Heap Compression and Binary I/O in Haskell

Malcolm Wallace Colin Runciman
Dept of Computer Science
University of York, UK
{malcolm,colin}@cs.york.ac.uk

Abstract

Two new facilities for Haskell are described: compression of data values in memory, and a new scheme for binary I/O. These facilities, although they can be used individually, can also be combined because they use the same binary representations for values. Heap compression in memory is valuable because it enables programs to run on smaller machines, or conversely allows programs to store more data in the same amount of memory. Binary I/O is valuable because it makes the file storage and retrieval of heap data structures smooth and painless. The combination of heap compression and binary I/O allows data transfer to be both fast and space-efficient.

All the facilities described have been implemented in a variant of Röjemo's *nhc* compiler. Example applications are demonstrated, with performance results for space and speed.

To appear in the 2nd ACM Haskell Workshop, June 1997, Amsterdam, NL.

1 Introduction

1.1 Data representation

Implementors of lazy functional languages tend to use an internal representation of data which is uniform, based on graphs of heap cells. A value is either atomic, occupying one unit of space, or it is structured and each of its components occupies one unit of space, in turn either an atomic value or a pointer to another structured value.

There are good reasons for this memory model. Functional programming systems make little distinction between values and expressions, so a memory cell must be capable of representing either form. Also, functions may be polymorphic and it is therefore very useful for one unit of memory space to be able to hold a value or expression of any type, no matter how simple or complex.

However there are various occasions when this internal representation is inconvenient:

1. **Static data.** Some programs have a large quantity of essentially static data, such as the import table in a compiler, or the dictionary in a natural-language processing system. The standard representation of this data is somewhat bulky. Also, the reasons for using the standard representation do not apply: the data can be fully evaluated early in the computation (so it contains no expressions), and its type is also fully known (there is no remaining polymorphism). In such a situation, a more compact representation can allow more data to be stored. With care, the overhead of garbage collection can also be reduced.

2. **Secondary storage.** Some programs perform output with the intention of being able to retrieve the information from secondary storage in a later input operation. For instance, some compilers generate interface files which are read back during later compilation of separate modules; some applications save large quantities of internal state information for later analysis (perhaps by a tracer or profiler). Haskell's standard mechanism for I/O allows program data to be transferred only in a textual representation. This requires an expensive translation at both output and input stages. It is much more convenient to be able to use the same binary data representation both in memory and on files, so that I/O can be a cheap bulk transfer from one to the other. The standard graph representation does not lend itself to this approach.

3. **Foreign language interfaces.** A Haskell programmer may occasionally wish to call a routine written in C, for instance to perform a system call. Data holding the same semantic information is frequently represented differently in each language, and it must therefore be *marshalled* before passing from one to the other. Again, the ability to define and use a common representation would be very convenient.

4. **Embedded systems.** Device control is also related to I/O. Programs have a high-level view of certain data structures used for control and monitoring. At some stage these structures must be mapped onto narrow bitfields within individual machine registers.

5. **Communication.** I/O in the form of dynamic transmission of data between processes, whether across a network or on the same machine, can be hindered if it must rely on either a shared memory model or on textual representations.

In this paper we show how the programmer can have control over data representation with little compromise of the high-level abstraction facilities which make functional languages attractive.

We provide a means by which the internal representation of program data can be specified to a fine-grained binary level. The type of a value is used to determine how it can be represented as a sequence of bits. Functions are defined to transfer a value between the standard graph representation and these compressed bit sequences. The two conversion functions form the methods of a type class. A small set of primitive operators is used in the definition of the conversion functions, and the I/O monad is used for sequencing. Instances of the compression type class can be derived automatically by the compiler. The programmer can also define custom instances.

An extension to the I/O library is also provided, allowing values to be transferred between the functional program and the rest of the world in compact binary form.

All the examples and underlying facilities described in this paper have been implemented using *nhc* version 1.3 [9].

We leave the question of data representations for foreign language interfaces and embedded systems control as interesting lines of future work. (See however our earlier work on embedded systems [15, 16].)

1.2 Motivation

This work is funded by Canon Research Centre Europe Ltd., who are developing complex software for new ranges of products. Functional languages are very attractive for reasons of programmer productivity, the ease of rapid prototyping, and maintainability of the emerging software. Together, these benefits can bring a product to market more quickly, which is a considerable commercial advantage. Five issues of concern however are:

1. saving memory space in the final product;
2. achieving fast and efficient I/O;
3. interfacing to other product modules written in C/C++;
4. running software in an embedded product;
5. communication between multiple processes.

The most important issue from a commercial perspective is probably the first: saving memory leads to a direct saving on mass-production costs. This sets the scene for our work on compressing heap data and performing binary I/O. However, as the introduction has outlined, one common theme which underlies all five issues is data representation. A declarative solution to the representation problem has the potential to address many challenges from the software engineering arena.

2 A Class of Types with Bit-Vector Representations

To recap, data in a functional language system is usually represented as a linked graph structure, where each link and terminal node typically occupies one machine word. However, it is possible to use type information about values to squeeze the representation down to a much smaller sequence of bits.

