Improving Persistent Data Manipulation for Functional Languages^{*}

Kevin Hammond University of Glasgow Dave McNally University of St. Andrews

Patrick M. Sansom University of Glasgow Phil Trinder University of Glasgow[†]

Abstract

Although there is a great deal of academic interest in functional languages, there are very few large-scale functional applications. The poor interface to the file system seems to be a major factor preventing functional languages being used for large-scale programming tasks. The interfaces provided by some widely-used languages are described and some problems encountered with using these interfaces to access persistent data are discussed. Three means of improving the persistent data manipulation facilities of functional languages are considered: an improved interface to the file system, including a good binary file implementation; an interface to a database; and the provision of orthogonal persistence. Concrete examples are given using the functional programming language, Haskell.

1 Introduction

Modifying a file is a common operation in application programs. Frequently, a program needs to change only small parts of a large file or files. For example, to record the arrival of a shipment, a program controlling an inventory might update the information about a single item in the stock file. We term reading or writing part of a file without needing to read or write the entire file *incremental* read/write.

There are some inherent problems associated with modifying files in a functional language. Firstly, when any data structure is modified, the original data structure must be preserved and a new copy constructed¹. If this is not done, then referential transparency may be lost. The cost of replication may be high for large data structures such as files. Secondly, the order in which update and read operations are performed is significant. In a functional language the new logical versions provide a handle on the sequencing, but may force an unnatural style on the programmer.

^{*}This paper appeared in Functional Programming, Glasgow 1992, Workshops in Computing, Springer Verlag, 1993.

[†]This work is supported by the SERC GRASP and Bulk Data Types Projects, a Royal Society of Edinburgh Fellowship, and a Commonwealth Scholorship.

Authors' address: Dept of Computing Science, Glasgow University, Glasgow, Scotland. Email: {kh,trinder,sansom}@dcs.glasgow.ac.uk,djm@cs.st-andrews.ac.uk

¹It may be possible to avoid this for unshared data structures [3].

The following sections critique several existing languages before discussing means of improving file-handling within functional programs. There appear to be two broad generations of file interfaces in current functional languages. The first generation includes the interfaces provided by LML[2], Miranda[13], KRC [12] and Orwell[14]. These interfaces present several difficulties, including the lack of incremental read/write, evaluation-order issues, type issues, the loss of type information, the loss of sharing within stored data structures and even the loss of referential transparency. More recent languages such as Hope+[7], Haskell[4], or Concurrent Clean[9] have improved second-generation file interfaces, based on continuations, response/request and event I/O, respectively. None of these interfaces provides incremental read/write primitives, though they may be added in each case².

It is worth distinguishing between problems which are fundamental to manipulating persistent data and those which arise simply because features have been omitted from current languages. Incremental read/write and loss of sharing within persistent data structures are problems that can be solved by careful re-design and implementation. Indeed preliminary proposals for both are included in Section 4. Preserving type security over files appears to be more difficult with the static type systems based on name equivalence which are used in most functional languages.

Three means of improving the persistent data manipulation facilities of functional languages are investigated. Firstly, and in the short-term, a language can be better integrated with the file system. Secondly, for large amounts of data or concurrent access a language can be grafted onto a database: Software AG's Natural Expert[10] is an example of this approach. Thirdly, and in the longer-term, a language can be made fully persistent, as has been done with Staple or Poly/ML [5, 6].

The remainder of this paper is structured as follows. Section 2 describes file-handling in non-persistent languages. Section 3 outlines the problems encountered using these constructs. Section 4 investigates the means of improving the file manipulation facilities of functional languages. Section 5 concludes.

2 Current File Constructs

2.1 Read/Write Operations

All non-trivial functional languages provide some mechanism for reading and writing files. These mechanism are not necessarily functional, however. For example, KRC's well documented *read* and *write* functions have the following specification.

- write fname x will print the value of x into the file called fname.
- read fname will return the contents of the file fname.

