JFP **33**, e12, 18 pages, 2023.

doi:10.1017/S095679682200017X

FUNCTIONAL PEARL

Certified, total serialisers with an application to Huffman encoding

RALF HINZE
RPTU Kaiserslautern-Landau, Germany
(*e-mail:* ralf.hinze@cs.rptu.de)

1 Introduction

The other day, I was assembling lecture material for a course on Agda. Pursuing an application-driven approach, I was looking for correctness proofs of popular algorithms. One of my all-time favourites is Huffman data compression (Huffman, 1952). Even though it is probably safe to assume that you are familiar with this algorithmic gem, a brief reminder of the essential idea may not be amiss.

A common representation of a textual document on a computer uses the ASCII character encoding scheme. The scheme uses seven or eight bits to represent a single character. The idea behind Huffman compression is to leverage the fact that some characters appear more frequently than others. Huffman encoding moves away from the fixed-length encoding of ASCII to a variable-length encoding, in which more frequently used characters have a shorter bit encoding than rarer ones. As an example, consider the text

```
eight␣in␣the␣evening
```
(1.1)

and observe that the letter `e` occurs four times, whereas `v` occurs only once. The code table shown below gives a possible encoding for the eight different letters occurring in the text.

␣	111	e	01	g	1101	h	000
i	100	n	101	t	001	v	1100

(1.2)

The more frequent a character, the shorter the code. Using the encoding above, the text (1.1) is compressed to the following bit string:

```
01100110100000111110010111100100001111011100011011001011101
```

The compressed text only occupies 59 bits, comparing favourably to the 160 bits of the standard ASCII encoding.

There is an impressive proof of correctness and optimality given in Coq (Théry, 2004). Unfortunately, the formalisation is slightly too elaborate to be presented in a single

lecture as it consists of more than 6K lines of code. The excellent text book on "Verified Functional Programming" (Stump, 2016) contains a case study on "Huffman Encoding and Decoding." Alas, even though framed in Agda, the chapter treats neither correctness nor optimality.

So I decided to give it a go myself. Compared to the formalisation in Coq, my goals were more modest and more ambitious at the same time. More modest, as I decided to concentrate on correctness: decoding an encoded text gives back the original string. More ambitious, as I was aiming for a compositional design: Huffman compression should work seamlessly with standard bit serialisers.

Setting up a library for de/serialisers turned out to be a challenge in its own right. Indeed, we dedicate half of the pearl to its design and implementation. From an implementation perspective, a serialiser is like a pretty *p*rinter, except that its output, a bit string, is not particularly pretty. Likewise, a deserialiser can be seen as a simple recursive-descent *p*arser. A parser is a partial function: it fails with an error message if it does not recognise the input; a recursive-descent parser may also fail to terminate. The library addresses the two sources of partiality in different ways. The first type of failure is unavoidable: if a deserialiser is applied to faulty data, it may unexpectedly exhaust the input. However, a deserialiser applied to the output of its mate should return the original value, like a Huffman decoder. To establish correctness, the library offers a set of *p*roof combinators that allow the user to link parsers to printers. The second type of failure, non-termination, should ideally be avoided altogether: we aim for *total parsing combinators*, parsers that come with a termination guarantee. It will be interesting to see how this goal can be achieved in the spirit of combinator libraries, hiding the nitty-gritty details from the user. Now, without further ado, let's see the core library in action.

The complete Agda code can be found in the accompanying material. Appendix 1 lists some basic types and functions.

2 Sneak preview

To illustrate the trinity of *p*rinter, *p*arser, and *p*roof combinators, we use the type of bushes, defined below, as a running example. (A similar datatype is required later for Huffman encoding, so our efforts are not wasted.)

```
data Bush : Set where
  Leaf : Bush
  Fork : Bush → Bush → Bush
```

For each type A whose elements we wish to serialise, we define a module called *Codec-A*.

```
module Codec-Bush where
```

Printing combinators. A bit string or code is generated using ϵ, $\_\cdot\_$, `0`, and `I`. The encoder for bushes employs all four combinators.

https://doi.org/10.1017/S095679682200017X Published online by Cambridge University Press

```
encode : Bush → Code
encode (Leaf)     = 0 · ϵ
encode (Fork t u) = I · encode t · encode u · ϵ
```

An encoder for elements of type A is typically defined by *structural* recursion on A, *encode* is no exception. Each constructor is encoded by a suitable number of bits followed by the encodings of its arguments, if any. For the example at hand, a single bit suffices. In general, we require $\lceil \log_2 n \rceil$ bits if the datatype comprises n constructors. (Of course, we may also apply Huffman's idea to the encoding of constructors using variable-length instead of fixed-length bit codes.) Concatenation _·_ is associative with ϵ as its neutral element, so the trailing ϵs seem a bit odd—odd, but convenient as we shall see shortly.

Parsing combinators. Decoders can be most conveniently defined using ***do***-notation as the type of decoders has the structure of a monad.

```
{-# TERMINATING #-}
decode′ = 0: return Leaf
          I: do t ← decode′; u ← decode′; return (Fork t u)
```

The decoder performs a case analysis on the first bit of the implicit input bit string, and then, it decodes the arguments, if any. In the best, or, worst tradition of Haskell, *decode*′ is defined recursively. Unfortunately, Agda is not able to establish its termination. All is safe, however: the case analysis `0:` … `I:` … consumes a bit, so both recursive calls process strictly smaller bit strings. The pragma preceding the definition communicates our findings to Agda.

