Utrecht University
Department of Information and Computing Sciences

A Verified Low-Level Formatting EDSL in Agda

Master's Thesis

Marcell van Geest
ICA-3824985

Supervisors:
dr. Wouter Swierstra
prof. dr. Johan Jeuring

October 31, 2016

Abstract

Data is of little use when it stays inside one program's memory – almost always, information needs to be serialised for storage, or sent to other programs via a low-level interface. Binary storage format and communication protocol definitions strive to establish a precise description of files or messages in order to ensure senders and receivers will agree on their interpretation of their communication. Unfortunately, many standards rely on complex and potentially ambiguous descriptions, some purely textual, some based on C struct definitions, to accomplish this. As a result, there can be multiple subtly different implementations conforming to these specifications, leading to hard-to-find bugs. A more formal approach has the potential to avoid this issue by fixing formats rigidly.

Agda is a dependently typed total functional programming language and a proof assistant. We develop a domain-specific language embedded in Agda that is both precise enough to avoid any confusion and powerful enough to describe real-world formats and protocols. The former is ensured by a pair of encoding and decoding algorithms (a pretty-printer and a parser), accompanied by a proof that they are (half)inverses; the latter is demonstrated by describing the format of IPv4 packets in this EDSL.

Contents

Chapter 1

Introduction

The ever-increasing popularity of web applications, cloud solutions, and thin clients are but a few indicators of the general trend towards software systems that distribute computation across multiple computing units. 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, composed of a programming language's basic data structures, to a *low-level* representation like strings of letters or streams of 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 `struct`s and `union`s to describe the format of messages. This method of description, though consistent with long-standing documentation practices, unfortunately allows for ambiguities and internal inconsistencies.

Many format description languages have been created to alleviate these concerns by providing more formal, computer-processable descriptions of formats. 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.

We set out to design a format description language that is *descriptive* (usable for describing existing protocols), *embedded* in a general-purpose programming language (Agda), and *verified* (equipped with algorithms that provenly keep data consistent during round trips).

In this chapter, we describe various existing languages and their relative strengths and weaknesses, after which we use the next chapter to introduce the surprisingly complex example that led to the creation of this language. We present and explore our design in chapter 3, and verify that it is sufficiently powerful to implement the motivating example in chapter 4. Finally, in chapter 5, we point out its place in the extensive design space of format description languages.

1.1 Conventions

All code in this text is valid or abbreviated Agda code, to be used in conjunction with version 2.5.1.1 of Agda, unless noted otherwise.

The code uses version 0.12 of the Agda Standard Library. The following imports are assumed to be in effect throughout the entire text:

```
open import Algebra
open import Category.Monad
open import Data.Bool
open import Data.Empty
open import Data.Fin as F using (Fin; toℕ)
open import Data.List as L using (List)
open import Data.Maybe
open import Data.Nat as N
import Data.Product as P
open import Data.Unit using (⊤; tt)
open import Data.Vec as V using (Vec)
open import Function
open import Relation.Binary
open import Relation.Binary.PropositionalEquality as PE
  using (_≡_; refl; sym; cong)
open import Universe
```

1.2 Domain-specific languages for formats

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 [14], 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 [8], 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 parametrising 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 analyse and reason about.

Finally, Devil, a DSL and tool package by Mérillon et al. [15], 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.

1.3 An embedded, dependently typed DSL

The previous section explored the field of DSLs for data format descriptions, all of which use their own particular syntaxes and therefore need custom tools for all processing and analysis. An *embedded* DSL, on the other hand, inherits the tools and knowledge available for the language it is embedded in (the *host language*); this is but one of the advantages listed by Hudak [10].

Oury and Swierstra [16, 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.

The following subsections discuss the aforementioned EDSL, closely following the original presentation. Deviations have been marked as such when significant, and we have omitted constructs that are not relevant to our project.

1.3.1 Simple universes

Universes are a fundamental generic programming tool for representing types and thereby enabling the implementation of algorithms that operate on them. Universes are necessary because most programming languages – in particular Haskell and Agda – do not support *directly* pattern-matching on types. There are various obstacles for implementing such a feature in the language itself, such as the possibility that types contain inductive occurrences of themselves or other types, and that the set of types is open (additional types can be declared

at will) while the set of patterns of a function is closed. An (Agda) universe is a type whose values represent a subset of the set of (Agda) types; clearly, it *is* possible to pattern match on such a value. Both Haskell and Agda contain *reflection* features that make it possible to create conversions between a type and its description in some universe.

An extremely simple universe that comes to mind for describing types in a binary setting is that of all n-bit binary words, for any n. As this universe will later be enhanced to describe formats, we will call it "Format Type" (`FT`).

```
data FT₁ : Set where
  word : (n : ℕ) → FT₁
```

A description of a *type* is hardly useful without a facility for end users to create *values* that belong to it. We can implement this either as a function or as a data type. Both these alternatives have disadvantages: the function option seems to require a proof-theoretically stronger host language than the data type option (see section 1.5), while the data type option leads to a data type that is not strictly positive. Choosing the function option in accordance with the literature, we get the following *interpretation function*, which maps a type description from our universe to its corresponding Agda type:

```
⟦_⟧₁ : FT₁ → Set
⟦ word n ⟧₁ = Vec Bool n
```

Note that we can characterise the subset of types that `FT₁` describes as $\{⟦x⟧ \mid x \in FT_1\} \subset Set$.

Considering that any non-trivial binary format will juxtapose several of these words, we need to add some kind of combination mechanism. Arguably the simplest such mechanism – as shown by its prevalence in generic programming libraries in Magalhães and Löh [13] – is binary products. Universes that allow inductive construction of products are called *closed under products*. Adding binary products yields the following universe and interpretation function, where `P` is `Data.Product` from the standard library:

```
data FT₂ : Set where
  word : (n : ℕ) → FT₂
  _×_ : FT₂ → FT₂ → FT₂

⟦_⟧₂ : FT₂ → Set
⟦ word n ⟧₂ = Vec Bool n
⟦ t₁ × t₂ ⟧₂ = ⟦ t₁ ⟧₂ P.× ⟦ t₂ ⟧₂
```

We could now describe a data format with a 32-bit "source" and "destination address" – perhaps IPv4 addresses – as follows:

```
Source+Destination : FT₂
Source+Destination = word 32 × word 32
```

1.3.2 Parsing and pretty-printing

After a data type has been described using a universe description, it should be possible to pretty-print and parse values without further ado; in other words, we expect the following to be available:

```
parse : (t : FT₂) → List Bool → Maybe (⟦ t ⟧₂ P.× List Bool)
pretty-print : (t : FT₂) → ⟦ t ⟧₂ → List Bool
```

Oury and Swierstra describe [16, section 3.5 and 3.4, respectively] both these functions in detail, and their implementations are uninteresting. Note that, for lack of space, no relevant correctness proof is given or hinted at, making their implementations *unverified*. The kind of correctness that is of primary interest here is that parsing is a left inverse of pretty-printing:

```
left-inv : (t : FT₂) → (d : ⟦ t ⟧₂) →
  (P.proj₁ <$> parse t (pretty-print t d)) ≡ just d
```

In words, this represents a round-trip property: if we take a value that belongs to a format, pretty-print it, then parse it, parsing must succeed and yield the original value.

Much of the value of the Agda EDSL to be created would come from a verified implementation, as it is a feature much harder to create – and therefore usually not present – in specialised DSLs and host languages, and it ensures the algorithms can be trusted as much as the proof-theoretical correctness of Agda.

1.3.3 Universes with dependencies

As thoroughly argued in the original paper, the simple universe described in section 1.3.1 is inadequate for describing more interesting formats. A very common example is a format which first specifies a *length*, then expects a piece of data of the given length. Clearly, this does not fit in our fixed-length-words-and-products universe.

Our universe therefore needs to be expanded with a mechanism for allowing the *type* of later members to depend on the *value* of earlier members; one expects this to be simple in a *dependent* host language. Recall the definition of the *dependent product* from `Data.Product`, without universe polymorphism:

```
record Σ (A : Set) (B : A → Set) : Set where
  constructor _,_
  field
    proj₁ : A
    proj₂ : B proj₁
```

Adding `P.Σ` to the previous universe is similar to adding `P._×_`. To achieve this, a small but significant change needs to be made: `FT` will need to be mutually dependent with the interpretation function. It is this dependence (a declaration technicality, not a dependent type) that is forbidden in many host languages, but fortunately allowed in Agda, by separating the type declarations from the implementations as follows:

```
data FT₃ : Set
⟦_⟧₃ : FT₃ → Set

data FT₃ where
  word : (n : ℕ) → FT₃
  _×_ : FT₃ → FT₃ → FT₃
  Σ : (x : FT₃) → (⟦ x ⟧₃ → FT₃) → FT₃
```

```
⟦ word n ⟧₃ = Vec Bool n
⟦ t₁ × t₂ ⟧₃ = ⟦ t₁ ⟧₃ P.× ⟦ t₂ ⟧₃
⟦ Σ t f ⟧₃ = P.Σ ⟦ t ⟧₃ (λ v → ⟦ f v ⟧₃)
```

This dependent universe is capable of representing the ubiquitous "length then data" structure. The following represents a 16-bit length field with a corresponding data field:

```
Length+Data : FT₃
Length+Data = Σ (word 16) (λ len → word (toℕ (decode len)))
```

Note that the user is required to be explicit about how to *encode* the length. A possible (and common) encoding to be used by the preceding example is binary encoding, implemented by a pair of functions `encode : {l : ℕ} → Fin (2^ l) → Vec Bool l` and its inverse `decode`. The appearance of the `Fin` data type indicates that there is an upper bound to the data's length. Indeed, as the length field *itself* must have a fixed length, there will be such an upper bound for any given encoding scheme, although of course the field length and encoding scheme can be chosen so as to make that arbitrarily large.

Note also that Oury and Swierstra [16, section 3.1] choose to add one level of abstraction by using a more flexible `base` constructor instead of `word`. This `base` constructor represents more than just the type of n-bit vectors: its argument is a value from *another* universe, the "value universe". This universe describes common and easily processable types such as `Bool`, `Char`, and `Fin n`. Parsing and pretty-printing algorithms for values of these types are defined as two separate functions, which are then called by `parse` and `pretty-print`. Defining the universe using the `base` constructor would allow a more simple definition of `Length+Data`, at the expense of complicating the explanation of the universe.

1.3.4 Adding derived and uninteresting data

Another common part of both storage and communication formats is *derived* information: fields whose values are to be computed from other fields, such as checksums. Constant data such as separators and version numbers can be "computed" from the empty set of fields. Naturally, we would like to remove the burden of this computation from the end user, which also prevents any mistakes or omissions, making the correctness of parsing and pretty-printing depend only on the correctness of the format description.

Hence, what we need is a constructor for which the interpretation function *does not require data.* Its pretty-printing semantics must allow the user to specify how its value is to be *derived* from other data, while its parsing semantics may simply be *skipping* the parsed value (parsing it is required to locate the start of the next value), or *verifying* that the value is equal to the expected value. Without further ado, we introduce the `calc` constructor.

```
data FT₄ : Set
⟦_⟧₄ : FT₄ → Set

data FT₄ where
  word : (n : ℕ) → FT₄
  calc : (x₁ : FT₄) → ⟦ x₁ ⟧₄ → FT₄
  _×_ : FT₄ → FT₄ → FT₄
```

```
  Σ : (x : FT₄) → (⟦ x ⟧₄ → FT₄) → FT₄

⟦ word n ⟧₄ = Vec Bool n
⟦ calc x₁ v ⟧₄ = ⊤
⟦ t₁ × t₂ ⟧₄ = ⟦ t₁ ⟧₄ P.× ⟦ t₂ ⟧₄
⟦ Σ t f ⟧₄ = P.Σ ⟦ t ⟧₄ (λ v → ⟦ f v ⟧₄)
```

For this new constructor, `parse` will attempt to read (then discard) a value of type `⟦ x₁ ⟧₄`, while `pretty-print` will output the constructor's second argument, a value of that type. Note that this is a *constant* value, so it might not be immediately obvious how `calc` allows the addition of *derived* data; it only does this when used in conjunction with the dependent pair. The following example demonstrates this, using a function `parity : (n : ℕ) → Vec Bool n → Vec Bool 1` to calculate a byte's parity.

```
Data+Checksum : FT₄
Data+Checksum = Σ (word 8)
  (λ d₁ → calc (word 1) (parity 8 d₁))
```

1.4 A verified, separate-direction approach

Danielsson [7] describes an approach for creating string-based parsers and pretty-printers in Agda with the property mentioned in section 1.3.2 by requiring users to write a separate parser and pretty-printer. The pretty-printer is based on the standard Hughes approach [11] and is parametrised by the parser, which takes the shape of a grammar, to enforce the aforementioned round-trip property. The classical pretty-printing combinators are provided for which the proof of the property is included, along with a combinator that accepts a "manual" proof for extensibility. The downside that the pretty-printer and grammar must be written separately is offset by the built-in ability to format documents in multiple styles.

