Orthogonal Serialisation for Haskell

Jost Berthold

Department of Computer Science
University of Copenhagen
berthold@diku.dk

Abstract. Data serialisation is a crucial feature of real-world programming languages, often provided by standard libraries or even built-in to the language. However, a number of questions arise when the language in question uses demand-driven evaluation and supports higher-order functions, as is the case for the lazy functional language Haskell. To date, solutions to serialisation for Haskell generally do not support higher-order functions and introduce additional strictness.
This paper investigates a novel approach to serialisation of Haskell data structures with a high degree of flexibility, based on runtime support for parallel Haskell on distributed memory platforms. This serialisation has highly desirable and so-far unrivalled properties: it is truly orthogonal to evaluation and also does not require any type class mechanisms. Especially, (almost) any kind of value can be serialised, including functions and IO actions. We outline the runtime support on which our serialisation is based, and present an API of Haskell functions and types which ensure dynamic type safety of the serialisation process. Furthermore, we explore and exemplify potential application areas for orthogonal serialisation.

1 Introduction

Serialisation of data is a crucial feature of real-world programming systems. Being able to write out and read in data structures without complications provides a simple and straightforward way of saving and restoring an application's configuration, and generally enables a program's state to persist between runs. Mainstream and scripting languages like Java, Ruby, Perl and Python provide powerful libraries for this purpose by default, concentrating on ease of use (consider e.g. Python's pickle interface or the omnipresent JSON format for object-oriented web programming). There is considerable interest and demand for general serialisation features in Haskell. In the past year, we have come across related requests on mailing lists or talked about it in personal discussion at several opportunities [1, 12, 14, 13]. For the functional world, the characteristic equal treatment of a program (functions) and its (heap) data poses additional challenges to the design and implementation of serialisation features. Functions (in particular *higher-order functions*) are first-class citizens and should be serialisable as other data. Another complication is added if the language in question uses demand-driven evaluation: how should serialisation treat *unevaluated data*?

J.Hage and M.Morazan (Eds.), IFL 2010, LNCS 6647, 2011.

The lazy purely functional language Haskell [9] has both properties, yet existing serialisation approaches for Haskell do not preserve either of them.

To overcome this deficiency, this paper proposes and explores a new approach to serialisation of Haskell data structures, based on runtime support for parallel Haskell. Implementations of parallel Haskell variants for distributed memory systems obviously require to transfer data from one heap to another. This transfer is based on breadth-first packing and unpacking of graph structures from the Haskell heap, which can represent Haskell data structures of various kind, independent of the evaluation state. In other words, the parallel runtime includes an internal built-in serialisation mechanism with novel and interesting properties. In this paper, we explore the potential and technical limits of using the parallel Haskell graph packing routines to serialise data in Haskell:

- In Section 3, we propose a serialisation mechanism based on runtime support, which is orthogonal in two ways:
  + Orthogonal to evaluation. That is, both normal form and non-normal form data can be serialised.
  + Orthogonal to type. That is, values of any data type can be serialised, no type class mechanisms are required. The notable exception are special concurrency types (MVar and transactional variables), serialisation of which is not supported by the runtime system.
- We show how to re-establish dynamic type safety for this type orthogonal serialisation support of the runtime system, by suitable Haskell wrapper code.
- In Section 4, we propose and exemplify application areas for our approach which cannot be realised using other serialisation approaches.

2 Related Work

As already mentioned in the beginning, serialisation is a common standard feature present in many programming languages. We put the focus of our discussion on approaches specific to lazy functional languages, considering mechanisms for serialisation and persistence.

A very simple, yet unsatisfying, serialisation (and thereby data persistence) is to use the Show and Read class instances. Data types which define instances for these classes can simply be written to a string (`show`) and parsed back in (`read`). Efficiency can be drastically improved by using binary I/O (various libraries have been proposed), and integrated approaches like [25], which uses additional type classes, offer more programming comfort. More recently, we also see efforts to improve efficiency by reproducing the original sharing structure when reading in data [6].

However, all these approaches require to completely evaluate the data which are serialised, and serialisation of infinite or cyclic data structures will send the program into an infinite loop[1]. The real challenge in combining serialisation and

[1] In the Haskell wiki [11], we have found allusions to a library SerTH that allegedly supports cyclic data structures and uses template haskell. The provided links are

laziness is to serialise only partially evaluated values. Only if data evaluation and data serialisation are truly orthogonal, a library for persistence can be established where previous evaluations are reused later. Efforts have been made in the past to join lazy functional languages and persistence in an orthogonal way.

