J. Functional Programming 5 (3): 317–343, July 1995

Funser: a functional server for textual information retrieval †

DONALD A. ZIFF
Department of Computer Science, University of Chicago, IL, USA
(email: ziff@cs.uchicago.edu*)*

STEPHEN P. SPACKMAN
Computational Linguistics, DFKI Saarbrücken, Germany
(email: stephen@acm.org*)*

KEITH WACLENA
University of Chicago Library, Chicago, IL, USA
(email: k-waclena@uchicago.edu*)*

Abstract

This paper describes a data-intensive application written in a lazy functional language: a server for textual information retrieval. The design illustrates the importance of interoperability, the capability of interacting with code written in other programming languages. Lazy functional programming is shown to be a powerful and elegant means of accomplishing several desirable concrete goals: delivering initial results promptly, using space economically, and avoiding unnecessary I/O. Performance results, however, are mixed.

Capsule Review

The re-implementation of a component of a real application in a lazy functional language is described. The component, Funser, performs the data-intensive task of textual information retrieval. The perceived advantages of a non-strict program are that initial results can be delivered to the user quickly, and both unnecessary work and space allocation can be avoided.

At the time the construction of Funser was begun, the authors had first to implement their own language to interoperate with existing foreign-language application components. The functional Funser achieves the stated goals: initial results are produced promptly, space-usage is good and overall execution time is comparable with, and often better than, the original imperative implementation. Users appreciate the prompt delivery of results.

Unfortunately, absolute performance is disappointing, and a shell script Funser-emulator was written which consistently outperforms the functional Funser. The script delivers early results into a file that can be browsed by the user. The major reason for the poor performance is that the interpreter is very slow: approximately 6 times slower than the Gofer interpreter.

† Programs and documentation related to this work are available at the following addresses:

- ftp://cs.uchicago.edu/pub/software/funser
- gopher://cs-gopher.uchicago.edu/11/software/funser
- http://cs-www.uchicago.edu/software/funser

In effect the functional program is a prototype for the shell-script implementation which is now used in the application.

1 Introduction

Is it possible to write a 'real' application in a pure functional language? This question has been asked many times, at conferences, on Internet newsgroups, and in papers, and perhaps the answer is beginning to be yes. This paper describes work that attempts to answer the question in the context of full-text information retrieval; for this kind of 'real' application, we have found the answer to be, for now, maybe.

The researchers at the Center for Information and Language Studies (CILS) at the University of Chicago had a unique opportunity to explore this issue. From 1990–1992, CILS was also the home of ARTFL (American and French Research on the Treasury of the French Language), a cooperative project between the University of Chicago and France's *Centre National de Recherche Scientifique*. It distributes access to the *Trésor de la Langue Française* database (TLF), a 700 Mbyte collection of full texts of French literature (Morrissey and del Vigna, 1983). CILS researchers had already developed and were supporting ARTFL's primary means of giving dial-up and Internet users access to the TLF: the 'PhiloLogic' (ARTFL, 1989) system. This system had been available to ARTFL subscribers since 1989, and attracted a steady following.

There were several strong reasons to consider the possibility of replacing this system. First, there were certain features that the old system lacked. Second, the ARTFL project had reached an agreement with their French collaborators to embark on a project to demonstrate the feasibility of a CD-ROM based system. Since the raw size of the database is around 700 MByte, slightly greater than the capacity of a CD, and indices typically range in size from half to the equal of the bulk of the database itself, it was clear that special attention would have to be paid to data compression. Third, and partly in response to this, CILS researchers had done significant work on data compression for textual information retrieval systems (Bookstein *et al.*, 1991). Fourth, an independent CILS project, a system for manipulation of text as a collection of abstract structured objects, was ready for testing (Deerwester *et al.*, 1992). Finally, there had been continuing interest at CILS in using lazy functional programming techniques for textual information retrieval (Deerwester *et al.*, 1990). It was natural to attempt to unite all these efforts in a single system on which the ARTFL Project's base of users could then be set loose.

2 System architecture

With this opportunity came several important constraints that shaped the project. Most important of these was the necessity of distributing the task of implementation among the available personnel, using different implementation languages as appropriate. It would have been impractical (and perhaps impossible) to require that all the programming be done in a lazy functional language. First, as we shall

https://doi.org/10.1017/S0956796800001386 Published online by Cambridge University Press

see in more detail, it was not clear that a lazy functional programming language implementation could be found at that time which met the requirements of even the part of the project to which its proponents thought it most naturally suited: the retrieval engine itself. Second, there were perhaps even greater difficulties in matching a lazy language to other parts of the project. The compression algorithms, for example, relied heavily on bit-field manipulation and array access, both of which would have to be implemented extremely efficiently in a real system. Moreover, the backgrounds of the programmers involved, and the existing software resources they could draw on, pointed towards encouraging each group to use the implementation language they found most natural to the task at hand, while maintaining a rigid discipline of abiding by clean interfaces designed in advance.

It should be noted, however, that the motivation for the use of other programming languages was not the desire for impure features. One of the reasons that textual information retrieval is so well suited to pure functional programming is that the database is essentially static. There is no question of desiring side-effects. In fact, from the system designers' perspective, other programming language implementations were used to stand in for future pure functional implementations. Our goal was, eventually, to write the entire system in a pure functional language; so even those components which were not so written initially were designed to be compatible with this approach.

In such a context, the pure functional style would seem to be the least common semantic denominator for interfacing, and the only option for future migration. But in fact, bringing the functional programming perspective to the interfaces proved to be of more direct benefit. Designing as though each side of each interface was to be implemented in a pure functional language, regardless of whether this was the immediate plan for either, we found that the elegance of functional interfacing techniques provided a robustness and power that greatly smoothed the overall implementation process.

The architecture of the new system was also profitably simplified and controlled by retaining the top-level structure of the original system, which had two major components: a user interface (the 'PhiloLogic' program itself, known as 'Philo'), and an underlying retrieval server, 'Philis' (ARTFL, 1989a). In fact, other user interfaces also communicated with Philis, but Philo was the most heavily used, and it is the only one with which we shall be concerned here. Philis and Philo were implemented as separately compiled programs, communicating via network link; the interface between the two was a desk-calculator style textual command language.

Building on this foundation, the redesign effort could center on Philis, the server, rather than on the user interface, Philo. The initial design strategy was to provide a compatible reimplementation of Philis. In fact, since new features were introduced, some changes to Philo were eventually necessary to take full advantage of the new system.

Within Philis, we may distinguish the following components:

- The retrieval engine, called the *Funser* for *fun*ctional *ser*ver.
- The index access method: in this implementation, the GNU project's widely available 'gdbm' hashed indexing package (FSF, 1989).

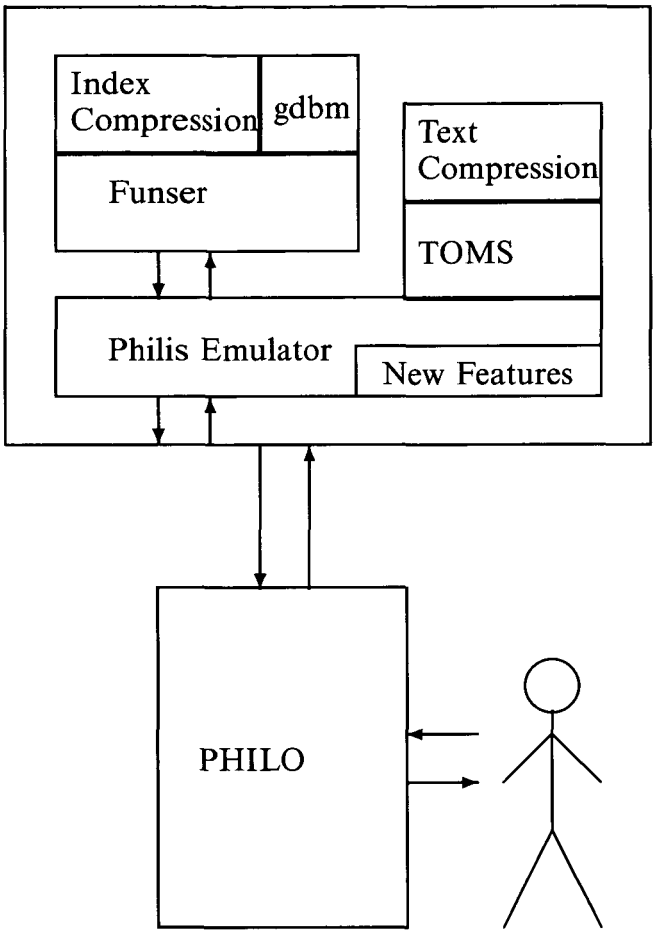

Fig. 1. System architecture overview

- The index compression (properly, decompression) software, by means of which the stored indices can be read from the CD.
- The Textual Object Management System (or TOMS), used for resolving encoded index entries into pointers into abstract text files.
- The text decompression software, which further resolves these pointers into textual input from the CD, expanding it as it is read in.
- The extensions module, which services certain special queries that need not concern us here.
- The Philis emulator, which reads commands in the Philis language, dispatches the work to the other components, and assembles the results.

