An Algebraic Interface for GETA Search Engine

Takuma Murakami[†] Zhenjiang Hu[†] Shingo Nishioka[†]
Akihiko Takano[††] Masato Takeichi[†]

[†]University of Tokyo
murakami@ipl.t.u-tokyo.ac.jp
{hu,nis,takeichi}@mist.i.u-tokyo.ac.jp
[††]National Institute of Informatics
aki@nii.jp

Abstract

GETA is a library that implements high performance method for *associative computation* to be used as a basis of various document processing including searching or clustering. We proposed an algebraic view for the GETA engine with concrete implementation of an interface which connects the high level view with the underlying efficient library. Users can write highly abstract programs in a calculational way through the interface and utilize program transformation techniques formalized as algebraic laws. The interface incorporates the algebraic operations into GETA basic operations which can be executed quite efficiently.

1 Introduction

GETA (Generic Engine for Transposable Association) is a software library for various document processing [5]. It has significant scalability that handles more than two millions of documents. One feature of GETA is the notion of *associative computation*; GETA implements the associative computation as a set of operations for users to make their own software such as search engines or text clustering software. The associative computation measures similarity between words or between documents. The combinations of the associative operation and other operations build up keyword searching, document clustering and other document processing.

However, the collection of the basic operations is rather large, and not well structured. It is hard to come up how to choose and combine them for desired functions. Furthermore, it is hard to study relations of some functions built from basic operations. From the viewpoint of library users, a simple and well-structured interface is very helpful for programming, that provides a systematic way to analyze, transform and optimize the code which interacts with GETA library.

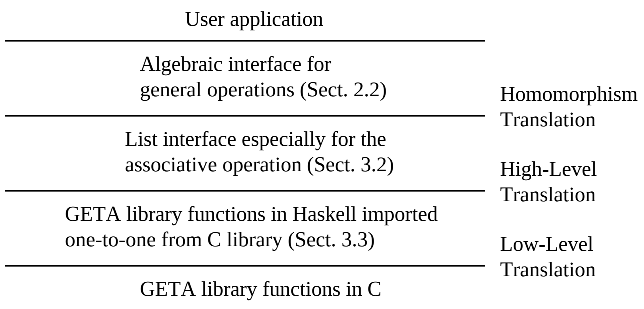

User application
Algebraic interface for general operations (Sect. 2.2)
List interface especially for the associative operation (Sect. 3.2)
GETA library functions in Haskell imported one-to-one from C library (Sect. 3.3)
GETA library functions in C
Homomorphism Translation
High-Level Translation
Low-Level Translation

Figure 1: Layered interfaces

In this paper we present a stratified interface which provides structured access to GETA and illustrate its use with an application system. Figure 1 illustrates the overview of the interface. The top of the layers offers a general interface for interacting with WAM. A list-based interface, which involves the associative operator, is greatly convenient to use with other list processing functions. Rather low level interface providing low level functions resides in the bottom of our layer.

In the rest of the paper Section 2 describes general method for manipulating databases in an algebraic framework. Section 3 shows more specific interface for the associative operation which is the core of GETA library. We demonstrate a searching system using our interface in Section 4 and conclude in Section 5. Note that we use a purely functional language Haskell [1] as the implementation language.

2 An Algebraic Interface for GETA

In this section we first introduce a data structure WAM (Word Article Matrix), a central data structure in GETA library, as well as other structures and operations. Because we are focusing on the interface to be used by library users, WAM and related structures and operations are described from the viewpoint of users rather than implementation details. Next we propose an algebraic model for WAM. The model provides a constructive view of WAM and accepts fairly general operations on WAM. Our interface translates the general operations into basic GETA operations as efficient as possible with the help of the function fusion and other program transformation techniques.

2.1 GETA from User's View

WAM represents the database of GETA. It can be modeled as a large matrix shown in Fig. 2 in which rows are indexed by names of documents (articles) and columns are indexed by words. Each element in the WAM shows how many

	rain	today	service
news	0	3	1
travel	0	1	2
weather	2	1	0
diary	0	1	0

Figure 2: A sample WAM

	rain	today	service
query	1	1	0

Figure 3: A sample query

times a word occurs in a document. It is the most important data structure with which text searching and other operations are performed.

