Datatype Generic Packet Descriptions

joint work with Marcell van Geest

Wouter Swierstra

About me

- Studied Mathematics and Computer Science in Utrecht
- PhD from University of Nottingham
- Postdocs at Chalmers and Nijmegen
- OCaml developer at Vector Fabrics
- Now Assistant Professor at Utrecht University

I've worked with all kinds of functional languages and interactive proof assistants.

I'm now visiting Galois on sabbatical for the summer.

The problem

Data comes in all kinds of shapes and sizes.

Some of the formats and protocols used to store or transmit information in binary can be quite complex.

Example: IPv4 header

<table>
<tr><th>Offsets</th><th>Octet</th><th colspan="8">0</th><th colspan="8">1</th><th colspan="8">2</th><th colspan="8">3</th></tr>
<tr><th>Octet</th><th>Bit</th><th>0</th><th>1</th><th>2</th><th>3</th><th>4</th><th>5</th><th>6</th><th>7</th><th>8</th><th>9</th><th>10</th><th>11</th><th>12</th><th>13</th><th>14</th><th>15</th><th>16</th><th>17</th><th>18</th><th>19</th><th>20</th><th>21</th><th>22</th><th>23</th><th>24</th><th>25</th><th>26</th><th>27</th><th>28</th><th>29</th><th>30</th><th>31</th></tr>
<tr><th>0</th><th>0</th><td colspan="4">Version</td><td colspan="4">IHL</td><td colspan="6">DSCP</td><td colspan="2">ECN</td><td colspan="16">Total Length</td></tr>
<tr><th>4</th><th>32</th><td colspan="16">Identification</td><td colspan="3">Flags</td><td colspan="13">Fragment Offset</td></tr>
<tr><th>8</th><th>64</th><td colspan="8">Time To Live</td><td colspan="8">Protocol</td><td colspan="16">Header Checksum</td></tr>
<tr><th>12</th><th>96</th><td colspan="32">Source IP Address</td></tr>
<tr><th>16</th><th>128</th><td colspan="32">Destination IP Address</td></tr>
<tr><th>20</th><th>160</th><td colspan="32" rowspan="4">Options (if IHL > 5)</td></tr>
<tr><th>24</th><th>192</th></tr>
<tr><th>28</th><th>224</th></tr>
<tr><th>32</th><th>256</th></tr>
</table>

Example: IPv4 packets

- The *Version* field must be equal to 0100 (i.e. 4)
- The *Internet Header Length* (IHL) specifies the number of 32-bit words the header is long. It must be at least 5;
- The *Header Checksum* occurs halfway through the header.
- The *Total Length* specifies the length of the packet in *bytes*. From the IHL and Total Length fields you can compute the length of the remaining data.

Problems

Parsing IPv4 is not easy.

The grammar is beyond context free: computing the length of the data or checksum involves non-trivial computations.

Problems

Parsing IPv4 is not easy.

The grammar is beyond context free: computing the length of the data or checksum involves non-trivial computations.

Specifying formats such as IPv4 is not easy

This is usually done through some combination of natural language, pseudocode, C structs/unions, RFCs,...

Problems

Parsing IPv4 is not easy.

The grammar is beyond context free: computing the length of the data or checksum involves non-trivial computations.

Specifying formats such as IPv4 is not easy

This is usually done through some combination of natural language, pseudocode, C structs/unions, RFCs,…

Can we do better?

This talk

We will try to design a **data type** for describing various binary formats.

From such descriptions, we can *generate* parsers and pretty printers.

Universiteit Utrecht

Faculty of Science
Information and Computing Sciences

This talk

We will try to design a **data type** for describing various binary formats.

From such descriptions, we can *generate* parsers and pretty printers.

By embedding this construction in a dependently typed programming language – such as Coq, Agda, or Idris – we can prove the desired round trip property relating parsing and pretty printing.

This talk

We will try to design a **data type** for describing various binary formats.

From such descriptions, we can *generate* parsers and pretty printers.

By embedding this construction in a dependently typed programming language – such as Coq, Agda, or Idris – we can prove the desired round trip property relating parsing and pretty printing.

