# Type-Safe and Portable Support for Packed Data

University of Kent, Canterbury, UK

University of Kent, Canterbury, UK

## Abstract

When components of a system exchange data, they need to serialise the data so that it can be sent over the network. Then, the recipient has to deserialise the data in order to be able to process it. These steps take time and have an impact on the overall system's performance.

A solution to this is to use **packed data**, which has a unified representation between the memory and the network, removing the need for any marshalling steps. Additionally, using this data representation can improve the program's performance thanks to the data locality enabled by the compact representation of the data in memory. Unfortunately, no mainstream programming languages support packed data, whether it's out-of-the-box or through a compiler extension.

We present `packed-data`, a Haskell library that allows for type safe building and reading of packed data in a functional style. The library does not rely on compiler modifications, making it portable, and leverages meta-programming to allow programmers to pack their own data types effortlessly.

We evaluate the usability and performance of the library, and conclude that it allows traversing packed data up to 60% faster than *unpacked* data. We also reflect on how to enhance the performance of library-based support for packed data.

Our implementation approach is general and can easily be used with any programming languages that support higher-kinded types.

**2012 ACM Subject Classification** Information systems → Data layout

**Keywords and phrases** program optimisation, data structures, data layout, packed data

**Digital Object Identifier** 10.4230/LIPIcs.ECOOP.2025.38

**Category** Experience Paper

**Supplementary Material** *Software (Source Code)*: `https://github.com/Arthi-chaud/packed-data` [8], archived at `swh:1:dir:027d47580c14b18a2969e17a715e86c2094246ad`

## 1 Introduction

Components within a system may need to exchange data. Consequently, data needs to be sent in a format that is processable by the recipient. For example, web application programming interfaces (web APIs) usually serialise data using the Java-Script Object Notation (JSON) before sending it over the network. On the other hand, the receiver of the data has to deserialise the response into a processable object; only then will it be able to use the data. For example, a web client has to parse the JSON received before being able to, say, compute the average length of the usernames in a list.

The most common serialisation formats used for these kinds of interactions are JSON, XML and HTML. They all have a simple, human-readable syntax, and it is very easy to serialise and deserialise data with them. However, because of their syntax, these formats make the data considerably larger, leading to a higher memory footprint and longer transfer times. Additionally, de/serialisation has a cost and can impact the performance of large exchange-heavy systems.

© Arthur Jamet and Michael Vollmer;
licensed under Creative Commons License CC-BY 4.0

Alternatively, such components could use a binary format to exchange data. For example, one could use Google's Protocol Buffers (Protobuf)[1], which generates code (for many languages) to serialise/deserialise data to/from a binary format. This leads to smaller payloads (compared to human-readable formats), and thus smaller transfer times. On the other hand, Cap'n Proto[2] allows using binary data as-is, without any marshalling steps, but its encoding leads to bigger buffers compared to Protobuf's.

*Packed* data is binary data that can be used as-is, safely and without any deserialisation step. Thus, it can be easily persisted to and read from the disk without any conversion cost. Additionally, the minimal layout information contained in the packed data leads to a smaller overhead compared to Cap'n Proto. Moreover, unlike objects in memory, packed data does not have pointers and nested fields are inlined within the parent structure. Consequently, the data locality enabled by the packed format can leverage better performance (e.g. using the L1 cache). This can help speed up programs that traverse large data trees, like compilers with abstract syntax trees (AST) or web browsers with document object models (DOM).

The current state-of-the art is Gibbon, a compiler that transforms programs that use pointer-based structures into programs that use this packed representation [14, 13, 9, 11]. Gibbon supports a small subset of the Haskell programming language, and compiles recursive functional programs operating on trees into C code operating on packed bytestrings. For example, Figure 1b shows the memory layout of a binary tree (defined in Figure 1a) for a C program. Gibbon's packed representation of that same tree (defined in Figure 1c) is given in Figure 1d: instead of containing pointers, the parent node contains its children directly, contiguously in memory.

Writing C code for traversing packed data by hand is not trivial, and requires raw and untyped pointer manipulations, leading to code that is hard to read and maintain. Figure 2 gives an example of such code, which computes the sum of the leaves in a packed tree. Using a specialised compiler like Gibbon can alleviate this concern by generating the tricky code automatically (and using a typed intermediate language that verifies packed data layout invariants [13]), but introducing a dependency on an experimental research compiler is not realistic for some software projects. Additionally, Gibbon supports only a small subset of Haskell, lacking many convenient features, and also requires whole-program compilation, making integration into larger programs awkward.

This data layout has another a drawback: it does not allow accessing members of a data structure easily. Since fields are inlined and can have variable sizes (e.g. for recursive data types), they do not have a predefined position (or offset) within the parent structure. For example, consider a tree similar to the one from Figure 1, but where the root node's children are swapped. We know that the leaf is *after* the node, but we do not know the node's size or where it ends. The only way to access the leaf would be to traverse the first child. Depending on the size of that tree, it can lead to catastrophic performance issues. One solution to this problem is to use *indirections.* In Gibbon, indirections are numbers inserted (when requested) in the packed buffer before each field of a data structure. These numbers represent the size in bytes of the field they precede [13]. Indirections make the packed data larger, but they help avoid useless traversals of (potentially big) preceding fields before getting to the target one.

While the benefits of packed data have been proven, Gibbon is one of the only compilers that use this data layout natively. But Gibbon is not the only tool to leverage a condensed data layout: we already mentioned high-level libraries like Cap'n Proto, which allows using packed

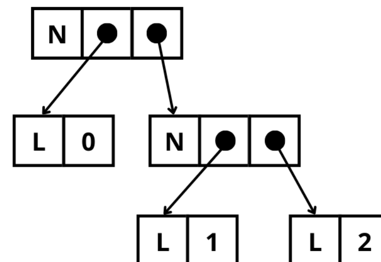
[1] `https://protobuf.dev`
[2] `https://capnproto.org`

```c
enum Tag { LEAF, NODE };

struct Tree {
  enum Tag tag;
  union {
    // Content of a leaf
    struct {
      long value;
    };

    // Content of a node
    struct {
      struct Tree *left;
      struct Tree *right;
    };
  };
};
```

**(a)** Definition of the tree, in C.




**(b)** Pointer-based representation.

```haskell
data Tree = Leaf Int
          | Node Tree Tree
```

**(c)** Definition of the tree in Gibbon/Haskell.

**(d)** Packed representation.

**Figure 1** Pointer-based and packed representation of a data tree.

data as-is, and Protobuf, to pack/unpack data. On the compiler-level, the Glasgow Haskell Compiler (GHC) also supports packing data, through *compact normal forms* (CNF) [15], but we can't manipulate it (e.g. access a field) in this form. Packed data remains to be leveraged by mainstream programming languages. The performance improvements enabled by this data format remain limited to experimental research projects.

