IMPLEMENTATION OF THE VITERBI ALGORITHM USING FUNCTIONAL PROGRAMMING LANGUAGES

Tristan Bull
Department of Electrical Engineering & Computer Science
University of Kansas
Lawrence, KS 66045
tbull@ittc.ku.edu

Faculty Advisors:
Erik Perrins and Andy Gill

ABSTRACT

In this paper, we present an implementation of the Viterbi algorithm using the functional programming language Haskell. We begin with a description of the functional implementation of the algorithm. Included are aspects of functional programming that must be considered when implementing the Viterbi algorithm as well as properties of Haskell that can be used to simplify or optimize the algorithm. Finally, we evaluate the performance of the Viterbi algorithm implemented in Haskell.

INTRODUCTION

Forward error correction(FEC) codes allow a receiver to correct errors in a received bit sequence by introducing parity bits into the transmitted signal. FEC codes are commonly used in wireless communications but have not yet been fielded in the telemetry world. However, FEC codes offer benefits to the telemetry world including the ability to provide reliable communication at very low signal-to-noise ratios. Additionally, FEC codes have already taken a role in the migration from serial streaming telemetry (SST) to integrated Network Enhanced Telemetry (iNet). Future communication systems will used trellis-based demodulators that will require an FEC decoder.

This paper focuses on one type of FEC code: convolutional codes (CC). Specifically, we focus on the Viterbi algorithm, a decoding algorithm for convolutional codes [1, Ch 12]. This paper is the first step in a larger project, to implement the CC decoder on field-programmable gate array (FPGA) hardware. Traditionally, a software model is first implemented in an imperative programming language such as MATLAB or C++. This model can later be used for equivalence checking with the hardware implementation. However, in general none of the coding done on the software model is reuseable in the hardware implementation. The hardware implementation is generally written in a hardware description language such as VHDL[1], which is very different syntactically and semantically from an ordinary programming language.

In this paper, we discuss a Viterbi algorithm implementation in the functional programming language, Haskell. Functional programming languages offer many benefits over imperative languages like C or MATLAB. Functions in Haskell are more like mathematical expressions than a list of steps for the computer to iterate through. Additionally, functions can be passed as arguments very easily and neatly. These

[1]VHSIC (Very High Speed Integrated Circuits) hardware description language

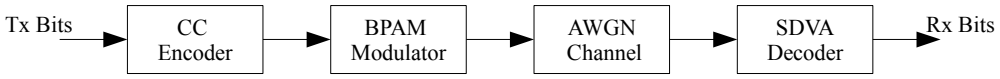

Figure 1: Block diagram of Viterbi algorithm simulation.

properties make Haskell appealing as a hardware description language as well. In fact, a tool called Lava already exists to describe digital circuits in Haskell. Haskell has a library called QuickCheck available that simplifies and automates equivalence checking [2]. So, designing our software model in Haskell and the hardware description in Lava will allow us to easily check the hardware implementation for correctness. Additionally, we will be researching methods of transforming normal Haskell functions into Lava.

In summary, our contributions are as follows:

- We implement the Viterbi algorithm in the Haskell programming language.
- We investigate the advantages and caveats of implementing the Viterbi algorithm in Haskell.
- We compare bit error rate (BER) performance of our implementation with a known implementation to give evidence of correctness.

VITERBI ALGORITHM

The Viterbi algorithm is a decoding algorithm for convolutional codes [1, Ch 12]. A simple block diagram of the system used to simulate the algorithm is presented in Figure 1. The simulation program first generates a block of 4096 bits and encodes the bits using a rate 1/2 (5,7) convolutional code (CC) encoder [1, Ch 12]. The encoded bits are then modulated using binary pulse-amplitude modulation (PAM), essentially just turning them into antipodal bits. The modulated data are then sent through an additive white Gaussian noise (AWGN) channel. Finally, the noisy data are supplied as input to the soft decision Viterbi algorithm (SDVA) decoder which, by definition, takes antipodal data as input and outputs decoded bits. Further details of the simulation process are included in the PERFORMANCE VALIDATION section.