1.5 Tools

Agda [3] and Coq [6] are two popular dependently-typed programming languages, both more or less suitable for data formatting-related development. Coq, being older, offers better extraction (compilation) facilities than Agda, but the same property makes it significantly less comfortable to work with: it lacks convenient mechanisms for a necessary generic programming tool.

The mutual dependence between the universe data type and the interpretation function in section 1.3 is known as *induction-recursion*; had we chosen to implement an interpretation data type instead, we would have had *induction-induction*. Agda allows inductive-inductive and inductive-recursive definitions, while Coq does not. Although Hancock et al. [9] describe a way to convert *small* inductive-recursive definitions into indexed data types, which are acceptable, it is uncertain whether our constructs belong to this category. At the same time, Altenkirch conjectures [1] that induction-induction is proof-theoretically no stronger than plain induction, which makes inductive-inductive definitions preferable to inductive-recursive variants.

Interestingly, Idris [5], another dependently typed language, has been used by Brady [4] to combine the ideas of the non-embedded DSL PacketTypes with dependent types. The resulting language has approximately the same expressive power as the EDSL described in section 1.3.

Chapter 2

Motivation and Goals

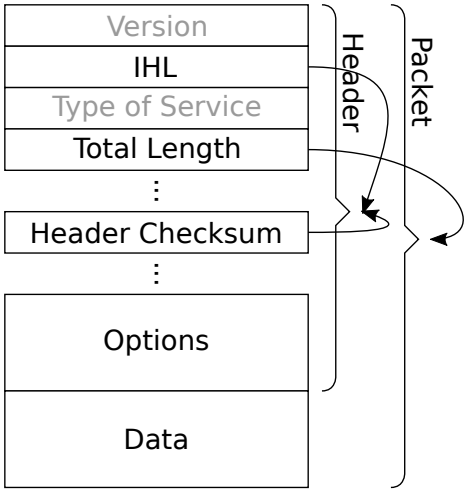
The previous chapter explored some of the dimensions in the design space of formatting languages along with a number of existing systems. This chapter discusses the desired features by introducing the motivating example, as well as the properties our design should satisfy.

2.1 Desired features

An everyday real-life example of a data format that can not be described within the EDSL set up in section 1.3 is IPv4 packets. (We will refer to that existing work as "the EDSL" in this section only.) To justify the claim that communication protocols contain hard-to-process descriptions, a copy of the representation of the IPv4 header [17, section 3.1], updated with changes from later RFCs, follows. In this table, [E] represents the ECN field.

<table>
<tr><td colspan="1">0</td><td colspan="1">1</td><td colspan="1">2</td><td colspan="1">3</td><td colspan="1">4</td><td colspan="1">5</td><td colspan="1">6</td><td colspan="1">7</td><td colspan="1">8</td><td colspan="1">9</td><td colspan="1">0</td><td colspan="1">1</td><td colspan="1">2</td><td colspan="1">3</td><td colspan="1">4</td><td colspan="1">5</td><td colspan="1">6</td><td colspan="1">0</td><td colspan="1">8</td><td colspan="1">9</td><td colspan="1">0</td><td colspan="1">1</td><td colspan="1">2</td><td colspan="1">3</td><td colspan="1">4</td><td colspan="1">5</td><td colspan="1">6</td><td colspan="1">7</td><td colspan="1">8</td><td colspan="1">9</td><td colspan="1">0</td><td colspan="1">1</td></tr>
<tr><td colspan="4">Version</td><td colspan="4">IHL</td><td colspan="6">DSCP</td><td colspan="2">[E]</td><td colspan="16">Total Length</td></tr>
<tr><td colspan="16">Identification</td><td colspan="4">Flags</td><td colspan="12">Fragment Offset</td></tr>
<tr><td colspan="8">Time to Live</td><td colspan="8">Protocol</td><td colspan="16">Header Checksum</td></tr>
<tr><td colspan="32">Source Address</td></tr>
<tr><td colspan="32">Destination Address</td></tr>
<tr><td colspan="24">Options</td><td colspan="8">Padding</td></tr>
</table>

To illustrate the deficiencies of the EDSL seen so far, the following diagram describes the dependencies between the various fields, i.e. which information is to be derived from what. The uninteresting fields have been greyed out or ellipsised, and the Data field was added to represent the packet's data, which immediately follows the header.



This diagram reveals three main aspects for which the existing EDSL lacks descriptive power.

Problem 1: Full-value calculations
The Header Checksum depends on the *entire* header.

This problem is highly conspicuous and unsolvable without modifications to the EDSL (that is, all workarounds hurt usability). In general, dependent type systems allows types to depend on values declared *earlier*, not later. As a consequence, the calculation constructor `calc` (section 1.3.4) only allows a calculation to refer *backward*, not forward, and it does not allow the result of a calculation to be placed in the middle of the input it requires. Note that the checksum calculation algorithm does not "require its own output to calculate itself" or approximate the output iteratively; it simply considers the output field to be filled with zeroes for the purpose of checksum calculation.

Problem 2: High-level and user-defined types
There is a mismatch between the types required by the EDSL (low-level types such as vectors of booleans) and types that are convenient for specifying second components (high-level types such as bounded natural numbers).

This problem arises when trying to describe the variable-length fields, Options and Data. The natural (and only) way to assign a variable length involves a dependent product (as demonstrated by the `Length+Data` example in section 1.3.3). That entails that the two length fields, Internet Header Length (IHL) and Total Length, need to be placed in the first component of a dependent product; the Options and Data fields must belong to the second component. The length fields will have types `word 4` and `word 16`, and therefore the second component will be a function that receives values of those two types in its argument. This is inconvenient for both the end user and the protocol description developer: the end user needs to manually *encode* the numbers into boolean vectors, while the protocol description developer needs to *decode* those numbers again to pass into the `word` constructor. More specifically, the second component will have to look like `(λ c → word (toℕ (decode (getField ... c))) × ...)`. Not only is this a usability issue, it also influences correctness: when the encoding of the fields in the first component changes (perhaps because of endianness), the decoding in the second component must also change – and

forgetting to apply such a change twice has actually led to a bug during this project.

Furthermore, *user-defined* types, such as data types representing enumerations, cannot be used in the description at all, which is another usability disadvantage. The original presentation of the EDSL did include a more flexible `base` constructor and corresponding value universe to allow more than just boolean vectors, but such a value universe is *closed*, so it can not contain types defined by format designers.

Problem 3: Awkward concepts
There is a mismatch between the data stored in the Internet Header Length (IHL) and Total Length fields, and the data available to or of interest to end users (in the pretty-printing and parsing case, respectively).

This problem is the trickiest to spot, but no less annoying. Even if we manage to solve the mismatch of *type* levels just explained, another problem remains: the *concepts* in the first component are not appropriate for the end users. A user interested in pretty-printing arrives knowing the length of the options that need to be pretty-printed. This user would then have to calculate, for example, the value of the Internet Header Length manually (this is the constant 5, representing the length of the mandatory header fields, plus the length of the options field *in 32-bit words*); this would then force the Options field to have the type `word (32 * (IHL - 5))`, which will require `subst` and properties of natural numbers to work with.

The general issue is that the protocol-mandated Internet Header Length and Total Length are not of any importance to the end user – the length of the Options and the length of the Data are what they would like to know (during parsing) or have immediate access to (during pretty-printing). In other words, we would like to explicitly name the length of the options `OL`, let the Options field have the easy-to-process type `word (32 * OL)`, and set the IHL field to the result of the calculation `OL + 5`. More generally, we need to allow parsing and pretty-printing a *transformed* variant of data, which is of course acceptable if the transformation is bijective.

In short, our first goal is thus to incorporate into the EDSL (1) calculations that can range over all data; (2) flexible high-level types; and (3) the ability to translate between user-friendly and protocol-mandated concepts. A description of the IPv4 packet format will serve as a demonstration of these extensions.

2.2 Desired properties

In an attempt to formalise the mechanisms of parsing and pretty-printing, Rendel and Ostermann [18] introduce the concept of *partial isomorphisms* (PIs). They represent a partial isomorphism as a Haskell data type with type arguments α and β defined as a pair of functions $f :: \alpha \to \mathit{Maybe}\ \beta$ and $g :: \beta \to \mathit{Maybe}\ \alpha$, with the invariant that "if $f\ a$ returns $\mathit{Just}\ b$, $g\ b$ returns $\mathit{Just}\ a$, and the other way around". In their section on future work, they say it "would be interesting to see how the invariants of *Iso* values could be encoded in a dependently typed language"; this encoding is actually rather straightforward:

```
record PI₁ (A B : Set) : Set where
  field
    ⇨ : A → Maybe B
    ⇦ : B → Maybe A
    left-inv : (a : A) → (b : B) → ⇨ a ≡ just b → ⇦ b ≡ just a
    right-inv : (b : B) → (a : A) → ⇦ b ≡ just a → ⇨ a ≡ just b
```

In this encoding, ⇨ represents f and ⇦ represents g.
The authors do, however, state a caveat:

> We will generally not be very strict [precise] with the invariant stated above (if $f\ a$ returns *Just* b, $g\ b$ returns *Just* a, and the other way around). In particular we will sometimes interpret this condition modulo equivalence classes [*sic*].

Considering it is Agda's task (and main virtue) to be very precise, we will need to be very explicit about any imprecision we want to introduce. Fortunately, what we want to express has been formalised as `IsEquivalence` and `Setoid` in the standard library, and we can define a relaxed variant of partial isomorphisms as follows.

```
record PI₂ (A B : Set) (_≈_ : Rel B _)
  (isEquivalence : IsEquivalence _≈_) : Set where

  SB : Setoid _ _
  SB = record { Carrier = B ; _≈_ = _≈_ ;
    isEquivalence = isEquivalence }

  _≋_ : Rel (Maybe B) _
  _≋_ = Setoid._≈_ (Data.Maybe.setoid SB)

  field
    ⇨ : A → Maybe B
    ⇦ : B → Maybe A
    left-inv : (a : A) → (b : B) → ⇨ a ≡ just b → ⇦ b ≡ just a
    right-inv : (b : B) → (a : A) → ⇦ b ≡ just a → ⇨ a ≋ just b
```

Note that the relaxation has only been applied to `B`-equality. Not only do the authors only relax the `B`-side, we can justify it more explicitly by pointing out that we're interested in formatting from a *high* (`A`) level to a *low* (`B`) level; we want to allow multiple low-level representations that parse to the same high-level value, while we are not interested in generating multiple low-level representations.

We modify the concept of partial isomorphisms slightly to get *semipartial isomorphisms* (SPIs), the concept of which is most clearly described by the following record:

```
record SPI (A B : Set) : Set where
  field
    ⇩ : A → B
    ⇧ : B → Maybe A
    left-inv : (x : A) → ⇧ (⇩ x) ≡ just x
```

This seems to be a weaker concept, considering that it only contains one property; in fact, it is stronger, as we can implement a conversion from an SPI to a relaxed PI:

```
SPI→PI₂ : (A B : Set) → SPI A B → P.∃₂ (PI₂ A B)
```

This can be a "faithful" conversion, in that we can use `just ∘′ ⇩` as `⇨` and `⇧` as `⇦` (so that the SPI and the PI effectively represent the same transformation). To prove the PI properties, we can choose as the equivalence relation on `B` the equality induced by `⇧` (i.e. two low-level values are equal if and only if they parse to the same high-level value). This "equality induced by a function" can be found in the standard library as `Relation.Binary.On.setoid`, and the proof can be completed in a dozen lines.