In this paper, we present a new effort of supporting packed data through a library. `packed-data` is a Haskell library for building and traversing packed data. Inspired by the example packed data interface from Bernardy et al.'s paper [4], the library contains a code generator that allows the programmer to easily use their own algebraic data types (ADT). We use Haskell in this paper, but our approach is not specific to the Haskell programming language. Thus, the library does not require any compiler modifications and can be implemented in any language with higher-kinded types. We present the design of this library along with a use-case for packed data (Section 2). Then, in Section 3, we evaluate the usability and performance of our library-based approach. Finally, we reflect on the limitations of our design in Section 4 and suggest improvements in Section 5.

The source code of `packed-data` is available on Hackage[3]. The package also includes the code of the benchmarks we present in Section 3.2.

Unless specified, all code snippets in this paper use the Haskell syntax.

[3] `https://hackage.haskell.org/package/packed-data`

```
long sum(char **t) {
  long res = 0;
  if (**t == LEAF) {
    *t = *t + 1;
    res = *((long *)*t);
    *t = *t + sizeof(long);
  } else if (**t == NODE) {
    *t = *t + 1;
    res = sum(t);
    res = res + sum(t);
  }
  return res;
}
```

**Figure 2** Computing the sum of the values in a packed tree, in C.

# 2 The `packed-data` Library

In this section, we will go through and explain the features of the `packed-data` library. Then, we will give an example of a writing operation on packed data, and finally we will cover a use-case of packed data, implemented with the library.

## 2.1 Features

As its name suggests, `packed-data` allows creating and manipulating packed data. Since the library is implemented in Haskell, we leverage the language's type-system to guarantee that packed data manipulation is type-safe. Thus, any bad use of a packed buffer will be detected and rejected at compile-time. Additionally, we hide pointer arithmetic details behind a functional API. This allows programmers to use packed data like they would use data streams, instead of a raw and untyped memory buffer. The library also includes a code generator which generates all the necessary functions to pack, traverse and unserialize custom ADTs. Finally, `packed-data` allows programmers to choose whether indirections are inserted in the packed data.

### 2.1.1 Packing Data

Packing data can be done in two ways: by using an intermediate typed buffer, or by serializing an object that already exists.

Consider a `Tree` ADT isomorphic to the one defined in Figure 1c. The code sample in Figure 3a uses the `Needs` intermediary buffer and functions like `startNode` and `writeLeaf` (generated automatically for the `Tree` type, see Section 2.1.3). Alternatively, the code snippet in Figure 3b shows how to *pack* a tree using the `pack` function. The buffers produced by these two approaches are identical.

The `Needs` type represents intermediary buffers that *needs* to be written to, before one can use them as packed data. Its definition in Figure 4 is inspired by Bernardy et al.'s examples [4]. `Needs` has two types parameters: the first one is the list (denoted `'[...]`) of the types of the values that the buffer expects before being considered *ready*, and the second type argument is the types of the values packed in the buffer once reified. These parameters allow for type-safety when writing data and producing the final buffer. By using lists as type parameters, we can represent multiple values packed next to each other in a single buffer.

```haskell
myPackedTree :: Packed '[Tree]
myPackedTree = finish needs
    where needs = withEmptyNeeds
        $
         do
            startNode
            writeLeaf 1
            startNode
            writeLeaf 2
            writeLeaf 3
```

**(a)** Using the "start" and "write" functions.

```haskell
-- Defined in packed-data
pack :: Packable a =>
    a -> Packed '[a]
pack = ...

-- Generated automatically
instance Packable Tree where
    ...

myTree :: Tree
myTree = Node
        (Leaf 1)
        (Node (Leaf 2) (Leaf 3))

myPackedTree :: Packed '[Tree]
myPackedTree = pack myTree
```

**(b)** Using the "pack" function.

**Figure 3** How to pack a value with `packed-data`.

For example, `Needs '[Int] '[Tree]` represents a buffer that *needs* a single integer, before it can be reified as a packed `Tree`. `Needs '[] '[Char, Int]` represents a buffer that is ready to be reified; it would contain a packed character, followed by a packed integer.

When a `Needs` is *ready* or *full* (i.e. its first type argument is an empty list `'[]`), we can produce the final packed buffer with `finish :: Needs '[] t -> Packed t`.

Finalised buffers take the form of a `Packed` value. This data type, defined in Figure 4 has a single type parameter which tells the type of values packed inside. For example, `Packed '[Tree, Tree]` is a buffer that contains two packed `Trees`.

```haskell
newtype Needs p t = Needs ByteString

newtype Packed p = Packed ByteString
```

**Figure 4** Definition of the "Needs" and "Packed" data type.

### 2.1.2 Traversing Buffers

We need a way to traverse packed data. To do so, we defined `PackedReader`, a computation modelling a cursor, *reading* from left to right. `PackedReader` is parameterised by the type of the `Packed` data to read, the type of the data after the cursor has been shifted, and the type of the value produced by the computation. For example, in Figure 5a, we define a function that computes the sum of two integers packed in the same buffer.

To illustrate the security that type-safety provides, consider the code snippet 5b, where we try to read a `Char` where the packed value is actually an integer. This code does not type-check, and an error will be raised at compile-time. This showcases how type-safety prevents programmers from reading values incorrectly and bad shifts of the `PackedReader`'s cursor.