2.1 Types and compression

If a type admits just n different values, any value of that type can be represented within $log(n)$ bits. It is clear how this can be applied to an enumerated type, with only nullary constructors. But the same observation also applies to more structured types. In Haskell a structured data value of a type T consists of an n-ary constructor followed by a sequence of n values, each of which belongs to some type $t_0..t_{n-1}$. Hence, a structured value can be represented in binary form by a vector of bits. The first portion of the vector identifies the constructor, and the remainder of the vector is a sequence of values, each also represented in binary form. Where a data type has more than one constructor, different values of the same type may occupy very different amounts of memory.

Because the precise details of bit-vector representation differ from type to type, the obvious mechanism to use is the ad-hoc polymorphism of type classes.

```
class Compress a where
  compress  :: a -> IO (Bin a)
  expand    :: Bin a -> a
```

The type `Bin a` is an *abstract* type, implemented by an extension of the Haskell runtime system. Notice that the `compress` function uses the I/O monad, whereas the `expand` function does not. The I/O monad is used primarily to enforce a correct sequence of operations during compression. Also, `compress` is strict in its first argument. It would be no good to us if objects were compressed lazily, because the whole idea is to save space; a lazy `compress` could easily retain a closure containing the original full-sized value against the day when the compressed version was used! For the same reason however, the expansion operation must remain pure and lazy. It is not known which components of the compressed value will be needed in the computation, and so it does not make sense to enforce strictness or sequencing via the I/O monad. We discuss these design choices further in the Future Work section.

2.2 Implementation issues

In order to be able to write instances of `Compress`, we introduce the following primitives.

```
wBin :: Int -> Int -> IO (Bin a)
rBin :: Int -> Bin a -> (Int, Bin b)
```

The intuition is that `wBin s n` writes integer value `n` into a bit vector of width `s`, returning a pointer to the beginning of the vector. Conversely, `rBin s b` reads an integer value of width `s` bits from the vector `b`, returning both the value and a pointer to the remainder of the bit vector beyond the value that has just been read.

Where are bit vectors stored? For our present purposes, it is convenient to store the vectors off the heap, in a separate area of memory which is not garbage-collected. (Again, this pragmatic choice is re-evaluated in the Future Work section.) When creating a bit vector, this area

of memory is treated like a sequential file with an internal state determining where to begin writing the next value. There is no operation to glue two bit vectors together. Rather, the explicit sequence of I/O operations performed during compression ensures that vectors are placed next to each other. The polymorphic return types of `wBin` and `rBin` ensure that the type inference system can regard these simple pointers as correctly typed. The class system ensures type safety by always selecting the correct instances of `compress` and `expand` for the values involved in any particular computation.

2.3 A small example: truth trees

Figure 1 shows a datatype of binary trees of Booleans, together with the instance definitions needed in order to compress it, and an example tree `t`.

In a standard graph representation such as that used in *nhc* [9], each Boolean occupies one word, each `Branch` occupies three words, and each `Leaf` occupies one word. The total space needed to represent `t` is at worst 27 words, or at best (assuming maximal sharing) 16 words. Implementations such as *Gofer* [4] use four words per `Branch`, bringing the total to between 32 and 20 words.

Under the bit-packing scheme, one bit is sufficient to distinguish branches from leaves, and one further bit distinguishes `True` from `False`. In total, the compressed `t` occupies exactly 17 *bits*. This is better than an order of magnitude saving – the compression ratio is between 15x and 60x, depending on word-size and the extent of sharing. Clearly this example is a best-case due to the high compressibility of Booleans, but we shall obtain very worthwhile compression ratios for more realistic applications (see section 3).

```
data Tree = Branch Tree Tree
          | Leaf Bool

instance Compress Bool where
  compress = wBin 1 . fromEnum
  expand   = toEnum . fst . rBin 1

instance Compress Tree where
  compress (Leaf b) =
    wBin 1 0 >>= \x->
        compress b >>
        return x
  compress (Branch l r) =
    wBin 1 1 >>= \x->
        compress l >>
        compress r >>
        return x
  expand c =
    let (i,c') = rBin 1 c in
    case i of
      0 -> Leaf (expand c')
      1 -> let (l,c'') = expand c'
               (r,_)   = expand c''
           in Branch l r

t :: Tree
t = Branch
      (Branch
        (Branch
          (Leaf True)
          (Branch (Leaf False)
                  (Leaf True)))
        (Branch (Leaf False)
                (Leaf True)))
      (Leaf False)
```

Figure 1: Compression for truth trees. The full bit-vector for `t` is 11101100011000100.

2.4 Derived and explicit instances

It would be tedious to write explicit `Compress` instance definitions for every datatype used in a program. Since the compression scheme we have described is very regular, we have modified the *nhc* compiler to generate instances of `Compress` automatically for datatypes with a `deriving` clause, for example:

```
data Tree = Branch Tree Tree
          | Leaf Bool
          deriving Compress
```

The programmer is still free to try more aggressive compression algorithms, by defining custom instances of the `Compress` class. We have experimented with alternative coding schemes where a knowledge of the expected frequency of values can be used to great advantage. For example, we have written a Haskell program which takes a simple value/frequency table for a type and generates a Haskell module containing the appropriate instance declarations for Huffman compression [1]. In our experience of specific applications, Huffman coding can roughly double the compression ratio. There are other fruitful avenues for compression, especially for character strings.

