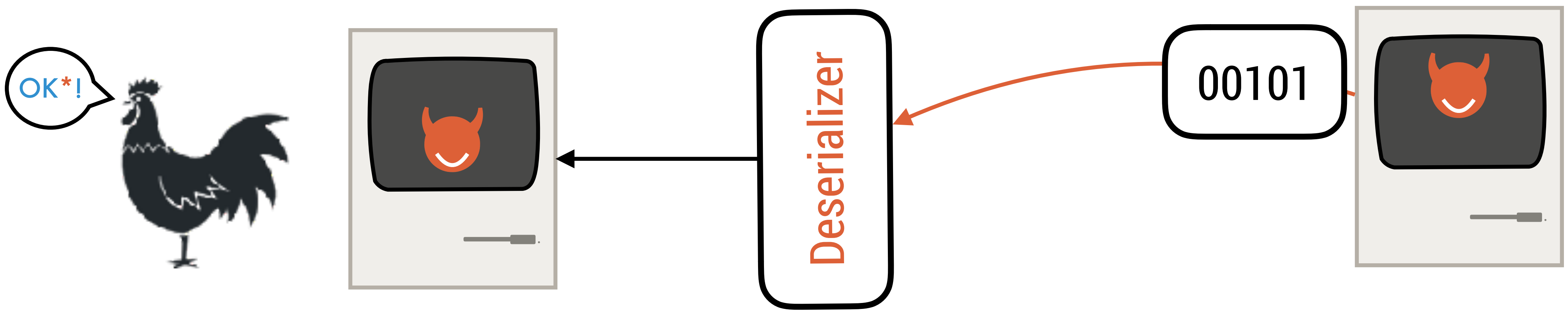
- Numerous developments of high-assurance software in proof assistants in the past five years:
  - CompCert C compiler
  - seL4 microkernel
  - FSCQ file system
- Assurance comes from formal guarantees* provided by proof assistant:



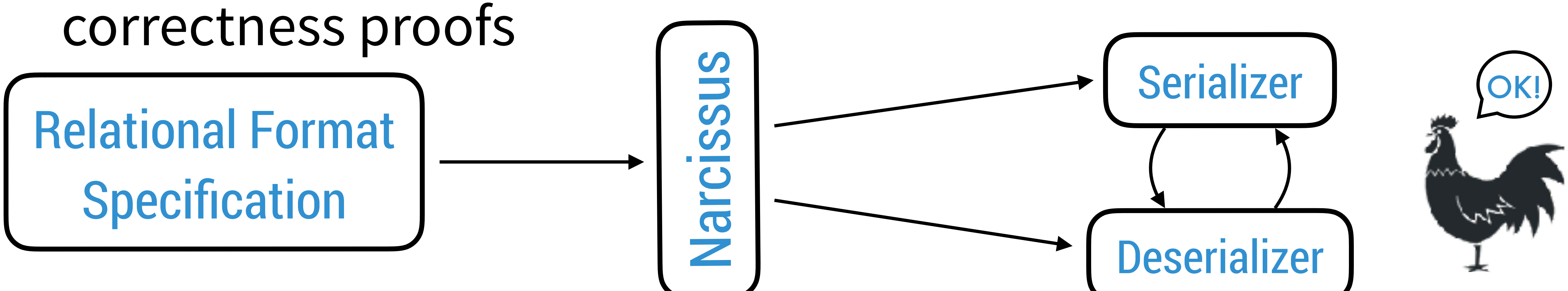

* w.r.t Trusted Base

# Narcissus

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




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



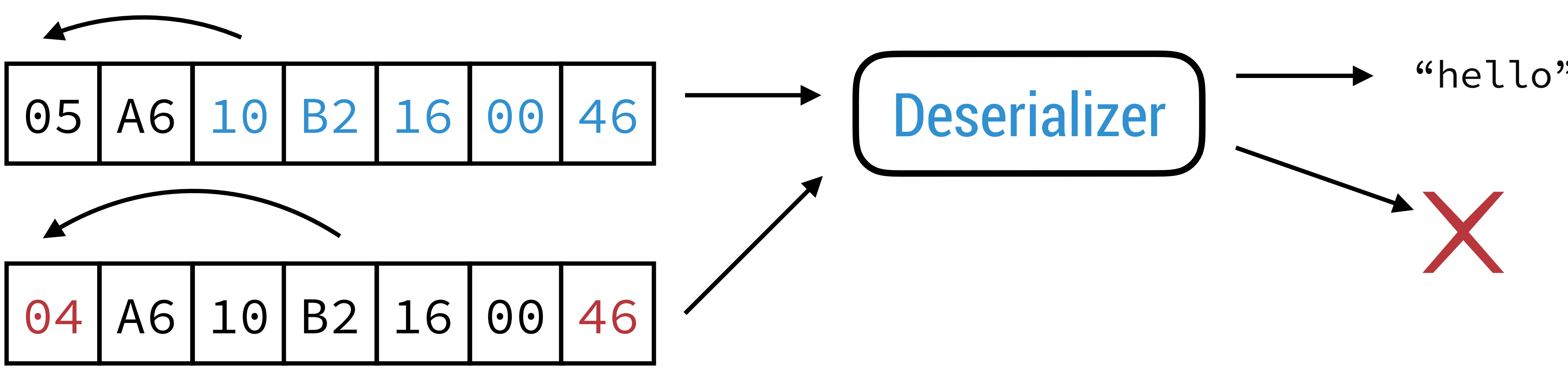

[1] An Empirical Study on the Correctness of Formally Verified Distributed Systems. Pedro Fonseca, Kaiyuan Zhang, Xi Wang, and Arvind Krishnamurthy.

# All Done?

- *Probably* unreasonable to incorporate synthesized decoders and decoders into every existing codebase.
  - Synthesized code is OCaml (working on verified C)
  - Assumes clean interface between communication and processing code
- How to leverage work to secure legacy code?

# From Verification to Fuzzing

- Formats can contain implicit dependencies
- These decoders are provably correct recognizers for the *entire* input format.




- Verification exposes latent dependencies in formats.
- Hypothesis: these dependencies can be leveraged to generate format-aware fuzzers.

# Today’s Talk

- Embedding Formats in Narcissus
- Synthesizing Correct-by-Construction encoders and decoders
- Leveraging these to generate format-aware fuzzers

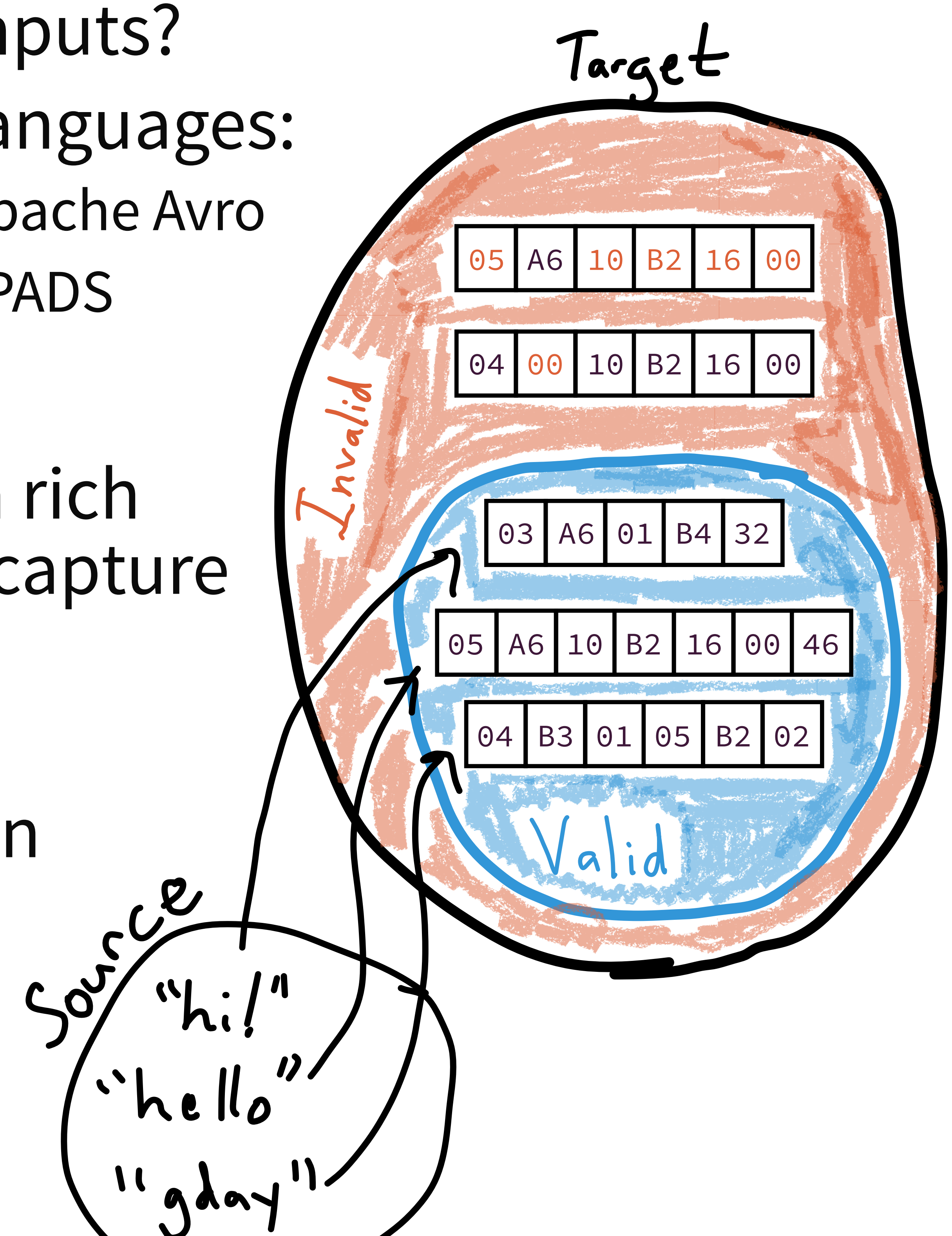
# Specifying Formats in Narcissus

- **First challenge:** specifying valid inputs?
- Established format specification languages:
  - Interface Generators: ASN.1, Protobuffs, Apache Avro
  - Format Specification Languages: binpac, PADS

- Internet servers were the original verification target, so we needed a rich enough specification language to capture legacy formats.

- **Solution (?):** functional description

```
format(s) = |s| ++ 166 ++ s
```



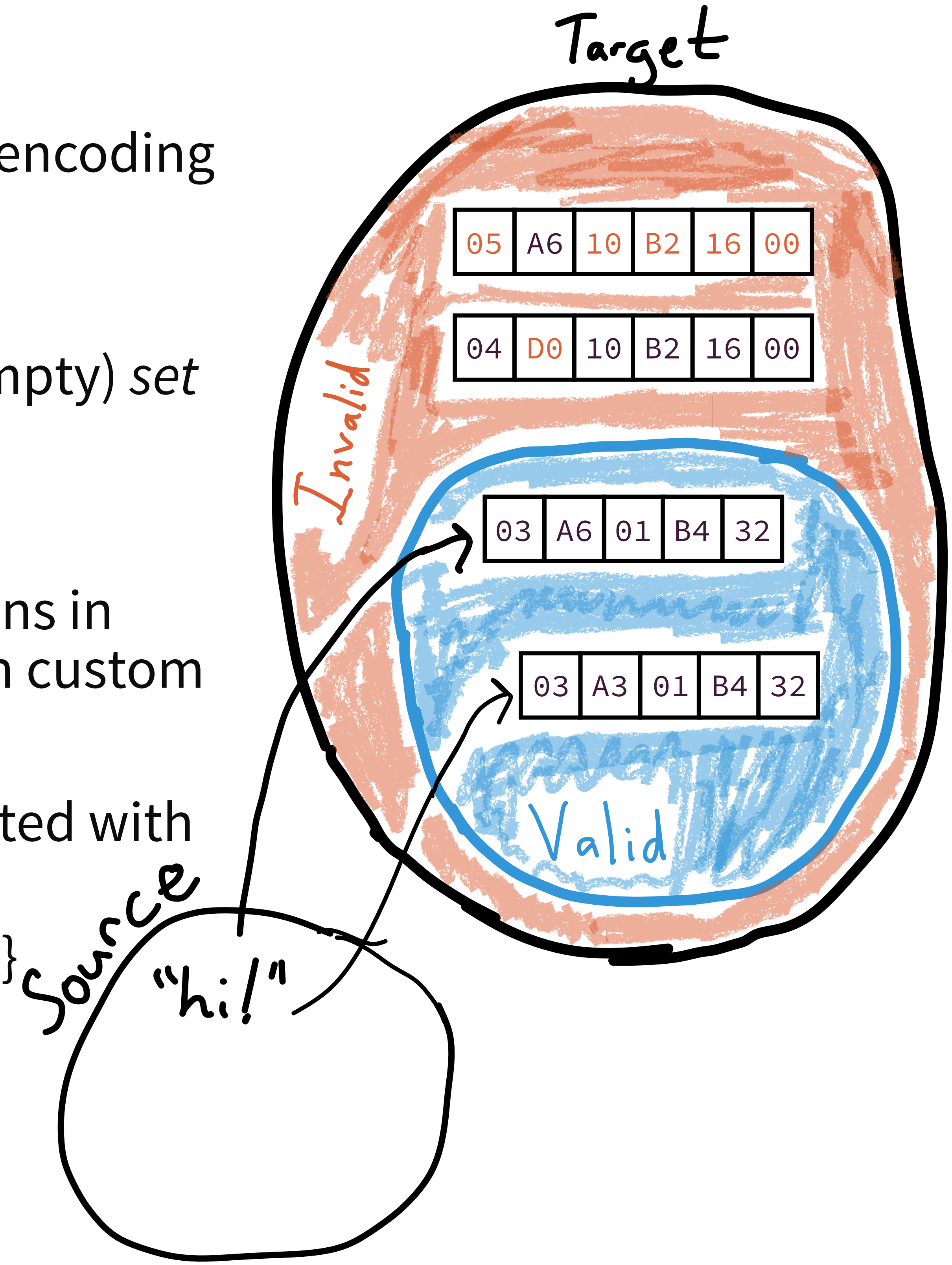
# Relational Specifications

- Many formats do not have a single canonical encoding of a source value
  - i.e. DNS packet compression
- **Solution**: map source values to a (possibly empty) *set* of target representations:

$$\text{format}(s) = |s| \mathbin{+\!\!\!+} \{n \mid n \leq 2^{17}\} \mathbin{+\!\!\!+} s$$

- These relations are represented as propositions in Coq's logic, so users can freely write their own custom format specifications
- Constraints on source values can be represented with set intersection:

$$\text{format}'(s) = \text{format}(s) \cap \{(s,t) \mid |s| \leq 2^{17}\}$$



# Simplifying Specifications

- Narcissus includes a library of common formats
  - Base formats for single data types
  - Combinators for composing formats

| Format | LoC | LoP | Higher-order |
|---|---|---|---|
| Sequencing (++) | 7 | 164 | Y |
| Termination (ε) | 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

# Simplifying Specifications

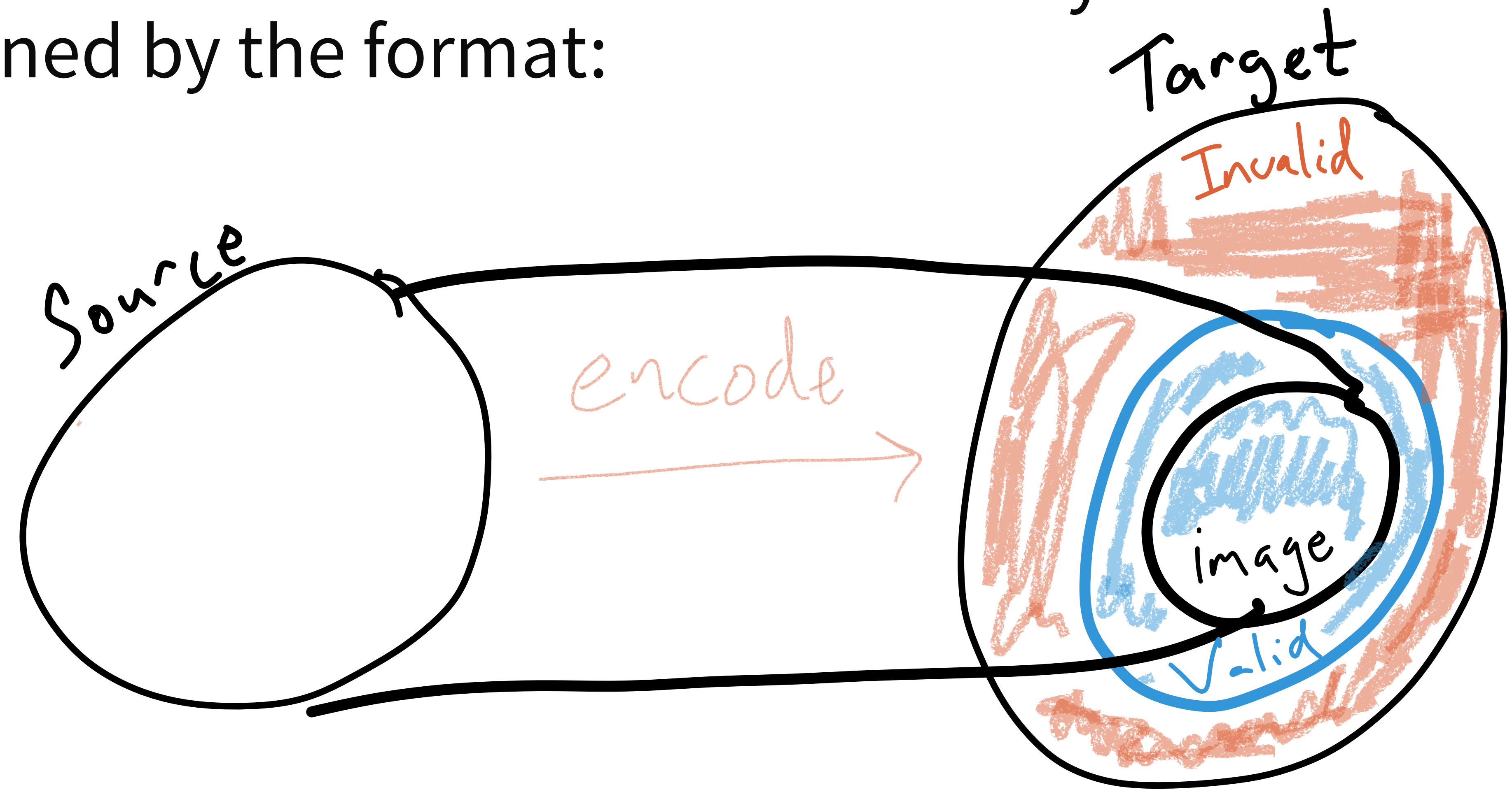
- Narcissus includes a library of common formats
  - <u>Base formats</u> for single data types
  - <u>Combinators</u> for composing formats

```
Definition IPv4_Packet_Format (ip4 : IPv4_Packet) :=
   format_nat 4 4 ++ format_nat 4 (5 + |ip4.Options|) ++ {n : char | true}
 ++ format_word ip4.TotalLength
 ++ format_word ip4.ID
 ++ {b : bool | true} ++ format_bool ip4.DF ++ format_bool ip4.MF ++ format_word ip4.FragmentOffset
 ++ format_word ip4.TTL ++ format_enum ProtocolCodes ip4.Protocol
 ++ IPChecksum_Valid
 ++ format_word ip4.SourceAddress
 ++ format_word ip4.DestAddress
 ++ format_list format_word ip4.Options ++ e.
```

| 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 | | | | | | | |

# Specifying Encoders and Decoders

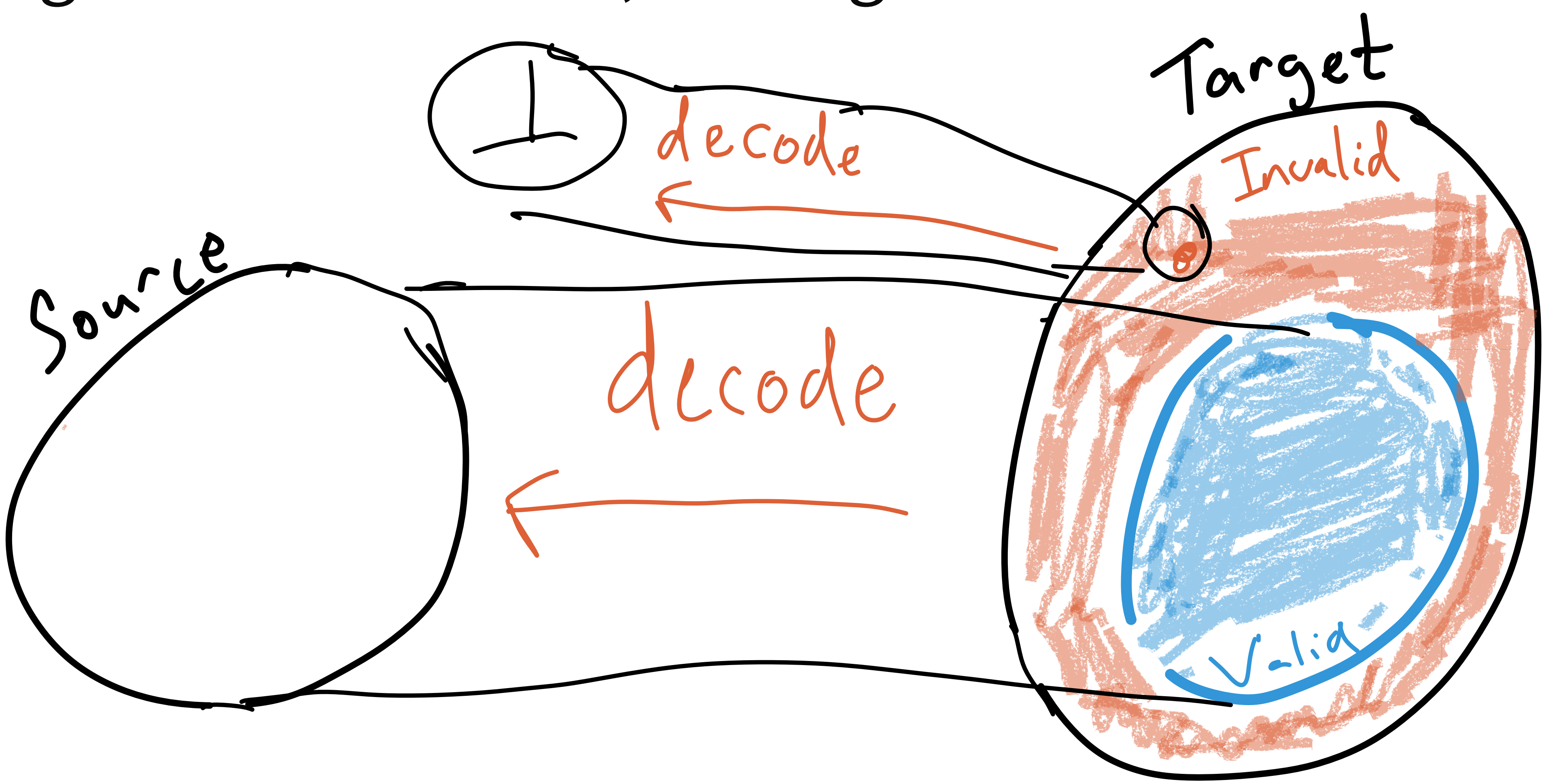
- A correct encoder is a function wholly contained in the relation defined by the format:




```
EncoderOK(Format, e) ≡ ∀s.Format ∋ (s, e(s))
```

# Specifying Encoders and 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




```
DecoderOK(Format, d) ≡ ∀t.Format ∋ (d(t), t)
                      ∧ d(t) = ⊥ → ∀v. Format ∌ (v, t)
```

# Deriving Encoders

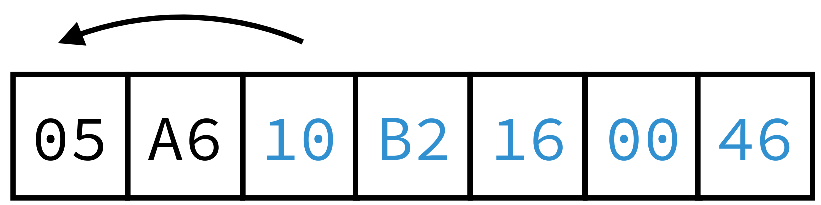
- Can phrase construction of a correct encoder as a user directed search for a function satisfying `EncoderOK`
  - Such searches are the bread and butter of theorem provers
- Key Observation: formats are inherently compositional, so this process can be decomposed into a series of small steps

```
format'(s) := {|s|} ⧺ {n | n ≤ 2^17} ⧺ {s} ∩ {(s,t) | |s| ≤ 2^32}
  ⊒ {|s|} ⧺ {0} ⧺ {s} ∩ ∩ {(s,t) | |s| ≤ 2^32}
  ⊒ {|s| ++ 0} ⧺ {s} ∩ {(s,t) | |s| ≤ 2^32}
  ⊒ {|s| ++ 0 ++ s} ∩ {(s,t) | |s| ≤ 2^32}
  ∋ if |s| ≤ 2^32 then |s| ++ 0 ++ s
```


