Seamless, Correct, and Generic Programming over Serialised Data

GUILLAUME ALLAIS, University of Strathclyde, UK

In typed functional languages, one can typically only manipulate data in a type-safe manner if it first has been deserialised into an in-memory tree represented as a graph of nodes-as-structs and subterms-as-pointers.

We demonstrate how we can use QTT as implemented in Idris 2 to define a small universe of serialised datatypes, and provide generic programs allowing users to process values stored contiguously in buffers.

Our approach allows implementors to prove the full functional correctness by construction of the IO functions processing the data stored in the buffer.

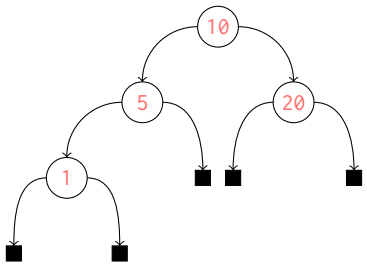
CCS Concepts: • **Computer systems organization** → **Embedded systems**; *Redundancy*; Robotics; • **Networks** → Network reliability.

Additional Key Words and Phrases: functional programming, correct-by-construction, idris, serialisation

ACM Reference Format:
Guillaume Allais. 2018. Seamless, Correct, and Generic Programming over Serialised Data. *J. ACM* 37, 4, Article 111 (August 2018), 31 pages. https://doi.org/XXXXXXX.XXXXXXX

1 INTRODUCTION

In (typed) functional language we are used to manipulating structured data by pattern-matching on it. We include an illustrative example below.




```
data Tree
  = Leaf
  | Node Tree Bits8 Tree

sum : Tree -> Nat
sum t = case t of
  Leaf => 0
  Node l b r =>
    let m = sum l
        n = sum r
    in (m + cast b + n)
```

On the left, an example of a binary tree storing bytes in its nodes and nothing at its leaves. On the right, a small Idris 2 snippet defining the corresponding inductive type and declaring a function summing up all of the nodes' contents. It proceeds by pattern-matching: if the tree is a leaf then we immediately return 0, otherwise we start by summing up the left and right subtrees, cast the byte to a natural number and add everything up. Simply by virtue of being accepted by the typechecker,

Author's address: Guillaume Allais, guillaume.allais@ens-lyon.org, University of Strathclyde, 16 Richmond Street, Glasgow, Scotland, UK, G1 1XQ.

0004-5411/2018/8-ART111 $15.00
https://doi.org/XXXXXXX.XXXXXXX

we know that this function is covering (all the possible patterns have been handled) and total (all the recursive calls are performed on smaller trees).

At runtime, the tree will quite probably be represented by constructors-as-structs and substructures-as-pointers: each constructor will be a struct with a tag indicating which constructor is represented and subsequent fields will store the constructors' arguments. Each argument will either be a value (e.g. a byte) or a pointer to either a boxed value or a substructure. If we were to directly write a function processing a value in this encoding, proving that a dispatch over a tag is covering, and that the pointer-chasing is terminating relies on global invariants tying the encoding to the inductive type. Crucially, the functional language allows us to ignore all of these details and program at a higher level of abstraction where we can benefit from strong guarantees.

Unfortunately not all data comes structured as inductive values abstracting over a constructors-as-structs and substructures-as-pointers runtime representation. Data that is stored in a file or received over the network is typically represented in a contiguous format.

We include below a textual representation of the above tree using `node` and `leaf` constructors and highlighting the data in red.

```
(node (node (node leaf 1 leaf) 5 leaf) 10 (node leaf 20 leaf))
```

This looks almost exactly like the list of bytes we get when using a naïve serialisation format based on a left-to-right in-order traversal of this tree. In the encoding below, leaves are represented by the byte `00`, and nodes by the byte `01` (each byte is represented by two hexadecimal characters, we have additionally once again highlighted the bytes corresponding to data stored in the nodes):

```
                (node (node leaf 1 leaf) 5 leaf)
01  01   01 00 01 00   05 00  0a 01 00 14 00
              (node leaf 1 leaf)
```

The idiomatic way to process such data in a functional language is to first deserialise it as an inductive type and then call the `sum` function we defined above. If we were using a lower-level language however, we could directly process the serialised data without the need to fully deserialise it. Even a naïve port of `sum` to C can indeed work directly over buffers:

```
1  int sumAt (uint8_t buf[], int *ptr) {
2    uint8_t tag = buf[*ptr]; (*ptr)++;
3    switch (tag) {
4      case 0: return 0;
5      case 1:
6        int m = sumAt(buf, ptr);
7        uint8_t b = buf[*ptr]; (*ptr)++;
8        int n = sumAt(buf, ptr);
9        return (m + (int) b + n);
10     default: exit(-1); }}
```

This function takes a buffer of bytes, and a pointer currently indicating the start of a tree and returns the corresponding sum. We start (line 2) by reading the byte the pointer is referencing and immediately move the pointer past it. This is the tag indicating which constructor is at the root of the tree and so we inspect it (line 3). If the tag is 0 (line 4), the tree is a leaf and so we return 0 as the sum. If the tag is 1 (line 5), then the tree starts with a node and the rest of the buffer contains first the left subtree, then the byte stored in the node, and finally the right subtree. We start by summing the left subtree (line 6), after which the pointer has been moved past its end and is now pointing at the byte stored in the node. We can therefore dereference the byte and move the pointer past it (line 7), compute the sum over the right subtree (line 8), and finally add up all the components, not

forgetting to cast the byte to an int (line 9). If the tag is anything other than 0 or 1 (line 10) then the buffer does not contain a valid tree and so we immediately exit with an error code.

As we can readily see, this program directly performs pointer arithmetic, explicitly mentions buffer reads, and relies on undocumented global invariants such as the structure of the data stored in the buffer, or the fact that the pointer is being moved along and points directly past the end of a subtree once `sumAt` has finished computing its sum.

Our goal with this work is to completely hide all of these dangerous aspects and offer the user the ability to program over serialised data just as seamlessly and correctly as if they were processing inductive values. We will see that Quantitative Type Theory (QTT) [Atkey 2018; McBride 2016] as implemented in Idris 2 [Brady 2021] empowers us to do just that purely in library code.

1.1 Seamless Programming over Serialised Data

Forgetting about correctness for now, this can be summed up by the the following code snippet in which we compute the sum of the bytes stored in our type of binary trees.

```
sum : Pointer.Mu Tree _ -> IO Nat
sum ptr = case !(view ptr) of
  "Leaf" # _ => pure Z
  "Node" # l # b # r =>
    do m <- sum l
       n <- sum r
       pure (m + cast b + n)
```

We reserve for later our detailed explanations of the concepts used in this snippet (`Pointer.Mu` in Section 5, `view` in Section 7.4). For now, it is enough to understand that the function is an `IO` process inspecting a buffer that contains a tree stored in serialised format and computing the same sum as the pure function seen in the previous section. In both cases, if we uncover a leaf (`"Leaf" # _`) then we return zero, and if we uncover a node (`"Node" # l # b # r`) with a left branch `l`, a stored byte `b`, and a right branch `r`, then we recursively compute the sums for the left and right subtrees, cast the byte to a natural number and add everything up. Crucially, the two functions look eerily similar, and the one operating on serialised data does not explicitly perform error-prone pointer arithmetic, or low-level buffer reads. This is the first way in which our approach shines.