Figure 1 is a schematic diagram of the system. In this diagram, arrows denote

https://doi.org/10.1017/S0956796800001386 Published online by Cambridge University Press

an interprocess communication link; adjacency denotes communication via direct procedure call.

In the present paper, we are concerned primarily with the Funser, the portion of the system that was implemented with a lazy functional language. Thus we will be especially interested in the nature of its communication with those components with which it interacts directly, and we need not concern ourselves much with the details of the other components.

3 Database and query model

To understand the task of the Funser, we must first describe the database model used. For retrieval purposes, the database is modeled by a tree of *textual objects* which correspond more or less to the natural structure of the texts: thus a database contains documents, each document contains chapters, chapters contain sentences, etc. Word occurrences are associated with their paths in the database tree; these in turn are represented as tuples of integers, called *coordinates.* Thus $\langle 10, 4, 2, 22 \rangle$ is associated with the twenty-second word of the second sentence of the fourth chapter of document ten (the ARTFL database, unfortunately, does not contain paragraph markings).

Note that the hierarchy is of uniform depth throughout the database. This is by design; the section mark-up collapses deeper structures such as the acts and scenes of a play. Thus, given the coordinate tuple of a textual object, one knows its type immediately from its length; word objects, for example, always correspond to quadruples, and sentences to triples. Using this model, it is simple to decide whether two words are in the same sentence; one just tests to see if the leading coordinates are the same. If they are, it is also easy to determine how many words they are apart, by subtracting the last coordinate. However, it is not easy to determine, from their coordinates, how far apart two arbitrary words are. These facts are reflected in the query model.

Let us note in passing that the coordinates for a word occurrence are originally created, when the database is loaded, by a separate application which interacts with the text through the TOMS. Hit-lists – the internal results of queries – are lists of these coordinates, and can be translated back to disk addresses and actual text using the TOMS, which in turn relies on our text decompression software. Coordinate lists known as *concordances* are what is actually stored, in compressed form, in each word's index entry, and these provide the ultimate basis for all of the Funser's computations.

Our query model is a variant of that discussed by Choueka *et al.* (1987), and is a generalization of that implied by the Philis command language. We consider two query types: the *co-occurrence search* and the *phrase search.* The co-occurrence search takes a tuple of sets of words $\langle \mathscr{A}_1, \mathscr{A}_2, \ldots \mathscr{A}_k \rangle$, where each $\mathscr{A}_i$ represents a set of words which are considered synonymous for that query. The solution is a list of k-tuples of coordinate tuples, $t_1 = \langle c_{1,1}, c_{1,2}, \ldots c_{1,k} \rangle, t_2 = \langle c_{2,1}, c_{2,2}, \ldots c_{2,k} \rangle, \ldots$, such that each $c_{i,j}$ is the coordinate tuple for the occurrence of some word in $\mathscr{A}_j$, and all

https://doi.org/10.1017/S0956796800001386 Published online by Cambridge University Press

the occurrences in each t_i differ only in the last position, and hence fall in the same sentence.

A phrase search is a variation of a co-occurrence search. The input and the solution are formally the same, with the additional restriction that each t_i consists of coordinates of words that immediately follow each other in a sentence. Thus the query is a request to find all the sentences in which a given phrase occurs, allowing perhaps for grammatical or semantic variants. In practice, wild-carding is permitted; formally, this is a shorthand for one of the sets $\mathscr{A}_i$ being the set of all words. In this work we refer to the sets $\mathscr{A}_i$ as disjunctions, and to the entire query as a conjunction of disjunctions. The main task of the Funser, then, is solving queries of these two kinds.

4 Why a lazy functional language?

It was the opinion of the system designers that lazy functional programming languages would be particularly appropriate for implementing this kind of retrieval. There were several potential benefits we hoped to demonstrate through the use of such a language:

- To do no unnecessary work. With a lazy language, we hoped that the evaluation mechanism could be exploited to save work at run-time, while leaving the program clean, clear and elegant. In our implementation, we concentrated on minimizing disk I/O; we attempt to read a block of information only if at least part of that block is needed to answer a query. Buneman *et al.* (1982) explored a similar approach in connection with relational database systems.
- To deliver initial results promptly. It is a commonplace of user-centered design that the most important speed measure of any system is how long it takes to deliver its first result to the user. If the user can be given an initial result to look at while the rest of the result is being computed, then the perceived performance of the system will be enhanced. In a database system, early initial results can be particularly important, since it is frequently only after seeing a few results that the user realizes that the query was badly posed. Again, this effect could be achieved using any programming language, with sufficient programming effort; it was our hope that lazy evaluation would provide it without any effort at all.
- To use memory economically. Friedman and Wise (1976) point out, in one of the earliest papers on lazy evaluation, that one potential benefit of lazy evaluation might be space efficiency. In a computation whose result is a list, a lazy evaluator would spend the space necessary to produce just the head of the list; then, only after a consumer had examined the head (so its space could presumably be reclaimed), would it expend the space to evaluate the next element, and so on. This is contrasted with an eager evaluator, which would use all the space (and time) necessary to calculate the entire list before allowing the consumer to examine any of it. With a 700 Mbyte database, this can be a disaster. Again, this effect could be achieved using any programming language, but with a lazy system, it could in principle be achieved without any special programming effort.

5 Index interface

One of the most important design questions was the choice of index interface. Both the index decompression software and the associational mechanism were implemented in C; how should they be packaged to interact with a pure functional system? One way to view the range of possibilities we confronted is in terms of granularity: how much work is accomplished, and how much data manipulated, by single primitives. There are two extremes:

- *Fine grain.* We might consider primitives at the lowest level appropriate to the data being decompressed; since each index entry contains a list of coordinates, we might consider primitives operating at the coordinate level.
- *Coarse grain.* We might consider a single index primitive, taking a word as an argument and evaluating to its entire concordance, the list of all the coordinates for all of its occurrences.

Each approach has advantages and disadvantages. Speaking for fine grained primitives is the observation that much work might be avoided through the lazy evaluation paradigm. The coarser the grain, the more work each primitive does, and the more likely it is that we didn't really need all of it. Similarly, lazier retrieval entails an overall reduction in the amount of I/O performed.

On the other hand, to achieve the laziest decompression of our index files, we would have had to write those algorithms entirely in the functional language – which as we have already noted might pose insurmountable efficiency problems, given the nature of the calculations – or set up an elaborate family of primitives which map to small amounts of work accomplished outside of the language. Similarly, a fine-grained interface would have limited our ability to interface with existing code libraries. In particular, while we initially employed the widely available gdbm hashed indexing package (FSF, 1989) as our associational mechanism, we wanted to be free to experiment with other techniques, such as the perfect hash code system developed by Fox *et al.* (1992) (whose developers tested it on keys drawn from the ARTFL database). Finally, we were driven towards coarser granularity by our desire to conceal our decompression algorithms and our associational mechanism.

Fortunately (or so we thought), there is a natural middle ground. Since the smallest physical unit in which the operating system can perform I/O is the disk block, there is a sharp knee in the savings resulting strictly from lazier input at this scale. At the same time, holding approximately a thousand entries, a disk block is large enough to amortize the setup time for decompression and any fragmentation losses that arise from discarding all compression state on block boundaries (thus simplifying a block-based interface). These interacting concerns led us to base our index access interface closely on the actual layout of the index on disk.