McNally [19, 18] has pioneered these questions with Staple, a programming system with a purely functional interpreter interface, related to Miranda and the then-upcoming Haskell. In an integrated whole-system approach, Staple supports two concepts of persistence: a store of persistent modules which evolve to more and more evaluated state transparently, and interactive stream persistence which allows the user to explicitly create, retrieve and update persistent values in the store. Stream I/O instead of monads and the simplistic interpreter interface characterise Staple as early pioneering work in the field. To our knowledge, no existing system is similar to the Staple persistent store, but it has set directions for several successors.

One such strand of work is Persistent Haskell [24, and earlier: [7]], which describes concepts for adding persistence to the GHC runtime, based on the GUM [26] runtime support for packing which we are using subsequently as well. As in GUM, special closures with fetch semantics are used for retrieving persistent data, which is stored in an external generic persistent object store. While these mechanisms remain completely transparent, the programming interface to the system requires to explicitly open a store and retrieve and store values. The approach of the authors is based on the same runtime system features as ours, yet the paper stays with a high-level design and does not present working solutions to the inherent problems (some of which are nevertheless discussed). One essential advantage is, however, that their design will preserve sharing across several serialised values.

The system which comes closest to what we outline here is the way Clean provides lazy dynamics [27]. Clean dynamics both solve the problem of runtime typing and retain the laziness of stored data, while allowing to transfer data between different applications. However, this transfer feature for unevaluated data requires an integrated systemic design around an "application repository" which contains all functions referenced by persistent data (we will see why this is necessary in the technical main part).

Very limited support for dynamics is included in the Haskell base library [10] as well, in `Data.Dynamic`. We mention it here because we are going to use the underlying `Data.Typeable` to solve typing problems in our approach. Based on the module `Data.Typeable` which provides runtime type reification and guarded type casts, data can be converted to a `Dynamic` and back to its original type (failing at runtime if the type is wrong). Haskell Dynamics are very limited, since a Dynamic can only be used in the same *run* of the same program (binary). In practice, this limits their use to up- and downcasting elements for a uniform container type.

---

however not accessible any more. This library seems to have perished (including all substantial information about its implementation techniques).

3 Implementation

3.1 Runtime System Support

Our implementation of heap data serialisation is based on functionality needed for the Eden [17] and GUM [26] parallel variants of Haskell, namely graph packing. The runtime system (RTS) for Eden and GUM contains methods to traverse and pack a computation graph in the heap in breadth-first manner, and the respective counterpart for unpacking. This is by far the most crucial and error-prone part of the parallel Haskell implementations Eden and GUM [17, 4, 26]. At the same time, it needs to be integrated part of the RTS. Outsourcing its functionality into a library appears to require rather specialised and complex supporting runtime features (we investigated in [3] and [2]).

Heap closures in GHC are laid out in memory as a header section containing metadata, followed by all pointers to other graph nodes, and then by all non-pointers[2]. The essential and first header field is the *info pointer*, pointing to information as e.g. the amount of pointers and non-pointers in the particular heap closure, and the entry code.

When packing data, the respective computation graph structure is traversed and serialised. During the traversal, unevaluated data (thunks) are packed as the function to apply and its arguments, so they can be evaluated on the receiver side. Newly met closures are packed as their header data and non-pointer data. Pointers will be reestablished from the global packet structure when unpacking. Cyclic graph structures are broken up by using back references into the previous data. I.e. when a closure is met again, the routine will not pack the actual closure data again, but a special marker (REF) for a back reference and the relative position of the previously packed closure in the packet. As this requires a starting marker field, normal closures will start by another marker (CLO). A third type of marker indicates static closures (constant applicative forms).

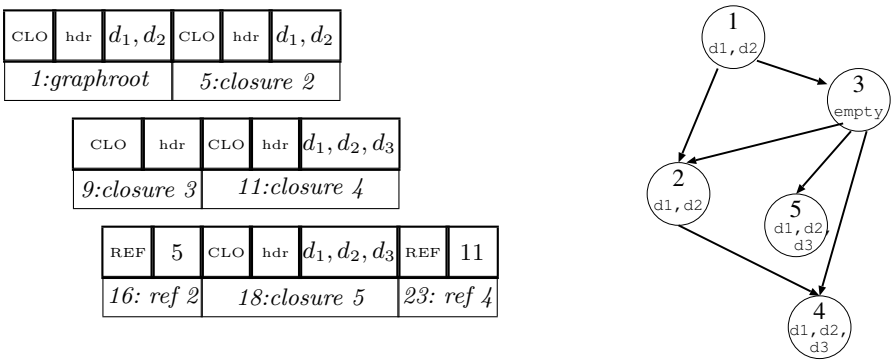

Fig. 1. Example serialisation packet for a computation graph

[2] There are some heap closures which use a *bitmap layout*, mixing pointers and non-pointers. We only describe the common case with standard layout here.

