Generic Packet Descriptions: Verified Parsing and Pretty Printing of Low-Level Data

Marcell van Geest  
Utrecht University  
The Netherlands  
marcell@marcell.nl

Wouter Swierstra  
Utrecht University  
The Netherlands  
w.s.swierstra@uu.nl

Abstract
Complex protocols describing the communication or storage of binary data are difficult to describe precisely. This paper presents a collection of data types for describing a binary data formats; the corresponding parser and pretty printer are generated automatically from a data description. By embedding these data types in a general purpose dependently typed programming language, we can verify once and for all that the parsers and pretty printers generated in this style are correct by construction. To validate our results, we show how to write a verified parser of the IPv4 network protocol.

CCS Concepts  
• Software and its engineering  
  → Functional languages: Domain specific languages;  
  → Theory of computation  
  → Type theory;

Keywords  
data type generic programming, dependent types, parsing, pretty printing

ACM Reference Format:
https://doi.org/10.1145/3122975.3122979

1 Introduction
There is a general trend towards software systems that distribute computation across multiple machines, as demonstrated by the ever-increasing popularity of web applications, cloud solutions, and thin clients. At the same time, as more and more administrative tasks are automated, long-term storage and accessibility of data becomes increasingly important. What these issues have in common is the need to communicate data effectively and without error, whether the communication takes place between clients and servers or past, current, and future incarnations of the same software system. This communication almost always involves translating between a high-level representation of data to a low-level representation like strings of letters or bits.

Traditionally, protocols for Internet communication are published as Requests for Comments, many-page documents that attempt to use a combination of plain written English, punctuation-based diagrams and common programming constructs such as C structs and unions to describe the format of messages. This method of description, though consistent with long-standing documentation practices, unfortunately may contain ambiguities or internal inconsistencies.

Several format description languages have been proposed to address these ambiguities and inconsistencies. Examples range from abstract constructs such as Backus-Naur grammars, through complex and thoroughly-engineered standalone languages such as ASN.1 and XML Schema, to implicit format descriptions like attribute-annotated .NET classes.

This paper takes a slightly different approach. We will show how to define an embedded domain specific language for describing data formats and derive the serialization and deserialization functions from these descriptions. More specifically, this paper makes the following novel contributions:

• We sketch the basic idea of data type generic programming using universes (Section 2). From such descriptions, we can generate the serialization and deserialization functions and prove the round trip property that relates them. Working with a single universe, however, does have its drawbacks. In particular, we identify two problems with the simple approach: data descriptions are polluted with information about encodings and data dependencies may only refer to existing fields.

• To address these limitations we define a series of increasingly complex universe transformations. Initially, we show how transform the types in our format descriptions, allowing programmers to shift incrementally from a high-level description capturing the relevant data to a low-level description that rigidly fixes the precise encoding as binary words (Section 4). A second kind of transformation extends existing descriptions with new information, such as checksums, that may be computed from existing data (Section 5).

• Finally, we validate our format description language by giving a full description of IPv4 (Section 6). This is particularly challenging as there are complex dependencies between the fields. There is a non-trivial amount of computation necessary to determine the length of an IPv4 packet; the checksum field appears halfway through the packet, before all the data has arrived. Precisely describing such dependencies is not at all straightforward.

2 Simple Universes
Universes are a fundamental generic programming tool for describing a collection of types, thereby enabling the definition of generic functions by induction over the structure of these types. In this section, we will illustrate how to use universes to describe data...
formats and generate the associated parsers in the dependently
typed programming language Agda [Bove et al. 2009; Norell 2007].
To warm up, we define the universe of format types (FT), as follows:

1. **data FT : Set where**
   - word : (n : ℕ) → FT
   - _⊗_ : FT → FT → FT

Values of type FT may consist of fixed width words or a product of
two types drawn from FT. Using this data type, we can already
describe a (fragment of) some data format, consisting of two 32-bit
words:

```
ft₁ : FT
ft₁ = word 32 ⊗ word 32
```

Note that values of FT are data; we can compute the corresponding
type that they represent as follows:

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

Unsurprisingly, we map the word constructor to vectors of bits and
the pairing constructor ⊗ to Agda’s pairs. Using this interpretation,
we can assign meaning to our ft₁ example: the type corresponding to
⟦ ft₁ ⟧ is indeed a pair of 32-bit words.

The advantage of using such a universe – as opposed to working with
the types directly – is that we can define generic functions by induction over the structure of our types. For example, we can define a parse function that tries to split its input into words of the
correct size:

```
parse : (f : FT) → List Bit → Maybe (⟦ f ⟧ × List Bit)
parse (word n) = splitList n
parse (t₁ ⊗ t₂) = _⊗_ <$> parse t₁ <*> parse t₂
```

We can define our parser using the applicative combinators [McBride
and Paterson 2008]. Here we have chosen to represent the input as
a list of bits, rather than a vector of fixed size. As we extend this universe with more constructors, we will no longer be able
to assume that we statically know the size of the input. For that
reason, the parser may fail and return nothing.

Parsing is not the only operation that we can define. Pretty
printing is trivial:

```
pp : (f : FT) → ⟦ f ⟧ → List Bit
pp (word n) bs = bs
pp (t₁ ⊗ t₂) (x₁, x₂) = pp t₁ x₁ ++ pp t₂ x₂
```

Finally, as we are working in Agda, we can prove the round trip
property relating parsing and pretty printing:

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

In fact, this property is too weak to be used recursively – the
proof succeeds using a slightly stronger variant:

```
roundTrip' : (f : FT) → (x : ⟦ f ⟧) → (rest : List Bit) →
  parse f (pp f x ++ rest) ≡ just (x, rest)
```

Although this may seem trivial in this small example, this does
illustrate a key advantage of this approach: embedding a domain
specific language into a language with dependent types enables us
to verify properties of our generic functions in Agda itself; this is
impossible in a standalone generic function in Agda itself.

### 2.1 Beyond Simple Universes

The universe described above can only describe products of fixed
width words. In practice, this is far too limited. There may be dep-
endencies between the different components of the products. For
example, a header field may contain information about the length of
the remaining data. In this way, the value of one field may influence
the type of the remaining format.