Physically, our index consists of two portions: an associative file and a concordance file. The associative file is managed by the associational mechanism: in this case, gdbm. Although, in principle, gdbm allows any amount of data to be associated with a key, performance is better in practice if the amount of data stored with a key is limited. Since some words are extremely frequent, their concordances are

stored indirectly, with the coordinates kept in one or more blocks of an external concordance file, with only less common words stored directly in the associational file. Our goal with the indirectly stored entries was then to minimize the number of external concordance blocks actually read.

Thus, from the C language side, the index interface looks like this. There are two entry points: one for access to the word look-up mechanism, and one for reading concordance blocks. The word look-up mechanism takes a single word (a C string) as input and returns differently depending on the word's index type. There are three cases:

1. **Not Found.** If the word does not exist in the index, a special flag value is returned.
2. **Immediate Concordance.** If the word is rare, so that its concordance is stored directly in the associative file, then its entire concordance is decompressed and placed in memory.
3. **Indirect Concordance.** If the word is frequent, its sorted concordance is stored indirectly in some number of concordance blocks. In this case, the associative file contains the concordance starting address and a block directory for the word; that is, for each block, it has the *coordinates* of the first occurrence in that concordance block. In this case, this directory is placed in memory.

The concordance-block reading mechanism takes as input the information from an indirect concordance directory entry. It decompresses the entire concordance block for that word, and puts all the resulting coordinates in memory. This can result in the transfer of a fair amount of data, 10 Kbytes or so. This is exactly the operation we wanted to minimize. In principle, using the directory, it should be possible to plan which blocks are actually needed for a given query.

This description of the C interface conceals some implementation details; for example, the input to the concordance-block reading mechanism. Concordance blocks are not always full; they are packed so that more than one word may have its concordance in a given block. Therefore the concordance-block reader requires two arguments: a block number and an offset at which to begin reading. The end of a concordance is flagged in the concordance file, so that the decompression algorithm knows to stop at the end of the block or the end of the concordance for that word, whichever comes first.

In addition, there are actually two classes of indirect concordance. Roughly half of the 450,000 or so words in the index are rare enough that direct concordances may be used. Almost all of the remaining words are stored indirectly as described above. The hundred or so most frequent words (which many systems just fail to index) are treated differently; for these *hyper-frequent* words, different concordance block compression tables are appropriate, and since even the directory of concordance blocks is too large to fit comfortably in the index file, their concordance is made doubly indirect. The index file contains only the total frequency and a pointer to a directory file. A second disk access is needed to read the directory from this file. Therefore, although the difference in layout is concealed by the C index interface,

```
interface Indexm where {

    type Word = [Char];
    data Hit = HitCons Int Int Int Int;
    type HitBlock = Array Int Hit;
    type Freq = Int;
    data IndexResult = NotFound |
                       Immed HitBlock |
                       Indir Freq [(Hit, HitBlock)];
    indexm :: Word -> IndexResult;

}
```

Fig. 2. Interface to index, as a Haskell module

the concordance-block reader actually takes one more argument; a flag indicating whether this word is hyper-frequent.

Given this C interface, the problem becomes how to present it to a lazy functional language. We shall discuss this in detail when we discuss implementation. For now, so that we may concentrate on retrieval algorithms, let us imagine that the implementation language were Haskell (Hudak *et al.*, 1991). How should the functionality just described be presented as a module?

We chose the interface shown in figure 2: access to the index is given through the `indexm` function: it takes a word and returns a value of type `IndexResult`. This type models the three possible responses from the associational mechanism. `NotFound` is just a flag value. `Immed` bears the actual concordance as a `HitBlock`, which is an array (indexed by integers) carrying `Hit`; note that, in this case, the frequency of the word is available as the size of the array. The third case, `Indir` is more complicated. In this case, the total frequency of the word, `Freq`, is directly available as the first component of the constructor. The second component provides the directory and the concordance, represented as a list of pairs, `Hit` and `HitBlock`. The `Hit` is guaranteed to be the same as the first array element in the corresponding `HitBlock`, so it is redundant semantically. However, in implementation, evaluation of the `indexm` function only forces the first elements of this list of pairs into memory. The second elements, the concordance blocks, remain as suspended calls to the C concordance reader. Thus, if a Standard-ML programmer were to look at this type declaration, it might seem that the entire concordance of even a very frequent word is loaded into memory as soon as one looks it up, but thanks to lazy evaluation, only the portions of the concordance that are needed are actually read from disk.

6 Retrieval data structures and algorithms

6.1 The 'Sequence' data structure

The intention of the indirect concordance directory was that it would act as a one-level search tree; the elements of type `Hit` serve as guards to the nodes which are concordance blocks. At the functional programming level, we augmented these

https://doi.org/10.1017/S0956796800001386 Published online by Cambridge University Press

```
interface Seqs where {

    data (Ord a) => Seq a;

    seq2List :: (Ord a) => (Seq a) -> [a];
    seqEmptyP :: (Ord a) => (Seq a) -> Bool;
    seqFirst :: (Ord a) => (Seq a) -> a;
    seqGuard :: (Ord a) => (Seq a) -> a;
    seqLength :: (Ord a) => (Seq a) -> Int;
    seqSplit :: (Ord a) => (Seq a) -> a -> (Seq a, Seq a);

}
```

Fig. 3. Interface to sequences, as a Haskell module

structures to full search trees, implemented through a data structure called the *sequence*. Sequences represent concordances and all intermediate stages of query computation.

The interface to the sequence datatype is shown in figure 3. Abstractly, a sequence is a sorted collection of data of an arbitrary ordered type. Of course, sequences were designed to carry the `Hit` type, for which lexicographic component order corresponds to natural database order. The public interface to sequences is through the following functions; all but the last take a single sequence as input, and return as follows:

- `seq2List` returns the contents of the input sequence as a sorted list.
- `seqEmptyP`, an emptiness predicate, returns true if the sequence contains no elements.
- `seqFirst` returns the first element of the sequence.
- `seqGuard` returns an item of the carrier type which is $\leq$ all items in the sequence. Note that the guard is not necessarily an element in the sequence. Thus `seqGuard` is not the same as `seqFirst`, and pragmatically it is assumed to be cheaper.
- `seqLength` returns the number of items in the sequence. It is semantically equivalent to the length of the list produced by `seq2List` but is again assumed to be cheaper.
- `seqSplit` takes a sequence and a second argument, called the *split-point*, a value of the carrier type of the sequence. It splits the input sequence into a pair of sequences, the first containing just those items $<$ the argument, the second containing those $\geq$.

In developing the Funser program, sequences were implemented in several different ways. First a basic implementation was written: sequences as sorted lists. Here the split method is implemented by sequential search through the list. Next, operations were defined, such as union and intersection, implemented using only the selectors above. Then, more specialized implementations of the same abstract type were provided, for greater efficiency. Finally, certain operations were optimized to be sensitive to the implementation type of their arguments.

https://doi.org/10.1017/S0956796800001386 Published online by Cambridge University Press

The system ultimately used four implementations for the sequence datatype, represented by the following data declaration:

```
data (Ord a) => Seq a = BVSeq a Int (BnddVec a) |
                        ListSeq a Int [Seq a] |
                        ItemListSeq a Int [a] |
                        EmptySeq;
```

Each of the constructors (except the last) carries a guard and the length of the sequence as the first two components. The elements of the sequence are stored differently in each case, as follows:

- `BVSeq`. Here the elements of the sequence are stored in a special type called a `BnddVec`, for "bounded vector". A `BnddVec` is a section of an array, given as an array and a pair of bounds (not the same as the bounds of a Haskell array):

  ```
  data BnddVec b = MakeBnddVec (Int, Int) (Array Int b)
  ```

  The bounds of the array section are given externally to facilitate the `seqSplit` operation. A `BVSeq` may be split by binary searching the section of the array contained in its `BnddVec`; the portions before and after the split-point may each be made into a `BnddVec` using the same underlying implementation array, but with different bounds. In this way, the overhead of copying elements is avoided. The other sequence operations are easy to implement. This implementation is used for the concordance words with an `Immed` index type, and for each `HitBlock` of the concordance of words with an `Indir` index type.
- `ListSeq`. This is a sequence of sequences, and behaves as a search tree. Every member of each child sequence is required to precede all members of subsequent sequences; that is, as in a search tree, the contents of a left sub-tree must precede those of the right sub-tree. The `seqSplit` function is implemented by searching the guards of the child sequences sequentially, then splitting that child (if any) whose elements span the split-point. `seq2List` is implemented by tree-recursion. The other functions are similar. This implementation is used to represent the entire concordance of a frequent word, and to represent most intermediate results of query processing.
- `ItemListSeq`. Here the elements are stored simply as a list. `seqSplit` is implemented by sequential search; thus, this implementation can be slow. However, it can be used as a default, when nothing else is appropriate.
- `EmptySeq`. A special nullary constructor is used to represent sequences that are known *a priori* to be empty. The `seqGuard` function returns an error when given `EmptySeq` as input.