`PackedReader` is actually an *indexed* monad, where the type parameters represents the state of the computation [2], i.e. the expected types of the data to traverse, and the types of the packed data after the cursor has been moved.

```
readInt :: PackedReader
    '[Int] r Int

sumInts :: PackedReader
    '[Int Int] '[] Int
sumInts = do
    i1 <- readInt
    i2 <- readInt
    return (i1 + i2)
```

**(a)** Reading Packed integers.

```
readChar :: PackedReader
    '[Char] r Char

example :: PackedReader
    '[Int, Int] '[] Int
example = do
    i1 <- readInt
    c <- readChar -- Type error
    return i1
```

**(b)** Invalid reading of packed values.

```
getSum :: Packed '[Int, Int] -> Int
getSum packed = runReader sumInts packed
```

**(c)** Execution of a PackedReader.

**Figure 5** Reading packed values with `PackedReader`.

This allows us to leverage Haskell's type-system to ensure type-safe data manipulation, and the language's native syntax (e.g. `do`-notation) to give the code an imperative feel. To actually execute a `PackedReader` computation, we use the `runReader` function. An example is given in Figure 5c.

The code generator described in Section 2.1.3 provides a "case" function: a `PackedReader` which simulates pattern matching. Figure 6 is an example of a packed tree traversal, using the generated `caseTree` (defined in Figure 8b). This function computes the sum of the leaf values in a packed tree. The first lambda to pass (lines 4-7) is the function to execute if the packed tree is a `Leaf`, and the second one (lines 8-12) is for when the packed tree is a `Node`.

```
1  sumTree :: PackedReader '[Tree] '[] Int
2  sumTree =
3      caseTree
4          ( do
5              n <- reader
6              return n
7          )
8          ( do
9              left <- sumTree
10             right <- sumTree
11             return (left + right)
12         )
```

**Figure 6** Computing the sum of the values in a Tree, using "caseTree".

### 2.1.3 Code Generator

Defining how to encode and decode packed data is a critical aspect of packed data manipulation. Since we deal with bytes, it is very easy to make mistakes, and tools like the type-system cannot be helpful here. `packed-data` takes away the programmer's responsibility to write encoding/decoding functions for their custom data types by providing a code generator.

Powered by Template Haskell [10], this code generator defines the following for a given ADT:

- An instance of the `Packable` typeclass, providing the `pack` function (seen in Figure 3b)
- An instance of the `Unpackable` typeclass, providing the `reader` function, to unpack a value from a packed stream.
- A "case" function, to simulate pattern matching on a packed object (see Section 2.1.2)
- A "transform" function, similar to "case"; it helps generate packed data out of `Packed` values (an example of its usage is given in Section 2.2)

The code generator's entry point also takes as parameter a list of flags, which change its behaviour. Currently, these flags only deal with indirections (described in Section 2.1.4).

To invoke the code generator, the programmer has to call the `mkPacked` function. An example is given in Figure 7.

```
$(mkPacked ''Tree [InsertFieldSize])
```

**Figure 7** Invocation of the code generator.

### 2.1.4 Indirections

The code generator accepts two flags:

- `InsertFieldSize`: Inserts indirections before each field of the packed ADT.
- `SkipLastFieldSize`: Does not insert an indirection before the last field. We do not always need to jump over a final field, and not adding an indirection allows saving space in memory.

These flags will change the definition of the functions produced by the code generator. Figure 8 illustrates these changes. Notice in Figure 8a that each field is preceded by a `FieldSize` (a type alias to 32-bit integers we use as indirections), absent from the definition in Figure 8b, generated without passing indirection flags. This type transparency regarding indirections makes it easier to think about how to traverse a data structure: should we traverse everything to get to the nth field, or can we use indirections to jump there?

```
caseTree ::
    PackedReader '[FieldSize, Int] r a ->
    PackedReader '[FieldSize, Tree, FieldSize, Tree] r a ->
    PackedReader '[Tree] r a
```

**(a)** Definition of the "case" function for Trees, when the "InsertFieldSize" is on.

```
caseTree ::
    PackedReader '[Int] r a ->
    PackedReader '[Tree, Tree] r a ->
    PackedReader '[Tree] r a
```

**(b)** Definition of the "case" function for Trees, when no indirection flags are passed to the generator.

**Figure 8** Changes on the API caused by indirections.

Consequently, code that manipulates packed data changes accordingly to the indirections flags given to the code generator. For example, consider Figure 9 where we traverse a binary tree to get its right-most value. In Figure 9a, we have to *read* each left sub-trees to get to the right one. This is not optimal, as it causes a complete traversal of that sub-tree. However, in Figure 9b, we *skip* over a field using the `FieldSize` that precedes it. One drawback is that we have to explicitly skip `FieldSize`s we do not care about (before an integer value for example).

```
getRightMost :: PackedReader '[Tree] r Int
getRightMost =
    caseTree
        readInt
        ( do
            _ <- readTree
            getRightMost
        )
```

**(a)** Traversing a tree, without indirections.

```
getRightMost :: PackedReader '[Tree] r Int
getRightMost =
    caseTree
        ( do
            skipFieldSize -- Skipping leading indirection
            readInt
        )
        ( do
            skipWithFieldSize -- Skipping the left subtree
            skipFieldSize -- Skipping the right subtree's indirection
            getRightMost
        )
```

**(b)** Traversing a tree, with indirections.

**Figure 9** Tree traversals in `packed-data`.

## 2.2 Example: Incrementing values of a packed tree

The packed data produced by the library is immutable. Since we are working with pointers and raw memory access under the hood, giving writing abilities (even through a monad) would break the purity and referential transparency of the library's API. Writing inside a buffer used by multiple `Packed` values would impact each of these variables, possibly leading to corruption. Just like in functional languages, it is not possible to do in-place modifications on packed data; instead we have to duplicate it.

To do so, we can build a `Needs` value while traversing a packed value. This is what the generated "transform" function is for: it behaves similarly to the "case" function (in the sense that it helps with pattern matching on packed values, and it can be used recursively), but it forces its arguments to be `PackedReader`s that produce `Needs` values. It also helps the programmer by writing leading tags and inserting indirections when necessary.

