Regular Paper

A Compact Speech Decoder Based on Pure Functional Programming

Takahiro Shinozaki,[†1,†2,†3] Masakazu Sekijima,[†1] Shigeki Hagihara[†1] and Sadaoki Furui [†1]

The history of automatic speech recognition started with small vocabulary, isolated word recognition tasks. It has now been improved to work for continuous speech recognition tasks with a large vocabulary. One of the core software in a speech recognition system is called a decoder, which takes speech feature sequence as an input, searches for a word sequence that best matches to the input, and outputs the sequence. Possible number of word sequences exponentially increases according to the length of input speech. Therefore, an efficient search strategy is required. Source code of existing decoders is complex and amounts to more than 10000 lines. Because of the complexity of the software, it is a bottleneck for speech researchers to modify an existing decoder to try a new idea. Seeking for a new framework to compactly describe a decoder for prototyping and educational purposes, we have applied the pure functional programming. Our developed pure functional decoder is written in Haskell and named "Husky". Thanks to the high abstraction ability of the pure functional language, Husky has only 400 lines, which is significantly smaller than existing decoders. In this paper, we first briefly introduce the basics of the speech recognition algorithm. Then, the design and implementation of Husky are described. Finally, the performance of Husky is demonstrated by large vocabulary continuous speech recognition experiments.

1. Introduction

Automatic speech recognition is a technique to convert spoken words to text. Today's speech recognition systems are based on noisy channel model[1]. That is, we encode a word sequence to speech waveform through our throat and mouth changing their shapes by muscles. Because the encoding always have variation, the waveform varies even for the same word sequence. Speech recognizers receive the waveform and try to guess the original word sequence based on statistical models that model the relationship between word sequences and speech waveform. In 1960's, the research of speech recognition centered on simple tasks like recognizing a small number of (order of 10-100 words) isolated words[2]. After a half century, now the speech recognition systems have been improved to handle continuously spoken large vocabulary tasks (order of 10000 to 1000000 words).

With the accumulation of the technological progress, an emerging problem is the complexity of software that implements the up-to-date speech recognition systems. In order to investigate a new statistical model or a new idea in the research of speech recognition, it is often required to modify existing software to train statistical models and to perform recognition experiments. However, such software is large and complex, and it takes a lot of time to understand where to modify. This is a bottleneck for speech processing research 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 statistical models used in speech recognition. By developing a program based on the library functions, the required effort for the coding can be largely reduced. However, it is difficult to design library functions predicting all possible future extensions. Therefore, it is often the case that the new idea requires modifications of the library functions and this sometimes requires a lot of effort when the library is large and complex.

Another approach is to develop a single software that supports a generalized statistical framework that includes statistical models used in speech recognition systems. Software that implements Bayesian network[3] can be regarded as an example of this approach. The framework of Weighted Finite State Transducer (WFST)[4] also provides a generalized framework to represent statistical models and is used to implement speech decoders, which is a software to perform speech decoding. Given such software, various statistical models can be investigated without modifying its program code as far as they fit in the supported class of models. However, once a new model or a training/decoding algorithm goes beyond the existing framework, a modification to existing software is again required (eg.[5] etc.). The WFST framework has also an effect to simplify the software structure to implement decoders compared to other conventional approaches,

[†1] Tokyo Institute of Technology
[†2] JST, Research Seeds Program
[†3] Moved to Chiba University

and thus reduces the effort to extend an existing code. However, existing WFST decoders written in procedural languages are still have large code sizes.

Apart from speech recognition, pure functional programming has been studied in the software engineering area as a next generation programming paradigm[6]. In the functional programming, a program consists of functions where a function can be an argument of another function or can be a returned value. In the pure functional programming, there is no side effect with function application unlike the procedural programming. Because of this, lazy evaluation is easily adopted as its evaluation scheme[7]. By using lazy evaluation, it is possible to separate a structure of function's relationship and the actual timing of computation. These properties of the pure functional programming provide mechanisms for higher abstraction and modularization in software description. Moreover, the pure functional programming is said to be advantageous in parallel processing in the future since there is freedom in the order of function evaluation. While the history of the research on pure functional programming is long, it is just recent that practical pure functional programming has become possible.

