A Persistence Library for Haskell

André Santos Bruno Abdon

Centro de Informática - UFPE

Abstract

Like most functional programming languages, Haskell doesn't offer adequate support for using persistent data structures. In this paper we present a simple persistence library for Haskell, offering a purely functional interface for handling persistent data types. The proposed scheme can also be used to support persistence in other functional languages.

1 Introduction

Currently, there are few ways of dealing with persistent data structures in Haskell, all of them inadequate for most applications that need persistent data structures.

Using the standard Haskell libraries the only way to have a persistent data structure is to convert it to a string and write it to a file, and to read it back you do the opposite: read it from a file into a string and then convert the string back to the original data structure. These conversions are usually done through the `read` and `show` standard functions. File reading and writing is done using the `readFile` and `writeFile` functions, which read and write strings to and from files.

```
readFile :: FilePath -> IO String
writeFile :: FilePath -> String -> IO()
```

An example of a program written using this format is given below, which reads a list of tuples containing pairs of names and telephones from a file (as a string represented by the variable `text`, which is then converted to a list of tuples using the `read` function), then adds a new pair to the list, naming it `newList`, and then writes it back, using the `show` function to convert the list of tuples back to a string and the `writeFile` function to write the string to the file:

```
main = readFile "agenda.txt" >>= \ text ->
       let newList = ("John",2211212):(read text)
       in writeFile "agenda.txt" (show new_list)
```

This simple mechanism has many drawbacks:

- the data is rewritten entirely when a `writeFile` is issued, even if minimal or no modification occurred in the data structure;

- although reading the file through `readFile` is done lazily, it can easily read and bring the entire file to memory, in order to be able to rebuild the data structure in memory. A search in a binary search tree created this way, for example, instead of possibly reading and traversing only part of the file, can easily end up reading the entire file in order to parse it and build the appropriate data structure to perform the search.

- since these functions convert data to and from strings, usually through automatically generated functions (by deriving instances for the `show` and `read` functions), the storage format is very naive, and the efficiency of the conversion is also compromised.

- there is no provision for reading and writing into the same file, which may lead to errors in the program above, depending on the Haskell implementation. For example, with the read operation done lazily, if one writes to the file before reading the entire file the results of the read operations are unpredictable in some implementations.

The main advantage of this approach is that, once the file is read into a string and parsed, the data structure can be manipulated in a completely functional framework, since the only operations that explicitly involve IO (i.e. use the IO Monad) are the `readFile` and `writeFile` functions.

Another option is the use of lower level primitives from the Haskell standard IO library, similar to the ones provided by languages like C. In this case one can use functions like `openFile` to open a file, which returns a file `Handle`, which can then be used to operate on the contents of that file. The operations one is supposed to perform over a handle are reading and writing characters (using `hGetChar` and `hPutChar` functions) or changing the handle's current position. Unfortunately, these operations need to be ordered in a proper sequence, and therefore they are all IO operations. This forces programmers to write more imperative-looking code, in order to force the order of execution. Also, in order to store anything other than `Char`s using this approach, one has to write his own conversion functions, and create and maintain his own file structure, programming in a very low level.

The lack of adequate binary file manipulation functions is a big limitation of Haskell, which has been strongly criticized by the community. Some proposals and non-standard libraries have emerged, but these provide low level file access primitives like the ones presented above, and therefore they don't address our concerns about a simple and high level persistence mechanism.

Our objective in this work is to present an alternative library for writing persistent data, which combines the convenience of writing most of the code without explicit IO operations, with more efficient ways of reading and writing to a persistent store, without the need to read (or rewrite) the entire data structure unless necessary.

2 The library design

Our idea of a transparent persistence mechanism for Haskell follows the approach adopted by some object oriented databases. In object oriented languages like Java or C++, there are essentially three mechanisms for dealing with persistent data:

- using simple file access primitives. This approach is similar to that provided by Haskell, either reading and writing entire data structures using a *serialization* mechanism, or using low level file access primitives.

- using relational databases. In this approach one has to write (possibly with the support of some software tools) mappings from objects into relational tables. For more complex objects, which reference other objects, one usually needs to store references to unique identifiers for each object, and access these objects (through queries, usually in different tables) in order to recreate the original complex object in memory. The access to these relational databases is usually done through mechanisms (such as ODBC or JDBC) which do not follow the object oriented principles of the language.