In GETA library documents and words are in the form of vector: a document is a row that is regarded as a multiset of words. Dually, a word is a column that is regarded as a set of documents in which the word occurs. Therefore WAM can be seen as either *collection of documents* or *collection of words*. The duality is important to the associative operation.

The associative operation is the main operation of GETA. Consider a keyword search for an example. An input for keyword search is a set of words which can be regarded as a document in GETA. In fact we regard the set of words as a virtual document. Associative operation is to calculate *similarity* between the input vector and each document in the WAM by means of similarity function which can be freely defined. The result is a list of similarities between the query and all documents in the WAM. Let the words "rain today" be a query for the sample WAM in Fig. 2. We regard the query as a document which consists of two words "rain" and "today" one for each. Figure 3 shows the virtual document from the query. Assuming that the similarity between two vectors is the inner product of them for simplicity, then the result should be a column vector illustrated in Fig. 4. In the vector the numbers for each row indicate their similarity to the query document. Simultaneously, we can consider the resulting vector as a virtual word which occurs three times in the document "news", once in "travel", and so on.

It should be noted that the associative operation is completely dual. By transposing the WAM matrix one database can be simultaneously used for calculating similarities between words and between documents. With this model the duality is naturally expressed as transposition of matrices or vectors. The ability of transposing, which means switching between the world of documents and words, makes GETA a comprehensive library especially suitable for document processing.

In addition to the associative operation, we may manipulate WAM database itself. It includes operations to get information on WAM, to change contents

	result
news	3
travel	1
weather	3
diary	1

Figure 4: Result of associative operation

of WAM, to extent WAM, or to calculate values from contents of WAM. These operations make GETA close to a full-fledged database library. Although the new operations are not so efficient as the associative operation, they can be effectively combined with the associative operation to develop new applications.

2.2 An Algebraic Model for WAM

Because desired operations vary by users, it is not so good to provide a fixed set of operations. Figure 5 is a list of C functions exported by GETA library which adopts this approach to permit low-level access to the library. Users can fully take advantage of flexibility of the use of such functions but need to develop appropriate combinations of them.

On the other hand, we choose to provide a general way to define operations on WAM. The benefit of this approach is not limited to the flexibility but includes affinity for program transformation. In this framework we offer a higher-order function that denotes an algebra on WAM. All operations should be systematically defined as homomorphisms from the algebra, which in turn be fused among each other as Section 2.3. Hence combinations of these operations are brought together to reduce the number of access to WAM database.

Apart from the implementation we propose a constructive view for WAM. We consider WAM is constructed by the following three constructors:

$$\begin{array}{lll} |\cdot| & :: & (\mathit{Doc}, \mathit{Word}, \mathit{Occ}) \rightarrow \mathit{Wam} \\ (\ominus) & :: & \mathit{Wam} \rightarrow \mathit{Wam} \rightarrow \mathit{Wam} \\ (\phi) & :: & \mathit{Wam} \rightarrow \mathit{Wam} \rightarrow \mathit{Wam} \end{array}$$

The constructor $|\cdot|$ takes a triplet that specifies the number of occurrences of a word in a document and constructs a WAM as a 1×1 matrix. The constructors $(\ominus)$ and (ϕ) connect two WAMs horizontally and vertically respectively. Any WAM can be constructed with these three constructors. Additionally, these constructors satisfy the following properties:

$$\begin{array}{lll} (w_1 \ominus w_2) \ominus w_3 & = & w_1 \ominus (w_2 \ominus w_3) \\ (w_1 \phi w_2) \phi w_3 & = & w_1 \phi (w_2 \phi w_3) \\ (w_1 \ominus w_2) \phi (w_3 \ominus w_4) & = & (w_1 \phi w_3) \ominus (w_2 \phi w_4) \end{array}$$