By applying the pure functional programming to describe a WFST based speech recognition system, it is expected that the system can be described compactly in a highly abstracted manner. By using that system as a baseline, any modification would be possible with minimum effort. However, the 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. In this study, we empirically investigate the application of the pure functional programming to describe a speech recognition system. A pure functional WFST decoder "Husky" is implemented and evaluated.

The organization of this paper is as follows. The basics of speech recognition is described in Section 2. Some central concepts of the pure functional programming is briefly reviewed in Section 3. The design of our pure functional decoder is shown in Section 4. Experimental conditions are described in Section 5 and the evaluation results are shown in Section 6. Conclusions and future works are given in Section 7.

2. Principles of automatic speech recognition

In a speech recognition system, speech features that represent speech characteristics are first extracted from speech waveform. Then, a word sequence that best matches to the speech features is searched based on statistical models. The statistical models are kinds of knowledge of speech and language that are stored in a computer. In the following, their principles are overviewed.

2.1 Feature extraction

Speech waveform obtained from a microphone is a one dimensional sequential data as shown in Figure 1. To analyze the waveform, it is sliced into overlapping fixed-length small segments. The typical length of the segment is 25ms and the overlap is 15ms (in other words, each segment shifts at 10ms). The Fourier transform is applied at each segment to obtain a power spectrum vector. A graph obtained by the sequence of power spectrum vectors is call a spectrogram. Figure 2 is an example of spectrogram obtained from the speech waveform shown in Figure 1. The time-frequency power pattern represents speech characteristics. For speech recognition, a couple of more transforms are applied to the spectrum vector to better extract speech characteristics and a feature vector is obtained. Typically, the dimension of the feature vector is around 20 to 40. Several features have been investigated. One of the popular features is Mel-Frequency Cepstrum Coefficients (MFCC)[8].

When the shift of the segment is 10ms, 100 feature vectors are produced for every second. Time step using the shift as a unit is called a frame. In other words, there are 100 frames per second when the shift is 10ms.

2.2 Maximum a posteriori (MAP) decoding

Let X be a sequence of feature vectors $x_1, x_2, \cdots, x_T$ and W be a sequence of words $w_1, w_2, \cdots, w_N$, where T is the number of frames of the speech and N is the length of the word sequence. It is assumed that all the words are members of a vocabulary V. In the framework of maximum a posteriori decoding, the word sequence that has the maximum a posteriori probability is searched[9] as shown in Equation (1). For continuous speech recognition, the number of words N is not known and is decided as part of the search. The role of a speech decoder software is to perform the argmax operation given an input X and output the

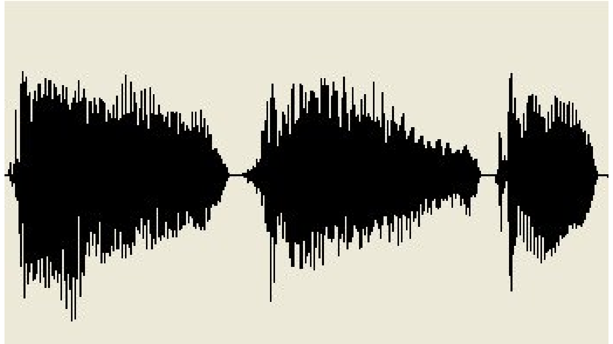

Fig. 1 An example of speech waveform. The horizontal axis is time and the vertical axis is amplitude of the signal.

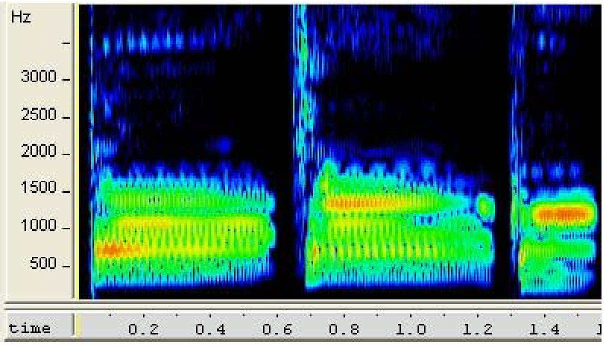

Fig. 2 Spectrogram obtained from speech waveform shown in Figure 1. The horizontal axis is time and the vertical axis is frequency.

