A Probabilistic Approach to the Problem of Automatic Selection of Data Representations*

Tyng–Ruey Chuang Wen L. Hwang
Institute of Information Science
Academia Sinica
Nankang, Taipei 11529, Taiwan
E-mail: trc@iis.sinica.edu.tw, whwang@iis.sinica.edu.tw

November 25, 1995

Abstract

The design and implementation of efficient aggregate data structures has been an important issue in functional programming. It is not clear how to select a good representation for an aggregate when access patterns to the aggregate are highly variant, or even unpredictable. Previous approaches rely on compile–time analyses or programmer annotations. These methods can be unreliable because they try to predict program behaviors before they are executed.

We propose a probabilistic approach, which is based on Markov processes, for automatic selection of data representations. The selection is modeled as a random process moving in a graph with weighted edges. The proposed approach employs coin tossing at run–time to aid choosing suitable data representations. The transition probability function used by the coin tossing is constructed in a simple and common way from a measured cost function. We show that, under this setting, random selection of data representations can be quite effective. The probabilistic approach is applied to an simple example, and the results are compared to some deterministic selection algorithms.

1 Introduction

How to design and implement efficient aggregate data structures has been a major concern for both the designers and users of functional programming languages. See, for example, Chuang and Goldberg [2, 3], Okasaki [9], Schoenmakers [11] and Shao, Reppy, and Appel [13, 14]. The problem becomes more complicated if access patterns to aggregates are highly variant, or even unpredictable. A common situation occurs where there are several representations of an aggregate, with one representation being more efficient than the others for certain operations but worst for the remaining operations, and *vice versa*. Which representations should one chooses, given that there is no *a priori* knowledge of what operations, and how often, the aggregates will be mostly used for?

This is known as the data representation selection problem for very high–level programming languages. See, for example, Schonberg, Schwartz, and Sharir [12]. The goal is to determine a suitable representation for aggregates of builtin abstract data types (such as sets and arrays) such that the aggregates will exhibit good performance. The problem occurs as well for user–defined abstract data types, where there may exist multiple representations of the data type but each with different performance characteristics. Naturally, aggregates of the data type will require different representations

*This research is supported, in part, by National Science Council under contract NSC 84–2213–E–001–004. This report is achieved as Technical Report TR–IIS–95–011 at the Institute of Information Science, Academia Sinica.

in different program contexts in order to achieve good performance. Previous approaches to the data representation selection problem have relied heavily on compile–time analyses or programmer annotations to help selecting a good implementation of the data structure. These approaches can be very unreliable since they try to predict a program's behavior before it is executed.

Another approach is to design a representation for the abstract data type such that, though not the best possible in every situation, its performance is not too bad for all situations. This representation is used for *all* aggregates of the abstract data type, and designers and users of the data type now spare themselves of the problem of selecting the right representation. One drawback of this approach is that users of the abstract data type may pay for cost they do not ask for. For example, in some context the users may not use at all certain functionality of the data type. Nevertheless the performance of all aggregates of that type is degraded because they all have to accommodate this extra functionality into their representation.

This paper takes a different view of the data representation selection problem, and presents a probabilistic approach to solve the problem. We view the data representation selection problem as an on–line problem in the following way. There are several representations of an aggregate, and it costs each representation certain amount of time to process each kind of operations. These representations can be converted to on another at a cost. There is a sequence of requests consisting of various kinds of operations to be served by the aggregate. The goal is to make choices, as the requests arrive, which representation to serve the current request, and if necessary, to perform a conversion between representations, such that the total cost of serving the entire request sequence is small.

Note that we have shift the decision of making the selections from compile–time to run–time. But a run–time choice may still be inappropriate if it only relies on history of the request sequence to make the current selection. There are two reasons for this. First, history is no indication of the future — the run–time choice may just be as inaccurate as the compile–time choice. Secondly, keeping the history around increases the space requirement of program execution — the space/time overhead incurred by run–time choices may be too high to make them feasible. Probabilistic techniques have been used in on–line algorithms to avoid the above two problems. See, for example, Fiat, Karp, Luby, McGeoch, Sleator, and Young [7], Karp [8], and Raghavan and Snir [10]. Random choices, based on carefully devised principles, can often be shown to make few bad decisions in the long term. Furthermore, random choices can often be "memoryless," in the sense that they only depend on the current state of execution, but not on previous states.

Though the idea of random choices quite simple and appealing, to the best of our knowledge, we find few applications of probabilistic approaches to the data representation selection problem. In Chuang [2], a randomization technique is used to implement purely functional arrays for efficient multithreaded read/update operations, and is shown to be effective. In this paper we further develop a general framework based on probabilistic choices to solve the data representation selection problem. In Section 2 of this paper, we will describe a somewhat simple example to be used in this paper for both illustration and experimentation. Several other examples taken from the literature is described in this section as well. Section 3 will present the general framework and some preliminary analyses. An actual implementation and some experimental results are presented in Section 4. Related implementation issues are discussed in Section 5. Section 6 discusses related and future work.

2 Examples

We use a somewhat simple example thorough this paper as a demonstration, but the technique applies to others. Suppose that we want to implement an abstract data type named bag that supports just three kinds of operations: creation of an empty bag, insertion of an integer to a bag, and query to a bag for a given integer to see if it is there. Of course, we may also support deletion of an integer

from a bag, and so on. But right now let us assume there are just three kinds of operations: creation, insertion, and membership query.