Figure 1 shows an example packet (on the left) which serialises a subgraph of five nodes (depicted on the right), with a line of explaining comments below the packet data. For simplicity, header data is assumed to be just one word, the info pointer. The graph traversal is breadth-first from left to right. As indicated, closures 2 and 4 appear twice in the packet, the second time only as references. All pointer fields are left out in the packet; they will be filled with references to the subsequent closures upon unpacking, using a closure queue.

As the reader might already have noticed from the description, this packing routine relies on the closure layout information and info pointers being pre-shared between the sender and receiver. The packing algorithm in its current form also uses static memory addresses of the program's (and libraries') functions. In consequence, the algorithm assumes that the exact same binary is receiving the data. This limitation is discussed at the end of this section. Furthermore, packing may currently fail because a graph structure in the heap is bigger than a configurable maximum buffer size. The RTS could reallocate a bigger buffer in this case, but this is not implemented yet.

On the plus side, graph packing works on most closure types in GHC and is therefore to a large extent independent of the data type to be packed. The notable exception are primitive data structures for synchronisation, mutable variables (MVars) and transactional variables, which cannot be packed. Typically, packing and transfering data structures which contain MVars does not make sense, since they imply and store external system state. Examples include IO file handles, as well as semaphores and other synchronisation structures. Apart from this restriction, serialisation is completely orthogonal to the Haskell type system.

3.2 Heap to Array: Unsafe Dynamics

As the first step towards our Haskell serialisation, we define a primitive operation which, instead of sending a serialised graph over the wire, returns the data in a Haskell byte array. A byte array is a primitive data type in GHC which provides a chunk of heap for unboxed non-pointer values. This type is the basis of every higher-level array which contains unboxed values (raw data, as opposed to "boxed" values which are stored in pointer arrays).

Primitives and IO-monadic wrapper

```
serialize# :: a -> State# s -> (# State# s, ByteArray# #)
deserialize# :: ByteArray# -> State# s -> (# State# s, a #)

heapToArray :: a -> IO (UArray Int Word)
heapFromArray :: UArray Int Word -> IO a
```

Shown here are the types of the primitive operations and two small wrapper functions which provide an IO-monadic version and lift the result type to an unboxed array (`UArray`) of Words. These two functions `heapToArray` and `heapFromArray` provide the minimum wrapper around the primitive operation: the functions return a regular Haskell data structure in the IO monad.

When used in a disciplined manner, these functions can already prove useful, for they contain all necessary data to reconstruct the serialised structure in its current evaluation state. However, no type safety is provided: any type of value can be serialised and later unpacked and used as any other data type, interpreting the raw data in the heap closures in an incorrect way. A programmer might well be aware of this problem, but make wrong assumptions on the type inference defaults in the compiler, as the following example illustrates.

An unintended type cast

```
lucky = do let list = map (2^) [30..40]          -- defaults to [Integer]
           pack <- heapToArray list
           ...
           copy <- heapFromArray pack            -- type of copy??
           putStrLn (show (length copy : copy))  -- fixes copy :: [Int]
                                                 -- (length :: [a] -> Int)
```

In our example, a list of whole numbers is packed (its default type is a large (GMP-based) `[Integer]`), and most list elements will exceed 32bit in size. When unpacking the data, explicit type annotations or the type context of the copy (here, the added `length copy`) might lead to interpreting it as `[Int]` instead. Only the first number in the list retains its correct value 2^{30}, and only because `Integer`s are handled by machine instructions and stored in a format similar to `Int` when they are small enough. For the following (larger) numbers, the data structures in the Haskell heap are GMP-specific. Their addresses are misinterpreted as 32bit `Int` values, leading to the following wrong program output:

Output when running `lucky`*:*

```
[11,1073741824,28788544,45565760,45565760,45565760,45565760,45565761,...]
```

Unpacking a serialised value into a value of the wrong type can lead to all kinds of runtime errors. The subsequent code makes false assumptions about its heap representation and enters the data structure at the wrong point. In the best case, the two types have a "similar" underlying heap representation, but might still be misinterpreted, as in our example above. Other cases might lead to complete failure.