2.5 Limitations on compressible values

It can be seen that the `compress` function is strict. This means that compression is really only suitable for data which is largely static: it may be computed once, but it then remains constant and useful for the rest of the program's run. Examples of such applications have already been noted: a compiler's import table, a natural-language dictionary, the configuration options for a GUI.

There are some other limitations to the representation scheme outlined here.

1. During compression, any sharing in the original structure is lost, because an in-lined copy is made at every site of the sharing. This is inevitable because the purpose of compression is to flatten out the links from the graph structure, keeping only the terminal values and their sequence. As a result of this restriction, neither cyclic nor infinite structures can be compressed.

2. The heap representation of functions cannot be compressed, since a machine address cannot easily be reduced in size. However, it is certainly possible to compress the code itself, expanding it only when it is needed. Just-in-time dynamic compilation is showing good results in this area – see for instance Wakeling's recent work [14]. The main idea is that function code is generated in a compact bytecode representation. This is then expanded at runtime by an on-the-fly compiler into native code which is stored temporarily in the heap. When the heap is full, the native code is thrown away. The bytecode is re-compiled to native code if it is required again.

3. Compressed values are accessed sequentially. For instance, in a compressed binary tree one locates the right subtree by first deciphering the left subtree. This is fine if the left subtree will be used in the same computation anyway, but in general, access to right-lying components is more expensive than to left-lying components. There are at least two ways of improving this situation. Firstly, when constructing a bit-sequence to represent a component, one can precede it with a short bit-sequence representing its length. This costs more space, but allows unneeded components to be skipped over quickly. See section 4.4 for a fuller sketch of this idea in the context of I/O. A second alternative is to choose the level of data-structure at which compression is best employed. Section 3.3 describes a judicious choice in an example application, where the entries at the leaves of a tree are compressed, but the tree structure itself remains in ordinary form. If the tree structure still takes up too much space, one might use an array of binary pointers instead.

3 Example application: a dictionary of types

For a realistic illustration of the value of runtime heap-compression of data, we have chosen a small but significant part of the *nhc* Haskell compiler and made a useful stand-alone tool from it.

When compiling a module which contains `import` declarations, *nhc* reads an interface file for each imported module. The interface file contains only type declarations, instance declarations, and function names annotated with their type. These declarations are needed for type inference in the importing module.

The interface files are stored in textual format (an issue to which we shall return in a later section), which is parsed to an internal tree-like structure for representing types. Under normal circumstances the type data is retained for the type-checking phase of compilation and then discarded. However, we have re-used the interface parser in writing a browser for the type information. Interface files are read and stored in a hashed lookup structure. The user enters function names at an interactive command line, and the tool reports the types for those functions. This is a very simple database application, but the database contains recursively structured information rather than pure text.

```
data IndTree t =
    Leaf t
  | Fork Int (IndTree t) (IndTree t)

itind  :: Int -> IndTree a -> a
--  itind i it
--  returns the i'th entry from the tree it

itmap  :: Int -> (a->b)
          -> IndTree a
          -> IndTree b
--  itmap f it
--  applies the function f to every leaf in the tree it

itmapm :: Int -> (a->IO b)
          -> IndTree a
          -> IO (IndTree b)
--  itmapm f t
--  applies the monadic function f to every leaf in
--  the tree it, returning the result in the I/O monad
```

Figure 2: Index tree and operations.

3.1 Data structure

For the database, we use a simple indexed binary tree type `IndTree`, outlined in Figure 2. In addition to some standard tree operations, we use a version of `itmap` embedded within the I/O Monad, called `itmapm`, which both ensures that the updated tree is built before the program continues (as a way of avoiding a build-up of update-closures [10]), and also permits the mapped function to be in the I/O monad (for instance `compress`).

The entries in this tree are buckets of *(function-name, type)* pairs. We use a simple hash function on every name to produce an index into the tree. If two names collide at the same index, we store both entries in the same bucket. By choosing an appropriately sized tree, the average bucket size is small and hence a bucket can be efficiently represented just as a list.

3.2 Introducing compression

An appropriate level at which to introduce compression to this lookup structure is on buckets of entries: whilst the index structure is relatively small and accessed frequently, the buckets are individually relatively large and are accessed infrequently. Perhaps an equally good choice would be to compress only individual entries within buckets.

Figure 3 gives an outline of the main program. We omit the datatype definitions and requests for derived instances for strings, pairs, and type declaration trees. The program first builds the `IndTree` and then performs lookup on it. It differs from a compression-less program

```
declstree ::
 Int -> [Decl TokenId]
 -> IO (IndTree
         [(String, Decl TokenId)])
leafCompress ::
 IndTree [(String, Decl TokenId)]
 -> IO
    (IndTree
      (Bin
        [(String, Decl TokenId)]))
leafCompress = itmapm compress

main =
  readFiles ".hi" >>= \inp->
  let h = chooseHashTreeSize inp in
  declstree h (parsedecls inp)
    >>= \pt->
  leafCompress pt >>= \cpt->
  browse cpt

browse cpt =
  untilCatch isEOFError
    (putStr "type browser> " >>
     getLine >>= \inp->
     mapM_
       (putStrLn.showDecl.snd)
       (map (select cpt)
            (words inp))
    )

select cpt w =
  filter
    ((==w).fst)
    (expand (itind (hash w) cpt))
```

Figure 3: Type browser tool

only in the definition and use of the monadic function `leafCompress`, and a single application of `expand`, each shown in bold type.