- using object oriented databases. These offer a programming and access method much closer to the concepts of the languages, and keep in persistent stores direct links between objects that have references to other objects. Therefore, for complex objects, these databases sometimes also offer better performance than relational databases.

For functional languages, there are libraries that offer access to relational databases, but although they are an important tool to access existing databases, they are not easy to use to access more elaborate data structures, since the mapping of data structures to and from relational tables would have to be done by the programmer.

Our approach uses ideas taken from some object oriented databases: the proposed library offers a number of abstract persistent data structures, that will work as containers for other data structures. Each container varies only by the interface it provides and its policy (and efficiency) for storing data.

For example, we provide persistent lists (`PList t`), which provides functions for the basic list operations, but which are persistent. Similarly, we provide other persistent abstract data structures, such as bags (similar to lists but without ordering), trees, and we intend to provide other useful and important persistent data types as containers, such as sets and efficient indexed data structures.

These containers are typed, in a sense that in a given container, all data inside it is of the same type, similarly to what we have with ordinary polymorphic lists and trees. Therefore if, say, *Book* is a data type, one can have a `PTree` of *Book*s or a `PList` of *Books*, in which it would be possible to store values of type *Book*.

Another limitation of our approach is that we support persistence only of evaluated data, therefore we cannot store infinite lists, or functions.

Once one of these storages is created or opened, it is possible to insert and delete data into and from it without the need for explicit sequencing of operations through the use of the IO monad, just like is done with any standard data structure. An IO action is only issued when `close` or `commit` function are called, which commit the changes (i.e. actually write to the file the insertions, deletions or updates that ocurred). This is done by defining the roots that are used to access the data structure, such as the head of a `PList`, or the root of a `PTree` in a file. While a container is open, all read or write operations can be performed identically to the way one manipulates non-persistent data structures, and with the same semantics. Insertions, deletions and commits may occur at any time, with no restriction on the order.

The decision to have explicit commit functions is done to avoid syncronizing to the file all the time, which would be inneficient and often unnecessary. If, for example one is sorting a list, we don't want to persist all the intermediate stages and ilist structures generated

When a persistent storage is opened, its contents aren't brought to memory immediately. They are read from the file as they are demanded, just like with `readFile`. A `commit` operation will flush the data in memory back to the file, acting as if it was ending a transaction started after the previous call to `commit` (or since the file was opened).

The liveness of the data in memory (wether it is kept in memory or reread) is defined by the standard memory management system, and the space is retrieved by garbage collection if it is not needed anymore.

The storage is structured in such a way that the minimum amount of information is actually read from the file when a value is read from the persistent container.

2.1 A persistence container

We will now describe one of the containers, `PBag`, which implements a persistent bag data type, but most of the operations are identical for the other containers. Part of the interface of the `PBag` container is shown below:

```
newPb     :: (Persistent a) => FilePath -> IO (PBag a)
openPb    :: (Persistent a) => FilePath -> IO (PBag a)
commitPb  :: (Persistent a) => PBag a -> IO (PBag a)
closePb   :: (Persistent a) => PBag a -> IO ()
gcClosePb :: (Persistent a) => PBag a -> FilePath -> IO ()
```

These are the only functions that perform explicit IO operations. Every persistence container has similar functions for creating an empty container, opening an existing container, committing the changes and closing the container.

Notice that in order for any data type to be stored in a `PBag` container, it has to be an instance of the `Persistent` type class, which is indicated by the `(Persistent a) =>` clause in the type definition. This type class defines how the data is laid out in the file.

In the next section we will explain this class and how the programmer writes such an instance for a data type. For now, it is sufficient to say that one can easily generate a simple version of the interface using the standard `read` and `show` functions, or create a more elaborate and possibly more efficient storage structure if necessary.

A `PBag` may be created with the function `newPb` or opened with `openPb`. The `newPb` and `openPb` functions receive a path to the file in which the data is kept, and return a `PBag` data structure. If a non-existent file name is passed to `openPBag`, a new, empty `PBag` is silently created and returned. The typechecker will automatically determine the type of the `PBag` based on the first insertion or deletion operation.

The `commitPb` operation receives a `PBag` which might have been altered and that should be stored, and writes it to the file, returning an identical `PBag`, but now guaranteed to be stored in the file. The programmer should not use the first `PBag` after `commitPb` is called, he should use only the returned `PBag`. Using the first `PBag` may lead to writing the same data more than once to the file, but the semantics is preserved.