The index is integrated with the sequence datatype by the following function:

```
searchAsSequence :: Word -> Seq Hit;
```

This function calls `indexm` and converts the `IndexResult` value to a `Seq`. Thus, if the word is `NotFound`, an `EmptySeq` is returned; if it is `Immed`, a `BVSeq` is returned; if it is `Indir`, a `ListSeq` is returned. It is important to note that, since the total

https://doi.org/10.1017/S0956796800001386 Published online by Cambridge University Press

```
mergeSeqs :: (Ord a) => (Seq a) -> (Seq a) -> Seq a

mergeSeqs s1 s2 | seqEmptyP s2 = s1
mergeSeqs s1 s2 | seqEmptyP s1 = s2
mergeSeqs s1 s2 | (seqFirst s2) < (seqFirst s1) = mergeSeqs s2 s1
mergeSeqs s1 s2 = makeListSeq (mSeqHelp s1 s2)
  where mSeqHelp s1 s2 =  if (seqEmptyP after)
                          then [before, s2]
                          else (before:(mSeqHelp s2 after))
    where (before,after) = seqSplit s1 (seqFirst s2)
```

Fig. 4. A function for merging two sequences

frequency of any word is directly represented in the associative file, it always possible to know the length of the corresponding sequence, without forcing the evaluation of any of the concordance decompression which might be embedded in it.

The public interface to the sequence datatype does not include the concrete constructors, but does include construction functions, which are used as intermediate query results are created. The most important of these is:

```
makeListSeq :: (Ord a) => [Seq a] -> Seq a
```

In implementation, this corresponds fairly directly to the `ListSeq` constructor.

6.2 Programming using sequences

The operation of merging two sequences is an easy example of how sequences are manipulated. Figure 4 shows a Haskell implementation of this operation. Note that this implementation uses only the public interface to sequences; it can be optimized by specializing to particular implementation types.

In this example, we assume for simplicity that the two sequences to be merged carry distinct elements; this is usually the case in our application, since they represent occurrences of different words. Essentially, this algorithm is the same as the straightforward method of merging two sorted lists, but `seqSplit` is used to find the points where the two sequences interleave. The result is constructed from a list of sequences produced by the recursive helping routine `mSeqHelp`. Assuming, without loss of generality, that the sequences are not empty and that the first element of `s1` precedes the first element of `s2`, `s1` is split at the first element of `s2`, and the two parts are called `before` and `after`. `before` is the first sequence of the result list, and the remainder is produced by a recursive call using `s2` and `after`.

Recall the co-occurrence search query. Here we have sets of words $\mathscr{A}_1, \mathscr{A}_2, \ldots \mathscr{A}_k$. Each of the $\mathscr{A}_i$ is a disjunction; in the degenerate case, when there is only one set $\mathscr{A}$, the solution is simply the merged list of all the occurrences of words in $\mathscr{A}$. Each disjunction can be handled by the following simple function (`foldl1` and `map` are from the Haskell standard prelude):

```
disjunct :: [Word] -> Seq Hit;
disjunct wlist =
   foldl1 mergeSeqs (map searchAsSequence wlist)
```

https://doi.org/10.1017/S0956796800001386 Published online by Cambridge University Press

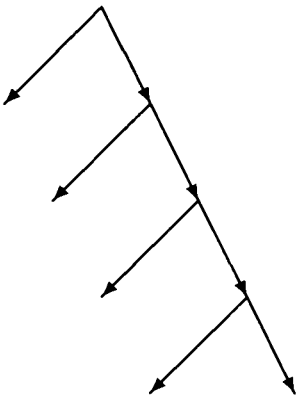

Fig. 5. A bad search tree

That is, the `searchAsSequence` function is mapped over the list of words to get the sequence of coordinates for each word, then the sorted union of all these occurrences is accumulated by applying `mergeSeqs` to the sequences in the list.

Now, an entire query can be solved by solving all the disjunctions, then reducing the result by an `intersectSeqs` function. Actually, the operation is not extensional intersection, since the input is sequences of word occurrences, and the output is a sequence of tuples of word occurrences: those occurrences which group together in sentences. Still, let us call this `intersectSeqs`. A simple version of `intersectSeqs` can be written that is very similar in structure to `mergeSeqs`: the result is a list constructed recursively by using the two sequences to index into each other; but here only the common part is saved. Given `intersectSeqs`, a function for solving an entire query is simple (here the input, `query`, is a list of disjunct queries, that is, a list of lists of words):

```
conjunct :: [[Word]] -> Seq Hit;
conjunct query =
   foldl1 intersectSeqs (map disjunct query)
```

6.3 Distribution over structure

The algorithm just given is correct, but is not as efficient as one would like. Consider a disjunction of two frequent words. The act of merging the sequences for their concordances creates a structure like that shown in figure 5; a deep, sparse tree. When such a tree is intersected with a rare term, the algorithm must sequentially traverse the tree, until the first part near the rare word is found. This forces the computation of the merge of the two frequent words, even in areas nowhere near

https://doi.org/10.1017/S0956796800001386 Published online by Cambridge University Press

the infrequent word. The problem is that the implementations of `mergeSeqs` and `intersectSeqs` tend to flatten the structure of their input. It would be better if the sequence that resulted from a merge retained the bushiness of its input.

This effect is achieved by optimizing `mergeSeqs` and `intersectSeqs` to be sensitive to the implementation type of their sequence arguments. In fact, this optimization does not depend on the particular operations `mergeSeqs` and `intersectSeqs`; it can be expressed as a 'functional', that is, as a function which takes an operation as one of its arguments. In particular, a binary operation that distributes over sequence structure (as we shall define below), can be performed in a structure-preserving way by a process analogous to tree recursion.

To be formal, let us denote sequences by capital letters $A, B, \ldots$ and elements by small letters, $a, b, \ldots$ Let us write the result of splitting sequence A at element x as $A^{<x}$ and $A^{>x}$, for the sequence of elements in A before x, and the sequence of elements in A after x, respectively. Finally, for the list sequence construction function `makeListSeq` let us simply write sequences in parentheses. Thus, by (A, B) we mean the list sequence with child sequences A and B. Then we will say that an operation $\oplus$ *distributes over sequence structure* if for any sequences A and B and arbitrary element z, $A \oplus B$ contains the same elements as $(A^{<z} \oplus B^{<z}, A^{>z} \oplus B^{>z})$.

In Haskell, this is the same as saying that the operation `op` distributes over sequence structure if the two expressions:

```
seq2List (seqA `op` seqB)
```

and

```
let
   (seqAbefore,seqAafter) = seqSplit seqA z
   (seqBbefore,seqBafter) = seqSplit seqB z
in
   seq2List (makeListSeq [(seqAbefore `op` seqBbefore),
                          (seqAafter `op` seqBafter)]
```

evaluate to lists with the same elements, for arbitrary sequences `seqA` and `seqB` and any element `z`.

An operation that distributes over sequence structure is one which works locally on the elements of the sequences that are near each other. Notice that both `mergeSeqs` and `intersectSeqs` are distributive in this sense (`intersectSeqs` is, in fact, a bit tricky: it distributes over the sequences of *sentence* objects, rather than word objects).