- These proofs can be automated

# Deriving Decoders

- Can do the same for decoders, but correctness of subdecoders now depends on other parts of the encoded value:



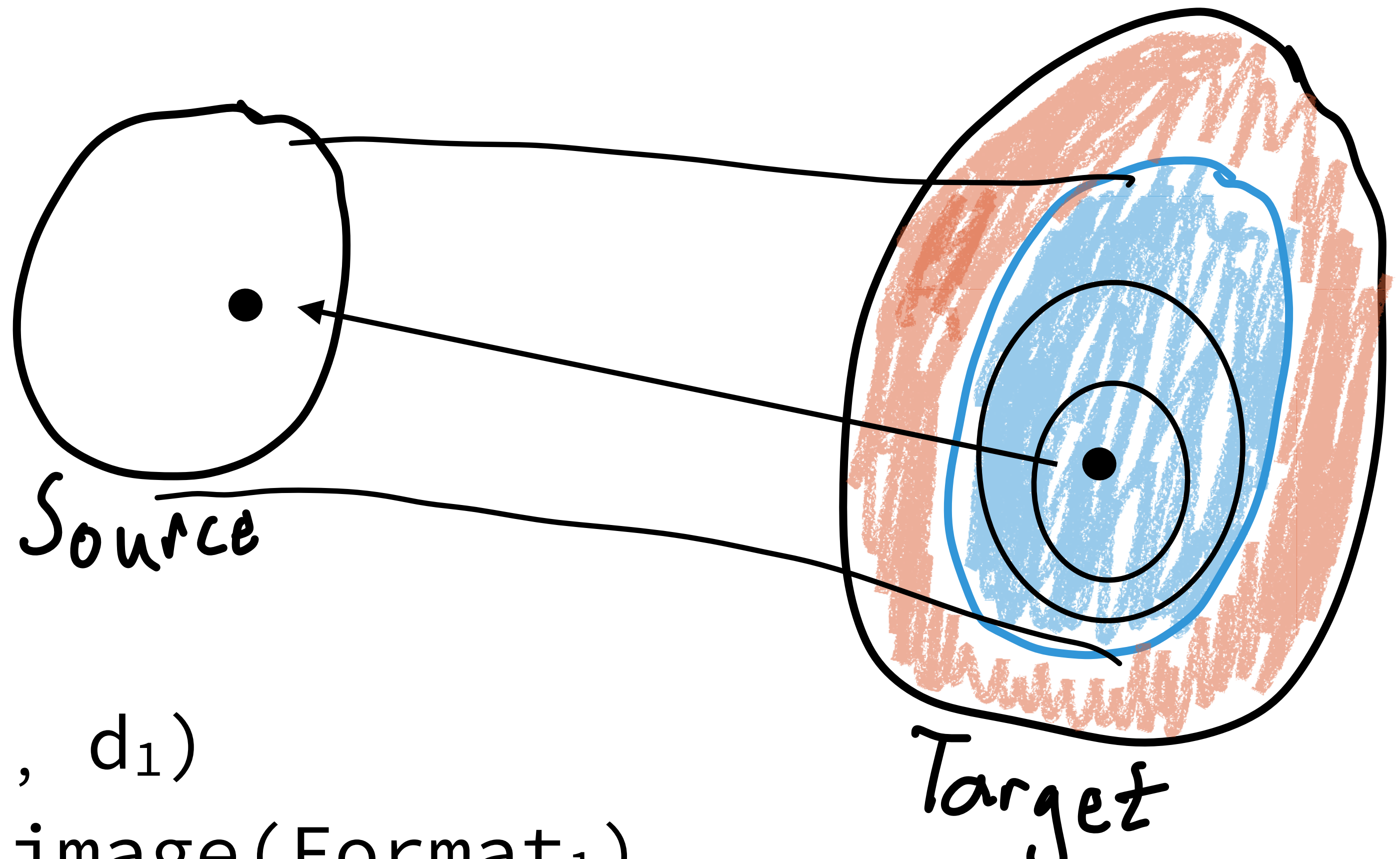

- DNS— compressed domains are pointers
- DNS— resource record tag determines how payload is parsed
- SDN— versions effects available options
- ZIP— position of start of central directory depends on EOCD

```
∀n. DecoderOK({s} ∩ {(s,t) | |s| = n}, decodeList n)

where decode 0 [] = Some []
      decode n (c : t) = decode (n - 1) t >>= \l -> c : l
      decode _ _   = None
```

# Deriving Decoders$^2$

- <u>Key idea</u>: keep track of dependence data when decomposing proof:




$$
\begin{aligned}
& \mathtt{DecoderOK}(\mathtt{Format}_{1'},\ d_1) \\
\wedge\ & \mathtt{image}(\mathtt{Format}_{1'}) = \mathtt{image}(\mathtt{Format}_1) \\
\wedge\ & \mathtt{DecoderOK}(\mathtt{Format}_2 \cap \{(s,t) \mid \exists t'.\ (v,\ t') \in \mathtt{Format}_{1'} \\
& \qquad\qquad\qquad\qquad\qquad\qquad \wedge\ \ (s,\ t') \in \mathtt{Format}_1\}, \\
& \qquad\qquad d_2(v)\ ) \\
& \to \mathtt{DecoderOK}(\mathtt{Format}_1 \mathbin{+\!\!+} \mathtt{Format}_2,\ d_1 \mathbin{>\!\!>\!=} d_2)
\end{aligned}
$$

# Deriving Decoders$^2$

- <u>Key idea</u>: keep track of dependence data when decomposing proof:

```
DecoderOK({|s|} ++ {n | n ≤ 2^17} ++ {s} ∩ {(s,t) | |s| ≤ 2^32}, ?)
```

↖

```
DecoderOK({n | n ≤ 2^17} ++ {s} ∩ {(s,t) | |s| ≤ 2^32} ∩ {v = |s|}, ? v)
```

↖

```
DecoderOK({s} ∩ {(s,t) | |s| ≤ 2^32} ∩ {v = s} ∩ {n ≤ 2^17}, ? v n)
```

↖

```
DecoderOK({(s,t) | |s| ≤ 2^32} ∩ {v = |s|} ∩ {n ≤ 2^17} ∩ {l = s}, ? v n l)
```

↖

```
DecoderOK({(s,t) | |s| ≤ 2^32 ∧ v = |s| s ∧ ≤ 2^17 ∧ l = s}, l)
```

# Deriving Decoders[2]

- Key idea: keep track of dependence data when decomposing proof:

```
DecoderOK({|s|} ⧺ {n | n ≤ 2¹⁷} ⧺ {s} ∩ {(s,t) | |s| ≤ 2³²},
          v <- decodeChar;
          n <- decodeChar;
          l <- decodeList v;
          if n <= 2¹⁷ then return l else None)
```

# Narcissus in Action

- MirageOS is a library operating system for secure, high-performance network applications written in OCaml
- Replaced network stack of MirageOS with extracted OCaml implementations of synthesized decoders.
- Found one problem in the test suite.

| **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

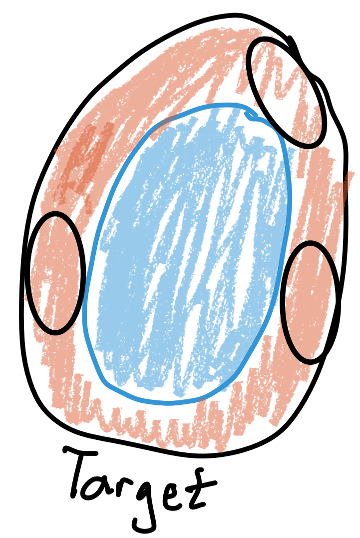
- But, *probably* unreasonable to incorporate synthesized decoders and decoders into every existing codebase.
- How can we leverage this to secure legacy systems?

# Towards Format-Aware Fuzzers

- The final decoder synthesis step contains the accumulated dependencies embedded in the format:

```
DecoderOK({(s,t) | |s| ≤ 2^32 ∧ n ≤ 2^17 ∧ v = |s| ∧ l = s}, ?)
```

  - invariants on the original input data
  - invariants on the shape of the target values
  - dependencies between bytes of the target values

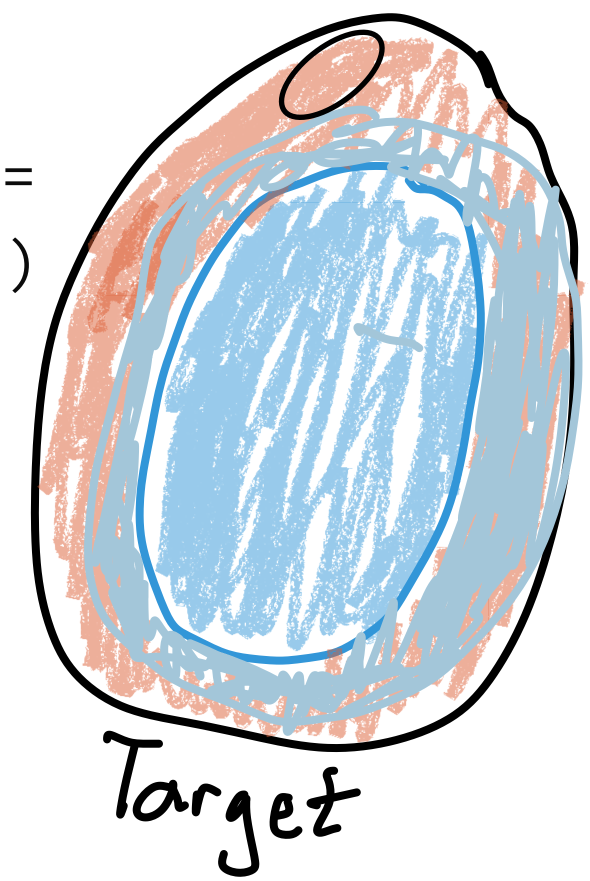



- Idea: violating any one of these these dependencies yields an input not included in the format
- Can we selectively break these dependencies to "fuzz" the format in a smart way?
- Generate predicates for behavioral property testing?

# Gradual Fuzzing

- We don’t need to formalize the full format to get useful fuzzers:
  - Only specifying certain fields tests dependencies between these fields
  - Rest of the target value is “don’t care” bits:

```
Definition IPv4_Packet_Format (ip4 : IPv4_Packet) :=
   format_nat 4 4 ++ format_nat 4 (5 + |ip4.Options|)
 ++ {n : char | true}
 ++ {n : 16 words | true}
 ++ format_list format_word ip4.Options ++ e.
```




- Gradually specify complex formats, hitting low-hanging bits first

# Conclusion

- Today's talk:
  - Embedding Formats in Narcissus
  - Synthesizing Correct-by-Construction encoders and decoders
  - Leveraging these to generate format-aware fuzzers

Thoughts?