Crucially, we will use *dependent types* to mix computation and static type safety.

Warm up: universes

```
data FT : Set where
  word : (n : ℕ) → FT
  _⊗_ : FT → FT → FT

⟦_⟧ : FT → Set
⟦ word n ⟧   = Vec Bit n
⟦ t₁ ⊗ t₂ ⟧  = ⟦ t₁ ⟧ × ⟦ t₂ ⟧
```

A pair of a data type that describes a collection of types (FT) and the decoding function mapping descriptoins to their corresponding types is sometimes known as a **universe**.

Generating parsers from descriptions

```
Parser : Set → Set
Parser a = List Bit → Maybe (a × List Bit)

parse : (f : FT) → Parser ⟦ f ⟧
parse (t₁ ⊗ t₂)      = _,_ <$> parse t₁ <*> parse t₂
parse (word n)       = take n
  where
  take : (n : ℕ) → Parser (Vec n Bit)
```

Pretty printing data

$$
\begin{array}{lcl}
\mathtt{pp} : (\mathtt{f} : \mathtt{FT}) \to [\![\, \mathtt{f} \,]\!] \to \mathtt{List\ Bit} & & \\
\mathtt{pp}\ (\mathtt{word}\ \mathtt{n})\ \mathtt{bs} & = & \mathtt{toList}\ \mathtt{bs} \\
\mathtt{pp}\ (\mathtt{t}_1 \otimes \mathtt{t}_2)\ (\mathtt{x}\ ,\ \mathtt{y}) & = & \mathtt{pp}\ \mathtt{t}_1\ \mathtt{x} \mathbin{+\!\!+} \mathtt{pp}\ \mathtt{t}_2\ \mathtt{y}
\end{array}
$$

Round trip correctness

```
roundTrip : (f : FT) → (x : ⟦ f ⟧) →
  parse f (pp f x) == just (x, [])
```

Proof by induction on our description `f`.

Limitations

- This gives us a very limited language for describing products of fixed size words.
- There are no dependencies between a field and the *type* of the remaining fields, e.g., a length field specifying the size of the remaining data;
- There are no constraints on the *values* that fields may assume, e.g., a checksum field that is computed from all other fields or a constant field that must be equal to 0100.

Universiteit Utrecht

Faculty of Science
Information and Computing Sciences

Interlude: sigma types

```
data Pair (a : Set) (b : Set) : Set where
  _,_ : a → b → Pair a b
```

Note: the type of the constructor is *not* dependent.

What if the type of the second component can depend on the value of the first?

Interlude: sigma types

```
data Pair (a : Set) (b : Set) : Set where
  _,_ : a → b → Pair a b
```

Note: the type of the constructor is *not* dependent.

What if the type of the second component can depend on the value of the first?

```
data Σ (a : Set) (b : a → Set) : Set where
  _,_ : (x : a) → b x → Σ a b
```

Constructive equivalent of existential quantification.

Take two

```
data FT : Set where
  word   : (n : ℕ) → FT
  _⊗_    : FT → FT → FT
  calc   : (t : FT) → ⟦ t ⟧ → FT
  sigma  : (t : FT) → (⟦ t ⟧ → FT) → FT

⟦_⟧ : FT → Set
⟦ word n ⟧      = Vec Bit n
⟦ t₁ ⊗ t₂ ⟧     = ⟦ t₁ ⟧ × ⟦ t₂ ⟧
⟦ calc t v ⟧    = ⊤
⟦ sigma t f ⟧   = Σ ⟦ t ⟧ (\ v → ⟦ f v ⟧)
```

This universe now relies on *induction recursion*.

Universiteit Utrecht

Faculty of Science
Information and Computing Sciences

Examples: checksum

```
checksumedByte : FT
checksumedByte =
  sigma (word 7) (\ d → calc (word 1) (parity d))
  where
    parity : Vec Bit n → Bit
```

- A 7 bit word;
- Followed by a single parity bit.