The Viterbi algorithm is implemented in two steps. First, we generate a trellis from the received bits. An example trellis is shown in Figure 2. At any point in the transmission, there are four possible states. Each dot corresponds to a state. Each edge corresponds to a state transition, or branch. Each branch is assigned a branch metric, which is related to the Euclidean distance between the branch state and the last transmitted symbol. The equation we used to calculate the branch metric is simple:

$$
\begin{aligned}
\mathrm{BM}(rr, bb) &= (r_0 - b_0)^2 + (r_1 - b_1)^2 && (1)\\
&= r_0^2 - 2r_0b_0 + b_0^2 + r_1^2 - 2r_1b_1 + b_1^2 \\
&= \cancel{r_0^2} - 2r_0b_0 + \cancel{b_0^2} + \cancel{r_1^2} - 2r_1b_1 + \cancel{b_1^2} && (2)\\
&\equiv r_0b_0 + r_1b_1. && (3)
\end{aligned}
$$

In (1), rr corresponds to the received symbol while bb corresponds to the branch symbol, which is dependent on the convolutional encoder used. In (2), all scalar terms (including -1) are cancelled out.

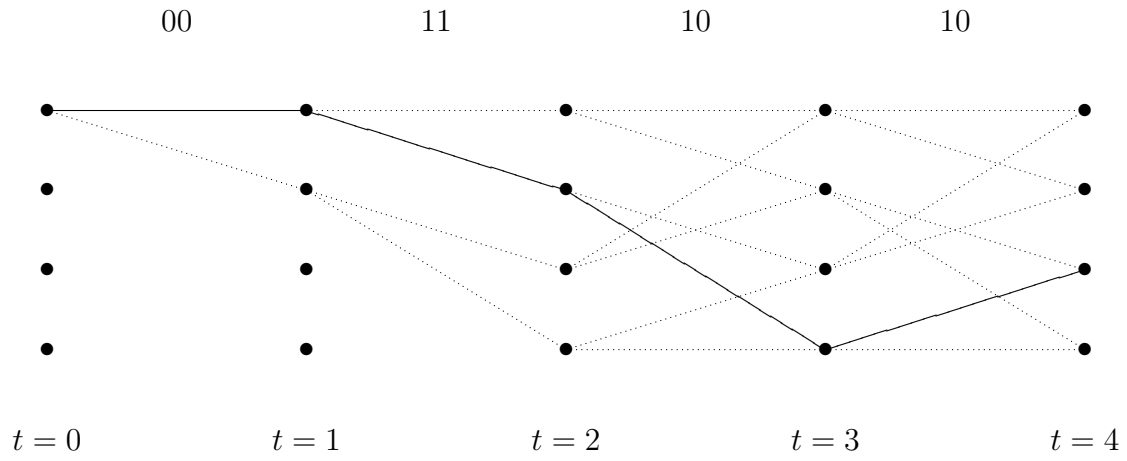

Figure 2: Viterbi Algorithm Trellis

We are able to simplify the calculation because we are not concerned with finding the actual value for Euclidean distance. Basic properties of equality state that $\text{argmin}(a \times x + b, a \times y + b) == \text{argmin}(x, y)$. Assuming that there is no noise, for values of r and b between -1 and 1, the branch metric will be between -2 and 2. If $rr = bb$, then the branch metric will be 2. Likewise, if $r_0 = -b_0$ and $r_1 = -b_1$, the branch metric will be -2. Thus, the minimum Euclidean distance in (1) corresponds with the maximum correlation in (3).

Each state in the trellis contains an ordered pair of Floating point numbers, which represent the path metric (λ) for each branch entering the state:

$$\lambda_{\text{new}} = \lambda_{\text{prev}} + r_0 b_0 + r_1 b_1. \tag{4}$$

The path metric at any state is a cumulative value representing the path of minimum distance to that state. Remember, to minimize distance, we maximize the branch metric. Thus, λ_{prev} is simply the path metric of the previous state, which is equal to the maximum value in the ordered pair at the previous state. The second part of the path metric equation is the branch metric derived in Equation (3).