These laws ensure equivalence of WAMs independently of the way they have been combined.

```
int wam_init(char const *);
WAM *wam_open(char const *);
void wam_close(WAM *);
int wam_size(WAM *, int);
char const *wam_id2name(WAM *, int, unsigned int);
unsigned int wam_name2id(WAM *, int, char const *);
int wam_elem_num(WAM *, int, unsigned int);
int wam_freq_sum(WAM *, int, unsigned int);
int wam_max_elem_num(WAM *, int);
int wam_max_freq_sum(WAM *, int);
int wam_total_elem_num(WAM *, int);
int wam_total_freq_sum(WAM *, int);
struct syminfo *wsh(struct syminfo const *, int, WAM *,
int, int, int *, int *);
struct syminfo *wwsh(struct syminfo const *, int, WAM *,
int, int, int *, int *, void *);
struct syminfo *wstem(char const *, WAM *,
int, char const *, int *);
int wam_setopt(WAM *, int, char *);
int wam_get_vec(WAM *, int, u_int,
struct xr_vec const **);
char const *wam_handle(WAM *);
```

Figure 5: Some GETA library functions in C

The three constructors together with the three laws form an *algebra*. After that we only have to think of the algebra instead of real WAM because the algebra captures the property that we want to focus on.

Given the constructors we can consider the following homomorphism for manipulating WAM:

$$\begin{array}{lcl}
(\![k, \obar, \ominus]\!)\ |(a,b,c)| & = & k\ (a,b,c) \\
(\![k, \obar, \ominus]\!)\ (a \mathbin{-\!\!\!\circ\!\!\!-} b) & = & (\![k, \obar, \ominus]\!)\ a \obar (\![k, \obar, \ominus]\!)\ b \\
(\![k, \obar, \ominus]\!)\ (a \mathbin{\phi} b) & = & (\![k, \obar, \ominus]\!)\ a \ominus (\![k, \obar, \ominus]\!)\ b
\end{array}$$

This homomorphism brings the algebra $(|\cdot|, (\mathbin{-\!\!\!\circ\!\!\!-}), (\mathbin{\phi}))$ to another algebra $(k, \obar, \ominus)$. Apparently the destination algebra must satisfy the same associative property below:

$$\begin{array}{lcl}
(x_1 \obar x_2) \obar x_3 & = & x_1 \obar (x_2 \obar x_3) \\
(x_1 \ominus x_2) \ominus x_3 & = & x_1 \ominus (x_2 \ominus x_3) \\
(x_1 \obar x_2) \ominus (x_3 \obar x_4) & = & (x_1 \ominus x_3) \obar (x_2 \ominus x_4)
\end{array}$$

An example of homomorphism is the function *wc* which counts the number of words in the whole database.

$$\begin{array}{l}
wc :: \ Wam \to Occ \\
wc = (\![thd, +, +]\!) \\
\qquad \text{where } thd\ (a,b,c) = c
\end{array}$$

Another example is the function *contains* that lists documents containing the word w.

$$\begin{array}{ll} contains & :: Word \to Wam \to [Doc] \\ contains\ w & = (\!|f, +\!\!+, +\!\!+|\!) \\ & \quad \text{where } f\ (a, b, c) = \ \text{if } b == w \ \&\& \ c >= 1 \\ & \qquad\qquad\qquad\qquad\quad \text{then } [a] \\ & \qquad\qquad\qquad\qquad\quad \text{else } [] \end{array}$$

With the framework of homomorphism we can freely cross over the algebras on WAM. It means that each algebra represents the way to process data in WAM. Therefore we are now having a general tool to describe our operation on WAM. Furthermore, as long as we denote our operations in the framework of this algebraic view, we can automatically combine the operations as in next section.

2.3 Function Fusion

One of the most notable benefits from the algebraic framework is the *function fusion* [4]. With respect to the algebra on WAM in the previous section we can define the fusion law as follows.

$$\frac{g\ (a \oplus b) = g\ a \oplus' g\ b \qquad g\ (a \ominus b) = g\ a \ominus' g\ b}{g \circ (\!|k, \oplus, \ominus|\!) = (\!|g \circ k, \oplus', \ominus'|\!)}$$

Given a homomorphism on WAM $(\!|k, \oplus, \ominus|\!)$ and a function g which satisfy the condition, we can fuse them and obtain a new homomorphism. Therefore the number of access to WAM does not increase, which avoids performance deterioration.