Figure 10 gives an example of a function that takes a packed tree and produces another where all the leaf values are incremented by one. We use the `transformTree` function (line 3) instead of `caseTree`, as the former invokes the `startLeaf`/`startNode` functions for us and thus

helps keeping the code simple. The first `PackedReader` (lines 4-7) passed to `transformTree` reads the integer of a packed `Leaf` and calls the `write` function from the `Packable` typeclass. The second `PackedReader` (lines 8-12) calls the `incrementPacked` function recursively for each branch and builds back the packed `Node` using the `applyNeeds` function.

```
1  incrementPacked :: PackedReader '[Tree Int] r (Needs '[] '[Tree Int])
2  incrementPacked =
3      transformTree
4          ( do
5              n <- readInt
6              return (write (n + 1))
7          )
8          ( do
9              left <- incrementPacked
10             right <- incrementPacked
11             return (applyNeeds left >> applyNeeds right)
12         )
```

**Figure 10** Incrementing values of a packed tree.

## 2.3 Use case: Web Server

To showcase the perks of using packed data, let's take the example of a very simple web API that has two endpoints "`/sum`" and "`/right-most`". These endpoints respectively compute the sum of the leaves' values and get the right-most value in a packed tree passed to the request body.

Using the Scotty web framework[4], we would implement the endpoints as in Figure 11a. The endpoints are very simple: we get the packed tree, call the corresponding function `sumTree` and `getRightMost` (from Figures 6 and 9a) and send back the result.

The most interesting part of these endpoints is how we get the packed tree (on lines 4 and 9): Scotty's `body` function returns the raw request's body as a *lazy* `ByteString`. However, a `Packed` value relies on a *strict* `ByteString`. Thus, we need to use the `toStrict` function to do the conversion. This operation is $\mathcal{O}(n)$, but considering that a JSON-based API would traverse the entire `ByteString` to parse JSON as well, this is acceptable. `unsafeToPacked` is a $\mathcal{O}(1)$ operation to cast a `ByteString` into a typed `Packed` value. Unlike a JSON-based API, we do not marshal the request body, saving both memory and time.

On the client side (Figure 11b), we enjoy a similar kind of optimisation, as the packed tree can be sent as-is (if the client already uses packed data). Additionally, both `fromPacked` and `fromStrict` (line 5) are $\mathcal{O}(1)$ operations, making the preparation of a request lightweight.

When running this server in parallel with one that uses *unpacked* trees and JSON, we notice that the former runs noticeably faster than the later (Proper benchmarks on packed vs. unpacked data are done in Section 3.2).

[4] `https://hackage.haskell.org/package/scotty`

```
endpoints :: ScottyM ()
endpoints = do
    get "/sum" $ do
        packedTree <- unsafeToPacked . toStrict <$> body
        sum <- liftIO $ runReader sumTree packedTree
        response sum

    get "/right-most" $ do
        packedTree <- unsafeToPacked . toStrict <$> body
        rightMost <- liftIO $ runReader getRightMost packedTree
        response rightMost
```

**(a)** Handling packed data in a web REST API Server.

```
request methodGet "/sum" [] reqBody
  where
    packedTree :: Packed '[Tree]
    packedTree = ...
    reqBody = fromStrict (fromPacked packedTree)
```

**(b)** Sending packed data to a web REST API.

**Figure 11** Packed data in a web REST API.

# 3 Evaluation

To understand the impact of using the library to handle packed data, we evaluate it in two ways. First, we take a look at its usability and portability, by comparing how to manipulate packed data and unpacked data. Then, we look at the performance through a benchmark on simple tree traversals.

## 3.1 Practical considerations

Since computations on packed data are lifted, it imposes limitations on how code can be written. `packed-data` tries to fill the gap between Haskell's features and what one might want to do with packed information, without modifying the compiler. However, this approach has some shortcomings. Let's take a look at five aspects of our library-based approach to evaluate the usability of `packed-data`.

### 3.1.1 Type-safety and maintainability

`packed-data` provides a high-level and type-safe API for low-level data manipulation. It keeps the user from doing raw and unsafe pointer arithmetic by abstracting these operations behind a functional API.

As an example, consider Figures 6, 12 and 2 where we compute the sum of the leaves of a packed tree using `packed-data` and pointer arithmetic in Haskell and C. In the later cases, dereferencing and pointer arithmetic bloats the code with multiple operations on numeric values that are not very expressive. In both languages, the type-checker will not be able to detect any pointer misuse. On the other hand, when using `packed-data` (Figure 6), pointer arithmetic is hidden away behind functions like `reader`, `caseTree` and the `do` expressions,

which guarantee correct pointer manipulation. Therefore, our library makes code that manipulates packed data more readable, maintainable and memory-safe, compared to code that does raw pointer arithmetic.

```
sumPackedNonMonadic :: Ptr Word8  -> IO (Int, Word8)
sumPackedNonMonadic ptr = do
    tag <- peek ptr :: IO Word8
    let !nextPtr = ptr `plusPtr` 1
    case tag of
        0 -> do
            !n <- peek nextPtr
            let !nextPtr1 = plusPtr ptr 9
            return (n, nextPtr1)
        1 -> do
            (!left, !r) <- go (castPtr nextPtr)
            (!right, !r1) <- go r
            let !res = left + right
            return (res, r1)
        _ -> undefined
```

**Figure 12** Summing the values in a tree using raw pointer manipulation in Haskell.

### 3.1.2 Portability

`packed-data` does not require modifying the compiler or the runtime system. This means that the library's code does not deal with the compiler's implementation details. Thus, `packed-data` can virtually be used with any modern version of GHC (as of today, the library has been tested with GHC versions 9.2 to 9.12). The library can be simply used as a third-party package, with no further constraints.

Not relying on the compiler's internal workings also means that `packed-data` could be reimplemented in other languages with similar type systems like Scala, Rust or TypeScript.

### 3.1.3 Code Generator's Applicability

Thanks to its code generator (powered by meta-programming), `packed-data` can be used with any user-defined algebraic data type, with the exception of unboxed types (denoted with a trailing `#`, like `ByteArray#`) and types that use them.

It also requires very little boilerplate code (see Figure 7 from Section 2.1.3). Using a Template Haskell entry point like `mkPacked` is a common approach used by other popular Haskell libraries like Aeson[5] for JSON encoding/decoding.

### 3.1.4 Generated Code and Flags

We saw in Section 2.1.4 that flags could change the definition of the functions created by the code generator, while the names of the generated functions do not change. But what if we wanted to call the code generator twice for the same ADTs, passing different flags? This would lead to the definition of two functions with the same name (e.g. `caseTree`) in the same

[5] `https://hackage.haskell.org/package/aeson`

module, which is not allowed in Haskell. A workaround would be to call the code generator separately in dedicated modules, so that the different definitions are split, and we can import these modules according to our needs.

