論文 / 著書情報
Article / Book Information

Title	Compact speech decoder based on pure functional programming
Author	Takahiro Shinozaki, Masakazu Sekijima, Shigeki Hagihara, Sadaoki Furui
Journal/Book name	Proc. APSIPA ASC 2011, , ,
Issue date	2011, 10

Compact Speech Decoder Based on Pure Functional Programming

Takahiro Shinozaki* Masakazu Sekijima† Shigeki Hagihara† and Sadaoki Furui†
* Chiba University, Chiba, Japan
E-mail: shinot@chiba-u.jp Tel: +81-43-290-3256
† Tokyo Institute of Technology, Tokyo, Japan

***Abstract*—Current speech recognition systems are implemented using very large and complex programs. This makes it difficult to learn such systems and test a new idea in speech recognition that requires program modification. Therefore, a compact system suitable for educational and prototyping purposes is required. We developed a very compact speech decoder based on pure functional programming, which has been studied in software engineering as a means to describe complex systems in a highly abstracted manner. The decoder is based on weighted finite state transducer and is described using less than 400 lines of codes. The performance of the decoder is demonstrated with large vocabulary continuous speech recognition experiments using a trigram language model and a cross-word triphone hidden Markov model.**

I. Introduction

To test a new statistical model or a new idea in speech recognition, it is often required to modify existing software to learn statistical models and to perform recognition experiments. However, such software is large and complex, and it takes a large amount of time to understand where to modify. This forms a bottleneck for speech processing researchers and even a barrier for researchers in related areas who are interested in speech recognition.

One possible approach to this problem is to prepare software libraries to manipulate probabilistic models used in speech recognition. By developing a program based on the library functions, the required time for coding can be largely reduced. However, it is difficult to design library functions for predicting all possible future extensions. Therefore, it is often the case that a new idea requires modifications of the library functions, and this may require a large amount of time when the library is large and complex.

Another approach is to develop software that supports a generalized probabilistic framework that includes statistical models used in speech recognition systems. Software that implements Bayesian network [1] or weighted finite state transducer (WFST) [2] can be regarded as an example of this approach. Given such software, various probabilistic models can be investigated without modifying the program code as long as they fit in the supported class of models. However, once a new model or a training/decoding algorithm goes beyond the existing framework, modification to an existing code is again required (eg. [3] etc.).

In software engineering, pure functional programming has been studied as a programming paradigm that has high abstraction and modularization ability [4]. In functional programming, a program consists of functions in which a function can be an argument of another function or can be a returned value. In pure functional programming, there is no side effect in terms of function application, unlike procedural programming such as C and C++. Because of this, lazy evaluation is easily adopted as the evaluation scheme [5]. By using lazy evaluation, it is possible to separate a structure of a function's relationship and the actual timing of computation. These properties of pure functional programming provide a mechanism for higher abstraction and modularization in software description. Moreover, it is said to be advantageous in parallel processing in the future since there is freedom in the order of function evaluation.

It is expected that a speech recognition system can be described compactly in a highly abstracted manner by applying pure functional programming. By using that system as a baseline, any modification would be possible with minimum effort. However, pure functional programming is quite different from procedural programming as it does not have variables, and it is not clear how it can be applied to describe a large-vocabulary continuous speech recognition system. Therefore, we empirically investigated the application of pure functional programming to describe a speech recognition system. As the first step, a pure function-based WFST decoder "Husky" is implemented and evaluated.

The organization of this paper is as follows. The background and current status of pure functional programming is briefly described in Section II. Speech recognition based on WFST is briefly reviewed in Section III. A few design points of our developed pure functional decoder is discussed in Section IV. Experimental conditions are described in Section V, and the results are shown in Section VI. Conclusions and future work are given in Section VII.

II. Pure functional programming

In procedural programming, such as C and C++, a program has a state represented by variables, which are updated step by step in the program execution. Depending on the state, the same function with the same arguments can result in different values at different times.

In contrast to procedural programming, there are no variables in pure functional programming and the functions have no side effect. Pure functional programming originated from

the lambda calculus and has a long history, but it has mainly been of interest in academic research. However, with the development of practical languages such as Haskell [6] and Clean, it has recently started to be used in engineering. The concepts that characterize pure functional programming include pure functions, first-class functions, recursion, and lazy evaluation. These concepts are briefly described in the following subsections.

A. Pure functions

Pure functions have no side effect. In pure functional programming, all functions are pure functions. Therefore, a function always returns the same value when the same arguments are given. This contributes to reducing bugs and increasing freedom of the compiler for optimization.