# Conclusion

- Today’s talk:
  - Embedding Formats in Narcissus
  - Synthesizing Correct-by-Construction encoders and decoders
  - Leveraging these to generate format-aware fuzzers

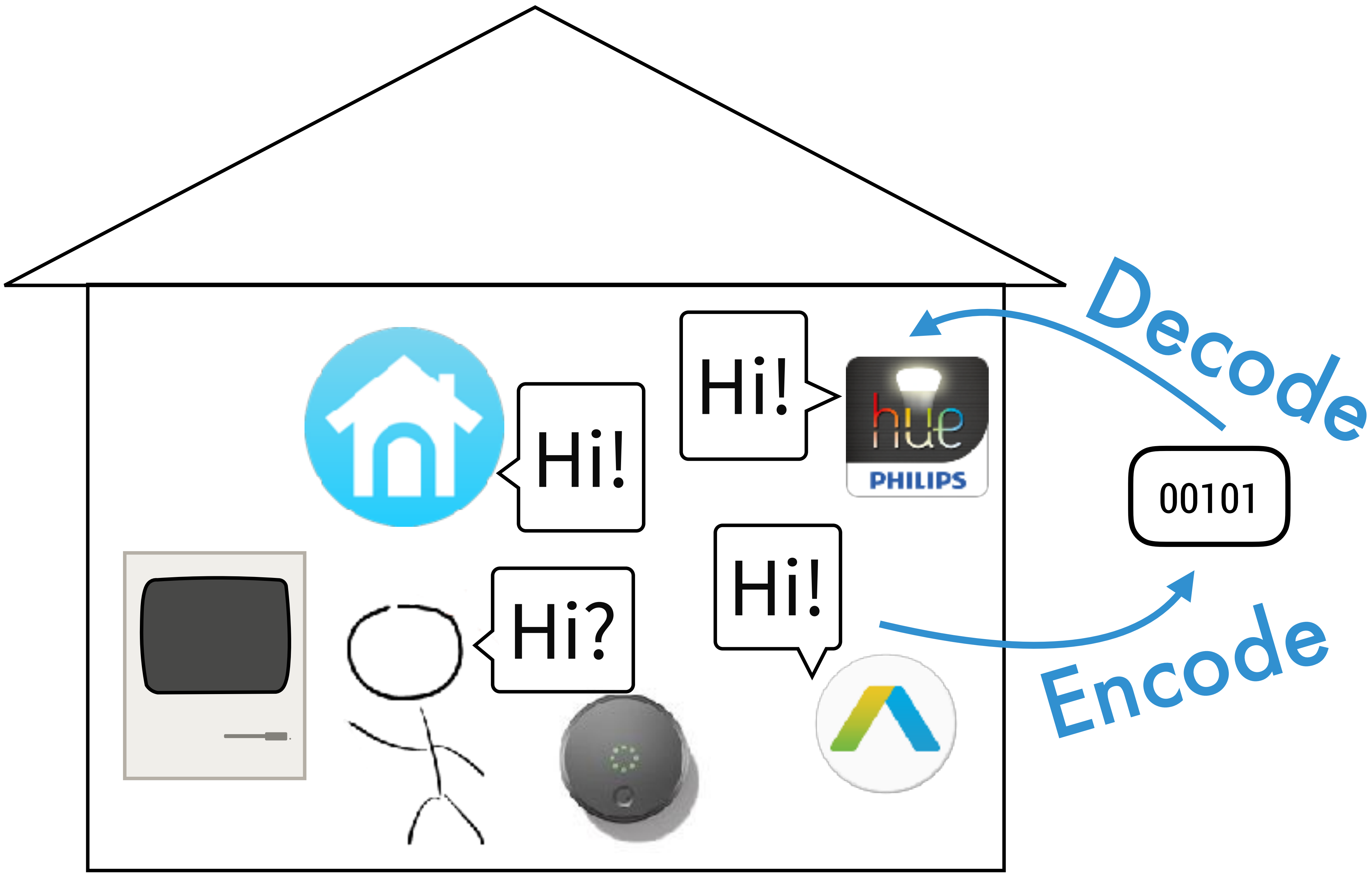
- Next Steps:
  - Evaluation?
  - Thoughts?

# Conclusion

- Today's talk:
  - Embedding Formats in Narcissus
  - Synthesizing Correct-by-Construction encoders and decoders
  - Leveraging these to generate format-aware fuzzers

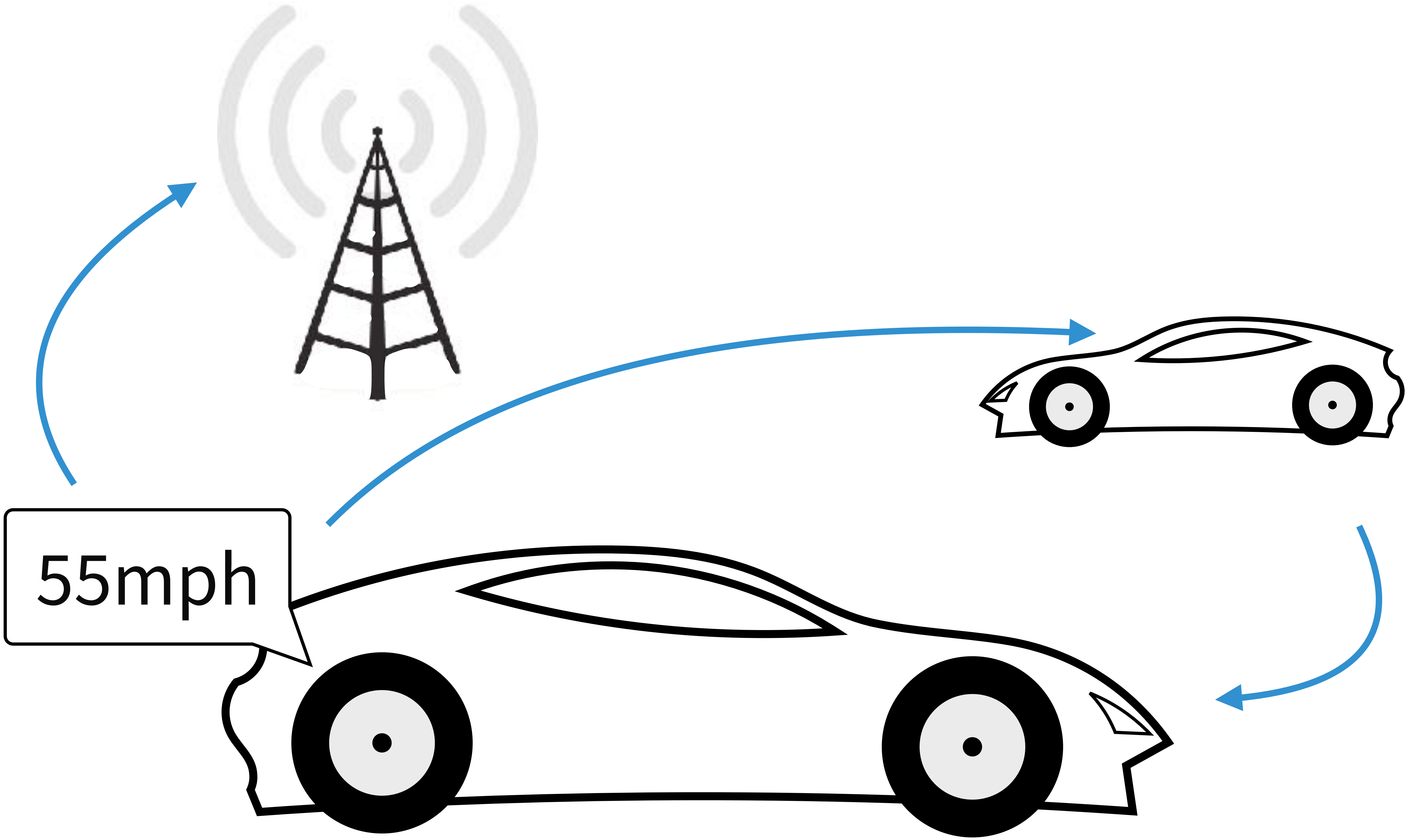
- Next Steps:
  - Evaluation?
  - Thoughts?

# Computers are Multiplying



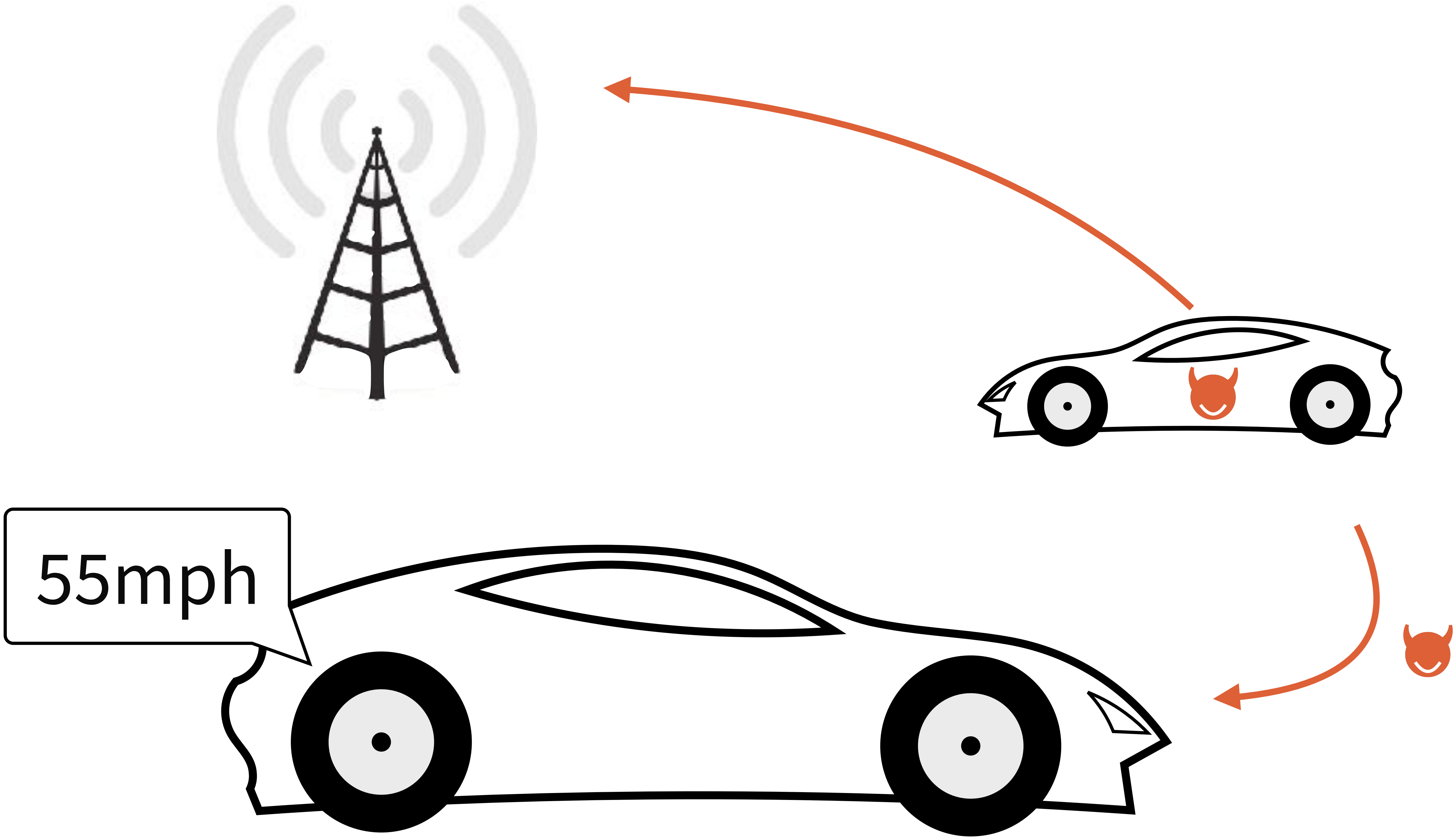

- Relationship between encoded + decoded data important
- Bugs lead to miscommunication

# Communication is Multiplying




- Decoders present attack surface for malicious packets
- NTSB will likely mandate V2V communication within the next decade

# Communication is Multiplying




- Decoders present attack surface for malicious packets
- NTSB will likely mandate V2V communication within the next decade

# Why Worry?

Since 2013:

And Many More!