One major difference between the two functions is that we can easily prove some of the pure function's properties by a structural induction on its input whereas we cannot prove anything about the `IO` process without first explicitly postulating the `IO` monad's properties. Our second contribution tackles this issue.

1.2 Correct Programming over Serialised Data

We will see that we can refine that second definition to obtain a correct-by-construction version of `sum`, with almost exactly the same code.

```
sum : Pointer.Mu Tree t ->
      IO (Singleton (Data.sum t))
sum ptr = case !(view ptr) of
  "Leaf" # _ => pure [| Z |]
  "Node" # l # b # r =>
    do m <- sum l
       n <- sum r
       pure [| [| m + [| cast b |] |] + n |]
```

In the above snippet, we can see that the `Pointer.Mu` is indexed by a phantom parameter: a runtime irrelevant `t` which has type (`Data.Mu Tree`). And so the return type can mention the result of the pure computation (`Data.sum t`). `Singleton` is, as its name suggests, a singleton type (cf. Section 6) i.e. the natural number we compute is now proven to be equal to the one computed by the pure `sum` function. The implementation itself only differs in that we had to use idiom brackets [McBride and Paterson 2008], something we will explain in Section 6.2.

In other words, our approach also allows us to prove the functional correctness of the `IO` procedures processing trees stored in serialised format in a buffer. This is our second main contribution.

1.3 Generic Programming over Serialised Data

Last but not least, as Altenkirch and McBride demonstrated [Altenkirch and McBride 2002]: "With dependently (sic) types, generic programming is just programming: it is not necessary to write a new compiler each time a useful universe presents itself."

In this paper we carve out a universe of inductive types that can be uniformly serialised and obtain all of our results by generic programming. In practice this means that we are not limited to the type of binary trees with bytes stored in the nodes we used in the examples above. We will for instance be able to implement a generic and correct-by-construction definition of `fold` operating on data stored in a buffer whose type declaration can be seen below (we will explain how it is defined in Section 7.5).

```
fold : {cs : Data nm} -> (alg : Alg cs a) ->
       forall t. Pointer.Mu cs t ->
       IO (Singleton (Data.fold alg t))
```

This data-genericity is our third contribution.

1.4 Plan

In summary, we are going to define a library for the seamless, correct, and generic manipulation of algebraic types in serialised format.

Section 2 introduces the language of descriptions capturing the subset of inductively defined types that our work can handle. It differs slightly from usual presentations in that it ensures the types can be serialised and tracks crucial invariants towards that goal. Section 3 gives a standard meaning to these data descriptions as strictly positive endofunctors whose fixpoints give us the expected inductive types. We will use this standard meaning in the specification layer of our work. Section 4 explores the serialisation format we have picked for these trees: a depth-first, left-to-right infix traversal of the trees, with additional information stored to allow for the random access of any subtree. Section 5 defines the type of pointers to trees stored in a buffer and shows how we can use such pointers to write the corresponding tree to a file. Section 6 introduces the terminology of *views* and *singleton* types that is crucial to the art of programming in a correct-by-construction manner. Section 7 defines IO primitives that operate on serialised trees stored in an underlying buffer. They encapsulate all the unsafe low-level operations and offer a high-level interface that allows users to implement correct-by-construction procedures. Section 8 defines a set of serialisation combinators that allows users to implement correct-by-construction procedures writing values into a buffer. Section 9 discusses some preliminary performance results for the library.

2 OUR UNIVERSE OF DESCRIPTIONS

We first need to pin down the domain of our discourse. To talk generically about an entire class of datatypes without needing to modify the host language we have decided to perform a universe construction [Benke et al. 2003; Löh and Magalhães 2011; Morris 2007]. That is to say that we are

going to introduce an inductive type defining a set of codes together with an interpretation of these codes as bona fide host-language types. We will then be able to program generically over the universe of datatypes by performing induction on the type of codes [Pfeifer and Rueß 1999].

The universe we define is in the tradition of a sums-of-products vision of inductive types [Jansson and Jeuring 1997] where the data description records additional information about the static and dynamic size of the data being stored. In our setting, constructors are essentially arbitrarily nested tuples of values of type unit, bytes, and recursive substructures. A datatype is given by listing a choice of constructors.

2.1 Descriptions

We start with these constructor descriptions; they are represented internally by an inductive family `Desc` declared below.

```
data Desc : (rightmost : Bool) ->
            (static : Nat) -> (offsets : Nat) ->
            Type
```

This family has three indices corresponding to three crucial invariants being tracked. First, an index telling us whether the current description is being used in the `rightmost` branch of the overall constructor description. Second, the `static`ally known size of the described data in the number of bytes it occupies. Third, the number of `offsets` that need to be stored to compensate for subterms not having a statically known size. The reader should think of `rightmost` as an 'input' index whereas `static` and `offsets` are 'output' indices.

Next we define the family proper by giving its four constructors.

```
data Desc where
  None : Desc r 0 0
  Byte : Desc r 1 0
  Prod : {sl, sr, ol, or : Nat} ->
         Desc False sl ol -> Desc r sr or ->
         Desc r (sl + sr) (ol + or)
  Rec : Desc r 0 (ifThenElse r 0 1)
```

Each constructor can be used anywhere in a description so their return `rightmost` index can be an arbitrary boolean.

`None` is the description of values of type unit. The static size of these values is zero as no data is stored in a value of type unit. Similarly, they do not require an offset to be stored as we statically know their size.

`Byte` is the description of bytes. Their static size is precisely one byte, and they do not require an offset to be stored either.

`Prod` gives us the ability to pair two descriptions together. Its static size and the number of offsets are the respective sums of the static sizes and numbers of offsets of each subdescription. The description of the left element of the pair will never be in the rightmost branch of the overall constructors description and so its index is `False` while the description of the right element of the pair is in the rightmost branch precisely whenever the whole pair is; hence the propagation of the `r` arbitrary value from the return index into the description of the right component.

Last but not least, `Rec` is a position for a subtree. We cannot know its size in bytes statically and so we decide to store an offset unless we are in the rightmost branch of the overall description. Indeed, there are no additional constructor arguments behind the rightmost one and so we have no reason to skip past the subterm. Consequently we do not bother recording an offset for it.

2.2 Constructors

We represent a constructor as a record packing together a name for the constructor, the description of its arguments (which is, by virtue of being used at the toplevel, in rightmost position), and the values of the `static` and `offsets` invariants. The two invariants are stored as implicit fields because their value is easily reconstructed by Idris 2 using unification and so users do not need to spell them out explicitly.

```
record Constructor (nm : Type) where
  constructor (::)
  name : nm
  {static : Nat}
  {offsets : Nat}
  description : Desc True static offsets
```

Note that we used `(::)` as the name of the constructor for records of type `Constructor`. This allows us to define constructors by forming an expression reminiscent of Haskell's type declarations: `name :: type`. Returning to our running example, this gives us the following encodings for leaves that do not store anything and nodes that contain a left branch, a byte, and a right branch.

```
Leaf : Constructor String
Leaf = "Leaf" :: None

Node : Constructor String
Node = "Node" :: Prod Rec (Prod Byte Rec)
```

2.3 Datatypes

A datatype description is given by a number of constructors together with a vector (also known as a length-indexed list) associating a description to each of these constructors.

```
record Data (nm : Type) where
  constructor MkData
  {consNumber : Nat}
  constructors : Vect consNumber (Constructor nm)

Tree : Data String
Tree = MkData [Leaf, Node]
```

We can then encode our running example as a simple `Data` declaration: a binary tree whose node stores bytes is described by the choice of either a `Leaf` or `Node`, as defined above.