We have come across a well-known problem in parsing: a recursive-descent parser may fail to terminate if the underlying grammar is left-recursive. Consequently, when decoders are defined recursively, we need to make sure that each recursive call is guarded by at least one case combinator. Are we happy to accept the potential threat of non-termination? No! We should be able to do better than that. And indeed, the parser library features a grown-up version of case analysis that allows for safe recursive calls.

```
decode : Decoder Bush
decode = 0-rec: (λ bush → return Leaf)
         I-rec: (λ bush → do t ← bush; u ← bush; return (Fork t u))
```

The combinator 0-*rec*:_I-*rec*:_ can be seen as a recursion principle for bit strings; it passes the ability to issue recursive calls to each branch. All is well now: Agda happily accepts the definition; *decode* terminates on all inputs.

Proof combinators. We have created two independent artefacts: an encoder and a decoder. It remains to relate the two: we require that decoding an encoded value gives back the original value. To capture this property, we make use of a ternary relation, relating bit strings, values, and decoders: $s \langle\ a\ \rangle p$ holds iff p "applied" to s returns a. The correctness criterion then reads

$$\forall\ (a : A) \to encode\ a \langle\ a\ \rangle\ decode. \tag{2.1}$$

https://doi.org/10.1017/S095679682200017X Published online by Cambridge University Press

The library offers six proof combinators for discharging the proof obligation, presented as proof rules below. (Each rule *is* actually the type of the proof combinator—"propositions as types" at work.)

$$\frac{}{\epsilon \ \langle\ a\ \rangle\ return\ a}\,\epsilon\text{-}axiom \qquad \frac{s\ \langle\ a\ \rangle\ p \quad t\ \langle\ b\ \rangle\ q\ a}{s \cdot t\ \langle\ b\ \rangle\ p\ \gg\!=\ q}\,\cdot\text{-}rule$$

The axiom relates the unit of the monoid to the "unit" of the monad; the rule relates the multiplication of the monoid to the "multiplication" of the monad.

$$\frac{s\ \langle\ a\ \rangle\ p}{\mathtt{O} \cdot s\ \langle\ a\ \rangle\ (\mathtt{O}\colon p\ \mathtt{I}\colon q)}\,\mathtt{O}\text{-}rule \qquad \frac{t\ \langle\ a\ \rangle\ q}{\mathtt{I} \cdot t\ \langle\ a\ \rangle\ (\mathtt{O}\colon p\ \mathtt{I}\colon q)}\,\mathtt{I}\text{-}rule$$

The case parser consumes a bit and then acts as one of the branches. The recursive variant additionally passes "itself" to the branch.

$$\frac{s\ \langle\ a\ \rangle\ P\ (\mathtt{O}\text{-}rec\colon P\ \mathtt{I}\text{-}rec\colon Q)}{\mathtt{O} \cdot s\ \langle\ a\ \rangle\ (\mathtt{O}\text{-}rec\colon P\ \mathtt{I}\text{-}rec\colon Q)}\,\mathtt{O}\text{-}Rule \qquad \frac{t\ \langle\ a\ \rangle\ Q\ (\mathtt{O}\text{-}rec\colon P\ \mathtt{I}\text{-}rec\colon Q)}{\mathtt{I} \cdot s\ \langle\ a\ \rangle\ (\mathtt{O}\text{-}rec\colon P\ \mathtt{I}\text{-}rec\colon Q)}\,\mathtt{I}\text{-}Rule$$

The proof of correctness then typically proceeds by structural recursion on the type of the to-be-encoded values, here the type of bushes.

```
correct′ (Leaf)     = O-Rule ϵ-axiom
correct′ (Fork t u) = I-Rule (·-rule (correct′ t) (·-rule (correct′ u) ϵ-axiom))
```

Two remarks are in order.

The definition of *encode* features trailing ϵs so that all three artefacts, *p*rinters, *p*arsers, and *p*roofs, exhibit *exactly* the same structure. Any deviation incurs additional proof effort as monoid and monad laws do not hold on the nose, for example, $s \cdot \epsilon$ is not definitionally equal to s, only propositionally.

Each proof combinator takes a couple of implicit arguments—the variables appearing in the proof rules. Agda happily infers these arguments from the data provided, with one exception: the *·-rule* needs p and q as *explicit* arguments. Consequently, the proof is actually more verbose.

```
correct : ∀ (t : Bush) → encode t ⟨ t ⟩ decode
correct (Leaf)     = O-Rule ϵ-axiom
correct (Fork t u) =
  I-Rule (·-rule decode (λ t′ → do u′ ← decode; return (Fork t′ u′)) (correct t)
         (·-rule decode (λ u′ →                 return (Fork t  u′)) (correct u)
         (ϵ-axiom)))
```

On the positive side, the additional arguments show very clearly how the parsing process proceeds, creating a *Fork* in two steps.

Codecs. For convenience, the three entities are wrapped up in a record.

```
record Codec (A : Set) : Set where
  field encode : A → Code
        decode : Decoder A
        correct : ∀ (a : A) → encode a ⟨ a ⟩ decode
```

https://doi.org/10.1017/S095679682200017X Published online by Cambridge University Press

```
data Natural : Set where
  zero : Natural
  succ : Natural → Natural
module Codec-Natural where
  encode : Natural → Code
  encode (zero)   = 0 · ϵ
  encode (succ n) = 1 · encode n · ϵ

  decode : Decoder Natural
  decode = 0-rec: (λ natural → return zero)
           1-rec: (λ natural → do n ← natural; return (succ n))

  correct : ∀ (n : Natural) → encode n ⟨ n ⟩ decode
  correct (zero)   = 0-Rule ϵ-axiom
  correct (succ n) = 1-Rule (·-rule decode (λ n′ → return (succ n′)) (correct n)
                            (ϵ-axiom))

Natural : Codec Natural
Natural = record {Codec-Natural}
```

Fig. 1. A codec for natural numbers.