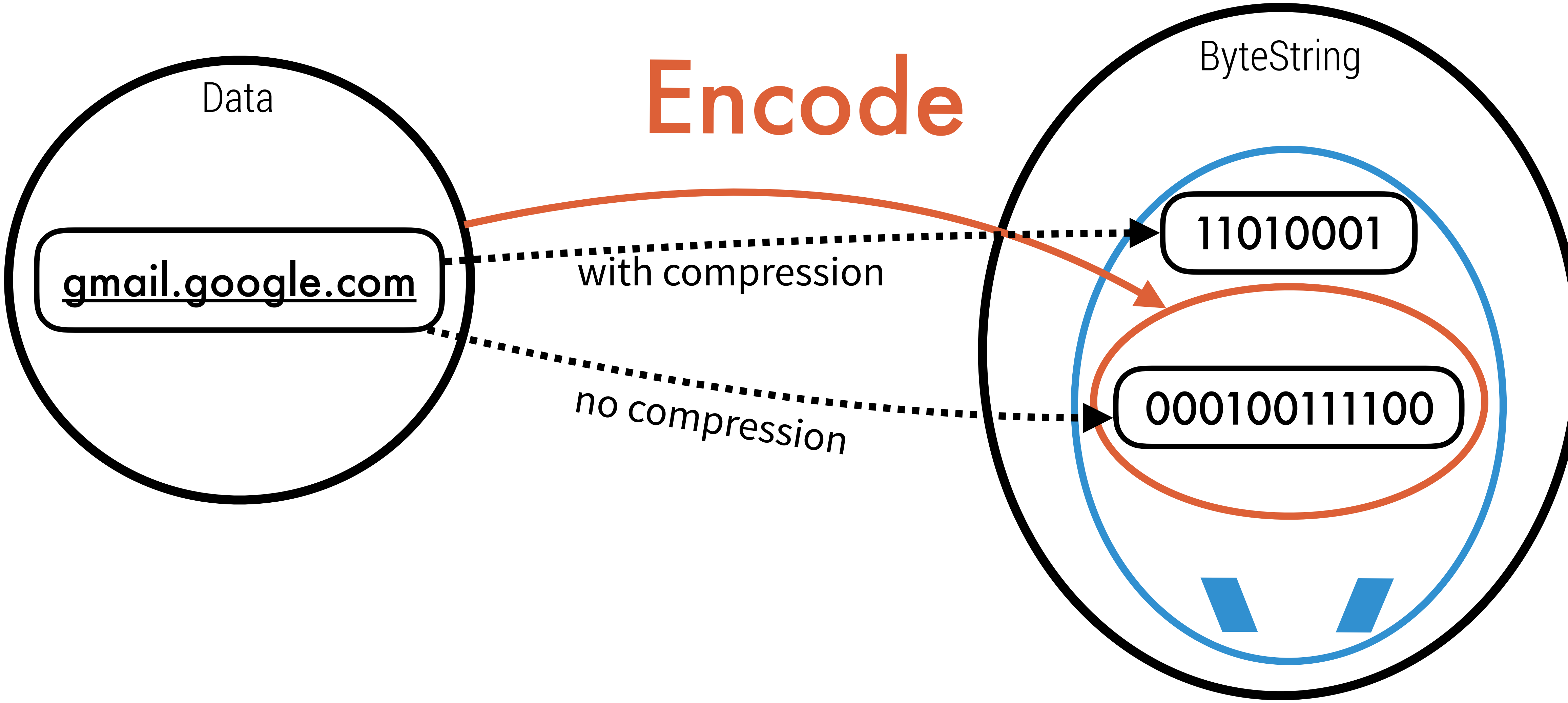
# Established Solutions

- Interface Generators:
  - ASN.1, Protobuffs, Apache Avro
  - Data format defined by system
- Format Specification Languages:
  - binpac, PADS, Packet Types
  - New formats still require modifying code generator
- User-Extensible Systems:
  - Nail
  - No formal guarantees

# Today's Talk

- Narcissus:
  - Framework for synthesizing encoders and decoders from formats
  - User extensible
  - Correct-by-Construction
- Generating performant code
- Integration into high-assurance systems

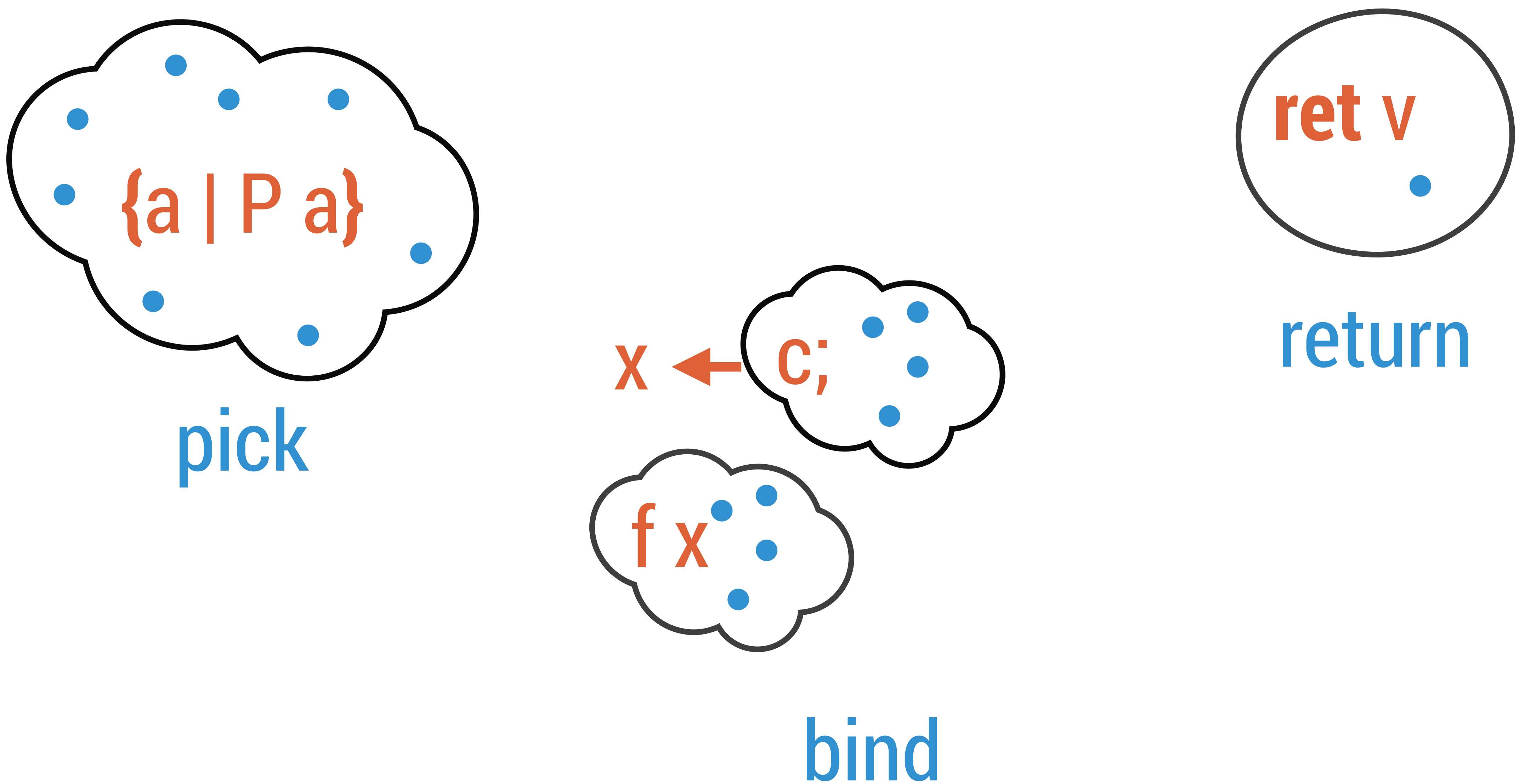
# Specifying Formats




- Mapping is not one-to-one: compression, unspecified fields
- Key Idea: specify set of valid encodings for value as a binary relation
- Encoder always maps into valid set

# Specifying Formats

**Key Idea:** Represent formats as functional programs in the nondeterminism monad.



# Computations

**Key Idea:** Represent formats as functional programs in the nondeterminism monad.

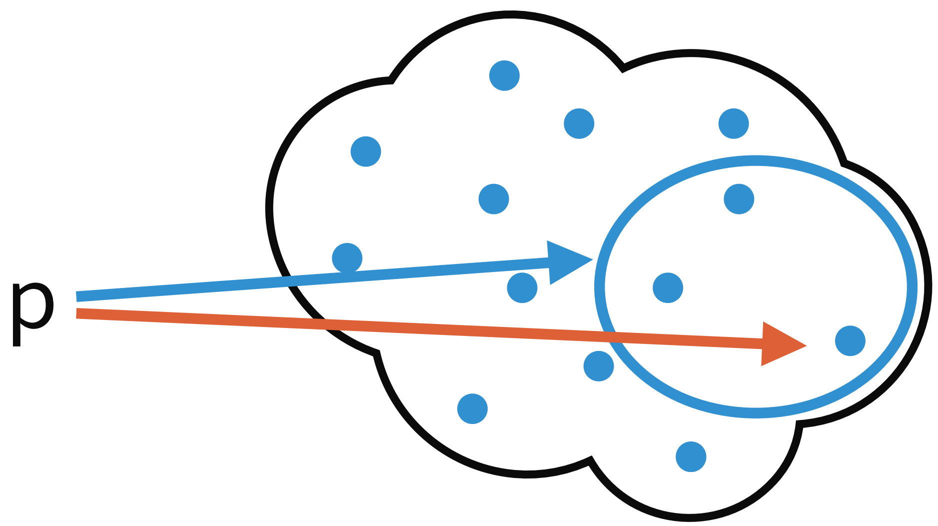
```
  0   1   2   3   4   5   6   7   8
+---+---+---+---+---+---+---+---+---+
|          NUMREADINGS               |
+---+---+---+---+---+---+---+---+---|
|                                    |
/               ID                   /
|                                    |
+---+---+---+---+---+---+---+---+---+
|       CLASS          |0  |0  |0  |
+---+---+---+---+---+---+---+---+---+
|                                    |
/             READINGS               /
|                                    |
+---+---+---+---+---+---+---+---+---+
```

```
Packet := ⟨ID :: string,
           readings :: list word⟩

SimpleFormat (p : Packet) :=
 b1 ← formatNat |p!readings|;
 b2 ← formatString p!ID;
 b3 ← {w : word | w < 32};
 b4 ← formatList encodeWord p!readings;
 ret (b1++b2++b3++b4)
```

# Specifying Correct Encoders

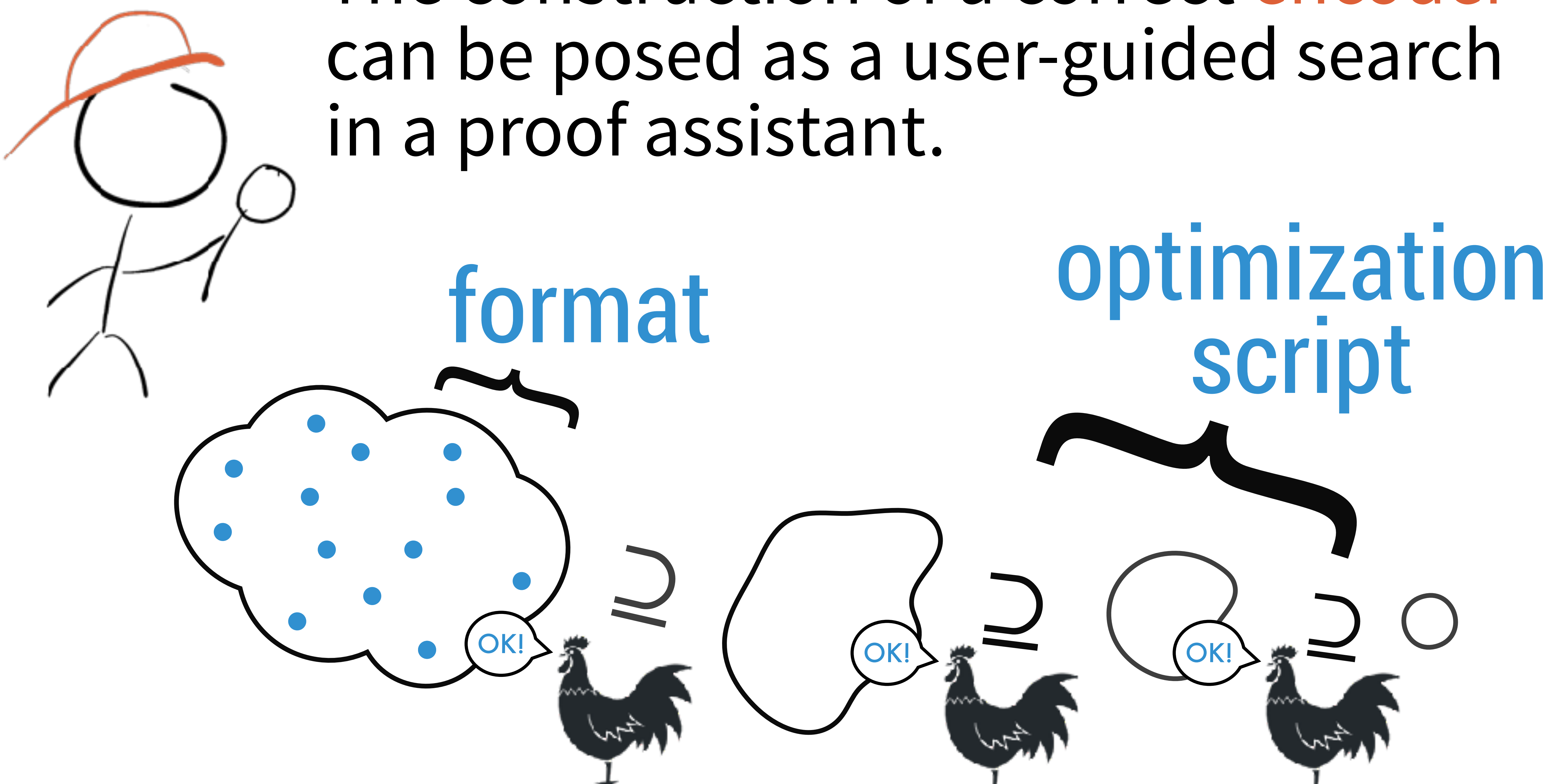
A correct encoder is a function wholly contained in the relation defined by the format.