Examples: checksum

```
checksumedByte : FT
checksumedByte =
  sigma (word 7) (\ d → calc (word 1) (parity d))
  where
    parity : Vec Bit n → Bit
```

- A 7 bit word;
- Followed by a single parity bit.

The corresponding type ⟦ `checksumedByte` ⟧ is

```
Σ (Vec Bit 7) (\d → ⊤)
```

Examples: length field

```
lengthData : FT
lengthData =
  sigma (word 32) (\ d → word (fromBits d))
  where
    fromBits : ⟦ Vec Bit n ⟧ → ℕ
```

- A 32-bit word – the length field;
- Followed by a word of that length;

Universiteit Utrecht

Faculty of Science
Information and Computing Sciences

Examples: length field

```
lengthData : FT
lengthData =
  sigma (word 32) (\ d → word (fromBits d))
  where
    fromBits : ⟦ Vec Bit n ⟧ → ℕ
```

- A 32-bit word – the length field;
- Followed by a word of that length;

The corresponding type ⟦ `lengthData` ⟧ is:

```
Σ (Vec Bit 32) (\d → Vec Bit (fromBits d))
```

Parsing

```
parse : (f : FT) → Parser ⟦ f ⟧
parse (sigma x f) input with parse x input
... | just (y , rest)  = _,_ <$> pure y
                             <*> parse (f y)
... | nothing          = nothing
parse (calc t x) input with parse t input
... | just (y , rest)  = if x == y
                         then just ( tt , rest)
                         else nothing
... | nothing          = nothing
```

Parsing sigma types is (almost) the same as parsing pairs; to parse derived fields, we parse the desired value and check it is what we expect.

What next?

We can update pretty printing and round trip proofs.

What next?

We can update pretty printing and round trip proofs.

In ideal world, where all binary formats are defined by type theorists, we would now be done.

What next?

We can update pretty printing and round trip proofs.

In ideal world, where all binary formats are defined by type theorists, we would now be done.

But…

What next?

We can update pretty printing and round trip proofs.

In ideal world, where all binary formats are defined by type theorists, we would now be done.

But...

- The computations involved are complicated! To make matters worse, we need to mix information about *how* to encode/decode data in the actual descriptions.
- Computed data (such as the header checksum in IPv4) may occur before all data on which it relies is present.

Plan of attack

1. Define a richer universe of file formats
2. Define a predicate that makes it clear when something is 'trivial' to parse/pretty print – we can use this to generate the (de)serialization functions.
3. Define *transformations* on this richer universe, adding new fields or massaging data somehow.

Universe

```
data DT : Set₁ where
  leaf    : Set → DT
  _⊗_     : DT → DT → DT
  sigma   : (c : DT) → (⟦ c ⟧ → DT) → DT

⟦_⟧ : DT → Set
⟦ leaf A ⟧      = A
⟦ l ⊗ r ⟧      = ⟦ l ⟧ × ⟦ r ⟧
⟦ sigma t f ⟧  = Σ ⟦ t ⟧ (\ x → ⟦ f x ⟧)
```

We define a universe closed under abritrary types, products, and dependent products.

Universe

```
data DT : Set₁ where
  leaf   : Set → DT
  _⊗_    : DT → DT → DT
  sigma  : (c : DT) → (⟦ c ⟧ → DT) → DT
```

- This universe is 'large' – it contains arbitrary other types. We can resolve this easily enough by parametrizing our development by a base universe.
- We arguably don't need both products and dependent products. We find it useful to distinguish between *sequencing* (products) and *dependency* (sigma types).

Universe: example

```
length+word : DT
length+word =
  sigma (leaf ℕ) (\ len → leaf (Vec Bits len))
```

This allows us to specify dependencies *independently* of the encoding – this is particularly useful as dependencies and computations become more complex.

Parsing?

Of course, these descriptions contain *arbitrary* data – in particular data such as functions that cannot be serialized easily.

If the description contains only binary words in the leaves, we can parse it easily enough.

Parsing?

Of course, these descriptions contain *arbitrary* data – in particular data such as functions that cannot be serialized easily.