Now that we have a language that allows us to give a description of our inductive types, we are going to give these descriptions a meaning as trees.

3 MEANING AS TREES

We now see descriptions as functors and, correspondingly, datatypes as the initial objects of the associated functor-algebras. This is a standard construction derived from Malcolm's work [Malcolm 1990], itself building on Hagino's categorically-inspired definition of a lambda calculus with a generic notion of datatypes [Hagino 1987].

In our work these trees will be used primarily to allow users to give a precise specification of the IO procedures they actually want to write in order to process values stored in buffers. We expect these inductive trees and the associated generic programs consuming them to be mostly used at the **`0`** modality i.e. to be erased during compilation.

3.1 `Desc`s as Functors

We define the meaning of descriptions as strictly positive endofunctors on `Type` by induction on said descriptions. `Meaning` gives us the action of the functors on objects.

```
Meaning : Desc r s n -> Type -> Type
Meaning None x = ()
Meaning Byte x = Bits8
Meaning Rec x = x
Meaning (Prod d e) x
  = Tuple (Meaning d x) (Meaning e x)
```

```
record Tuple (a, b : Type) where
  constructor (#)
  fst : a
  snd : b
```

Both `None` and `Byte` are interpreted by constant functors (respectively the one returning the unit type, and the one returning the type of bytes). `Rec` is the identity functor. Finally (`Prod d e`) is interpreted as the pairing of the interpretation of `d` and `e` respectively. We use our own definition of pairing rather than the standard library's because it gives us better syntactic sugar.

This gives us the action of descriptions on types, let us now see their action on morphisms. We once again proceed by induction on the description.

```
fmap : (d : Desc r s o) -> (a -> b) -> Meaning d a -> Meaning d b
fmap None f v = v
fmap Byte f v = v
fmap (Prod d e) f (v # w) = (fmap d f v # fmap e f w)
fmap Rec f v = f v
```

All cases but the one for `Rec` are structural. Verifying that these definitions respect the functor laws is left as an exercise for the reader.

3.2 `Data` as Trees

Given a datatype description `cs`, our first goal is to define what it means to pick a constructor. The `Index` record is a thin wrapper around a finite natural number known to be smaller than the number of constructors this type provides.

```
record Index (cs : Data nm) where
  constructor MkIndex
  getIndex : Fin (consNumber cs)
```

We use this type rather than `Fin` directly because it plays well with inference. In the following code snippet, implementing a function returning the description corresponding to a given index, we use this to our advantage: the `cs` argument can be left implicit because it already shows up in the type of the `Index` and can thus be reconstructed by unification.

```
description : {cs : Data nm} -> (k : Index cs) ->
              let cons = index (getIndex k) (constructors cs) in
              Desc True (static cons) (offsets cons)
description {cs} k
  = description (index (getIndex k) (constructors cs))
```

This type of indices also allows us to provide users with syntactic sugar enabling them to use the constructors' names directly rather than confusing numeric indices. The following function runs a decision procedure `isConstructor` at the type level in order to turn any raw string `str` into the corresponding `Index`.

```
fromString : {cs : Data String} -> (str : String) ->
             {auto 0 _ : IsJust (isConstructor str cs)} ->
             Index cs
fromString {cs} str with (isConstructor str cs)
 _ | Just k = MkIndex k
```

If the name is valid then `isConstructor` will return a valid `Index` and Idris 2 will be able to **auto**matically fill-in the implicit proof. If the name is not valid then Idris 2 will not find the index and will raise a compile time error. We include a successful example on the left and a failing test on the right hand side (**failing** blocks are only accepted in Idris 2 if their body leads to an error).

```
indexLeaf : Index Tree
indexLeaf = "Leaf"
```

```
failing
  notIndexCons : Index Tree
  notIndexCons = "Cons"
```

Once equipped with the ability to pick constructors, we can define the type of algebras for the functor described by a `Data` description. For each possible constructor, we demand an algebra for the functor corresponding to the meaning of the constructor's description.

```
Alg : Data nm -> Type -> Type
Alg cs x = (k : Index cs) -> Meaning (description k) x -> x
```

We can then introduce the fixpoint of data descriptions as the initial algebra, defined as the following inductive type.

```
data Mu : Data nm -> Type where
  (#) : Alg cs (assert_total (Mu cs))
```

Note that here we are forced to use **assert_total** to convince Idris 2 to accept the definition. Indeed, unlike Agda, Idris 2 does not (yet!) track whether a function's arguments are used in a strictly positive manner. Consequently the positivity checker is unable to see that `Meaning` uses its second argument in a strictly positive manner and that this is therefore a legal definition.

Now that we can build trees as fixpoints of the meaning of descriptions, we can define convenient aliases for the `Tree` constructors. Note that the leftmost `(#)` use in each definition corresponds to the `Mu` constructor while later ones are `Tuple` constructors. Idris 2's type-directed disambiguation of constructors allows us to use this uniform notation for all of these pairing notions.

```
leaf : Mu Tree
leaf = "Leaf" # ()
```

```
node : Mu Tree -> Bits8 -> Mu Tree -> Mu Tree
node l b r = "Node" # l # b # r
```

This enables us to define our running example as an inductive value:

```
example : Mu Tree
example = node (node (node leaf 1 leaf) 5 leaf) 10 (node leaf 20 leaf)
```

3.3 Generic Fold

`Mu` gives us the initial fixpoint for these algebras i.e. given any other algebra over a type `a`, from a term of type `(Mu cs)`, we can compute an `a`. We define the generic `fold` function over inductive values as follows:

```
fold : {cs : Data nm} -> Alg cs a -> Mu cs -> a
fold alg (k # t) = alg k (assert_total $ fmap _ (fold alg) t)
```

We first match on the term's top constructor, use `fmap` (defined in Section 3.1) to recursively apply the fold to all the node's subterms and finally apply the algebra to the result.

Here we only use **assert_total** because Idris 2 does not see that `fmap` only applies its argument to strict subterms. This limitation could easily be bypassed by mutually defining an inlined and specialised version of `(fmap _ (fold alg))`. as we demonstrate in Appendix A. In an ideal type theory these supercompilation steps, whose sole purpose is to satisfy the totality checker, would be automatically performed by the compiler [Mendel-Gleason 2012].

Further generic programming can yield other useful programs e.g. a generic proof that tree equality is decidable or a generic definition of zippers [Löh and Magalhães 2011].

4 SERIALISED REPRESENTATION

Before we can give a meaning to descriptions as pointers into a buffer we need to decide on a serialisation format. The format we have opted for is split in two parts: a header containing data that can be used to check that a user's claim that a given file contains a serialised tree of a given type is correct, followed by the actual representation of the tree.