For each serialisable type $A : Set$ we define a codec $\mathbb{A} : Codec\ A$. You may want to view *Codec* as a property of a type, or as a type class and *Codec A* as an instance dictionary.

```
Bush : Codec Bush
Bush = record {Codec-Bush}
```

The definition makes use of a nifty Agda feature: the syntax *record* $\{M\} : R$ constructs a record of type R, using suitable components from the module M.

For reference, Figure 1 shows a codec for a second example, the naturals. This completes the description of the interface. Next, we turn to the implementation, starting with parsing combinators as their design dictates everything else.

3 Parsing combinators

The goal is clear: our parsing combinators should come with a termination guarantee as Agda is not able to establish termination of recursively defined decoders. To understand the cure, let us delve a bit deeper into the problem.

First, we introduce the type of bit strings. A bit string is either empty, written [], or a binary digit, `0` or `1`, followed by a bit string.

```
data Bits : Set where
  []    : Bits
  0 1 : Bits → Bits
```

The problem. A decoder is a deterministic parser, combining state, bit strings of type *Bits*, and partiality, the *Maybe* monad (see Appendix 1). Not using any special combinators, a decoder for bushes can be implemented as follows.

https://doi.org/10.1017/S095679682200017X Published online by Cambridge University Press

```
{-# TERMINATING #-}
decode : Bits → Maybe (Bush × Bits)
decode []       = undefined
decode (0 bs)   = return (Leaf , bs)
decode (1 bs)   = do (t , bs′)  ← decode bs
                     (u , bs″) ← decode bs′
                     return (Fork t u , bs″)
```

Observe how the state, the bit string, is threaded through the program. Putting the termination spectacles on, the first recursive call is benign as *bs* is an immediate substring of 1 *bs*. The second recursive call is, however, problematic as *decode* is applied to the string *returned* from the first call. Agda has no clue that bs' is actually smaller than *bs*.

The recursion pattern is an instance of *course-of-values recursion* or *strong recursion* where a function recurses on all smaller values, for example, where $f\,(n+1)$ is computed from $f\,(0), \ldots, f\,(n)$. Perhaps a short detour is worthwhile to remind ourselves of this induction and recursion principle.

Detour: strong versus structural recursion. Say, $P : \mathbb{N} \to \mathit{Set}$ is a property of the naturals, established by strong induction. Strong induction can be reduced to structural induction by proving a stronger property: $Q\,m = P\,0 \wedge \cdots \wedge P\,m$, or, avoiding ellipsis $Q\,m = \forall\, n \to m \geqslant n \to P\,n$.

Say, $f : \mathbb{N} \to A$ is a function from the naturals, defined by strong recursion. Strong recursion can be reduced to structural recursion by defining a stronger function: $g\,m : \forall\, n \to m \geqslant n \to A$. Consider as an example,

```
f : (n : ℕ) → ℕ
f (zero)   = 1
f (succ n) = sf n where
  sf : (i : ℕ) → ℕ
  sf (zero)   = f (zero)
  sf (succ i) = f (succ i) + sf i
```

This is the simplest, non-trivial example I could think of: $f\,(n+1)$ is given by the sum of all smaller values: $f\,(0) + \cdots + f\,(n)$. (Do you see what f computes?) Its stronger counterpart is given by

```
g : (m : ℕ) → ∀ n → m ⩾ n → ℕ
g m       (zero)   (zero)      = 1
g (succ m) (succ n) (succ m⩾n) = sg n m⩾n where
  sg : (i : ℕ) → m ⩾ i → ℕ
  sg (zero)   below = g m (zero)   below
  sg (succ i) below = g m (succ i) below + sg i (⩾transitive below succ-n⩾n)
```

If we remove the first and the third argument of g and the second argument of the helper function *sg*, we get back the original definition: $g\,m\,n\,m{\geqslant}n \equiv f\,n$ for all natural numbers m and n with $m{\geqslant}n$: $m \geqslant n$. In particular, f can be reduced to its counterpart: $f\,n = g\,n\,n \geqslant \mathit{reflexive}$. Observe that g is defined by *structural* recursion on the first argument; the helper function maintains the invariant that the "actual" argument i is at most m.

https://doi.org/10.1017/S095679682200017X Published online by Cambridge University Press

The solution. Returning to our application, we would like to define decoders by strong induction over the length of bit strings. As a preparatory step, we replace the original definition of *Bits* by an *indexed type* that records the length.

```
data Bits : ℕ → Set where
  []  :              Bits (zero)
  0 1 : Bits n → Bits (succ n)
```

Compared to the number-theoretic example above, there is one further complication: we need to relate the length of the residual bit string *returned* by a parser to the length of its input. Exact numbers are not called for; it suffices to know that the length of the output is at most the length of the input. To this end, we introduce $Bits\preccurlyeq n$, the type of bit strings of length below a given upper bound n.

```
record Bits≼ (n : ℕ) : Set where
  constructor _such-that_
  field {size} : ℕ
        bits   : Bits size
        below  : n ≽ size
```

The record features three fields, the first of which is hidden, indicated by curly braces. If the decoder transmogrifies, say, $bs : Bits\ i$ to $bs' : Bits\ j$ with $i\succcurlyeq j : i \succ j$, then the residual output of type $Bits\preccurlyeq i$ is given by bs' *such-that* $i\succcurlyeq j$.