searched word sequence W^*.

$$
\begin{aligned}
W^* &= \underset{W}{\operatorname{argmax}} P\left(W|X\right) &(1)\\
&\approx \underset{W}{\operatorname{argmax}} P_{\Lambda}\left(W|X\right) &(2)\\
&= \underset{W}{\operatorname{argmax}} P_{AM}\left(X|W\right) P_{LM}\left(W\right) &(3)\\
&\approx \underset{W,S}{\operatorname{argmax}} \left(\left(\prod_{t=1}^{T} P_{obs}\left(x_t|s_t\right) \right) P_{trn}\left(S|W\right) \right) P_{LM}\left(W\right) &(4)
\end{aligned}
$$

Since the true probability $P\left(W|X\right)$ is not known, it is approximated by a probability score $P_{\Lambda}\left(W|X\right)$ evaluated by using a speech model Λ as shown in Equation (2). Usually, it is decomposed as shown in Equation (3) using the Bayes' theorem. P_{AM} is referred to as acoustic model and it models the relationship between word symbols (or phone symbols associated to the words) and feature vectors. P_{LM} is called a language model and it models how the word sequence likely to occur in a language. Usually, hidden Markov model (HMM) is used as the acoustic model[10]. When HMM is used, Equation (3) is further approximated as shown in Equation (4), where $S = s_1, s_2, \cdots, s_T$ is called a state sequence. The HMM state s_t is an element of a state set $\{ps_1, ps_2, \cdots, ps_M\}$ where M is the number of total states and ps_m is a state name. $P_{obs}\left(x_t|s_t\right)$ and $P_{trn}\left(S|W\right)$ are referred to as state observation probability and state transition probability, respectively. For the language model, N-gram model is often used[11]. The parameters of these models are estimated using speech and text databases.

The search space of the Equation (3) is prohibitively huge. For example, if the vocabulary size $|V|$ is 10000 and the number of words N is 10, the possible number of word sequences is $10000^{10} = 10^{40}$. Therefore, an efficient search algorithm is required that utilizes the structure of the speech model and adopts some approximations. Many approaches have been investigated for this purpose[12]. Among them, WFST based speech decoders[4] are getting popular since WFST gives mathematically organized flexible framework. Details of WFST-based decoding is described in the next subsection.

2.3 Weighted finite state transducer (WFST)

WFST is a finite state machine with input and output symbols, and transition weights. Figure 3 shows an example of WFST that has $\{$“sil'', “b'', “i'', “g'', “$t''\}$ as input symbols and $\{$“bit'', “bat'', $\epsilon\}$ as output symbols. A special symbol ϵ represent the null symbol. Many of the statistical model components in speech recognition systems can be represented as a WFST. Moreover, once they are represented as a WFST, they can be composed to a single WFST using mathematically defined operations.

When a WFST of HMM state transitions and a WFST of a N-gram language

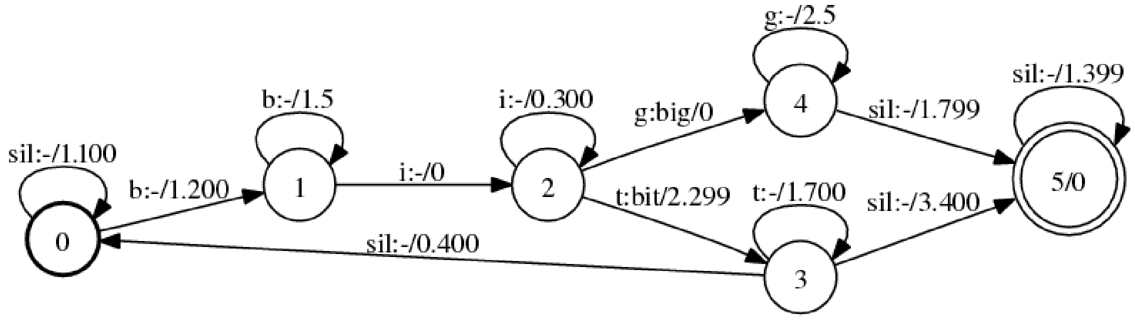

Fig. 3 An example of a WFST. In this case, an input phone state symbol is either "b", "i", "g", "t", or "sil" and an output word symbol is either "bit" or "bat". "-" indicates ϵ (null) symbol. A number associated to each arc is a weight of the transition. State 0 is an initial state and state 5 is a final state.

