Verified Reversible Programming for Verified Lossless Compression

James Townsend and Jan-Willem van de Meent
University of Amsterdam

Abstract

Lossless compression implementations typically contain two programs, an encoder and a decoder, which are required to be inverse to one another. We observe that a significant class of compression methods, based on asymmetric numeral systems (ANS), have shared structure between the encoder and decoder—the decoder program is the 'reverse' of the encoder program—allowing both to be simultaneously specified by a single, reversible function. To exploit this, we have implemented a small reversible language, embedded in Agda, which we call 'Flipper'. Agda supports formal verification of program properties, and the compiler for our reversible language (which is implemented as an Agda macro), produces not just an encoder/decoder pair of functions but also a proof that they are inverse to one another. Thus users of the language get formal verification 'for free'. We give a small example use-case of Flipper in this paper, and plan to publish a full compression implementation soon.

It has been known since the work of Claude Shannon in the late 1940s that there is a deep connection between probabilistic modelling and lossless data compression. In [12], Shannon showed that the best achievable compressed size for random data is equal to the expected negative log-probability, a quantity which he called the 'entropy'. Practical approaches to lossless compression work by assuming a model P over input data x, and compressing x to a size close to the negative log-probability $-\log_2 P(x)$, using a coding method such as Huffman coding [4], arithmetic coding [9], or asymmetric numeral systems (ANS) [2]. Recently, a number of papers have shown that modern deep generative models can be used to achieve state-of-the-art compression rates [14, 7, 15, 3, 1, 11, 17, 6]. These recent works all use ANS, a method which, due to its simplicity and superior performance, has also become ubiquitous in production compression systems.[1]

Our work on this paper began with an observation: in ANS-based methods, the encoder and decoder programs appear to have the same structure. Indeed, we realised that it made sense to view them as different interpretations of *the same* program, one 'doing' the program and the other 'undoing' it. Following this idea to its logical conclusion, we have implemented a small, *reversible* domain specific language, called Flipper, embedded in Agda [10], via an Agda macro. The potential advantages of using Flipper are significant—the amount of user code is reduced, and accidental violation of the inverse property (a common cause of bugs in lossless compression) is no longer possible.

Flipper programs inhabit an Agda record type:[2]

```
record _↔_ (A B : Set) : Set where
  field
```

[1] A list is maintained at `encode.su/threads/2078-List-of-Asymmetric-Numeral-Systems-implementations`, which includes various ANS programs with state-of-the-art performance.

[2] We would have preferred the names 'do' and 'undo' to 'apply' and 'unapply', but unfortunately 'do' is a reserved keyword in Agda.

$$
\begin{aligned}
&\mathsf{apply} : A \to B \\
&\mathsf{unapply} : B \to A \\
&\mathsf{prfa} : \forall\ a \to \mathsf{unapply}\ (\mathsf{apply}\ a) \equiv a \\
&\mathsf{prfb} : \forall\ b \to \mathsf{apply}\ (\mathsf{unapply}\ b) \equiv b
\end{aligned}
$$

The Flipper macro takes a user implementation of apply in $A \to B$, and compiles it into an inhabitant of $A \leftrightarrow B$, complete with an implementation of unapply and the two necessary correctness proofs.

The compiler only accepts functions expressed in an 'obviously' reversible form, where the reverse interpretation can be seen by literally rotating the source code by 180° (more detail on this below). Despite this restriction on form, we have found Flipper to be a surprisingly effective language for implementing lossless compression, and surprisingly enjoyable to use.

There is a tradition of 'reversible' programming languages going back at least as far as [8], and we took particular inspiration from the pure functional reversible languages rFun [16, 13] and Theseus [5]. We believe that Flipper is the first in this tradition to support dependent types, and as far as we know the first to be formally verified. We also think that embedding the reversible language in a well established host language, with excellent editor integration, makes writing software in Flipper significantly easier than existing reversible languages. The power of these features is demonstrated by our implementation of ANS-based compression in Flipper, which to our knowledge is the most sophisticated compression method to have been implemented in a reversible language to date.[3]

1 The Flipper language

The Flipper compiler accepts an $\mathsf{apply} : A \to B$, expressed as a pattern-matching lambda term with a special property: rotating the term by 180°, and "re-righting" any upside down symbols, results in a valid term in $B \to A$, which we can use as our unapply.