$\forall$p. `SimpleFormat(p)` $\ni$ `SimpleEncoder(p)`

# Deriving Correct Encoders

The construction of a correct encoder can be posed as a user-guided search in a proof assistant.



# Properties of Refinement

- Preorder

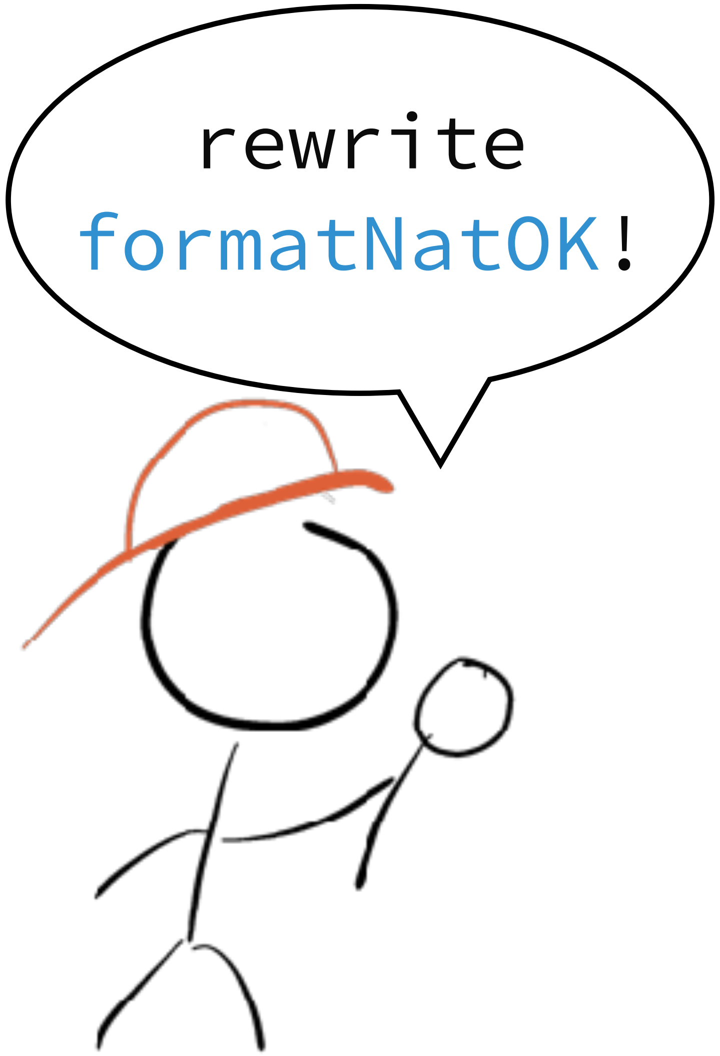
$$\frac{a \supseteq b \qquad b \supseteq c}{a \supseteq c}\ \textsc{Trans}_{\supseteq} \qquad\qquad \frac{}{a \supseteq a}\ \textsc{Refl}_{\supseteq}$$

- Respected by sequencing

$$\frac{a \supseteq b}{r \leftarrow a; f(r) \ \supseteq\ r \leftarrow b; f(r)}\ \textsc{Seq1}_{\supseteq}$$

$$\frac{\forall r, f(r) \supseteq f'(r)}{r \leftarrow a; f(r) \ \supseteq\ r \leftarrow a; f'(r)}\ \textsc{Seq2}_{\supseteq}$$

# Deriving Correct Encoders




```
SimpleFormat (p : Packet) :=
 b₁ ← formatNat |p!readings|;
 b₂ ← formatString p!ID;
 b₃ ← {w : word | w < 32};
 b₄ ← formatList encodeWord p!readings;
 ret (b₁++b₂++b₃++b₄)
```

⊇

```
SimpleFormat (p : Packet) :=
 b₁ ← encodeNat |p!readings|;
 b₂ ← formatString p!ID;
 b₃ ← {w : word | w < 32};
 b₄ ← formatList encodeWord p!readings;
 ret (b₁++b₂++b₃++b₄)
```

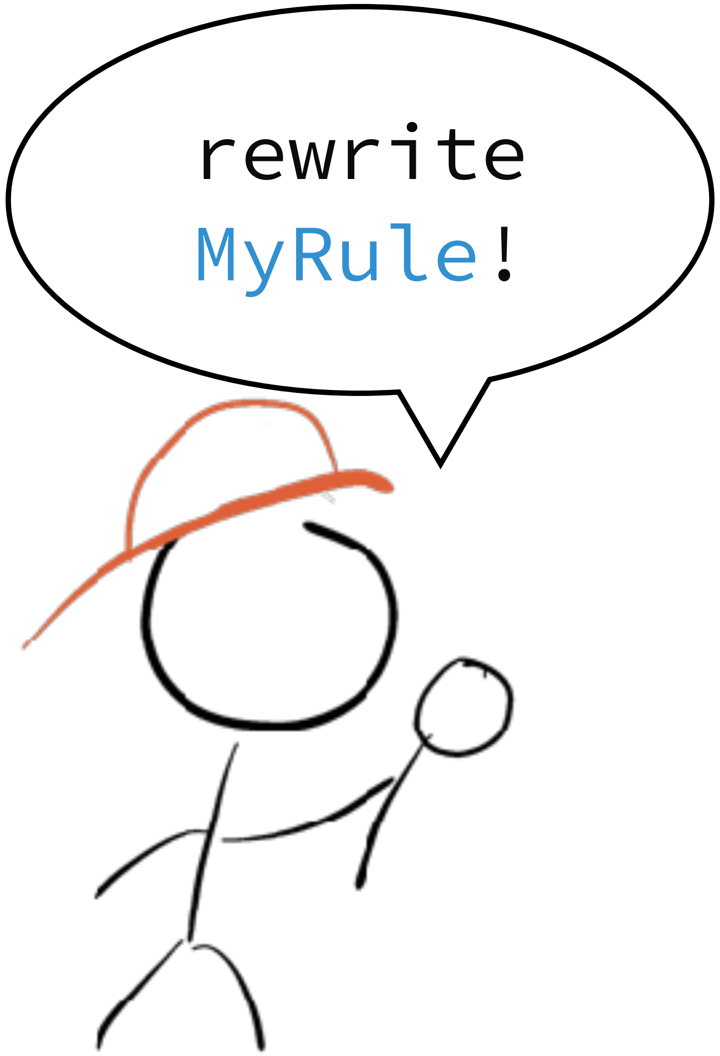
# Deriving Correct Encoders


```
SimpleFormat (p : Packet) :=
 b1 ← encodeNat |p!readings|;
 b2 ← formatString p!ID;
 b3 ← {w : word | w < 32};
 b4 ← formatList encodeWord p!readings;
 ret (b1++b2++b3++b4)
```

$\supseteq$

```
SimpleFormat (p : Packet) :=
 b1 ← encodeNat |p!readings|;
 b2 ← encodeString p!ID;
 b3 ← {w : word | w < 32};
 b4 ← formatList encodeWord p!readings;
 ret (b1++b2++b3++b4)
```

# Deriving Correct Encoders




```
SimpleFormat (p : Packet) :=
 b₁ ← encodeNat |p!readings|;
 b₂ ← encodeString p!ID;
 b₃ ← {w : word | w < 32};
 b₄ ← formatList encodeWord p!readings;
 ret (b₁++b₂++b₃++b₄)
```

⊇

```
SimpleFormat (p : Packet) :=
 b₁ ← encodeNat |p!readings|;
 b₂ ← encodeString p!ID;
 b₃ ← ret 0;
 b₄ ← formatList encodeWord p!readings;
 ret (b₁++b₂++b₃++b₄)
```

# Deriving Correct Encoders

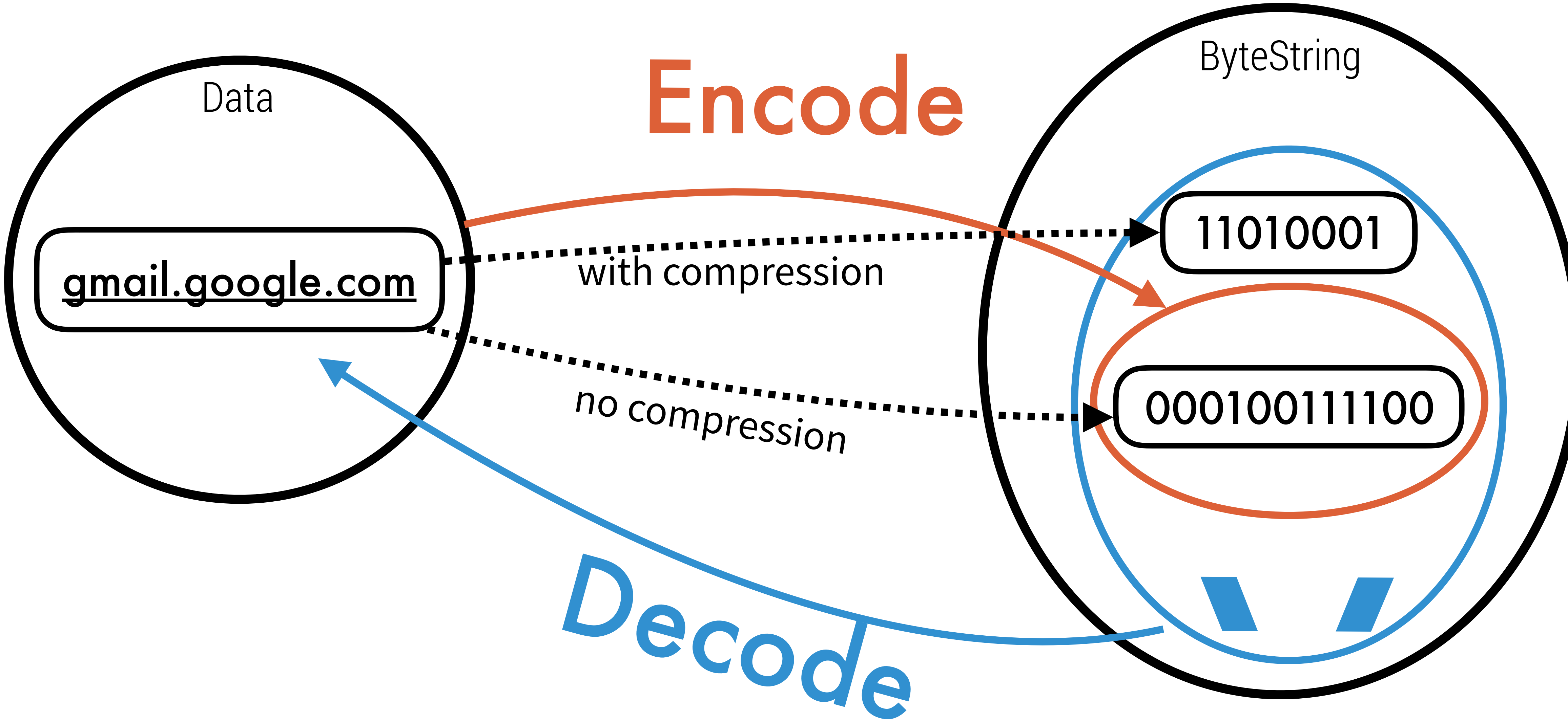

```
SimpleFormat (p : Packet) :=
 b₁ ← encodeNat |p!readings|;
 b₂ ← encodeString p!ID;
 b₃ ← ret 0;
 b₄ ← formatList encodeWord p!readings;
 ret (b₁++b₂++b₃++b₄)
```

⋓

```
SimpleEncoder (p : Packet) :=
encodeNat |p!readings| ++ encodeString p!ID ++ 0
++ encodeList encodeWord p!readings
```

- Users can safely add their own formats and rewrite rules
- Rewrites can be packaged together into single optimization tactic

# Specifying Correct Decoders




$$\text{Valid}^{-1}\ b\ P \triangleq \{\ p \mid b \in \text{Valid}\ p \wedge P\ p \wedge \neg\exists\ p.\ b \in \text{Valid}\ p \wedge P\ p \rightarrow p = \bot\ \}$$

# Deriving Correct Decoders

The construction of a correct decoder can **also** be posed as a user-guided search in a proof assistant.