One simple but quite effective case is the fusion with *map* function. *map* is a commonly-used function to modify values in a data structure without changing the structure itself. The *map* for WAM is defined as a homomorphism.

$$map\ f = (\!|f, (\oplus), (\ominus)|\!)$$

According to the fusion law for WAM, *map* can be automatically fused with an arbitrary homomorphism $(\!|k, \oplus, \ominus|\!)$ as following:

$$(\!|k, \oplus, \ominus|\!) \circ map\ f = (\!|k \circ f, \oplus, \ominus|\!)$$

The obtained homomorphism can be again fused with other homomorphism repeatedly. Because we provide only the form of homomorphism to define operations on WAM, there must be many chances to synthesize functions.

3 Interface Translations

Our interface for Haskell is built without side-effect in order to cooperate with pure functions for easy program transformation. It is based on the assumption that the back-end database of GETA is invariable through the execution of a program. This assumption is reasonable because GETA database is large and the update is done by batch processing.

The translation from functional code to GETA library calls is done in three levels as in Fig. 1. Operations written as the form of homomorphism are at first translated to rather intuitive list-based code. The purely functional code is then translated into more primitive intermediate code. This intermediate code is almost one-to-one mapping from the original GETA library calls written in C, which of course have side effects. The non-pure aspects including side effects or initialization/de-initialization calls are encapsulated within the IO monad of Haskell.

The purpose of the low level translation is inter-language translation. It calls C library functions corresponding to Haskell functions given from the high level layer. The FFI (Foreign Function Interface) feature of Haskell serves this translation.

3.1 Homomorphism Translation

With our interface users can write programs both for the algebraic interface and for list interface. The first step of our translations is to translate the former code into the latter code. Recall the properties that all algebras must satisfy:

$$\begin{array}{lcl} (x_1 \varobar x_2) \varobar x_3 & = & x_1 \varobar (x_2 \varobar x_3) \\ (x_1 \ominus x_2) \ominus x_3 & = & x_1 \ominus (x_2 \ominus x_3) \\ (x_1 \varobar x_2) \ominus (x_3 \varobar x_4) & = & (x_1 \ominus x_3) \varobar (x_2 \ominus x_4) \end{array}$$

According to the properties, mapping from homomorphism to list processing is straightforward. It just implements the three data constructors of WAM through the list interface and translates the definition of homomorphism to an equivalent function manipulating lists.

Note that we can even write the homomorphism from WAM to the list-based view, named *listview* below:

$$\begin{array}{l} \mathit{listview} :: \ \mathit{Wam} \rightarrow [\mathit{Vec}] \\ \mathit{listview} = (\!|k, f, (+\!\!+)|\!) \\ \qquad\qquad \text{where } k\ (a, b, c) = [[(b, c)]] \\ \qquad\qquad\qquad\quad f = \mathit{zipWith}\ (+\!\!+) \end{array}$$

3.2 High-Level Translation

The visible surface of our list interface is significantly simple. It stands upon the model of vectors as lists and WAM as a list of lists. Most operations including the associative operation are implemented as binary operators, so-called *combinators*.

In concrete Haskell programs vectors representing documents and words are of the type `Vec` which is naturally defined as

```
type Vec = [(String, Occ)]
```

	rain	today	service
news	0	3	1

Figure 6: A vector for the document “news”

WAM is used as an abstract data type. All combinators which access WAMs follow the view that a WAM is a matrix as a collection of vectors. For example, the combinator `/>` extracts a row vector indexed by a given name from a WAM. The type of the combinator is

```
(/>) :: String -> Wam -> Vec
```

Let w be a WAM shown in Fig. 2, then the expression

```
"news" /> w
```

denotes the vector of “news” document in the WAM as in Fig. 6.