3.3 Phantom Types: Type-safe Dynamics in one Program Run

In order to provide more type safety, we wrap the array containing the serialised data inside a structure which uses a phantom type:

Serialisation data structure for type safety

```
data Serialized a = Serialized { packetSize :: Int
                               , packetData :: ByteArray#}
serialize   :: a -> IO (Serialized a)
deserialize :: Serialized a -> IO a
```

Passing the original type to a `Serialized` type constructor in this way ensures that the type checker refuses ill-typed programs with a meaningful error message. The type is not arbitrary any more, but included in the data structure passed to the deserialisation function (as a *phantom type*, at compile time). Data can be restored in a typesafe manner within the same program run now. In our example, the type `[Int]` inferred from concatenating the unpacked list to an `Int` propagates up to the original (exposing the `Int` overflow one can expect).

Type propagating up

```
works = do let list = map (2^) [30..40]      -- inferred: [Int]
           pack <- serialize list             -- inferred: Serialized [Int]
           ...
           copy <- deserialize pack
           putStrLn (show (length copy : copy)) -- fixes the type: [Int]
                                                -- (length :: [a] -> Int)
-- output: [11,1073741824,-2147483648,0,0,0,0,0,0,0,0,0]
```

3.4 Type-safe Persistence

With the previously presented code in place, an obvious idea is to store and later retrieve the array which represents the serialised data. Essentially this is what we need in order to realise *persistence*, i.e. keeping partially evaluated data in an external store and loading it into a program. Using `Show` and `Read` instances with `read . show == id` for the `Serialized a` type allows one to write a representation to a file as a string and parse it. Much more space efficient, yet conceptually equivalent, the `Binary` interface can be used. Two problems become apparent when doing so, only one of which can be solved easily.

Dynamic Type Checks. The *first problem* is again one of typing: when reading `Serialized a` from a file and deserializing the represented data, the type a of these data has got to be accurate, either given by the programmer or inferred. Since a user can (attempt to) read from any arbitrary file, this type check needs to happen at runtime. Figure 2 shows how to realise this dynamic type check using the runtime type reification provided by `Data.Typeable`,

First, the `Serialized` data structure now needs to include type information.

```
data Serialized a = Serialized { packetSize :: Int
                               , packetType :: TypeRep
                               , packetData :: ByteArray#}
```

Second, the represented type has got to be established and checked when reading in and deserialising data. With the phantom type, the right place to do this type check is inside the `Read` instance for `Serialized`, requiring a `Typeable` context (instances can be automatically derived with GHC). Data loaded into the running program must have the appropriate type, and the program will otherwise halt with a runtime error and report the type mismatch.

```
instance Typeable a => Show (Serialized a)
  where ... -- showing packet in human-readable and parsable format.

parseP :: ReadS (Int,String,[Word]) -- Parser matching the Show format
parseP =  ...  -- not shown. Returns packet size, type string, values.

instance Typeable a => Read (Serialized a)
    where readsPrec _ input
           = case parseP input of
              [((size,tp,dat),r)] ->
                    let !(UArray _ _ _ arr# ) = listArray (0,size-1) dat
                        t = typeOf (undefined::a)
                    in if show t == tp
                           then [(Serialized size t arr# , r)]
                           else error ("Type error during parse: "
                                       ++ show t ++ " vs. " ++ tp)
              other -> error "No parse"
```

Fig. 2. Serialisation structure, `Read` instance

Figure 3 shows the definition of a binary instance for type `Serialized`, using the same type check mechanism, and file interface functions `encodeToFile` and `decodeFromFile`, in analogy to `encodeFile` and `decodeFile` provided by the Binary module itself.

```
instance Typeable a => Binary (Serialized a) where
    put (Serialized sz tp bArr#)
        = do let typeStr = show tp
                 arr     = UArray 0 (sz-1) sz bArr# :: UArray Int Word
             put typeStr
             put arr
    get = do typeStr <- get :: Get String
             uarr    <- get :: Get (UArray Int Word)
             let !(UArray _ _ sz bArr#) = uarr
                 tp = typeOf (undefined :: a) -- for type check
             if (show tp == typeStr)
                   then return ( Serialized sz tp bArr# )
                   else error ("Type error during parse:\n\tExpected "
                               ++ show tp ++ ", found " ++ typeStr ++ ".")

encodeToFile :: Typeable a => FilePath -> a -> IO ()
encodeToFile path x = serialize x >>= encodeFile path

decodeFromFile :: Typeable a => FilePath -> IO a
decodeFromFile path = decodeFile path >>= deserialize
```

Fig. 3. Serialisation structure, `Binary` instances

Using `Typeable` in our implementation now restricts the serialisation to monomorphic types. Furthermore, the check compares not the `TypeRep` itself, but its string representation – `TypeRep` is just an ID that changes from run to run. Both limitations are introduced by the GHC implementation of `Data.Typeable`.