Once the trellis is generated, the next step is to find the survivor path. The survivor path is the path with minimal Euclidean distance (and maximum path metric). To find the survivor path, we start at the end of the transmission and find the branch with the highest path metric. We then follow that branch to its previous state. At this state we once again pick the path with the highest path metric and follow that branch to it's previous state. This process continues until we reach the starting state. In Figure 2, the solid black line represents the survivor path. The branch symbols for that path are includeded at the top of the figure. The Viterbi algorithm is discussed in much further detail in [1, Ch 12].

HASKELL IMPLEMENTATION

Traditional programming languages such as C or Java are considered imperative languages. Algorithms are expressed in an iterative fashion. Functional programming languages, like Haskell, express algorithms more as mathematical expressions than as a lists of steps [3]. Normal Haskell functions and variables have no side effects. In computer science, a side effect is a change of state or an interaction with the outside world. This means that all functions are deterministic: the output is dependent only on the input. Also, variables do not change. In an imperative language a variable is a label for a block of memory

or the value stored in that block of memory. In Haskell, a variable is a label for some value. This means that Haskell variables do not actually vary at all! Unsurprisingly, functions are a very important part of functional programming. Functions are values just like variables. A function can have another function as input or output. While all of this may seem unnatural at first to the C or MATLAB programmer, it allows for higher levels of abstraction and much more concise code. An example of a Haskell function for calculating the branch metric is presented below:

```
ed :: (Float,Float) -> (Float,Float) -> Float
ed (x1,y1) (x2,y2) = x1*x2 + y1*y2
```

First, the type of the function is given. Function types are listed in order from the first argument to the last argument followed by the type of the output. The Euclidean distance function, `ed`, accepts two-tuples as arguments, each containing two floating point numbers, and returns a floating point number. A tuple is a data structure similar to a list with a few exceptions. A list in Haskell can only contain elements of a single type, while a tuple can contain elements of different types. Also, a list can be of variable length, while the number of elements in a tuple must be known. In this case, each tuple contains a pair of values. The line after the type contains the actual function. We simply multiply the `x` and `y` values from the two ordered pair inputs and sum the results. This function, `ed`, is used to calculate the branch metric discussed in the previous section.

Haskell has a built-in type called `Array`, which can be used to create indexed multi-dimensional arrays. Haskell includes functions to get/update elements of an array. However, for this project we wanted additional functionality. We wanted to be able to use many of the matrix operations such as reshape, cross product, add, and subtract that are available in MATLAB. So we created a matrix module including a new type called `Matrix` and wrote functions that performed these operations on our new data structure. The `newtype Matrix` contains a "boxed up" two-dimensional `Array`:

```
newtype Matrix a = Matrix (Array (Int,Int) a)
```

`newtype` is a keyword in Haskell used to declare a new data type. In this case, we are building a new data type called `Matrix` which can contain elements of any type. The pair, `(Int,Int)` correspond to row and column indices in a two-dimensional array. In our Haskell implementation of the Viterbi algorithm, the trellis is represented as a $4 \times N$ matrix of ordered pairs, where N is the number of symbols in the received data. Each row in the matrix corresponds to one of the four states, and each column corresponds to a symbol. Each element contains the path metric data at that state. An example path metric matrix is presented below:

```
    (Just 2.0,Nothing)     (Just 0.0,Nothing) (Just 0.0,Just (-2.0))    (Just 0.0,Just 2.0)
(Just (-2.0),Nothing)     (Just 4.0,Nothing) (Just 0.0,Just (-2.0))    (Just 0.0,Just 2.0)
    (Nothing,Nothing)  (Just (-2.0),Nothing)    (Just 2.0,Just 0.0) (Just (-2.0),Just 8.0)
    (Nothing,Nothing)  (Just (-2.0),Nothing) (Just 6.0,Just (-4.0))    (Just 2.0,Just 4.0)
```

This matrix was generating using the four symbol sequence described in Figure 2. Haskell contains a built-in type called `Maybe` which is essentially an optional value. A value of type `Maybe Float`, for example, either contains a value of type `Float` (called `Just Float`) or `Nothing`. The `Maybe` type is discussed further in [3]. The `Maybe` type was used in the Viterbi algorithm to denote invalid branches in the first two symbols. The first column in the matrix above corresponds to the states at $t = 1$ in the trellis in Figure 2. The `Nothing` values correspond to states which have zero or one branch entering them.