For instance, the following binary snippet is a hex dump of a file containing the serialised representation of a binary tree belonging to the type we have been using as our running example. The raw data is semantically highlighted: 8-bytes-long `offsets`, a `type` description of the stored data, some `nodes` of the tree and the `data` stored in the nodes.

```
87654321  00 11 22 33 44 55 66 77 88 99 AA BB CC DD EE FF
00000000: 07 00 00 00 00 00 00 00 02 00 02 03 02 01 03 01
00000010: 17 00 00 00 00 00 00 00 01 0c 00 00 00 00 00 00
00000020: 00 01 01 00 00 00 00 00 00 00 00 01 00 05 00 0a
00000030: 01 01 00 00 00 00 00 00 00 00 14 00
```

More specifically, this block is the encoding of the `example` given in the previous section and, knowing that a `leaf` is represented here by `00` and a `node` is represented by `01` the careful reader can check (modulo ignoring the type description and offsets for now) that the data is stored in a depth-first, left-to-right traversal of the tree (i.e. we get exactly the bit pattern we saw in the naïve encoding presented in Section 1).

4.1 Header

In our example, the header is as follows:

```
07 00 00 00 00 00 00 00 02 00 02 03 02 01 03
```

The header consists of an offset allowing us to jump past it in case we do not care to inspect it, followed by a binary representation of the `Data` description of the value stored in the buffer. This can be useful in a big project where different components produce and consume such serialised values: if we change the format in one place but forget to update it in another, we want the program to gracefully fail to load the file using an unexpected format. We detail in Section 10.2.1 how dependent type providers can help structure a software project to prevent such issues.

The encoding of a data description starts with a byte giving us the number of constructors, followed by these constructors' respective descriptions serialised one after the other. `None` is represented by `00`, `Byte` is represented by `01`, (`Prod d e`) is represented by `02` followed by the representation of `d` and then that of `e`, and `Rec` is represented by `03`.

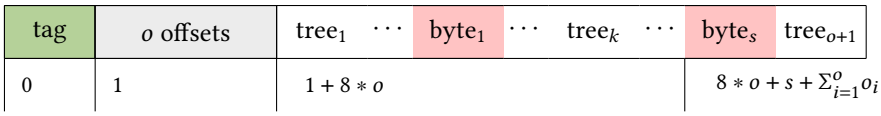
Looking once more at the header in the running example, the `Data` description is indeed 7 bytes long like the offset states. The `Data` description starts with `02` meaning that the type has two constructors. The first one is `00` i.e. `None` (this is the encoding of the type of `Leaf`), and the second one is `02 03 02 01 03` i.e. `(Prod Rec (Prod Byte Rec))` (that is to say the encoding of the type of `Node`). According to the header, this file does contain a `Tree`.

4.2 Tree Serialisation

Our main focus in the definition of this format is that we should be able to process any of a node's subtrees without having to first traverse the subtrees that come before it. This will allow us to, for instance, implement a function looking up the value stored in the rightmost node in our running

example type of binary trees in time linear in the depth of the tree rather than exponential. To this end each node needs to store an offset measuring the size of the subtrees that are to the left of any relevant information.

If a given tag is associated to a description of type (`Desc True s o`) then the representation in memory of the associated node will look something like the following.

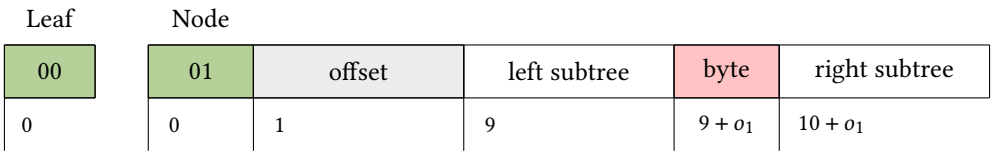


On the first line we have a description of the data layout and on the second line we have the offset of various positions in the block with respect to the tag's address.

For the data layout, we start with the tag then we have o offsets, and finally we have a block contiguously storing an interleaving of subtrees and s bytes dictated by the description. In this example the rightmost value in the description is a subtree and so even though we have o offsets, we actually have $(o + 1)$ subtrees stored.

The offsets of the tag with respect to its own address is 0. The tag occupies one byte and so the offset of the block of offsets is 1. Each offset occupies 8 bytes and so the constructor's arguments are stored at offset $(1 + 8 * o)$. Finally each value's offset can be computed by adding up the offset of the start of the block of constructor arguments, the offsets corresponding to all of the subtrees that come before it, and the number of bytes stored before it; in the case of the last byte that gives $1 + 8 * o + \Sigma_{i=1}^{o} o_i + s - 1$ hence the formula included in the diagram.

Going back to our running example, this translates to the following respective data layouts and offsets for a leaf and a node.



Now that we understand the format we want, we ought to be able to implement pointers and the functions manipulating them.

5 MEANING AS POINTERS INTO A BUFFER

Now that we know the serialisation format, we can give a meaning to constructor and data descriptions as pointers into a buffer. For reasons that will become apparent in Section 7.5 when we start programming over serialised data in a correct-by-construction manner, our types of 'pointers' will be parameterised not only by the description of the type of the data stored but also by a runtime-irrelevant inductive value of that type. For now, it is enough to think of these indices as a lightweight version of the 'points to' assertions used in separation logic [Reynolds 2002] when reasoning about imperative programs. We expand on this analogy in Appendix C where we also discuss the connection with the combinators defined in Section 7.

5.1 Tracking Buffer Positions

We start with the definition of the counterpart to `Mu` for serialised values.

```
record Mu (cs : Data nm) (t : Data.Mu cs) where
  constructor MkMu
  muBuffer : Buffer
  muPosition : Int
  muSize : Int
```

A tree sitting in a buffer is represented by a record packing the buffer, the position at which the tree's root node is stored, and the size of the tree. Note that according to our serialisation format the size is not stored in the file but using the size of the buffer, the stored offsets, and the size of the static data we will always be able to compute a value corresponding to it.

```
record Meaning (d : Desc r s o) (cs : Data nm)
               (t : Data.Meaning d (Data.Mu cs)) where
  constructor MkMeaning
  subterms : Vect o Int
  meaningBuffer : Buffer
  meaningPosition : Int
  meaningSize : Int
```

The counterpart to a `Meaning` stores additional information. For a description of type (`Desc r s o`) on top of the buffer, the position at which the root of the meaning resides, and the size of the layer we additionally have a vector of `o` offsets that allow us to efficiently access any value we want.

5.2 Writing a Tree to a File

Once we have a pointer to a tree in a buffer, we can easily write it to a file be it for safekeeping or sending over the network.

```
writeToFile : {cs : Data nm} -> FilePath ->
              forall t. Pointer.Mu cs t -> IO ()
writeToFile fp (MkMu buf pos size) = do
  desc <- getInt buf 0
  let start = 8 + desc
  let bufSize = 8 + desc + size
  buf <- if pos == start then pure buf else do
    Just newbuf <- newBuffer bufSize
      | Nothing => failWith "\{__LOC__} Couldn't allocate buffer"
    copyData buf 0 start newbuf 0
    copyData buf start size newbuf start
    pure buf
  Right () <- writeBufferToFile fp buf bufSize
    | Left (err, _) => failWith (show err)
  pure ()
```

We first start by reading the size of the header stored in the buffer. This allows us to compute both the `start` of the data block as well as the size of the buffer (`bufSize`) that will contain the header followed by the tree we want to write to a file. We then check whether the position of the pointer is exactly the beginning of the data block. If it is then we are pointing to the whole tree and the current buffer can be written to a file as is. Otherwise we are pointing to a subtree and need to separate it from its surrounding context first. To do so we allocate a new buffer of the right size and use the standard library's `copyData` primitive to copy the raw bytes corresponding to the header first, and the tree of interest second. We can then write the buffer we have picked to a file and happily succeed.