As a first example, consider the 'flippable' function which swaps the elements of a pair:

$$
\begin{aligned}
&\mathsf{pair\text{-}swp} : \forall\ \{A\ B\} \to A \times B \leftrightarrow B \times A \\
&\mathsf{pair\text{-}swp} = \mathsf{F}\ \lambda\ \{\ (a\ ,\ b) \to (b\ ,\ a)\ \}
\end{aligned}
$$

Here the Flipper compiler, F, is applied to a lambda term. Flipping that term gives

$$\{\ (v\ `\ q) \leftarrow (q\ `\ v)\ \}\ \curlyvee.$$

Rerighting symbols (including the _,_ constructor/pattern) and moving the λ over to the left, we see that in this case

$$\mathsf{unapply} = \lambda\ \{\ (b\ ,\ a) \to (a\ ,\ b)\ \}.$$

Note that, in order for the rotated unapply to be a valid Agda function, it is necessary that each bound variable in a flippable must be used exactly once. To emphasize the view of apply and unapply as different interpretations (or orientations) of the same program, from here on we will use a more concise, symmetrical syntax for flippable definitions:

$$
\begin{aligned}
&\mathsf{pair\text{-}swp'} : A \times B \leftrightarrow B \times A \\
&\mathsf{pair\text{-}swp'} = \mathsf{F}\ \{\ (a\ ,\ b) \leftrightarrow (b\ ,\ a)\ \}
\end{aligned}
$$

[3] We plan to publicly release Flipper and the ANS compression implementation soon.

$$
\begin{array}{lr}
x & \text{variables} \\
c & \text{Agda constructors} \\
T & \text{Agda terms} \\
p \ ::= x \mid (c\ [\ p\ ]) & \text{patterns} \\
f \ ::= \mathsf{F}\ \{\ bs\ \} \mid T & \text{flippables} \\
bs ::= b \mid b\ ;\ bs & \text{branches} \\
b \ ::= p \leftrightarrow B & \\
B ::= p \mid p_1\ \langle\ f\ \rangle\ p_2 \leftrightarrow B &
\end{array}
$$

Figure 1: Grammar of the Flipper language. The notation $[\ p\ ]$ stands for a list of zero or more patterns. Note that *infix* Agda constructors, such as `_,_`, are also allowed in patterns.

New flippables can be built from existing ones: the syntax $a\ \langle\ f\ \rangle\ b \leftrightarrow T$ is interpreted in apply as "apply the flippable f to the variable a and bind the result to b in T", equivalent to the expression let b = apply f a in T. We can, for example, compose two flippables as follows:

```
_⨟_ : (A ↔ B) → (B ↔ C) → A ↔ C
f ⨟ g = F { a ↔ a ⟨ f ⟩ b ↔
                b ⟨ g ⟩ c ↔ c }
```

To allow for conditional branching, flippable expressions can contain multiple clauses, corresponding to distinct input/output patterns. For example, the following flippable swaps the branches of a sum type:

```
sum-swp : Either A B ↔ Either B A
sum-swp = F { (left  a) ↔ (right a)
            ; (right b) ↔ (left  b)
            }
```

In order for a pattern lambda with multiple clauses to be flippable, the body expressions on the right hand side must partition the output type: each possible constructor must appear *exactly once*.

Finally, between being bound and being used, variable names are considered 'in scope', and can be freely referred to inside $\langle\ \ldots\ \rangle$ (this doesn't count as a 'use'), an example is the way that a is referred to in the flippable uncurry combinator:

```
uncurry : (A → B ↔ C) → A × B ↔ A × C
uncurry f =
  F { (a , b) ↔ b ⟨ f a ⟩ c ↔ (a , c) }
```

2 Bits back coding in Flipper

As a brief example use case, we show how to implement the 'bits-back with ANS' (BB-ANS) method from [14] in Flipper. This is not an implementation of ANS itself, but a method for composing ANS-based codecs in order to encode data using a latent variable model.

We assume a model over data x where we only have access to the joint distribution $P(x, z)$ for some 'latent' variable z, and an approximate posterior $Q(z \mid x)$, but *not* to the marginal

$$P(x) = \int_z P(x, z)\ \mathrm{d}z. \tag{1}$$

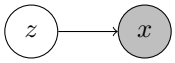

Figure 2: Graphical model with latent variable z and observed variable x.

We define a flippable Encoder type, parameterized by a compressed message type, C, and the type X of data to be compressed:

```
Encoder : Set → Set → Set
Encoder C X = C × X ↔ C
```

The apply function of an Encoder accepts some compressed data in C and a symbol in X and returns a new element of C from which the original compressed data and the symbol can both be recovered. An Encoder can be used to compress a list of elements of X, by (flippably) folding over the list.

The BB-ANS method can be expressed in Flipper as:

```
bb-ans : (P_Z     : Encoder C Z)
         (P_X|Z   : Z → Encoder C X)
         (Q_Z|X   : X → Encoder C Z)
         → Encoder C X
bb-ans P_Z P_X|Z Q_Z|X =
  F { (c , x) ↔
      c       ⟨ flip (Q_Z|X x) ⟩ (c , z) ↔
      (c , x) ⟨ P_X|Z z        ⟩ c       ↔
      (c , z) ⟨ P_Z            ⟩ c       ↔ c }
```

This matches the Python implementation presented in [14, Appendix C], but with only one function required instead of two.