If the description contains only binary words in the leaves, we can parse it easily enough.

```
data IsLowLevel : DT → Set where
  leaf    : IsLowLevel (leaf (Vec Bit n))
  pair    : IsLowLevel l →
            IsLowLevel r →
            IsLowLevel (l ⊗ r)
  sigma   : IsLowLevel c →
            ((x : ⟦ c ⟧) → IsLowLevel (d x)) →
            IsLowLevel (sigma c d)
```


Universiteit Utrecht

Faculty of Science
Information and Computing Sciences

Parsing?

```
data IsLowLevel : DT → Set where
  ...

parse : (f : FT) → IsLowLevel f → Parser ⟦ f ⟧
```

The definition of `parse` is pretty much identical to what we saw previously.

Proofs of the `IsLowLevel` predicate can be generated automatically for most formats.

Parsing?

```
data IsLowLevel : DT → Set where
   ...

parse : (f : FT) → IsLowLevel f → Parser ⟦ f ⟧
```

The definition of `parse` is pretty much identical to what we saw previously.

Proofs of the `IsLowLevel` predicate can be generated automatically for most formats.

Assuming all data is binary words, we can call our parse function. But what if it isn't?

Conversions

We can specify how to convert from one representation to another:

```
data Conversion (t₁ t₂ : DT) : Set where
  convert : (enc : ⟦ t₁ ⟧ → ⟦ t₂ ⟧) →
            (dec : ⟦ t₂ ⟧ → Maybe ⟦ t₁ ⟧) →
            ((x : ⟦ t₁ ⟧) → (dec (enc x) ≡ just x))
            Conversion t₁ t₂
```

This is a *semi-partial isomorphism* – or shift in representation – between two types, $[\![\ t_1\ ]\!]$ and $[\![\ t_2\ ]\!]$.

Categories of conversions

These conversions are closed under composition.

```
_⊙_ : Conversion t₁ t₂ →
       Conversion t₂ t₃ →
       Conversion t₁ t₃
```

And we can define an identity conversion:

```
idConvert : Conversion t t
```

Example: Explicit Congestion Notification (ECN)

```
data ECN : Set where
  Non-ECT   : ECN
  ECT0      : ECN
  ECT1      : ECN
  CE        : ECN

enc : ECN → Vec Bit 2
dec : Vec Bit 2 → ECN
enc-dec : (x : ECN) → dec (enc x) ≡ just x
```

Conversions describe the shift in representation of *one* field – but how do we extend this to handle a complete description?

Converting descriptions

```
data DTX : DT → Set₁ where
  convert : Conversion t₁ t₂ → DTX t₁
  pair    : DTX l → DTX r → DTX (l ⊗ r)
  sigma   : DTX c → ((x : ⟦ c ⟧) → DTX (d x)) →
            DTX (sigma c d)
```

You can think of `DTX t` as a transformation on the description `t` – applying certain data conversions at specific points in the description.

Once again, there is an identity transformation and we can compute the reflexive-transitive closure, `DTX*`.

Example: converting natural numbers to bits

```
length+word : DT
length+word =
  sigma (leaf ℕ) (\ len → leaf (Vec Bits len))

length+word+enc : DTX length+word
length+word+enc = sigma int32 (\ len → copy)
  where
    int32 : Conversion ℕ (Vec Bit 32)
    copy : Conversion t t
```

Example: converting natural numbers to bits

```
length+word : DT
length+word =
  sigma (leaf ℕ) (\ len → leaf (Vec Bits len))

length+word+enc : DTX length+word
length+word+enc = sigma int32 (\ len → copy)
  where
    int32 : Conversion ℕ (Vec Bit 32)
    copy : Conversion t t
```

Describing the lengths of IPv4 packets in this style is really worth the additional effort.

Calculating new types and values

```
extendType   :  DTX t → DT
extendValue  :  (tx : DTX t) →
                ⟦ t ⟧ → ⟦ extendType tx ⟧
```

Using these functions we can calculate a new modified description, call the corresponding parser, and convert the result back to the desired format.

What's missing?