A bag can be implemented as a list. The list representation provides constant time insertion by appending the inserted integer at the front of the list. A membership query, however, will takes time linear to the the length of list for the worst case. A bag can be implemented by a balanced search tree as well. Insertion and membership query then each takes logarithmic time, with respect to the number of nodes in the tree. Clearly if the bag will be used mostly for insert operations, then a list representation is preferable. If we have a large bag and the number of membership queries is huge, then we would prefer a balanced search tree representation. But what are the precise criteria for preferring one representation over the other?

The problem can be rephrased as the following. We want to serve a sequence of bag operations that starts with a create operation and followed by some number of insert and query operations. We have no *a prior* knowledge of what the sequence looks like, including the number of operations in the sequence. The problem is to decide which bag representation one should use, and, if more than one representation is preferred, when should one convert one representation to the other?

Other examples of this kind of characteristics abound in functional programming. For example, Chuang [2] discusses the performance tradeoff between two representations of arrays (one good for update operations and one good for read operations) and how to make probabilistic choices on–line. Okasaki [9] presents an implementation of "random–access lists" that is good for both list and array operations. Performance of the list operations of the random–access lists, however, are not as good as those of the straightforward list representation; so is the case for the array operations. Users of mostly list operations (or mostly array operations, for the matter) suffer from such an implementation as a result. We can use the probabilistic technique developed in this paper to mix two representations of random–access lists (but each with different performance characteristics, such as pure lists and pure arrays) to get a more adaptive representation. Note that the bag example demonstrated in this paper can be easily made into random–access lists as well.

3 Models and Analyses

We take a probabilistic view of the problem of selecting a suitable aggregate representation to serve a sequence of operations. The way the aggregate is implemented may change over time to better serve the incoming operations. In particular, we model the change of representations over time as a Markov process.

A Markov process can be described informally as a set of states and a chance process that moves around through these states. In this paper, the states of the Markov process are just the different ways an aggregate can be implemented, and our goal is to determine good transition probabilities between the states such that the total cost of serving a sequence of operations over time is small. We use Markov processes as models because of their simplicity, and because of the rich techniques developed for them in the literature. See, for example, Chung [4] and Doyle and Snell [6]. Markov processes are also "memoryless," in the sense that the probability of moving from one state to another one depends only on the current state.

The following fixes notation convention that will be used later in this paper.

s is a set of k distinct states $s_1, s_2, \ldots, s_k$. Each state represents a particular way the aggregate is implemented. It is equally well for an aggregate to be in any one state because each representation will provide the same functionality (although at a different cost, see below).

P is a k–by–k transition probability matrix for s such that, being in state s_i, it will move to state s_j

with probability $P_{i,j}$. Naturally,

$$\begin{aligned} 1 \geq P_{i,j} &\geq 0 \quad \text{for all } i,j, \text{ and} \\ \sum_{j=1}^{k} P_{i,j} &= 1 \quad \text{for all } i. \end{aligned}$$

C is a k–by–k cost matrix with

$$\infty > C_{i,j} > 0 \quad \text{for all } i,j.$$

$C_{i,j}$ is the cost of making a move from s_i to s_j. C is not necessarily symmetric.

e is a vector of length k, and e_i is the expected cost of making a move out of state s_i. That is, $e = [e_1, e_2, \ldots, e_k]^T$, and

$$e_i = \sum_{j=1}^{k} P_{i,j} C_{i,j} \quad \text{for all } i.$$

E is a k–by–k matrix, representing the expected costs of the random walks introduced by C and P. $E_{i,j}$ is the expected cost of first reaching s_j starting from s_i. We can formulate E by the following recurrent equation:

$$E_{i,j} = P_{i,j} C_{i,j} + \sum_{k \neq j} P_{i,k} (C_{i,k} + E_{k,j}) \quad \text{for all } i,j.$$

$\alpha, \beta, \gamma, \ldots$ symbolize the kinds of operations supported by the aggregate. The set ω consists of all kinds of operations.

n is the current size of the aggregate.

For example, each bag aggregate has two representations. We let state s_1 be the list representation and s_2 the balanced search tree representation. The two representations are convertible to each other and will support all bag operations. $P_{1,1}$ will be the probability that, while in the list representation, the next bag operation will be performed on the same representation, and $P_{1,2}$ the probability that the next bag operation will be performed on the balanced search tree representation (which itself is converted from the current list representation). Likewise for $P_{2,2}$ and $P_{2,1}$.

Furthermore, $C_{1,1}$ is the cost of performing a bag operation while the bag is implemented as a list. $C_{1,2}$ is the cost of converting the list representation to the balanced search tree representation, *plus* the cost of performing the bag operation on the new representation. Likewise, for $C_{2,2}$ and $C_{2,1}$. Hence, value e_1 is the expected cost of performing a bag operation while the bag is in its list representation, and $E_{1,2}$ is the expected cost of transforming the list representation of a bag into the balanced search tree representation while serving requests from the incoming operation sequence.