Now we may define a function `lSeqRec` (for 'list sequence recursion'), which applies a distributive operation to two sequences, while preserving as much as possible of their structure. To see this, let us consider an example: two sequences, A and B, and a distributive binary function $\oplus$. Let us suppose that both A and B are list sequences with $A = (A_1, A_2, A_3, A_4, A_5)$ and $B = (B_1, B_2, B_3)$. Let a_i be the guard value of A_i and b_i be the guard of B_i and let us suppose that if we merge the two lists of guards we discover that:

$$a_1 \leq a_2 \leq a_3 \leq b_1 \leq b_2 \leq b_3 \leq a_4 \leq a_5$$

https://doi.org/10.1017/S0956796800001386 Published online by Cambridge University Press

```
lSeqRecHelp op opC xs ys@(y1:moreYs) lastY =
   let
      (xsUptoPivot,(pivot:moreXs)) = (splitUpto (`seqLE` y1) xs)
      (pivBefore,pivAfter) = seqSplit pivot (seqGuard y1)
      thisChild = makeListSeq (xsUptoPivot ++ [pivBefore]) `op` lastY
      rest = if (null moreXs)
             then [pivAfter `op` (makeListSeq ys)]
             else lSeqRecHelp opC op ys moreXs pivAfter
   in
      thisChild : rest
```

Fig. 6. List sequence recursion function

Then it is clear that none of the elements in A_1 and A_2 can interact by $\oplus$ with any of the elements of B_1 (or any of the other B_i that follow) since all the elements of A_1 and A_2 must precede a_3, which itself is $\leq b_1$. In fact, only the elements of A_3 that follow b_1, that is $A_3^{>b_1}$, may have any interaction with B_1 and following. Thus, the first term in the result is $(A_1, A_2, A_3^{<b_1}) \oplus E$, where E is the empty sequence.

Now, by the same reasoning, only the sequences B_1, B_2 and $B_3^{<a_4}$ can interact with $A_3^{>b_1}$, so the next term in the result is $A_3^{>b_1} \oplus (B_1, B_2, B_3^{<a_4})$. Similarly, the last term is $(A_4, A_5) \oplus B_3^{>a_4}$.

In general, this can be implemented quite neatly. Figure 6 shows the recursive helping function that does the real work of finding the list of sequences that constitute the result. The first time this function is called, it is passed as its first two arguments both the operation `op` and the same operation with its arguments reversed `opC` (since the operation need not be commutative, but the recursive structure exchanges the roles of the two sequences with each call). Its next two arguments are the two lists of sequences, with the list whose leftmost child is further to the left ($[A_1, A_2, A_3, A_4, A_5]$ in the example) first. We shall discuss the last argument, `lastY` later.

The function `splitUpto` is like the standard prelude function `span`, which breaks a list according to a predicate, returning as a pair the maximal prefix all of whose members satisfy the given predicate and the remainder of the list. But in order to provide ready access to the last item satisfying the predicate (the `pivot`), `splitUpto` cleaves the input list just before it, one position further to the left. Thus, when called with the predicate `(seqLE y1)`, which compares the guard of its sequence argument with that of `y1`, `splitUpto` identifies as the `pivot` the sequence that needs to be split. So, in the first call, `xsUptoPivot` gets bound to $[A_1, A_2]$, `pivot` gets bound to A_3, and `moreXs` gets bound to $[A_4, A_5]$.

The pivot sequence is then itself split, so that `pivBefore` gets bound to $A_3^{<b_1}$ and `pivAfter` to $A_3^{>b_1}$. Finally the current term of the result, `thisChild`, is computed by applying the operation to $(A_1, A_2, A_3^{<b_1})$ and the empty sequence, which is passed in at the first call as the argument `lastY`. The rest of the result is produced recursively by switching the roles of `xs` and `ys`, exchanging the operation `op` and its counterpart, and passing in $A_3^{>b_1}$ as `lastY` to interact in the next term. The recursion terminates when `splitUpto` has consumed its input. Figure 7 shows the top-level structure of the result for this example.

https://doi.org/10.1017/S0956796800001386 Published online by Cambridge University Press

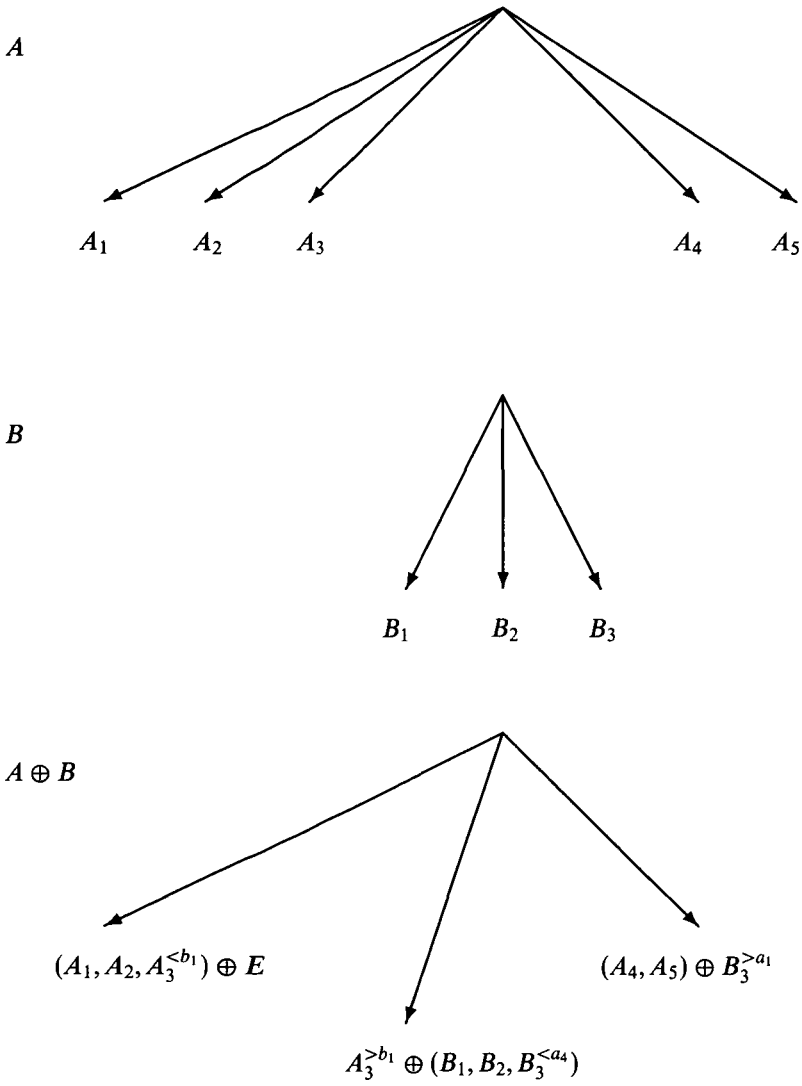

Fig. 7. Making a good search tree

This may seem like a complicated structure to set up, especially if it's never used; but lazy evaluation helps us here. This structure isn't really built unless it's needed. Suppose that A and B are components in a disjunction (so $\oplus$ is `mergeSeqs`), and suppose that $A \oplus B$ is then intersected with a rare term C. Say its first occurrence is c_1 and $b_3 \leq c_1$. Then when we intersect $A \oplus B$ with C, we split $A \oplus B$ at c_1, so the portion of $A \oplus B$ which lies before c_1 is never forced. In particular, the blocks A_1 and A_2 are never read. In fact, as evaluation proceeds, B_1 and B_2 are not read either; the only blocks read are the ones which contain occurrences after c_1.

In practice, the system does even better than this, because we employ the standard technique of ordering the disjunctions from smallest to largest before intersecting them (Date, 1976). So long as a query contains at least one fairly rare disjunction, processing is likely to be fast; at each step the size of the result is bounded above by

https://doi.org/10.1017/S0956796800001386 Published online by Cambridge University Press

the size of the current partial result, and is often much smaller. Furthermore, because of the effect of fixed-size blocks, the larger, later disjuncts are proportionately better indexed and precise indexing from small partial results greatly reduces their transfer volume. This technique is effective because the frequencies of the words are stored in the index file; thus sorting the disjunctions does not force the entire concordance.

Phrase searches proceed by exactly the same algorithm, except that the implementation of `intersect` is slightly different: not only must the words be in the same sentence, they must also be at the right distance from one another.

7 Source of demand

Once the logic of the functional program had been designed, it was next necessary to design its interface with the system that calls it, the Philis emulator. Perhaps the controlling aspect of this design was planning the source of demand which would drive the functional system.

For a time, we considered implementing the entire Philis language emulator in the functional language. In this case, the demand on the system would come directly from the user interface (when it asked to examine part of a result, only then would it be computed). But the problem with this approach is that it doesn't provide for simultaneous work by the retrieval system and the user. Having computed an initial result, the system would wait until the user interface asked for more before resuming computation. Our goal was not only to provide prompt initial results, but also, while the user examined those results, to continue processing. But this requires a different model of demand than just that induced by user interface actions.