The `vaTrellis` function is presented below:

```
vaTrellis :: Matrix Float -> Matrix ((Maybe Float),(Maybe Float))
vaTrellis rx = (Matrix a)
    where a = array ((0,0),(3,un-1))
              $[((x,0), (bm1 x 0,Nothing)) | x <- [0,1]] ++
              [((x,0), (Nothing,Nothing)) | x <- [2,3]] ++
              [((x,1), (((bm1 x 1) `mAdd` p1 x 1),Nothing)) | x <- [0..3]] ++
              [((x,y), (((bm1 x y) `mAdd` p1 x y),(bm2 x y) `mAdd` p2 x y))
               | y <- [2..un-1], x <- [0..3]]
          bm1 x y = Just $ ed (list2Tuple (getRow bs (l2r@@(x*2,0)))) (list2Tuple (getRow rxDat y))
          bm2 x y = Just $ ed (list2Tuple (getRow bs (l2r@@(x*2+1,0)))) (list2Tuple (getRow rxDat y))
          p1 x y = max (fst (a!(l2r@@(2*x,0) `div` 2,y-1))) (snd (a!(l2r@@(2*x,0) `div` 2,y-1)))
          p2 x y = max (fst (a!(l2r@@(2*x+1,0) `div` 2,y-1))) (snd (a!(l2r@@(2*x+1,0) `div` 2,y-1)))
          un     = numRows rxDat
          rxDat  = matReshape rx (numRows rx `div` 2) 2
          l2r    = genL2R gcc
          bs     = matMap antip $ genBS gcc
```

This function accepts a list of received symbols in the form of a $1 \times 2N$ matrix of floating point numbers, and outputs a $4 \times N$ `Matrix` of type `(Maybe Float, Maybe Float)` where N is the number of transmitted (2-bit) symbols. Haskell allows the programmer to include a `where` clause containing defintions of local functions and variables. In the code above, the `where` clause includes the majority of the functionality. The variable a is defined as the array inside the matrix that is returned by the function. The first line `array ((0,0),(3,un-1))` includes the array bounds. The lower bounds are `(0,0)` and the upper bounds are `(3,un-1)` where `un` is equal to the number of symbols in the transmission. The next four lines build the contents of the array. The contents of an array are stored in a list where each element is of the form `((idx,idy),v))` where `idx` is the column index, `idy` is the row index, and `v` is the value stored in that element. To build this list, we use a series of Haskell list comprehensions, which are similar to set comprehensions (or set-builder notation) in mathematics. A complete explanation of Haskell list comprehensions is beyond the scope of this paper. We encourage the reader to read the appropriate section in [3] for more information. Suffice it to say that a list comprehension is a syntactic shortcut for building a list from one or more lists. In this case, we are building a list of matrix elements from lists of row and column indices. After `a` is defined, we define a series of helper functions which are used to get the branch metric and path metric data. `bm1` and `bm2` are used to calculate the branch metric for even and odd branches respectively. The type of the `bm` functions is given below:

```
bm :: Int -> Int -> Maybe Float
```

The functions accept an x and y index for a state in the trellis and calculates the branch metric data at that state. The functions `p1` and `p2` determine which path metric (referred to as λ_{prev} in the previous section) to use at a given state. Their type is the same as the `bm` functions. They accept an x and y index for a state and return the maximum value of the previous state. `rxDat` reshapes the input data (a single column matrix) into a two column matrix so that there is one 2-bit symbol on each row. `l2r` generates a list of left-to-right indices that are used to determine a branch's left index, given its right index. `bs` generates antipodal branch state data.