Another thing to consider is that the `Packed` data type does not us tell whether indirections are present in the packed data. Therefore, if a user wants to build trees with and without indirections, it is up to them to use the correct "case" and "reader" functions generated with the correct flags.

We should also note that the same code cannot be reused with instances of the `Packable` typeclass generated with different indirection flags. Figures 8 and 9 illustrate this problem: because of the strong typing of the packed data, `PackedReader`s need to explicitly skip over indirections (with `skipFieldSize`) when they are present.

### 3.1.5 Pattern Matching

One of Haskell's strengths is its powerful pattern matching support. Unfortunately, its syntax is not overload-able, meaning that we can not use it to pattern match on packed data. As a solution, the code generator defines a "case" function which takes lambdas as parameter, one for each constructor of the ADT (an example is given in Figure 6).

However, an issue arises from not being able to use the standard syntax: we can't do nested pattern matching. In Haskell, we can pattern-match on a constructor's fields, while with our "case" functions, pattern matching only considers constructors.

`packed-data` is a portable solution that provides type-safety for packed data manipulation. It is flexible (with flags for the code generator) and can be easily integrated to most modern Haskell projects. Even though the API sticks to Haskell's functional and pure paradigm, it does not support features like nested pattern matching.

## 3.2 Performance

We evaluate the performance of `packed-data` by comparing the execution time of simple tree traversals with pointer-based Haskell, Gibbon, C and Golang. We focus on four kinds of full tree traversals: computing the sum of values in a tree, evaluating an AST for a simple arithmetic language, getting the right-most node and incrementing the leaves' values. Table 1 reports the execution times of these tree traversals.

We should note that we will not compare the size of the packed data with buffers created by Cap'n Proto. This is because `packed-data`'s serialisation format is very similar to Gibbon's, and Vollmer et al. have already demonstrated that the compiler's packing format is more space-efficient than Cap'n Proto [13]. For the same reason, we will not benchmark Cap'n Proto's traversals, as the same paper showcases that Gibbon's traversals are generally faster than Cap'n Proto. Thus, we can use Gibbon's performance to indirectly compare `packed-data` with Cap'n Proto.

For each benchmark, each tree is built symmetrically, meaning that a tree of "size" 1 will have $2^1 = 2$ leaves, a tree of "size" 5 will have $2^5 = 32$ leaves, etc. The C and Haskell code were benchmarked using the Criterion library[6](with Haskell's Foreign Function Interface (FFI) to invoke the C code with negligible overhead). The Gibbon code was compiled with the `-p` and `--no-gc` options to use a packed layout and disable garbage collection. The Golang

[6] `https://hackage.haskell.org/package/criterion`

code was benchmarked using the `testing` library[7]. Code was compiled using GCC 11.4, GHC 9.10, Golang 1.18 and the latest Gibbon version as of March 2025. All benchmarks were run on an Intel machine with two Xeon Gold 6244 CPUs at 3.60 GHz, with 32Gb of RAM, running Ubuntu 22.04 LTS.

In this section, we will ignore the benchmark cases prefixed with *Non-monadic*, we will get back to them in Section 4.

For summing the values in a tree, using `packed-data` can provide up to a 20% speed-up compared to native Haskell. It also beats C by running 15% faster. We can note that, for this specific scenario, having indirections slightly slows down the traversal (around 2%): we do not need them to optimise the traversal, so we skip over them, leading to additional smaller jumps.

Evaluating an AST is similar to summing the values of a tree, as it requires a full traversal of a tree, without skipping any fields. The AST used for this benchmark is one for a small arithmetic language with integer values, addition, subtraction, multiplication, and division. This means that we will not have to deal with two constructors/tags like for the previous benchmark (for leaves and nodes), but five (one for the value, and one for each arithmetic operation). Similarly to the previous benchmark, using `packed-data` provides a 60% speed-up compared to regular Haskell. The *Unpacking, then traverse* line shows the runtime for functions that unpacks the packed AST, and then evaluates it. This case shows that doing so leads to an 8.5x slowdown compared to native Haskell and illustrates the cost of unpacking a value before processing it, i.e. that `time(unpacking)+ time(eval_unpacked)> time (eval_packed)`.

The third benchmark focuses on getting the right-most value in a tree. Unlike the previous cases, here a lot of jumps will be done, as we do not need to go through each branch of the tree. However, using `packed-data`, the operation is 10000 times slower than regular Haskell, as we need to traverse the entire packed tree to reach the last value. But this benchmarks illustrates the benefits of indirections: when they are present, we can skip over subtrees, avoiding a lot of useless traversals. Indirections allow this computation to be only four times slower than unpacked Haskell.

In the final benchmark, we evaluate the performance of incrementing the leaves' values in a tree. Note that the C and Golang benchmarks increment the trees *in-place*, while the others create a new tree. Here, Haskell is much slower than C, since Haskell's functional paradigm forces such operations to produce an entirely new tree. In `packed-data`, this can be done by using the `traverse` function and calling the `finish` function. The performance of `packed-data` are surprisingly comparable with regular Haskell. This can be justified by the fact that `packed-data` only does one big memory allocation for the entire buffer, instead of one per tree node. In the *Unpacking, increment and repack* case, we deserialise the packed tree, increment the leave's values (using the same function as for the native Haskell case), and then repack the tree. The *Deserialise and increment, and repack* case produces an unpacked tree already incremented (i.e. the leave's values are incremented in the "case" function's lambdas) and then re-serialiases the tree. The latter case has better performance than the former, as it avoids an additional traversal and the creation of a new unpacked tree. Again, these last two cases show that it is faster to use unpacked data as-is, instead of deserialising it first, whether it is for read-only or writing operations.