3 Conclusion

We have presented the Flipper language, for which we have implemented a compiler as an Agda macro. Flipper can be used to reduce the amount of code, and to avoid a class of bugs, when implementing lossless compression programs. We hope to release Flipper, as well as the full ANS compression implementation in Flipper, as soon as possible.

Acknowledgements

Thanks to Wouter Swierstra and Heiko Zimmerman for their advice and feedback on drafts of this paper. This publication is part of the project "neural networks for efficient storage and communication of information" (with project number VI.Veni.212.106) of the research programme "NWO-Talentprogramma Veni ENW 2021" which is financed by the Dutch Research Council (NWO).

References

[1] Rianne van den Berg, Alexey A. Gritsenko, Mostafa Dehghani, Casper Kaae Sønderby, and Tim Salimans. IDF++: Analyzing and Improving Integer Discrete Flows for Lossless Compression. In *International Conference on Learning Representations (ICLR)*. 2021.

[2] Jarek Duda. Asymmetric Numeral Systems. 2009. arXiv: `0902.0271 [cs, math]`.

[3] Emiel Hoogeboom, Jorn Peters, Rianne van den Berg, and Max Welling. Integer Discrete Flows and Lossless Compression. In *Advances in Neural Information Processing Systems 32*. 2019, pp. 12134–12144.

[4] David A. Huffman. A Method for the Construction of Minimum-Redundancy Codes. In *Proceedings of the IRE* 40.9 (1952), pp. 1098–1101.

[5] Roshan James and Amr Sabry. Theseus: A High Level Language for Reversible Computing. In *Conference on Reversible Computation (RC)*. Work in progress report. 2014.

[6] Diederik Kingma, Tim Salimans, Ben Poole, and Jonathan Ho. Variational Diffusion Models. In *Advances in Neural Information Processing Systems*. Vol. 34. 2021, pp. 21696–21707.

[7] Friso Kingma, Pieter Abbeel, and Jonathan Ho. Bit-Swap: Recursive Bits-Back Coding for Lossless Compression with Hierarchical Latent Variables. In *Proceedings of the 36th International Conference on Machine Learning*. 2019, pp. 3408–3417.

[8] Christopher Lutz. Janus: A Time-Reversible Language. *Letter to R. Landauer*. 1986.

[9] Alistair Moffat, Radford M. Neal, and Ian H. Witten. Arithmetic Coding Revisited. In *ACM Transactions on Information Systems (TOIS)* 16.3 (1998), pp. 256–294.

[10] Ulf Norell. Towards a Practical Programming Language Based on Dependent Type Theory. Department of Computer Science and Engineering, Chalmers University of Technology, 2007.

[11] Yangjun Ruan, Karen Ullrich, Daniel S. Severo, James Townsend, Ashish Khisti, Arnaud Doucet, Alireza Makhzani, and Chris Maddison. Improving Lossless Compression Rates via Monte Carlo Bits-Back Coding. In *Proceedings of the 38th International Conference on Machine Learning*. 2021, pp. 9136–9147.

[12] Claude. E. Shannon. A Mathematical Theory of Communication. In *The Bell System Technical Journal* 27.3 (1948), pp. 379–423.

[13] Michael K. Thomsen and Holger B. Axelsen. Interpretation and Programming of the Reversible Functional Language RFUN. In *Proceedings of the 27th Symposium on the Implementation and Application of Functional Programming Languages*. IFL '15. New York, NY, USA, 2015, pp. 1–13.

[14] James Townsend, Thomas Bird, and David Barber. Practical Lossless Compression with Latent Variables Using Bits Back Coding. In *International Conference on Learning Representations (ICLR)*. 2019.

[15] James Townsend, Thomas Bird, Julius Kunze, and David Barber. HiLLoC: Lossless Image Compression with Hierarchical Latent Variable Models. In *International Conference on Learning Representations (ICLR)*. 2020.

[16] Tetsuo Yokoyama, Holger B. Axelsen, and Robert Glück. Towards a Reversible Functional Language. In *Reversible Computation*. Lecture Notes in Computer Science. Berlin, Heidelberg, 2012, pp. 14–29.

[17] Mingtian Zhang, Andi Zhang, and Steven McDonagh. On the Out-of-distribution Generalization of Probabilistic Image Modelling. In *Advances in Neural Information Processing Systems*. Vol. 34. 2021, pp. 3811–3823.