A second form of dependency may arise between the fields. For
example, consider a checksum field, whose value consists of a hash
computed from other fields. There is no way to restrict the value
that a certain field may assume in the current design of our universe.

To handle the two scenarios described above, we will extend our
simple universe with two new constructors. Doing so will make the
universe FT and its decoding function ⟦⟧ mutually recursive; more
precisely, we define our universe using induction-recursion [Dybjer
and Setzer 1999] as follows:

1. **data FT where**
   - word : (n : ℕ) → FT
   - _⊗_ : FT → FT → FT
   - calc : (t : FT) → ⟦ t ⟧ → FT
   - sigma : (t : FT) → (⟦ t ⟧ → FT) → FT

The FT data type introduces two new constructors, calc and sigma.
The calc constructor represents fields that store a value computed
from prior fields, described by the format t. The sigma construc-
tor extends our universe with dependent pairs, where the type of
the second component may depend on the value of the first. The
corresponding interpretation of both these constructors is straight-
forward: the calc format does not store interesting data, whereas
the sigma constructor is mapped to Agda’s Σ-type, representing
dependent pairs.

Using this universe, we can describe more complicated formats.
For example, the following format consists of a 8-bit field, followed by
a parity bit:

```
ft₂ : FT
ft₂ = sigma (word 8) (λ d → calc (word 1) (parity d))
```