Finally, we have all the gadgets in place, to define a decoder that is approved by Agda's termination checker. As to be expected, its type is more involved. In a sense, *decode* says: I can decode a bit string of size i, but I am happy to accept any bit string of smaller size; I promise to return a bush and a string, the length of which is at most the size of the string given to me.

```
decode : (i : ℕ) → ∀ {j} → i ≽ j → Bits j → Maybe (Bush × Bits≼ j)
decode i        (zero)     []      = undefined
decode (succ i) (succ i≽j) (0 bs)  = return (Leaf , bs such-that succ-n≽n)
decode (succ i) (succ i≽j) (1 bs)  =
  do (t , bs' such-that j≽k)  ← decode i i≽j bs
     (u , bs'' such-that k≽m) ← decode i (≽transitive i≽j j≽k) bs'
     return (Fork t u , bs'' such-that ≽transitive succ-n≽n (≽transitive j≽k k≽m))
```

Voilà, *decode* is defined by *structural* recursion on the natural number i. To reduce clutter, the actual size of the bit string is passed implicitly. In the recursive case, we have $bs : Bits\ j$, $bs' : Bits\ k$, and $bs'' : Bits\ m$ with $j \succ k$ and $k \succcurlyeq m$. All that remains to be done is to hide the plumbing of state and proofs.

Hiding the plumbing. A decoder is a function from bit strings to an optional pair of things and bit strings, promising not to increase the length of the string.

```
IDecoder : Set → (ℕ → Set)
IDecoder A i = ∀ {j} → i ≽ j → Bits j → Maybe (A × Bits≼ j)
```

Both *IDecoder A* and $Bits\preccurlyeq$ are indexed by naturals numbers, and they are functions of type $\mathbb{N} \to Set$. But there is more to them, they are actually functors: *IDecoder A* is a

https://doi.org/10.1017/S095679682200017X Published online by Cambridge University Press

covariant functor from the preorder $(\mathbb{N}, \succcurlyeq)$, viewed as a category, to the category of sets and total functions. Its action on arrows is defined

```
lower : (i ≽ j) → (IDecoder A i → IDecoder A j)
lower i≽j p = λ j≽k bs → p (≽transitive i≽j j≽k) bs
```

The function *lower* makes precise the idea that a decoder can be applied to any shorter bit string. By contrast, $Bits_{\preccurlyeq}$ is a contravariant functor: we may relax the upper bound.

```
raise : (i ≽ j) → (Bits≼ j → Bits≼ i)
raise i≽j (bs such-that j≽k) = bs such-that ≽transitive i≽j j≽k
```

Turning to the implementation of the monad operations, the definition of *return* is standard, except for the proof that the length is unchanged.

```
return : A → IDecoder A i
return a = λ i≽j bs → Maybe.return (a , bs such-that ≽reflexive)
```

The implementation of monadic bind is more interesting.

```
relax : i ≽ j → Maybe (A × Bits≼ j) → Maybe (A × Bits≼ i)
relax i≽j = Maybe.map (λ (a , bs) → (a , raise i≽j bs))
_>>=_ : IDecoder A i → (A → IDecoder B i) → IDecoder B i
p >>= q = λ i≽j bs →
  p i≽j bs  Maybe.>>=  λ (a , bs′ such-that j≽k) →
    relax j≽k (lower i≽j (q a) j≽k bs′)
```

The value a returned by p is passed to the continuation q. Since we wish to apply $q\ a$ to the residual input bs', we need to lower its index. This move has a knock-on effect: the upper bound of q's result is consequently too low and needs to be relaxed in a final step.

The case combinator dispatches on the first bit of the input bit string. It fails if the input is exhausted. This is *the* single source of undefinedness—the correctness proofs guarantee that this failure never happens for "legal" inputs.

```
0:_I:_ : (p q : IDecoder A i) → IDecoder A i
(0: p I: q) (zero)     []     = undefined
(0: p I: q) (succ i≽j) (0 bs) = relax succ-n≽n ((lower succ-n≽n p) i≽j bs)
(0: p I: q) (succ i≽j) (1 bs) = relax succ-n≽n ((lower succ-n≽n q) i≽j bs)
```

Again, we first lower the index and then adjust the upper bound of the result. The proof $succ\text{-}n\succcurlyeq n : succ\ n \succcurlyeq n$ records the fact that *bs* is one element shorter compared to `0` *bs* or `1` *bs*.

It is time to bring in the harvest! The implementation of recursive case analysis merits careful study.

```
0-rec:_I-rec:_ : (P Q : ∀ {i} → IDecoder A i → IDecoder A i)
                 → (∀ {i} → IDecoder A i)
0-rec: P I-rec: Q = recurse _ where
  recurse : ∀ i → IDecoder _ i
  recurse i      (zero)     []     = undefined
```

https://doi.org/10.1017/S095679682200017X Published online by Cambridge University Press

```
recurse (succ i) (succ i≥j) (0 bs) = relax succ-n≥n (P (recurse i) i≥j bs)
recurse (succ i) (succ i≥j) (1 bs) = relax succ-n≥n (Q (recurse i) i≥j bs)
```

The recursor is defined by structural induction on its first argument. The entire set-up—the use of indexed types, keeping track of bounds—serves the sole purpose of enabling this definition. Observe that the type of branches involves a local quantifier: the branches work for any index, in particular, they happily accept the recursive call at index $i - 1$.