As a final note, we observed via some preliminary benchmarking that the packed data approach appears to perform worse on Apple ARM devices, where the non-packed (pointer-chasing) code is generally faster. We speculate that this may be due to ARM either handling

[7] `https://pkg.go.dev/testing`

**Table 1** Execution times of tree traversals.

| Language / Tree Size | | 1 | 5 | 10 | 15 | 20 |
|---|---|---|---|---|---|---|
| C | | 7.56 ns | 57.33 ns | 1.76 us | 84.04 us | 6.06 ms |
| Haskell | | 16.21 ns | 202.68 ns | 6.47 us | 207.17 us | 6.56 ms |
| Gibbon | | 23.00 ns | 359.00 ns | 13.97 us | 234.59 us | 6.28 ms |
| packed-data | ① | 20.99 ns | 150.31 ns | 4.86 us | 157.94 us | 5.14 ms |
| | ② | 20.68 ns | 152.82 ns | 4.84 us | 156.19 us | 5.22 ms |
| | ③ | 9.11 ns | 133.68 ns | 4.47 us | 143.39 us | 4.68 ms |
| | ④ | 8.95 ns | 134.16 ns | 4.54 us | 145.97 us | 4.91 ms |
| Golang | | 5.30 ns | 111.30 ns | 3.86 us | 165.05 us | 16.49 ms |

①: Without indirections. ②: With indirections.
③: Non-monadic, without indirections. ④: Non-monadic, with indirections.

**(a)** Execution times for summing the value in a tree.

| Language / Tree Size | | 1 | 5 | 10 | 15 | 20 |
|---|---|---|---|---|---|---|
| C | | 10.74 ns | 121.19 ns | 4.65 us | 181.40 us | 9.24 ms |
| Haskell | | 16.22 ns | 278.23 ns | 8.67 us | 280.97 us | 13.20 ms |
| Gibbon | | 73.00 ns | 723.00 ns | 10.60 us | 181.50 us | 5.34 ms |
| packed-data | ① | 24.08 ns | 168.80 ns | 5.25 us | 168.18 us | 5.41 ms |
| | ② | 27.84 ns | 471.86 ns | 15.69 us | 751.96 us | 111.96 ms |
| | ③ | 10.89 ns | 151.21 ns | 5.11 us | 164.25 us | 5.27 ms |
| Golang | | 5.29 ns | 108.90 ns | 4.42 us | 180.51 us | 15.41 ms |

①: Without indirections. ②: Unpacking, then traverse. ③: Non-monadic, without indirections.

**(b)** Execution times for evaluating a packed AST for arithmetic expressions.

| Language / Tree Size | | 1 | 5 | 10 | 15 | 20 |
|---|---|---|---|---|---|---|
| C | | 7.35 ns | 9.20 ns | 10.75 ns | 28.23 ns | 86.15 ns |
| Haskell | | 11.63 ns | 15.52 ns | 19.45 ns | 23.33 ns | 27.30 ns |
| Gibbon | | 29.00 ns | 39.00 ns | 62.00 ns | 93.00 ns | 74.00 ns |
| packed-data | ① | 21.68 ns | 93.63 ns | 2.80 us | 95.27 us | 3.24 ms |
| | ② | 21.67 ns | 25.85 ns | 32.36 ns | 43.06 ns | 112.77 ns |
| | ③ | 20.94 ns | 215.35 ns | 6.60 us | 349.04 us | 67.94 ms |
| | ④ | 9.29 ns | 77.98 ns | 2.82 us | 95.95 us | 3.32 ms |
| | ⑤ | 8.12 ns | 10.54 ns | 16.52 ns | 26.86 ns | 94.32 ns |
| Golang | | 3.45 ns | 8.58 ns | 18.25 ns | 26.61 ns | 35.05 ns |

①: Without indirections. ②: With indirections. ③: Unpacking, then traverse.
④: Non-monadic, without indirections. ⑤: Non-monadic, with indirections.

**(c)** Execution times for getting the right-most value in a tree.

| Language / Tree Size | | 1 | 5 | 10 | 15 | 20 |
|---|---|---|---|---|---|---|
| C | | 7.33 ns | 35.45 ns | 1.79 us | 82.26 us | 7.68 ms |
| Haskell | | 19.50 ns | 294.18 ns | 10.55 us | 501.31 us | 75.83 ms |
| Gibbon | | 71.00 ns | 635.00 ns | 30.40 us | 486.78 us | 14.42 ms |
| packed-data | ① | 41.99 ns | 469.68 ns | 16.80 us | 648.15 us | 74.50 ms |
| | ② | 60.29 ns | 821.67 ns | 33.45 us | 1.99 ms | 245.64 ms |
| | ③ | 42.57 ns | 580.03 ns | 21.97 us | 1.03 ms | 165.04 ms |
| Golang | | 6.61 ns | 129.60 ns | 4.29 us | 179.44 us | 16.97 ms |

①: Using NeedsBuilder. ②: Unpacking, increment and repack.
③: Deserialise and increment, and repack.

**(d)** Execution times for incrementing the values in a tree.

pointer-chasing within a tree better, or punishing unaligned access more severely. Full investigation of packed data performance on ARM architecture is future work. Regardless, even on ARM, there was a significant benefit to operating directly on packed data rather than the “unpack, process and repack” round-trip.

We conclude that `packed-data` can provide better performance than native Haskell, but not in every case. In the best scenarios, the speed-ups can reach 60%, compared to unpacked data, but in the worst scenarios, it is considerably slower (from 4 to 10000 times). In the next section, we speculate on why sometimes the library performs slower than native Haskell.

# 4 Reflection

We concluded in the previous section that the performance of `packed-data` were satisfactory. This can be explained by the fact that we make the most of the L1 cache and do not jump around using pointers. In this section we consider an alternative way of writing code in Haskell for packed data that can provide better performance than `packed-data`. This approach, however, is far less maintainable and safe.

In Haskell, monads are not defined at the compiler level, they are regular data structures defined in Haskell’s `base` standard library. Therefore, even though they are a big part of the functional paradigm, they do not enjoy any dedicated compiler optimisation.

To manipulate packed data, `packed-data` uses two monads: `IO`, required to do any kind of low-level operation (like dereferencing a pointer using `peek`), and `PackedReader`, which provides the high-level API for packed data traversals.

Since monadic operations can be as optimised as any other kind of computation (e.g. through inlining), inevitably, using two monads on top of each other like `packed-data` must lead to some computing overhead.

We argued in Section 3.1.1 that `packed-data`’s API allowed for readable and type-safe code. Figure 13b is an example of what traversing packed data without `packed-data` would look like. It is safe to say that this code is way less pleasant to read and harder to maintain than with the library (Figure 13a). However, it does get rid of one level of computation induced by the `PackedReader` monad.