This means that a program that copies the contents of a file "old" to a file "new" can be written as simply:

²In fact, we have been informed that incremental I/O primitives have now been added to Concurrent Clean, in response to an earlier draft of this paper.

```
write "new" (read "old")
```

The semantics of *read* and *write* are not so simple. The *read* function is lazy: the contents of the file are retrieved when demanded by the program. The *write* operation is hyper-strict: a single write is performed when the result of the *write* function is demanded. Data is written to the file using the normal printing function, so the integer 103 would be written as the string "103", for example. Nothing prevents written data being read as a value of a different type. In addition, the evaluation order may determine the result of a series of reads and writes. This is undesirable since referential transparency may be lost.

2.2 Response/Request Streams

Haskell uses streams of responses and requests to communicate with the file system, and indeed with the operating system in general. A Haskell program that interacts with the file system is a stream-processing function that produces a stream of requests for file operations and receives a stream of responses corresponding to the requests. More formally, file manipulating programs the have type [Response] -> [Request] and, if Name and IOError are some sensible values,

```
data Request = ReadFile Name | WriteFile Name String | ...
data Response = Success | Str String | Failure IOError | ...
```

The operations have the obvious meanings, for example the program that copies "old" to "new" files can be written:

The body of main is the list containing the ReadFile and WriteFile requests. The parameter is the list of responses, with the first response giving the result of the ReadFile request.

The Haskell file interface is much more elaborate than described here. However, the only significant difference from the viewpoint of persistent data is that files may contain data of type **Bin** instead of **String**. This type is an implementation-defined, space-efficient representation, which can be used for most data types. It is described further in Section 3.3.

3 Issues

3.1 Incremental Read/Write

Compilers are generally the largest programs written in most functional languages. A compiler reads the entire source file and generates a complete target file, possibly repeating this process in several passes. Most functional languages manage this type of interaction well. However, many file application programs do not have this pattern of interaction. Instead they retrieve and modify only a small part of a file or files. For example to record the delivery of an item of stock in a warehouse, the record associated with just that item is modified. Staple[5] is the only functional language which the authors are aware of that permits the modification of part of a file (or persistent data structure) without rewriting the entire file. In the warehouse example this would require rewriting the entire stock file just to record the delivery of a single item, a clearly undesirable effect.

Both the **read** operation and the **ReadFile** request provide incremental reading, in part because the contents of the file are only retrieved when demanded by the program. Hence a query about some item in the stock file would only need to read the stock file until the item was encountered. The remainder of the file need not be read, and so only half of the file is read on average. In contrast, in a language with random access files, only the desired item would be read.

3.2 State

The state, or contents, of a file may change during the execution of a program: the file may be written by this or another program. Without care, referential transparency may be lost in a program which reads a file that could change.

The **read** operation in KRC and other languages can either not observe the changing state of a file or is not referentially transparent. It is possible to define functions that do not permit the observation of state changes. Hence when the KRC function

getold = read "old"

is first called it returns the contents of the file named "old". Even if the contents of "old" are changed, subsequent invocations of getold will return the original value.

Miranda's **read** operation apparently allows the changing state of a file to be observed. This option violates referential transparency as the same expression **read** "new" can return different values when evaluated at different times. Programmers are admonished: "users who write Miranda programs that read and write the same file are warned that they are treading on dangerous ground and that the behaviour of such programs is unlikely to be reliable" [13].

In Haskell every **ReadFile** request notionally retrieves the entire file and so must, at least logically, duplicate the file to prevent subsequent writes to the same file from changing the value read. If the file is long, duplicating it might be an expensive operation.

Duplication of the file by requests like **ReadFile** can be avoided by readlocking the file. In the prototype Glasgow Haskell implementation, a write to a read-locked file causes the file to be renamed. The renamed file is deleted if the program terminates normally. This has the (significant) advantage that no file duplication need occur. However, under an OS such as Unix, the applications are responsible for maintain these locks.

3.3 Printable Representation

The type of the file contents in both the write operations and the WriteFile request is String. Non-string data is typically coerced into a string by a show function, for example 130 becomes "130". Storing a printable representation has some serious disadvantages.