In order to open a `PBag` (e.g., in another run of the program) it should have been properly closed. The `closePb` function does this task. After closed, a `PBag` shouldn't be referenced anymore, since this will cause an error/exception.

Finally, after insertions, deletions and commits, some unused space is created in the file, as garbage left by those operations. This wasted space increases the file size, and this file space may be reclaimed by closing the PBag with the gcClosePB functions, which forces a garbage collection on the file, in order to recover the wasted space.

Apart from the opening, committing and closing functions, no other operation over the persistence container needs explicit IO sequencing. As said before, adding and removing data from a PBag, PList or other persistence container is done in the same way as with any other abstract data structures, as can be seen from the type definitions of the functions below:

```
addPb    :: (Persistent a) => (PBag a) -> a -> (PBag a)
delPb    :: (Persistent a) => PBag a -> a -> (PBag a)
pbToList :: (Persistent a) => PBag a -> [a]
filterPb :: (Persistent a) => (a -> Bool) -> PBag a -> [a]
mapPb    :: (Persistent a, Persistent b) => (a -> b) -> PBag a -> PBag b

addPl    :: (Persistent a) => PList a -> a -> PList a
delPl    :: (Persistent a) => PList a -> a -> PList a
plToList :: (Persistent a) => PList a -> [a]
(<!!>)   :: (Persistent a) => PList a -> Int -> a
headPl   :: (Persistent a) => PList a -> a
tailPl   :: (Persistent a) => PList a -> PList a
(<++>)   :: (Persistent a) => PList a -> PList a -> PList a
filterPl :: (Persistent a) => (a -> Bool) -> PList a -> PList a
mapPl    :: (Persistent a) => (a -> b) -> PList a -> PList b
```

The addPb function inserts an element into a PBag (as does addPl for PLists), and the delPb finds and deletes a value from the storage. Both functions return a new PBag, modified according to the function. Depending on the data structure used for the persistent store, most of the existing data structure is preserved and shared after these operations, minimising the creation of new parts of the data structure in memory and in the file.

Some other useful functions are provided for the persistence containers, some of them specific to the container. It is possible to apply a *map* or *filter* function over a PBag with the mapPb and filterPb functions respectively, just as it is for PLists, calling either mapPl or filterPl. It is also possible to extract the elements of a PBag or PList back into to a Haskell list, using pbToList and plToList.

Over PLists are defined some useful common list operation such as headPl and tailPL, the concatenation operation (<++>) and access to elements via its position, with (<!!>) (equivalent to the Haskell (!) operator).

At first sight it may seem that type classes could be used to support overloading of the similar operations, such as opening, adding and deleting elements. We could use a class Container with open, add, del and other operations for that. Unfortunately this elegant implementation had problems with the lower level file access, where it had to infer the type of a persistent store from its use. This lead us to use different function names for each persistence container. We intend to investigate this issue again in the future.

3 The Persistent class

Haskell uses the concept of type classes as a way to specify that a given data type has a specific set of functions (specified by the type class) defined for it.

For a data type to be stored in one of the persistence containers, it needs to implement a set of functions, defined by the Persistent class. This class has the following signature:

```
class Persistent a where
        put :: Handle -> a -> IO (TPos a)
        get :: Handle -> IO a
        getF :: Handle -> (TPos a) -> (a,(TPos b))

        putAt :: Handle -> (TPos a) -> a -> IO()
        getAt :: Handle -> (TPos a) -> IO a
        getFAt :: Handle -> (TPos a) -> a
```

These functions tell the library implementation how to lay out and read back the data to and from the file. This signature was taken from [WR98], where a library for accessing binary files in Haskell is presented. We reimplemented these functions using standard Haskell (they used lower level C functions). This has disadvantages, since with standard Haskell we can only read and write characters to files, which forces us to do a lot of type conversions. This is certainly less efficient than the implementation of [WR98], but our approach was to stick to standard Haskell as much as possible.

These functions are used for reading and writing data structures to and from the files, by the persistence containers, and therefore they will never be called directly by the programmer, only by the persistence library containers. Therefore, the high level programmer normally will not have to use (or understand) them, since simple mechanisms for defining these instances are provided by the library and explained below.