Please note that the place of our type check is essentially different to dynamics in Clean [27], where a pattern match on types is performed at the time of unpacking a value from dynamic (corresponding to `deserialize`). Clean's approach is more liberal and even allows to "try" several different type matches in one function, as well as polymorphism through the dynamic apply function.

References to Raw Memory Addresses. The *second*, more serious *limitation* of the approach is the use of static information and memory addresses (info pointers and static functions) in the packing code. The packing algorithm in its current form directly uses the memory addresses of functions in the program and in libraries which it uses, as well as static layout information. The latter could be easily fixed by duplicating this layout information in the packet. However, directly using code addresses in memory assumes that the exact same binary is receiving the data and that no code relocation takes place.

Dynamic assignment of code addresses (relocatable code) can be dealt with by packing offsets to a known reference point (as also mentioned in [24]). Another possibility is to inspect the application at runtime using binary utilities like `nm`. However, if an application is recompiled after making changes to its code, the addresses of static data and the compiler-generated names will necessarily change, thereby invalidating previously produced packet data without a chance of correction for the new binary.

Well-understood, the ability to store and retrieve only partially evaluated data is the main advantage of our proposal, and this property *conceptually requires* to keep references to a program's functions if they are needed in the serialised computation. Clean [27] achieves this by a system-wide application store which contains all code referenced from a saved dynamic, requiring special tools for its maintenance (transfer to other machines, deletion, garbage collection of the application store). We consider this a slightly too system-invasive and demanding solution.[3] It would be better to achieve a compromise design where serialisation packets stay independent and self-contained (at the price of higher failure potential at runtime).

To enable data transfer between several applications (and there is no fundamental difference between this and data exchange between different versions of one application), big changes to the packing format will be necessary. Not only does the code need to replace static addresses by relative references, the packet needs to include the static data itself, metadata as well as chunks of code for functions of the program, which has to be dynamically linked at runtime. In contrast, it is easy to make the system produce an understandable runtime error

[3] In a personal conversation [23], the author came to know that in practice, Clean uses a "reduced" version of Dynamics without application store, which poses similar problems.

```
checkpoint :: Typeable a => FilePath -> IO a -> IO a
checkpoint name actions = encodeToFile name actions >> actions

recover :: Typeable a => FilePath -> IO a
recover name = doesFileExist name >>= \b ->
               if b then decodeFromFile name >>= id
                    else error ("No checkpoint file " ++ name)
```

Fig. 4. Basic checkpointing constructs

message when loading data with the wrong version. This can be achieved by including a hash value of the program code into the `Serialized` data and checking that value while parsing, as another dynamic consistency check. All this can be realised purely at the Haskell level.

4 Potential Applications

4.1 Checkpointing Long-running Applications

Checkpointing is the process of storing a snapshot of an application's current state during runtime, in order to restart the application in this state after interruptions.

With the presented serialisation approach, we can realise a simple checkpointing mechanisms to recover long-running Haskell applications from external failures and interruptions, by serialising suitable parts of the running program (as a sequence of IO actions) and storing them for a potential recovery.

Figure 4 shows two utility functions which we propose for this purpose. The first function `checkpoint` serializes the given sequence of IO `actions` (which might be the whole remainder of a program at a certain stage of program execution) and executes it afterwards. Previous variable bindings referred to by the sequence are automatically included in this serialisation, in their current evaluation state. The second function loads an IO action from a file and executes it. To achieve good performance, both functions use the binary file interface functions defined in Figure 3.
These functions can now be used directly at the top level, as sketched here:

```
main = do args <- getArgs
          if not (null args) && head args == "-r"
           then recover "checkpt"
           else do x1 <- computation_1 args
                   actions_1 x1
                   checkpoint "checkpt" $ do
                     actions_2 x1
                     x2 <- computation_2 x1
                     checkpoint "checkpt" $ do
                       more_actions x1 x2
                       ...
```

However, the program needs to be structured in a somewhat unelegant "continuation-capturing" style: Every `checkpoint` needs to capture the *entire* remainder of the program. The example code uses the IO monad, and checkpoints therefore have to be established at the very top level; the calling context of a subordinate function cannot be accessed from inside that function. A slightly more elegant solution would be to use the continuation monad instead of the IO monad, and a monad transformer to embed IO actions. A `callCC`-like mechanism can then capture and serialise the entire program continuation – but again, the program needs to be rewritten to use monad `ContT () IO ()` instead of `IO` in every function that involves establishing checkpoints.