After the trellis is generated, the next step is to determine the survivor path. We wrote a function that accepts a matrix generated by the vaTrellis function and outputs decoded bits. That function is called `getSym`:

```
-- getSym takes a viterbi algorithm matrix (see vaTrellis) and a starting column
-- and generates symbol data by walking back through the matrix.
--getSym :: Matrix (Maybe Float, Maybe Float) -> Int -> [Int]
getSym m y = flatten $ reverse $ getSym' ss m y
    where possSS = getCol m y
          ss = foo possSS
          getSym' x m y
             | y >= 0 = issueBS x (m@@(x,y)) : getSym' (st x (m@@(x,y))) m (y-1)
             | otherwise = []
```

The function `getSym` relies on three helper functions: `foo`, `st`, and `issueBS`. The type of `foo` is given below:

```
foo :: [(Maybe Float, Maybe Float)] -> Int
```

This function accepts a column from the trellis and returns the index of the row containing the highest value. This is used to determine the starting state in the last column of the trellis (the first symbol to be decoded).

The type of `st` is given below:

```
st :: Int -> (Maybe Float,Maybe Float) -> Int
```

`st` takes a current state (an `Int`) and an ordered pair containing path metric data, and outputs the next state. This function is used to move from right to left in the trellis, following the survivor path.

The type of the function `issueBS` is given below:

```
issueBS :: Int -> (Maybe Float, Maybe Float) -> [Int]
```

This function takes a current state and an ordered pair of path metric data, and outputs the branch state. The branch state is the decoded bit(s) and is dependant on the convolutional encoder used. In the `getSym` function, we use the `where` clause to define some functions and variables used in main function definition. The variable `possSS` contains the last column in the trellis. The variable `ss` determines the starting state by running `foo` on `possSS`. Finally, `getSym'` combines all of the functions discussed above to move from right to left in the trellis, generating a list of decoded bits. The type of getSym' is included below:

```
getSym' :: Int -> Matrix (Maybe Float, Maybe Float) -> Int -> [[Int]]
```

The first argument to `getSym'`, x, is the row index of the current state. The next argument is the trellis matrix, and the final argument, y, is the column index of the current state. There are two things that must be considered before the result of `getSym'` is output to the user. First, since getSym' is recursing from the beginning to the end of the data transmission, we must reverse the decoded bit sequestion. Seond, the function returns a list of lists of `Int` because `issueBS` returns the decoded bits for each symbol in the form of a list. Even though the rate 1/2 convolutional code only has one decoded bit per 2-bit symbol, other convolutional encoders may have multiple bits. Thus, we return a list of bits. To solve these two problems we call the built-in function `reverse` to reverse the bit sequence returned by `getSym'` and we call the function `flatten` which turns a list of type `[[Int]]` into a list of type `[Int]`.

The two functions `vaTrellis` and `getSym` make up the majority of our Viterbi algorithm implementation in Haskell. By calling `vaTrellis` on a column matrix of encoded data, and then calling `getSym` on the trellis data structure, we are able to decode a sequence of convolutionally encoded bits.