Instances for the Persistent class are provided by our library for the simple standard types, like `Int`, `Char` and `Bool`, and for some common type constructors, like lists and tuples. But for user-defined types, is will be necessary to write instances. They can be written in two ways:

- if the programmer always wants to access individual items stored in a container as single entity, he can use a set of simple functions provided by the library, which use the `read` and `show` functions to convert the data to and form strings, just like in the example using `readFile` and `writeFile` in the beginning of this paper. Notice that differently from that, now the granularity of access is for each item in the container, rather than, say, an entire list. This is useful for simple data types.

- write the detailed conversion functions, taking advantage of its characteristics and reading larger parts of the data structures more efficiently and/or more lazily.

Based on recent proposals for extensions of the Haskell language, we believe it is also possible to provide automatic derivation of this second form in the future, similar to the mechanism provided specifically for some standard classes like `Read` and `Show`.

3.1 Default instances

As we mentioned for data types that already have instances for `Read` and `Show`, we can use predefined functions, called `put_default`, `get_default` and `getF_default`, in order to have an instance for the `Persistent` class. Using this simple solution, the data type itself will be read and written as a single string, instead of lazily (through the use of the references to other parts of its structure).

We could have written these defaults directly in the class definition for `Persistent`, but if we did it there we would have to impose that in order to be an instance of `Persistent` a type would need to have instances for `Read` and `Show`, as shown below:

```
class (Show a, Read a) => Persistent a where
  ...
  put = put_default
  get = get_default
  getF = getF_default
```

We believe that to impose this restriction would be unnecessary, and therefore decided to keep these default definitions out of the `Persistent` class.

3.2 Complete instances

If the programmer decides to write instances for his data types in order to have more control over them, he will need to know a bit more about the behavior of the functions defined by the `Persistent` class, which we give below together with an example.

The `put` function takes a `Handle` and some data value as its arguments and writes the data at the `Handle`'s current position, returning a `TPos` indicating the position in the file where the data has just been written. Conversely, the `get` function returns the value found at the `Handle`'s current position. Both `put` and `get` use IO, and, as a side-effect, update the `handle`'s current position to the position right after the data just read (or written).

`getF` requires no IO to be performed. Taking a `Handle` and a `TPos` as arguments, it returns the data found in the `Handle` at the position pointed by the `TPos` argument. As it is not supposed to change the `Handle`'s current position (what would be an side-effect), it also returns a pointer reporting the position right after the data read, to allow sequential reading operations.

An example of an implementation of these methods is shown below for a type `GFigure` (Graphic Figure):

```
data GFigure = Circle Int
               | Rectangle Int Int

instance Persistent GFigure where
        put h (Circle radius) =
              hTell h >>= \here ->
              put h 'C'                >>
              put h radius             >>
              return (TPos here)

        put h (Rectangle width height) =
              hTell h >>= \here ->
              put h 'R'                >>
              put h width              >>
              put h height             >>
```

```
        return (TPos here)

get h =
        get h >>= \cons ->
        case cons of
             'C' -> get h >>= \radius ->
                    return (Circle radius)
             'R' -> get h >>= \width  ->
                    get h >>= \height ->
                    return (Rectangle width height)

getF h (TPos pos) =
        case cons of
             'C' -> let (radius,endPos) = getF h sndPos in
                    (Circle radius,endPos)
             'R' -> let (width,heiPos)  = getF h sndPos
                        (height,endPos) = getF h heiPos
                    in (Rectangle width height,endPos)
        where (cons,sndPos) = getF h (TPos pos)
```

A geometric figure, as represented by the type `GFigure` is either a circle, with an integer radius or a rectangle with its integers width and height. To write a `GFigure` into a file, we first ask the file's current position (with `hTell`) keeping the value in the variable `here`. A `Char` is then written to tell `Circle`s from `Rectangle`s: a 'C' indicates the former while an 'R' marks the later. Then the constructor arguments are written one by one (the radius for the circle and the width and height for the rectangle). Since `radius`, `width` and `height` are `Int`s, and the `Int` instance for the `Persistent` class is provided, they can be written and later read with no further concerns.

The `get` method is a mirror image of the `put`. It first `get`s a character then determines whether it is `get`ting a circle or a rectangle. According to each case it will issue a number of `get` operations needed to acquire all the arguments of the constructor, which will be returned by the function.