Both partial and semipartial isomorphisms form monoids with the identity isomorphism (that always succeeds and leaves the type unchanged) as the identity element and straightforward composition of each field as the operation. This structure ensures that sequential composition is always possible and preserves the coveted round-trip property. Therefore, we our second goal will be to ensure that all parts of our system are semipartial isomorphisms.

Chapter 3

Design

In this chapter, we will apply the techniques from chapter 1 to the problems in chapter 2, gradually developing a suitable solution. The final design is based on the dependently typed format descriptions from section 1.3, but separates the concept of *juxtaposition* and the concept of *extension*.

Some data types and functions will be iteratively adjusted throughout several sections. In these cases, we use identifiers `P₁`, `P₂`, …, `Pₙ` in the Agda code, and any reference to `P` in the text is intended as pointing to the latest `Pₙ`.

3.1 Universe

We set up the following universe:

```
mutual
  infixr 2 _×_
  data DT : Set₁ where
    leaf : Set → DT
    _×_ : DT → DT → DT
    Σ : (c : DT) → (⟦ c ⟧ → DT) → DT

  ⟦_⟧ : DT → Set
  ⟦ leaf A ⟧ = A
  ⟦ l × r ⟧ = ⟦ l ⟧ P.× ⟦ r ⟧
  ⟦ Σ c d ⟧ = P.Σ ⟦ c ⟧ (λ x → ⟦ d x ⟧)
```

We will often view values of `DT` (also known as *type codes*) as "type trees", in which each `leaf` is a leaf that holds a type and the other two constructors are internal nodes that hold only structural information: just the order (`_×_`) or the value-to-type relation (`Σ`) of subtrees. To avoid confusion, we will use the names *left* and *right* for the first and second components of simple products (`_×_`), and the names *constant* and *dependent* for the first and second components of dependent products (`Σ`). The letters `l`, `r`, `c`, and `d` will be used as suffixes for identifiers (of various types) related to left, right, constant, and dependent components, respectively.

Only to justify calling this a "universe", we can show this is a universe as defined by the standard library:

```
DT-universe : Universe _ _
DT-universe = record { U = DT ; El = ⟦_⟧ }
```

We can also restate the `Length+Data` example from section 1.3.3:

```
SimpleLength+Data : DT
SimpleLength+Data = Σ
  (leaf (Vec Bool 16))
  (λ len → leaf (Vec Bool (toℕ (decode len))))
```

Even this simple example displays the ugliness caused by the lack of high-level types (problem 2 on page 11): the first (constant) component contains a binary representation of the length, which needs to be `decode`d for use in the second (dependent) component. Because `DT` allows `leaf` constructors to contain arbitrary types, we can instead use "clean", more principled data types, leading to a high-level description. In this case, `Fin` would be the most natural choice for the first (constant) component:

```
ProperLength+Data : DT
ProperLength+Data = Σ
  (leaf (Fin (2^ 16)))
  (λ len → leaf (Vec Bool (toℕ len)))
```

Considering the task at hand, `DT` is an exceedingly simple universe, bringing only two – or, as explained below, one – operations to the table. Its leaves, on the other hand, contain arbitrary types and therefore encompass all type-based trickery Agda allows. The advantage of splitting out these concepts into a separate data type will be shown in section 3.5. Note, however, that the features offered by `DT` are more or less equivalent to those offered by Agda's `record`s: the universe offers ordered, (possibly) dependent juxtaposition of fields.

We will now discuss a few conspicuous design choices which require further justification.

`DT` uses induction-recursion

It is clear that, just like the universe described in section 1.3.3, `DT` is inductive-recursive because of the `Σ` constructor. As discussed in section 1.5, support for induction-recursion is present in Agda and Idris, but not in Coq, limiting the portability of this solution. It seems possible to write this universe using induction-induction by turning `⟦_⟧` into a data type:

```
mutual
  {-# NO_POSITIVITY_CHECK #-}
  data DT′ : Set₁ where
    leaf : Set → DT′
    _×_ : DT′ → DT′ → DT′
    Σ : (c : DT′) → (DD′ c → DT′) → DT′

  data DD′ : DT′ → Set where
    leaf : {A : Set} → A → DD′ (leaf A)
    _,′_ : {l r : DT′} → DD′ l → DD′ r → DD′ (l × r)
    _,_ : {c : DT′} {d : DD′ c → DT′} →
      (x : DD′ c) → DD′ (d x) → DD′ (Σ c d)
```

The downside is that `DT′` is not strictly positive, as evidenced by Agda's (valid) complaint if the pragma is omitted:

```
DT′ is not strictly positive, because it occurs
in the type of the constructor _,′_
in the definition of DD′, which occurs
to the left of an arrow
in the type of the constructor Σ
in the definition of DT′.
```

Much of the development described in this text was, uncatastrophically, carried out using the `DT′` definition, which indicates that this *might* be a "reasonable" definition despite it not being strictly positive, just like traditional `quicksort` is a "reasonable" (terminating) definition despite the fact that it calls itself on a value that is not strictly structurally smaller than the corresponding argument. In practice, `DT′` has the distinct advantage that it allows a developer to pattern-match directly on the value `d` in a function of type `{t : DT′} → (d : DD′) → ...`, which – along with not having to name the implicit argument – makes quite a few definitions easier to read; on the other hand, this is a small gain compared to the devastating effects of a "bad" data type can have, as demonstrated by the canonical `Omega` example.

`DT` is rather large (it does not fit in `Set`)

Allowing arbitrary types in `leaf` enhances usability: both format designers and users can use domain-specific plain Agda datatypes directly. It also has not presented any practical difficulty throughout the project: neither computations nor proofs seem to become more complex because of the largeness. If required, the universe can be stratified by adding a separate value universe `V : Set` and interpretation function `⟪_⟫ : V → Set` and parametrising `DT` by it, which allows `DT` to have type `Set`.

`DT` is rather small (`Set` does not fit in it)

If a user would like to format actual *types* with this system, she's out of luck: the type `⟦ leaf A ⟧` is `A`, which is of type `Set`, and therefore a value of that type cannot be a type itself. If the goal is indeed to format types, the stratification mentioned above is a suitable solution, as it allows pattern-matching (if `V` allows it), while one cannot pattern-match on types. It would also be possible to parametrise, or even index, `DT` by a level, at the cost of having to add universe polymorphism to most functions working with this system.

`_×_` is superfluous

Just like `P._×_` is defined in terms of `P.Σ`, `_×_` could very easily be implemented in terms of `Σ`:

```
_×′_ : DT → DT → DT
l ×′ r = Σ l (λ _ → r)
```

The advantage of the separate constructor `_×_` lies in the lack of dependencies between its arguments' types. This makes it much easier to compute with, and

especially reason about, than `Σ`. Understandably, `with`, `rewrite` (which is no longer implemented in terms of `with`) and instance search have more and more trouble generating correct functions as the complexity of dependencies increases. Therefore, the inability to make a function work (or prove a property) for `_×_` is a much stronger indicator of the unimplementability of the function (or the falsity of the property) than the failure to do the same for `Σ`.

3.2 Parsing and pretty-printing

A downside of allowing the `leaf` constructor to contain any data type is that there are many types that cannot be sensibly pretty-printed and parsed (e.g. function types). We therefore need to define a concept of *low-level* `DT`s that allow pretty-printing and parsing. Furthermore, we would like to define low-levelness in a way that makes the pretty-printing and parsing algorithms as simple as possible (with minimal specialised code). The simplest useful definition of low-levelness is that of being composed only of fixed-length vectors of bits, represented as boolean values. The following data type encodes this idea:

```
data IsLowLevel : DT → Set where
  instance leaf : {n : ℕ} → IsLowLevel (leaf (Vec Bool n))
  instance _×_ : {l r : DT} →
    IsLowLevel l → IsLowLevel r → IsLowLevel (l × r)
  instance Σ : {c : DT} {d : ⟦ c ⟧ → DT} →
    IsLowLevel c → ((x : ⟦ c ⟧) → IsLowLevel (d x)) →
    IsLowLevel (Σ c d)
```

Not having to "state the obvious" is crucial for usability, both for format designers and end users. The Agda 2.5 instance search mechanism is, in practice, usually powerful enough to construct instances of `IsLowLevel` even for the `Σ` case, despite the fact that that requires constructing a function. A value of type `IsLowLevel SimpleLength+Data`, for the example given in the previous section, can be found automatically. An instance search can be started for any value using the `it`-pattern, which is surprisingly not in the standard library:

```
it : ∀ {a} {A : Set a} → {{x : A}} → A
it {{x}} = x
```

When `d` is a function that pattern-matches on its argument, instance search refuses to try each case, making it necessary to build the instance manually. In those cases, it *is* usually possible to give an instance for the troublesome cases and let instances for its subtrees and supertrees be found automatically, as demonstrated in the following example.

```
IntricateD : (x : ⟦ leaf (Vec Bool 1) ⟧) → DT
IntricateD (false V.∷ V.[]) = leaf (Vec Bool 2)
IntricateD (true V.∷ V.[]) = leaf (Vec Bool 5)

Intricate : DT
Intricate = leaf (Vec Bool 1) ×
  Σ (leaf (Vec Bool 1)) IntricateD

IntricateILL : IsLowLevel Intricate
```

```
IntricateILL = it where
  IntricateDILL : (x : ⟦ leaf (Vec Bool 1) ⟧) →
    IsLowLevel (IntricateD x)
  IntricateDILL (false V.∷ V.[]) = it
  IntricateDILL (true V.∷ V.[]) = it
  instance
    IntricateΣILL : IsLowLevel
      (Σ (leaf (Vec Bool 1)) IntricateD)
    IntricateΣILL = Σ it IntricateDILL
```

Note that the instance `IntricateΣILL` cannot currently be generated even if `IntricateDILL` is marked as an instance, but given `IntricateΣILL`, the generation of `IntricateILL` is no problem.

Finally, some nice-to-have conversion functions depend on the presence of `leaf ⊥` in some trees. Since *ex falso sequitur quodlibet*, we can validly claim that such leaves are low-level. We add the following constructor, which is a little more general:

```
from-⊥ : {t : DT} → (⟦ t ⟧ → ⊥) → IsLowLevel t
```

The instance search will not attempt to construct the required function, which means making it try this constructor is a waste of time, so we do not declare it as an instance.

Given an instance of `IsLowLevel t`, pretty-printing is easy, that is, we can turn a value of type `⟦ t ⟧` into a list of bits:

```
toList : {t : DT} → IsLowLevel t → ⟦ t ⟧ → List Bool
toList leaf d = V.toList d
toList (illl × illr) (dl P., dr) =
  toList illl dl L.++ toList illr dr
toList (Σ illc illd) (dc P., dd) =
  toList illc dc L.++ toList (illd dc) dd
toList (from-⊥ f) d = ⊥-elim (f d)
```

Parsing, the opposite operation, is a little more complex. We first need a helper function that splits a list into the leading `n` elements and the rest (if possible) and a property guaranteeing that behaviour:

```
splitAt′ : ∀ {a} {A : Set a} →
  (n : ℕ) → List A → Maybe (Vec A n P.× List A)
splitAt′-all : ∀ {a} {A : Set a} →
  (n : ℕ) → (xs : Vec A n) → (rest : List A) →
  splitAt′ n (V.toList xs L.++ rest) ≡ just (xs P., rest)
```

Now we can define `fromList`:

```
fromList : {t : DT} → IsLowLevel t →
  List Bool → Maybe (⟦ t ⟧ P.× List Bool)
fromList (leaf {n}) xs = splitAt′ n xs
fromList (illl × illr) xs with fromList illl xs
fromList (illl × illr) xs | nothing = nothing
fromList (illl × illr) xs | just (rl P., rxs₁)
  with fromList illr rxs₁
... | nothing = nothing
```

```
... | just (rr P., rxs₂) = just ((rl P., rr) P., rxs₂)
fromList (Σ illc illd) xs with fromList illc xs
fromList (Σ illc illd) xs | nothing = nothing
fromList (Σ illc illd) xs | just (rc P., rxs₁)
  with fromList (illd rc) rxs₁
... | nothing = nothing
... | just (rd P., rxs₂) = just ((rc P., rd) P., rxs₂)
fromList (from-⊥ f) xs = nothing
```

The standard, and much cleaner, approach to writing parsing functions in functional languages is to use a parsing monad, as described by Hutton and Meijer [12]. In this case, too, it seems appealing to use the monad `List Bool → Maybe (A P.× List Bool)`, the combination of the state monad on boolean lists and the `Maybe` monad. This approach works well for the `_×_` case, but unfortunately, it is too weak for the `Σ` case: the `Maybe` monad's `_>>=_` (bind) has the *non-dependent* type `{A B : Set} → Maybe A → (A → Maybe B) → Maybe B` (minus universe polymorphism), while the line `with fromList (illd rc) rxs₁` is effectively a *dependent* function.