3.3 Results

We test the type-browsing tool by supplying as input the interface file for the Standard Prelude, and then requesting the type of all 260 prelude functions. We first compare speeds (running *nhc*'s byte-code interpreter on a 50MHz microSparc processor). The compressed version inevitably has a slower access rate than the standard version. We then compare memory space, finding the compressed version to be much more compact than the standard version, as expected.

Time
Without compression, it takes 24.75s to read the file, parse it, and build the index tree, and a further 11.55s to retrieve and display all the entries in turn. This gives a rate (discounting initialisation time) of 23.1 entries returned per second, and a total computation time of 36.30s.
With compression, it takes the same 24.75 seconds to read the file, parse it, and build the initial index tree. It takes a further 25.40 seconds to compress the entries into a new tree, followed by 30.0 seconds to retrieve and display all the entries in turn. This gives a rate (discounting initialisation time) of 8.7 entries returned per second, and a total computation time of 80.15s.

Space
Without compression, the parsing stage uses a peak of 250kb, followed by a constant usage of 150kb of heap memory for the lookup stage.
With compression, the parsing stage again uses a peak of 250kb, followed by a constant usage of about 8kb of heap memory for the lookup stage, and about 16kb of off-heap bit vectors, totalling 24kb.
In this example application, the compression ratio is greater than 6x. We have studied some other applications which demonstrate a broadly similar compression ratio.

3.4 Issues raised by the example

The main problem with the program as it stands is that although the compression achieved during lookup is very worthwhile, the space profile is dominated by the requirement for a large initial heap before the data can ever be compressed. A large proportion of both time and space in the computation are being devoted to the initial parsing of the text file. The compression stage also accounts for a significant part of the time, though this is compensated by the subsequent reduction in space usage.

One might wonder whether the 250kb of space needed for parsing a data structure that turns out to need only 150kb is excessive – perhaps due to a poor choice of parser combinators? In fact the combinators used in the example, in common with all the components of *nhc*, were designed for space efficiency [8]. Even carefully crafted text parsers can cause space irregularities: the difficulty of avoiding them emphasises a need for an alternative more efficient mechanism.

A possible solution to the space problem would be to rework the program structure for greater laziness – to compress as we parse, rather than having two essentially separate passes over the data.

However, another solution addresses the time problem as well as the space problem: store the compressed data directly in a binary file. The text file is parsed and compressed once; all subsequent uses avoid this stage and load the binary representation directly. The next section describes how we have added binary file I/O to Haskell in a manner analogous to compression.

4 A Class of Types for Binary I/O

It would be very convenient to be able to perform I/O directly to/from heap memory, in order to store data structures in a file for later use by a different run of the same program, or perhaps to transmit values between two processes. The linked-graph model of the heap makes this tricky to implement however [13]. Some form of *flattening* is required before a value is amenable to storage or transmission.

The only current standard Haskell mechanism for transferring data is by conversion to and from a textual format, using the `Show` and `Read` classes. While having the benefit of readability, this approach can often be very inefficient. Good implementations of the `Read` class are rare, and programmers still frequently write custom parsers for their Haskell data. Even these, as we have seen, can be slow and memory-hungry.

Our scheme for heap data compression offers an alternative flattening operation which follows a uniform pattern and like the textual classes can be derived automatically for almost any datatype (the restrictions are noted in an earlier section). A binary I/O library based on these ideas should be much more efficient than parsing and printing text.

4.1 The programmer's view

As before, we define a type class for values which can be transmitted in binary format.

```
class BinIO a where
  put  :: BinHandle -> a -> IO ()
  get  :: BinHandle -> IO a
```

The type `BinHandle` is an abstract type analogous to the ordinary text-file `Handle`, but specific to binary files. Like `compress`, but unlike `expand`, both `put` and `get` operations return results in the I/O monad, because here we are dealing with true I/O. Also note that `get` does *not* take a pointer to a value as an argument. Rather, it reads values from the file sequentially, starting at the current position recorded in the state of the I/O monad.

Instances of the `BinIO` class are written using the primitives:

```
putBits :: BinHandle
           -> Int -> Int -> IO ()
getBits :: BinHandle
           -> Int -> IO Int
```

The intuition is that `putBits h s n` writes integer value `n` into a field of width `s` bits at the current position in the file denoted by `h`. Conversely, `getBits h s` reads an integer value of width `s` bits from the current position in the file `h`, returning just the value, and implicitly updating the file pointer.

As before, the explicit sequence of I/O operations performed during output or input ensures that components of a value are placed next to each other in the file, and read back in the same order.

Various other auxiliary functions are needed to complete the library, such as the operations to open and close binary files. These just mirror the existing operations in the textual I/O library.

```
openBinFile  :: FilePath -> IOMode
                  -> IO BinHandle
closeBinFile :: BinHandle -> IO ()
```

One point worth mentioning is that binary files, like textual files, are not type safe across runs. That is, one can write a value to a file as one type and read it back as another. We do not attempt to address this question (see however [7]), leaving it to the programmer to do the sensible thing.