Now that we have pointers and can save the tree they are standing for, we are only missing the ability to look at the content they are pointing to. But first we need to introduce some basic tools to be able to talk precisely about this stored content.

6 INTERLUDE: VIEWS AND SINGLETONS

The precise indexing of pointers by a runtime-irrelevant copy of the value they are pointing to means that inspecting the buffer's content should not only return runtime information but also refine the index to reflect that information at the type-level. As a consequence, the functions we are going to define in the following subsections are views.

6.1 Views

A view in the sense of Wadler [Wadler 1987], and subsequently refined by McBride and McKinna [McBride and McKinna 2004] for a type T is a type family V indexed by T together with a function which maps values t of type T to values of type $V\ t$. By inspecting the $V\ t$ values we can learn something about the t input. The prototypical example is perhaps the 'snoc' ('cons' backwards) view of right-nested lists as if they were left-nested. We present the `Snoc` family below.

```
data Snoc : List a -> Type where
  Lin : Snoc []
  (:<) : (init : List a) -> (last : a) -> Snoc (init ++ [last])
```

By matching on a value of type (`Snoc xs`) we get to learn either that `xs` is empty (`Lin`, nil backwards) or that it has an initial segment `init` and a last element `last` (`init :< last`). The function `unsnoc` demonstrates that we can always *view* a `List` in a `Snoc`-manner.

```
unsnoc : (xs : List a) -> Snoc xs
unsnoc [] = Lin
unsnoc (x :: xs@_) with (unsnoc xs)
  _ | [<] = [] :< x
  _ | init :< last = (x :: init) :< last
```

Here we defined `Snoc` as an inductive family but it can sometimes be convenient to define the family recursively instead. In which case the `Singleton` inductive family can help us connect runtime values to their runtime-irrelevant type-level counterparts.

6.2 The `Singleton` type

The `Singleton` family has a single constructor which takes an argument `x` of type `a`, its return type is indexed precisely by this `x`.

```
data Singleton : {0 a : Type} -> (x : a) -> Type where
  MkSingleton : (x : a) -> Singleton x
```

More concretely this means that a value of type (`Singleton` t) has to be a runtime relevant copy of the term t. Note that Idris 2 performs an optimisation similar to Haskell's `newtype` unwrapping: every data type that has a single non-recursive constructor with only one non-erased argument is unwrapped during compilation. This means that at runtime the `Singleton` / `MkSingleton` indirections will have disappeared.

We can define some convenient combinators to manipulate singletons. We reuse the naming conventions typical of applicative functors which will allow us to rely on Idris 2's automatic desugaring of *idiom brackets* [McBride and Paterson 2008] into expressions using these combinators.

```
pure : (x : a) -> Singleton x
pure = MkSingleton
```

First `pure` is a simple alias for `MkSingleton`, it turns a runtime-relevant value `x` into a singleton for this value.

```
(<$>) : (f : a -> b) -> Singleton t -> Singleton (f t)
f <$> MkSingleton t = MkSingleton (f t)
```

Next, we can 'map' a function under a `Singleton` layer: given a pure function `f` and a runtime copy of `t` we can get a runtime copy of `(f t)`.

```
(<*>) : Singleton f -> Singleton t -> Singleton (f t)
MkSingleton f <*> MkSingleton t = MkSingleton (f t)
```

Finally, we can apply a runtime copy of a function `f` to a runtime copy of an argument `t` to get a runtime copy of the result `(f t)`.

As we mentioned earlier, Idris 2 automatically desugars idiom brackets using these combinators. That is to say that `[| x |]` will be elaborated to `(pure x)` while `[| f t1 ··· tn |]` will become `(f <$> t1 <*> ··· <*> tn)`. This lets us apply `Singleton`-wrapped values almost as seamlessly as pure values.

We are now equipped with the appropriate notions and definitions to look at a buffer's content.

7 INSPECTING A BUFFER'S CONTENT

We can now describe the combinators allowing our users to inspect the value they have a pointer for. We are going to define the most basic of building blocks (`poke` and `out`), combine them to derive useful higher-level combinators (`layer` and `view`), and ultimately use these to implement a generic correct-by-construction version of `fold` operating over trees stored in a buffer (cf. Section 7.5).

Readers may be uneasy about the unsafe implementations of the basic building blocks but we argue that it is a necessary evil by drawing an extended analogy to separation logic in Appendix C.

7.1 Poking the Buffer

Our most basic operation consists in poking the buffer to unfold the description by exactly one step. The type of the function is as follows: provided a pointer for a meaning `t`, we return an `IO` process computing the one step unfolding of the meaning.

```
poke : {0 cs : Data nm} -> {d : Desc r s o} ->
       forall t. Pointer.Meaning d cs t ->
       IO (Poke d cs t)
```

The result type of this operation is defined by case-analysis on the description. In order to keep the notations user-friendly, we mutually define a recursive function `Poke` interpreting the straightforward type constructors and an inductive family `Poke'` with interesting return indices.

```
Poke : (d : Desc r s o) -> (cs : Data nm) ->
       Data.Meaning d (Data.Mu cs) -> Type
Poke None _ t = ()
Poke Byte cs t = Singleton t
Poke Rec cs t = Pointer.Mu cs t
Poke d@(Prod _ _) cs t = Poke' d cs t
```

Poking a buffer containing `None` will return a value of the unit type as no information whatsoever is stored there.

If we access a `Byte` then we expect that inspecting the buffer will yield a runtime-relevant copy of the type-level byte we have for reference. Hence the use of `Singleton`.

If the description is `Rec` this means we have a substructure. In this case we simply demand a pointer to it.

Last but not least, if we access a `Prod` of two descriptions then the type-level term better be a pair and we better be able to obtain a `Pointer.Meaning` for each of the sub-meanings. Because Idris 2 does not currently support definitional eta equality for records, it will be more ergonomic for users if we introduce `Poke'` rather than yielding a `Tuple` of values. By matching on `Poke'` at the value level, they will see the pair at the type level also reduced to a constructor-headed tuple.

```
data Poke' : (d : Desc r s o) -> (cs : Data nm) ->
             Data.Meaning d (Data.Mu cs) -> Type where
  (#) : Pointer.Meaning d cs t ->
        Pointer.Meaning e cs u ->
        Poke' (Prod d e) cs (t # u)
```

The implementation of this operation proceeds by case analysis on the description. As we are going to see shortly, it is necessarily somewhat unsafe as we claim to be able to connect a type-level value to whatever it is that we read from the buffer. Let us go through each case one-by-one.

```
poke {d = None} el = pure ()
```

If the description is `None` we do not need to fetch any information from the buffer and can immediately return `()`.

```
poke {d = Byte} el = do
  bs <- getBits8 (meaningBuffer el) (meaningPosition el)
  pure (unsafeMkSingleton bs)
```

If the description is `Byte` then we read a byte at the determined position. The only way we can connect this value we just read to the type index is to use the unsafe combinator **`unsafeMkSingleton`** to manufacture a value of type `(Singleton t)` instead of the value of type `(Singleton bs)` we would expect from wrapping `bs` in the `MkSingleton` constructor. As we explain in Appendix C.2.1, in separation logic this would correspond to declaring an axiom about the `poke` language construct.

```
poke {d = Prod {sl, ol} d e} {t} (MkMeaning sub buf pos size) = do
  let (subl, subr) = splitAt ol sub
  let sizel = sum subl + cast sl
  let left = MkMeaning subl buf pos sizel
  let posr = pos + sizel
  let right = MkMeaning subr buf posr (size - sizel)
  pure (rewrite etaTuple t in left # right)
```

If the description is the product of two sub-descriptions then we want to compute the `Pointer.Meaning` corresponding to each of them. We start by splitting the vector of offsets to distribute them between the left and right subtrees.