model are composed in this order[⋆1], the input symbol of the composed WFST is an HMM state, and the output symbol is a word or ϵ. Let p be a path that starts from the initial WFST state and ends at the final state. Also, let the input symbol sequence along p be $S = I(p)$ and the corresponding output symbol sequence be $W = O(p)$. Then, the WFST weight score accumulated along p is equal to $P_{trn}(S|W)\, P_{LM}(W)$ because of the WFST construction. When it is applied to speech recognition, state observation score $P_{obs}(x_t|s_t)$ is computed and the result is combined with the WFST score to search the best word sequence W^* as shown in Equation (5), which is equivalent to Equation (4).

$$P^* = \operatorname*{argmax}_{P} \left(\prod_{t=1}^{T} P_{obs}(x_t|s_t) \right) P_{trn}(S|W)\, P_{LM}(W),$$
$$W^* = O(P^*) \tag{5}$$

In order to efficiently compute Equation (5), a graph called lattice is useful. Figure 4 shows an example of a lattice. The horizontal axis of the lattice is the feature vectors of an utterance aligned to time frames and the vertical axis is the WFST state. In the lattice, node at k-th row and t-th column corresponds to k-th WFST state and t-th time frame. An arc from $(k,t-1)$ node to (k',t) node corresponds to a WFST arc from k-th WFST state to k'-th WFST state at time t. The arc has the same weight as the corresponding WFST arc. The arc also has an observation score $P_{obs}(x_t|s)$, where s is the input label of the WFST arc and x_t is the feature vector at time t. By multiplying these WFST weight and observation scores along a path p on the lattice from the initial node (0,0) to the final node (K, T), a score that is equal to $\left(\prod_{t=1}^{T} P_{obs}(x_t|s_t)\right) P_{trn}(S|W)\, P_{LM}(W)$ is obtained. Therefore, finding a maximum score path on the lattice is equivalent to computing Equation (5). By taking a log of the weight and the observation scores, the products are converted to summations. By multiplying -1 to these log-domain scores to negate, the problem is converted to finding a minimum score path. To efficiently find the minimum score path, dynamic programming can be used.

Figure 5 is a pseudo code for a dynamic programming based minimum score path search algorithm. The scores mentioned in the code are the negative log score. In the code, lattice arc is a data structure that include input/output symbols and the weight of the corresponding WFST arc. The function "Insert" inserts (arc, score) pair to the hypothesis list according to the ending node of the arc. That is, if there is no arc with the same ending node in the list, it simply add the pair to it. However, if an arc with the same ending node already exist in

[⋆1] When a pronunciation dictionary is used that defines a mapping from a word to its pronunciation, it is also converted to WFST and is integrated.