- The file must be reparsed on input.
- A text file is marginally less secure as it is easily edited.
- Typically the data expands in size. For example a 4-byte real 3.1789E12 becomes the 9-byte string "3.1789E12".
- Type information may be lost. In particular type abstractions may be breached. For example was "160" a height or a weight? See also Section 3.5
- Without explicit efforts to preserve it, sharing within data structures is lost.

To avoid these problems Haskell introduced the Binfile construct. Bin is a primitive abstract data type and values of type Bin are implementationdependent representations of values in the language. There is a class Binary of types that can be coerced into type Bin. The coercions to and from Bin are performed by showBin and readBin respectively. The request WriteBinFile is identical to WriteFile except that the file contents are of type Bin. Because an internal representation has been used,

- The file does not need to be reparsed on input,
- The file is slightly more secure, and
- Data expansion is reduced.

There are some serious limitations with current implementations, however.

- A Binfile can only be used within a single implementation, i.e. one that uses the same internal representation.
- Sharing within data structures may still be lost by current implementations.
- Type information is still lost, including the breaking of type abstraction.

In section 4.2 we argue that both the sharing and type abstraction problems can be overcome by a suitable modification of the Binfile implementation.

3.4 Hyperstrict Writes

Both the write operation and the WriteFile request are hyperstrict in the value written to the file. Hyperstrictness precludes the preservation of

• Partially evaluated, and hence any potentially infinite data structures.

• Data structures containing functions.

It has been argued that, to make error detection easier, only completely evaluated values should be stored in data files. If unevaluated values are stored, then a program may encounter an erroneous closure that is the legacy of an unknown program that was evaluated at some unknown time [11]. In contrast, the Staple persistent functional language permits the storage of partially evaluated values.

3.5 Type Issues

Preserving all of the type information associated with persistent values is difficult. Section 3.3 described how type abstractions could be broken if structural type equivalence is used. Our preferred solution is to use dynamic name equivalence, similar to the static name equivalence used for internal types in most functional languages, and to provide a mapping between the static and dynamic types. This design avoids loss of type abstraction, without needing to modify the type system (as may be necessary with full dynamic types). There are still several problems, however:

- 1. It must be possible to convert any stored value into its equivalent internal form, and no other. This can be achieved by suitable mappings between the static and dynamic types, as suggested above.
- 2. It is impossible to communicate data between two independent programs unless they share identical types. This is a consequence of requiring strong type abstraction.
- 3. Since there is no strong connection between a program and its data, a programmer may change a type without changing the persistent data which uses that type. In this case, some mechanism must be provided to convert between old and new type representations, if the existing data is still to be used.

Although we believe these are very important issues we will not consider them further here, since they are general issues which apply to all persistent systems and not to functional languages per se. Instead, we concentrate our attention on proposals to improve the underlying facilities for persistent data manipulation.

3.6 Summary

The Haskell Binfile and response/request constructs overcome or ameliorate many of the problems with read and write operations. However, some problems remain. The most important of these is the lack of incremental read/write. Coping with state changes, and preserving both type information and sharing within data structures are also desirable.

4 Improvements

4.1 Indexed File System

Incremental Read/Write

As argued above (in Section 3.1) any serious functional language requires support for incremental file read/write. This section outlines one possible design based on indexed files.

A new, optional, indexed file type could be added to Haskell's existing text and binary file types. The associated requests are similar to the existing **ReadVal** and WriteVal optional requests. The model underlying indexed files is that of an indexed sequential file that stores key, value pairs. Strings have been used to represent keys because they have a complete order and also because values of most data types can be easily converted into a string using **show**. The stored values must be of type **Bin**. Finally, the present Haskell file mode (a Boolean) is generalised to include a mode which allows both reading and writing.

```
type Index = String
data Mode = RMode | WMode | RWMode
data Request = ...
| OpenIxFile Name Mode
| ReadIx File Index
| DeleteIx File Index Bin
| ReplaceIx File Index Bin
| ReplaceIx File Index Bin
| ReadNextIx File
| ReadPrevIx File
```

OpenIxFile is analogous to **OpenFile**, and the existing **CloseFile** is generalised to also close indexed files. **ReadIx** returns a key, value pair corresponding to the least key greater than or equal to the key supplied. This facilitates programs that, for example, list all names beginning with "D". **InsertIx** of an existing key value is an error. **DeleteIx** or **ReplaceIx** of a key that does not exist is an error. **ReadNextIx** (**ReadPrevIx**) returns the key, value pair with the next (previous) greatest key, on a newly opened file it returns the first key, value pair

While indexed files do allow incremental read/write, they provide no more type security than the existing file types. In fact, because values may be stored incrementally they allow the creation of heterogeneous files, that is a file might contain an Integer and a String.