Although we can describe certain data formats using our uni-
verse, we still need to extend our simple parse and pp functions
to handle the two new constructors. Fortunately, the definition of
parse remains relatively straightforward. The sigma constructor
essentially behaves the same as pairs (⊗); the only difference is
that the dependency between the components prevents us from
using the usual applicative combinators directly. Finally, to handle
derived fields, we can check that when data is successfully parsed,
it coincides with the expected value:

```
parse : (f : FT) → List Bit → Maybe (⟦ f ⟧ × List Bit)
parse (word n) = splitList n xs
parse (t₁ ⊗ t₂) = _._ <$> parse t₁ <$> parse t₂
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 (t, rest) else nothing
... | nothing = nothing

The pp function can be extended similarly, as can the proof of our roundTrip property.

While we can express simple data formats using this universe, there are still several important problems that remain to be addressed. Firstly, in this universe we must mix the (de)serialization code with computations. Consider the following example, consisting of an 8-bit length field, followed by a word of that length:

f₃ : FT
f₃ = sigma (word 8) (λ d → word (toNat d))

where

toNat : Vec Bit n → ℕ
As the first component stores a vector of bits, we need to convert it explicitly to a natural number using the toNat function, before we can use it as part of the remaining format description. If we would like to vary the encoding of natural numbers – switching from big-endian to little-endian, for example – we would need to update all the references to d in the remainder of the description. Such explicit conversions clutter the format description; ideally, we would like to work with the underlying natural number directly, handling the decoding and encoding separately. For small examples, such as f₃, these explicit decodings may not seem too problematic, but for larger descriptions, such as the IPv4 cases study presented later the amount of computation necessary to describe the dependencies between fields is more substantial.

A second problem with this universe is that the order of the fields in many existing data formats do not always follow the order in which we might expect. For example, the checksum of an IPv4 packet appears halfway through the header, before the actual data has been received. The dependent pairs and calculated fields we have seen so far, allow us to describe dependencies on previous fields; we cannot easily describe forward dependencies on later fields using this universe.

While this simple universe and the corresponding definitions provide evidence that the approach we wish to take may be viable, there is still some work to be done to make the approach scale well to realistic formats. In the coming sections we will address the two problems sketched above.

### 3 Beyond Bits

To avoid working explicitly with the low-level bit representation of our data when defining formats, we define the following alternative universe:

mutual
data DT : Set₁ where
  leaf : Set → DT
  _⊗_ : DT → DT → DT
  sigma : (c : DT) → ([| c |] → DT) → DT

[|] : DT → Set
[leaf A] = A
[I ⊗ r] = ([I] × [r])
[sigma t f] = Σ [t] (λ x → [| f x |])

We will often view values of DT as "type trees", where each leaf is a leaf that holds a type and the other two constructors are nodes holding structural information: the order (_⊗_) or the dependency relation (sigma) of subtrees. Although sigma is strictly more general than _⊗_, we have found it useful to make the distinction between order and dependency.

Our previous universe, FT, could only represent nested pairs of words. As a result, the dependency introduced sigma constructor must 'decode' any interesting information from its low-level representation. By allowing arbitrary types to appear in the leaves of our DT descriptions, we no longer have to include these low-level conversions in the definitions of our formats, but this generality comes at a price. Firstly, the DT type is now large, i.e., it has type Set₁. This can be remedied easily enough by parametrizing our definitions by a base universe, thereby stratifying our construction. More importantly, however, we can no longer define our parse and pp functions directly; as we allow arbitrary types to occur in the leaves, these may include functions or other values that cannot be easily converted to bits. We will address this in two steps:

- We will define the predicate, IsLowLevel, describing when a description only contains binary words in the leaves, and may therefore be serialized and deserialized in the same fashion as the FT universe we saw previously;
- We define transformations relating different format descriptions, that can describe how to map high-level data (that is easy to manipulate) to low-level data (that is easier to (de)serialize).

#### 3.1 Low-Level Descriptions

When the leaves of a format description only consist of binary words, we will refer to such a description as low-level. This intuition is made precise by the following predicate:

data IsLowLevel : DT → Set where
  instance leaf : IsLowLevel (leaf (Vec Bit n))
  instance _⊗_ : IsLowLevel l → IsLowLevel (l ⊗ r) → IsLowLevel (l ⊗ r)
  instance sigma : IsLowLevel c → ((x : [c]) → IsLowLevel (d x)) → IsLowLevel (sigma c d)

By declaring the constructors of the IsLowLevel type as instances [Devriese and Piessens 2011], Agda can automatically construct IsLowLevel proofs for most formats that we will cover in this paper.

Given an instance of IsLowLevel t, pretty printing and parsing is no harder than for the FT universe we saw in the previous section. It is straightforward to adapt those definitions to two functions:

pp : {t : DT} → IsLowLevel t → [| t |] → List Bit
parse : {t : DT} → IsLowLevel t → List Bit → Maybe ( [| t | ] × List Bit)
Correctness The correctness property relating parse and pp is now straightforward to prove:

\[
\text{roundTrip} : \{ t : \text{DT} \} \rightarrow (\text{ill} : \text{IsLowLevel} \text{t}) \rightarrow
\]
\[
(\text{d} : \left[ t \right]) \rightarrow (\text{rest} : \text{List Bit}) \rightarrow
\]
\[
\text{parse ill pp ill d + rest} \equiv \text{just (d, rest)}
\]

3.2 A High-Level Example

To demonstrate the power of DT, we implement a simple tagged union, a structure in which the value of one field (the tag) determines the type of another.

data Tag : Set where

  boolean : Tag
  floating : Tag

taggedUnion : DT

taggedUnion = \text{sigma (leaf Tag) contents}

  where

  contents : Tag \rightarrow DT
  contents boolean = \text{leaf Bool}
  contents floating = \text{leaf Float}

Note that the format description taggedUnion is not low-level (although we know conversions from and to binary words exist for each of Tag, Bool, and Float).

4 Data Conversion

Although we can parse and pretty print low-level descriptions, it can sometimes be easier to define the more general high-level descriptions. This is particularly important when there is a complex dependency between various data fields: working with the low-level representation of information makes it much harder to describe the desired relation. We define the Conversion relation between two descriptions as follows:

data Conversion (t1 t2 : DT) : Set where

  convert : (\{ t1 \} : \left[ t1 \right]) \rightarrow

  (\{ t1 \} : \left[ t1 \right]) \rightarrow

  \text{convert} \text{tx t1} \rightarrow \text{maybe (}\left[ t1 \right]) \rightarrow

  ((x : \left[ t1 \right]) \rightarrow (\text{just} x)) \rightarrow