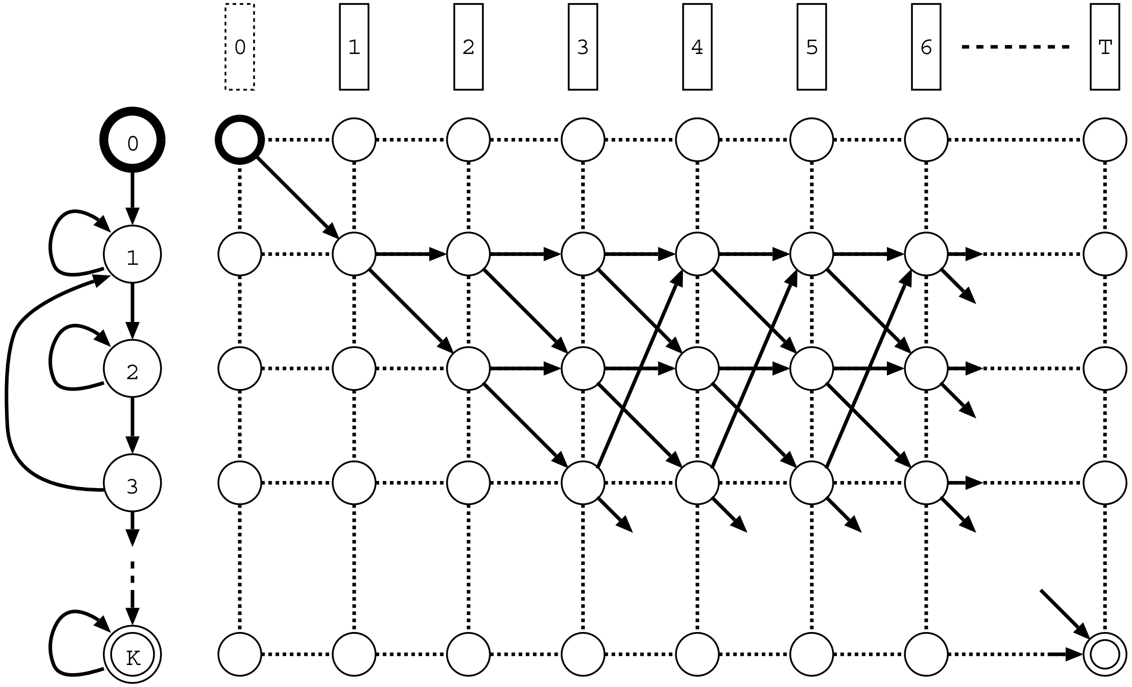

Fig. 4 An example of a lattice. In the lattice, node at k-th row and t-th column corresponds to k-th WFST state and t-th time frame. An arc from $(k,t-1)$ node to (k',t) node corresponds to a WFST transition from k-th state to k'-th state at time t. The zero-th feature vector is a dummy.

the list, it holds either of the pairs with smaller score. The function "Expand" returns new (arc, score) pairs for arcs starting from the ending node of the given arc. The new score is a sum of the score of the original pair, WFST weight score of the new arc, and observation score for the input label and the feature vector at that time frame. The function "Append" appends a list of hypothesis to the current partial lattice. The process proceeds step by step according to the time frame. When all the frames are processed, the minimum score path is obtained by backtracking the lattice by choosing the arc with the lowest accumulated score one by one along the ending node to the initial node. This algorithm is called frame synchronous one-pass search.

In real application to speech recognition, a handling of ϵ input symbol is also required. When there is a WFST arc with an ϵ input symbol, a transition occurs without consuming a time frame, which corresponds to a vertical transition in the lattice.

3. Pure functional programming

In the procedural programming, such as C and C++, a program has a state that is represented by variables, and the variables 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.

```
// Initialize a hypothesis list hyps with a single element (0,0).
hyps := (); // list of a pair of ( lattice_arc , accumulated_score)
lattice := (); // list of hyps
Insert(hyps, (dummyInitArc, 0.0)); // dummy arc to the initial lattice state (0,0) with score 0.0.
// Main loop
for t:=1 to T do
begin
  hypsTmp := ();
  for h in hyps do
  begin
    Insert(hypsTmp, Expand(h, x[t]));
  end;
  hyps := hypsTmp;
  Append(lattice, hyps);
end;
bestPath := Backtrack(lattice);
```

Fig. 5 A pseudo code for a dynamic programming based minimum score path search algorithm.

In contrast to the procedural programming, there is no variable in the pure functional programming and the function has no side effect. It is originated from the lambda calculus and has a long history, but it has mainly been interested in academic research. However, with the development of practical languages such as Haskell[13)] and Clean, it has recently started to be used in engineering. The concepts that characterize the pure functional programming include pure functions, first-class functions, recursion and lazy evaluation. They are briefly described in the following subsections.

3.1 Pure functions

Pure functions have no side effect. In the 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 reduce bugs and increases 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[14)]. A monad is a kind of abstract data type constructor used to represent computations. By using a monad, Haskell separates I/O with other part of the program keeping the pureness of the program.

3.2 First-class functions

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

3.3 Recursion

In the pure functional programming, there is no variable. Therefore, there is no syntax to represent “for” or “while” loop. Instead, iteration is accomplished via recursion. Recursive functions invoke themselves to repeat an operation. In general, recursion needs a stack to keep track of the recursive function calls and 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.