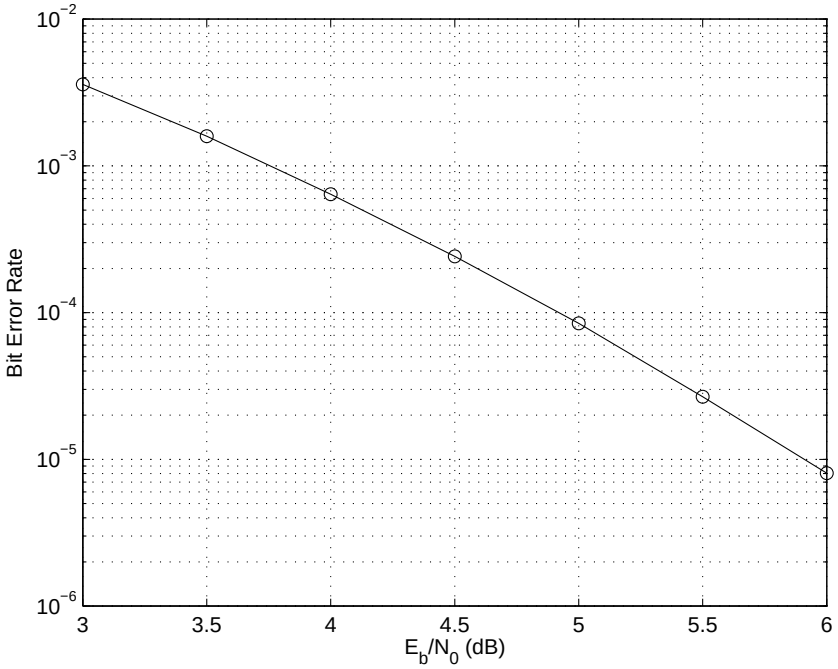

Figure 3: Bit error rate as a function of E_b/N_0 of Rate = 1/2 convolutional code.

PERFORMANCE VALIDATION

We used simulation to measure algorithm correctness. In the simulation, a rate 1/2 (5,7) convolutional code was used with 4096 bit codewords. We ran the simulation using input data with energy per bit to noise power spectral density ratios (E_b/N_0) of 3dB to 6dB in 0.5dB increments. For each E_b/N_0 value, we ran the simulation until at least 25,000 bit errors were recorded. Results are shown in Figure 3. This curve matches the curve presented in [4, Ch 11], giving confidence that our implementation is correct.

CONCLUSIONS AND FUTURE WORK

The final goal of this research project is to implement a transmitter and receiver with error correction coding on a field-programmable gate array (FPGA). We are planning to use Lava for the hardware design. Lava is a Haskell tool for hardware design and verification. Lava circuits can be simulated directly or compiled into VHDL (VHSIC (Very High Speed Integrated Circuits) hardware description language) or EDIF (Electronic Design Interchange Format) code. Unlike traditional hardware description languages such as VHDL, Lava has the ability to describe circuit *layout* as well as behavior. Lava uses Haskell abstractions to provide abstractions in the circuit description. For example, it is difficult to understand how circuits are connected together in VHDL. In Lava, a circuit is represented as a function. Haskell functions can take functions as arguments making it easy to describe how circuits are connected without the need to name intermediate signals [5]. The Lava design is available in [6]. Haskell has the QuickCheck library that allows for easy equivalence checking [2]. Having the Viterbi algorithm implemented correctly as a regular Haskell program will allow for easy equivalence checking with our Lava implementation. Additionally, we will be researching methods of transforming regular Haskell functions (like the algorithm presented in this paper) into Lava circuits.

ACKNOWLEDGEMENT

The authors would like to thank the Test Resource Management Center (TRMC) Test and Evaluation/-Science and Technology (T&E/S&T) Program for their support. This work was funded by the T&E/S&T Program through the U.S. Army Program Executive Office for Simulation, Training and Instrumentation (PEO STRI), contract number W900KK-09-C-0018 for High-Rate High-Speed Forward Error Correction Architectures for Aeronautical Telemetry (HFEC).

REFERENCES

[1] T. K. Moon, *Error Correction Coding: MatheMatical Methods and Algorithms*. New Jersey: Wiley, 2005.

[2] K. Claessen and J. Hughes, "QuickCheck: A lightweight tool for random testing of Haskell programs," in *Proc. of International Conference on Functional Programming (ICFP)*, ACM SIGPLAN, 2000.

[3] S. Peyton Jones, ed., *Haskell 98 Language and Libraries – The Revised Report*. Cambridge, England: Cambridge University Press, 2003.

[4] S. Lin and D. Costello, *Error Control Coding*. New York: Prentice Hall, 2004.

[5] M. S. Satnam Singh, "Designing FPGA circuits in Lava," 1997.

[6] P. Bjesse, K. Claessen, M. Sheeran, and S. Singh, "Lava: Hardware Design in Haskell," in *International Conference on Functional Programming*, pp. 174–184, 1998.