The much-desired correctness property is now almost disappointingly easy to prove. We must be careful, however, to find the correct statement. The `IsLowLevel`-supplemented restatement of `left-inv` from section 1.3.2 looks sensible:

```
left-inv′ : {t : DT} → (ill : IsLowLevel t) → (d : ⟦ t ⟧) →
  (P.proj₁ <$> fromList ill (toList ill d)) ≡ just d
```

It is, however, too weak to use in a recursive position, as it leaves the contents of the second component of a successful call to `fromList` unspecified: `fromList` must return the remaining elements of the list there, without loss, insertion, or alteration. The appropriate approach is the following:

```
left-inv : {t : DT} → (ill : IsLowLevel t) →
  (rest : List Bool) → (d : ⟦ t ⟧) →
  fromList ill (toList ill d L.++ rest) ≡ just (d P., rest)
```

The simpler lemma can be recovered easily.

3.3 Extension

We can now manipulate the values of low-level type descriptions, but we lack the tools to work with high-level descriptions, which we found to be crucial for usability and robustness (problem 2 on page 11). We introduce the concept of *extension*, a description of a transformation between (the values of) two data types. By ensuring this is a semipartial isomorphism, we can – using sequential composition – prepend such transformations to the semipartial isomorphism created in the previous chapter. In other words, we can start with a high-level type, apply an extension to bring it down to a low-level type, then apply the previous chapter's tools to transform to lists of booleans.

The simplest possible kind of extension is that in which the recursive constructors simply recurse, and each leaf is converted using its own small semipartial isomorphism. This effectively encodes treewise compositionality of semipartial isomorphisms.

```
data DTX₁ : DT → Set₁ where
  convertLeaf : {A : Set} → (B : Set) →
    (⇩ : A → B) → (⇧ : B → Maybe A) →
    ((x : A) → (⇧ (⇩ x) ≡ just x)) →
    DTX₁ (leaf A)
  _×_ : {l r : DT} → DTX₁ l → DTX₁ r → DTX₁ (l × r)
  Σ : {c : DT} {d : ⟦ c ⟧ → DT} →
    DTX₁ c → ((x : ⟦ c ⟧) → DTX₁ (d x)) → DTX₁ (Σ c d)
```

Instead of just converting leaves, we can also convert entire subtrees; we can recover `convertLeaf` in the obvious way.

```
data DTX₂ : DT → Set₁ where
  convert : {t₁ : DT} → (t₂ : DT) →
    (⇩ : ⟦ t₁ ⟧ → ⟦ t₂ ⟧) → (⇧ : ⟦ t₂ ⟧ → Maybe ⟦ t₁ ⟧) →
    ((x : ⟦ t₁ ⟧) → (⇧ (⇩ x) ≡ just x)) →
    DTX₂ t₁
  _×_ : {l r : DT} → DTX₂ l → DTX₂ r → DTX₂ (l × r)
  Σ : {c : DT} {d : ⟦ c ⟧ → DT} →
    DTX₂ c → ((x : ⟦ c ⟧) → DTX₂ (d x)) → DTX₂ (Σ c d)

convertLeaf₂ : {A : Set} → (B : Set) →
    (⇩ : A → B) → (⇧ : B → Maybe A) →
    ((x : A) → (⇧ (⇩ x) ≡ just x)) →
    DTX₂ (leaf A)
convertLeaf₂ B ⇩ ⇧ p = convert (leaf B) ⇩ ⇧ p
```

Given these data types, we expect to be able to define the following functions, where `extendValue` is to play the role of ⇩ in the semipartial isomorphism.

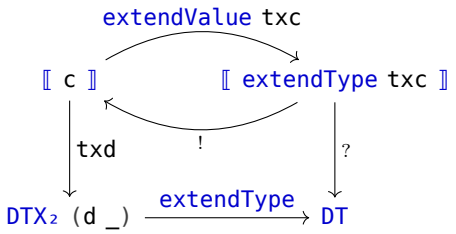
```
extendType₂ : {t : DT} → DTX₂ t → DT
extendValue₂ : {t : DT} → (tx : DTX₂ t) →
  ⟦ t ⟧ → ⟦ extendType₂ tx ⟧
```

Although all definitions so far in this section are arguably the most natural ones possible, they lead to serious difficulties when trying to define these (equally sensible) functions:

```
extendType₂? (convert t₂ ⇩ ⇧ p) = t₂
extendType₂? (txl × txr) = extendType₂? txl × extendType₂? txr
extendType₂? {Σ c d} (Σ txc txd) = Σ (extendType₂? txc) {!!}
```

To explain the difficulty of the hole for the second (dependent) component, we present its goal and context textually as well as graphically:

```
Goal: ⟦ extendType txc ⟧ → DT
————————————————————————————————
txd : (x : ⟦ c ⟧) → DTX₂ (d x)
txc : DTX₂ c
d   : ⟦ c ⟧ → DT
c   : DT
```



In the diagram, the function required by the hole is the arrow labelled ?. The natural way to construct it would be to compose `extendType`, `txd` and the arrow labelled ! – but we do not have a function implementing !. In plain English, we can get the extended second component type if we have the *original* first component value, but all we have is the *extended* first component value. To resolve this, we need to define the inverse operation, retraction; fortunately, we already needed that for ⇑.

```
retractValue₂ : {t : DT} → (tx : DTX₂ t) →
  ⟦ extendType₂ tx ⟧ → Maybe ⟦ t ⟧
```

Note that this makes `extendType` and `retractValue` mutually dependent, which is somewhat surprising. Since `retractValue` returns a `Maybe` (retraction might fail), the definition of `extendType` for the second component needs to use `maybe′`, the non-dependent `Maybe` eliminator, and the correct definition becomes:

```
extendType₂Dep = (λ x⇒ → maybe′
  (λ x⇐ → extendType₂ (txd x⇐))
  (leaf ⊥)
  (retractValue₂ txc x⇒))
```

The definitions of `retractValue` and `extendValue` are uninteresting, except for the `Σ` case, which leaves us with a goal of type `⟦ maybe′ (λ x⇐ → extendType₂ (txd x⇐)) (leaf ⊥) (retractValue₂ txc (extendValue₂ txc dc)) ⟧`. The solution to this is to simultaneously give the proof of the round-trip property (`left-inv`) as it applies to extension and retraction:

```
retractExtendId₂ : {t : DT} → (tx : DTX₂ t) → (d : ⟦ t ⟧) →
  retractValue₂ tx (extendValue₂ tx d) ≡ just d
```

Rewriting with this proof allows us to give the value for the second component.

Now we have set up the extension machinery, we have a method of converting high-level types into low(er)-level ones, eventually arriving at a type that `IsLowLevel` and whose values can therefore be pretty-printed and parsed by `toList` and `fromList`. Recall the `ProperLength+Data` example from section 3.1:

```
ProperLength+Data : DT
ProperLength+Data = Σ
  (leaf (Fin (2^ 16)))
  (λ len → leaf (Vec Bool (toℕ len)))
```

The problematic part is the first component, which is not low-level, but this is easily fixed by applying standard binary encoding. The following extension results in a fully low-level type. (The `copy` extension is that which leaves a type subtree unchanged, i.e. the identity semipartial isomorphism.)

```
ProperLength+DataExt : DTX₂ ProperLength+Data
ProperLength+DataExt = Σ
  (convert
    (leaf (Vec Bool 16))
    encode
```

```
      (just ∘′ decode)
      (λ x → cong just (decode∘encode x)))
    (λ len → copy₂)
```

Finally, note that in the process of adding tools for high-level types (problem 2), we have also created a solution for the problem of awkward concepts (problem 3): we can use an extension not only for "boring" conversions between values of high- and low-level types, but also for complex conversions between values of entire subtrees containing user-friendly and protocol-mandated values. It must be noted that extension is *powerful* enough to describe such conversions, but its quality – mainly in terms of *usability* – might not suffice; more on this in the case study (chapter 4).

3.4 Repeated extension

Often it is convenient to chain multiple extensions to allow a gradual conversion to a fully low-level data type. For example, a checksum is almost always calculated from the binary representation of a set of fields, which implies the most natural implementation would be one extension that translates all fields to their binary representations, followed by another that adds the checksum. Based on sequential composition of semipartial isomorphisms, we can define the concept of repeated extension, which allows a user to extend zero or more times.

```
data DTX₂* : DT → DT → Set₁ where
  base : {t : DT} → DTX₂* t t
  step : {t₁ t₂ : DT} →
    DTX₂* t₁ t₂ → (tx : DTX₂ t₂) → DTX₂* t₁ (extendType₂ tx)
```

The first and second `DT` arguments of `DTX*` represent the source and target type, respectively. `DTX*` is in fact the reflexive transitive closure of `DTX` viewed as a binary relation on `DT`.

The following easy-to-implement functions accompany this definition. Note that `extendType*` is only included for reasons of symmetry, because its task is only to return the target type, which is just `t₂`.

```
extendType₂* : {t₁ t₂ : DT} → (txs : DTX₂* t₁ t₂) → DT
extendValue₂* : {t₁ t₂ : DT} → (txs : DTX₂* t₁ t₂) →
  ⟦ t₁ ⟧ → ⟦ t₂ ⟧
retractValue₂* : {t₁ t₂ : DT} → (txs : DTX₂* t₁ t₂) →
  ⟦ t₂ ⟧ → Maybe ⟦ t₁ ⟧
```

The round-trip property, defined as follows, is also easy to prove:

```
retractExtendId₂* : {t₁ t₂ : DT} →
  (txs : DTX₂* t₁ t₂) → (d : ⟦ t₁ ⟧) →
  retractValue₂* txs (extendValue₂* txs d) ≡ just d
```

3.5 Extension with insertion

To solve the problem of calculated additional data (problem 1 on page 11), we introduce another constructor in the extension datatype. The idea is for this

constructor to contain a function that calculates data from *the entire, top-level value*, so it always has access to all data independent of its position (unlike the `calc` constructor of section 1.3.4). To do this, we need to parametrise `DTX` with the top-level type, yielding the following:

```
data DTX₃ (ttop : DT) : DT → Set₁ where
  convert : {t₁ : DT} → (t₂ : DT) →
    (⇩ : ⟦ t₁ ⟧ → ⟦ t₂ ⟧) → (⇧ : ⟦ t₂ ⟧ → Maybe ⟦ t₁ ⟧) →
    ((x : ⟦ t₁ ⟧) → (⇧ (⇩ x) ≡ just x)) →
    DTX₃ ttop t₁
  _×_ : {l r : DT} → DTX₃ ttop l → DTX₃ ttop r →
    DTX₃ ttop (l × r)
  Σ : {c : DT} {d : ⟦ c ⟧ → DT} →
    DTX₃ ttop c → ((x : ⟦ c ⟧) → DTX₃ ttop (d x)) →
    DTX₃ ttop (Σ c d)
```

We will need an additional data type that indicates where to insert:

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

We can now define the new constructor, which contains the calculation function:

```
insert : {t : DT} → (t' : DT) → Side →
  (⟦ ttop ⟧ → ⟦ t' ⟧) →
  DTX₃ ttop t
```

In words, this new constructor takes as (non-hidden) arguments an *inserted type*, the side at which a subtree of that type should be inserted, and an *insertion function*.