Figure 1: Improved BinFile implementation

4.2 Improved Binfile Implementation

An alternative approach is to provide an improved Binfile implementation. We believe that a carefully constructed implementation of the Haskell Binfile can provide a file system which preserves sharing and enables incremental read/write. Such an implementation may provide many of the benefits of a persistent system without the high implementation cost of providing fully orthogonal persistence (as described in Section 4.4).

The idea is to give the runtime system the task of moving the data between the heap and the file. This process can be made to preserve the structure within the file and restore it in the heap when the data is subsequently read. It is initiated by modified WriteBinFile and ReadBinFile requests:

- The new WriteBinFile request forces evaluation of the data and writes it into the file. File offset pointers are used within the Binfile to preserve the structure of the original data.
- The new ReadBinFile request opens the file and reads the first closure. Rather than reading the entire data structure, special "read" closures are constructed which contain the *file* and an *offset* within the file for each data item which should be read (see figure 1). This constitutes an "address" which can be used to access the data in the file. When the program demands the value of a "read" closure the runtime system reads the file and constructs the closure. A random access file facility, like the Unix *seek* system call, is required to provide efficient *file/offset* access.

Preserving Sharing

Sharing is preserved by maintaining two mappings: (heap address $\rightarrow file/offset$) and $(file/offset \rightarrow heap address)^3$. Before a closure is written into a file the heap address is looked up in the (heap address $\rightarrow file/offset$) mapping: If the closure already resides in the file being written the offset returned is used to reference the file copy of the closure. If the closure does not reside in the file being written it is written into the file. Entries are added to both mappings indicating the new *file/offset* for this heap address, and the new offset used to reference the file copy of the closure. A similar process occurs when a closure is read except that the (*file/offset* \rightarrow heap address) mapping is used to determine if the *file/offset* is already in the heap.

This process of transferring data between the heap and a file is very similar to the transfer between the local and global memories described for the parallel GRIP machine[8]. Here the (heap address \rightarrow *file/offset*) mapping is implemented by attaching an extra word to every closure containing its global memory address (if present). GRIP does not yet implement the (*file/offset* \rightarrow heap address) mapping so sharing is lost when reading from global memory, with multiple copies being created in the local memory. A similar technique to that described above could be used to preserve sharing.

Incremental Read/Write

If a Binfile is read modified and then written back to the same file one might expect the old file to be removed and a new modified version created. This need not happen. Instead the runtime system can extend the existing Binfile writing back the modified data structure. Now any part of the data which has not been modified already resides in the file. This will be detected when the (heap address \rightarrow *file/offset*) mapping is examined and the unmodified data need not be written back to the file — the file offset given by the mapping is used.

By making use of appropriate data structures, such as trees, modified versions of the data can be built and the changes written. Consider the following batch update transaction processor:

The **Bin** data structure stored in the file, "data", is a tree. Only the path(s) modified by the updates will be written back to the file. This program makes use of irrefutable pattern-matching (~) to match the response lazily.

The final step of the WriteBinFile request is to update the file header to reference the new data. This commits the entire write. Subsequent ReadBinFile requests will now return the new data. If an error occurs before the header is

³These are address translation tables. Persistent systems may refer to the latter mapping as the PIDLAM — Persistent Identifier to Local Address Mapping.

written, the old version of the data remains intact and will be accessed by subsequent ReadBinFile requests instead. Transaction processing can be simulated by performing a WriteBinFile request data after each committed transaction.