4.2 Implementation of buffering

The buffering required for I/O on binary files is more complicated than that for textual files. Textual I/O assumes that every value is transferred as a whole number of bytes, and hence the minimum buffering unit is the byte. With binary I/O it must be possible to transfer a single bit at a time.

The approach taken in our prototype implementation is to layer a second buffer on top of the standard byte-oriented buffers. This second layer consists of a single byte per file: for output, it accumulates bits until it is full, at which time it is flushed into the ordinary byte-oriented system; for input, it receives a byte from the ordinary system, which is then drained bit-by-bit into the Haskell program. We have implemented this mechanism as a Haskell module.

4.3 Truth trees revisited

Recall the binary trees of Booleans from section 2.3. Instance definitions for binary I/O, using the same bit encoding as for compression, are shown in Figure 4. As before, these can be derived automatically by the compiler.

4.4 Bit transfer between memory and files

There is a broad similarity between the `Compress` and `BinIO` classes. The same binary representation can be produced and interpreted by both, whether in memory or on file. One of our aims in developing binary I/O was to use this identity of representation to increase the efficiency of transfer. Yet the situation described so far permits only values in the standard graph-in-the-heap representation to be put into files or retrieved. If the compressed representation is wanted in *both* memory and file, then the value must 'pass through' the standard representation, suffering the process of two very similar translations.

So we need an instance of the `BinIO` class for the memory-compressed `Bin a` types, using special primitives to implement the bulk transfer of the binary values.

```
instance BinIO (Bin a) where
  put = primDirectPut
  get = primDirectGet
```

```
instance BinIO Bool where
  put h =
    putBits h 1 . fromEnum
  get h =
    getBits h 1 >>=
    return . toEnum

instance BinIO Tree where
  put h (Leaf b) =
    putBits h 1 0 >>
    put h b
  put h (Branch l r) =
    putBits h 1 1 >>
    put h l >>
    put h r
  get h =
    getBits h 1 >>= \i->
    case i of
      0 -> get h >>= return . Leaf
      1 -> get h >>= \l->
           get h >>= \r->
           return (Branch l r)
```

Figure 4: Binary I/O instances for truth trees.

```
newtype SizedBin a =
        SB (Int, Bin a)

class Bin a =>
      SizedCompress a where
  sizedCompress ::
          a -> IO (SizedBin a)
  sizedExpand ::
          SizedBin a -> a
  sizedCompress v =
      compress v >>= \bv->
      primBinSize bv >>= \s->
      return SB (s,bv)
  sizedExpand (SB (s,bv)) =
      expand bv

instance BinIO (SizedBin a) where
  put h (SB (s,bv)) =
      put h s >>
      primDirectPut h s bv
  get h =
      get h >>= \s->
      primDirectGet h s >>= \bv->
      return (SB (s,bv))
```

Figure 5: Sized binary values, and efficient bulk I/O.

But this scheme has a hidden difficulty. Bulk transfer can only be efficient if the size of the value is known. If the size is not known, then the transfer must remain interpretive, since `Bin a` values are of variable size. Fortunately, this oversight is easy to correct, at the cost of a little overhead in space. We introduce a new type of sized binary values, and extend the `Compress` class with a subclass `SizedCompress` which simply attaches size information to compressed values (see Figure 5).

No other instances of `SizedCompress` need ever be written – the default definitions given in the class declaration are sufficient. Our new instance of `BinIO` for all `SizedBin a` types can now implement bulk transfer correctly, by placing or reading the size of the binary value immediately before the value itself. Note that a size need only be attached to the *outermost* structure of a value: the internal structure within a value may comprise many compressed components stored without sizes.

The extra space needed for storing size information could reduce or even cancel the benefit of compression if applied to many values of only modest size. However, our intention is that a program should only attach sizes to larger compressed values involved in bulk I/O. The larger and more complex a compressed value is, the more valuable it will be to transfer it in bulk rather than interpretively, and the smaller the proportion of space taken up by the size information.

One further, less serious, space cost is associated with bulk transfer of bits. The problem is a potential misalignment of values. Imagine the file pointer is set at bit 3 and the memory compression pointer is set at bit 6, immediately before a transfer takes place. It would be possible to fix the alignment byte-by-byte during the transfer, but this would be detrimental to efficiency. The alternative is to insist that sized binary values are exactly aligned to a byte boundary. On average, this solution costs 3.5 bits per value, since between 0 and 7 extra bits are needed to pad the value to a byte boundary. This is a minor cost compared to what has already been paid to store sizes. In practice, byte-alignment is easy to implement.

4.5 An extension for random-access file I/O

A more radical way to avoid the remaining costs of bulk I/O and memory storage for large data structures is to compute with the data structure held entirely (in compressed form) in a file. This is a common technique in the database world. The main requirement over and above what is already available is the need for random-access to files of compressed data.