Benchmarks using this non-monadic approach are included in Table 1 and labelled as *Non-monadic*. In the summing and AST evaluation benchmarks, this approach yields slightly better performance (between 2 and 6% compared to the monadic approach). However, in the third benchmark, with indirections, it is 19% faster than its `packed-data` counterpart. But it is still 3.5 times slower than regular Haskell, possibly because this computation relies heavily on following indirections and jumping around in the serialized tree. This leads to an extensive use of the `IO` monad, slowing down the traversal.

This confirms that our monadic approach to packed data manipulation causes some kind of computing overhead. However, this is specific to Haskell. Implementations for other languages might not suffer from this, but they should be able to leverage similar performance enhancements. As future work, we could use staging and compile-time evaluation to eliminate that computing overhead.

# 5 Future Work

While `packed-data` is a working library, it can be improved in many ways; or it can serve as a stepping stone for further research on the topic. In this section, we enumerate possible future work that can be done on the library or in this research domain.

```
evalPacked :: PackedReader '[AST] r Int32
evalPacked =
    caseAST
        reader
        (opLambda (+))
        (opLambda (-))
        (opLambda (*))
        (opLambda div)
  where
    {-# INLINE opLambda #-}
    opLambda ::
        (Int32 -> Int32 -> Int32) ->
        PackedReader '[AST, AST] r Int32
    opLambda f = R.do
        left <- evalPacked
        right <- evalPacked
        R.return (f left right)
```

**(a)** Using packed-data.

```
evalPackedNonMonadic :: Packed (AST ': r) -> IO Int32
evalPackedNonMonadic packed = fst <$> go (unsafeForeignPtrToPtr fptr)
  where
    (BS fptr _) = fromPacked packed
    go :: Ptr Word8 -> IO (Int32, Ptr Word8)
    go ptr = do
        tag <- peek ptr :: IO Word8
        let !nextPtr = ptr `plusPtr` 1
        case tag of
            0 -> do
                !n <- peek nextPtr :: IO Int32
                return (fromIntegral n, plusPtr nextPtr (sizeOf n))
            1 -> opLambda (+) nextPtr
            2 -> opLambda (-) nextPtr
            3 -> opLambda (*) nextPtr
            4 -> opLambda div nextPtr
            _ -> undefined
    {-# INLINE opLambda #-}
    opLambda :: (Int32 -> Int32 -> Int32) -> Ptr Int32 -> IO (Int32,
        Ptr Word8)
    opLambda f ptr = do
        (!left, !r) <- go $ castPtr ptr
        (!right, !r1) <- go r
        let !res = left `f` right
        return (res, r1)
```

**(b)** Using pointer arithmetic.

**Figure 13** Evaluating a packed AST.

Building packed data (through a `Needs` buffer) is slow (four times slower than building native objects). Internally, it relies on the `bytestring-strict-builder` library[8] which, in itself provides good performance[9]. However, as for packed data traversals, building packed data is implemented through a monad, `NeedsBuilder`. As discussed in Section 4, even though it provides a functional and type-safe API, such approach can cause some computing overhead. A future version of `packed-data` could focus on making the packing process faster.

`packed-data` only supports its own packing layout. However, some systems could take advantage of data packed with a custom or pre-defined layout. Take JSON for example: the client of an API that serves JSON data would not have to deserialise the server's response, and could use it as is. This is also applicable for web browsers and HTML, especially since web pages can be sizeable. A next step for `packed-data` would be to handle data packed with a more complex layout like JSON, HTML or CBOR[10].

We have seen that accessing a field in a packed structure is not a straightforward task: we have to either use indirections or traverse the preceding fields. We could draw inspiration from Chilimbi, Davison & Larus' work [5] and reorder fields according to their access pattern, moving the most accessed ones first in a packed structure [11]. This would help programs access "popular" fields more easily, with fewer jumps in the packed buffer.

In `packed-data`, indirections take the form of a 32-bit integer. We could definitely save some space in the packed buffer by choosing the size of this indirection accordingly to the following field. For example, we do not need a 32-bit indirection for a four-characters string, one byte would be enough. To go even further, some packed data do not need indirections at all, if their size can be guesses statically (e.g. a packed character will always be one byte, a 16-bit number two bytes, etc.). In that case, indirections are useless. We can avoid inserting them (reducing the size of the packed buffer) and deduce the size of that packed data by relying on the type-checker and a couple of typeclasses.

As mentioned in Section 3.1, we could also allow the same code on packed data to be reused whether the data has been packed with or without indirections.

Another area of improvement would be not to insert indirections in the packed buffer itself, but in a separate heap object. This would allow for packed data to be smaller and enable parallelism in tree traversals.

Finally, because of the data layout used by `packed-data`, memory access can be unaligned. This noticeably affects performance on ARM CPUs. In the future, `packed-data` could include a flag that forces the insertion of padding in the packed buffer, to align tags, indirections and fields.

The portability of `packed-data` across other languages remains theoretical. While its implementation does not rely on modifying the compiler or the runtime system, it does heavily rely on Haskell's type-system. It would be interesting to implement `packed-data` in an other typed language like Scala, TypeScript or Rust.

We concluded in Section 4 that our library-based approach in Haskell suffered from a computing overhead caused by the `IO` monad, blocking any possible future enhancement of the library's performance. Deeply-embedded domain-specific languages (EDSLs) could be a solution to this problem. EDSLs expose functions that do not actually execute an operation,

[8] https://hackage.haskell.org/package/bytestring-strict-builder
[9] https://github.com/haskell-perf/strict-bytestring-builders
[10] https://cbor.io

but instead produce an AST [12]. These ASTs can then be optimised [1] and compiled into a full-blown program in the host language or produce a standalone executable [3]. A few EDSLs have performance as their main objective [1], and to reach it, it's not uncommon to rely on a backend written in a lower-level language, like C [6].

To provide better performance, `packed-data` could be reimplemented as a deeply embedded DSL: at compile time, using Template Haskell, we could produce an AST and generate C code. Using Template Haskell's quotation feature, we could inject the call to that C code into the Haskell program, using the language's C FFI. This way, we would still have a type-safe and functional API, while enjoying C's fast performance [7]. Furthermore, this C code could be reused as a common backend for implementations of `packed-data` in other languages.

## 6 Conclusion

Exchanging data using a packed format can be faster than when using a human-readable format like JSON, as the recipients can use the serialised data as-is, avoiding any deserialisation step. Manipulating packed data can also help save computation time, as it avoids chasing pointers. However, this format is not easy to adopt, as most implementations require either compiler modifications or a dedicated compiler.