Of course, this goal amounts to some form of speculative evaluation. We considered two approaches to this. One was using an implementation language with a speculative evaluation mechanism built in, as has been described by Partridge (1991) and Mattson (1993), among others. Another was to use a lazy language but to provide demand to the lazy system by some external program, written in a conventional, eager language. This program would mediate between the user interface and the lazy system, by anticipating the user's demands.

While we preferred the first solution for its elegance, we chose the second for pragmatic reasons. To a large degree, this decision was shaped by the design goal of remaining basically compatible with the original Philo/Philis interface. In this system, the query results are read as a linear stream. But query results, as we have seen, have a natural tree structure which the Philo/Philis interface chose to ignore. We wanted, eventually, to address the problem of speculative evaluation in this context. If users may browse query results in a tree-oriented fashion, which sub-trees should be speculatively evaluated first, and how? Arranging for an external program to provide demand according to this structured model would have been difficult, perhaps difficult enough to make a true speculative evaluator the clear choice. But since Philo simply reads the fringe of the query tree from left to right, there is no difficulty in the present system in deciding which redex to evaluate. Simulating linear speculative evaluation is simple: one need only to provide a mediator program that continually asks the lazy system for more output, while serving the user's requests in parallel.

For this and other reasons, we decided to implement the Philis emulator in C, and give to it, rather than to the user interface, the responsibility of regulating demand on the functional system. We simply decided that the user's act of posing a query should be interpreted as a demand for all (or at least a lot) of the results of that query. But what if, as we anticipate, after a user examines initial results of a query, he or she decides to abandon it? If the system has already been supplied with demand, there has to be some way of telling it to stop. In our design, the interface between the functional system and the Philis emulator includes two channels: one, an I/O channel, for supplying demand in the usual way and reading results, and the other an out-of-band channel for carrying the 'stop!' message. As we shall see below, this was implemented rather crudely.

8 Implementation

Given the constraints of the overall system design, what functional language should be chosen for the functional server portion? The most important constraint was interoperability, the possibility of interacting with software written in other languages. At the time this work was done we came to the sad conclusion that no language implementation was available that even began to support the interoperability we needed, so we reluctantly began to consider writing our own, or hacking an existing implementation.

There is no question that designing and implementing a programming language for the sake of writing a single application is a drastic measure indeed, and our experience does little to advertise for this approach. Still, there are several factors which mitigate this. One was that one of the goals of this project was explicitly to show functional language implementors what they need to provide in order for it to be possible to write this kind of application. Thus we were willing to put up with more pain than we would have if our goal was simply to have a working program. Another was our interest in perhaps eventually placing a sophisticated and somewhat 'self-optimizing' language in the hands of the users and seeing what use they would make of it.

Crucially, though, the structure of pure functional programming languages is such that simple implementations are quite easy. This has been demonstrated by several "toy" languages whose implementations have been available for study for several years; see, for example, Augustsson's Small (Augustsson, 1985), which we used in an earlier phase of our research (Deerwester *et al.*, 1990) and Ramsdell's Alonzo (Ramsdell, 1989) (the design and name of our language are derived from this work). Indeed, the very first version of our interpreter, with an integrated compiler (later discarded), was implemented by one person (the second author) in the space of about four days.

8.1 Alfonzo

The design of our implementation language, Alfonzo (Ziff and Waclena, 1991), was not intended to be innovative; our main goal was to have an implementation which

permitted easy extension and interoperability. Thus we based Alfonzo on well-known models; basically, Alfonzo is a pure Lisp, the purely functional portion of Scheme (Clinger and Rees, 1991), with lazy evaluation semantics, integer and character data types only, and a few enhancements for Information Retrieval data which will be discussed below.

Indeed, much of our implementation was rudimentary, as compared with the state of the art. We used one of the oldest and (with respect to the performance of the compiled code) slowest methods: combinator reduction. We translated from Alfonzo to combinators via the λ-calculus, a trivial transformation for our subset of Scheme, then from λ-calculus to combinators essentially by Turner's algorithm (Turner, 1979). The combinator code was interpreted by an indirectly threaded (Dewar, 1975) graph reduction machine written in standard C. In our implementation of this technique each object was tagged with a pointer to a descriptor holding the addresses of functions providing the output functions, garbage collection support (Spackman, 1992), and reduction rules (Koopman and Lee, 1989) for its class. The interpreter itself is reduced to a simple dispatch loop.

Certainly, by using simple combinator reduction, we greatly constrained the processing efficiency of our programs in this initial implementation. Still, we dared to hope that the resulting program might be practical, based on the intuition that Information Retrieval is essentially I/O bound, so that some gross CPU inefficiencies might be tolerable.

The key feature of our implementation in this context was that it is, as far as is possible, table-driven. In particular, the primitive functions (combinators) used in the combinator code were specified in tables of C preprocessor macros, and associated, when the interpreter was linked, with arbitrary object modules. In this way we were free to change the set of true combinators used by the compiler as experience with the system grew, but more importantly we could access pre-existing libraries written in other languages by designing a pure functional interface to them, wrapping them as combinators (actually, they are more analogous to arithmetic primitives), and recompiling the interpreter. The ease with which this could be done allowed us not only to experiment with various interfaces, but to add semi-official primitives for such purposes as debugging.

One important technical difficulty with this system was the interface between the C code and the functional system's heap. As we have already observed, the evaluation of a single retrieval primitive can result in the introduction of a tremendous amount of data into the heap. An appropriate interface had to be designed to allow the C code to present the data to the functional program, so that the heap management system knew about it. Also, there was the possibility that the heap would overflow during the evaluation of a primitive, triggering a garbage collection, which might result in important pointers losing their currency. Fortunately, all of our primitives can predict, after doing a little work, how much heap space they will need. A procedure was provided which reserved an arbitrary amount of heap space, triggering a garbage collection if necessary, so that no further checks for storage availability would be needed as data subsequently entered the heap.

Another feature of the implementation that became important was our heap

management and garbage collection schemes. We used a simple semi-space, stop and copy collector, but this was supplemented by a one-bit reference count, a mechanism that distinguishes shared and unshared pointers, as suggested by Wise and Friedman (1977). This allows operations to detect operands that have never been shared and can be reused or returned to a free list as soon as they have been used, reducing the frequency of garbage collection.

In the case of our application, it was particularly our wish that the enormous amount of data that enters and leaves the heap in the course of servicing a single query would, much of it, never become shared. In this respect we were successful. This effect, however, came at the cost of considerable code complexity in the interpreter and ultimately cost more time than it saved. Such sharing-detection requires almost as much work as full reference-counting, and is furthermore extremely difficult to get right by hand; indeed, errors in updating the shared bits led to most of the serious bugs in the interpreter implementation.

8.2 The retrieval primitives

To implement the index interface described as a Haskell module in figure 2, we first extended the Alfonzo interpreter with two new primitives:

`indexrdblk`. This primitive reads a block of concordance. It takes arguments that map directly to calls to the corresponding C interface.

`indexm`. This is the associational primitive. It takes a word and returns according to the cases described above: if the word is not found, it returns `nil`; if it has an immediate concordance, it returns a vector of its coordinates; if it has an indirect concordance, it returns three values: the total frequency of the word, a vector containing its directory, and a parallel vector containing applications of the `indexrdblk` primitive.

In practice, the `indexrdblk` primitive is not part of the Alfonzo language, nor of the intermediate code, since no Alfonzo language construct translates to a use of `indexrdblk`. The only way it can arise is as the result of evaluation of an `indexm` primitive. Of course, since evaluation is lazy, if an `indexm` primitive is evaluated and results in the creation of one or more invocations of `indexrdblk` primitives, these are not evaluated until there is demand for them.

8.3 Handling output: the interface to Philis

We considered several ways of allowing the Philis emulator to control the demand placed on the Funser. One approach was to design a simple command interpreter to be written in Alfonzo, which would accept two kinds of commands: queries, and requests for the the next hit in the result; a wait loop would then be built in to the Philis emulator, so that if it was not busy servicing a request from the user interface, it would continue to issue 'next' commands to the Funser, and each new query command would signal that any remaining results of the last query were no longer needed. But this presents difficulties. Should the Philis emulator wait for

the response to a 'next' command? What if there were no more results, but it was computationally expensive to find that out? It would be undesirable to have the emulator freeze until it got some response from the Funser, so it would still be necessary to have an out-of-band channel for a 'stop' message.