Unfortunately this scheme causes fragmentation: Binfiles increase monotonically in size with old data cluttering the file. The *file* itself must be garbage collected to recover this unused space. A simple utility program which copies exactly the live data from the old Binfile to a new file would suffice for a basic implementation. For example, the following Haskell program could be used:

main ~(Bn old: _) = [ReadBinFile "old", WriteBinFile "new" old]

The actual copying would be performed by the runtime system with the strict WriteBinFile request forcing the lazy read from "old".

One Persistent File

Instead of implementing Binfiles as multiple files in the operating system, all Binfiles could be written into a single file. This file could itself be a Binfile containing a single value of type **Tree String Bin**. The tree provides an index implementing the logical file naming structure. The actual **Bin** values are stored at the leaves. All Binfile requests would then access this one file directly, using the index to perform the necessary filename \rightarrow **Bin** mapping.

This idea could be extended to make the single file a persistent store. The runtime system would then allocate objects in this store rather than manipulating the file directly. The store would provide the required garbage collection and locking facilities. Such a scheme would not pay the runtime cost of implementing the entire language on top of the store as with the Staple system[5]. Local computations are still executed in the local heap with data being transferred between the program and the store by Binfile requests. In particular, the runtime garbage collection costs (which are large for Staple) would be no greater than for a normal functional program.

This is still not an orthogonally persistent system, however: neither functions nor suspensions could be stored in the persistent store.

Type Checking

Dynamic type checking needs to occur when the data is extracted from a Bin value by readBin. This Bin value may have been read from a file or created in the heap by a showBin during the run of this program. In either case suitable type information can be attached to the Bin value by showBin (see figure 1). In fact, the principal purpose of these functions is to provide type security when coercing data to and from Bin values.

The fact that showBin and readBin are incremental (e.g. showBin :: Binary $a \Rightarrow a \Rightarrow Bin \Rightarrow Bin$) adds some complication when using these functions⁴. It would be preferable to use non-incremental versions instead (e.g. showBin :: Binary $a \Rightarrow a \Rightarrow Bin$), and define readBin to return a suitable error if the dynamic type check fails.

Any developments in dynamic type checking could be incorporated into this scheme by attaching appropriate type information to **Bin** values. Indeed

⁴An incremental showBin might also cause loss of sharing in some circumstances.

we hope that such a Binfile implementation would encourage further research into this area.

4.3 Database

To implement real applications with large amounts of data, many programming languages have been integrated with a database. For example, C with embedded Ingres SQL. A functional language that has been integrated with a database is Software AG's Natural Expert [10]. Their model directs certain requests to the database in addition to handling the standard operating system requests.

The advantages of this approach are as follows:

- Incremental read/write are provided.
- Sharing within data structures is preserved by the referential integrity of the database.
- Concurrent access to data is provided by transactions.
- A DoQuery request permits efficient interrogation of the database.

The disadvantages are as follows:

- The type problem is compounded by the mismatch between the higherorder type system in the functional language and the data types supported by the database (e.g. tuples and relations in a relational database).
- The **DoQuery** request requires meta-programming to construct the string representing the query.
- Portability might be lost as the databases may not be available on some architectures.

This approach would seem most suitable for applications that require incremental read/write of, efficient queries over, or concurrent access to large quantities of data.

4.4 Orthogonal Persistence

Conventional languages only allow certain types of data to persist, e.g. strings in LML or sequences of most types in Pascal. Languages with orthogonal persistence permit data of any type to persist. Staple[5] is a functional language with orthogonal persistence, and Poly/ML[6] is a near-functional language with orthogonal persistence.

The advantages of this approach are as follows.

• Incremental read and write can be implemented on persistent data structures, but the programmer must explicitly reconstruct a new version of the data structure. New versions of only some data structures can be cheaply constructed. For example a new version of a tree can be cheaply constructed, but a new version of an association-list cannot. There are several data structures that are useful for storing persistent data but cannot be cheaply copied[11].

- Sharing within a data structure can be preserved.
- Concurrent and lazy evaluation are both possible. See Section 3.4.
- Persistence is elegant: transfer to and from persistent storage is performed without explicit programmer control.

The disadvantages are as follows.