We can readily build the pointer for the `left` subdescription: it takes the left offsets, the buffer, and has the same starting position as the whole description of the product as the submeanings are stored one after the other. Its size (`sizel`) is the sum of the space reserved by all of the left offsets (`sum subl`) as well as the static size occupied by the rest of the content (`sl`).

We then compute the starting position of the right subdescription: we need to move past the whole of the left subdescription, that is to say that the starting position is the sum of the starting position for the whole product and `sizel`. The size of the right subdescription is then easily computed by subtracting `sizel` from the overall `size` of the paired subdescriptions.

We can finally use the lemma `etaTuple` saying that a tuple is equal to the pairing of its respective projections in order to turn `t` into (`fst t # snd t`) which lets us use the `Poke'` constructor `(#)` to return our pair of pointers.

```
poke {d = Rec} (MkMeaning _ buf pos size) = pure (MkMu buf pos size)
```

Lastly, when we reach a `Rec` description, we can discard the vector of offsets and return a `Pointer.Mu` with the same buffer, starting position and size as our input pointer.

7.2 Extracting one layer

By repeatedly poking the buffer, we can unfold a full layer. The result of this operation is defined by induction on the description. It is identical to the definition of `Poke` except for the `Prod` case: here, instead of being content with a pointer for each of the subdescriptions, we demand a full layer for them too.

```
Layer : (d : Desc r s o) -> (cs : Data nm) ->
        Data.Meaning d (Data.Mu cs) -> Type
Layer None _ _ = ()
Layer Byte _ t = Singleton t
Layer Rec cs t = Pointer.Mu cs t
Layer d@(Prod _ _) cs t = Layer' d cs t

data Layer' : (d : Desc r s o) -> (cs : Data nm) ->
              Data.Meaning d (Data.Mu cs) -> Type where
  (#) : Layer d cs t -> Layer e cs u -> Layer' (Prod d e) cs (t # u)
```

This function can easily be implemented by induction on the description and repeatedly calling `poke` to expose the values one by one.

```
layer : {0 cs : Data nm} -> {d : Desc r s o} ->
        forall t. Pointer.Meaning d cs t -> IO (Layer d cs t)
layer el = poke el >>= go d where

  go : forall r, s, o. (d : Desc r s o) ->
       forall t. Poke d cs t -> IO (Layer d cs t)
  go None p = pure ()
  go Byte p = pure p
  go (Prod d e) (p # q) = [| layer p # layer q |]
  go Rec p = pure p
```

7.3 Exposing the top constructor

Now that we can deserialise an entire layer of `Meaning`, the only thing we are missing to be able to generically manipulate trees is the ability to expose the top constructor of a tree stored at a `Pointer.Mu` position. Remembering the data layout detailed in Section 4.2, this will amount to inspecting the tag used by the node and then deserialising the offsets stored immediately after it.

The `Out` family describes the typed point of view: to get your hands on the index of a tree's constructor means obtaining an `Index`, and a `Pointer.Meaning` to the constructor's arguments (remember that these high-level 'pointers' store a vector of offsets). The family's index (`k # t`) ensures that the structure of the runtime irrelevant tree is adequately described by the index (`k`) and the `Data.Meaning` (`t`) the `Pointer.Meaning` is for.

```
data Out : (cs : Data nm) -> (t : Data.Mu cs) -> Type where
  (#) : (k : Index cs) ->
        forall t. Pointer.Meaning (description k) cs t ->
        Out cs (k # t)
```

The type of the `out` function is as expected: given a pointer to a tree `t` of type `cs` we can get a value of type (`Out cs t`). That is to say, we can get a view allowing us to reveal what the index of the tree's head constructor is.

```
out : {cs : Data nm} -> forall t. Pointer.Mu cs t ->
      IO (Out cs t)
```

The implementation is fairly straightforward except for another unsafe step meant to reconcile the information we read in the buffer with the runtime-irrelevant tree index.

```
out {t} mu = do
  tag <- getBits8 (muBuffer mu) (muPosition mu)
  let Just k = MkIndex <$> natToFin (cast tag) (consNumber cs)
    | _ => failWith "Invalid representation"
  let 0 sub = unfoldAs k t
  val <- (k #) <$> getConstructor k {t = sub.fst}
                     (rewrite sym sub.snd in mu)
  pure (rewrite sub.snd in val)
```

We start by reading the tag `k` corresponding to the constructor choice: we obtain a byte by calling `getBits8`, cast it to a natural number and then make sure that it is in the range $[0 \cdots$ `consNumber cs`$[$ using `natToFin`. We then use the unsafe **`unfoldAs`** postulate to step the type-level `t` to something of the form (`k # val`).

```
%unsafe
0 unfoldAs :
  (k : Index cs) -> (0 t : Data.Mu cs) ->
  (val : Data.Meaning (description k) (Data.Mu cs)
  ** t === (k # val))
```

The declaration of **`unfoldAs`** is marked as runtime irrelevant because it cannot possibly be implemented (`t` is runtime irrelevant and so cannot be inspected) and so its output should not be relied upon in runtime-relevant computations. Its type states that there exists a `Meaning` called `val` such that `t` is equal to (`k # val`)

Now that we know the head constructor we want to deserialise and that we have the ability to step the runtime irrelevant tree to match the actual content of the buffer, we can use `getConstructor` to build such a value.

```
getConstructor : (k : Index cs) ->
  forall t. Pointer.Mu cs (k # t) ->
  IO (Pointer.Meaning (description k) cs t)
getConstructor (MkIndex k) mu
  = let offs : Nat; offs = offsets (index k $ constructors cs) in
    getOffsets (muBuffer mu) (1 + muPosition mu) offs
  $ let size = muSize mu - 1 - cast (8 * offs) in
    \ subterms, pos => MkMeaning subterms (muBuffer mu) pos size
```

To get a constructor, we start by getting the vector of offsets stored immediately after the tag. We then compute the size of the remaining `Meaning` description: it is the size of the overall tree, minus 1 (for the tag) and 8 times the number of offsets (because each offset is stored as an 8 bytes

number). We can then use the record constructor `MkMeaning` to pack together the vector of offsets, the buffer, the position past the offsets and the size we just computed.

```
getOffsets : Buffer -> (pos : Int) ->
  (n : Nat) ->
  forall t. (Vect n Int -> Int -> Pointer.Meaning d cs t) ->
  IO (Pointer.Meaning d cs t)
getOffsets buf pos 0 k = pure (k [] pos)
getOffsets buf pos (S n) k = do
  off <- getInt buf pos
  getOffsets buf (8 + pos) n (k . (off ::))
```

The implementation of `getOffsets` is straightforward: given a continuation that expect `n` offsets as well as the position past the last of these offsets, we read the 8-bytes-long offsets one by one and pass them to the continuation, making sure that we move the current position accordingly before every recursive call.