In this paper, we investigate library-based support for packed data through `packed-data`. This Haskell library provides a type-safe and functional API that allows packing data, processing it, and unpacking it. It relies on Template Haskell and Haskell's type-system, meaning that this library could virtually be implemented in any language that supports meta-compilation and higher-kinded types.

Our benchmarks show that the library can leverage better performance than native Haskell (up to 60%), but in some scenarios like getting the right-most value in a tree, `packed-data` (with indirections) is four times slower. This is due to a computation overhead caused by the monadic approach used in the implementation, as benchmarks show that "pure" pointer arithmetic-based implementations can provide more competitive performance.

While a library-based approach for packed data support is feasible and portable, there is a performance and usability tradeoff to consider. We think that `packed-data` would benefit from staging and compile-time optimisations, for example, through an EDSL, which would generate and call C code. This would help get rid of computing overhead caused by the host language, and could be reused to build a common backend used by implementations of similar libraries in other languages.

## References


1 Johan Ankner and Josef Svenningsson. An EDSL approach to high performance haskell programming. In Chung-chieh Shan, editor, *Proceedings of the 2013 ACM SIGPLAN Symposium on Haskell, Boston, MA, USA, September 23-24, 2013*, pages 1–12. ACM, 2013. doi:10.1145/2503778.2503789.

2 Robert Atkey. Parameterised notions of computation. *J. Funct. Program.*, 19(3-4):335–376, 2009. doi:10.1017/S095679680900728X.

3 Emil Axelsson and Mary Sheeran. Feldspar: Application and implementation. In Viktória Zsók, Zoltán Horváth, and Rinus Plasmeijer, editors, *Central European Functional Programming School - 4th Summer School, CEFP 2011, Budapest, Hungary, June 14-24, 2011, Revised Selected Papers*, volume 7241 of *Lecture Notes in Computer Science*, pages 402–439. Springer, 2011. doi:10.1007/978-3-642-32096-5_8.

**4** Jean-Philippe Bernardy, Mathieu Boespflug, Ryan R. Newton, Simon Peyton Jones, and Arnaud Spiwack. Linear haskell: practical linearity in a higher-order polymorphic language. *Proc. ACM Program. Lang.*, 2(POPL):5:1–5:29, 2018. doi:10.1145/3158093.

**5** Trishul M. Chilimbi, Bob Davidson, and James R. Larus. Cache-conscious structure definition. In Barbara G. Ryder and Benjamin G. Zorn, editors, *Proceedings of the 1999 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Atlanta, Georgia, USA, May 1-4, 1999*, pages 13–24. ACM, 1999. doi:10.1145/301618.301635.

**6** Frank Dedden. Compiling an haskell edsl to c, 2018. URL: https://studenttheses.uu.nl/handle/20.500.12932/29176.

**7** Conal Elliott, Sigbjørn Finne, and Oege de Moor. Compiling embedded languages. *J. Funct. Program.*, 13(3):455–481, 2003. doi:10.1017/S0956796802004574.

**8** Arthur Jamet and Michael Vollmer. packed-data. Software, version 0.1.0.3. (visited on 2025-05-02). URL: https://hackage.haskell.org/package/packed-data.

**9** Chaitanya Koparkar, Mike Rainey, Michael Vollmer, Milind Kulkarni, and Ryan R. Newton. Efficient tree-traversals: reconciling parallelism and dense data representations. *Proc. ACM Program. Lang.*, 5(ICFP):1–29, 2021. doi:10.1145/3473596.

**10** Tim Sheard and Simon L. Peyton Jones. Template meta-programming for haskell. *ACM SIGPLAN Notices*, 37(12):60–75, 2002. doi:10.1145/636517.636528.

**11** Vidush Singhal, Chaitanya Koparkar, Joseph Zullo, Artem Pelenitsyn, Michael Vollmer, Mike Rainey, Ryan Newton, and Milind Kulkarni. Optimizing Layout of Recursive Datatypes with Marmoset: Or, Algorithms + Data Layouts = Efficient Programs. In Jonathan Aldrich and Guido Salvaneschi, editors, *38th European Conference on Object-Oriented Programming (ECOOP 2024)*, volume 313 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 38:1–38:28, Dagstuhl, Germany, 2024. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi:10.4230/LIPIcs.ECOOP.2024.38.

**12** Josef Svenningsson and Emil Axelsson. Combining deep and shallow embedding for EDSL. In Hans-Wolfgang Loidl and Ricardo Peña, editors, *Trends in Functional Programming - 13th International Symposium , TFP 2012, St. Andrews, UK, June 12-14, 2012, Revised Selected Papers*, volume 7829 of *Lecture Notes in Computer Science*, pages 21–36. Springer, 2012. doi:10.1007/978-3-642-40447-4_2.

**13** Michael Vollmer, Chaitanya Koparkar, Mike Rainey, Laith Sakka, Milind Kulkarni, and Ryan R. Newton. Local: a language for programs operating on serialized data. In Kathryn S. McKinley and Kathleen Fisher, editors, *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2019, Phoenix, AZ, USA, June 22-26, 2019*, pages 48–62. ACM, 2019. doi:10.1145/3314221.3314631.

**14** Michael Vollmer, Sarah Spall, Buddhika Chamith, Laith Sakka, Chaitanya Koparkar, Milind Kulkarni, Sam Tobin- Hochstadt, and Ryan Newton. Compiling tree transforms to operate on packed representations. In Peter Müller, editor, *31st European Conference on Object-Oriented Programming, ECOOP 2017, June 19-23, 2017, Barcelona, Spain*, volume 74 of *LIPIcs*, pages 26:1–26:29. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2017. doi:10.4230/LIPIcs.ECOOP.2017.26.

**15** Edward Z. Yang, Giovanni Campagna, Ömer S. Agacan, Ahmed El-Hassany, Abhishek Kulkarni, and Ryan R. Newton. Efficient communication and collection with compact normal forms. In Kathleen Fisher and John H. Reppy, editors, *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming, ICFP 2015, Vancouver, BC , Canada, September 1-3, 2015*, pages 362–374. ACM, 2015. doi:10.1145/2784731.2784735.