A problem is how to deal with input/output (I/O). It is necessary in practical programming to read data in a file. However, if a "fileRead" function takes a file name as its argument and returns the data in the file, the function is no longer pure. This is because a returned value of the function depends on the contents of the file, which is not an argument of the function. In Haskell, this problem is solved by using monads [7]. A monad is a kind of abstract data type constructor used to represent computations. By using a monad, Haskell separates I/O with other parts of the program and maintains the pureness of the program.

B. First-class functions

Functions can be arguments as well as returned values of other functions in functional programming. An example of such a function in calculus is the derivative operator, which takes a function and returns another function.

C. Recursion

There are no variables in pure functional programming. Therefore, there is no syntax to represent a "for" or "while" loop. Instead, iteration is accomplished via recursion. Recursive functions invoke themselves to repeat an operation. In general, recursion requires a stack to keep track of the recursive function calls, which consumes memory. However, a special case is tail recursion where the recursive call is made at the end of the function. In this case, the stack is not necessary and memory can be saved.

D. Lazy evaluation

When arguments of a function are evaluated before the function is called, it is called strict evaluation. Conversely, when evaluation of arguments is delayed until the last possible moment, it is called lazy evaluation. Lazy evaluation contributes to improving modularity by giving a means to separate connections of functions and actual timing of the computation. With lazy evaluation, it is possible to define a large or even infinite data structure and pass it to other functions and still achieve efficient execution. The match of lazy evaluation with variable substitution is unfortunately not good, and it is specifically used in pure functional programming. In Haskell, the default is lazy evaluation but a programmer can also choose strict evaluation.

III. Speech recognition based on WFST

A weighted finite state transducer (WFST) is a finite state machine with input and output tapes and transition weights. It has successfully been applied to speech recognition [2]. Many of the statistical model components in speech recognition systems, such as Hidden Markov Model (HMM) state transitions and a N-gram language model, can be represented as WFSTs. Moreover, they can be composed into a single WFST using mathematically defined operations. By running a dynamic programming (DP)-based search on the composed WFST, decoding can be efficiently performed.

Figure 1 shows an example of a WFST that takes a sequence of phone states as input and outputs a word sequence. When it is applied to speech recognition, a sequence of feature vectors is first extracted from the input speech signal. Typically, each feature vector corresponds to a 10 ms window in the original speech signal. Then, the acoustic likelihood of phone HMM states that correspond to the input symbols of the WFST is calculated for the feature vector at each time frame. The obtained acoustic likelihood is merged with the existing arc weight, which is usually derived from HMM state transitions and N-gram probabilities. With the updated weight, a minimum cost path that starts from the initial state is searched, consumes the feature vectors one by one at each transition, and terminates at the final state. The recognition output is a sequence of the output word symbols along the path.

IV. Design of pure functional decoder "Husky"

A. Specifications

Our functional decoder "Husky" is based on WFST and implemented using Haskell. It takes an AT&T format WFST definition file [2], an HTK format HMM state definition file [8], a list of feature files, and a configuration file as command line arguments. The I/O is implemented using a monad. In the program, a data structure that represents all the contents of a speech file is passed to a decoding function. Because of lazy evaluation, however, the contents of the file are actually read step by step.

B. Search algorithm

The decoding algorithm implemented in Husky is a frame synchronous one-pass search. The search proceeds by expanding a hypothesis list step by step from the first feature frame to the last. Initially, the hypothesis list has a single element, which is a pair of dummy arc to the initial state and a score of 0.0. Summation of a weight and an observation score that corresponds to the input symbols and a feature vector at that time frame are then computed for all the arcs that start from the ending nodes of arcs in the hypothesis list. After that, the hypothesis list is updated. To reduce the computational cost, the beam search strategy is used where the hypothesis with accumulated scores that are too large compared to others are removed from the hypothesis list.