The associative operator is also implemented as a combinator `*>`. The type is

```
(*>) :: Vec -> Wam -> Vec
```

in which the first argument is the query and the last argument is the result. Let `w` be the WAM in Fig. 2 and `q` be the vector in Fig. 3. Then

```
q *> w
```

yields the vector in Fig. 4 if the similarity is the inner product. This example shows that users can write programs in the way much like on lists.

In the list-based interface, data types are very common ones and combinators have no side effects so that library users can easily use this. Moreover, lists are the best data type for propagate data between software components because many libraries and standard functions accept lists as input and generate lists as output in Haskell. The purely functional list-based programs are translated by this layer to the programs closer to the underlying GETA library. The resultant programs, although still are Haskell programs, include specific data types and functions with side effects to cooperate with GETA's C library functions. The inter-language translation is done by the low level layer described in the next section.

3.3 Low-Level Translation

The low level layer bridges between Haskell and C. Because GETA library is written in C, we should at last call C functions. Haskell offers a schema to get across to and from other languages, called FFI (Foreign Function Interface) [2], which we utilize to call C functions and handle C data structures. Thanks to the high level layer we only have to perform one-to-one translation between Haskell and C functions.

For example, there is a function that opens WAM database in GETA library. The signature of the function in C is

```
WAM *wam_open(char *handle)
```

whose argument denotes the name of WAM to be opened and whose result is a pointer to a structure that represents a WAM. This function can be naturally translated to Haskell function

```
open :: String -> Wam
```

which takes a name of WAM and returns a data structure representing the WAM. In this example FFI plays an important role to arrange calling conventions or to handle pointers.

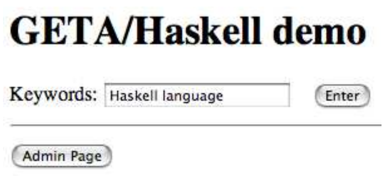
4 Application

As an application, we have developed an Web-based searching system of a mailing list about Haskell [7] using our interface. We make use of GETA library with our interface for searching engine and another high-level library for Web stuff. WASH (Web Authoring System Haskell) provides XHTML generation, CGI execution and session management [6]. Combining the two libraries with Haskell as a glue language, the system becomes notably compact in spite of its sophisticated functions and efficiency. The system provides normal keyword search, *associative search* and keyword extraction.

The name "associative search" comes from that the system takes some documents from a user and offers some documents associated with the query documents. Users only need to select documents they are interested in and readily obtain related documents. Associative search is done by combining two successive associative operations. The former one gives a set of words weighed by importance in query documents, then the latter one gives a set of documents from the set of words. In contrast to the simple examples so far, the system makes use of a complex similarity functions for the associative operation. Although we borrow predefined ones in GETA, GETA provides means to accommodate user defined similarity functions.

Figure 7 shows the entry window of the system. Note that the user interface is Web-based GUI realized by WASH. Users input some keywords into the text box and click the *Enter* button and get the result of the first keyword search as in Fig. 8. The Search Result window lists resulting mail entries which involve title, author and weights of the mail. The weight is the similarity between a mail and the query words calculated by GETA. Keywords listed above the mails are the important words extracted from the mails. It is natural that the queried words specified by the user are the top 2 of the extracted keywords. The keyword extraction is the transposed operation of keyword search in our model and written as

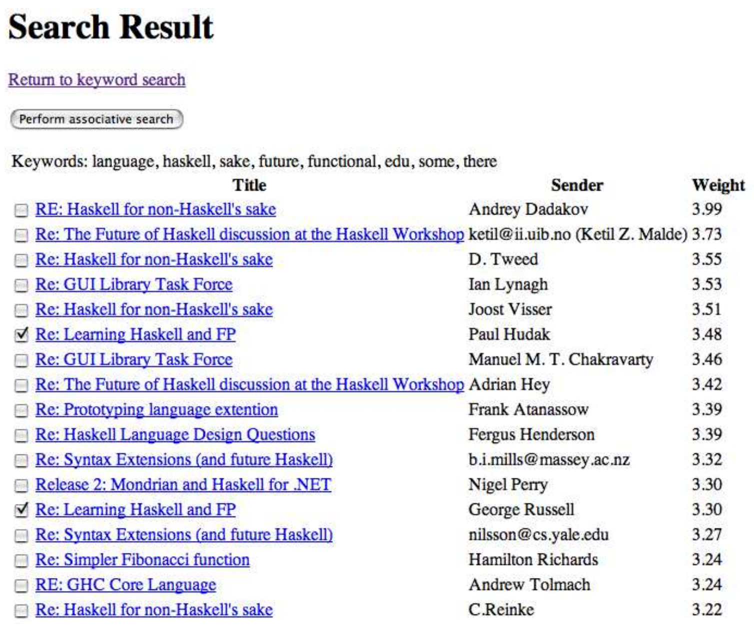
```
docs *> (trans (wam "haskell-ml"))
```