3.4 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 improve the 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 function, and still get efficient execution. The match of lazy evaluation with variable substitution is unfortunately not good, and it is specifically used in the pure functional programming. In Haskell, a programmer can choose the evaluation strategy. The default is lazy evaluation.

4. Design of a pure functional decoder “Husky”

4.1 Specification

Our functional decoder “Husky” is based on WFST and implemented using Haskell. It takes an AT&T format WFST definition file[4)], an HTK format HMM state definition file[15)], 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 whole contents of a speech file is passed to a decoding function. However, because of the lazy evaluation, contents of the file is actually read step by step.

4.2 Search algorithm

The decoding algorithm implemented in Husky is a frame synchronous one-pass search that is described in Section 2.3. 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 of a pair of dummy arc to the initial state and a score 0.0. Then, summation of a weight and a observation score that corresponds to the input symbols and a feature vector at that time frame are 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 having too large accumulated scores compared to others are removed from the hypothesis list.

A pseudo Haskell code is shown in Figure 6. The main “decode” function takes a WFST definition, an HMM state definition, and a feature sequence as arguments (line 2 in the code), and it 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.

4.3 Data structure

In the pure functional programming, there is no variable and it is impossible to update a value of a variable. This means, if a specific element of an array need to be updated, a new array needs to be generated where all but one element has a different value from the original one. When the array is large, this causes huge waste of memory and CPU. 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 large part of its body with the original in the backyard. Despite of the disadvantage, arrays are still useful in the 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, observation score is repeatedly computed for the same HMM state and the same feature vector. Therefore, it is important to have some caching mechanism. One solution is to prepare an array at each frame whose element is observation score for an HMM state. With the lazy evaluation, only the referred elements are actually computed. Another solution is to use a library function that implements memoization[13)] that has the same effect. The latter strategy is

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

Fig. 6 Pseudo Haskell code for one-pass beam decoding.

adopted in Husky.

5. Experimental setups

A standard evaluation set of the Corpus of Spontaneous Japanese (CSJ)[16] consisting of 10 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 by the MPE method[17] 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 6.8M words of academic and extemporaneous presentations from the CSJ and the dictionary size was 30k. The HMM was trained by using the HTK toolkit and WFSTs were made using the AT&T toolkit[4]. The WFST composed from the HMM state transitions and the language model had 25M nodes and 49M arcs. To compile Husky, the Haskell compiler GHC ver 6.10.4 was used.

6. Experimental results

Table 1 summarizes basic characteristics of the pure functional WFST decoder Husky. As for comparison purpose, julius[18] and T^3 [19] decoders are also listed. Julius is a decoder that is distributed with its source code written in the C language. It has been developed for 20 years and widely used in the world. T^3 is a relatively new decoder based on WFST and written in C++. Note these decoders support different functions so it is not fair to directly compare their source code sizes. Especially, julius equips rich peripheral functions to support various applications. Nevertheless, their code sizes are shown in the table. As can be seen in the table, the code size of Husky is only 400 lines whereas it is 100k for julius and 30k for T^3. Despite of the difficulty of the direct comparison, it can be seen that the code size of Husky is extremely small compared to the other decoders. The Husky and T^3 decoders are both based on WFST and they are similar with that point. The main difference of Husky and T^3 in terms of supported functions is that on the fly WFST composition is only implemented in T^3 [20] at this stage.

In terms of word recognition accuracy, which is a ratio that expresses how much of words in speech is correctly recognized, Husky gave 81.1% for the test set. This was almost the same as the accuracy 81.2% given by T^3. These accuracies were slightly better than 79.8% given by julius. On the other hand, in the current Husky implementation, it took 40 Gbyte 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 the 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 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.

Table 1 Comparisons of decoders in term of code sizes and recognition performance. Word recognition accuracy is a ratio that expresses how much of words in speech is correctly recognized. The real time factor (RTF) is a ratio of CPU time and the length of input speech. Larger RTF indicates larger CPU cost.

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

Table 2 Number of lines in the 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

Table 2 shows the break down of the code size of Husky. The main and search functions are described with less than 120 lines. Even including comments and blank lines, it is less than 400 lines. This compactness is the characteristics of Husky and is advantageous for prototyping and educational purposes.