The solution we finally adopted was glorious in its brutal simplicity: the Philis emulator started up a new Funser process for each query, instructing it to write all of its result to a file. In this way, the Funser had continuous demand, as soon as the query was started. Then, in response to user interface commands, the Philis Emulator allowed the user to browse as much of the results file as had so far been written. The out-of-band 'stop' command was implemented by sending an interprocess 'kill' signal. Coincidentally, this is similar to a technique Turner used in the read-eval-print loop of SASL (Turner, 1981). This approach worked quite smoothly, although it certainly is not elegant.

Lazy browsing was the source of most of the significant changes to the original user interface program, Philo. It had been oriented toward conventional eager evaluation: the very first information about a query which it posted to the user was the total number of hits in the result. Naturally, under the new system, this is the last thing the interface knows. Instead, it was changed so that, if Philo became idle, it would ask the Philis emulator how many hits were then available, so that a running total of the number of hits found so far was maintained in a corner of the screen. Meanwhile, it allowed the user to browse the hits that were available in exactly the same manner that had been possible with the original Philo. Since this included looking at Bibliographic references and browsing the full text near hits, this could keep the user busy while the Funser continued to compute (these services, by the way, are examples of the responsibilities of the Philis emulator which it performed without using the Funser). An interesting question to be evaluated was how the user community would react to lazy browsing.

8.4 The Funser program

Implementing the Funser program, using the Alfonzo language and interpreter augmented by the retrieval primitives, was not an easy task. This was largely because the Alfonzo system lacks so many useful features that are commonly found in functional programming systems. At the Alfonzo level, the most troubling difficulties were the lack of a type system, pattern-matching and separate compilation. At the system level, the biggest problem was the lack of debugging support.

These problems interact. It is easy to make mistakes in decomposing a complicated structure such as a list of pairs of a pair and a quadruple; anyone who, programming in Lisp, has mistaken `cadr` for `cdadr` is familiar with this sort of difficulty. A partial reimplementation of the Funser using Haskell B (Augustsson, 1994) showed how helpful typing and pattern-matching are in this regard. Because of these lacks, long hours were spent debugging problems that could have been caught as type errors.

Because so many errors were detected only at run-time, the lack of run-time debugging was particularly troublesome. Some makeshift debugging support was added to the system, such as a `warning` pseudo-combinator that evaluates its first

https://doi.org/10.1017/S0956796800001386 Published online by Cambridge University Press

argument, prints it, throws it away, then returns its second argument. But this was very clumsy.

In this context, the difficulty of writing formatting functions to make the various intermediate results readable for debugging became quite annoying. We developed a library of typed show functions, inspired by Lazy ML's `show_int`, `show_char`, `show_pair`, etc. (Augustsson and Johnsson, 1994), but a typed language would have noticed during type-checking when the types of the formatting functions did not match those of the data being formatted. In Alfonzo, these type errors were detected as further crashes during run-time. Again, the partial reimplementation of the Funser in Haskell B showed how convenient derived instances of Haskell's `Text` type class can be.

All these factors combined to make writing and debugging the Funser much more difficult than it would have been in a more modern functional language. Finally, the fact that separate compilation was never implemented meant that a five minute delay was incurred each time a bug was corrected, as the 2000 lines of the Funser were run once more through the rather slow Alfonzo compiler (itself written in Scheme).

9 Performance

9.1 The good news

In many ways, the system's performance was surprisingly good. Most of the qualitative goals were achieved.

Transient space usage was good. Our heap monitoring tools showed that the space usage was essentially determined by the total number of terms in the query. This is understandable since at least one block of concordance for each term would have to be present in memory in order to decide which one is next in a disjunct; this situation is analogous to merging n sorted lists, since at least the head of each list must be forced. Moreover, the heap usage was *not* related to the total amount of data which had to be processed to compute the total query result.

Initial results were delivered promptly, certainly more promptly than the entire result. In fact, for many queries, the system performed better than the original Philis. Table 1 shows some performance comparisons. The first two columns compare the Funser with the original Philis. The queries are simple co-occurrences: in each case a single word is searched against another single word. The timings are averages over three runs. The first two queries involve a frequent word, *aspects*, and a rare word, *aboieront*. Philis is sensitive to the order in which the words are specified, particularly so in this situation. Naturally, Funser performs about the same regardless of order. The tests were all run on a Sun SparcStation with no local disk; the indices were on remote conventional disk. The timings are as reported by the Unix `time` command: 'R', 'U' and 'S' stand for real, user and system time. 'Real' times should be taken with a grain of salt: they are affected by system load. User time is time spent in the user program, while system time is time spent doing system calls. The latter is an approximate measure of I/O time. For the Funser, the time until the first response

https://doi.org/10.1017/S0956796800001386 Published online by Cambridge University Press

Query	Philis			Funser				Emulator			
	R(F)	U	S	R	F	U	S	R	F	U	S
aboieront & aspects	0.4	0.1	0.2	3.6	(3.6)	2.9	0.3	3.2	(3.2)	1.0	0.5
aspects & aboieront	48	4.7	6.1	3.7	(3.7)	2.7	0.3	3.2	(3.2)	1.2	0.6
privilèges & libertés	58	7	8.9	13.8	3	5.1	1.7	3.2	3	1.2	0.6
libertés & privilèges	40	4.7	6.6	5.5	3	4.7	0.3	3.2	3	1.3	0.5
passion & amour	*exhausts memory*			322	4	269	10.3	19.9	5	6.7	1.3
amour & passion	*exhausts memory*			317	4	264	13	23.5	14	14.9	3.0

Table 1. *Performance of the three systems on some simple queries. 'R', 'U' and 'S' are real, user, and system times. 'F' is real time until first response*

is shown, marked 'F'. For Philis, this is the same as overall real time. First response time is not relevant for the first two queries, since they produce a single result.

Note the final two searches for which Philis produces no results. This is an illustration of the fact that we achieved good transient space usage, and that the old Philis did not. Philis crashes trying to read huge concordance entries into memory.

For the last searches it should be noted that the total elapsed time, some five minutes, is quite long. However, the time until the first result is printed is much shorter. In general, the Funser delivers initial results quite quickly, especially when there is a lot of output to be produced. Initial response time varies somewhat with the complexity of the query. The worst case, of course, is a query which requires a lot of computation in order to produce no output; but this is also the worst case for conventional systems.

Our experience with the one-bit reference count scheme was mixed. As we have noted, we found it tricky to implement; it was the source of most of our interpreter bugs. But the technique was shown to be applicable to our task; for simple queries, such as single disjunctions, all the coordinate data was freed immediately after it was examined, without ever becoming shared, so garbage collections were indeed infrequent.

Unfortunately, the overhead of the scheme more than offset the time saved by garbage collecting less frequently, and in consequence the timings shown were derived with one-bit reference counts disabled. This effect may be understood by observing that although, unlike true reference-counting, the one-bit scheme does no additional work once an object has become shared, it still expends effort proportional to the number of objects that it manages to free: at the end of each reduction it must perform free-list maintenance to allow their storage to be reused (it may also have

https://doi.org/10.1017/S0956796800001386 Published online by Cambridge University Press

deleterious effects on the instruction cache and pre-fetch mechanisms by increasing the size and the number of paths through the primitives). But the overhead of stop-and-copy garbage collection is related only to the total amount of data live at garbage collection time. The very fact that made the one-bit scheme attractive for our application – that we filter a great deal of data without ever sharing it – interacts with lazy evaluation to produce an extremely low transient space loading, which is the very best case for the stop-and-copy mechanism.

9.2 The bad news

Unfortunately, the performance of the Funser overall was disappointing, particularly on the simplest searches. For searches involving a single word, the Funser took much longer overall than the original Philis, although initial results were quicker. In an attempt to isolate where the the Funser was losing, a Funser emulator was written, using C wrappers to the index primitives and standard Unix pipeline-oriented data processing commands. By comparing this Funser emulator to the Funser, we hoped to learn how much time was lost to processing overhead. The third column of Table 1 shows the performance of the Funser emulator on the same queries.