Ultimately, the four combinators enable the user to construct decoders that work for any index.

```
Decoder : Set → Set
Decoder A = ∀ {i} → IDecoder A i
```

4 Pretty-printing combinators

For reasons of efficiency, encoders use John Hughes' representation of lists (Hughes, 1986), specialised to bit strings.

```
Code : Set
Code = ∀ {i} → Bits i → Bits≥ i
```

Encoders are dual to decoders. While decoders promise not to increase the size of the bit string, encoders guarantee not to decrease its size. To capture this invariant, we need the dual of *Bits*≼, bit strings of size *above* a given bound:

```
record Bits≥ (n : ℕ) : Set where
  constructor _such-that_
  field {size} : ℕ
        bits   : Bits size
        above  : n ≼ size
```

Composition of functions *and* proofs then serves as concatenation with the identity as its neutral element.

```
ϵ : Code
ϵ = λ bs → bs such-that ≼-reflexive
_·_ : Code → Code → Code
f · g = λ bs → let bs′ such-that i≼j = g bs in
               let bs″ such-that j≼k = f bs′ in
               bs″ such-that ≼-transitive i≼j j≼k
```

The combinators `0` and `I` increase the size by one.

```
0 I : Code
0 = λ bs → 0 bs such-that n≼succ-n
I = λ bs → 1 bs such-that n≼succ-n
```

As an amusing aside, the proof that *Code* is a monoid makes use of the fact that $(\mathbb{N}, \preccurlyeq)$ is a category: ≼-*transitive* is associative with ≼-*reflexive* as its neutral element.

https://doi.org/10.1017/S095679682200017X Published online by Cambridge University Press

5 Proof combinators

Decoders are partial functions. Given an arbitrary bit string, a decoder might fail, unexpectedly exhausting the input. The correctness criterion (2.1) guarantees that this does not happen if a decoder is applied to an encoding. This guarantee is, in a sense, the sole purpose of the whole exercise. The ternary relation _⟨_⟩_ underlying (2.1) relates a code, an element, and a decoder.

```
_⟨_⟩_ : Code → A → Decoder A → Set
code ⟨ a ⟩ decode =
  ∀ k → (bs : Bits k) → ∀ i → (i≻j : i ≻ size (code bs)) →
    decode i≻j (bits (code bs)) ≡ Just (a , bs such-that above (code bs))
```

The function *code* prepends some bits to *bs*; the decoder removes this prefix returning the value *a* and *bs*. This relation must hold for any upper bound *i*.

The implementation of the proof combinators is fairly straightforward. The code is, however, not too instructive, so we content ourselves with two illustrative examples. (The definition of the remaining combinators can be found in the accompanying material.)

The axiom *ϵ-axiom* holds definitionally: Agda can show its correctness solely by unfolding definitions, witnessed by the use of *≡-reflexive* : $a \equiv a$.

```
ϵ-axiom : ϵ ⟨ a ⟩ return a
ϵ-axiom i bs j j≻k = ≡-reflexive
```

To establish the `0`-*rule*,

```
0-rule : (s ⟨ a ⟩ p) → (0 · s ⟨ a ⟩ (0: p I: q))
0-rule premise i bs (succ j) (succ j≻k) =
  ≡-congruent (relax succ-n≻n) (premise i bs (succ j) (≻transitive succ-n≻n j≻k))
```

we apply the premise to the argument of *relax succ-n≻n*.

Now that the core library is in place, we can finally deal with the motivating example, Huffman Encoding. Its implementation turns out to be a nice exercise in datatype-generic programming.

6 Application: Huffman encoding

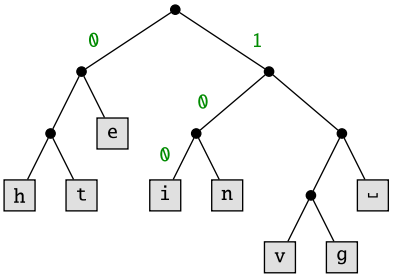
Before diving into the mechanics of Huffman compression, let us state the guiding principles of the design: (1) we would like to seamlessly combine statistical compression with standard serialisers; (2) we aim to avoid irrelevant detail, such as: Is the code unambiguous? Does the alphabet contain at least two characters? Huffman *encoding* typically relies on a dictionary, mapping characters to codes; Huffman *decoding* makes use of a code tree. Does the dictionary contain a code for each to-be-encoded character? Is the code tree actually related to the dictionary? And so forth, and so forth . . . In other words, we are aiming for a clean, lean interface.

We proceed in three steps, defining a codec for a single character, for a sequence of characters, and for Huffman code trees.

https://doi.org/10.1017/S095679682200017X Published online by Cambridge University Press

6.1 Encoding and decoding characters

The central property of variable-length encodings is that no code is a prefix of any other code. This property is guaranteed by construction if a code tree is used instead of a code table. The code tree corresponding to Table (1.2) is depicted below.



The character `'i'`, for example, is encoded by 1 (0 (0 [])). So, as a first requirement, we assume that a code tree is given to us. A code tree is either a singleton, written ⌊ *c* ⌋, or a binary node *t* ⅄ *u*, where *t* and *u* are code trees (the bushes of Section 2 are code trees without codes).

```
data Tree (Alphabet : Set) : Set where
  ⌊_⌋  : Alphabet → Tree Alphabet
  _⅄_ : Tree Alphabet → Tree Alphabet → Tree Alphabet
```

The code tree above is represented by

```
code = ((⌊ 'h' ⌋ ⅄ ⌊ 't' ⌋) ⅄ ⌊ 'e' ⌋)
         ⅄ ((⌊ 'i' ⌋ ⅄ ⌊ 'n' ⌋) ⅄ ((⌊ 'v' ⌋ ⅄ ⌊ 'g' ⌋) ⅄ ⌊ '␣' ⌋))
```

Using this encoding, we can compress the text (1.1), but not `genius`, as neither `u` nor `s` have an encoding. This motivates the second requirement: we need evidence that the to-be-compressed character is contained in the code tree.

```
data _∈_ (c : Alphabet) : Tree Alphabet → Set where
  here  : c ∈ ⌊ c ⌋
  left  : c ∈ t → c ∈ (t ⅄ u)
  right : c ∈ u → c ∈ (t ⅄ u)
```

The evidence $c \in t$ can be seen as a path into the tree. For example,

```
i-am-in :  'i' ∈ code
i-am-in = right (left (left here))
```

certifies that `'i'` is in the code tree *code*: starting at the root, it can be found going to the right and then twice to the left.