  \text{Conversion} \text{t1 t2}

We can convert t1 to t2 provided we have a semipartial isomorphism between the corresponding types (cf. partial isomorphisms [Rendel and Ostermann 2010]). Such a semipartial isomorphism consists of an encoding function (\text{\ perp}), a partial decoding function (\text{\ perp}), and a proof that decoding an encoded value succeeds and leaves the data intact.

There is an identity Conversion that leaves all data intact:

idConversion : \text{Conversion} \text{t1 t1}

idConversion = \text{convert} \text{id (}\lambda x \rightarrow \text{refl})

Similarly, we can show that our conversions are closed under composition.

We introduce a new data type, DTX, that can be used to transform the types stored in an existing description. Using such transformations, we can describe how to serialize high-level data independently of a (more high-level) format description. The definition of the DTX data type follows that of our description universe DT:

data DTX : DT \rightarrow \text{Set where}

  convert : \text{Conversion} \text{t1 t2} \rightarrow \text{DTX} \text{t1}
parseWithoutRest \{ t \} ill xs with parse \{ t \} ill xs
  ... | just (v, []) = just v
  ... | _ = nothing

Composing this parser with retractValue gives us a parser for the
high-level type length+word:

parse-length+word : List Bit \rightarrow \text{Maybe} [length+word]
parse-length+word = parseWithoutRest \{ length+word+enc \} ill
  \Rightarrow retractValue encode-length+word

With this definition of conversions between representations,
DTX, we have a method of shifting from a high-level description to
a low-level one.

4.1 Repeated Extension

Often it is convenient to chain together multiple transformations,
gradually converting a type to its low-level representation. As our
semipartial isomorphisms are closed under composition, we can
define the reflexive-transitive closure of our DTX data type as follows:

\textbf{data DTX*} : DT \rightarrow DT \rightarrow \text{Set}_1 \textbf{where}
  base : DTX* t t
  step : DTX* t1 t2 \rightarrow (tx : DTX t2) \rightarrow
          DTX* t1 (extendType2 tx)

The corresponding functions computing the underlying data
description and conversion functions are straightforward:

extendType* : DTX* t1 t2 \rightarrow DT
extendValue* : DTX* t1 t2 \rightarrow \{ t1 \} \mapsto \{ t2 \}
retractValue* : DTX* t1 t2 \rightarrow \text{Maybe} \{ t1 \}

Using DTX* we can chain together various conversions, gradually
shifting from a more abstract data type describing the desired
data to its low-level binary representation. This addresses the first
of the two problems we mentioned in at the end of Section 2. We
will tackle the remaining problem – forward dependencies in data
formats – in the coming section.

5 Inserting New Fields

In the examples we have seen so far, we have shown how to compute
new fields from the existing ones. In practice, however, dependencies
between fields may be arbitrary. As we mentioned previously,
the checksum in the IPv4 header occurs before all the data has been
parsed. To deal with such new fields, we will adapt our DTX data
type.

Previously, the DTX data type could describe conversions from
a high-level to low-level representation of the same data. By
adding a new constructor to the DTX data type, we will facilitate the
definition of other transformations on our DT type. In particular,
this new constructor, insert, will be used to insert a new field in
an existing description. To ensure that the insert constructor may
compute a new field from any data – even data that has not yet
been parsed – we need access to the entire top-level value of our
description. To ensure this information is available, we parametrise
our DTX type with the top-level description, top, as follows:

\textbf{data DTX} (top : DT) : DT \rightarrow \text{Set}_1 \textbf{where}
  convert : Conversion t1 t2 \rightarrow DTX top t1
  _\otimes_ : DTX top l \rightarrow DTX top r \rightarrow DTX top (l \otimes r)
  sigma : DTX top c \rightarrow ((x : [c]) \rightarrow DTX top (d x)) \rightarrow
          DTX top (sigma c d)
  insert : (t' : DT) \rightarrow \text{Side} \rightarrow ([\text{top}] \rightarrow [t']) \rightarrow
          DTX top t

The only modifications to the existing constructors is the addition
of the new type parameter, top. The new insert constructor is more
interesting. It takes three arguments: the type of the new field being
inserted (t'); the Side describing where to insert the new field; and
a computation describing how to compute a value of [t'] from
the parsed data [top]. The Side type simply records whether the
new field should come before or after the current data:

\textbf{data Side} : \text{Set where}
  left : Side
  right : Side

We use this Side information to compute new types and extend
values in the suitable direction. This becomes apparent when updat-
ing the following functions, adding a new branch to handle
insertions:

extendType : \{ t : DT \} \rightarrow DTX top t \rightarrow DT
extendValue : \{ t : DT \} \rightarrow (tx : DTX top t) \rightarrow
           \{ \text{top} \} \rightarrow \{ tx \} \rightarrow \{ \text{extendType} tx \}
retractValue : \{ t : DT \} \rightarrow (tx : DTX top t) \rightarrow
            \{ \text{extendType} tx \} \rightarrow \text{Maybe} \{ t \}
extendType (t = t) (insert t' left) = t' \otimes t
extendType (t = t) (insert t' right) = t \otimes t'
extendValue (insert t' left f) dtop d = (f dtop, d)
extendValue (insert t' right f) dtop d = (d, f dtop)
retractValue (insert t' left) (_, dr) = just dr
retractValue (insert t' right) (dl, _) = just dl

We can use the insertions to extend existing descriptions with
additional information. For example, we can extend our running
example with a checksum field before the actual data is received:

+checksum : DTX length+word+enc length+word+enc
+checksum = insert (leaf Bit) left checksum

where
  checksum : [length+word+enc] \rightarrow \text{Bit}

As we saw previously, we can still compute the resulting description
of type DT and generate its corresponding parser.

\textbf{Tying the knot} We will often want to refer to extensions where
both DT arguments are equal, that is, DTX t t for some t. For
convenience, we define top-level synonyms subscripted with an 's'
(an abbreviation of ‘self').

\textbf{data DTXt} : DT \rightarrow \text{Set}_1
\textbf{DTXt} t = DTX t t
extendTypet = extendType
extendValuet : \{ t : DT \} \rightarrow (tx : DTXt t) \rightarrow
            \{ t \} \rightarrow \{ extendType t tx \}
extendValuet tx d d = extendValuet tx d d
retractValuet tx d = retractValuet tx d
Checking inserted values Our implementation of the function retractValue simply discards inserted values. Especially when a checksum is involved, it would be beneficial to check if the parsed value matches the expected one. We can define a generic function for parsing and checking:

retractAndCheck : \{ t : DT \} \to (tx : DTX t) \to
\begin{align*}
&\begin{cases}
&\text{[ extendType } tx \mapsto \text{[ extendType } tx \mapsto \text{Bool} \mapsto \\
&\text{[ extendType } tx \mapsto \text{Maybe } \{ t \}}
\end{cases}
\end{align*}

This function is implemented by first running retractValue to discard any derived fields. If this succeeds, we can recompose their expected value using extendValue. Finally, we can check if the computed values equal the original value stored using the Boolean equality check that is passed as an argument to retractAndCheck.