All functions that work on `DTX` need to be updated with the new `ttop` parameter:

```
extendType₃ : {ttop t : DT} → DTX₃ ttop t → DT
extendValue₃ : {ttop t : DT} → (tx : DTX₃ ttop t) →
  ⟦ ttop ⟧ → ⟦ t ⟧ → ⟦ extendType₃ tx ⟧
retractValue₃ : {ttop t : DT} → (tx : DTX₃ ttop t) →
  ⟦ extendType₃ tx ⟧ → Maybe ⟦ t ⟧
```

The implementations of the updated functions are exactly the same as for their insertionless variants (as defined in the previous section), except for the added cases for the `insert` instructors. They are as follows:

```
extendType₃ {t = t} (insert t' left f) = t' × t
extendType₃ {t = t} (insert t' right f) = t × t'

extendValue₃ (insert t' left f) dtop d = f dtop P., d
extendValue₃ (insert t' right f) dtop d = d P., f dtop

retractValue₃ (insert t' left f) (dl P., dr) = just dr
retractValue₃ (insert t' right f) (dl P., dr) = just dl
```

We will often want to refer to extensions where both `DT` arguments are equal, i.e. `DTX₃ t t` for some `t : DT`. Such extensions behave exactly like their insertionless, one-argument variants. For convenience, we define top-level synonyms subscripted "s" ("self").

```
DTX₃ₛ : DT → Set₁
DTX₃ₛ t = DTX₃ t t
extendType₃ₛ : {t : DT} → DTX₃ₛ t → DT
extendType₃ₛ = extendType₃
extendValue₃ₛ : {t : DT} → (tx : DTX₃ₛ t) →
  ⟦ t ⟧ → ⟦ extendType₃ₛ tx ⟧
extendValue₃ₛ tx d = extendValue₃ tx d d
```

It is important to note that while *insertion* is an operation that can elegantly be added to extension, *removal* of a leaf or subtree is certainly not. 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; in practice, this problem manifests itself as the impossibility of implementing `retractValue` for the `remove` constructor.

3.6 Compositional extension with insertion

A shortcoming of the `insert` constructor as it is currently defined is the fact that – when dependent products are involved – it requires calculations to work for *all* values, even values that do not correspond to the *current* extension. As a concrete example, suppose we have the following type: a dependent product of a number less than five and a vector of that many numbers less than five. (Five is an example, any other number would have worked; bounded natural numbers (`Fin`) have the advantage that they can easily be encoded to binary.)

```
PartialMaximumDep : Fin 5 → DT
PartialMaximumDep i = leaf (Vec (Fin 5) (toℕ i))

PartialMaximum : DT
PartialMaximum = Σ (leaf (Fin 5)) PartialMaximumDep
```

Suppose we have a function that can calculate the maximum of at least one bounded natural number: `maximum : {m n : ℕ} → Vec (Fin m) (suc n) → Fin m`. If the goal is to create an extension that appends the maximum of the number vector *if it can be calculated*, we must start off with a top-level extension:

```
PartialMaximumExt? : DTX₃ₛ PartialMaximum
PartialMaximumExt? = Σ copy₃ PartialMaximumExtDep?
```

The extension for the second (dependent) component seems easy to define...

```
PartialMaximumExtDep? :
  (c : Fin 5) → DTX₃ PartialMaximum (PartialMaximumDep c)
PartialMaximumExtDep? F.zero = copy₃
PartialMaximumExtDep? (F.suc i) =
  insert (leaf (Vec Bool 1)) right {!!}
```

The hole we are left with (the insertion function) has the ominous, but expectable, type `⟦ PartialMaximum ⟧ → Vec Bool 1`; the insertion function receives as its argument the entire top-level value. Unfortunately, if the first (constant) component of this value is `F.zero`, the second (dependent) component *does not contain enough numbers to call* `maximum` *on*! In fact, there is exactly one such

value, namely `F.zero P., V.[]`. What is the insertion function to return in this case?

Of course, by inspecting the definitions of `extendValue`, it is easy to see that the insertion function will only ever be passed values that *do* contain one or more numbers. Consequently, it is possible and tempting to work around the issue by returning some nonsensical default value – perhaps `F.zero` – which would never be produced anyway. This would, however, introduce an unverified ugliness in the system: as long it is present in the code, we cannot claim that it is impossible for it to be produced, for example by a bug in `extendValue`.

Even worse, it is impossible to produce a default value for an arbitrary `DT` (it might contain `⊥`, function types, etcetera). This limitation becomes relevant when considering the natural *embedding* operation on extensions: if we have an extension with top-level type $\texttt{t}_1$, and we have some "larger" type $\texttt{t}_2$ that has $\texttt{t}_1$ as a subtree at some point, we want to be able to *embed* the extension into an extension with top-level type $\texttt{t}_2$. To write the `insert` case of this operation, we need to turn an existing insertion function $[\![\ \texttt{t}_1\ ]\!] \to [\![\ \texttt{t}\ ]\!]$ into the "larger" function $[\![\ \texttt{t}_2\ ]\!] \to [\![\ \texttt{t}\ ]\!]$. As argued above, a value of $[\![\ \texttt{t}_2\ ]\!]$ might not contain a value of $[\![\ \texttt{t}_1\ ]\!]$, but it is impossible to find a of default value of the unknown type $[\![\ \texttt{t}\ ]\!]$ to return in that case.

The underlying issue is that the insertion function in `insert` is too strong: it is required to work on *all* values of the top-level type. As mentioned before, we can easily see that the only values `extendValue` will ever pass to the insertion function of an `insert` contained within a second component of a `Σ` will have the same first component as the one used to produce the extension. In the example, `PartialMaximumExtDep?` pattern-matches on its argument; the insertions inside the `F.suc i` case will only act on values that have `F.suc i` as the first component – but the insertion function does not get access to this information.

We therefore need a way to "save the pattern-matching information in the extension". This motivates us to introduce two additional data types. The first, `Select`, acts as a "pointer" into a data type, describing for types `A` and `B` the relation "`A` can be found as a subtree in `B`"; the `Select` value represents the path to take from the root of `B` to get to `A`. It is defined as follows:

```
data Select (t : DT) : DT → Set where
  left : ∀ {l r} → Select t l → Select t (l × r)
  right : ∀ {l r} → Select t r → Select t (l × r)
  constant : ∀ {c d} → Select t c → Select t (Σ c d)
  dependent : ∀ {c d} → (x : ⟦ c ⟧) →
    Select t (d x) → Select t (Σ c d)
  stop : Select t t
```

Note that this data type has the "smaller" type as its first argument and the "larger" type as the second, which is the opposite of the arguments order of `DTX`. This is caused by a technical limitation (in Agda data type definitions, parameters must precede indices). Also note that when the path involves moving to a second component, the value of the first component is recorded.

With `Select` representing the notion of finding a smaller type in a larger type, we can define the notion `CanSelect`: whether a value *matches* a given `Select`.

```
data CanSelect {t : DT} : {ttop : DT} →
     Select t ttop → ⟦ t ⟧ → ⟦ ttop ⟧ → Set₁ where
```

```
  left : ∀ {l r} {dl : ⟦ l ⟧} {dr : ⟦ r ⟧}
    {s : Select t l} {dbot : ⟦ t ⟧} →
    CanSelect s dbot dl →
    CanSelect {ttop = l × r} (left s) dbot (dl P., dr)
  right : ∀ {l r} {dl : ⟦ l ⟧} {dr : ⟦ r ⟧}
    {s : Select t r} {dbot : ⟦ t ⟧} →
    CanSelect s dbot dr →
    CanSelect {ttop = l × r} (right s) dbot (dl P., dr)
  constant : ∀ {c d} {dc : ⟦ c ⟧} {dd : ⟦ d dc ⟧}
    {s : Select t c} {dbot : ⟦ t ⟧} →
    CanSelect s dbot dc →
    CanSelect {ttop = Σ c d} (constant s) dbot (dc P., dd)
  dependent : ∀ {c d} {dc : ⟦ c ⟧} {dd : ⟦ d dc ⟧}
    {s : Select t (d dc)} {dbot : ⟦ t ⟧} →
    CanSelect s dbot dd →
    CanSelect {ttop = Σ c d} (dependent dc s) dbot (dc P., dd)
  stop : {d : ⟦ t ⟧} → CanSelect stop d d
```

Unfortunately, this definition is rather notation-heavy. The core concept is that if we have `s : Select A B` ("`A` can be found as a subtree in `B`"), and we have `a : ⟦ A ⟧`, `b : ⟦ B ⟧`, we can define "`a` can be found in `b` by following `s`". Because of the types of `a` and `b`, we know this is *almost* always possible - the only case in which it is not is when there is some dependent pair in `B` for which the first component in `b` and the corresponding first component in `dependent` in `s` have different values.

In fact, the `Select` relation could be described more elegantly by noticing that it is the reflexive transitive closure of the "has immediate subtree" relation. Having defined that, we could simply use `Star` from `Data.Star` to get `Select` for free, including operators that *append* two selections and the proof that the append operator is associative. Unfortunately, there is no appropriately indexed version of `Star`, which we would need to define `CanSelect`. Therefore we define append operators manually:

```
_S▻▻_ : {t₁ t₂ t₃ : DT} →
  Select t₂ t₁ → Select t₃ t₂ → Select t₃ t₁

_CS▻▻_ : {t₁ t₂ t₃ : DT} {s₁ : Select t₂ t₁} {s₂ : Select t₃ t₂}
  {d₁ : ⟦ t₁ ⟧} {d₂ : ⟦ t₂ ⟧} {d₃ : ⟦ t₃ ⟧} →
  CanSelect s₁ d₂ d₁ → CanSelect s₂ d₃ d₂ →
  CanSelect (s₁ S▻▻ s₂) d₃ d₁
```

These functions can be implemented using simple recursion on the first argument.

Equipped with `Select` and `CanSelect`, we can define a new version of `DTX` in which the relation between the top-level type and the type currently being extended is recorded by means of a `Select`.

```
data DTX₄ (ttop : DT) : (t : DT) → Select t ttop → Set₁ where
  convert : ∀ {t₁ : DT} {s} → (t₂ : DT) →
    (⇩ : ⟦ t₁ ⟧ → ⟦ t₂ ⟧) → (⇧ : ⟦ t₂ ⟧ → Maybe ⟦ t₁ ⟧) →
    ((x : ⟦ t₁ ⟧) → (⇧ (⇩ x) ≡ just x)) →
    DTX₄ ttop t₁ s
```

```
  _×_ : ∀ {l r : DT} {s} →
    DTX₄ ttop l (s S▷▷ left stop) →
    DTX₄ ttop r (s S▷▷ right stop) →
    DTX₄ ttop (l × r) s
  Σ : ∀ {c : DT} {d : ⟦ c ⟧ → DT} {s} →
    DTX₄ ttop c (s S▷▷ constant stop) →
    ((x : ⟦ c ⟧) → DTX₄ ttop (d x) (s S▷▷ dependent x stop)) →
    DTX₄ ttop (Σ c d) s
  insert : ∀ {t : DT} {s} → (t' : DT) → Side →
    ((dtop : ⟦ ttop ⟧) → P.∃ (λ d → CanSelect s d dtop) → ⟦ t' ⟧) →
    DTX₄ ttop t s
```

In each inductive occurrence of `DTX`, the appropriate `Select` is appended to the current `Select`; crucially, in the `Σ` case, the current value of the first component (`x`) is stored in the `Select` that points to the second. In the `insert` constructor, the insertion function now receives a second argument: an object describing that there exists some smaller value `d : ⟦ t ⟧` such that we can traverse the first argument `dtop` along the current `Select` pointer `s` and arrive at this `d`.

This object solves the problem presented at the beginning of this section. To check it does for `PartialMaximum`, let us try to define the parity-adding extension. Both `PartialMaximumExt` and `PartialMaximumExtDep` are equal to the earlier (question-marked) attempts. The hole we could not fill, the insertion function, is now filled using the following simple implementation:

```
f : {i : Fin 4} → (dtop : ⟦ PartialMaximum ⟧) →
  P.∃ (λ d → CanSelect {ttop = PartialMaximum}
    (dependent (F.suc i) stop) d dtop) →
  (Fin 5)
f ((F.suc i) P., d) (.d P., dependent stop) = maximum d
```

Fully pattern-matching on the `CanSelect` value allows us to discover that the first component is `F.suc i` and the second component is in fact `d`, and we can easily apply `maximum` and return the calculated maximum element.