Perhaps surprisingly, no further requirements are necessary: our Huffman compressor takes a pair such as (`'i'` , *i-am-in*), consisting of a single character and a proof that this character is contained in the code tree. Observe that the *type* of the second component depends on the *value* of the first component (in the "propositions as types" paradigm a proposition such as $c \in t$ is a type). In other words, the data are a *dependent pair*, an element of a Σ type (*Alphabet* is an implicit type argument, indicated by curly braces).

https://doi.org/10.1017/S095679682200017X Published online by Cambridge University Press

```
In : {Alphabet : Set} → Tree Alphabet → Set
In {Alphabet} t = Σ Alphabet (λ c → c ∈ t)
```

Elements of the type *In t* can be seen as *self-certifying data*: data that can be readily serialised.

```
module Codec-In where
  encode : ∀ {t : Tree Alphabet} → In t → Code
  encode (c , here  ) = ϵ
  encode (c , left  p) = 0 · encode (c , p) · ϵ
  encode (c , right p) = 1 · encode (c , p) · ϵ
```

The encoder turns the evidence, the path into the code tree, into a bit string, ignoring both the code tree and the character!

The decoder has to resurrect the path. For this to work out, the code tree is required.

```
  decode : ∀ (t : Tree Alphabet) → Decoder (In t)
  decode ⌊ c ⌋ =    return (c , here)
  decode (t ⅄ u) = 0: (do (c , p) ← decode t ; return (c , left  p))
                   1: (do (c , p) ← decode u; return (c , right p))
```

The bits steer the search, the decoder stops if a leaf is reached. Note that we do *not* use the recursive case combinator; *decode* is recursively defined by induction on the structure of the code tree.

The proof of correctness replicates the structure of encoder and decoder.

```
  correct : ∀ {t : Tree Alphabet} → ∀ (c : In t) → encode c ⟨ c ⟩ decode t
  correct (c , here)    = ϵ-axiom
  correct (c , left  p) = 0-rule (·-rule (decode _) (λ (c′ , p′) →
                            return (c′ , left p′)) (correct (c , p)) ϵ-axiom)
  correct (c , right p) = 1-rule (·-rule (decode _) (λ (c′ , p′) →
                            return (c′ , right p′)) (correct (c , p)) ϵ-axiom)
```

For brevity, we henceforth omit the correctness proof—in all the examples, it can be mechanically derived from the encoder.

Provided with a code tree, the Huffman serialiser happily deals with a single self-certified character.

```
𝕀n : ∀ (t : Tree Alphabet) → Codec (In t)
```

Before we tackle the encoding of strings, it is worthwhile recording what we do *not* need to assume. We do not require that the character has a unique code; it may appear multiple times in the code tree. (Perhaps, bit strings are used to exchange secret messages. Using different codes for the same letter may slightly improve security.) Neither do we require that each element of *Alphabet* appears in the code tree. (If the underlying alphabet is large—the current Unicode standard defines 144,697 characters—then a text will rarely include each character.) Finally, the compressor works for arbitrary code trees, including trees that consist of a single leaf. These extreme cases feature a fantastic compression rate as no bits are required. (Usually some care must be exercised to avoid sending the decoder

https://doi.org/10.1017/S095679682200017X Published online by Cambridge University Press

into an infinite loop. The decoder presented in Stump (2016) exercises this care but fails with an error message instead!)

6.2 Encoding and decoding sequences

Let us be slightly more ambitious by not only considering strings, sequences of characters, but sequences of arbitrary encodable values. There are at least two options for defining a suitable container type; we discuss each in turn.

The obvious choice is to use standard lists, given by the definition below.

```
data List (Elem : Set) : Set where
  []  : List Elem
  _::_ : Elem → List Elem → List Elem
```

The type features two data constructors; to distinguish between empty and non-empty lists we need one bit.

```
module Codec-List {A : Set} (𝔸 : Codec A) where
  encode : List A → Code
  encode []       = 0 · ϵ
  encode (a :: as) = 1 · encode 𝔸 a · encode as · ϵ

  decode : Decoder (List A)
  decode = 0-rec: (λ list → return [])
           1-rec: (λ list → do a ← decode 𝔸; as ← list; return (a :: as))
```

The codec for *List* takes a codec for A to a codec for *List* A.

```
𝕃𝕚𝕤𝕥 : {A : Set} → Codec A → Codec (List A)
```

You may recognise the type pattern if you are familiar with Haskell's type classes or with datatype-generic programming.

Unfortunately, if we compose the list encoder with our Huffman encoder, then the compression factor suffers, as the code for each character is effectively prolonged by one bit. By contrast, the standard Huffman compressor, which operates on strings, simply concatenates the character codes; the decompressor keeps decoding until the input is exhausted. Alas, this approach is in conflict with our modularity requirements: such a codec cannot be used as a component of other codecs. (Imagine compressing tree-structured data such as HTML, using the statistical compressor for leaves, paragraphs of text.)