A bag in fact supports three kinds of operation: creation, insertion, and membership query. Let α represent the insert operation, β represent the membership query, and γ represent the create operation. Then $\omega = \{\alpha, \beta, \gamma\}$, and a request sequence starts with a γ operation and followed by interwoven α or β operations.

For now it suffices to consider n, the aggregate size, fixed, and the request sequence consisting of only one kind of operation. For example, we may concern ourselves of serving a sequence of membership queries to a bag of exactly n integers. We will later show how to extend the Markov framework to aggregates supporting multiple kinds of operations and of variant sizes.

Recall that a membership query takes logarithmic time for a balanced search tree, and it takes linear time for a list in the worst cases. If the bag is already in its balanced search tree representation, then we may want to continue to use the representation to support further membership queries. Troubles occurs when the bag is in its list representation. Should we perform membership query on the list presentation (which is costly for each query but is tolerable if there are few of them), or should we convert immediately the list representation to the balanced search tree representation and then serve all the queries on the new representation (which initially will cost $O(n \log n)$ time for the conversion, but pays off if there are many queries)? Note that this is a difficult decision to make at compile–time because often we cannot predict the number of membership queries the bag has to serve.

The idea is that, while in the list representation, the bag should gradually change to the balanced search tree representation over time to better serve membership queries. This gradual change of implementation is modeled as a Markov process where representations change according to P, the transition probability matrix. It will be desirable if the Markov process bears the following two properties:

(1) The expected cost of serving a request (while making the transition to the new representation) is comparable to the original cost of serving the request as if no conversion occurs.

(2) The expected cost of eventually converting to the new representation is comparable to the cost of an immediate conversion.

Property (1) make sure that, in the short term (*i.e.*, there are few membership queries in the sequence), the expected cost of serving a request is comparable to the case when the conversion to new representation is simply not worthy. Property (2) make sure that, in the long term (*i.e.*, there are many membership queries), the expected cost of converting to new representation is still comparable to the case where an immediate conversion is most desirable.

Formally, we can put it in the following way. Let s_i be the current state, and let s_j be the preferred state. A state s_j is *preferred* if $C_{j,j}$ is the smallest among all $C_{h,h}$, where $s_h \in \{s_1, s_2, \ldots, s_k\}$. That is, the cost of serving a request is smallest at state s_h. Then we want to (1) compare e_i (the expected cost of serving a request at state s_i) to $C_{i,i}$ (the original cost of serving the request as if no conversion occurs), and (2) compare $E_{i,j}$ (the expected cost of eventually converting to the preferred representation) to $C_{i,j}$ (the cost of an immediate conversion). In the above example of bag aggregates, we may call balanced search tree the preferred representation because it takes less time to serve a membership query, and we want to convert a bag from the list representation to the balanced search tree representation if there are many membership queries.

In general, there may be more than one preferred states. (For example, there may be several representations of a bag aggregate which serve membership queries equally well.) Assuming for the moment that the preferred states are absorbing.[1] That is, $P_{j,j} = 1$ if s_j is preferred. Let s, the set of all states, be partitioned into two subsets B and D, where B is the set of the u absorbing states, and D be the set of the remaining v non–absorbing states. Let s be reordered such that the absorbing states come before non–absorbing states.

We then write P as the following

$$P = \begin{pmatrix} I & 0 \\ R & Q \end{pmatrix}$$

where I is a u–by–u identity matrix, and 0 a u–by–v matrix with all 0. R is a v–by–u matrix representing the transition probabilities from non–absorbing states to absorbing states, and Q is a v–by–v matrix representing the transition probabilities between non–absorbing states.

[1] This requirement is not really necessary, but make easier the proof that follows.

Let e_D be a vector of length v, describing the expected costs of making a move out of the v non–absorbing states, and let E_D be a vector of length v, describing the expected costs of first reaching an absorbing state from non–absorbing states.

Lemma 3.1 $E_D = (I - Q)^{-1} e_D$ ◇

PROOF. Recall that

$$E_{i,j} = P_{i,j} C_{i,j} + \sum_{k \neq j} P_{i,k} (C_{i,k} + E_{k,j})$$

for all i, j. Because we are interested in the expected cost of first reaching *any* absorbing state from a non–absorbing state, we can simply reformulate the above as

$$\begin{aligned} E_i &= \sum_{j \in B} P_{i,j} C_{i,j} + \sum_{j \in D} P_{i,j} (C_{i,j} + E_j) \\ &= \sum_{j \in B \cup D} P_{i,j} C_{i,j} + \sum_{j \in B \cup D} P_{i,j} E_j \\ &= e_i + (PE)_i \end{aligned}$$

for all $i \in B \cup D$, where we let $e_i = E_i = 0$ for $i \in B$.

In matrix form, it follows that $E = e + PE$ and $(I - P)E = e$. Recall that

$$P = \begin{pmatrix} I & 0 \\ R & Q \end{pmatrix}$$

and we have

$$\begin{pmatrix} 0 & 0 \\ R & I - Q \end{pmatrix} \begin{pmatrix} 0 \\ E_D \end{pmatrix} = \begin{pmatrix} 0 \\ e_D \end{pmatrix}$$

It follows that $(I - Q)E_D = e_D$ and $E_D = (I - Q)^{-1} e_D$. □

The matrix $N = (I - Q)^{-1}$ is called the *fundamental matrix* for the absorbing Markov process P. The entries $N_{i,j}$ of this matrix have the following probabilistic interpretation: $N_{i,j}$ is the expected number of times that the process will be in state s_j before absorption when it started in s_i (where both s_i and s_j are no absorbing state). See Doyle and Snell [6].