The `getF` function seems a bit more elaborated, but it is nothing but a slightly modified version of `get`. It does exactly the same, reading a character to determine what is the figure and then reading the constructor arguments. The notable difference relies on the fact that there's no IO action at all, and since the handle's current position doesn't change, sequential reading is achieved using the pointer returned by the last `getF` call. e.g., as the `height` of the rectangle had been written in the handle right after its `width`, the pointer returned while `getF`ing the `width`, `heiPos`, is used to `getF` the `height` (in "`getF h heiPos`").

4 The library implementation

The implementation underlying any of the containers offered by the library should allow data in memory and data in disk to coexist in the same data structure. The data structures used are as simple as trees or linked lists. But in order to achieve the desired transparency between disk and memory storage, all links in a structure are allowed to be either a memory pointer, a standard file pointer, or a file pointer stored on disk, and all user data in the storages may be in a file (persistent), in memory (not yet commited) or both (in a file and in the memory cache). This is achieved through the use of the data type `MaybeP`:

```
data MaybeP a = NP a                         --non persistent
        | P (TPos a)                         --persistent
        | PR (TPos a) a                      --persistent with a copy
```

`MaybeP` turns any data type into one of three alternatives:

- it is not persistent, therefore is just a copy of the data itself, held in memory;
- it is persistent, and therefore not yet in memory. It is just a pointer to the area in the file `(TPos a)` where the actual data is stored, and therefore may be read;
- it has both the reference to the area in the file where the data is stored, but also has a copy of it in memory.

This way, a list implemented in our scheme would look and act as an ordinary list, being formed by a constructor (*Cons*) having two pointers, the first pointing to the data containing the head of the list (a list element) and the second pointing to the list's tail, which will be either another `Cons` cell or a `Nil`, indicating the list's end. This is precisely the same structure used by the standard list data type in memory! Allowing these pointers to point to either memory or file and allowing part (or all) of the structure to be on disk is what gives the persistence containers the power they have.

```
data List a = Nil
            | Cons (MaybeP a) (MaybeP (List a))
data Tree a = Empty
            | Node (MaybeP (Tree a)) (MaybeP (Tree a)) (MaybeP a)
```

A `PList` returned by an `open` operation is actually nothing more than a file pointer pointing to the `PList`'s head on disk. Inserting some element at such `PList` would create a new *Cons* cell with its data value in memory but with its tail (a file reference in the original `PList`) still on disk. Commiting this new list would write only the new node into the file, with a reference to the rest of the list already on the file, and update the reference to the head of the list in the file.

```
type PList a = (Handle, MaybeP (List a))
type PBag  a = (Handle, MaybeP (Tree a))
```

Deleting an element at a certain index i on a `PList` will create a new `PList` in which every *Cons* cell with an index lower than i is in memory, but whose data values are still on the disk, because the new *Cons* cells created during insertion will use the same data pointers to the old elements on the disk.

A `commitPl` call causes the data in memory *cons* cells to be flushed to the file. New *Cons* cells are created at the end of the file and the new structure returned by `commitPl` will have pointers to them, instead of keeping them in memory. Writing new *Cons* cells at the end of the files instead of overwriting the old file contents is extremelly important, since it is the key step to keep referential transparency within the `PLists` – all the old pointers are still valid, since the file content at any given point is never changed during a session. Notice that after a commit operation writes the data to a file, the `PList` or `PBag` that is returned by the operation is semantically identical to the one received[1], but its representation has changed, since non-persistent (`NP`) cells will have been replaced by persistent (`P`) or persistent and redundant (`PR`) cells.

The need for an in-file garbage collection comes from the decision of appending all new information at the end of the file, maintaining all the existing structure intact. In memory, when some data becomes unreachable, the runtime system's garbage collector frees its space for further use. But the in-file unreachable data structures can't be collected by such mechanism, so it became necessary to provide a function to garbage collect the file.

By transferring valid user data to free spaces, the (in-file) garbage collector makes in-memory file pointers to become invalid, turning the persistent storage structure inconsistent. To avoid the creation of such undesired faulty storage, the garbage collection is only allowed at the time of closing the persistent store.

Notice that the possible need for recovering wasted space in files is also present in relational as well as object oriented databases. There, usually the wasted space is reclaimed either through compaction operations (similar to our in-file garbage collection) or by reusing the wasted space with new data. In the future we intend to investigate ways to support also the reuse of wasted space in our implementation.

5 Related Work

Much work has been done to achieve an elegant way to integrate persistence into Haskell and to functional languages in general, with the most diverse goals, approaches and achievements.