So, our diagnosis is: To equip a program with failover safety through checkpoints requires considerable restructuring by the programmer. That said, it seems promising to provide "checkpointed" versions of monadic computation combinators like `sequence` and `mapM` to support and facilitate this manual restructuring (the type of the checkpoint functions in Figure 4 is based on `IO a` instead of a unit return `IO ()` precisely to allow this). A checkpoint will typically be established at a particular stage of execution, for instance after each step of an iteration. Figure 5 shows how to realise a combinator `sequenceC` where previous results are accumulated and stored in a checkpoint after each step of a monadic sequence. A monadic `mapM` with checkpoints can be expressed using `sequenceC`, and used as shown in the `example`. Introducing a number of checkpoints to a program remains an explicit restructuring task for the programmer, but such checkpointed IO-monadic combinators greatly facilitate the job.

```
sequenceC :: Typeable a => FilePath -> [IO a] -> IO [a]
sequenceC name ms = seqC_acc [] ms
    where seqC_acc acc   []   = return (reverse acc)
          seqC_acc acc (m:ms) = do x <- m
                                   checkpoint name (seqC_acc (x:acc) ms)
-- build mapM_C with sequenceC:
mapMC :: Typeable b => FilePath -> (a -> IO b) -> [a] -> IO [b]
mapMC name f xs = sequenceC name (map f xs)

-- usage example:
example args = -- either the recover option "-r" has been given...
               let doRecover = not (null args) && head args == "-r"
               -- or we read an argument n (if any given)
                   n = if null args then 10 else read (head args)::Int
               in do xs <- if doRecover then recover "seqC_test"
                                        else mapMC "seqC_test" doIt [1..n]
                     putStrLn (show xs)
    where doIt :: Int -> IO Int
          doIt x = ... -- an expensive computation
```

Fig. 5. Combinator for checkpointed sequence, with usage example

4.2 Persistent Memoisation for Frequently Used Applications

A second potential application area for the approach (with its present limitations) can be to alleviate computational load of frequently used applications in a production environment, by *memoisation*.

Memoisation [20] is a well-known technique to speed up function calls in a computer program by storing results for previously-processed inputs. This technique is particularly well suited for languages that support higher-order functions: automatic memoisation can be readily provided by a library function. For the language Haskell, several such libraries exist [15, 8, 21, 5], with only small differences between them: Both Elliot's `MemoTrie` library [8] and Palmer's `Memocombinators` [21] use a Trie-based store [15], another library we have found [5] relies on Haskell Generics [16].

Extending these memoisation techniques to more than one run of the program is possible using our proposed serialisation. *Persistent memo tables* for supportive data structures and repeatedly used functions can be built up, serialised to persistent storage at shutdown, and loaded (deserialised) into memory at startup. In order to realise persistent memoisation, a program using memoised functions has to be extended by suitable init and shutdown mechanisms. The shutdown routine will serialise all memoised functions to a file, with their memo tables obtained so far. The init routine of a subsequent run will then load these existing memo table dumps if they exist, or otherwise use the "fresh" definition in the program.

```
f :: Integer -> Integer
f = memo f' -- assuming a suitable memoisation library
 where f' x = ... -- can use f (not f') recursively

main = do haveFile <- doesFileExist "f_memo.cache"
          f_memo <- if haveFile then decodeFromFile "f_memo.cache"
                                else return f
          ...
          -- all code must use (and pass around) f_memo
          ...
          encodeToFile file_f f_memo
```

Fig. 6. Usage example for a memoised function in a program

A simple way to realise this is sketched in Figure 6. Function `f_memo` is either loaded from a serialised file or used from the original definition in a file, before doing any work. Then, after the main execution, this function is serialised into a file to be loaded in the next program execution.