Note that Lemma (3.1) show how to calculate $E_{i,j}$ for any two states $s_i \neq s_j$. We simply view s_j absorbing, and let $B = \{s_j\}$, and $D = \overline{B}$. For $E_{i,i}$, the expected cost of returning to itself when starting from state s_i, we can use the following equation:

$$E_{i,i} = P_{i,i} C_{i,i} + \sum_{j \neq i} P_{i,j} (C_{i,j} + E_{j,i}) = e_i + \sum_{j \neq i} P_{i,j} E_{j,i}$$

where e_i and $E_{j,i}, j \neq i$, are already known.

The problem remains: given C, how to construct a P such that e_i and $E_{i,j}$ are each comparable to $C_{i,i}$ and $C_{i,j}$. We use the following heuristics to construct a P from C, which we call the "local" construction:

$$P_{i,j} = \frac{\frac{1}{C_{i,j}}}{\sum_{j=1}^{k} \frac{1}{C_{i,j}}}$$

We then bound e_i and $E_{i,j}$ in the following lemma.

Lemma 3.2 Let s_j be the preferred state, $B = \{s_j\}$, and $D = \overline{B}$. Then, for the "local" construction,

$$\begin{aligned} e_i &\leq k \cdot \min_j C_{i,j} \\ E_i &\leq \frac{\hat{e}}{1-\hat{q}} \end{aligned}$$

for each $i \in D$; where k is the number of states, and

$$\begin{aligned} \hat{e} &= \max_{i \in D} e_i \\ q_i &= \sum_{j \in D} P_{i,j} \\ \hat{q} &= \max_{i \in D} q_i \end{aligned}$$

◇

PROOF. First of all, for a "local" construction, we have, for each $i \in D$,

$$e_i = \sum_j P_{i,j} C_{i,j} = \frac{k}{\sum_j \frac{1}{C_{i,j}}} = k \frac{\prod_j C_{i,j}}{\sum_k \frac{\prod_j C_{i,j}}{C_{i,k}}} \leq k \frac{\prod_j C_{i,j}}{\frac{\prod_j C_{i,j}}{\min_j C_{i,j}}} = k \cdot \min_j C_{i,j}$$

Furthermore, q_i is the probability that, when in state s_i, the next move will not reach any preferred state, and e_i the expected cost of that move. It follows that $\hat{q}$ is the upper bound of the probability that, while not in the preferred states, the next move still do not reach them, and $\hat{e}$ an upper bound of the expected cost of such a move. We then have $\frac{1}{1-\hat{q}}$ as an upper bound of the total number of times it stays at the non–preferred states when starting from them, and $\frac{\hat{e}}{1-\hat{q}}$ an upper bound for E_i, the expected cost of all the moves before reaching any preferred state, starting from a non–preferred state. □

The bound for e_i is good, but the one for E_i is quite loose. We are currently working on a better bound. For the bag example, however, there are only two states s_1 and s_2, for the list and balanced search tree representations. It follows from the above lemma that

$$P = \begin{pmatrix} P_{1,1} & P_{1,2} \\ P_{2,1} & P_{2,2} \end{pmatrix} = \begin{pmatrix} \frac{C_{1,2}}{C_{1,1}+C_{1,2}} & \frac{C_{1,1}}{C_{1,1}+C_{1,2}} \\ \frac{C_{2,2}}{C_{2,1}+C_{2,2}} & \frac{C_{2,1}}{C_{2,1}+C_{2,2}} \end{pmatrix}$$

$$e_1 = P_{1,1}C_{1,1} + P_{1,2}C_{1,2} = \frac{2C_{1,1}C_{1,2}}{C_{1,1}+C_{1,2}} \leq 2\min\{C_{1,1}, C_{1,2}\} \leq 2C_{1,1}$$

and

$$E_{1,2} = \frac{1}{1-P_{1,1}} e_1 = (\frac{C_{1,1}+C_{1,2}}{C_{1,1}})(\frac{2C_{1,1}C_{1,2}}{C_{1,1}+C_{1,2}}) = 2C_{1,2}$$

Similarly, $e_2 \leq 2C_{2,2}$ and $E_{2,1} = 2C_{2,1}$.

3.1 Aggregates with Multiple Kinds of Operations

In the bag example, after creation, each bag in fact supports two kinds of operations: insertion and membership query. Insertion is better performed in list, and membership query is more efficient in balanced search tree. In general, for each kind of operations $\phi \in \omega$, it will has its own cost matrix C^ϕ. Furthermore, when operating ϕ, the representation of the aggregate should be chosen based on a Markov process derived from C^ϕ.

Let y_i^ϕ be the cost of operating ϕ in state s_i, and $X_{i,j}$ the the cost of converting an aggregate from state s_i to state s_j. Then, by definition,

$$C_{i,j}^\phi = \begin{cases} y_i^\phi & \text{if } i = j \\ X_{i,j} + y_j^\phi & \text{if } i \neq j \end{cases}$$

From each C^ϕ, we then construct a Markov process P^ϕ to model the change of representations when operating ϕ.