Component Library

$$\frac{}{\mathtt{formatNat}^{-1}\ b\ \mathbb{T} \ni \mathtt{decodeNat}(b)}$$

$$\frac{}{\mathtt{formatString}^{-1}(b)\ \mathbb{T} \ni \ldots}$$

Invariant on List Elements

Know Length

$$\frac{\mathtt{format}_A^{-1}\ b\ P_A \ni \mathtt{decode}_A(b) \qquad \mathbf{Q}(l) \rightarrow \forall a \in l.\ P_A(a) \qquad \mathbf{Q}(l) \rightarrow |l| = n}{\mathtt{formatList}^{-1}\ \mathtt{format}_A\ b\ \mathbf{Q} \ni \mathtt{decodeList}\ \mathtt{decode}_A(b,\ n)}$$

# Deriving Correct Decoders

The construction of a correct decoder can **also** be posed as a user-guided search in a proof assistant.

Component Library

$$\frac{\mathrm{format}_A{}^{-1}\ b\ P_A \ni \mathrm{decode}_A(b) \qquad \forall a.\ P_A(a) \to \mathrm{format}_B{}^{-1}\ b\ Q \ni \mathrm{decode}_B(a, b) \qquad \mathbf{Q}(ab) \to P_A(\pi\ ab)}{\mathrm{format}_A;\ \mathrm{format}_B{}^{-1}\ b\ \mathbf{Q} \ni (b', a) \leftarrow \mathrm{decode}_A\ b\ P_A;\ \mathrm{decode}_B\ (a, b')\ \mathbf{Q}}$$

$$\frac{\forall a'.\ \mathbf{Q}(a') \to a' = a}{\mathbf{ret}\ [\,]^{-1}\ b\ \mathbf{Q} \ni \mathrm{Some}\ a}$$

# Deriving Correct Decoders


```
SimpleDecoder (b : ByteString) :=
SimpleFormat⁻¹ b 𝕋
```

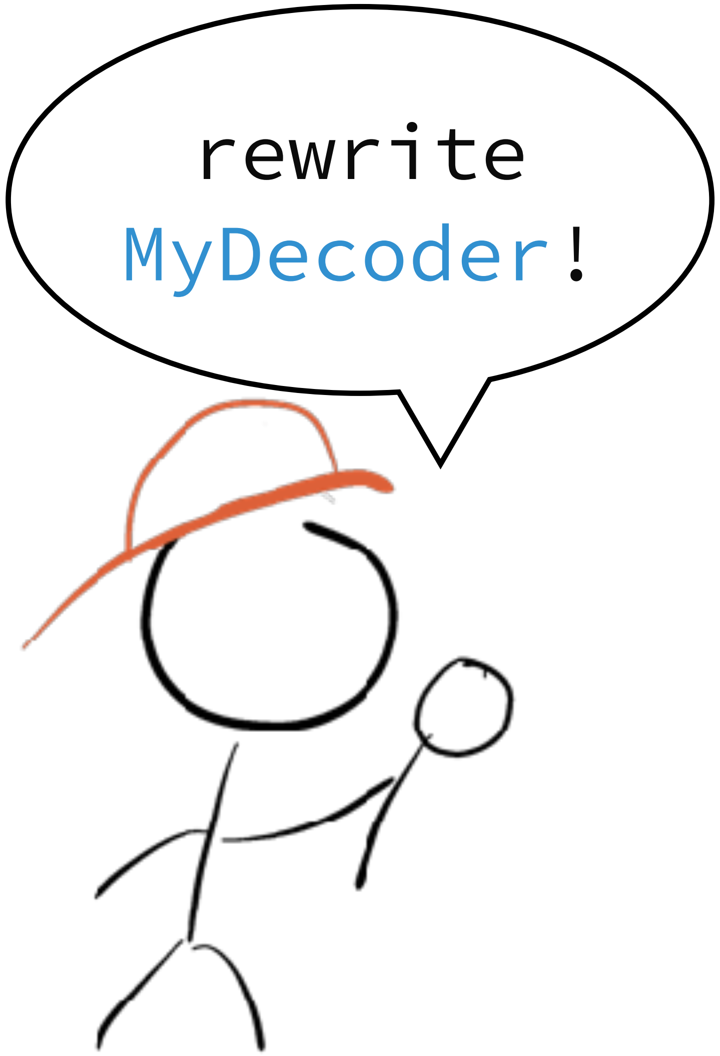
$\supseteq$

```
SimpleDecoder (b : ByteString) :=
  (n, b) ← decodeNat(b); ???
```

$\supseteq$

```
SimpleDecoder (b : ByteString) :=
  (n, b) ← decodeNat(b);
  (s, b) ← decodeString(b); ???
```

# Deriving Correct Decoders



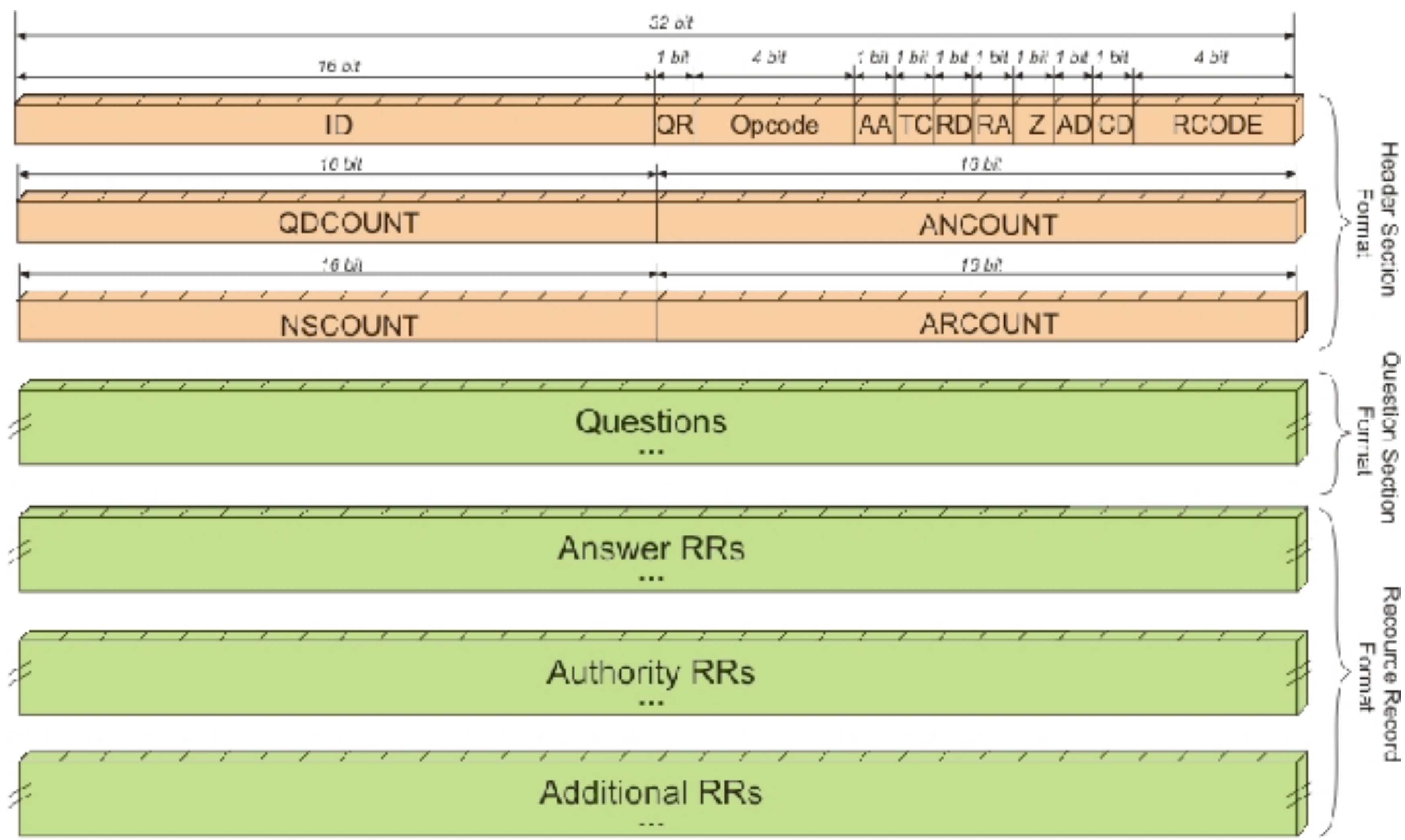

```
SimpleDecoder (b : ByteString) :=
 (n, b) ← decodeNat(b);
 (s, b) ← decodeString(b); ???
```

$\subseteq$

```
SimpleDecoder (b : ByteString) :=
 (n, b) ← decodeNat(b);
 (s, b) ← decodeString(b);
 (n', b) ← decodeNat(b);
 if (n' < 32) then
    (rs, b) ← decodeList(b, n);
    return ⟨ID :: s, readings ::rs⟩
else Error
```

# Parsing DNS Packets

- Synthesized decoder for DNS Packets (RFC 1035)
  - Specification ≤ 110 LOC
  - Valid: Compressed + Uncompressed packets
  - Data-dependent behavior used to parse response sections
  - Variable-type resource records



# Narcissus in Action

# Evaluation

| **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 |

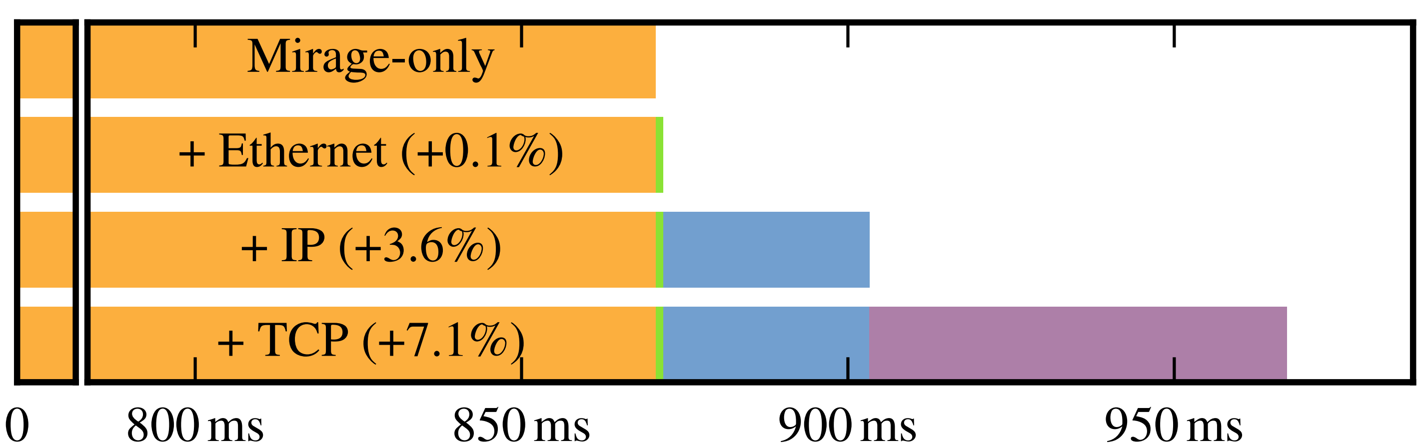
Derived Decoders

| **Format** | **LoC** | **LoP** | Higher-order |
|---|---|---|---|
| Sequencing (**ThenC**) | 7 | 164 | Y |
| Termination (**DoneC**) | 1 | 28 | Y |
| Conditionals (IfC) | 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