A drawback of the code shown here is that the memoised function is loaded inside the main function, and has to be passed around as an argument to every function which was originally using `f`. This can be remedied by loading the file

```
{-# NOINLINE f_memo2 #-}       -- this will be executed once, because
f_memo2 = unsafePerformIO $ do -- of lazy evaluation.
     haveFile <- doesFileExist "f_memo2.cache"
     if haveFile then decodeFile "f_memo2.cache" >>= deserialize
                 else let f = memo f'
                          f' x = ... -- can use f recursively
                      in return f
main = do ...                                  -- f_memo2 can be used for f
          ...                                  -- in the entire program
          encodeFile "f_memo2.cache" f_memo2 -- and is saved at shutdown
```

Fig. 7. Less intrusive, however `unsafe` memoisation variant

inside f as an "unsafe global" (using `unsafePerformIO`, see [22]), as we show in Figure 7. In this way, the function will be loaded from the given file automatically. Saving it at shutdown still needs to be done explicitly, but the program can remain otherwise identical to the non-memoised version. However, the memo effect in this version relies on the compiler pragma not to inline the function – especially not at the place where the file is written at shutdown.

We have experimented with the different memoisation libraries and did proof-of-concept implementations for all of them, and also for a naïve list-based memoisation. For the latter, care had to be taken to prevent the compiler from optimising away the memoisation effect by aggressive inlining or let floating, but the memoisation libraries generally provide suitable compiler pragmas. `MemoTrie` [8] appears to be the most widely accepted, but `Memocombinators` [21] produced slightly smaller memoisation data for our test program (fibonacci). In all, the differences between these libraries are very minor, and all can be used with our approach to memoise functions persistently across program runs.

5 Conclusions and Future Work

We have presented an approach to serialisation for Haskell data structures which is orthogonal to both evaluation and data types, and can be fitted with dynamic type checks by suitable Haskell wrapper functions. Haskell heap structures representing data can be serialised independently of their evaluation state, and restored later by the same program (but potentially in a different run of it). Our approach is, to a large extent, also orthogonal to data types; no special serialisation type class is used. By Haskell wrapper functions using the `Typeable` class, we have re-established type safety at runtime, ensuring that ill-typed usage will be at least dynamically detected and results in meaningful runtime error messages instead of unspecific internal failure.

The base implementation for our approach is directly carried over from previous research in the area of parallel Haskell implementations. As such, our approach has conceptual and technical limits; the most relevant being that data can only be read by the very same version of a program. This limitation notwith-

standing, we have pointed out and demonstrated important application areas for the approach in its current form. Easy-to-use checkpointing constructs for iterative computations in the IO monad have been proposed, and we have shown how memoisation techniques can be made persistent with only minor effort. Very few related approaches have been been investigated in the past, and to our knowledge, no previous work has investigated the technical details in comparable depth or exemplified the application areas as we did here.

As future work, we plan to investigate how the Haskell code and the RTS routines should be extended to provide better support for serialisation. The version checksums for serialised data which we have briefly outlined appears to be a straightforward extension at the Haskell level. In contrast, considerable modifications to the packing routine need to be made in order to support data exchange between several (versions of) applications. It would require to make the packet format entirely self-contained and to avoid dependence on any static data; necessarily a very long-term goal. A way to achieve this could be to include dynamically linkable code chunks in the packet. Parallel Haskell implementations for distributed memory systems can as well profit from these improvements, in the form of extended platform independence and flexibility.

Availability. The runtime support on which we build our implementation is available as part of the Eden-version of GHC[4], currently at release 6.12.3. Haskell code is available on request from the author.

Acknowledgements. We would like to thank all colleagues from Marburg, Edinburgh and St.Andrews who were and are supporting the development of parallel Haskell on distributed memory platforms. Special thanks go to Phil Trinder, who brought up the initial idea to work on serialisation as a separable strand, and to an anonymous referee who pointed us to the continuation monad.

References

1. Augustsson, L.: Personal communication (September 2009), about a possible Haskell serialisation feature, during the Haskell Symposium 2009.
2. Berthold, J.: Explicit and implicit parallel functional programming: Concepts and implementation. Ph.D. thesis, Philipps-Universität Marburg, Germany (June 2008), http://archiv.ub.uni-marburg.de/diss/z2008/0547/
3. Berthold, J., Loidl, H.W., Al Zain, A.: Scheduling Light-Weight Parallelism in ArTCoP. In: Hudak, P., Warren, D. (eds.) PADL'08. LNCS, vol. 4902, pp. 214–229. Springer, Heidelberg (2008)
4. Berthold, J., Loogen, R.: Parallel Coordination Made Explicit in a Functional Setting. In: Horváth, Z., Zsók, V. (eds.) IFL'06. LNCS, vol. 4449. Springer, Heidelberg (2007)
5. Claessen, K.: Memoisation module based on Haskell generics. available online, http://www.cse.chalmers.se/~ms/TR0912/Memo.hs, accessed 2010-03-15
6. Corona, A.: Refserialize-0.2.7: Write to and read from Strings maintaining internal memory references. Haskell Library on Hackage, http://hackage.haskell.org/package/RefSerialize, accessed 2010-10-21