There were two problems we set out to solve initially:

- The computations need to mix information about *how* to encode/decode data in the actual descriptions.

What’s missing?

There were two problems we set out to solve initially:

- The computations need to mix information about *how* to encode/decode data in the actual descriptions.

Done

What's missing?

There were two problems we set out to solve initially:

- The computations need to mix information about *how* to encode/decode data in the actual descriptions.

Done

- Computed data (such as the header checksum in IPv4) may occur before all data on which it relies is present.

What's missing?

There were two problems we set out to solve initially:

- The computations need to mix information about *how* to encode/decode data in the actual descriptions.

Done

- Computed data (such as the header checksum in IPv4) may occur before all data on which it relies is present.

How can we *add new fields* to existing descriptions?

Idea: a new constructor for the DTX type

```
data DTX (top : DT) : DT → Set where
  ...
  insert : ...
```

We add a new type parameter to the DTX type, representing the ‘global’ description that we’re transforming.

We add a new constructor to the DTX type, inserting new fields to an existing description.

Idea: a new constructor for the DTX type

```
data DTX (top : DT) : DT → Set where
  ...
  insert : (t' : DT) → Side → (⟦ top ⟧ → ⟦ t' ⟧)
           → DTX top t

data Side : Set where
  left right : Side
```

Calling `insert t' left f` – inserts a field of type `t'` before the current format, calculated from the value of all the other fields (⟦ `top` ⟧) using the function `f`.

Example: inserting a checksum

```
t : DT
t = length+word+enc
checksummed : DTX t t
checksummed = insert (leaf Bit) left checksum
  where
   checksum : ⟦ t ⟧ → Bit
```

Here we can insert a checksum bit *before* the remaining data.

Calculating new types and values

```
extendType : {t : DT} → DTX top t → DT
extendType {t = t} (insert t' left _)  = t' ⊗ t
extendType {t = t} (insert t' right _) = t ⊗ t'

extendValue : (tx : DTX top t) →
  ⟦ top ⟧ → ⟦ t ⟧ → ⟦ extendType tx ⟧
extendValue (insert t' left f) dtop d
  = (f dtop , d)
extendValue (insert t' right f) dtop d
  = (d , f dtop)
```

Once again, given any description transformation (a value of type DTX), we can compute the resulting description and convert the results of parsing.

Checking parsed values

Of course, we cannot decide whether or not a checksum is correct before parsing the remaining data. We can, however, read in the data and check it validity post hoc:

```
check : (tx : DTX t t) →
  ⟦ extendType tx ⟧ → Maybe ⟦ t ⟧
```

Alternatively, we can define single top-level function that parses and validates all data.

Drawbacks

The type of the function used to compute derived fields in the insertion constructor is:

$[\![\ \texttt{top}\ ]\!] \rightarrow [\![\ \texttt{t'}\ ]\!]$

Given the entire top-level data structure, we need to compute a value of type `t'`.

But what if we want to perform *conditional* extensions on existing data?

Example: maximum element of a vector

```
vecBits : DT
vecBits = sigma ℕ (\ len → Vec Bit len)

insertMax : DTX vecBits vecBits
insertMax = sigma copy iMax
  where
  iMax : (len : ℕ) → DTX vecBits (Vec ℕ len)
  iMax zero     = copy
  iMax (suc n)  = insert ℕ right maxVec
  maxVec : ⟦ vecBits ⟧ → ℕ
```

We need to compute the maximum of *any* vector, even if we only want to add a new field to *non-empty* vectors.

Solution

The argument to `insert` that calculates new values has the type:

```
⟦ top ⟧
  → ⟦ t' ⟧
```

We want to make it more specific, allowing it to refer to the current subtree:

```
(d : ⟦ top ⟧) → (v : ⟦ t ⟧) → Select d v
  → ⟦ t' ⟧
```

In our maximum vector example, this would correspond to having a non-empty vector as argument, rather than having to handle all possible cases.