We achieve this by an extension to the binary I/O class:

```
class BinIO a =>
      RandomAccessBinIO a where
  putAt :: BinHandle
           -> a -> IO (FilePtr a)
  getAt :: BinHandle
           -> FilePtr a -> IO a
```

The intuition for the new operations is that `putAt` returns a pointer to the start location of the value, and `getAt` uses such a pointer to find the value. Values of the new abstract type `FilePtr a` do not refer specifically to any file. At the implementation level, they contain only a byte offset from the start of the file and a bit offset within the referenced byte. It is up to the programmer to use these file pointers sensibly on the correct files.

```
main =
  readFiles ".hi" >>= \inp->
  if parsingText then
    let h= chooseHashTreeSize inp
    declstree h (parsedecls inp)
      >>= \pt->
    leafCompress pt   >>= \cpt->
    sizedCompress cpt >>= \ccpt->
    openBinFile "db.dat" WriteMode
      >>= \db->
    put db ccpt >>
    closeBinFile db
  else return ()
  >>
  openBinFile "db.dat" ReadMode
    >>= \db->
  get db >>= \ccpt->
  closeBinFile db >>
  let cpt = sizedExpand ccpt in
  browse cpt
```

Figure 6: Direct binary transfer to memory.

One difficult issue is whether it should be permissible to read and write the same random-access binary file. The possibility of intermixed read and write access does complicate the implementation of buffering still further. For the moment, we disallow the possibility, not on point of principle, but to keep things simple.

5 Type dictionary revisited

We return to the type-browser tool of section 3 to illustrate the facilities of binary I/O. Here are two new versions of the tool which use file I/O.

5.1 Direct binary transfer to memory

In the first version of the type tool, not only did textual parsing take a long time, but compression took a similar amount of time. We can eliminate both of these stages of the computation by storing the compressed data structure in a file between program runs. Figure 6 shows the additions to the original program in order to store the entire data structure into a file, and to reload it in subsequent runs.

5.2 Layered compression

One point worth noting about this example is the type of the value stored in the file.

```
ccpt ::
    SizedBin
      (IndTree
        (Bin
          [(String, Decl TokenId)]))
```

Not only are the entries at the leaves of the index tree compressed, but the whole of the index tree itself is compressed too. What happens when a compressed value is compressed again by virtue of residing within another value? The bit-vector is simply copied in-line, without modification. When the outer value is expanded, the inner compressed values simply remain in compressed form, true to the type signature. So in this example, because there are two stages of compression, there are also two stages of expansion. The result of `sizedExpand ccpt` is a tree with a branch structure in the heap but leaves of compressed entries.

5.3 Indexed binary files

So far the indexed data structure containing compressed entries has been loaded (one way or another) into memory. We now illustrate how both the compressed entries and the indexing structure may be stored in files. The intention here is to store data entries in one file, but the file-pointers which reference the data file in a different index file. The index file is flat, containing just a sequence of file pointers. Lookup proceeds as follows: first apply the hash function to the string key, giving an integer n; now read the n'th entry from the index file; and finally use this value as a pointer into the data file to retrieve the true compressed entry. This is a very similar mechanism to that used in many previous systems, for example, the *gdbm* library of C database routines.

Figure 7 shows the version of the program using indexed files. We gloss over the issue of file-pointer arithmetic here, which is easy to program but tedious to read. Pointer arithmetic is needed simply to allow the n'th value of compressed size s to be read from the index file – that is, starting at the $(n\text{-}1)*s$'th bit.

5.4 Results

To test the new versions of the type-browsing tool we again supply as input the interface file for the Standard Prelude, and request the types of all 260 prelude functions.

Time

Reading the entire structure from binary file into memory, it takes 0.20s to read the file and perform a first-stage decompression, and a further 30.24s to retrieve and display all the entries in turn. This gives a rate (discounting initialisation time) of 8.6 entries returned per second, and a total computation time of 30.44s.

When the index structure is stored in and read from two files, it takes 36.74s to read the text, parse it, build the initial index tree, and write it out to the new files, then a further 32.01s to retrieve and display all the entries in turn. This gives a rate (discounting initialisation time) of 8.1 entries returned per second, and a total computation time of 68.75s.

```
treeIO ::
    BinHandle -> BinHandle
    -> IndTree
           [(String, Decl TokenId)]
    -> IO ()
treeIO datf indf =
  itmapm
    (\v->
      sizedCompress v >>= \bv->
      putAt datf bv   >>= \fp->
      put indf fp)

main =
  ...
  treeIO datf indf hsize pt >>
  ...
  browse datf indf

browse datf indf =
  untilCatch isEOFError
    (putStr "type browser> " >>
     getLine >>= \inp->
     mapM_
       (\w->
         select datf indf w >>=
         mapM_
           (putStrLn.showDecl.snd)
       ) (words inp)
    )

select datf indf w =
  getAt indf (hash w) >>= \fp->
  getAt datf fp >>= \e->
  return (filter ((==w).fst)
                 (sizedExpand e))
```

Figure 7: Indexed binary files.

When the index structure is kept only in the precomputed files, it takes 0.09s to prepare for reading, and a further 32.01s to retrieve and display all the entries in turn. This gives a rate (discounting initialisation time) of 8.1 entries returned per second, and a total computation time of 32.10s.

Space

When the entire structure is read from a binary file into memory, there is no initial parsing stage. Memory usage peaks at 10kb of heap, and averages at 8kb of heap, plus a constant 16kb of off-heap bit vectors. When the index structure is kept only in the precomputed files, again there is no parsing stage. Memory usage peaks at 4.5kb of heap, and averages at 2.5kb of heap. There no bit vectors in memory. The files occupy 16kb (data) and 408 bytes (index).