3.2 Variant–Sized Aggregates

Some operations will change the size of an aggregate. Often the performance of the aggregate is affected as its size grows or shrinks. For example, an insertion makes the size of a bag grow by one. If the bag is implemented by a balanced search tree, then each subsequent insertion or query to the bag costs more time than it does to the original bag. Therefore, the cost matrix C not only is parameterized by w, the kinds of operations supported by the aggregate, it is also a function of n, the size of the aggregate. Let's write $C[n]$ for the cost matrix at size n, and $P[n]$ for the corresponding transition probability.

Notice that, in general, we want to pre–compute P such that, at the moment of serving a request, we can make a quick decision based on P to choose a suitable implementation. Since C is a function of n it will be impractical to pre–compute P for all size n. If we delay the construction of P until run–time, where n is known, then the overhead for making a probabilistic choice at run–time may be too large to render the whole scheme impractical.

As a compromise, we use the following way to estimate P: Pre–compute only $P[2^m], m \in N$. When processing a request to an aggregate of size n with $2^{m-1} < n \leq 2^m$, make a probabilistic choice based on $P[2^m]$. This estimation of $P[n]$ works out well in practice, but is biased against implementations whose sizes grow faster than the others.

3.3 The Algorithm

Given:
A specification of an abstract data type that supports operations of kind $\phi \in \omega$, and k representations $s_1, s_2, \ldots s_k$ of the abstract data type.

Preprocessing:
Measure $C^\phi[n]$ where $\phi \in \omega$ and $n = 2^m, m \in N$.
Build $P^\phi[n]$ from $C^\phi[n]$ using the "local" heuristics.

On–Line Service:
An aggregate a is requested to serve a ϕ operation. Let s_i be the current state of the aggregate, and n its size. Suppose $2^{m-1} < n \leq 2^m, m \in N$. Then

- Make a probabilistic choice based on $P_i^\phi[2^m]$, let s_j be the new state.
  - If $s_i = s_j$, then return $\phi_i(a)$.
  - Otherwise first mutate a from state s_i to state s_j, then return $\phi_j(a)$.

4 Experimentation

We have conducted an experiment, under Standard ML of New Jersey 0.93, to measure the effectiveness of the proposed approach. We implement an integer bag by two different representations: a list with all the integers in the bag, and a mapping that maps an integer to the number of times it has appeared in the bag. The map is taken from the SML/NJ 0.93 library, and is very efficiently implemented by a balanced search tree. The signature of the bag aggregate, as well as its implementations in the the list and map representations, are described in SML in Figure 1 in Appendix A.

Each representation supports three kinds of operation: insert operation (`insert`), membership query (`member`), and creation of a empty bag (`void`). In addition, each representation also supports the following operations: `size` that returns the size of a bag, and `list2bag` and `bag2list` that convert between an integer bag and an integer list. (Do not confuse this list to the list representation of a bag. See Figure 1 for details.) The function `size` is used to determine which cost matrix C (hence, which transition matrix P) to use. It is a constant time function. The function `list2bag` and `bag2list` mediate between the list and map representations for conversion purposes.

The performance of the two representations is measured by a separate program by timing the execution time of insertions and queries, each for aggregates of different sizes. The data is shown in Table 1, and is used to construct cost matrices $C[n]$. A functor is then written to accept the two representations (as well as their performance data), build the cost matrices, construct the transition probability matrices, and produce a representation that (based on algorithms outlined in Section 3.3), when serving an insertion or a query, will make a probabilistic choice on whether or not to first perform a conversion. The skeleton of the functor is shown in Figure 2 in Appendix A.

We run a set of simple benchmarks to evaluate the performance of the probabilistic scheme. The results are shown in Table 2. The probabilistic scheme is never the fastest. However, its performance is between those of map and list representations, except in a benchmark — $(\alpha^{990}\beta^{10})^{20}$ — where it is worse than both. There is a simple explanation: in this case both the map and list representations happen to run the benchmark using about the same time, while the probabilistic scheme pays additional overhead (such as generating random numbers and performing conversions) for making run–time choices.

We also observe that the map library provided by SML/NJ 0.93 is very efficient; it is fastest except in only two occasions where it loses to the list representation. Notice that in the two occasions, the probabilistic scheme also beats the map representation. The greedy algorithm, which always converts to the preferred state when serving a request, often performs badly because the conversion cost cannot be amortized by the subsequent (short) sequence of operations of the same kind. We may make the following likely implication: In order for compile–time analyses or programmer annotations to be effective in making dynamic selection of data representations, the conversion costs between representations must be taken into account.

5 Discussion

We face several problems when performing the experimentation. First of all, it is really a tedious job to make an accurate measurement of the cost matrices. (That is also one of the reason why we have not experimented with aggregate that supports more than two kinds of operations, and which also has more than two different representations.) For example, the least measurable unit of time in SML/NJ 0.93 is 0.01 second, and we have to repeatedly run the measurement code and divide the accumulated time. We also have to discount the time spent on spurious activities in the measure code, such as the generation of sample operation sequences and the skeleton loops to carry out the operations.