Selection (sketched)

To make the type of insertion more precise, we update the type of our transformations:

```
data _▷_ : DT → DT → Set where
  ...

data DTX (top : DT) :
  (t : DT) → (s : top ▷ t) → Set where
    _⊗_ :  DTX top l (s >> fst) →
           DTX top r (s >> snd) →
           DTX top (l ⊗ r) s
  ...
```

This lets us give the more precise type to the `insert` constructor.

Recap

- A universe describing arbitrary types closed under (dependent) products.

Recap

- A universe describing arbitrary types closed under (dependent) products.
- A predicate stating when elements of this universe are ‘easy’ to parse.

Recap

- A universe describing arbitrary types closed under (dependent) products.
- A predicate stating when elements of this universe are 'easy' to parse.
- Generic parse and pretty print functions – satisfying the expected round trip property.

Recap

- A universe describing arbitrary types closed under (dependent) products.
- A predicate stating when elements of this universe are 'easy' to parse.
- Generic parse and pretty print functions – satisfying the expected round trip property.
- Transformations allowing you to modify data representation or insert new fields.

Recap

- A universe describing arbitrary types closed under (dependent) products.
- A predicate stating when elements of this universe are 'easy' to parse.
- Generic parse and pretty print functions – satisfying the expected round trip property.
- Transformations allowing you to modify data representation or insert new fields.

Together this gives you a 'DSL' for describing binary data.

Case study: IPv4

Offsets	Octet	0								1								2								3							
Octet	Bit	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
0	0	Version				IHL				DSCP						ECN		Total Length															
4	32	Identification																Flags			Fragment Offset												
8	64	Time To Live								Protocol								Header Checksum															
12	96	Source IP Address																															
16	128	Destination IP Address																															
20	160	Options (if IHL > 5)																															
24	192																																
28	224																																
32	256																																

Types of fields

- **Enumerations** – such as the ECN or Protocol fields – these are easy to model in Agda; we can describe their low-level encodings later using a suitable conversion.
- **(Bounded) natural numbers** – the Total Length or Internet Header Length are big-endian integers. We typically use `Fin (2 ^ 32)` to describe a 32-bit integer – this makes the computations using these fields easier. Conversions describe how to serialize such values to words.

Types of fields (continued)

- **Constants** – the Version field must be 0100. These can be inserted into a description.
- **Variable length words** – the Data and Options field contain words of variable length. Using `sigma` types we can capture the dependency between fields. The calculations involved can be a bit messy...

Results

Complete description of IPv4 in four steps:

1. Basic definition describing the data stored in a packet.
2. Insert constant fields and converting convenient lengths to their actual representation.
3. Binary encoding of all high-level data.
4. Insertion of checksums.

Compiled to Haskell and tested against existing IPv4 implementations.

Results

Complete description of IPv4 in four steps:

1. Basic definition describing the data stored in a packet.
2. Insert constant fields and converting convenient lengths to their actual representation.
3. Binary encoding of all high-level data.
4. Insertion of checksums.

Compiled to Haskell and tested against existing IPv4 implementations.

Found a bug in our implementation – choice of big-endian vs. little-endian of integers.

Further work

- Better error messages when parsing fails.
- Named fields – either using Strings, reflection, or singleton types.
- Proofs of (in)equalities are no fun in Agda.

Take home messages

Data type generic programming uses type structure to derive new functions.

Take home messages

Data type generic programming uses type structure to derive new functions.

In this domain, we've studied how to *transform* such descriptions to accommodate for external constraints, imposed by existing binary formats.

Take home messages

Data type generic programming uses type structure to derive new functions.

In this domain, we've studied how to *transform* such descriptions to accommodate for external constraints, imposed by existing binary formats.

Instead of writing arbitrary transformations on descriptions, we have a 'deep embedding' of various well-behaved transformations.

Take home messages

Data type generic programming uses type structure to derive new functions.

In this domain, we've studied how to *transform* such descriptions to accommodate for external constraints, imposed by existing binary formats.

Instead of writing arbitrary transformations on descriptions, we have a 'deep embedding' of various well-behaved transformations.

From these transformations, we can calculate new type descriptions and their associated parsers for realistic binary formats.

Questions?