[4] http://www.mathematik.uni-marburg.de/~eden/

7. Davie, T., Hammond, K., Quintela, J.: Efficient Persistent Haskell. In: Clack, C., Hammond, K., Davie, T. (eds.) IFL'98 – Draft Proceedings. London, UK (September 1998), available online
8. Elliott, C.: Memo trie library. Haskell Library on Hackage, http://hackage.haskell.org/package/MemoTrie, accessed 2010-10-20
9. Haskell 2010 Language Report. edited by Simon Marlow. Available online (June 2010), http://www.haskell.org/
10. Haskell Hierarchical Libraries: Base library, version 4.2.0.2. Haskell Library on Hackage, http://hackage.haskell.org/package/base, accessed 2010-10-21
11. Haskell Wiki. Wiki, http://www.haskell.org/haskellwiki/, accessed 2010-10-20
12. Haskell Café: Discussion on "bulk synchronous parallel". accessed 2010-07-20, http://www.haskell.org/pipermail/haskell-cafe/2010-April/076593.html
13. Haskell Café: Discussion on "how to serialize thunks?". accessed 2010-07-23, http://www.haskell.org/pipermail/haskell-cafe/2006-December/020786.html
14. Haskell Café: Discussion on "persist and retrieve of IO type?". accessed 2010-07-20, http://www.haskell.org/pipermail/haskell-cafe/2010-April/076121.html
15. Hinze, R.: Memo functions, polytypically! In: Jeuring, J. (ed.) Proc. of 2nd Workshop on Generic Programming, WGP'00 (Ponte de Lima, Portugal, July 2000). pp. 17–32. Tech. Report UU-CS-2000-19, Utrecht Universiteit
16. Lämmel, R., Jones, S.P.: Scrap your boilerplate: a practical design pattern for generic programming. ACM SIGPLAN Notices 38(3), 26–37 (Mar 2003)
17. Loogen, R., Ortega-Mallén, Y., Peña-Marí, R.: Parallel Functional Programming in Eden. Journal of Functional Programming 15(3), 431–475 (2005)
18. McNally, D.J.: Models for Persistence in Lazy Functional Programming. Ph.D. thesis, University of St.Andrews (1993)
19. McNally, D.J., Davie, A.J.T.: Two models for integrating persistence and lazy functional languages. ACM SIGPLAN Notices 26(5), 43–52 (1991)
20. Michie, D.: 'Memo' functions and machine learning. Nature 218, 19–22 (1968)
21. Palmer, L.: Memo combinator library (data-memocombinators). Haskell Library on Hackage, http://hackage.haskell.org/package/data-memocombinators, accessed 2010-10-20
22. Peyton Jones, S.: Tackling the Awkward Squad: monadic input/output, concurrency, exceptions, and foreign-language calls in Haskell (2002), available online at http://research.microsoft.com/~simonpj/
23. Plasmeier, R.J., Koopman, P.: Personal communication (September 2010), about the Clean Dynamics implementation in practical use and its pragmatic limitations, during IFL 2010.
24. Quintela, J.J., Sánchez, J.J.: Persistent Haskell. In: Moreno-Díaz, R., Buchberger, B., Freire, J.L. (eds.) EUROCAST 2001. LNCS, vol. 2178, pp. 657–667. Springer, Heidelberg (2001), presented earlier, at IFL '98, as [7]
25. Santos, A., Abdon Monteiro, B.: A Persistence Library for Haskell. In: Musicante, M.A., Haeusler, E.H. (eds.) SBLP'2001 - V Simpósio Brasileiro de Linguagens de Programação. Proceedings. Curitiba (May 2001)
26. Trinder, P., Hammond, K., Mattson Jr., J., Partridge, A., Peyton Jones, S.: GUM: a Portable Parallel Implementation of Haskell. In: PLDI'96. pp. 78–88. ACM Press, Philadephia (1996)
27. Vervoort, M., Plasmeijer, R.J.: Lazy Dynamic Input/Output in the Lazy Functional Language Clean. In: IFL'02. LNCS, vol. 2670. Springer, Heidelberg (2003)