Generic packet descriptions

Verified parsing and pretty printing of low-level data

Marcell van Geest
Utrecht University
marcell@marcell.nl

Wouter Swierstra
Utrecht University
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 datatype generic programming, dependent types, parsing, pretty printing

ACM Reference format:
Marcell van Geest and Wouter Swierstra. 2017. Generic packet descriptions. In *Proceedings of ACM SIGPLAN Conference on Type-driven Development, Oxford, UK, September 03, 2017 (TyDe'17),* 12 pages.
DOI: 10.1145/nnnnnnn.nnnnnnn

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 datatype 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 nails 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.

TyDe'17, Oxford, UK
2017. 978-x-xxxx-xxxx-x/YY/MM…$15.00
DOI: 10.1145/nnnnnnn.nnnnnnn

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. 2009a; Norell 2007).

To warm up, we define the universe of format types (FT), as follows:

```
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 ft_1 example: the type corresponding to ⟦ ft_1 ⟧ 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) = pp t₁ ++ pp t₂
```

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, [])
```

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 domain specific language.

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 *dependencies* 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:

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

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 constructor 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 straightforward: 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 universe, 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 (tt , 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:

```
ft₃ : FT
ft₃ = 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 litle-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 ft_3, 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
  ⟦ l ⊗ r ⟧      = ⟦ l ⟧ × ⟦ 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 Set_1. 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).

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 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:

```
roundTrip : (ill : IsLowLevel t) →
  (rest : List Bool) → (d : ⟦ t ⟧) →
  parse ill (pp ill d ++ rest) ≡ just (d , rest)
```

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 described the desired relation. We define the Conversion relation between two descriptions as follows:

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

We can convert t_1 to t_2 provided we have a *semipartial isomorphism* between the corresponding types (Rendel and Ostermann 2010). Such a semipartial isomorphism consists of an encoding function (⇓), a partial decoding function (⇑), and a proof decoding an encoded value succeeds and leaves the data intact.

There is an identity Conversion that leaves all data intact:

```
idConversion : Conversion t₁ t₁
idConversion = convert id id (λ x → refl)
```

Similarly, we can show that our conversion 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 → Set₁ where
  convert : Conversion t₁ t₂ → DTX t₁
  _⊗_ : DTX l → DTX r → DTX (l ⊗ r)
  sigma : DTX c → ((x : ⟦ c ⟧) → DTX (d x)) →
          DTX (sigma c d)
```

The cases for sigma and _⊗_ follow the structure of the description. The only interesting constructor, convert, describes a change in description by giving a Conversion between the values inhabiting the old and the new description. Crucially, such changes are guaranteed to *preserve information*.

There is a trivial transformation that does no interesting conversion:

```
copy : DTX t
copy = convert idConversion
```

If we revisit the simple ft_3 example that we saw previously, we can now define this in two steps:

```
length+word : DT
length+word =
  sigma (leaf ℕ) (λ len → leaf (Vec Bits len))
length+word+enc : DTX dt
length+word+enc = sigma int32 (λ len → copy)
  where
    int32 : Conversion (leaf ℕ) (leaf (Vec Bit 32))
    copy : {dt : DT} → DTX dt
```

Crucially, we have now decoupled the *dependencies* in our formats and the way in which data is *encoded*. Note that we are somewhat sloppy here: there is no semipartial isomorphism between natural numbers and Vec Bit 32. For the sake of convenience, we chose to use naturals numbers in this example instead of the more precise Fin (2 ^ 32) or Σ (n : ℕ) . (n < 32).

We can assign semantics to the values of type DTX in two ways. Firstly, we can compute a new description of type DT, arising from applying DTX. Secondly, a value of type DTX t describes how to convert any data of type ⟦ t ⟧ to a value of this new description.

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

Note that due to the contravariance introduced by the function type in the sigma constructor, defining these functions requires the map in the opposite direction also:

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

Using these definitions, we can now generate a parser for the length+word+enc example described above:

```
result : Set
result = ⟦ extendType length+word+enc ⟧
parseResult : List Bit → Maybe (result × List Bit)
parseResult = parse {extendType length+word+enc} ll
  where
  ll : IsLowLevel (extendType length+word+enc)
```

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

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 reflexitive-transitive closure of our DTX data type as follows:

```
data DTX* : DT → DT → Set₁ where
  base : DTX* t t
  step : DTX* t₁ t₂ → (tx : DTX t₂) →
         DTX* t₁ (extendType₂ tx)
```

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