It may seem disappointing that this check cannot be done immediately during parsing, yet this should not come as a surprise. The inserted fields may rely on data that has not yet been parsed; we need to completely parse the structure before we can decide if their value is valid or not. We can, however, provide the top-level function retractAndCheck, that both parses and validates values.

What about deletions? It is important to note that while we can handle insertions in this fashion, we want to avoid deletions of a leaf or subtree. Conceptually, the problem with deletion transformations is that they would drop data from high-level values in a way that makes it impossible to recover it from low-level values; this problem manifests itself when trying to implement retractValue for the delete constructor. Fortunately, we haven’t needed to use deletions anywhere and we think they would not be particularly useful (except perhaps for removing intermediate results).

5.1 Beyond Top-Level Insertion
While the insert constructor defined so far works well in many cases, it still has its shortcomings. We will try to illustrate the problems with its current definition in a small example. Consider the following format, consisting of a length field followed by a vector of natural numbers of that length:

vecNats : DT
vecNats = sigma (leaf N) (\lambda \, len \to leaf (Vec N len))

Now suppose we want to add an additional field to this description containing the maximum element of the vector when it is non-empty. To do so, we start to define the following extension of vecNats:

insertMax : DTX vecNats
insertMax = sigma copy iMax
where
iMax : (\lambda \, len : \mathbb{N}) \to DTX vecNats (leaf (Vec N len))
iMax zero = copy
iMax (suc n) = insert (leaf N) right maxVec
maxVec : \{ vecNats \} \to \mathbb{N}

Here insertMax copies the first component of the Sigma type; if the first component is zero, the empty vector is also copied. If the first component is greater than zero, we would like to insert a new field after the vector. To do so, we use the insert constructor and specify that we want to insert a new leaf \mathbb{N} to the right of the current field. Finally, we need to define how to compute the maximum element of a vector - this is where we run into a problem.

The type of maxVec, as dictated by insert, states that it must accept any top-level value. In this specific case, the type of the top-level value \{ vecNats \} is equal to \Sigma N (\lambda \, len \to Vec len) – that is, a vector of arbitrary length. Even if we have already learned that the vector is non-empty by pattern matching on the first element of the \Sigma-type in iMax, we still have to define a function that will compute the maximum element of any vector – even an empty one.

By manually inspecting the calls to maxVec, we can see that it will never be called with an empty vector as argument. It may therefore be tempting to return a dummy value when the vector is empty, but doing so would create room for error: the type is not precise and an inadvertent call to maxVec could introduce bugs and validate results into the output.

The underlying issue is that the type of the insert constructor is imprecise: it forces the function that computes the value of the new field to accept all values of the top-level type. To address this, we will allow the computation function to only take some of those values, namely those that actually contain t, the subtree of the top-level type that is currently being transformed.

Subtrees To explicitly reference a specific subtree of a data description of type DT, we define the Subtree relation below:

data Subtree (t : DT) : DT \to Set where
fst : Subtree t l \to Subtree t (l \otimes r)
snd : Subtree t r \to Subtree t (l \otimes r)
\pi1 : Subtree t c \to Subtree t (\pi \in \Sigma c d)
\pi2 : (x : \{ c \}) \to Subtree t (\pi \in \Sigma c d)
stop : Subtree t t

A value of type Subtree t_2 t_3 singles out a subtree t_2 : DT in the larger description t_1 : DT. For the sake of convenience, we introduce the following type synonym:

_ \leftrightarrow_ : DT \to DT \to Set
\begin{align*}
t_1 \leftrightarrow t_2 & = \text{Subtree } t_2 t_1
\end{align*}

This notation more clearly suggests that t_2 is 'smaller' than t_1. It is easy enough to show that this subtree relation is transitive:

\begin{align*}
_\leftrightarrow_ \circ _\leftrightarrow_ & \to _\leftrightarrow_ \\
\pi_1 & : _\leftrightarrow_ \circ _\leftrightarrow_ \to _\leftrightarrow_ \\
\pi_2 & : _\leftrightarrow_ \circ _\leftrightarrow_ \to _\leftrightarrow_ \\
\end{align*}

Using this subtree relation, we would like to define a projection function that selects the designated subtree:

select : t_1 \leftrightarrow t_2 \to \{ t_1 \} \leftrightarrow \{ t_2 \}

When defining the case for the second component of dependent pairs, \pi_2, however, we run into a problem:

select (\pi_2 \times t) (\pi \in \Sigma c d) = ... 

The problem is that the first component of the tree in which we are selecting (\pi \in \Sigma c d) may not be the same as the first component in our description subtree (\pi). As a result, the recursive call that we would like to make – select t y – will not type check, and we appear to be stuck.

To resolve this, we introduce a predicate Select, that captures the graph of the select function, restricted to those inputs where the selection is guaranteed to succeed. The constructors of Select closely follow those of Subtree:

data DTX (top : DT) : (t : DT) → (s : top ▷ t) → Set₁ where
  convert : Conversion t₁ t₂ → DTX top t₁ s 
  ⊗∥ : DTX top l (s ⊗∥ fst stop) → DTX top r (s ⊗∥ snd stop) → DTX top (l ⊗∥ r) s 
  sigma : DTX top c (s ⊗∥ π₁ stop) → ((x : [ c ]) → DTX top (d x) (s ⊗∥ π₂ x stop)) → DTX top (sigma c d) s 
  insert : (t' : DT) → Side → ((d : [ top ]) → (v : [ t ])) → Select s d v → [ t' ] → DTX top t s 

Figure 1. Transformation data type DTX

data Select : t₁ → t₂ → [ t₁ ] → [ t₂ ] → Set₁ where
  fst : Select s x v → Select (fst s) (x , y) v 
  snd : Select s y v → Select (snd s) (x , y) v 
  π₁ : Select s c v → Select (π₁ s) (sigma c d) v 
  π₂ : Select s d v → Select (π₂ s) (sigma x d) v 
  stop : Select stop t t 

The only interesting case is that for the second component of a dependent products, π₂. There we require that the argument in our selection x and the first component of the sigma coincide. Note that we have left out numerous implicit arguments from these definitions, not all of which can be inferred by Agda.

Equipped with the Select relation, we can define a final version of the key transformation data type DTX in Figure 1. This final version explicitly records the relation the between the top-level type and the type currently being extended using an additional Subtree index.

In each inductive occurrence of DTX, the appropriate Subtree is appended to the selection being constructed; crucially, in the sigma case, the current value of the first component (x) is stored in the selection being constructed. In the insert constructor, the insertion function now receives a second argument: an object describing that there exists some smaller value of type [ t ] such that we can traverse the top-level data d along the current Subtree pointer s to find the desired value at the subtree.

Using this definition, we can revisit our insertMax transformation. The definition is hardly changed. Using the same constructors that we did previously, the type of the problematic maxVec function that we must provide becomes:

maxVec : { s : vecNats ▷ leaf (Vec N (suc n))} → 
  (d : [ vecNats ]) → 