Finally, for convenience, we might want to recover `insert` as defined in the previous section, where it takes an insertion function with only one argument. To do this, we need to pass an insertion function that ignores the new `CanSelect` argument:

```
insertSimple : {ttop t : DT} {s : Select t ttop} →
  (t' : DT) → Side → (⟦ ttop ⟧ → ⟦ t' ⟧) →
  DTX₄ ttop t s
insertSimple t' s f = insert t' s (λ dtop _ → f dtop)
```

3.7 Embedding

For format descriptions, an essential usability factor is reusability: once we have defined a format, we want to be able to embed it in other formats. Embedding a type description `t : DT` is easy: one can just place `t` in any hole requiring a `DT`, including a hole in a larger type that is expecting a subtree. Embedding an extension (`DTX`), on the other hand, is harder, because of the first `DT` argument,

which is the top-level type. The previous chapter worked out the requirements for this embedding to be possible. The function that represents this embedding has the following type:

```
embedDTX₄ : {t₁ t₂ t : DT} {s : Select t t₁} →
  (s' : Select t₁ t₂) → DTX₄ t₁ t s → DTX₄ t₂ t (s' S▷▷ s)
```

In words, given an object `s` that states that `t₁` can be found as a subtree of `t₂` and an extension of `t` with top-level type `t₁`, we get an extension of the same type with the larger top-level type `t₂`. The implementation follows the recursive structure of `DT` by pattern-matching on `t₂` and is uninteresting except for a technical challenge: the recursive calls return extensions with a third (`Select`) argument shaped `s₁ S▷▷ (s₂ S▷▷ s₃)`, while the respective holes require `(s₁ S▷▷ s₂) S▷▷ s₃`. This is easily solved by applying `subst` using the associativity property of `_S▷▷_`, but the use of `subst` leads to complications when trying to prove properties about the enclosing function.

Furthermore, as also mentioned in the previous chapter, the `insert` case requires "enlarging" a function `⟦ t₁ ⟧ → ⟦ t ⟧` to `⟦ t₂ ⟧ → ⟦ t ⟧`. This transformation is called `deepen` and typed as follows. The implementation recurses on `s'`.

```
deepen : {t₁ t₂ t t' : DT} →
  (s : Select t t₁) → (s' : Select t₁ t₂) →
  ((dtop : ⟦ t₁ ⟧) → P.∃ (λ d → CanSelect s d dtop) → ⟦ t' ⟧) →
  (dtop : ⟦ t₂ ⟧) → P.∃ (λ d → CanSelect (s' S▷▷ s) d dtop) → ⟦ t' ⟧
```

To show the power of embedding, let us revisit the `PartialMaximum` example from the previous chapter. Using `insertSimple`, one can very easily write a self-extension (`DTXₛ`) that inserts the parity bit; embedding can then place it inside the second (dependent) component without the need for inspecting a value of `CanSelect`.

```
PartialMaximumExtDep′ : (c : Fin 5) →
  DTX₄ PartialMaximum (PartialMaximumDep c) (dependent c stop)
PartialMaximumExtDep′ F.zero = copy₄
PartialMaximumExtDep′ (F.suc i) = embedDTX₄
  (dependent (F.suc i) stop)
  (insertSimple (leaf (Fin 5)) right maximum)

PartialMaximumExt′ : DTX₄ₛ PartialMaximum
PartialMaximumExt′ = Σ copy₄ PartialMaximumExtDep′
```

3.8 Equality

As concisely explained by Altenkirch, McBride and Swierstra [2, section 1], equality is no trivial topic in systems based on intensional type theory, such as Agda. One of the most conspicuous issues is the lack of built-in functional extensionality for the standard propositional equality, that is, the following property:

```
extensionality : ∀ {a b} →
  {A : Set a} {B : A → Set b} {f g : (x : A) → B x} →
  (∀ x → f x ≡ g x) → f ≡ g
```

Note that this property is only missing from (unprovable), but not inconsistent with Agda's implementation of type theory. This means we could `postulate` it if we wished, but we would like to minimise the number of axioms added. Sadly, the lack of functional extensionality is relevant when trying to prove two types equal if they contain any dependent products. The second (dependent) component of a dependent product is defined as a function, so although it might be easy to prove two such functions pointwise equal, that does not suffice for standard propositional equality `_≡_`.

An interesting pair of properties involving equality would be the preservation of meaning by `embedDTX`. Concretely, we would like `extendType` and `extendValue` to have the same effect for a given extension and any embedding of that extension. Assuming we have some flexible enough equality `_≋_`, these properties can be expressed as follows:

```
embedDTX-preserves-type : {t₁ t₂ t : DT} {s : Select t t₁} →
  (s' : Select t₁ t₂) → (tx : DTX₄ t₁ t s) →
  extendType₄ tx ≋ extendType₄ (embedDTX₄ s' tx)

embedDTX-preserves-value : {t₁ t₂ t : DT} {s : Select t t₁} →
  (s' : Select t₁ t₂) → (tx : DTX₄ t₁ t s) →
  (d₁ : ⟦ t₁ ⟧) → (d₂ : ⟦ t₂ ⟧) → CanSelect s' d₁ d₂ →
  (d : ⟦ t ⟧) →
  extendValue₄ tx d₁ d ≋ extendValue₄ (embedDTX₄ s' tx) d₂ d
```

In fact, it is not possible to even *write* the latter property in terms of `_≡_`, because not even the types of the left and right argument are equal and that equality is homogeneous. These two issues (the lack of functional extensionality and the requirement that the types be equal) imply that propositional equality is too strong. In cases like these, it is common (see the standard library) to define a weaker equality. The first attempt is to define a *per-constructor* equality, such as this:

```
data _≣′_ : DT → DT → Set₁ where
  leaf : {A : Set} → leaf A ≣′ leaf A
  _×_ : {l₁ l₂ r₁ r₂ : DT} →
    l₁ ≣′ l₂ → r₁ ≣′ r₂ → (l₁ × r₁) ≣′ (l₂ × r₂)
  Σ : {c₁ c₂ : DT} {d₁ : ⟦ c₁ ⟧ → DT} {d₂ : ⟦ c₂ ⟧ → DT} →
    c₁ ≣′ c₂ → {!!} → Σ c₁ d₁ ≣′ Σ c₂ d₂
```

As usual, the hard case is the equality of second (dependent) components: we need to state the property that `d₁` and `d₂` are pointwise equal, but their argument types are not the same...

One solution is to introduce an (implementable) function `≣′-coerce : {t₁ t₂ : DT} → t₁ ≣′ t₂ → ⟦ t₁ ⟧ → ⟦ t₂ ⟧`, name the equality of the *first* (constant) components `eqc` and write the required equality of the *second* (dependent) components as `(x : ⟦ c₁ ⟧) → d₁ x ≣′ d₂ (≣′-coerce eqc x)`. This solution is asymmetrical, and that asymmetry leads to serious complications when trying to prove that `_≣′_` is an equivalence relation (and most likely also when trying to use any property involving `_≣′_` in another proof).

A better solution is to quantify over both `c₁` and `c₂` and then require those values to be equal according to some other equality. The equality of second components then becomes `(dc₁ : ⟦ c₁ ⟧) → (dc₂ : ⟦ c₂ ⟧) → dc₁ ≋ dc₂ → d₁ dc₁`

`≡′ d₂ dc₂`. This leaves us with the question of which value equality to use for `dc₁ ≋ dc₂`. As their types differ, the obvious candidate is standard heterogenous equality (`Relation.Binary.HeterogeneousEquality`), but that leads to similar proof problems, as we are (rightfully) unable to prove the types equal in the second component case. These problems can be fixed by using a *per-constructor* definition similar to `_≡′_`, namely `data _≋′_ : {t₁ t₂ : DT} → ⟦ t₁ ⟧ → ⟦ t₂ ⟧ → Set`. The final hurdle is that because `⟦_⟧` is a function, Agda cannot infer the values of `t₁` and `t₂`, so we need a notation in which those are made explicit:

```
data ⟨_∋_⟩≅⟨_∋_⟩ :
     (t₁ : DT) → ⟦ t₁ ⟧ → (t₂ : DT) → ⟦ t₂ ⟧ → Set₁ where
  leaf : {A : Set} {x : A} → ⟨ leaf A ∋ x ⟩≅⟨ leaf A ∋ x ⟩
  _,′_ : ∀ {l₁ l₂ r₁ r₂ : DT} {dl₁ dl₂ dr₁ dr₂} →
     ⟨ l₁ ∋ dl₁ ⟩≅⟨ l₂ ∋ dl₂ ⟩ →
     ⟨ r₁ ∋ dr₁ ⟩≅⟨ r₂ ∋ dr₂ ⟩ →
     ⟨ l₁ × r₁ ∋ dl₁ P., dr₁ ⟩≅⟨ l₂ × r₂ ∋ dl₂ P., dr₂ ⟩
  _,_ : ∀ {c₁ c₂ : DT} {d₁ : ⟦ c₁ ⟧ → DT} {d₂ : ⟦ c₂ ⟧ → DT}
     {dc₁ dc₂ dd₁ dd₂} →
     ⟨ c₁ ∋ dc₁ ⟩≅⟨ c₂ ∋ dc₂ ⟩ →
     ⟨ d₁ dc₁ ∋ dd₁ ⟩≅⟨ d₂ dc₂ ∋ dd₂ ⟩ →
     ⟨ Σ c₁ d₁ ∋ dc₁ P., dd₁ ⟩≅⟨ Σ c₂ d₂ ∋ dc₂ P., dd₂ ⟩
```

Finally, we can define a useful variant of `_≡′_`:

```
data _≡_ : DT → DT → Set₁ where
  leaf : {A : Set} → leaf A ≡ leaf A
  _×_ : {l₁ l₂ r₁ r₂ : DT} →
     l₁ ≡ l₂ → r₁ ≡ r₂ → (l₁ × r₁) ≡ (l₂ × r₂)
  Σ : {c₁ c₂ : DT} {d₁ : ⟦ c₁ ⟧ → DT} {d₂ : ⟦ c₂ ⟧ → DT} →
     c₁ ≡ c₂ →
     ((dc₁ : ⟦ c₁ ⟧) → (dc₂ : ⟦ c₂ ⟧) →
        ⟨ c₁ ∋ dc₁ ⟩≅⟨ c₂ ∋ dc₂ ⟩ → d₁ dc₁ ≡ d₂ dc₂) →
     Σ c₁ d₁ ≡ Σ c₂ d₂
```

Both these relations are equivalence relations (which is moderately difficult to prove), and we can define a number of useful functions and properties:

```
≋-to-≡ : {t : DT} {d₁ d₂ : ⟦ t ⟧} →
  ⟨ t ∋ d₁ ⟩≅⟨ t ∋ d₂ ⟩ → d₁ ≡ d₂
≡-coerce : {t₁ t₂ : DT} →
  t₁ ≡ t₂ → ⟦ t₁ ⟧ → ⟦ t₂ ⟧
≋-≡-coerce : {t₁ t₂ : DT} → (eq : t₁ ≡ t₂) →
  (d : ⟦ t₁ ⟧) → ⟨ t₁ ∋ d ⟩≅⟨ t₂ ∋ ≡-coerce eq d ⟩
```

In short, it turns out that the appropriate equalities for both values and types are *per-constructor* equalities, and the most usable value equality is heterogeneous.

The aforementioned properties of embedding can be proved, albeit quite onerously, with the appropriate equalities: `embedDTX-preserves-type` using `_≡_` and `embedDTX-preserves-value` using `_≋_`. One useful application of these properties is that we can easily derive the following property of `IsLowLevel`:

```
ill-coerce : {t₁ t₂ : DT} →
  t₁ ≡ t₂ → IsLowLevel t₁ → IsLowLevel t₂
```

Combining this property with `embedDTX-preserves-type`, we are able to transform an instance of `IsLowLevel` for the target type of an extenion into an instance for any embedding of it, increasing reusability.