7.4 Offering a convenient `View`

We can combine `out` and `layer` to obtain the `view` function we used in our introductory examples in Section 1.1. A (`View cs t`) value gives us access to the (`Index cs`) of `t`'s top constructor together with the corresponding `Layer` of deserialised values and pointers to subtrees.

```
data View : (cs : Data nm) -> (t : Data.Mu cs) -> Type where
  (#) : (k : Index cs) ->
        forall t. Layer (description k) cs t ->
        View cs (k # t)
```

The implementation of `view` is unsurprising: we use `out` to expose the top constructor index and a `Pointer.Meaning` to the constructor's payload. We then user `layer` to extract the full `Layer` of deserialised values that the pointer references.

```
view : {cs : Data nm} ->
       forall t. Pointer.Mu cs t ->
       IO (View cs t)

view ptr = do k # el <- out ptr
              vs <- layer el
              pure (k # vs)
```

It is worth noting that although a `view` may be convenient to consume, a performance-minded user may decide to directly use the `out` and `poke` combinators to avoid deserialising values that they do not need. We present a case study in Appendix B comparing the access patterns of two implementations of the function fetching the byte stored in a tree's rightmost node depending on whether we use `view` or the lower level `poke` combinator.

By repeatedly calling `view`, we can define the correct-by-construction generic deserialisation function that turns a pointer to a tree into a runtime value equal to this tree.

```
deserialise : {cs : Data nm} -> forall t.
              Pointer.Mu cs t -> IO (Singleton t)
```

We can measure the benefits of our approach by comparing the runtime of a function directly operating on buffers to its pure counterpart composed with a deserialisation step. For functions like `rightmost` that only explore a very small part of the full tree, the gains are spectacular: the

process operating on buffers is exponentially faster than its counterpart which needs to deserialise the entire tree first (cf. Section 9).

7.5 Generic Fold

The implementation of the generic `fold` over a tree stored in a buffer is going to have the same structure as the generic fold over inductive values: first match on the top constructor, then use `fmap` to apply the fold to all the substructures and, finally, apply the algebra to the result. We start by implementing the buffer-based counterpart to `fmap`. Let us go through the details of its type first.

```
fmap : (d : Desc r s o) ->
       (0 f : Data.Mu cs -> b) ->
       (forall t. Pointer.Mu cs t -> IO (Singleton (f t))) ->
       forall t. Pointer.Meaning d cs t ->
       IO (Singleton (Data.fmap d f t))
```

The first two arguments to `fmap` are similar to its pure counterpart: a description `d` and a (here runtime-irrelevant) function `f` to map over a `Meaning`. Next we take a function which is the buffer-aware counterpart to `f`: given any runtime-irrelevant term `t` and a pointer to it in a buffer, it returns an `IO` process computing the value (`f t`). Finally, we take a runtime-irrelevant meaning `t` as well as a pointer to its representation in a buffer and compute an `IO` process which will return a value equal to (`Data.fmap d f t`).

We can now look at the definition of `fmap`.

```
fmap d f act ptr = poke ptr >>= go d where

  go : (d : Desc{}) -> forall t. Poke d cs t ->
       IO (Singleton (Data.fmap d f t))
  go None {t} v = pure byIrrelevance
  go Byte v = pure v
  go (Prod d e) (v # w)
    = do fv <- fmap d f act v
         fw <- fmap e f act w
         pure [| fv # fw |]
  go Rec v = act v
```

We poke the buffer to reveal the value the `Pointer.Meaning` named `ptr` is pointing at and then dispatch over the description `d` using the `go` auxiliary function.

If the description is `None` we use `byIrrelevance` which happily builds any (`Singleton t`) provided that `t`'s type is proof irrelevant.

If the description is `Byte`, the value is left untouched and so we can simply return it immediately.

If we have a `Prod` of two descriptions, we recursively apply `fmap` to each of them and pair the results back.

Finally, if we have a `Rec` we apply the function operating on buffers that we know performs the same computation as `f`.

We can now combine `out` and `fmap` to compute the correct-by-construction `fold`: provided an algebra for a datatype `cs` and a pointer to a tree of type `cs` stored in a buffer, we return an `IO` process computing the fold.

```
fold : {cs : Data nm} -> (alg : Alg cs a) ->
       forall t. Pointer.Mu cs t ->
       IO (Singleton (Data.fold alg t))
```

We first use `out` to reveal the constructor choice in the tree's top node, we then recursively apply (`fold alg`) to all the substructures by calling `fmap`, and we conclude by applying the algebra to this result.

```
fold alg ptr
  = do k # t <- out ptr
       rec <- assert_total (fmap _ _ (fold alg) t)
       pure (alg k <$> rec)
```

We once again (cf. Section 3.3) had to use `assert_total` because it is not obvious to Idris 2 that `fmap` only uses its argument on subterms. This could have also been avoided by mutually defining `fold` and a specialised version of (`fmap (fold alg)`) at the cost of code duplication and obfuscation. We once again include such a definition in Appendix A.

8 SERIALISING DATA

So far all of our example programs involved taking an inductive value apart and computing a return value in the host language. But we may instead want to compute another value in serialised form. We include below one such example: a `map` function which takes a function `f` acting on bytes and applies it to all of the ones stored in the nodes of our type of `Tree`s.

```
map : (f : Bits8 -> Bits8) ->
      (ptr : Pointer.Mu Tree t) ->
      Serialising Tree (Data.map f t)
map f ptr = case !(view ptr) of
  "Leaf" # () => "Leaf" # ()
  "Node" # l # b # r => "Node" # map f l # [| f b |] # map f r
```

It calls the `view` we just defined to observe whether the tree is a leaf or a node. If it's a leaf, it returns a leaf. If it's a node, it returns a node where the map has been recursively applied to the left and right subtrees while the function `f` has been applied to the byte `b`.

In this section we are going to spell out how we can define high-level constructs allowing users to write these correct-by-construction serialisers.

8.1 The Type of Serialisation Processes

A serialisation process for a tree `t` that belongs to the datatype `cs` is a function that takes a buffer and a starting position and returns an `IO` process that serialises the term in the buffer at that position and computes the position of the first byte past the serialised tree.

```
record Serialising (cs : Data nm) (t : Data.Mu cs) where
  constructor MkSerialising
  runSerialising : Buffer -> Int -> IO Int
```

We do not expect users to define such processes by hand and in fact prevent them from doing so by not exporting the `MkSerialising` constructor. Instead, we provide high-level, invariant-respecting combinators to safely construct such serialisation processes.

8.2 Building Serialisation Processes

Our main combinator is `(#)`: by providing a node's constructor index and a way to serialise all of the node's subtrees, we obtain a serialisation process for said node. We will give a detailed explanation of `All` below.

```
(#) : {cs : Data nm} -> (k : Index cs) ->
      {0 t : Meaning (description k) (Data.Mu cs)} ->
      All (description k) (Serialising cs) t ->
      Serialising cs (k # t)
```

The keen reader may refer to the accompanying code to see the implementation. Informally (cf. Section 4.2 for the description of the format): first we write the tag corresponding to the choice of constructor, then we leave some space for the offsets, in the meantime we write all of the constructor's arguments and collect the offsets associated to each subtree while doing so, and finally we fill in the space we had left blank with the offsets we have thus collected.