  (v : Vec N (suc n)) → 
  Select s d v → N

That is, we are now guaranteed that the vector is non-empty, allowing us to compute its maximum element.

Note that this definition of insert is strictly more general than the one we have seen previously. To recover the previous version, we simply pass an insertion function that ignores the Select and [ t ] arguments:

insertSimple : (t' : DT) → Side → 
  (□ [ top ]) → [ t' ]) → DTX top t s 
insertSimple t' s f = insert t' s (λ d _ _ → f d)

This completes the definition of our DTX data type, that allows us to describe well-behaved changes to our data descriptions DT. In the coming section, we will apply these tools to a more realistic data format.

6 Case Study: IPv4

We are now ready to define a precise data format of IPv4 packets. We will do so in a series of steps. We will begin by giving a data description, that is a value of type DT, that captures the ‘essence’ of IPv4 packets, even if it leaves out certain fields or details regarding the binary representation of this information. We will then proceed by defining a series of transformations, values of type DTX, that convert this high-level description to actual IPv4 packets. We will start by designing the our initial DT format to be as user-friendly as possible.

6.1 The Initial Description

Table 1 illustrates the structure of IPv4 packet format. While we do not want to cover the individual fields in precise detail, there are six different categories of fields in which they can be divided:

<table>
<thead>
<tr>
<th>Constants</th>
<th>The Version field must contain the constant value 0100.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bounded natural numbers</td>
<td>Fields such as the Internet Header Length (IHL), the Total Length, the Identification, the Fragment Offset, the Time to Live, the Header Checksum, and the Addresses contain simple bounded natural numbers without limitations that should be encoded using big-endian binary encoding. Since they have no limitations and any special meaning must be assigned to them at a higher level, they can be represented as Fin (2 ^ n), where n is the length of the field in bits.</td>
</tr>
<tr>
<td>Enumerations</td>
<td>The Explicit Congestion Notification and the Protocol can be seen as finite enumerations. They can be represented using simple algebraic data types. We will define functions that encode and decode these values to Boolean vectors. If desired, an unknown constructor can be added to the Protocol data type to cover protocols that are too uncommon to list.</td>
</tr>
<tr>
<td>Flags</td>
<td>The Differentiated Services Code Point (DSCP) and the Flags are flag structures. Because each flag is usually described by some fixed number of bits (the flags are &quot;orthogonal&quot;), these can be implemented as multiple enumeration fields.</td>
</tr>
<tr>
<td>Options</td>
<td>These have a more complex underlying structure, but we will assume that this has been dealt with (perhaps by a separate, smaller format description), and just encode them as a Boolean vector. This vector has length 32 * len, where len is a number between 0 and 10: the IHL field is only four bits long, which means the header is at most 15 words long, and the fixed fields span 5 words.</td>
</tr>
<tr>
<td>Data</td>
<td>Although not part of the header, data is part of the packet, and so a description of packets must encompass it. Data is a Boolean vector, the length of which is constrained in a complex way that we describe below.</td>
</tr>
</tbody>
</table>
The Internet Header Length and Total Length are involved in a non-trivial calculation. Consequently, we aim to insert those into the data by applying extensions; all other fields will need to be present in the initial description.

**Fixed-Length Fields**

Although we developed a complete description of IPv4, having discussed these categories, we will describe a small part of the complete implementation. The Explicit Congestion Notification, for example, can be represented as an enumeration with four elements.

**data ECN : Set where**

Non-ECT = ECN
ECT0 = ECN
ECT1 = ECN
CE = ECN

We can define the serialization and deserialization functions easily enough:

ECN→Bit = ECN → Vec Bit 2
ECN→Bit Non-ECT = 1 : 1 : []
ECN→Bit ECT0 = 0 : 1 : []
ECN→Bit ECT1 = 1 : O : []
ECN→Bit CE = O : O : []

Bit→ECN = Vec Bit 2 → ECN
Bit→ECN (1 : 1 : []) = Non-ECT
Bit→ECN (0 : 1 : []) = ECT0
Bit→ECN (1 : O : []) = ECT1
Bit→ECN (O : O : []) = CE

ECN→Bit = (x : ECN) → Bit→ECN (ECN→Bit x) ≡ x

Similar definitions can be developed for the other simple fields storing finite enumerations.

**Variable-Length Fields**

We are left with the two more complex variable-length fields, Options and Data. We will use two natural numbers for these lengths: OL (Option Length), which counts 32-bit words, and DL (Data Length), which counts bytes. The combined upper bound of these values is given by the Total Length field: the total number of bytes cannot equal or exceed $2^{16}$ – which, using the identifiers chosen, translates to the property $4 \times (5 + OL) + DL < 2^8$. Considering that we already had the restriction that $OL \leq 10$, the type of a pair of numbers that satisfies exactly the appropriate conditions can be expressed using the following sigma type:

$$\text{Lengths} = \Sigma (N \times N)$$

$$\lambda \{(DL, OL) \mapsto$$

$$OL \leq 10 \times 4 \times (5 + OL) + DL < 2^{16}\}$$

This type is complex, but this is a direct consequence of the IPv4 specification. We could get a simpler type by letting DL be a bounded natural number whose upper bound is chosen such that even the greatest OL would not make the Total Length exceed $2^{16}$. Although the low-level output of such a type would be valid and thus “downwards correctness” would be preserved, this sacrifices “upwards correctness”: there would be low-level packets that are valid according to the specification, but cannot be parsed into the high-level type.

Now the following data type definition approximating the IPv4 packet format can be defined as follows:

**IPv4Type : DT**

**IPv4Type = sigma header1 options+data where**

header1 : DT

header1 = leaf Lengths

⊕ leaf ECN → ECN

⊕ leaf (Vec Bit 32) → Source

⊕ leaf (Vec Bit 32) → Destination

options+data : [] header1 [] → DT

options+data (((DL, OL), ), ) = leaf (Vec Bit (32 * OL)) → Options

⊕ leaf (Vec Bit (8 * DL)) → Data

At the top-level, we introduce a dependent pair. The first component of this pair (header1) contains the lengths and three fields: the Explicit Congestion Notification, the Source Address; and the Destination Address. For the sake of brevity, we have omitted several other fields – they can be implemented as described above and added before and after the Explicit Congestion Notification. The second component (options+data) contains the Options and Data vectors, whose lengths depend on information from the header.

**6.2 Transformations**

To bring the initial description down to a low-level description, we apply three transformations on the approximation IPv4Type that we defined above. We will discuss each of these transformations separately.

---

**Table 1. IPv4 header format**

<table>
<thead>
<tr>
<th>Field</th>
<th>Location in IPv4</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Version</td>
<td>0</td>
<td>Number of IPv4 versions (4)</td>
</tr>
<tr>
<td>Header Length</td>
<td>1</td>
<td>Total number of bytes</td>
</tr>
<tr>
<td>Total Length</td>
<td>1</td>
<td>Number of bytes</td>
</tr>
<tr>
<td>Identification</td>