We can save the additional bit per list item if we keep track of the total number of items. In other words, we may want to use a vector, a length-indexed list, instead of a standard list.

```
data Vector (Elem : Set) : Natural → Set where
  ⟨⟩  : Vector Elem zero
  _,_ : Elem → Vector Elem n → Vector Elem (succ n)
```

The constructor names are chosen to reflect that a vector is really a nested pair. Like the type of lists, *Vector* features two data constructors. However, only one of these is available

https://doi.org/10.1017/S095679682200017X Published online by Cambridge University Press

for each given length—there is only one shape of vector. So the good news is that *no bits* are required to encode the constructors.

```
module Codec-Vector {A : Set} (𝔸 : Codec A) where
  encode : {i : Natural} → Vector A i → Code
  encode ⟨⟩       = ϵ
  encode (a , as) = encode 𝔸 a · encode as · ϵ

  decode : (i : Natural) → Decoder (Vector A i)
  decode (zero)   = return ⟨⟩
  decode (succ i) = do a ← decode 𝔸; as ← decode i; return (a , as)
```

The encoder and the decoder are driven by the length argument, which is passed implicitly to *encode* and explicitly to *decode*.

```
𝕍ector : {A : Set} → Codec A → (i : Natural) → Codec (Vector A i)
```

Of course, for our application at hand we need to store the length alongside the vector, another case for dependent pairs:

```
List′ : Set → Set
List′ A = Σ Natural (Vector A)
```

Inspecting the type on the right-hand side, we note that we already have a codec for naturals and vectors, so we are left with defining a codec for Σ-types.

Encoding dependent pairs is not too hard, the main challenge is to get the type signature right. A dependent pair $(a , b) : \Sigma\ A\ B$ consists of a field $a : A$ and a field $b : B\ a$. This dependency is reflected in the types of the codecs:

```
module Codec-Σ {A : Set} (𝔸 : Codec A)
               {B : A → Set} (𝔹 : (a : A) → Codec (B a)) where
  encode : Σ A B → Code
  encode (a , b) = encode 𝔸 a · encode (𝔹 a) b · ϵ

  decode : Decoder (Σ A B)
  decode = do a ← decode 𝔸; b ← decode (𝔹 a); return (a , b)
```

The implementation is identical to a codec for standard pairs, except that the first field is passed to the codec for the second field.

```
𝕊igma : {A : Set} → Codec A →
        {B : A → Set} → ((a : A) → Codec (B a)) →
          Codec (Σ A B)
```

Assembling the various bits and pieces,

```
𝕃ist′ 𝔸 = 𝕊igma ℕatural (𝕍ector 𝔸)
```

is a space-efficient codec for sequences. Well, not quite. A moment's reflection reveals that we have not gained anything: 𝕃ist 𝔸 and 𝕃ist′ 𝔸 produce bit strings of exactly the same length. (This tells you something about the relation between lists, naturals, and vectors.) We have chosen to represent naturals in unary. Consequently, the vector length n

https://doi.org/10.1017/S095679682200017X Published online by Cambridge University Press

is encoded by a bit string of length $n + 1$. That said, a cure is readily at hand: we simply switch to a binary or ternary representation.

We seize the opportunity and define a general codec for representation changers. Given an isomorphism $A \cong B$, we can turn a codec for A into a codec for B. The functions witnessing the isomorphism are called *to* and *fro*.

```
module Codec-Iso (iso : A ≅ B) (𝔸 : Codec A) where
  encode : B → Code
  encode b = encode 𝔸 (fro iso b)

  decode : Decoder B
  decode = do a ← decode 𝔸; return (to iso a)
𝕄𝕒𝕡 : (A ≅ B) → (Codec A → Codec B)
```

We chose to replace unary numbers by ternary[1] numbers, obtaining the final implementation of the encoder for sequences:

```
𝕃𝕚𝕤𝕥′ : {A : Set} → Codec A → Codec (List′ A)
𝕃𝕚𝕤𝕥′ 𝔸 = 𝕊𝕚𝕘𝕞𝕒 (𝕄𝕒𝕡 Ternary≅Natural 𝕋𝕖𝕣𝕟𝕒𝕣𝕪) (𝕍𝕖𝕔𝕥𝕠𝕣 𝔸)
```

Our library is taking shape. s

We can, for instance, confirm that the introductory example (1.1) is compressed to 59 bits—we assume below that *text* : *Vector* (*In code*) 20 is provided from somewhere.

```
_ : size (encode (𝕍𝕖𝕔𝕥𝕠𝕣 (𝕀𝕟 code) 20) text []) ≡ 59
_ = reflexive
_ : size (encode (𝕃𝕚𝕤𝕥′ (𝕀𝕟 code)) (20 , text) []) ≡ 67
_ = reflexive
```

Additional 8 bits are needed to store the length.

6.3 Encoding and decoding code trees

All that remains to be done is to define a codec for Huffman trees. The type of code trees is a standard container type, so this is a routine exercise by now—the codec for bushes, see Section 2, serves nicely as a blueprint.

```
module Codec-Code-Tree {A : Set} (𝔸 : Codec A) where
  encode : Tree A → Code
  encode ⌊ a ⌋   = 0 · encode 𝔸 a · ϵ
  encode (t ⅄ u) = I · encode t · encode u · ϵ

  decode : Decoder (Tree A)
  decode = 0-rec: (λ tree → do a ← decode 𝔸; return ⌊ a ⌋)
           I-rec: (λ tree → do t ← tree; u ← tree; return (t ⅄ u))
𝕋𝕣𝕖𝕖 : {A : Set} → Codec A → Codec (Tree A)
```