3.9 Simultaneous construction

Tervoort [20] explored many different protocol description languages in the context of (concurrency-related) testing. He distilled a format description language in which fields are declared with their types directly alongside with *codecs*, which describe how those types should be transformed into a low-level representation. For example, he describes an archived file [20, p. 15] using, in part, the following definition:

```
record ArchivedFile with
    size is Integer(min=0, max=2^64-1)
        as BigEndian(length=64, signed=false)
    file_data is Binary(length=size*8)
        as FixedLengthBinary
end
```

The downside of `DT` and `DTX` in this regard is that "codecs" (`convert` values) are not specified textually close to the corresponding fields, which harms readability for large records. We can define a data type `DT&X` which allows simultaneous construction of a `DT` and a corresponding `DTX`, along with a function to extract said values:

```
mutual
  data DT&X : Set₁ where
    leaf : (A : Set) → DTX₃ₛ (leaf A) → DT&X
    _×_ : DT&X → DT&X → DT&X
    Σ : (c : DT&X) → ((x : ⟦ P.proj₁ (fromDT&X c) ⟧) → DT&X) →
      DT&X

  fromDT&X : DT&X → P.Σ DT DTX₃ₛ
```

The implementation of `fromDT&X` is involved but uninteresting. Assuming that we have an encoding extension `bigEndianConversion : {l : ℕ} → DTX₃ₛ (leaf (Fin (2^ l)))` (perhaps defined in a library), this construct allows us to define the same format as `ArchivedFile` as follows:

```
archivedFile : DT&X
archivedFile = Σ
  (leaf (Fin (2^ 64)) bigEndianConversion)
  (λ len → leaf (Vec Bool (toℕ len)) copy₃)
```

Note that `DT&X` only allows self-extensions (`DTXₛ`), which means the extension for any field can only use the value of *that* field in any calculations inside it. This of course prevents the description of checksums and other common constructs, but it is a limitation that is strongly implied by Agda's basic capabilities: we would need a form of laziness that would allow us to refer to the resulting top-level type during the simultaneous `DT&X`-construction, but it contains no such thing (nor is it obvious that such a mechanism would have consistent semantics or would be implementable).

Chapter 4

Case Study: IPv4

Equipped with the universe and extension as defined in the previous chapter, we are ready to define the format of IPv4 packets. A distinction should be made between the end-user-facing high-level "IPv4 format" that is a high-level description of a packet's data and the actual low-level IPv4 format that has an `IsLowLevel` instance. As powerful extension steps can be taken to translate between the two formats, the former is not bound by the order or even concepts of the latter, and we should start by designing the end-user-facing format to be as user-friendly as possible.

4.1 The end-user-facing type

Summarising the tabular representation of the specification on page 10, there are six categories of fields in the IPv4 format:

Constants The Version field must contain the constant value 0100.

Bounded natural numbers The Internet Header Length, the Total Length, the Identification, the Fragment Offset, the Time to Live, the Header Checksum, and the Addresses are fields that 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 enumerations. They can be implemented as simple (non-dependent) Agda data types, and functions that encode and decode them to boolean vectors can be written manually or derived using Agda's reflection features. 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 exactly some 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 * l`, where `l` is a number between 0 and 10, inclusive: 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 – see below.

The Version is a constant value, and as explained in the description of problem 3 on page 12, the Internet Header Length and Total Length are inconvenient for end users. Consequently, we aim to insert those into the data by applying extensions; all other fields will need to be present in the end-user-facing type.

Fixed-length fields

Although we developed a *complete* description of IPv4 during this project, having discussed these categories, we think it suffices to present the implementation of only a *single* example: an enumeration, the Explicit Congestion Notification.

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

ECN→Bool : ECN → Vec Bool 2
ECN→Bool Non-ECT = false V.∷ false V.∷ V.[]
ECN→Bool ECT0 = true V.∷ false V.∷ V.[]
ECN→Bool ECT1 = false V.∷ true V.∷ V.[]
ECN→Bool CE = true V.∷ true V.∷ V.[]

Bool→ECN : Vec Bool 2 → ECN
Bool→ECN (false V.∷ false V.∷ V.[]) = Non-ECT
Bool→ECN (true V.∷ false V.∷ V.[]) = ECT0
Bool→ECN (false V.∷ true V.∷ V.[]) = ECT1
Bool→ECN (true V.∷ true V.∷ V.[]) = CE

ECN↔Bool : (x : ECN) → Bool→ECN (ECN→Bool x) ≡ x
ECN↔Bool Non-ECT = refl
ECN↔Bool ECT0 = refl
ECN↔Bool ECT1 = refl
ECN↔Bool CE = refl
```

Similar definitions were developed for the other simple fields.

Variable-length fields

We are left with the two more complex variable-length fields. As discussed in section 3.1, the ability to contain high-level types is present in our universe description precisely for fields that second (dependent) components' types depend upon. We have two variable-length fields, Options and Data, for which lengths need to be placed somewhere. Suppose we 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 already had the restriction that `OL ≤ 10`, the type of a pair of numbers that satisfies exactly the appropriate conditions can be expressed as follows:

```
Lengths = P.Σ
  (ℕ P.× ℕ)
  (λ { (DL P., OL) → OL ≤ 10 P.× 4 * (5 + OL) + DL < 2^ 16 })
```

This type is regrettably complex, but 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 constructed. We define the two components of the dependent pair are defined as separate named bindings for ease of manipulation. The first (constant) component contains the lengths and three fields: the Explicit Congestion Notification and the Addresses. We omit the other fields – they can be implemented as described above and added before and after the Explicit Congestion Notification. The second (dependent) component contains the Options and Data vectors.

```
IPv4TypeCons : DT
IPv4TypeCons =
  leaf Lengths ×
  (leaf ECN ×
    (leaf (Vec Bool (2^ 32)) × leaf (Vec Bool (2^ 32))))

IPv4TypeDep : ⟦ IPv4TypeCons ⟧ → DT
IPv4TypeDep (((DL P., OL) P., _) P., _) =
  leaf (Vec Bool (32 * OL)) × leaf (Vec Bool (8 * DL))

IPv4Type : DT
IPv4Type = Σ IPv4TypeCons IPv4TypeDep
```

4.2 Extensions

To bring the end-user-facing type down to low-level, we apply three extensions; we will discuss each separately. For brevity, we omit the code of the extensions. Furthermore, we will use `DTX₃` instead of the newest iteration (`DTX₄`), as it leads to less cluttered definitions, and the compositionality of the `DTX₄` is not needed. Naturally, our actual implementation does use `DTX₄`.

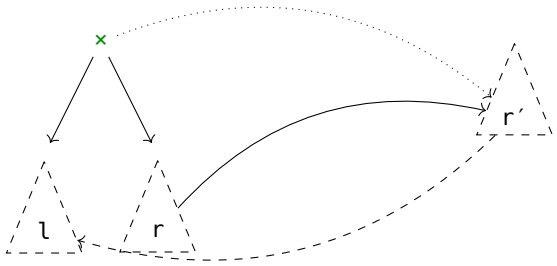
4.2.1 First extension: mixing

The first extension has two responsibilities: adding the constant Version field, and transforming the convenient `Lengths` into the protocol-mandated fields.

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 "smart constructor" 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 : {l r : DT} {ttop : DT} →
  (tx : DTX₃ (l × r) r) →
  (rf : ⟦ extendType₃ tx ⟧ → Maybe ⟦ l ⟧) →
  ((d₁ : ⟦ l ⟧) → (d₂ : ⟦ r ⟧) →
    rf (extendValue₃ tx (d₁ P., d₂) d₂) ≡ just d₁) →
  DTX₃ ttop (l × r)
```

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



On the left we see the source type of the extension – the pair `l × r` – and on the right the result type. The result type is the result of extending `r` along the extension `tx`, as represented by the solid arrow. That extension's top-level type is not `ttop`, but `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 (right) component and recover the first (left) component of the pair. Of course, a proof of the round-trip property is also required.

With this smart constructor in our toolbox, we can define this first extension: adding in the IHL and the TL proceeds by the use of `insert`, performing the required arithmetic on `OL` and `DL`; recovery involves pattern-matching on the right component's value, but is not difficult. The first (constant) component of the result type of this extension is the following:

```
IPv4TypeCons1 : DT
IPv4TypeCons1 = (leaf ECN ×
```

```
  (leaf (Fin (2^ 4)) {- IHL -} × leaf (Fin (2^ 16)) {- TL -})) ×
  (leaf (Vec Bool (2^ 32)) × leaf (Vec Bool (2^ 32)))
```

The dependent type is processed only using the identity extension `copy`.

4.2.2 Second extension: binary encoding

The second extension brings all fields that are not yet low-level down to low-level. The just-added IHL and TL are processed using using big-endian binary encoding, while `ECN→Bool` and the two related functions are used for the ECN. Again, the second (dependent) component is simply copied, and the first (constant) component is now as follows:

```
IPv4TypeCons2 : DT
IPv4TypeCons2 = (leaf ECN ×
  (leaf (Vec Bool 4) {- IHL -} × leaf (Vec Bool 16) {- TL -})) ×
  (leaf (Vec Bool (2^ 32)) × leaf (Vec Bool (2^ 32)))
```

4.2.3 Third extension: checksum insertion

The third extension 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 extensions, the type of the second (dependent) component (which we need to pattern-match on to extract the Options) is no longer the friendly function at the start: the definition of `extendType` has added two applications of `retractValue` on top, as described at the end of section 3.3. The solution is to add `with`-patterns matching on those applications of `retractValue`, recovering the original `Lengths` and allowing easy access to Options. After the checksum is inserted, the first (constant) component contains all necessary values:

```
IPv4TypeCons3 : DT
IPv4TypeCons3 = ((leaf ECN ×
  (leaf (Vec Bool 4) {- IHL -} × leaf (Vec Bool 16) {- TL -})) ×
    leaf (Vec Bool 16) {- Checksum -}) ×
  (leaf (Vec Bool (2^ 32)) × leaf (Vec Bool (2^ 32)))
```

4.2.4 `IsLowLevel` instance

The final piece of the puzzle is the instance of `IsLowLevel` for the result type of the third extension. Because of the presence of `retractValue` in the type of the second component as described above, this is not entirely trivial. Fortunately, the second component *was already low-level* in the end-user-facing type description. With the use of the `from-⊥` constructor of `IsLowLevel`, we can write a function that shows that `IsLowLevel` is preserved for a second component of a dependent pair whose first component is extended while the second component is just copied.

The instance for the first component, on the other hand, can be found by instance search, and this completes our implementation: we now have a convenient end-user-facing type, a series of extensions that transform it into the

low-level representation, and an instance that allows users to pretty-print and parse values to and from lists of booleans.

4.3 Testing

We were curious to see whether the packets produced by our pretty-printer would be recognised by "official" parsers. Although there is a surprising scarcity of Haskell IPv4 parsers that can handle the entire protocol (including, for example, options), the least we could do was test the checksum calculation. The checksum implementation of the *network-house* Haskell library was used for testing 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! In this situation, the round-trip property is of no use: certainly, round-trips are also possible if most of the packet's fields are reversed. Careful reading is therefore still highly advisable.

During testing, we discovered that the space and time efficiency of the typechecker's evaluator appears to be significantly worse than the efficiency of the same code compiled to Haskell using the GHC backend, then executed using GHCi. We later received the advice that this is most likely caused by excessive laziness in our own checksum function and we could improve this by using a recently added forcing primitive.

Chapter 5

Discussion and Variants

Both designing a language and exploring its capabilities are processes that involve decisions between more or less equally preferable alternatives. In this chapter, we justify some design decisions, compare the expressive power of our system to that of others, and consider the relevance of some unfinished work.

5.1 Comparison with features of record types

Although we claimed in section 3.1 that our universe approximates Agda's records, a few features are still missing. Most prominently absent are field names and the ability to self-reference (induction and coinduction).

5.1.1 Field names

Field names could actually be added for *simple* products with relative ease, yielding the "Data Type with Name" ($\mathtt{DT}_n$) universe:

```
open import Data.String
data DTₙ : Set₁ where
  leaf : String → (A : Set) → DTₙ
  _×_ : DTₙ → DTₙ → DTₙ
```

We need to define a proof object stating that a type really contains a field with a given name:

```
data HasLeaf (f : String) : DTₙ → Set₁ where
  instance
    leaf : {A : Set} → HasLeaf f (leaf f A)
    left : {l r : DTₙ} → HasLeaf f l → HasLeaf f (l × r)
    right : {l r : DTₙ} → HasLeaf f r → HasLeaf f (l × r)
```

It is then possible to define functions that look up types and values by field name:

```
lookupType : (f : String) → (t : DTₙ) →
  (h : HasLeaf f t) → DTₙ
lookupValue : (f : String) → (t : DTₙ) →
  (h : HasLeaf f t) → ⟦ t ⟧ₙ → ⟦ lookupType f t h ⟧ₙ
```

Instances of `HasLeaf` can be found by Agda's instance search via the `it` pattern (as described on page 18). Although name uniqueness is enforced neither in `DTN` nor in `HasLeaf`, instance search refuses to return an instance if it is not the unique instance for that type, which ensures that lookups using the `it` pattern only succeed when unique names are used.

This structure could be extended further to allow naming entire subtrees by also requiring a name in the `_×_` constructor; names could be made optional by using `Maybe String` instead of `String`; and so on.