As expected, the Funser emulator out-performed the Funser significantly on simple queries. But more surprising was the fact that the simple Funser Emulator out-performed the Funser on complicated queries also. This was true even though the Funser emulator made no attempt to minimize disk I/O; it always read the entire concordance of every word in the query. Yet, on almost all queries, it performed well. There are two reasons for this. First, was the slowness of combinator reduction. Our interpreter performs about six times slower than the Gofer system (Jones, 1994), a popular interpreted lazy functional system. But second, our intuition that the entire task was I/O-bound was to some degree wrong. Put another way, if you are going to go to some trouble to decide to read only one disk block instead of three, but then those three happen to be next to each other, you had better not have gone to a *lot* of trouble.

Note that the timings here include the latency of starting up a new process for each query. For both the Funser and the Funser Emulator, this is appropriate, since in the actual Philo/Philis system, they each would indeed be re-invoked each time. The original Philis, however, was invoked only once per user session, and so avoided this cost. Still, this overhead is unmeasurably small in our test environment, and the comparative success of the Funser emulator shows that it is not prohibitive in practice.

10 Conclusions and open questions

In the working system which has been released to the public, and which is used in more than 500 sessions per month, the Funser is replaced by the Funser emulator, now somewhat cleaned up and improved. Thus, instead of producing a 'real' application, we have demonstrated yet again that lazy functional programming can be useful for prototyping, something that is now fairly widely accepted. By work-

ing from the functional programming perspective, we have exposed the structure of the information retrieval problem: the sequence data structure is both a useful description of the task and an addition to the functional programmer's tool-box. But although the structure of the retrieval algorithms is elegant, the Funser program itself is not a model of clarity.

These problems are in large part consequences of having had to design and implement our own language. This was a practical necessity at the time, given the constraints under which we were operating, but the situation has changed, and the time is ripe for a re-implementation of the Funser on a richer platform. Recently, functional systems (such as Haskell B) have begun to offer some of the same features that were so crucial to the Funser, and which we developed independently, such as support for incorporating primitives written in C, and for interaction between foreign code and the heap management system.

A re-implementation of the Funser in Haskell would certainly be more elegant and easier to debug, and might well be competitive with the replacement system, the Funser Emulator. Whether competitive or not, a re-implementation would provide an occasion to study some of the interesting questions the Funser experience has raised. For example, can an optimal point be found in the trade-off between the overhead required to economize disk accesses and the overhead of doing one or two too many? Exactly where such a point is to be found depends on the relative speed of the application and the data source. One likely scenario, a distributed retrieval system, where data comes from a remote server rather than a local disk, shifts the balance in favor of a stingier approach, as does increased processor speed. Could heuristics be developed to help tune the system to a particular setting? The model of the database layout and the granularity of the retrieval primitives are also variables worth studying.

Lazy browsing, too, has a lot of possibilities. Since the structure of the result can with the techniques we have developed be arranged to follow the hierarchical structure of the database, it would be natural to support hierarchical browsing: the user is first presented with the (lazy) list of authors in whose works hits appear; the user selects one and is presented with the list of documents by that author, and so on. In a hierarchical browsing system, the simple techniques we used to simulate speculative evaluation would no longer apply, since there is no unique next item of output that can be precomputed. If the system is to guess, it must guess wisely, for unpredictable performance is as annoying to users as long latency. The problem of discarding unwanted lines of speculation also becomes more serious; killing the evaluator is hardly an option when the base computation is still needed to supply other streams of data. The best approach is to provide true, concurrent, speculative execution.

The most significant result of this work has been the demonstration that this problem domain is one in which a lazy functional approach fits very well. For while the Funser system was replaced by an emulator, this emulator performs according to the same specification as the lazy functional version, allowing the user to browse query results as they are generated; and the users do perceive the advantages of prompt delivery of initial results, especially for queries producing bulky output.

Their response has been overwhelmingly positive, even though the emulator performs poorly in comparison to the old, pre-Funser Philis on certain complex queries. This should be taken as a challenge and an opportunity for language implementors: can your system support a re-implementation of the Funser which out-performs the current Funser emulator? If so, you may boast of an application that will be used by more than 500 users a month.

References

American and French Research on the Treasury of the French Language (1989a) *The Philis Command Interpreter: A User's Overview.* Chicago, IL.

American and French Research on the Treasury of the French Language (1989b) *User's Guide to the PhiloLogic Prototype System (5.3).* Chicago, IL.

Augustsson, L. (1985) Small, a small interactive functional system. Technical Report 28, Department of Computer Sciences, Chalmers University of Technology, Göteborg, Sweden, September.

Augustsson, L. (1994) *Haskell B. User's Manual.* Chalmers University, Göteborg, Sweden.

Augustsson, L. and Johnsson, T. (1994) *Lazy ML User's Manual.* Chalmers University, Göteborg, Sweden.

Bookstein, A., Klein, S. T. and Ziff, D. A. (1991) The ARTFL data compression project. In: *Proc. RIAO Conference*, Barcelona, Spain.

Buneman, P., Frankel, R. E. and Nikhil, R. (1982) An implementation technique for database query languages. *ACM Trans. Database Systems*, **7**(2):164–186.

Choueka, V., Fraenkel, A. S., Klein, S. T. and Segal, E. (1987) Improved techniques for processing queries in full-text systems. In: *Proc. 10th Annual ACM SIGIR Conference*, New Orleans, LA, pp. 306–315, June.

Clinger, W. and Rees, J., eds. (1991) Revised report on the algorithmic language scheme. *ACM Lisp Pointers*, **4**: July–September.

Date, C. J. (1976) *An Introduction to Database Systems.* Addison-Wesley.

Deerwester, S. C., Ziff, D. A. and Waclena, K. (1990) An architecture for full text retrieval systems. In: *DEXA 90: Database and Expert Systems Applications*, Vienna, Austria, pp. 22–29, September.

Deerwester, S., Waclena, K. and LaMar, M. (1992) A textual object management system. In: *Proc. 15th Annual ACM SIGIR Conference*, Copenhagen, Denmark, pp. 126–139, June.

Dewar, R. B. K. (1975) Indirect threaded code. *Comm. ACM* **18**(6): 330–331, June.

Fox, E. A., Heath, L. S., Chen, Q. F. and Daoud, A. M. (1992) Practical minimal perfect hash functions for large databases. *Comm. ACM* **35**(1): 105–121, January.

Free Software Foundation, Inc. (1989) *GNU dbm, Release 1.4.* Cambridge, MA.

Friedman, D. P. and Wise, D. S. (1976) Output driven interpretation of recursive programs, or writing creates and destroys data structures. *Information Processing Letters* **5**(6): December.

Hudak, P., Peyton Jones, S., Wadler, P. *et al.*, eds. (1991) Report on the programming language Haskell, a non-strict, purely functional language, version 1.1. Technical report, Yale University, August.

Jones, M. P. (1994) The implementation of the Gofer functional programming system. Technical Report RR-1030, Yale University, Department of Computer Science.

Koopman, P. J., Jr. and Lee, P. (1989) A fresh look at combinator graph reduction (or, having a tigre by the tail). *SIGPLAN Notices* **24**(7): 110–119, July.

Mattson, J. S., Jr. (1993) *An Effective Speculative Evaluation Technique for Parallel Supercombinator Graph Reduction.* PhD thesis, Department of Computer Science and Engineering, University of California, San Diego.

Morrissey, R. and del Vigna, C. (1983) A large natural language data base. In: *Educom.*

Partridge, A. S. (1991) *Speculative Evaluation in Parallel Implementations of Lazy Functional Languages.* PhD thesis, Department of Computer Science, University of Tasmania.

Ramsdell, J. D. (1989) The Alonzo functional programming language. *SIGPLAN Notices* **24**(9): 152–157, September.

Spackman, S. P. (1992) Images of type. Master's thesis, Concordia University, Montreal.

Turner, D. A. (1979) A new implementation technique for applicative languages. *Software - Practice and Experience* **9**: 31–49.

Turner, D. A. (1981) The Compilation of an Applicative Language to Combinatory Logic. PhD thesis, Oxford University Programming Research Group.

Wise, D. S and Friedman, D. P. (1977) The one-bit reference count. *BIT* **17**: 351–359.

Ziff, D. A. and Waclena, K. (1991) The "Alfonzo" programming language. Technical Report 91-01, Center for Information and Language Studies, University of Chicago.