```
extendType*  : DTX* t₁ t₂ → DT
extendValue* : DTX* t₁ t₂ → ⟦ t₁ ⟧ → ⟦ t₂ ⟧
retractValue* : DTX* t₁ t₂ → ⟦ t₂ ⟧ → 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 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:

```
data DTX (top : DT) : DT → Set₁ where
  convert : Conversion t₁ t₂ → DTX top t₁
  _⊗_ : DTX top l → DTX top r → DTX top (l ⊗ r)
  sigma : DTX top c → ((x : ⟦ c ⟧) → DTX top (d x)) →
          DTX top (sigma c d)
  insert : (t' : DT) → Side → (⟦ top ⟧ → ⟦ t' ⟧) →
           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:

```
data Side : 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 updating the following functions, adding a new branch to handle insertions:

```
extendType : {t : DT} → DTX top t → DT
extendValue : (tx : DTX top t) →
  ⟦ top ⟧ → ⟦ t ⟧ → ⟦ extendType tx ⟧
retractValue : (tx : DTX top t) →
  ⟦ extendType tx ⟧ → Maybe ⟦ t ⟧

extendType {t = t} (insert t' left _)  = t' ⊗ t
extendType {t = t} (insert t' right _) = t ⊗ 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:

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

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

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').

```
DTX_s : DT → Set_1
DTX_s t = DTX t t

extendType_s : {t : DT} → DTX_s t → DT
extendType_s = extendType
extendValue_s : {t : DT} → (tx : DTX_s t) →
  ⟦ t ⟧ → ⟦ extendType_s tx ⟧
extendValue_s tx d = extendValue tx d d

retractValue_s : (tx : DTX_s t) →
  ⟦ extendType_s tx ⟧ → Maybe ⟦ t ⟧
retractValue_s tx d = retractValue 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 to perform this check:

```
check : (tx : DTX_s t) →
  (⟦ extendType_s tx ⟧ → ⟦ extendType_s tx ⟧ → Bool) →
  ⟦ extendType_s tx ⟧ → Maybe ⟦ t ⟧
```

This function is implemented by first running retractValue_s to discard any derived fields. If this succeeds, we can recompute their expected value using extendValue_s. 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 check.

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 a top-level function that both parses and validates data descriptions by first calling parse and subsequently validating the data using the check function.

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 removal is that it drops 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 remove constructor.

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) (λ len → 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_s vecNats
insertMax = sigma copy iMax
  where
  iMax : (len : ℕ) → DTX vecNats (leaf (Vec ℕ len))
  iMax zero    = copy
  iMax (suc n) = insert (leaf ℕ) right maxVec
  maxVec : ⟦ vecNats ⟧ → 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 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 Σ ℕ (λ len → 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 Σ-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:

```
data DTX (top : DT) : (t : DT) → (s : Subtree t top) → Set₁ where
  convert : Conversion t₁ t₂ → DTX top t₁ s
  _⊗_ : DTX top l (fst stop ◂◂ s) → DTX top r (snd stop ◂◂ s) → DTX top (l ⊗ r) s
  sigma : DTX top c (π₁ stop ◂◂ s) → ((x : ⟦ c ⟧) → DTX top (d x) (π₂ x stop ◂◂ s)) → 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

the type is not precise and an inadvertent call to maxVec could introduce bogus 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 → Set where
  fst  : Subtree t l → Subtree t (l ⊗ r)
  snd  : Subtree t r → Subtree t (l ⊗ r)
  π₁   : Subtree t c → Subtree t (sigma c d)
  π₂   : (x : ⟦ c ⟧) → Subtree t (d x) →
         Subtree t (sigma c d)
  stop : Subtree t t
```

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

```
_◃_ : DT → DT → Set
t₁ ◃ t₂ = Subtree t₁ t₂
```

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

```
_◂◂_ : t₁ ◃ t₂ → t₂ ◃ t₃ → t₁ ◃ t₃
```

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

```
select : t₁ ◃ t₂ → ⟦ t₂ ⟧ → ⟦ t₁ ⟧
```

When defining the case for the second component of dependent pairs, π_2, however, we run into a problem:

```
select (π₂ x t) (x' , y) = ...
```