GETA/Haskell demo

Keywords: Haskell language

Figure 7: Keyword search window

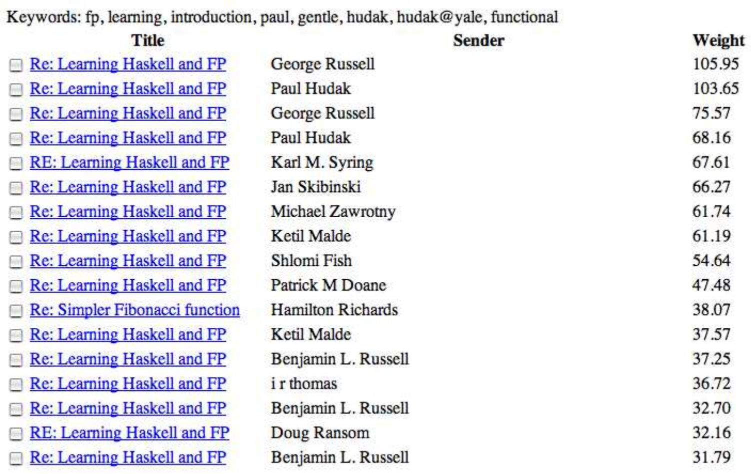


Search Result

Keywords: language, haskell, sake, future, functional, edu, some, there

	Title	Sender	Weight
☐	RE: Haskell for non-Haskell's sake	Andrey Dadakov	3.99
☐	Re: The Future of Haskell discussion at the Haskell Workshop	ketil@ii.uib.no (Ketil Z. Malde)	3.73
☐	Re: Haskell for non-Haskell's sake	D. Tweed	3.55
☐	Re: GUI Library Task Force	Ian Lynagh	3.53
☐	Re: Haskell for non-Haskell's sake	Joost Visser	3.51
☑	Re: Learning Haskell and FP	Paul Hudak	3.48
☐	Re: GUI Library Task Force	Manuel M. T. Chakravarty	3.46
☐	Re: The Future of Haskell discussion at the Haskell Workshop	Adrian Hey	3.42
☐	Re: Prototyping language extention	Frank Atanassow	3.39
☐	Re: Haskell Language Design Questions	Fergus Henderson	3.39
☐	Re: Syntax Extensions (and future Haskell)	b.i.mills@massey.ac.nz	3.32
☐	Release 2: Mondrian and Haskell for .NET	Nigel Perry	3.30
☑	Re: Learning Haskell and FP	George Russell	3.30
☐	Re: Syntax Extensions (and future Haskell)	nilsson@cs.yale.edu	3.27
☐	Re: Simpler Fibonacci function	Hamilton Richards	3.24
☐	RE: GHC Core Language	Andrew Tolmach	3.24
☐	Re: Haskell for non-Haskell's sake	C.Reinke	3.22

Figure 8: Result of keyword search



Keywords: fp, learning, introduction, paul, gentle, hudak, hudak@yale, functional