We also find the floating point support in SML lacking. For example, it will be really nice to be

kind	aggregate size					
sec./op.	2^5	2^6	2^7	2^8	2^9	2^{10}
insert	3.51×10^{-6}	3.51×10^{-6}	3.70×10^{-6}	3.53×10^{-6}	3.59×10^{-6}	3.59×10^{-6}
member	1.10×10^{-4}	2.00×10^{-4}	3.61×10^{-4}	7.62×10^{-4}	1.52×10^{-3}	3.75×10^{-3}
bag2list	2.90×10^{-6}	3.13×10^{-6}	2.75×10^{-6}	2.75×10^{-6}	2.87×10^{-6}	2.78×10^{-6}
list2bag	1.46×10^{-5}	1.46×10^{-5}	3.91×10^{-5}	5.86×10^{-5}	1.17×10^{-4}	2.34×10^{-4}

kind	aggregate size					
sec./op.	2^{11}	2^{12}	2^{13}	2^{14}	2^{15}	2^{16}
insert	3.52×10^{-6}	3.55×10^{-6}	3.52×10^{-6}	3.55×10^{-6}	3.59×10^{-6}	4.23×10^{-6}
member	7.81×10^{-3}	1.59×10^{-2}	3.38×10^{-2}	6.38×10^{-2}	1.30×10^{-1}	2.88×10^{-1}
bag2list	2.81×10^{-6}	2.85×10^{-6}	2.84×10^{-6}	2.96×10^{-6}	2.38×10^{-6}	2.83×10^{-6}
list2bag	4.69×10^{-4}	9.38×10^{-4}	3.13×10^{-3}	7.50×10^{-3}	1.50×10^{-2}	2.75×10^{-2}

(a) For the list representation.

kind	aggregate size					
sec./op.	2^5	2^6	2^7	2^8	2^9	2^{10}
insert	6.47×10^{-5}	7.81×10^{-5}	8.76×10^{-5}	9.89×10^{-5}	1.14×10^{-4}	1.26×10^{-4}
member	1.20×10^{-5}	1.19×10^{-5}	1.34×10^{-5}	1.59×10^{-5}	1.38×10^{-5}	1.83×10^{-5}
bag2list	1.81×10^{-4}	3.76×10^{-4}	7.23×10^{-4}	1.43×10^{-3}	2.81×10^{-3}	7.34×10^{-3}
list2bag	1.09×10^{-3}	2.62×10^{-3}	5.92×10^{-3}	1.35×10^{-2}	3.10×10^{-2}	9.02×10^{-2}

kind	aggregate size					
sec./op.	2^{11}	2^{12}	2^{13}	2^{14}	2^{15}	2^{16}
insert	1.36×10^{-4}	1.45×10^{-4}	1.45×10^{-4}	1.57×10^{-4}	1.73×10^{-4}	1.94×10^{-4}
member	7.17×10^{-6}	1.67×10^{-5}	2.40×10^{-5}	2.36×10^{-5}	1.54×10^{-5}	2.22×10^{-5}
bag2list	1.50×10^{-2}	3.03×10^{-2}	6.13×10^{-2}	1.25×10^{-1}	2.43×10^{-1}	5.00×10^{-1}
list2bag	2.15×10^{-1}	4.95×10^{-1}	1.08×10^{0}	2.42×10^{0}	5.28×10^{0}	1.15×10^{1}

(b) For the map representation.

Table 1: Performance measurement for both list and map representations of bag aggregates.

NOTE. The figure is measured by using the `System.Timer` structure of SML/NJ 0.93, on a 40 MHz SPARC workstation with 32 MB memory. Only user time is reported; garbage collection and system times are not measured. (SML/NJ 0.93 garbage collects at indefinite time.) Considerable care has been taken to make a (more or less) accurate measurement. However, fluctuations remain.
The bag elements are randomly drawn from the integer set $\{0, 2, \ldots, 2^{29} - 1\}$. In the above tables the time in entry 2^m is the average of the times for aggregate of sizes from $2^{m-1} + 1$ to 2^m, if m is small. If m is large, the times in the entry is the average of times for aggregate of sizes from $2^m - c$ to 2^m for some constant c. From the two tables, we can construct the cost matrices C's. For example,

$$\begin{aligned}
C^{\alpha}_{1,1}[2^5] &= 3.51 \times 10^{-6} \\
C^{\alpha}_{1,2}[2^5] &= 2.90 \times 10^{-6} + 1.09 \times 10^{-3} + 6.47 \times 10^{-5} \\
C^{\alpha}_{2,1}[2^5] &= 1.81 \times 10^{-4} + 1.46 \times 10^{-5} + 3.51 \times 10^{-6} \\
C^{\alpha}_{2,2}[2^5] &= 6.47 \times 10^{-5}
\end{aligned}$$

sequence	representation			
time in sec.	map	list	greedy	"local"
$(\alpha_{0.999}\|\beta_{0.001})^{10000}$	1.34	0.39	13.84	0.79
$(\alpha_{0.5}\|\beta_{0.5})^{10000}$	0.83	50.62	929.90	13.63
$(\alpha_{0.1}\|\beta_{0.9})^{10000}$	0.25	17.69	56.35	4.49
$(\alpha_{\frac{t-\ln t}{t}}\|\beta_{\frac{\ln t}{t}})^{10000}$	1.50	0.39	11.99	0.73
$(\alpha^{1000}\beta^{1000})^{10}$	1.64	219.16	9.16	14.15
$(\alpha^{990}\beta^{10})^{20}$	3.04	8.24	36.01	14.29
$(\alpha^{10}\beta^{990})^{20}$	0.30	7.86	1.00	3.17

Table 2: Performance of various representations of bag aggregates for some simple benchmarks.