Our persistent library implementation is based on Malcolm Wallace's binary library [WR98]. Wallace's library achieves orthogonal persistence elegantly through overloaded functions to read and write data to the file, which implicitly converts the persistent data into a stream of bits. The programmer is still given the task to write the code of the overloaded functions.

The file, in the Binary Library, is accessed through a `Handle`, in which data of different values may be stored. The programmer has the task to keep pointers to where values are stored in the file, which contrasts with our approach, where the file is viewed by the programmer as a collection of values of the same type. Reading data from files may be done lazily, whereas writing data to it is necessarily done via the IO monad, a limitation not present in our Library.

Hammond, McNally, Sansom and Trinder proposed an improved implementation for the old Haskell standard Binary Library in [KHT92]. In their implementation, the runtime system is given the task of moving the data between the heap and the file, so that sharing of data structures could be preserved transparently to the programmer. Heap pointers have correspondent in-file pointers. Just like in our persistent library, the programmer doesn't have raw access to the file, but instead he sees it as a collection of objects (data). The point here is that to achieve such an abstraction (including the preserving of sharing) their work couldn't be implemented simply by a library using standard Haskell, instead they had to rewrite parts of the compiler (runtime system). Also, although data of potentially any type could be written into the file, there's the need of a system dependent overloaded

[1] accessed through its interface.

function to coerce the data in and from the `Bin` type, just like we have to convert persistent data to a byte-granularity representation when using the persistent library. Also, the data inside the file is indexed by a string, which should be passed at insertion time.

Davie, Hammond and Quintela [TDQ98] implemented a Haskell interface to a generic persistent object store (POS [BM92]), on top of the Glasgow Haskell Compiler. In their approach, the persistent objects maintenance is done by the generic object store, which saves much of implementation effort. The system has its base on a low-level mapping between heap closures and POS persistent objects, which makes possible to achieve goals such as preserving memory sharing. It also makes the system much more difficult to port to other Haskell implementations. Another complicating issue is the need for dynamic typing for values retrieved from the persistent store.

Some applications require access to extremelly large storages of data, sometimes even with the need of concurrent access. In these cases, a database interface might appear as a more efficient solution to give a programming language the power to interact with persistent entities. Ichikawa [Ich95] has an interesting proposal of a database manipulation interface based on a state-transformer approach. In his system, all the updates to the data storage are strict, altough, due to an elaborated database versioning maintenance, retrieval operations may be done lazily.

6 Conclusions and Future Work

We have presented a library for supporting the use of persistent data structures in Haskell, which provides a simple and abstract mechanism for accessing and storing data. The library was implemented using the Glasgow Haskell Compiler, and is written in standard Haskell, except in the support for simultaneous functional file reading and writing. This is not supported in standard Haskell because this usually cannot be done functionally, since writes can cause side-effects in the file reading process. We needed to use a call to `unsafePerformIO`, but the semantics is preserved by the way the file is handled.

We have validated the basic design through an implementation and initial tests. We intend to extend this library with more efficient and useful persistence containers, with more efficient access and store operations, before releasing it. For example, we intend to provide structures based on indexes or hash tables. Another important point is to define efficient ways of reclaiming unused file space. Later we will also have to deal with more complex issues, such as supporting concurrency.

Acknowledgements

We would like to thank the support from CNPq for this work, and also the suggestions made by the anonymous reviewers to improve this paper.

References

[BM92] A. L. Brown and R. Morrison. A generic persistent object store. *Software Engineering Journal*, 7:111–114, 1992.

[Ich95] Yoshihiko Ichikawa. Database states in lazy functional programming languages: Imperative update and lazy retrieval. *In Proceedings of the Fifth International Workshop on Database Programming Languages*, pages 150–163, 1995.

[KHT92] Dave McNally Kevin Hammond, Patrick M. Sansom and Phil Trinder. Improving persistent data manipulation for functional languages. *1992 Glasgow Workshop on Functional Programming*, 1992.

[TDQ98] Kevin Hammond Tony Davie and Juan Quintela. Efficient persistent haskell. *10th International Workshop on the Implementation of Functional Languages*, 1998.

[WR98] Malcolm Wallace and Colin Runciman. The bits between the lambdas: Binary data in a lazy functional language. *Proceedings of ACM International Symposium on Memory Management*, October 1998.