Title	Sender	Weight
☐ Re: Learning Haskell and FP	George Russell	105.95
☐ Re: Learning Haskell and FP	Paul Hudak	103.65
☐ Re: Learning Haskell and FP	George Russell	75.57
☐ Re: Learning Haskell and FP	Paul Hudak	68.16
☐ RE: Learning Haskell and FP	Karl M. Syring	67.61
☐ Re: Learning Haskell and FP	Jan Skibinski	66.27
☐ Re: Learning Haskell and FP	Michael Zawrotny	61.74
☐ Re: Learning Haskell and FP	Ketil Malde	61.19
☐ Re: Learning Haskell and FP	Shlomi Fish	54.64
☐ Re: Learning Haskell and FP	Patrick M Doane	47.48
☐ Re: Simpler Fibonacci function	Hamilton Richards	38.07
☐ Re: Learning Haskell and FP	Ketil Malde	37.57
☐ Re: Learning Haskell and FP	Benjamin L. Russell	37.25
☐ Re: Learning Haskell and FP	i r thomas	36.72
☐ Re: Learning Haskell and FP	Benjamin L. Russell	32.70
☐ RE: Learning Haskell and FP	Doug Ransom	32.16
☐ Re: Learning Haskell and FP	Benjamin L. Russell	31.79

Figure 9: Result of associative search

in the code of the system.

The associative search is performed by clicking the *Perform associative search* button after marking some check boxes on the left column. In the example, two mails about learning Haskell are marked. The associative search gives the relevant mails as in Fig. 9 where most mails are on the selected topic.

The system involves 7241 mails as documents and 61245 different words in the WAM database and responds to queries within 1 second, which is adequately fast as an Web-based system. Just 100 lines of Haskell code offers the complete searching application.

5 Conclusion

In this paper we proposed a framework for utilizing GETA library from functional programs. Layered interfaces offer multiple levels of abstraction and usability in contrast to the monadic interface of our previous work [3]. The algebraic interface is devoted for general database manipulations. The list interface serves familiar style of programming that involves list data structure and combinator library. The stratified interfaces are transformed in top-down manner from abstract and purely functional code to the original C functions provided by GETA. At the same time, program transformation is done for function fusion or specialization to keep the functional programs as efficient as possible.

This framework provides a practical solution to implement document processing applications with functional programs. Our experimental system exhibits the low overhead and high functionality achieved by using two high level libraries, GETA and WASH. They perform their specific jobs efficiently so that we only need to combine them accordingly from the viewpoint considering a library as an abstract set of functions. Combined with the flexibility of GETA, the interface can also be used as a convenient environment for testing various methods of document processing.

References

[1] Richard Bird. *Introduction to Functional Programming using Haskell (2nd edition)*. Prentice Hall, 1998.

[2] Sigbjorn Finne, Daan Leijen, Erik Meijer, and Simon Peyton Jones. Calling hell from heaven and heaven from hell. In *Proceedings of the fourth ACM SIGPLAN International Conference on Functional Programming*, pages 114–125, Paris, France, 1999.

[3] Takuma Murakami, Shingo Nishioka, Zhenjiang Hu, and Akihiko Takano. Scripting GETA Searching Engine in Haskell. In *Proceedings of the 20th Symposium of Japan Society for Software Science and Technology*, Nagoya, Japan, 2003.

[4] Akihiko Takano and Erik Meijer. Shortcut Deforestation in Calculational Form. In *Proceedings of the seventh International Conference on Functional Programming Languages and Computer Architecture*, pages 306–313. ACM Press, 1995.

[5] Akihiko Takano, Shingo Nishioka, Makoto Iwayama, Toru Hisamitsu, Osamu Imaichi, and Hirofumi Sakurai. Information Access Based on Associative Calculation. In *Proceedings of Theory and Practice of Informatics*, volume 1963 of *Lecture Notes in Computer Science*, pages 187–201, Milovy, Czech Republic, 2000. Springer-Verlag.

[6] Peter Thiemann. WASH/CGI: Server-side Web Scripting with Sessions and Typed, Compositional Forms. In *Practical Aspects of Declarative Languages (PADL'02)*, volume 2257 of *Lecture Notes in Computer Science*, pages 192–208, Portland, Oregon, USA, 2002. Springer-Verlag.

[7] Archive of The Haskell Mailing List. `http://haskell.org/pipermail/haskell/`.