A complete comparison with the earlier versions of the tool is shown in Table 1. (All figures are for *nhc*'s bytecode interpreter measured on a 50Mhz microSparc processor; we achieve nearly identical speed ratios on a newer and faster 150Mhz R4000. The compression ratio of course remains constant across platforms.) It can be seen clearly that once the initial work has been done to store a data structure as a binary file, it is much quicker to read the binary file than to re-parse the textual equivalent. Access to individual entries is slower, because the work of interpretation has been deferred until the moment of expansion. Even so, a complete traversal of the compressed structure takes less time than the original version of the tool took to parse a text file and then traverse the lookup structure.

initial I/O	time (s)		space (kb)	
+ data repn.	set-up	queries	set-up	queries
text+heap	24.75	11.55	250	150
text+bits	50.13	30.01	250	8+16
binfile+bits	0.20	30.24	10	8+16
text+binfile	36.74	32.01	240	150
none+binfile	0.09	32.01	4	2+16

Table 1: Time and space costs for different versions of the type-dictionary program when the type of every function in the prelude is requested.

6 Related work

The Haskell language definition, prior to version 1.3, had a `Binary` class with two methods, `showBin` and `readBin`, converting data to and from a `Bin` datatype – the intended use was primarily for binary file I/O. We know of no implementations which supported the class, and the idea has now been dropped from the language standard. Two improvements on the previous design, introduced by our classes, are: parameterisation of `Bin` on the type being represented; and the choice of either interpretive I/O direct between uncompressed values in memory and a binary representation on disk (avoiding an intermediate binary representation in memory), or fast transfer of the binary representation.

The *hbc* compiler has a library which defines a class for `Native` conversions. The methods convert between a value and a list of bytes, which can then be used in textual I/O. There are three differences from our compression class: the data representation is byte-oriented, rather than bit-oriented; the byte vectors are otherwise untyped; and there is no use of monadic sequencing to control the evaluation order of conversion. The intended use of the `Native` class is for foreign language interfaces, and data transmission between processes (either via the file system or across a network). Our scheme permits a flexible style of data compression in addition to these applications.

Johan Jeuring is working on a *polytypic* scheme for data compression [3]. There is a close correspondence between polytypic programming and type classes. (Jansson and Jeuring's language system *PolyP* [2] is the first to provide facilities for polytypism.) His method separates a value's structure from its content, compressing the structural component in a very similar manner to our scheme, and then relying on standard textual methods to compress the content.

There is of course a wide literature on compression algorithms – see for example the comprehensive survey by Lelewer and Hirschberg [5].

Other work on reducing the amount of space used for data representation in functional languages (although without true compression) includes a significant body of work on *unboxing*; see for instance [6].

7 Conclusions and Future Work

A declarative approach to bit-level data representation opens up whole new application areas for functional languages. We have implemented mechanisms for the following types of computation, and demonstrated examples of their use:

1. Computing with a large data structure held in compressed form in memory.
2. Storage and retrieval of a large data structure to/from a binary file, with full representation in memory.
3. Computing with a large data structure held in compressed form in memory, and its bulk storage and retrieval to/from a binary file.
4. Computing with a large indexed data structure held entirely in files, not in memory.

The compression scheme presented in this paper can be derived automatically for almost all datatypes, following a standard pattern which gives significant space savings. However, it also leaves the programmer free to try more aggressive compression algorithms, by defining custom instances of the `Compress` class.

We suggest that typically, the use of heap compression will be considered by a programmer once the program is complete and has been profiled to identify and correct any space faults [11]. When the profile reveals that there is still a large amount of data occupying the heap, and that this data really is needed, the opportunity to compress it should be taken.

Compression and binary I/O could improve the performance of Haskell compilers. Recall that the type browser tool initially reads machine-generated interface files produced by *nhc*. Röjemo has found through profiling that a significant portion of the time taken to separately-compile a module is spent in reading interface files for imported modules. If the interface files were stored as binary files rather than textually, we conjecture that the compiler would run consistently faster. For human readability, a short and fast program to translate the binary format to text could be provided.

Many other applications could benefit from the ability to manipulate values in compact and binary representations, particularly those where it is desirable to hold a very large amount of information either in main memory or on secondary storage. Examples include databases, natural-language processing, and image processing. One application we intend to study fully is a tracer/debugger for Haskell programs [12]. It is very difficult to construct a complete trace of a large computation because of the huge amount of space required. A compressed store makes the task more feasible, especially if the compressed data structures can be stored progressively into a random-access file.

Other uses of bit-vector representations that we have not yet had time to explore include the description of machine registers for embedded-systems control, and the marshalling of data for foreign-language interfaces and inter-process communication. It could be argued that these latter applications are word- or byte-oriented rather than bit-oriented. However, the flexibility of representing data components at the finest granularity remains useful – for instance consider storing a Huffman-coded string inside an aligned block of bytes for transmission across a network.

One valid criticism of our compression class is that it uses the I/O monad, and so the description of binary representations is somewhat imperative. Since compression is not really doing proper input or output, a different choice, though still retaining the imperative style, would be to use a state-transformer monad. (However, the use of the I/O monad does suggest an opportunity for convergence between compression and binary I/O; see below). The further possibility of a more declarative description of representations deserves some investigation. Such a description would perhaps use basic combinators for juxtaposition (placing two bit vectors side-by-side), alignment, padding, trimming, and so on. One difficulty would be the treatment of strictness – when to force the compression of values.