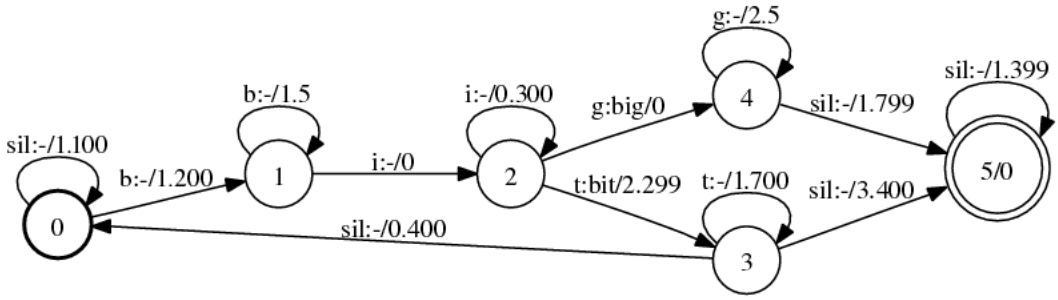

Fig. 1. Example of WFST that converts phone state sequence to word sequence. In this case, input phone state label is either "b", "i", "g", "t", or "sil" and output word symbol is either "bit" or "bat". "-" indicates epsilon (null) symbol. Numbers associated with each arc is weight of transition. State 0 is initial state and state 5 is final state.

```
1 decode :: WFST -> HMMSTATE -> FEATURESEQ -> LATTICE
2 decode wfst hmmStates featureSeq = decodeSub featureSeq ( initializeLattice ( initialState wfst))
3   where decodeSub featureSeq2 lattice
4           | featureSeq2 == [] = lattice -- Returns resulting lattice when all the frames are processed
5           | otherwise = decodeSub (tail featureSeq2) ((expand wfst hmmStates x activeHypotheses): lattice )
6           where activeHypotheses = pruneHypothesis (head lattice )
7                 x = head featureSeq2
```

Fig. 2. Pseudo Haskell code for one-pass beam decoding.

A pseudo Haskell code is shown in Figure 2. The main "decode" function takes a WFST definition, a set of HMM state definitions, and a feature sequence as arguments (line 2 in the code) and applies a sub function "decodeSub", which is a tail recursion. "Expand" expands the hypothesis list, and the result is appended to the lattice of that stage (line 5). For each recursive application of decodeSub, the feature frame proceeds one by one. When all the feature frames are processed, a lattice is returned (line 4). Output word sequence is obtained by backtracking the lattice from the last frame to the first.

C. Data structure

As stated above, there are no variables in pure functional programming, and it is impossible to update a variable value. This means, if a specific element of an array must be updated, a new array must be generated where all but one element has a different value from the original one. When the array is large, this wastes huge amount of memory and CPU time. One solution is to use monad to update arrays, and the other is not to use arrays. By using data structures such as lists and trees, it is possible to generate a whole new structure with one element being updated in the appearance, and share a large part of its body with the original in the backyard. Despite the disadvantage, arrays are still useful in pure functional programming when it is used as a constant. Once it is initialized, read access to an element of an array is efficient as it is $O(1)$.

In Husky, the WFST network and the HMM state definitions are stored in arrays as they are constant during the decoding once they are initialized. The hypothesis list is implemented as a tree, since it needs to be continuously updated.

In decoding, an observation score is repeatedly computed for the same HMM state and the same feature vector. Therefore, it is important to have a caching mechanism. One solution is to prepare an array at each frame whose element is the observation score for an HMM state. With lazy evaluation, only the referred elements are actually computed. Another solution is to use a library function that implements memoization [6], which has the same effect. The latter strategy is adopted in Husky.

V. Experimental setup

A standard evaluation set of the Corpus of Spontaneous Japanese (CSJ) [9] consisting of ten academic presentations given by male speakers was used as a test set. The length of each presentation is about 10 to 20 minutes and the total duration is 2.3 hours. A tied-state Gaussian mixture triphone HMM with 32 mixtures per state was used as the acoustic model. The HMM had 3000 states in total and was trained with the MPE method [10] using 254 hours of academic oral presentations from the CSJ training data. Feature vectors had 39 elements comprising 12 MFCCs and log energy, their delta, and delta delta values. The language model was a trigram (3-gram) model trained from 6.8 M words of academic and extemporaneous presentations from the CSJ and the dictionary size was 30 k. The HMM was trained using the HTK toolkit, and WFSTs were made using the AT&T toolkit [2]. The WFST composed from the HMM state transitions and the language model had 25 M nodes and 49 M arcs. To compile Husky, the Haskell compiler GHC ver 6.10.4 was used.

VI. Experimental results

Table I shows the break down of the code size of Husky. The main body of the Husky program including the main,

TABLE II
COMPARISONS OF DECODERS IN TERMS OF CODE SIZE AND RECOGNITION PERFORMANCE. WORD RECOGNITION ACCURACY IS RATIO THAT EXPRESSES HOW MANY WORDS IN SPEECH IS CORRECTLY RECOGNIZED. REAL TIME FACTOR (RTF) IS RATIO OF CPU TIME AND LENGTH OF INPUT SPEECH. LARGER RTF INDICATES LARGER CPU COST.

Decoder	Language	Type	Code size (lines)	Word recognition accuracy (Acc)	Memory (bytes)	Real time factor (RTF)
Julius	C	Heuristic	100 k	79.8%	400 M	8
T^3	C++	WFST	30 k	81.2%	400 M	3
Husky	Haskell	WFST	**400**	81.1%	40 G	62

TABLE I
NUMBER OF LINES IN HUSKY'S SOURCE CODE

Category	Lines
Data structure definitions	43
IO functions	95
Main and search functions, etc.	119
Comments, blank lines	125
Total	382

search, and Gaussian mixture likelihood evaluation functions are described with less than 120 lines in total. Even including comments and blank lines, it is less than 400 lines. This compactness is a characteristic of Husky and is advantageous for educational and prototyping purposes.