NOTE. For request sequences, we use the notation that, for example, $(\alpha_{0.999}|\beta_{0.001})^{10000}$ is a sequence of 10000 requests which, at any moment, an α operation occurs with probability 0.999 and a β operation occurs with probability 0.001. The probability an operation occurs may also depend on its ordinal number t in the sequence, like, $(\alpha_{\frac{t-\ln t}{t}}|\beta_{\frac{\ln t}{t}})^{10000}$. If there is no subscript, then the operation always occurs.
We compare the performance of the following four representations: the one that always uses the map representation, the one that always uses the list representation, the one that always converts to the preferred representation when performing an operation, and the one that uses the "local" transition probability. Only user time is reported; garbage collection time and system time are not measured.

able to express numbers like `Inf` and `NaN`, which are in the ANSI/IEEE Standard 754-1985. That will make easier the task of stating certain conversions between representations are impossible (*i.e.*, $C_{i,j} = \infty$).

We use the `ref` data type in SML to make mutable representation of an aggregate. This is not a problem *per se*, but is troublesome if the bag data type wants to be polymorphic to its element type. The SML typing rules will then insist a weak type variable for the bag element. Though in general this is the right thing to do, we do not see it is necessary in this context. All usages of the assignment operator `:=` are in the `Mix` functor in Figure 2, and it can be shown that the types of the new value and the old value are always the same. Nothing will go wrong there.

Last but not least, notice that the probabilistic scheme is also good for multithreaded aggregate accesses. The analysis in Section 3 depends only on the current state of the aggregate, not on any previous states. Also notice that we mutate the aggregate when making a conversion; hence repeated accesses to the aggregate will not need the same conversion again.

6 Related and future works

The problem of automatic selection of data representations, when put in a probabilistic framework, is closely related to the problem of random walks in a weighted graph. See for example, the important work of Borodin, Linial, and Saks [1], Coppersmith, Doyle, Raghavan, and Snir [5], and Doyle and Snell [6]. They often assume the cost matrices are symmetric, and use more complicated techniques to derive tight bounds of the probabilistic schemes involved. The competitive paging problem and its probabilistic solution of Fiat, Karp, Luby, McGeoch, Sleator, and Young [7] is related to the data

representation selection problem as well, though it also assumes symmetric cost matrices.

More analyses are needed for the "local" heuristics for building the transition probability matrix. In general it is not clear how well it performs when compared to an off–line optimal algorithm. We also need to exploit other construction of the transition probability matrix, perhaps by using techniques of multivariable constraint optimization.

Right now we also assume that, for aggregates of the same size, all operations of a given kind will cost the same amount of time. In general this is not true, and we need to look into this issue.

References

[1] Allan Borodin, Nathan Linial, and Michael E. Saks. An optimal on–line algorithm for metrical task system. *Journal of the Association for Computing Machinery*, 39(4):745–763, October 1992.

[2] Tyng–Ruey Chuang. A randomized implementation of multiple functional arrays. In *Proceedings of 1994 ACM Conference on Lisp and Functional Programming*, pages 173–184. Orlando, Florida, USA, June 1994. The proceedings also appears as *Lisp Pointers*, Volume VII, Number 3, July–September 1994. ACM Press.

[3] Tyng–Ruey Chuang and Benjamin Goldberg. Real–time deques, multihead Turing machines, and purely functional programming. In *Conference on Functional Programming Languages and Computer Architecture*, pages 289–298. University of Copenhagen, Denmark, June 1993. ACM Press. [This paper was reviewed in *ACM Computing Reviews*, 9408–0543, August 1994, page 425].

[4] Kai Lai Chung. *Elementary Probability Theory with Stochastic Processes*. Undergraduate Texts in Mathematics. Springer–Verlag, 1974.

[5] Don Coppersmith, Peter Doyle, Prabhakar Raghavan, and Marc Snir. Random walks on weighted graphs and applications to on–line algorithms. *Journal of the Association for Computing Machinery*, 40(3):421–453, July 1993.

[6] Peter G. Doyle and J. Laurie Snell. *Random Walks and Electric Networks*, volume 22 of *The Carus Mathematical Monographs*. The Mathematical Association of America, 1984.

[7] Amos Fiat, Richard M. Karp, Michael Luby, Lyle A. McGeoch, Daniel D. Sleator, and Neal E. Young. Competitive paging algorithms. *Journal of Algorithms*, 12:685–699, December 1991.

[8] Richard M. Karp. An introduction to randomized algorithms. *Discrete Applied Mathematics*, 34(1–3):165–201, November 1991.