7. Conclusion

In order to efficiently implement speech recognition systems while maintaining flexibility for future modification, application of the pure functional programming has been investigated. It has been shown 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 the state-of-the-art system. Future works include improving the computational efficiency and extending the system for model training and adaptation. The efficiency could be improved by utilizing a stream fusion method, tuning the evaluation strategy, and implementing some general graph operations as an intrinsic library.

The source of the functional decoder "Husky", a manual, and sample data are available at Takahiro Shinozaki's home page. The URL as of March 2011 is `http://www.furui.cs.titech.ac.jp/~shinot/`

Acknowledgments This research was partially supported by JST, Research Seeds Program. This work was also conducted as part of KAKENHI (21700188).

References

1) MacKay, D.J.: *Information Theory, Inference, and Learning Algorithm*, Cambridge university press (2005).
2) Juang, B.H. and Rabiner, L.R.: *Automatic Speech Recognition - A Brief History of the Technology* (2005).
3) Bilmes, J.: *GMTK: The Graphical Models Toolkit*, University of Washington, Electrical Engineering.
4) Mohri, M., Pereira, F. C.N. and Riley, M.: Weighted Finite-State Transducers in Speech Recognition, *Computer Speech and Language*, Vol. 16, No. 1, pp. 69–88 (2002).
5) Watanabe, S., Hori, T. and Nakamura, A.: Large vocabulary continuous speech recognition using WFST-based linear classifier for structured data, *Proc. Interspeech*, pp.346–349 (2010).
6) Hughes, J.: Why Functional Programming Matters, *Computer Journal*, Vol. 32, No.2, pp.98–107 (1989).
7) Hudak, P.: Conception, evolution, and application of functional programming languages, *ACM Comput. Surv.*, Vol.21, No.3, pp.359–411 (1989).
8) Davis, S. and Mermelstein, P.: Comparison of parametric representations for monosyllabic word recognition in continuously spoken sentences, *Acoustics, Speech and Signal Processing, IEEE Transactions on*, Vol.28, No.4, pp.357 – 366 (1980).
9) Glass, J., Chang, J. and McCandless, M.: A probabilistic framework for feature-based speech recognition, *Interspeech*, Vol.4, pp.2277 – 2280 (1996).
10) Rabiner, L.: A tutorial on hidden Markov models and selected applications in speech recognition, *Proceedings of the IEEE*, Vol.77, No.2, pp.257 –286 (1989).
11) Manning, C.D. and Schütze, H.: *Foundations of statistical natural language processing*, MIT Press, Cambridge, MA, USA (1999).
12) Aubert, X.L.: An overview of decoding techniques for large vocabulary continuous speech recognition, *Computer Speech and Language*, Vol. 16, No. 1, pp. 89 – 114 (2002).
13) Thompson, S.: *Haskell : the craft of functional programming*, Addison Wesley

(1999).

14) Wadler, P.: Comprehending Monads, *Mathematical Structures in Computer Science*, pp.61–78 (1992).

15) Young *et al.*, S.: *The HTK Book*, Cambridge University Engineering Department (2005).

16) Kawahara, T., Nanjo, H., Shinozaki, T. and Furui, S.: Benchmark test for speech recognition using the Corpus of Spontaneous Japanese, *Proc. SSPR2003*, pp.135–138 (2003).

17) Povey, D. and Woodland, P.: Minimum Phone Error and I-Smoothing for Improved Discriminative Training, *Proc. ICASSP*, Vol.I, pp.105–108 (2002).

18) Lee, A., Kawahara, T. and Doshita, S.: An efficient two-pass search algorithm using word trellis index, *Proc. ICSLP*, pp.1831–1834 (1998).

19) Dixon, P.R., Caseiro, D.A., Oonishi, T. and Furui, S.: The TITech large vocabulary WFST speech recognition system, *Proc. IEEE ASRU*, pp.443–448 (2007).

20) Oonishi, T., Dixon, P., Iwano, K. and Furui., S.: Optimization of On-the-fly Composition for WFST-based Speech Recognition Decoders, *IEICE Transactions*, pp. 1026–1035 (2009).