<td>2</td>
<td>Destination address</td>
</tr>
<tr>
<td>Flags</td>
<td>3</td>
<td>Source address</td>
</tr>
<tr>
<td>Fragment Offset</td>
<td>4</td>
<td>Data</td>
</tr>
<tr>
<td>DSCP</td>
<td>5</td>
<td>Differentiated Services Code Point</td>
</tr>
<tr>
<td>ECN</td>
<td>6</td>
<td>Explicit Congestion Notification</td>
</tr>
<tr>
<td>Time to Live</td>
<td>7</td>
<td>Time to live</td>
</tr>
<tr>
<td>Protocol</td>
<td>8</td>
<td>Protocol identifier</td>
</tr>
<tr>
<td>Header Checksum</td>
<td>9</td>
<td>Checksum</td>
</tr>
<tr>
<td>Source Address</td>
<td>10</td>
<td>Source address</td>
</tr>
<tr>
<td>Destination Address</td>
<td>11</td>
<td>Destination address</td>
</tr>
<tr>
<td>Options</td>
<td>12</td>
<td>Options</td>
</tr>
<tr>
<td>Padding</td>
<td>13</td>
<td>Padding</td>
</tr>
</tbody>
</table>

---
First Transformation: Mixing
The first transformation handles two separate aspects: adding the constant Version field and transforming the convenient Lengths into the fields as specified by the protocol.

The former is easily carried out by insert, but the latter is a different story. Not only do the two fields have to be calculated from the Lengths, they also have to be inserted into the fields during pretty printing and extracted during parsing. To ensure we get a general solution, we designed a new auxiliary function to produce a DTX for situations similar to this. The idea is to reuse the insertion mechanism already present in DTX for calculating and inserting the data (“mixing”) into the fields during pretty printing. For extraction, we have no features to reuse, and we require an explicit recovery function to extract and calculate the high-level values.

\[
\text{mix} : (tx : \text{DTX}(l \otimes r)) \rightarrow \\
\quad (rf : [\text{extendType} tx] \rightarrow \text{Maybe}[1]) \rightarrow \\
\quad ((d_1 : [1]) \rightarrow (d_2 : [r]) \rightarrow \\
\quad \quad \text{rf}(\text{extendValue} tx (d_1, d_2)) \equiv \text{just } d_1) \rightarrow \\
\quad \text{DTX top}(l \otimes r)
\]

As this type is rather daunting, we have illustrated the intended implementation as follows:

On the left we see the source type of the transformation – the pair \(l \otimes r\) – and on the right the result type. The result type is the result of transforming \(r\) along the transformation \(tx\), as represented by the solid arrow. That transformation’s top-level type is \(l \otimes r\), and information from that pair’s value can be used by insertion functions in insert; this is what the dotted arrow represents. Finally, the dashed arrow represents the recovery function \(rf\), whose job it is to extract the data from the extended second component and recover the first component of the pair. Of course, a proof of the round trip property is also required.

With this auxiliary function in our toolbox, we can define this first transformation: adding in the IHL and the TL proceeds by the use of insert, performing the required arithmetic on OL and DL; recovery involves some pattern matching, but is not difficult. The first component of the result type of this transformation is the following:

\[
\text{header}_2 : \text{DT} \\
\text{header}_2 = \\
\quad \text{leaf } \text{ECN} \quad \text{-- } \text{ECN} \\
\quad \otimes \text{leaf } (\text{Fin}(2^{\wedge}4)) \quad \text{-- } \text{IHL} \\
\quad \otimes \text{leaf } (\text{Fin}(2^{\wedge}16)) \quad \text{-- } \text{TL} \\
\quad \otimes \text{leaf } (\text{Vec Bit 32}) \quad \text{-- } \text{Source} \\
\quad \otimes \text{leaf } (\text{Vec Bit 32}) \quad \text{-- } \text{Destination}
\]

The second component of the sigma type (the options and data) can simply be copied.

Second Transformation: Binary Encoding
The second transformation ensures that all fields are low-level. The IHL and TL are processed using using big-endian binary encoding, while ECN→Bit and the two related functions are used for the ECN. Again, the second component of the sigma type is simply copied. After defining this transformation, the first component of our sigma type becomes:

\[
\text{header}_3 : \text{DT} \\
\text{header}_3 = \\
\quad \text{leaf } (\text{Vec Bit 2}) \quad \text{-- } \text{ECN} \\
\quad \otimes \text{leaf } (\text{Vec Bit 4}) \quad \text{-- } \text{IHL} \\
\quad \otimes \text{leaf } (\text{Vec Bit 16}) \quad \text{-- } \text{TL} \\
\quad \otimes \text{leaf } (\text{Vec Bit 32}) \quad \text{-- } \text{Source} \\
\quad \otimes \text{leaf } (\text{Vec Bit 32}) \quad \text{-- } \text{Destination}
\]

Third Transformation: Checksum Insertion
The third transformation calculates the checksum and inserts it at the appropriate location. Now that the data type is low-level, the checksum calculation itself is conceptually easy. The only serious hurdle is that because of the first two transformations, the type of the second component (which we need to pattern-match on to extract the Options) is no longer the high-level Lengths as it was initially; instead, the transformations that we have applied so far have added two applications of retractValue. To process these, requires several with-clauses to destruct these applications of retractValue, recovering the original Lengths and allowing easy access to Options. After the checksum is inserted, the first component of the sigma becomes:

\[
\text{header}_4 : \text{DT} \\
\text{header}_4 = \\
\quad \text{leaf } (\text{Vec Bit 2}) \quad \text{-- } \text{ECN} \\
\quad \otimes \text{leaf } (\text{Vec Bit 4}) \quad \text{-- } \text{IHL} \\
\quad \otimes \text{leaf } (\text{Vec Bit 16}) \quad \text{-- } \text{TL} \\
\quad \otimes \text{leaf } (\text{Vec Bit 16}) \quad \text{-- } \text{Checksum} \\
\quad \otimes \text{leaf } (\text{Vec Bit 32}) \quad \text{-- } \text{Source} \\
\quad \otimes \text{leaf } (\text{Vec Bit 32}) \quad \text{-- } \text{Destination}
\]

IsLowLevel Instance
The final piece of the puzzle is the instance of IsLowLevel for the result type of the third transformation. Because of the presence of retractValue in the type of the second component as we mentioned above, this is not entirely trivial. This requires a few auxiliary definitions, but many of the more tedious parts can be inferred using Agda’s instance arguments.

6.3 Testing
To test our description, we have checked whether the packets produced by our pretty printer would be recognized by official parsers. Although there are not many IPv4 parsers defined in Haskell that can handle the entire protocol (including, for example, options), the least we could do was test the checksum calculation. We used the checksum implementation of the network-house Haskell library
to test our implementation. Unfortunately, one defect was discovered: it turns out that we had overlooked the endianness of IPv4 and used a little-endian encoding where a big-endian encoding was expected. Although this was easily corrected by appropriate applications of reverse, this incident highlights an important fact: the (formally verified) round trip property ensures we can read back whatever we write, but it cannot exclude the possibility of format descriptions not matching the specification. Careful reading remains necessary, and comparing with other implementations helps to rule out mistakes in interpretation.

7 Discussion

7.1 Related Work