[9] Chris Okasaki. Purely functional random–access lists. In *SIGPLAN–SIGARCH–WG2.8 Conference on Functional Programming Languages and Computer Architecture*, pages 86–95. La Jolla, California, USA, ACM Press, June 1995.

[10] P. Raghavan and M. Snir. Memory versus randomization in on–line algorithms. *IBM Journal of Research and Development*, 38(6):683–707, November 1994.

[11] Berry Schoenmakers. *Data Structures and Amortized Complexity in a Functional Setting*. PhD thesis, Department of Mathematics and Computing Science, Eindhoven University of Technology, September 1992.

[12] E. Schonberg, J. T. Schwartz, and M. Sharir. An automatic technique for selection of data representations in SETL programs. *ACM Transactions on Programming Languages and Systems*, 3(2):126–143, 1981.

[13] Zhong Shao and Andrew W. Appel. Space–efficient closure representations. In *Proceedings of 1994 ACM Conference on Lisp and Functional Programming*, pages 150–161. Orlando, Florida, USA, June 1994. The proceedings also appears as *Lisp Pointers*, Volume VII, Number 3, July–September 1994. ACM Press.

[14] Zhong Shao, John H. Reppy, and Andrew Appel. Unrolling lists. In *Proceedings of 1994 ACM Conference on Lisp and Functional Programming*, pages 185–195. Orlando, Florida, USA, June 1994. The proceedings also appears as *Lisp Pointers*, Volume VII, Number 3, July–September 1994. ACM Press.

A Code

```
signature BAG =
sig
    type Bag
    val void:   unit -> Bag
    val insert: int * Bag -> Bag
    val member: int * Bag -> bool

    val size:     Bag -> int
    val bag2list: Bag -> int list
    val list2bag: int list -> Bag
end

functor BagByList () : BAG =
struct
   type Bag = int * int list
   fun void   ()          = (0,   [])
   fun insert (k, (n, l)) = (n+1, k::l)
   fun member (k, (_, l)) = exists (fn h => h = k) l

   fun size     (n, _) = n
   fun bag2list (n, l) = l
   fun list2bag     l  = (length l, l)
end

functor BagByMap (Map: INTMAP) : BAG =
struct
local
   fun conses (k, 0, l) = l
     | conses (k, n, l) = conses (k, n-1, k::l)
in
   type Bag = int Map.intmap
   val void = Map.empty
   fun insert (k, b) = case Map.peek (b, k) of
                            NONE   => Map.insert (b, k, 1)
                          | SOME n => Map.insert (b, k, n+1)
   fun member (k, b) = case Map.peek (b, k) of
                            NONE   => false
                          | SOME _ => true

   val size       = Map.numItems
   fun bag2list b = Map.fold conses b []
   fun list2bag l =     fold insert l (void ())
end
end
```

Figure 1: The signature for Bag, and its two representations.

NOTE. The integer map library (the signature is `INTMAP`, and the structure `IntMap`) from SML/NJ 0.93 is used to implement the `BagByMap` representation of `Bag`. The integer maps are implemented by trees of bounded balance; please consult SML/NJ 0.93 documentation for details. Notice that, if `structure U = BagByList()` and `structure V = BagByMap(IntMap)`, then `V.list2bag o U.bag2list` converts a bag from its list representation to the map representation, and `U.list2bag o V.bag2list` converts a bag from its map representation to the list representation.

```
signature BAG_COST_MATRIX =
sig
    val cVoid:     real
    val cInsert:   real vector
    val cMember:   real vector
    val cBag2list: real vector
    val cList2bag: real vector
end

signature TWO_BAGS =
sig
   structure U: sig  structure Bag: BAG; structure CostMatrix: BAG_COST_MATRIX  end
   structure V: sig  structure Bag: BAG; structure CostMatrix: BAG_COST_MATRIX  end
end

functor Mix (TwoBags: TWO_BAGS): BAG =
struct
   open TwoBags

   datatype Union = U of U.Bag.Bag
                  | V of V.Bag.Bag
   type Bag = Union ref

   fun size (ref (U bag)) = U.Bag.size bag
     | size (ref (V bag)) = V.Bag.size bag
   fun state (ref (U _)) = 0
     | state (ref (V _)) = 1
   fun conv (b as ref (U bag)) = (b := V (V.Bag.list2bag (U.Bag.bag2list bag)); b)
     | conv (b as ref (V bag)) = (b := U (U.Bag.list2bag (V.Bag.bag2list bag)); b)
   fun plainInsert (k, b as ref (U bag)) = ref (U (U.Bag.insert (k, bag)))
     | plainInsert (k, b as ref (V bag)) = ref (V (V.Bag.insert (k, bag)))

   val  ... P^insert ...  =
        ... constructed from C^insert, which is built from
            U.CostMatrix and V.CostMatrix ...

   fun insert (k, b) =
       case  ... the new state, which is decided by coin tossing and
                 based on P^Insert_(state b)[size b] ...  of
           0 => plainInsert (k, b)
         | 1 => plainInsert (k, conv b)

   .....
end
```

Figure 2: A skeleton of the SML code that mixes two representations of the bag aggregate into one.

NOTE. Actually, we do not use coin tossing if the aggregate size is small ($\leq 2^4$). The operation will always uses the current representation. Also, a `void` operation returns with equal probability an empty bag of either one of the two representations.