[1] Why? Well, the details are actually irrelevant for the application at hand, but since you asked: We deal with variable-width, not fixed-width numbers. Hence, the datatype *Ternary* features four constructors, one for each ternary digit plus an end-of-list constructor, see Appendix 1. Four constructors can be conveniently encoded using two bits, not wasting precious bit space. For example, the ternary representation of 20, the length of (1.1), is `0t 1 3 2`, which occupies $2 \cdot 4 = 8$ bits.

https://doi.org/10.1017/S095679682200017X Published online by Cambridge University Press

A Huffman compressor stores the code tree alongside the encoded text. Since the encoding depends on the tree this is another example of a dependent pair—actually, quite a nice one. (Once you start looking closely, you notice that dependent pairs are everywhere.)

```
Huffman : Set → Set
Huffman Alphabet = Σ (Tree Alphabet) (λ t → List′ (In t))
```

An element of the type is a dependent pair consisting of a code tree and a length-encoded list of self-certifying characters. The corresponding codec is obtained simply by changing the font.

```
Huffman : {Alphabet : Set} → Codec Alphabet → Codec (Huffman Alphabet)
Huffman Alphabet = Sigma (Tree Alphabet) (λ t → List′ (In t))
```

Voilà, again. The compositional approach shows its strengths. In particular, it is easy to change aspects. For example, to compress a list of lists of characters, we simply replace the inner codec by List′ (List′ (Compress *t*)).

7 Conclusion

Overall, this was an enjoyable exercise in interface design.

The six proof rules of Section 2 nicely summarise the interface; the user of the library needs to know little more. The implementation of the recursive case combinator posed the greatest challenge: how to hide the details of the termination proof behind the abstraction barrier of the combinator library? Our solution draws inspiration from Guillaume Allais' impressive library of total parsing combinators (Allais, 2018). Unfortunately, his approach was not immediately applicable as it is based on the fundamental assumption that a successful parse consumes at least one character, too strong an assumption for our application.

The combination of statistical and structural compression was a breeze, once the idea of self-certifying data was in place. The definition of codecs is an instance of datatype-generic programming (Jansson & Jeuring, 2002). It was pleasing to see that the technique carries over effortlessly to dependent types and proofs.

The modular implementation of the Huffman compressor provides some additional insight into the mechanics of Huffman encoding. The usual treatment of sequences is inherently non-modular and imposes an additional constraint: singleton code trees are not admissible. Our modular compressor needs roughly $2 \cdot \log_3 n$ additional bits to store the length n of the encoded text—a small price one is, perhaps, willing to pay for modularity.

Last but not least, Huffman-encoded data make a nice example for dependent pairs, an observation that is perhaps obvious, but which I have not seen spelled out before.

Conflicts of Interest

None.

https://doi.org/10.1017/S095679682200017X Published online by Cambridge University Press

Supplementary materials

For supplementary material for this article, please visit http://doi.org/10.1017/S095679682200017X

References

Allais, G. (2018). agdarsec — Total parser combinators. *Journées Francophones des Langages Applicatifs*. Available at: `http://gallais.github.io/index.html`.

Huffman, D. A. (1952). A method for the construction of minimum-redundancy codes. *Proc. IRE*. **40**(9), 1098–1101.

Hughes, R. J. M. (1986). A novel representation of lists and its application to the function "reverse". *Inf. Process. Lett*. **22**(3), 141–144.

Jansson, P. & Jeuring, J. (2002). Polytypic data conversion programs. *Sci. Comput. Program*. **43**(1), 35–75.

Stump, A. (2016). *Verified Functional Programming in Agda*. Association for Computing Machinery and Morgan & Claypool.

Théry, L. (2004). Formalising Huffman's algorithm. Research report. Università degli Studi dell'Aquila.

1 Appendix

Natural numbers. The standard ordering on the naturals, m succeeds n, is defined

```
data _≽_ : ℕ → ℕ → Set where
  zero : n ≽ zero
  succ : n ≽ m → succ n ≽ succ m
```

The datatype overloads the constructors *zero* and *succ*: the proof *zero* states that *zero* (the natural) is the least element; *succ* captures that *succ* (the successor function) is monotone. It is straightforward to show that $\succcurlyeq$ is reflexive and transitive and that $succ\ n \succcurlyeq n$.

```
≽reflexive  : n ≽ n
≽transitive : n ≽ m → m ≽ k → n ≽ k
succ-n≽n    : succ n ≽ n
```

The inverse relation, m precedes n, is given by $m \preccurlyeq n = n \succcurlyeq m$.

Partial functions. In Agda, a partial function from A to B can be modelled by a total function of type $A \to Maybe\ B$.

```
data Maybe (Elem : Set) : Set where
  Nothing : Maybe Elem
  Just    : Elem → Maybe Elem

undefined : Maybe A
undefined = Nothing
```

The type constructor *Maybe* forms a monad: *return* is the identity partial function and bind, "$\gg\!=$," denotes postfix application of a partial function.

```
return : A → Maybe A
return a = Just a
```

https://doi.org/10.1017/S095679682200017X Published online by Cambridge University Press

```
_>>=_ : Maybe A → (A → Maybe B) → Maybe B
Nothing >>= k = Nothing
Just a  >>= k = k a
```

It is straightforward to verify the monad laws.

Ternary numbers. For completeness, here is the type of ternary numbers.

```
data Ternary : Set where
  𝟘t : Ternary
  _1 _2 _3 : Ternary → Ternary
Eau-de-Cologne = 𝟘t 1 2 3 3 3 1 1 1
```

https://doi.org/10.1017/S095679682200017X Published online by Cambridge University Press