- Persistent languages use structural type equivalence and it is not clear how to integrate this with the name equivalence used in most functional languages.
- At present persistence is a new technology, and not well understood. Implementations are slow, small and experimental. Considerable effort is required to make a language persistent. In contrast, much less effort is required to produce a good Binfile implementation, as described in Section 4.2.

It is conceivable that the structural versus name equivalence issue can be resolved, and that persistent language technology will become well-established. In this case orthogonal persistence appears to offer the most elegant means of manipulating persistent data in a functional language.

5 Conclusion

The file system interfaces provided by current functional languages have been described, and some problems have been identified. Many of these problems are resolved or alleviated by the Haskell Binfile and response/request constructs. Some outstanding problems with the Haskell interface have been identified, the most important being the lack of incremental update. Following an earlier version of this paper, incremental I/O has now been implemented in at least one functional system (the Concurrent Clean compiler mentioned earlier). We hope that other implementors will be similarly motivated to improve their file system interface.

Three approaches to resolving the remaining problems with a response/ request interface have been investigated. We conclude that, in the short term, an improved file system interface with a better implementation of Binfiles will make it easier to implement many application systems. For applications requiring access to large quantities of data, concurrent access or efficient queries, a database interface may be necessary. In the long term, if the type equivalence issues can be resolved and persistent language technology becomes welldeveloped, orthogonal persistence appears to offer the most elegant means of manipulating persistent data in a functional language.

6 Acknowledgements

We would like to thank John Launchbury, John O'Donnell and Rinus Plasmeijer for reading and commenting on earlier versions of this paper.

References

- [1] Atkinson M.P. Programming Languages and Databases, in *Proceedings of the 4th International Conference on Very Large Databases*, 1978.
- [2] Augustsson L. A compiler for lazy ML, in Proceedings of the ACM Symposium on Lisp and Functional Programming, Austin Texas, USA, August 1984.
- [3] Bloss A. Update analysis and the efficient implementation of functional aggregates, in Proceedings of the 1989 IFIP Conference on Functional Programming Languages and Computer Architecture, London, September, 1989.
- [4] Hudak P. Wadler P. Peyton Jones SL. (Eds) Report on the Programming Language Haskell, Version 1.2, ACM SIGPLAN Notices, 27(5), May 1992.
- [5] McNally D. Joosten S. Davie A. Persistent Functional Programming, in Proceedings of the 4th International Workshop on Persistent Object Systems, Martha's Vineyard, Mass., USA, September, 1990.
- [6] Matthews D.C.J. A Persistent Storage System for Poly and ML. University of Cambridge Technical Report 102, January, 1987.
- [7] Perry N. Hope+C a continuation extension for Hope+. Internal Report, Department of Computing, Imperial College London, October, 1987.
- [8] Peyton Jones S.L. Clack C. Salkild J. Hardie M. GRIP a highperformance architecture for parallel graph reduction, in *Proceedings of* the IFIP conference on Functional Programming Languages and Computer Architecture, Portland. Kahn G. (Ed), Springer-Verlag LNCS 274, September, 1987.
- [9] Achten P.M., van Groningen J.H.G. and Plasmeijer M.J. High-level specification of I/O in functional languages, in *Functional Programming, Glasgow* 1992, Launchbury J. Sansom P.M. (Eds), Springer-Verlag, Workshops in Computing Science, 1992.
- [10] Software A.G. NATURAL EXPERT Reference Manual, Version 1.1.3, 1990.
- [11] Trinder P.W. A Functional Database. D.Phil. Thesis, Oxford University, December, 1989. Available as: Technical Monograph PRG-82, Programming Research Group, 8-11 Keble Road, Oxford OX1 3QD, England; Technical Report CSC 90/R10 Dept. of Computing Science, University of Glasgow, Scotland.
- [12] Turner D.A. Recursion Equations as a Programming Language in Functional Programming and its Application. Darlington J. Henderson P. Turner D.A. (Eds), Cambridge University Press, 1982.
- [13] Turner D.A. Miranda System Manual, Research Software Limited, 1987.
- [14] Wadler P. An Introduction to Orwell 4.07, Internal Document, Oxford University Programming Research Group, December, 1987.