The problem is that the first component of the tree in which we are selecting x' may not be the same as the first component in our original selection x. 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 Select : t₁ ◃ t₂ → ⟦ t₂ ⟧ → ⟦ t₁ ⟧ → 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 x s) (sigma x d) v
  stop : Select stop t t
```

The only interesting case is that for the second component of a dependent products, π_2. 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 : leaf (Vec ℕ (suc n)) ◃ vecNats} →
  (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.

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:

Constants The Version field must contain the constant value 0100.

Bounded natural numbers 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.

Enumerations 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.

Flags 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 "orthogonal"), these can be implemented as multiple enumeration fields.

Options These have a more complex underlying structure, but we will assume that 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.

Data 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.

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 = I :: I :: []
ECN→Bit ECT0    = O :: I :: []
ECN→Bit ECT1    = I :: O :: []
ECN→Bit CE      = O :: O :: []

Bit→ECN : Vec Bit 2 → ECN
Bit→ECN (I :: I :: []) = Non-ECT
Bit→ECN (O :: I :: []) = ECT0
Bit→ECN (I :: 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 * (5 + OL) + DL < 2 ^ 16. Considering that we

0	1	2	3	4	5	6	7	8	9	0	1	2	3	4	5	6	0	8	9	0	1	2	3	4	5	6	7	8	9	0	1
Version				IHL				DSCP						ECN		Total Length															
Identification																Flags		Fragment Offset													
Time to Live								Protocol								Header Checksum															
Source Address																															
Destination Address																															
Options																													Padding		

Table 1. IPv4 header format

already had the restriction that OL $\leqslant$ 10, the type of a pair of numbers that satisfies exactly the appropriate conditions can be expressed using the following sigma type:

```
Lengths = Σ (ℕ × ℕ)
            (λ {(DL , OL) →
              OL ⩽ 10 × 4 * (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 (header_1) 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.

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.

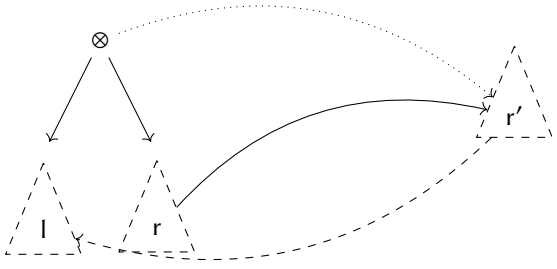
First transformation: mixing

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.

```
mix : (tx : DTX (l ⊗ r) r) →
  (rf : ⟦ extendType tx ⟧ → Maybe ⟦ l ⟧) →
  ((d₁ : ⟦ l ⟧) → (d₂ : ⟦ r ⟧) →
    rf (extendValue tx (d₁ , d₂) d₂) ≡ just d₁) →
  DTX top (l ⊗ 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 ⊗ 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 ⊗ 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:

```
header₂ : DT
header₂ =
  leaf ECN              -- ECN
  ⊗ leaf (Fin (2 ^ 4))  -- IHL
  ⊗ leaf (Fin (2 ^ 16)) -- TL
  ⊗ leaf (Vec Bit 32)   -- Source
  ⊗ leaf (Vec Bit 32)   -- Destination
```

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

Second transforming: 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:

```
header₃ : DT
header₃ =
  leaf (Vec Bit 2)      -- ECN
  ⊗ leaf (Vec Bit 4)    -- IHL
  ⊗ leaf (Vec Bit 16)   -- TL
  ⊗ leaf (Vec Bit 32)   -- Source
  ⊗ leaf (Vec Bit 32)   -- 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:

```
header4 : DT
header4 =
  leaf (Vec Bit 2)      -- ECN
  ⊗ leaf (Vec Bit 4)    -- IHL
  ⊗ leaf (Vec Bit 16)   -- TL
  ⊗ leaf (Vec Bit 16)   -- Checksum
  ⊗ leaf (Vec Bit 32)   -- Source
  ⊗ leaf (Vec Bit 32)   -- 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.

Testing

To test our description, we have tested whether the packets produced by our pretty-printer would be recognized by official parsers. Although there is 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. The formally verified round trip property did not exclude the possibility of such bugs.

7 Discussion

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, a DSL and related tools by Fisher and Gruber (2005), 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 `Pstruct` 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 allows 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. The example PADS type `Puint16_FW(:3 * len:)` 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 `len`. As expressions (delimited by colons) can be arbitrary C expressions, this parametrisation 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 Mérillon 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 the created descriptions can directly be reasoned about in a well-understood proof framework.

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.

Parsing and pretty printing There is a large body of work on parser combinators 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.

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. 2009b) 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 deriving functionality from type 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

Thorsten Altenkirch and Conor McBride. 2003. Generic programming within dependently typed programming. In *Generic Programming*. Springer US, 1–20.

Thorsten Altenkirch, Conor McBride, and Peter Morris. 2007. Generic programming with dependent types. In *Datatype-Generic Programming*. Springer, 209–257.

Roland Backhouse, Patrik Jansson, Johan Jeuring, and Lambert Meertens. 1998. Generic programming. In *International School on Advanced Functional Programming*. Springer Berlin Heidelberg, 28–115.

Ana Bove, Peter Dybjer, and Ulf Norell. 2009a. A Brief Overview of Agda–A Functional Language with Dependent Types. In *TPHOLs*, Vol. 5674. Springer, 73–78.

Ana Bove, Peter Dybjer, and Ulf Norell. 2009b. A brief overview of Agda–a functional language with dependent types. In *International Conference on Theorem Proving in Higher Order Logics*. Springer, 73–78.

Edwin Brady. 2011. Idris: systems programming meets full dependent types. In *Proceedings of the 5th ACM workshop on Programming languages meets program verification*. ACM, 43–54.

Edwin Brady. 2013. Idris, a general-purpose dependently typed programming language: Design and implementation. *Journal of Functional Programming* 23, 05 (2013), 552–593.

Pierre-Evariste Dagand. 2013. *A Cosmology of Datatypes*. Ph.D. Dissertation. University of Strathclyde.

Nils Anders Danielsson. 2013. Correct-by-construction Pretty-printing. In *Proceedings of the 2013 ACM SIGPLAN Workshop on Dependently-typed Programming*. ACM, 1–12.

Dominique Devriese and Frank Piessens. 2011. On the Bright Side of Type Classes: Instance Arguments in Agda. In *Proceedings of the 16th ACM SIGPLAN International Conference on Functional Programming (ICFP '11)*. 143–155. DOI: http://dx.doi.org/10.1145/2034773.2034796

Peter Dybjer and Anton Setzer. 1999. A finite axiomatization of inductive-recursive definitions. In *International Conference on Typed Lambda Calculi and Applications*. Springer Berlin Heidelberg, 129–146.

Kathleen Fisher and Robert Gruber. 2005. PADS: A Domain-specific Language for Processing Ad Hoc Data. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '05)*. ACM, New York, NY, USA, 295–304. DOI: http://dx.doi.org/10.1145/1065010.1065046

Pepijn Kokke and Wouter Swierstra. 2015. Auto in Agda: programming proof search. In *International Conference on Mathematics of Program Construction*. Springer, 276–301.

Per Martin-Löf. 1984. *Intuitionistic type theory*. Vol. 9. Bibliopolis Napoli.

The Coq development team. 2004. *The Coq proof assistant reference manual*. LogiCal Project. http://coq.inria.fr Version 8.0.

Conor McBride. 2010. Ornamental algebras, algebraic ornaments. *Journal of functional programming* (2010).

Conor McBride and Ross Paterson. 2008. Applicative programming with effects. *Journal of functional programming* 18, 01 (2008), 1–13.

Peter J McCann and Satish Chandra. 2000. Packet types: abstract specification of network protocol messages. *ACM SIGCOMM Computer Communication Review* 30, 4 (2000), 321–333.

Fabrice Mérillon, Laurent Réveillère, Charles Consel, Renaud Marlet, and Gilles Muller. 2000. Devil: An IDL for hardware programming. In *Proceedings of the 4th conference on Symposium on Operating System Design & Implementation-Volume 4*. USENIX Association, 2–2.

Ulf Norell. 2007. *Towards a practical programming language based on dependent type theory*. Ph.D. Dissertation. Chalmers University of Technology.

Nicolas Oury and Wouter Swierstra. 2008. The power of Pi. In *ACM Sigplan Notices*, Vol. 43. ACM, 39–50.

Tillmann Rendel and Klaus Ostermann. 2010. Invertible syntax descriptions: unifying parsing and pretty printing. In *ACM Sigplan Notices*, Vol. 45. ACM, 1–12.

Paul Van Der Walt and Wouter Swierstra. 2012. Engineering proof by reflection in Agda. In *Symposium on Implementation and Application of Functional Languages*. Springer, 157–173.