# Evaluation

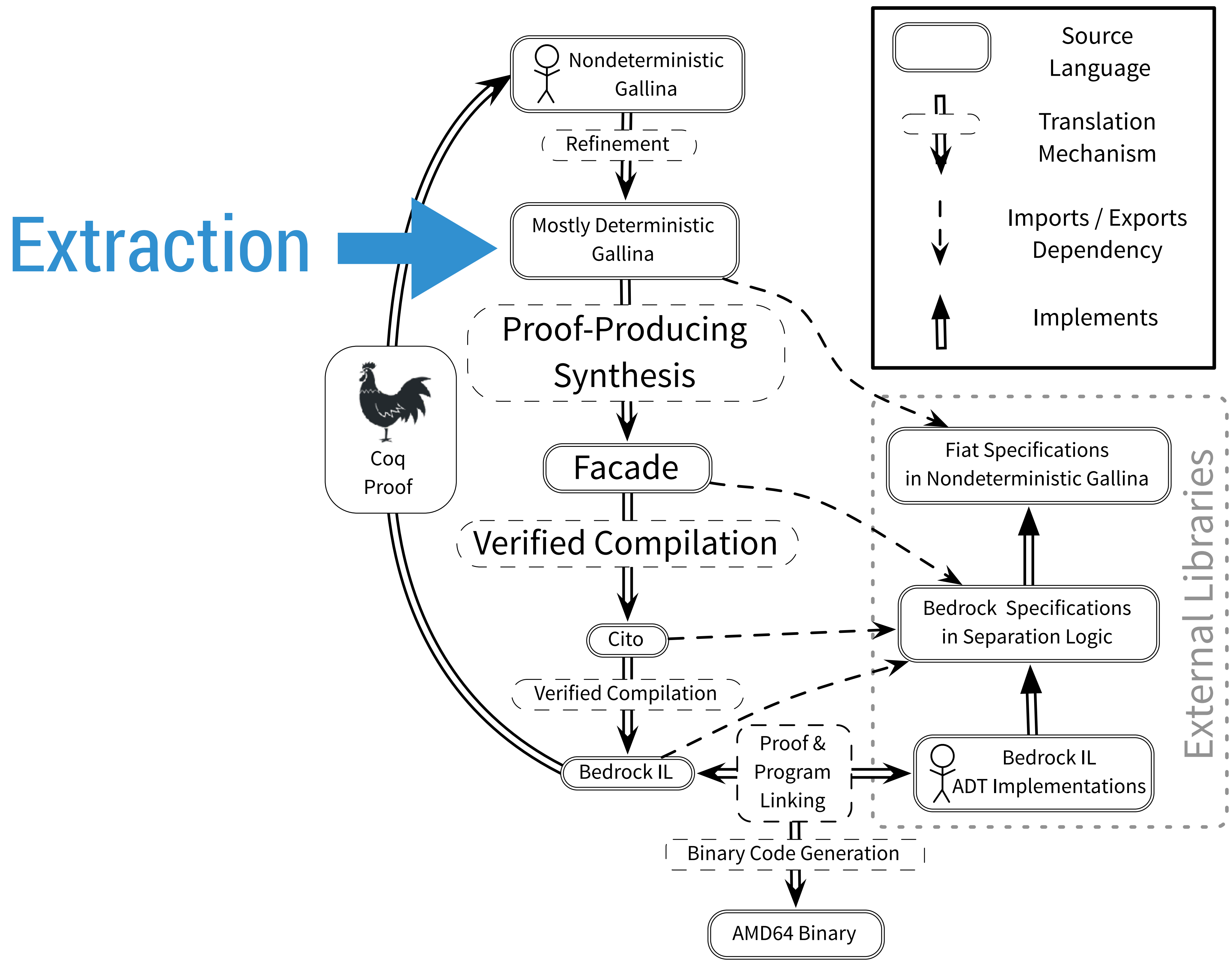
- MirageOS is a library operating system for secure, high-performance network applications written in OCaml
- Replaced network stack of MirageOS with extracted OCaml implementations of synthesized decoders.
- Found one problem in the test suite.

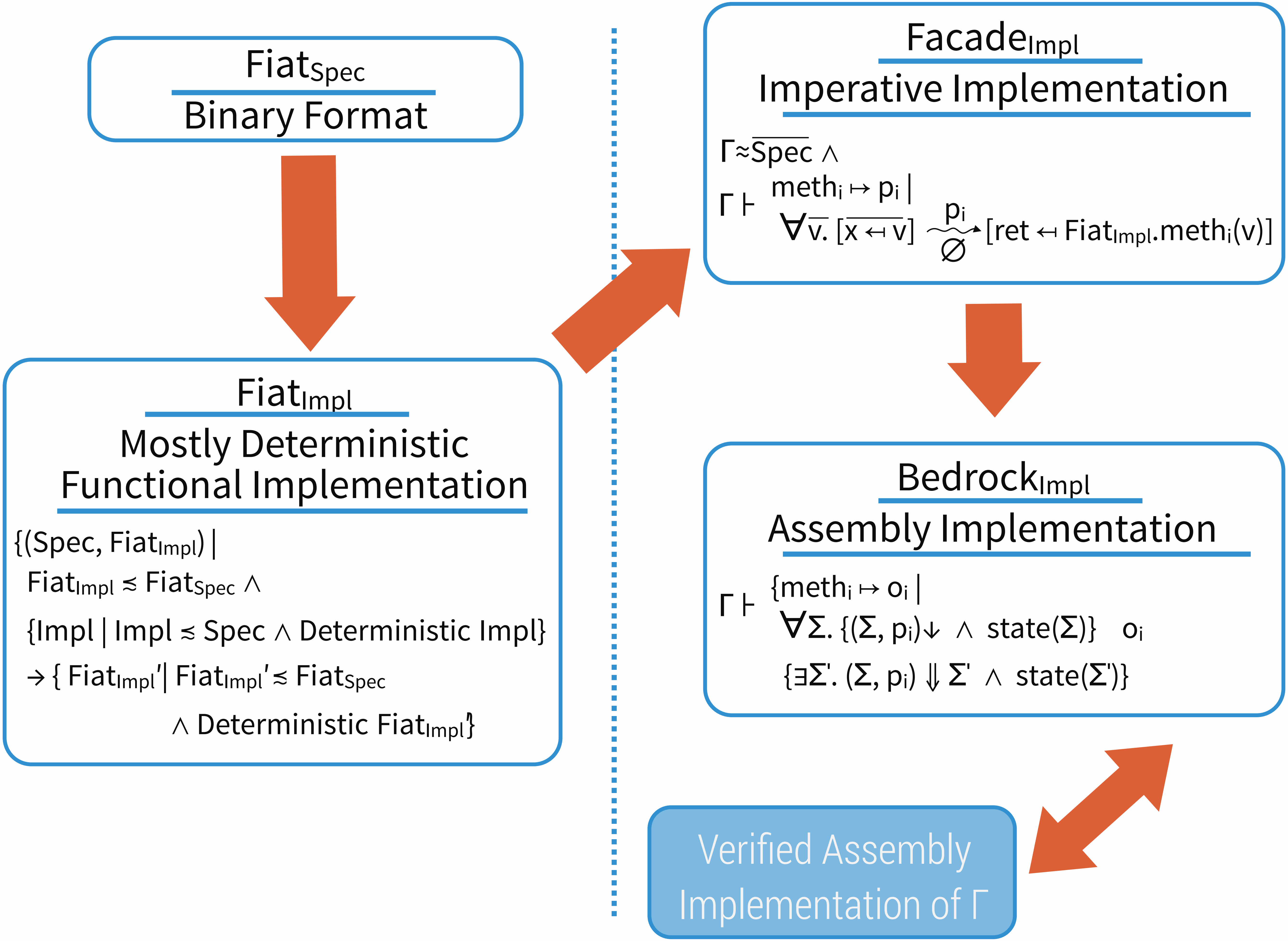



average page load time

performant?

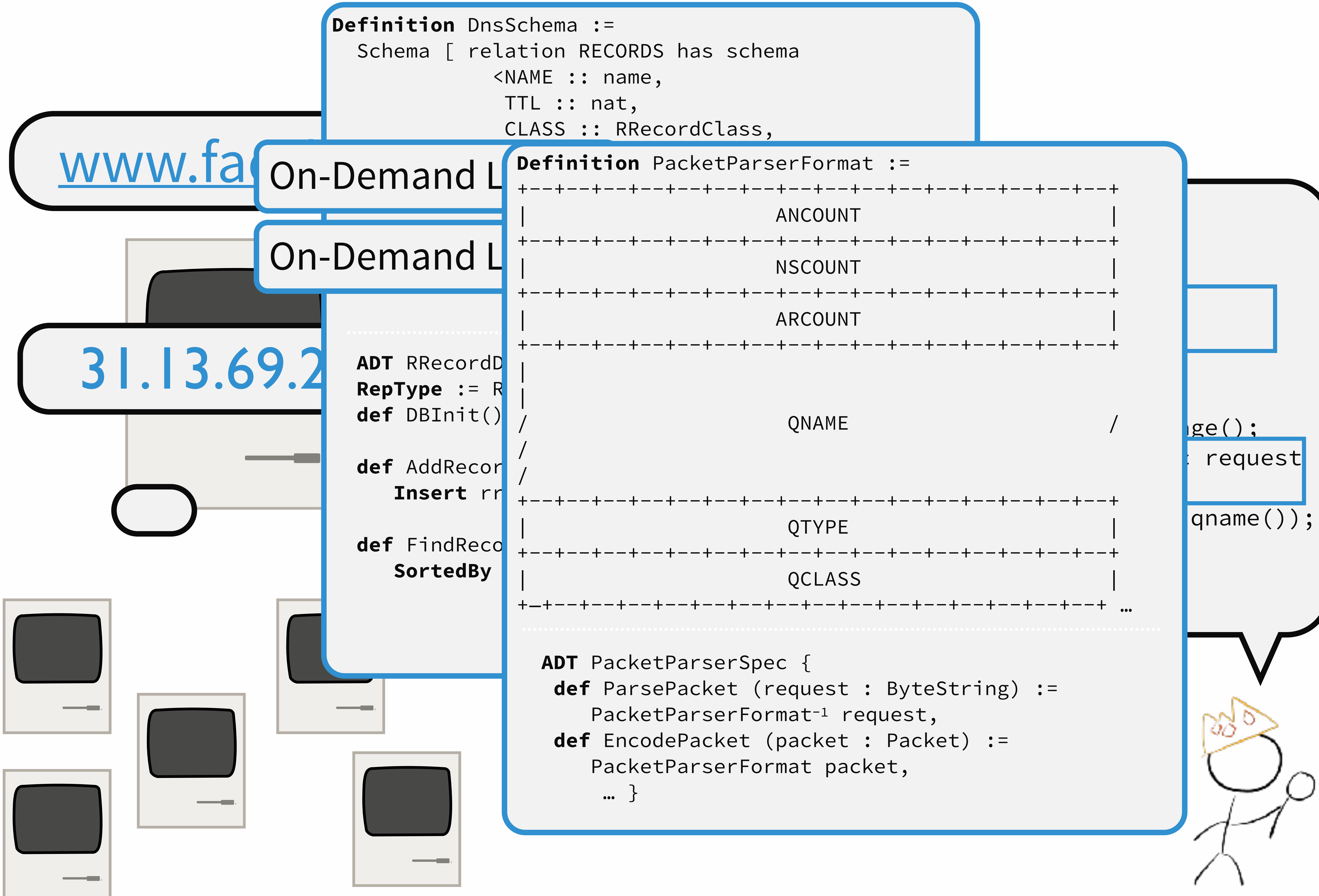
# Synthesizing Performant Code





Fiat_Spec
Binary Format
Fiat_Impl
Mostly Deterministic Functional Implementation
{(Spec, Fiat_Impl) | Fiat_Impl ≾ Fiat_Spec ∧ {Impl | Impl ≾ Spec ∧ Deterministic Impl} → { Fiat_Impl′| Fiat_Impl′≾ Fiat_Spec ∧ Deterministic Fiat_Impl}
Facade_Impl
Imperative Implementation
Γ≈Spec ∧
Γ ⊢ meth_i ↦ p_i | ∀v. [x ↤ v] p_i ∅ [ret ↤ Fiat_Impl.meth_i(v)]
Bedrock_Impl
Assembly Implementation
Γ ⊢ {meth_i ↦ o_i | ∀Σ. {(Σ, p_i)↓ ∧ state(Σ)} o_i {∃Σ'. (Σ, p_i) ⇓ Σ' ∧ state(Σ')}
Verified Assembly Implementation of Γ

# The Future?



www.fa
31.13.69.2
On-Demand L
On-Demand L
Definition DnsSchema :=
Schema [ relation RECORDS has schema
<NAME :: name,
TTL :: nat,
CLASS :: RRecordClass,
ADT RRecordD
RepType := R
def DBInit()
def AddRecor
Insert rr
def FindReco
SortedBy
Definition PacketParserFormat :=
ANCOUNT
NSCOUNT
ARCOUNT
QNAME
QTYPE
QCLASS
ADT PacketParserSpec {
def ParsePacket (request : ByteString) :=
PacketParserFormat⁻¹ request,
def EncodePacket (packet : Packet) :=
PacketParserFormat packet,
… }
ge();
request
qname());

fiat

## Implemented in Coq Proof Assistant

- Rich higher-order logic
  for specifying program behavior
- Powerful tactic language for automating search
  for exploring implementation space
- Small trusted code base
  for certifying implementation meets specification

# Today's Talk

- Narcissus:
  - Framework for synthesizing encoders and decoders from formats
  - User extensible
  - Correct-by-Construction

- Generating performant code
- Integration into high-assurance systems

Questions?