We still chose to omit field names from our main universe `DT` for several reasons. First of all, field names add some clutter to the development without providing significant functionality; they are advantageous for usability, so it would be wise to include them in a production-quality EDSL. Second, they introduce a dependency on `String` and rely heavily on instance search, both of which slow down type checking and increase the system's complexity. Third, field names in *dependent* products are harder to work with, as they lead to types of which *some* values do contain a given field and others do not. This calls for the introduction of a value-based `HasLeaf`, further complicating the system.

More important than all those reasons, however, is that field names inside the universe are not relevant to end users: it is much more important to be able to convert between values of type `⟦ t ⟧` and the corresponding record type. Automating those conversions is discussed in section 5.3.

5.1.2 Self-reference

Undoubtedly, self-references in data types (inductive and coinductive positions), are a basic feature of the type system, necessary for the definition of even common types like `List`. The usual solution for adding this feature to a universe is the introduction of an "inductive position" constructor, which is applied by many of the generic programming frameworks in [13].

One must keep in mind, however, that for inductive *record* types, "naked" self-references (a field of type `A` inside the – parameterless – record type `A`) are useless: it is impossible to construct a value of such a record, because any such construction must continue infinitely. A deeper self-reference such as a field of type `Maybe A` can be useful, but that requires a more complex self-reference constructor that allows the application of another type constructor.

Oury and Swierstra [16, section 3.6] hint at the most likely candidate in more concrete terms:

> The more general solution, however, would be to extend our universe with variables and a least-fixed point operation. [...] We have refrained from doing so as the resulting universe must deal with variable binding. Although the solution is not terribly complex (Morris et al. 2004), we felt the technical overhead would distract from the bigger picture.

The usefulness of these constructs is related to the presence or absense of *choice* in the format description, which we will discuss in the next section.

5.2 Choice and backtracking

Almost all format description languages, including ours, offer some form of *choice*: at a given point, a data type may contain either one group of fields or another. The EDSL introduced in section 1.3, for example, offers additional constructors not discussed there, most importantly `plus : FT₃ → FT₃ → FT₃`. Its semantics are defined as follows:

> The `plus` constructor introduces left-biased choice. A parser for `plus f₁ f₂` will try to parse the format determined by `f₁`. Only when that fails, will it try to parse `f₂`.

Accompanying this are constructors for explicit failure (`bad`) and explicit success (`end`). The effect of these semantics is that a parser needs to perform potentially very costly *backtracking*: a certain low-level element (bit) might have to be read multiple times, first in the process of trying `f₁`, then when trying `f₂`. (For certain classes of grammars, such as regular expressions, one can instead write a more efficient parser by precomputing an automaton. We hypothesise that our dependent product makes this impossible, because trying a subformat might require executing arbitrary Agda code, which could probably take too long to precompute.) We decided to focus on formats that do not require backtracking.

Note that the lack of `plus` does *not* imply that no variant of choice is present in our language. The dependent product allows for any finite number of choices for the second (dependent) component. Even an infinite number of choices can be represented, such as in `Σ ℕ (Vec Bool)`. Because of the pigeonhole principle, however, there is no way to construct a semipartial isomorphism between a type with an infinite number of inhabitants (such as `ℕ`) and `Vec Bool n` for some `n`, which is required to turn it into a low-level type.

The only requirement for choices represented by a dependent product is that the choice is made based solely on data that *precedes* the second (dependent) component. Nonetheless, there are formatting constructs that contain some kind of "inherent end", for which parsing does not require backtracking, yet the choice – most commonly, the choice of length – must be made by reading data for the second (dependent) component. The most prominent example are null-terminated strings, which should be read exactly until a null character is encountered. Such cases can be added, at the cost of some complexity, to our language by adding a constructor to the `IsLowLevel` data type:

```
special : (t : DT) →
  (aToList : ⟦ t ⟧ → List Bool) →
  (aFromList : List Bool → Maybe (⟦ t ⟧ P.× List Bool)) →
  (a-left-inv :
    (rest : List Bool) → (d : ⟦ t ⟧) →
    aFromList (aToList d L.++ rest) ≡ just (d P., rest)) →
  IsLowLevel′ t
```

Just like `convert` represents treewise compositionality of semipartial isomorphisms in `DTX`, `special` represents treewise compositionality of the pretty-printing and parsing algorithms: the types of the three arguments `aToList`, `aFromList`, and `a-left-inv` match those of `toList`, `fromList`, and `left-inv` from section 3.2, respectively.

5.3 Conversion from and to record types

The reader might have noticed that one kind of conversion, which is quite important for usability, is missing: the conversion between values of actual record types and values of their `DT`-based representations. The implementations of these conversion functions are completely determined by the shape of the record type, which means leaving it up to format designers to implement them is not optimally user-friendly.

Agda offers *reflection* for inspecting the shape of types, including record types, and for generating functions based on these shapes; this enables the automatic generation of conversion functions. This is explored in depth by Sijsling [19]: not only the value-level conversion functions, but even the value of `DT` describing a given record type can be generated automatically. The opposite, generating a record declaration from a `DT` value, is harder because declaring field names seems to be unsupported at the moment.

We decided not to implement automatic generation of conversion functions because we deemed the insight that could be gained by that (mainly engineering) process to be less than the cost of the required amount of programming. Additionally, Agda's reflection interface is not yet stable (according to the change log, it received a "massive overhaul" in a version released during the project), meaning we would risk producing a comparatively large software component that could very well break completely due to a new release.

5.4 Expressiveness: power of extension

Considering that we want each extension step to be a semipartial isomorphism, and the `convert` constructor allows the use of *any* semipartial isomorphism, extension is maximally powerful within the constraints we placed on our system. In practice, its power as perceived by format developers is limited by the expressiveness of Agda, as well as the library of "language primitives" (functions, since this is an EDSL) available to them.

During this project – especially during the implementation of the IPv4 protocol description – the expressiveness of Agda was not a limiting factor. The greatest inconvenience was the common clash between efficient implementations and those that are easy to reason about, as well as the performance of typechecker's evaluator. Having to prove termination and totality was never an issue. The majority of the IPv4 code consists of (equational) proofs regarding equalities and inequalities involving natural numbers.

Few language primitives are currently implemented. Extensions implementing big-endian and little-endian binary encoding, as well as unary encoding are provided. We looked at implementing string encoding algorithms such as UTF-8 and decided that their recursion structure and termination is simple and the majority of the work would be numerical calculations and related proofs. Automatically deriving extensions that translate between simple enumerations (characterised as parameter- and indexless data types with only nullary constructors) is another candidate for implementation using reflection.

Finally, as `mix` demonstrates, extension itself is extensible: common patterns of extension can be implemented in terms of `convert`.

Chapter 6

Conclusion

In this thesis, we surveyed existing format description languages, comparing their relative advantages and disadvantages and the extent to which their related algorithms have been proven correct. We then identified an example, the IPv4 packet format, that is particularly resistant to formal description. We set out to create a user-friendly, verified, embedded domain-specific language in Agda that is capable of describing this format.

The central idea that underlies our solution is the separation between the *universe*, which describes the shape of a type to be processed, and *extension*, a transformation that translates between high-level and low-level types with an attached proof of correctness. Universe-described types containing only low-level fields can easily be pretty-printed and parsed, and extension is constructed so as to enable adding data calculated from the high-level value to the low-level representation. By ensuring that both the low-level pretty-printer and parser and the transformation described by extensions are semipartial isomorphisms, we receive a proof of correctness of the entire system.

Furthermore, we recognised a lack of compositionality that hindered reuse and adjusted our design to fix this, and we developed appropriate concepts of equality to allow more general proofs of properties related to our algorithms.

Finally, we discussed many of our design decisions, explaining how and at which cost features missing from our final design could be added.

We have fulfilled both goals stated in chapter 2: our language is powerful enough to describe IPv4, as demonstrated in our case study, and its pretty-printing and parsing algorithms are verified to have a strong round-trip property. In addition to the central case study, we showed (in section 3.9) how our language partially subsumes a language powerful enough to describe various real-world protocols. In short, we have achieved our stated goals and developed a system with promising potential.

6.1 Future work

Throughout this thesis, we identified many different ways in which our work could be extended. A short summary follows.

- *Automatic conversion functions.* Agda's reflection capabilities should be powerful enough to derive conversion functions that can translate between

values of record types and values in our universe, eliminating the final usability hurdle for end users.

- *A library of common types and extensions.* Binary coded decimal, Chen-Ho encoding, IEEE 754 floating point numbers, and UTF string encodings are just a few examples of types and encodings for which representations could be made available in our language.
- *Backtracking.* As explained in section 5.2, our current language does not allow the description of formats that positively require backtracking.
- *Additional case studies.* Besides IPv4, we briefly looked at WebSocket, concluding its implementation would mainly be an Agda coding exercise rather than a good demonstration of this EDSL.
- *Improved Haskell interface.* As the power of Haskell's type system grows and more and more features approximating dependent types are added, it might be interesting to try to connect the Agda-based algorithms to Haskell data types.
- *Arbitrary value generation for testing.* Haskell's *QuickCheck* library is capable of generating arbitrary instances of data types for testing purposes. Tervoort [20] notes that naive generation of instances leads to underrepresentation of certain values; Yorgey [21] recently provided an algorithm for fixing this problem.

Acknowledgements

I would like to thank my supervisor, Wouter Swierstra, for his ever-constructive criticism of my work and writing, his valiant efforts to appreciate the subtle technical issues I encountered, and the many intense but productive discussions.

I thank João Paulo Pizani Flor and the attendees of the twenty-fourth Agda Implementors' Meeting for their insightful comments, and express my gratitude towards Will van Geest and Ágnes Soós for their support and encouragement.

Bibliography

[1] Thorsten Altenkirch. Faking induction-induction in Coq? `http://narkive.com/XFOMZMpb.9`. Retrieved 2016-04-14.

[2] Thorsten Altenkirch, Conor McBride, and Wouter Swierstra. Observational equality, now! In *Proceedings of the 2007 workshop on Programming languages meets program verification*, pages 57–68. ACM, 2007.

[3] Ana Bove, Peter Dybjer, and Ulf Norell. A brief overview of Agda–a functional language with dependent types. In *TPHOLs*, volume 5674, pages 73–78. Springer, 2009.

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

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

[6] The Coq Development Team. The Coq Reference Manual. `https://coq.inria.fr/distrib/current/refman/`. Retrieved 2016-04-14.

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

[8] Kathleen Fisher and Robert Gruber. 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, pages 295–304, New York, NY, USA, 2005. ACM.

[9] Peter Hancock, Conor McBride, Neil Ghani, Lorenzo Malatesta, and Thorsten Altenkirch. Small induction recursion. In *Typed Lambda Calculi and Applications*, pages 156–172. Springer, 2013.

[10] Paul Hudak. Building domain-specific embedded languages. *ACM Computing Surveys (CSUR)*, 28(4es):196, 1996.

[11] John Hughes. The design of a pretty-printing library. In *International School on Advanced Functional Programming*, pages 53–96. Springer, 1995.

[12] Graham Hutton and Erik Meijer. Monadic parsing in Haskell. *Journal of functional programming*, 8(04):437–444, 1998.

[13] José Pedro Magalhães and Andres Löh. A formal comparison of approaches to datatype-generic programming. In James Chapman and Paul Blain Levy, editors, *Proceedings Fourth Workshop on Mathematically Structured Functional Programming, Tallinn, Estonia, 25 March 2012*, volume 76 of *Electronic Proceedings in Theoretical Computer Science*, pages 50–67. Open Publishing Association, 2012.

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

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

[16] Nicolas Oury and Wouter Swierstra. The power of Pi. In *ACM Sigplan Notices*, volume 43, pages 39–50. ACM, 2008.

[17] J. Postel. Internet Protocol. RFC 791 (INTERNET STANDARD), September 1981. Updated by RFCs 1349, 2474, 6864.

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

[19] Yorick Sijsling. Generic programming with ornaments and dependent types. Master's thesis, Utrecht University, 2016.

[20] Tom Tervoort. A protocol specification language for aiding and testing implementations. Master's thesis, Utrecht University, 2016.

[21] Brent Yorgey. Boltzmann sampling for random generation of algebraic data types in Haskell. `https://github.com/byorgey/boltzmann`. Retrieved 2016-10-30.