A second criticism of our compression and binary I/O classes is that if instances are hand-written, they must define functions for conversion in *both* directions. This opens the possibility of errors where the value created by one operation is not correctly read back by the other. The two operations are in some sense inverses of each other. Perhaps a facility to define a single description of the binary representation should be provided, from which both conversion operations could be derived.

This also raises the issue of convergence between the two classes. Binary I/O and compression are very similar, and by default use the same representations. However, the possibility of error is again introduced because custom instances written by hand may fail to match each other. To what extent could the operations be merged into a single class, perhaps by regarding the off-heap bit-vector memory as just a special sort of file? The closest resemblance exists between the classes `RandomAccessBinIO` and `Compress`, yet there is still at least one important difference between them, namely that `expand` is a pure function while `getAt` is in the I/O monad.

In our implementation we chose to store bit vectors out of the heap, in a separate area of memory, without garbage collection. As noted above, this lends itself to the treatment of bit-space as just a special kind of file. However, a different implementation could allocate space for bit vectors in the heap, allowing full garbage-collection of compressed values. This would be attractive for some applications, for instance a network server which compresses

packets of data for transmission but then discards them. A heap-based implementation of bit vectors would cost a small amount of extra space for tagging the compressed values, and we conjecture that it could cost a significant amount of speed too. We intend to develop this alternative design for comparison with the current system.

Finally, there is a distant possibility that data compression and expansion could be applied to parts of the heap automatically. Where in our scheme the programmer must judiciously select data structures and apply the compression and expansion functions textually, perhaps in future the memory management system will be able to identify long-lived portions of the heap and transparently compress them during garbage collection, allowing re-expansion lazily by need.

Acknowledgements

Thanks are due to Canon Research Centre Europe Ltd., who have wholly funded this work. We also acknowledge the referees' very full and helpful comments on the first draft of this paper.

References

[1] D. A. Huffman. A method for the construction of minimum-redundancy codes. *Proc. IRE*, 40:1098–1101, 1952.

[2] P. Jansson and J. Jeuring. Polyp – a polytypic programming language extension. In *Proc. 24th ACM Symp. on Principles of Programming Languages (POPL'97)*, pages 470–482, Paris, January 1997. ACM Press.

[3] J. Jeuring. Polytypic data compression. *In preparation*, 1997.

[4] M. P. Jones. The implementation of the Gofer functional programming system. Technical report, Department of Computer Science, Yale University, May 1994.

[5] D. A. Lelewer and D. S. Hirschberg. Data compression. *ACM Computing Surveys*, 19(3):261–296, September 1987.

[6] S. L. Peyton Jones and J. Launchbury. Unboxed values as first class citizens in a non-strict functional language. In J. Hughes, editor, *Proc. 5th ACM Conf. on Functional Programming Languages and Computer Architecture (FPCA'91)*, pages 636–666, Cambridge, MA, August 1991. Springer LNCS 523.

[7] M. Pil. First class file I/O. In W. Kluge, editor, *Proc. 8th Intl. Workshop on Implementation of Functional Languages (IFL'96)*, pages 341–350, Bonn, Germany, September 1996. Institute of Computer Science, Christian-Albrechts-University, Kiel.

[8] N. Röjemo. *Garbage collection and memory efficiency in lazy functional languages*. PhD thesis, Department of Computer Science, Chalmers University of Technology, Sweden, 1995.

[9] N. Röjemo. Highlights from nhc – a space efficient haskell compiler. In *Proc. 7th Intl. Conf. on Functional Programming Languages and Computer Architecture (FPCA'95)*, pages 282–292, La Jolla, June 1995. ACM Press.

[10] C. Runciman. A virtual terminal. In C. Runciman and D. Wakeling, editors, *Applications of functional programming*, pages 60–73. UCL Press, 1995.

[11] C. Runciman and N. Röjemo. Heap profiling for space efficiency. In J. Launchbury, E. Meijer, and T. Sheard, editors, *2nd Intl. School on Advanced Functional Programming*, pages 159–183, Olympia, WA, August 1996. Springer LNCS Vol. 1129.

[12] J. Sparud and C. Runciman. Tracing lazy functional computations using redex trails. *submitted to PLILP'97*, April 1997.

[13] I. Toyn and A.J. Dix. Efficient binary transfer of pointer structures. *Software — practice and experience*, 24(11):1001–23, 1994.

[14] D. Wakeling. A throw-away compiler for a lazy functional language. In M. Takeichi and T. Ida, editors, *Fuji Intl. Workshop on Functional and Logic Programming*, pages 287–300, Susono, Japan, July 1995. World Scientific.

[15] M. Wallace. *Functional programming and embedded systems*. DPhil Thesis YCST 95/04, Department of Computer Science, University of York, January 1995.

[16] M. Wallace and C. Runciman. Lambdas in the Liftshaft — functional programming and an embedded architecture. In *Proc. 7th Intl. Conf. on Functional Programming Languages and Computer Architecture (FPCA'95)*, pages 249–258, La Jolla, June 1995. ACM Press.