Table II summarizes the basic characteristics of Husky. For comparison, Julius [11] and T^3 [12] decoders are also listed. Julius is a decoder that is distributed with its source code written in C language. It has been developing for 20 years and used worldwide. T^3 is a relatively new decoder based on WFST and written in C++. As can be seen in the table, the code size of Husky is much smaller than that for Julius (100 k lines) and for T^3 (30 k lines). Although it should be noted that the design goals and supported functions of Julius and T^3 are different from Husky. In particular, Julius is equipped with rich peripheral functions to support various applications. The Husky and T^3 decoders are similar in that both are based on WFST. A difference is that on-the-fly WFST composition is only implemented in T^3 [13] at this stage. Nevertheless, it can be said that Husky is the most compact decoder implementation that works as a large-vocabulary continuous speech recognizer.

In terms of word recognition accuracy, which is a ratio that expresses how many words in a speech are correctly recognized, Husky had an accuracy of 81.1% for the test set. This was almost the same as that of accuracy T^3 (81.2%). These accuracies were slightly better than that of Julius (79.8%). On the other hand, in the current Husky implementation, it took 40 Gbytes of memory and 62 times of real time to process the test set. These numbers are much larger than other decoders. One reason of the large memory and CPU cost is that memory allocation and garbage collection (GC) is repeated behind data passing among functions. Another reason might be a problem in the GC routine in GHC as we have encountered segmentation faults several times depending on the default heap memory size specified through the RTS option, which should never occur in pure functional programming regardless of how the application program is written.

VII. CONCLUSION

We developed a WFST decoder called Husky based on pure functional programming for educational and prototyping purposes. We showed that a decoder can be described with less than 400 lines including comments and blank lines. Speech recognition result shows that it achieves the same word accuracy as a state-of-the-art system. Future work includes improving the computational efficiency and extending the system to model training and adaptation. Efficiency may be improved by using a stream fusion method, tuning the evaluation strategy, and implementing general graph operations as an intrinsic library. Implementing online adaptation using the lazy evaluation framework is also of interest. Husky's source code is available at the first author's home page.

ACKNOWLEDGMENT

This research has been partially supported by JST, Research Seeds Program, and Japan Society for the Promotion of Science, Grant-in-Aid for Scientific Research (B) 21300062.

REFERENCES

[1] J. Bilmes, *GMTK: The Graphical Models Toolkit*, University of Washington, Electrical Engineering.
[2] M. Mohri, F. C. N. Pereira, and M. Riley, "Weighted finite-state transducers in speech recognition," *Computer Speech and Language*, vol. 16, no. 1, pp. 69–88, 2002.
[3] S. Watanabe, T. Hori, and A. Nakamura, "Large vocabulary continuous speech recognition using wfst-based linear classifier for structured data," in *Proc. Interspeech*, 2010, pp. 346–349.
[4] J. Hughes, "Why Functional Programming Matters," *Computer Journal*, vol. 32, no. 2, pp. 98–107, 1989.
[5] P. Hudak, "Conception, evolution, and application of functional programming languages," *ACM Comput. Surv.*, vol. 21, no. 3, pp. 359–411, 1989.
[6] S. Thompson, *Haskell : the craft of functional programming*, Addison Wesley, March 1999.
[7] P. Wadler, "Comprehending monads," in *Mathematical Structures in Computer Science*, 1992, pp. 61–78.
[8] S. Young *et al.*, *The HTK Book*, Cambridge University Engineering Department, 2005.
[9] T. Kawahara, H. Nanjo, T. Shinozaki, and S. Furui, "Benchmark test for speech recognition using the Corpus of Spontaneous Japanese," in *Proc. SSPR2003*, 2003, pp. 135–138.
[10] D. Povey and P.C. Woodland, "Minimum phone error and I-smoothing for improved discriminative training," in *Proc. ICASSP*, 2002, vol. I, pp. 105–108.
[11] A. Lee, T. Kawahara, and S. Doshita, "An efficient two-pass search algorithm using word trellis index," in *Proc. ICSLP*, 1998, pp. 1831–1834.
[12] P. R. Dixon, D. A. Caseiro, T. Oonishi, and S. Furui, "The TITech large vocabulary WFST speech recognition system," in *Proc. IEEE ASRU*, 2007, pp. 443–448.
[13] T. Oonishi, P.R. Dixon, K. Iwano, and S. Furui., "Optimization of on-the-fly composition for wfst-based speech recognition decoders," *IEICE Transactions*, pp. 1026–1035, Jul 2009.