**Domain specific languages for data description** Numerous domain specific languages (DSLs) have been designed for the broad purpose of describing the format of binary or similarly low-level data, each tackling the subject from its own perspective and each providing more or less support for certain formatting mechanisms and constructs.

PacketTypes, a DSL by McCann and Satish [2000], has a role analogous to ‘yacc’, in that it abstracts away the packet grammar into a separate specification language, and automatically creates recognisers for the packets. It comes with the basic primitive type bit and a “repeat n times” operator for forming words of bits. Record syntax, much like the C struct, is available for specifying a succession of fields (a “product record”) and a choice between many fields (a “sum record”).

Types can be refined to yield new types; refinements fix values of fields and allow overlaying of fields with fields of a more restrictive type. Certain classes of restrictions can be added to data types using where clauses, such as fieldA#numbytes <= 10. Although arbitrarily dependent types are not supported, where clauses can express the constraint that the length of one field must be equal to the value of some earlier field. Interestingly, this feature is not considered very important, as it is not highlighted in the paper.

PADS (Fisher et al. [2006]) tries to lessen the development effort needed for the processing of “ad-hoc data”, therefore focusing less on the formal aspects (e.g. correctness) of the problem. Its syntax is C-like, with keywords Pstructure and Punion for “product records” and “sum records”, respectively. The former of these can include literal strings “such as this one” which are parsed and pretty printed as constants. Each field is processed directly after its predecessor; the Precord modifier lets the user to specify a delimiter to parse and pretty print between fields.

PADS supports parametrizing types by values, in effect a rudimentary form of dependent types. For example, consider the PADS type Point16_FW(:3 * 1en:) that represents an unsigned 16-bit number to be read and written to exactly three times as many characters as the value earlier read or written as 1en. As expressions (delimited by colons) can be arbitrary C expressions, this parametrization is flexible and powerful; on the other hand, this design decision ties the entire system to C, which is notoriously hard to analyze and reason about.

Finally, Devil, a DSL and tool package by Méryillon et al. [2000], is an Interface Definition Language (IDL) for hardware functionalities. Although its advanced features focus on various hard-to-write but common procedures used in low-level IO access, its basic features are conceptually similar to the previous systems: it provides low-level registers and high-level variables and allows users to describe how data should be transformed to and from the higher level (“reading” and “writing”, respectively). It does not seem to contain any form of dependent typing. Importantly, various forms of verification are supported by Devil tools. These include verification of the correctness of Devil descriptions as well as runtime checks in generated code that ensure read data is correctly typed.

PacketTypes, PADS and Devil all come with code generators that can generate C code for “parsing” data from one description into another from a data format description.

**Generic programming** There has been a great deal of work on generic programming in the context of dependently typed languages. This work can be traced back as far as Martin-Löf’s work on universes [1984]. More recently, Altenkirch and McBride [2003] started to explore the relation between universes and the existing work on generic programming in Haskell [Backhouse et al. 1998]. This line of work was continued by Morris et al. [2007].

The approach taken in this paper most closely related to the work on **ornaments** [Dagand 2013; McBride 2010], providing a language to describe the relation between data types. Our universe transformations provide a similar ‘language’ for modifying data descriptions, inserting new fields or changing data representation.

**Embedded languages** Besides stand-alone languages, there are several different embedded DSLs that have pursued a similar line of research. Oury and Swierstra [2008, section 3] present, as an example of the power of dependent types, a prototypical Agda EDSL for describing data types in the context of parsing and pretty printing. The advantage of using Agda as a host language is that created descriptions can directly be reasoned about in a well-understood proof framework.

Wang and Gaspes [2008] have explored how to embed a data description language in Haskell. Brady [2011] has also considered parsing IPv4 packets using the dependently typed language Idris [Brady 2013]. Brady shows how to define a monadic language for splitting packets into pieces and explicitly checking whether these are well-formed. In contrast to the approach suggested here, where the parser and pretty printer are generated from a description, Brady’s work focuses on defining the packet parser. More recently, Delaware [2017] has started exploring the usage of Coq to embed a similar language.

**Parsing and pretty printing** There is a large body of work on parser combinator libraries and pretty printing in functional languages. We will only mention the most closely related work here that describes both parsing and pretty printing, and relates the two by means of a round trip property.

Rendel and Ostermann [2010] have designed a domain specific language embedded in Haskell for describing both parsing and pretty printing simultaneously. This has directly inspired some of our terminology regarding semi-partial isomorphisms. Danielsson [2013] has shown how to write a pretty printing library that guarantees a similar round trip property.

7.2 Further Work

While our case study has demonstrated that the transformations we have defined are powerful enough to describe realistic data formats, it has inspired us to consider additional areas of future work. First and foremost, we may want to decorate our data descriptions with
an explicit name or string. This not only serves as documentation explaining the structure of our description in more familiar terms, but could also be used to produce more meaningful error messages when parsing fails. Alternatively, many descriptions can be generated automatically from Agda’s existing record types using Agda’s reflection mechanism [Van Der Walt and Swierstra 2012]. This would make the data contained in our descriptions a great deal easier to define and manipulate.

Much of the effort in the development of the case study was spent on proving various tedious (in)equalities arising from constraints between the fields of IPv4 packets. Where other interactive proof assistants such as Coq [2004] provides tactics such as omega to discharge these proofs automatically, they typically require quite some effort in Agda. Providing hooks to integrate proof automation, such as Agda’s ring solver [Bove et al. 2009] or proof search mechanisms [Kokke and Swierstra 2015], would facilitate the definition of such descriptions.

Conclusions Data type generic programming is a powerful tool, enabling us to derive functionality from types automatically. Yet the functionality generated in this fashion may not always be precisely what users have in mind. This paper shows how to layer additional transformations on top of such type descriptions to customize the generated functionality further. We believe that these techniques may be applicable in other domains, combining the generality of data type generic programming and flexibility of hand-written code.

References