The `All` quantifier performs the pointwise lifting of a predicate over the functor described by a `Desc`. It is defined by induction over the description.

```
All : (d : Desc r s o) -> (p : x -> Type) -> Meaning d x -> Type
All None p t = ()
All Byte p t = Singleton t
All Rec p t = p t
All d@(Prod _ _) p t = All' d p t
```

If the description is `None` then there is nothing to apply the predicate to and so we return the unit type. If the description is `Byte` we only demand that we have a runtime copy of the byte so that we may write it inside a buffer. This is done using the `Singleton` family discussed in Section 6.2. If the description is `Rec` then we demand that the predicate holds. Finally, if the description is a the `Prod` of two subdescriptions, we once again use an auxiliary family purely for ergonomics. It is defined mutually with `All` and does the expected structural operation.

```
data All' : (d : Desc r s o) -> (p : x -> Type) ->
            Meaning d x -> Type where
  (#) : All d p t -> All e p u -> All' (Prod d e) p (t # u)
```

It should now be clear that (`All (description k) (Serialising cs)`) indeed corresponds to having already defined a serialisation process for each subtree.

This very general combinator should be enough to define all the serialisers we may ever want. By repeatedly pattern-matching on the input tree and using `(#)`, we can for instance define the correct-by-construction generic serialisation function.

```
serialise : {cs : Data nm} -> (t : Data.Mu cs) -> Serialising cs t
```

We nonetheless include other combinators purely for performance reasons.

8.3 Copying Entire Trees

We introduce a `copy` combinator for trees that we want to serialise as-is and have a pointer for. Equipped with this combinator, we are able to easily write e.g. the `swap` function which takes a binary tree apart and swaps its left and right branches (if the tree is non-empty).

```
swap : Pointer.Mu Tree t -> Serialising Tree (Data.swap t)
swap ptr = case !(view ptr) of
  "Leaf" # () => leaf
  "Node" # l # b # r => node (copy r) b (copy l)
```

We could define this `copy` combinator at a high level either by composing `deserialise` and `serialise`, or by interleaving calls to `view` and `(#)`. This would however lead to a slow implementation that needs to traverse the entire tree in order to simply copy it.

Instead, we implement `copy` by using the `copyData` primitive for `Buffer`s present in Idris 2's standard library. This primitive allows us to grab a slice of the source buffer corresponding to the tree and to copy the raw bytes directly into the target buffer.

```
copy : Pointer.Mu cs t -> Serialising cs t
copy ptr = MkSerialising $ \ buf, pos => do
  let size = muSize ptr
  copyData (muBuffer ptr) (muPosition ptr) size buf pos
  pure (pos + size)
```

This is the one combinator that crucially relies on our format only using offsets and not absolute addresses and on the accuracy of the size information we have been keeping in `Pointer.Mu` and `Pointer.Meaning`. As we can see in Section 9, this is spectacularly faster than a deep copying process traversing the tree.

8.4 Executing a Serialisation Action

Now that we can describe actions serialising a value to a buffer, the last basic building block we are still missing is a function actually performing such actions. This is provided by the `execSerialising` function declared below.

```
execSerialising : {cs : Data nm} -> {0 t : Data.Mu cs} ->
                  Serialising cs t -> IO (Pointer.Mu cs t)
```

By executing a (`Serialising cs t`), we obtain an `IO` process returning a pointer to the tree `t` stored in a buffer. We can then either compute further with this tree (e.g. by calling `sum` on it), or write it to a file for safekeeping using the function `writeToFile` introduced in Section 5.2.

8.5 Evaluation Order

The careful reader may have noticed that we can and do run arbitrary `IO` operations when building a value of type `Serialising` (cf. the `map` example in Section 8 where we perform a call to `view` to inspect the input's shape).

This is possible thanks to Idris 2 elaborating **do**-blocks using whichever appropriate bind operator is in scope. In particular, we have defined the following one to use when building a serialisation process:

```
(>>=) : IO a -> (a -> Serialising cs t) -> Serialising cs t
io >>= f = MkSerialising $ \buf, start =>
             do x <- io
                runSerialising (f x) buf start
```

By using this bind we can temporarily pause writing to the buffer to make arbitrary `IO` requests to the outside world. In particular, this allows us to interleave reading from the original buffer and writing into the target one thus having a much better memory footprint than if we were to first use the `IO` monad to build in one go the whole serialisation process for a given tree and then execute it.

9 BENCHMARKS

Now that we have the ability to read, write, and program directly over trees stored in a buffer we can run some experiments to see whether this allows us to gain anything over the purely functional programming style.

For all of these tests we generate a full tree of a given depth and compare the time it takes to run the composition of deserialising the tree and applying the pure function to the time it takes to run

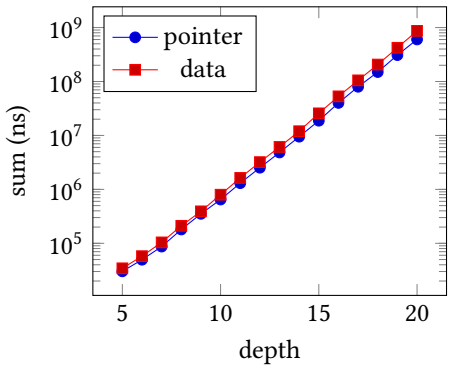
its pointer-based counterpart. Each test is run 20 times in a row, and the duration averaged. We manually run chezscheme's garbage collector before the start of each time measurement.

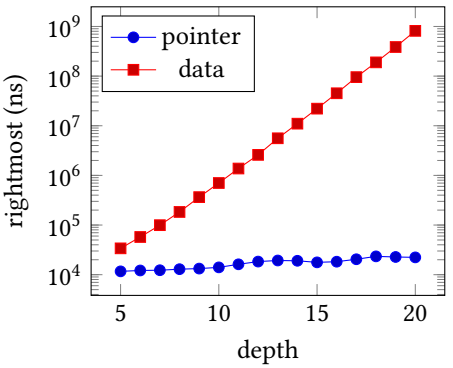
All of our plots use a logarithmic y axis because the runtime of the deserialisation-based function is necessarily exponential in the depth of the full tree.

The `sum` function explores the entirety of the tree and as such the difference between the deserialisation-based and the pointer-based functions is minimal.

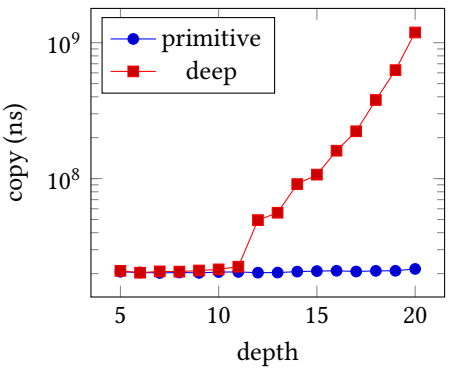
Measure memory footprint?

The `rightmost` function only explores the rightmost branch of the tree and we correspondingly see an exponential speedup for the pointer-based function which is able to efficiently skip past every left subtree.





The deep `copy` is unsurprisingly also exponential in the depth of the tree being copied whereas the version based on the `copyData` primitive for buffers is vastly faster.



10 CONCLUSION

We have seen how, using a universe of descriptions indexed by their static and dynamic sizes, we can define a precise language of values serialised in a buffer. This allowed us to develop a library to manipulate such trees in a seamless, correct, and generic manner either using low-level combinators like `poke` or high-level programs like a data-polymorphic `fold`. We then provided users with convenient tools to write serialisation processes thus allowing them to compositionally build correct-by-construction values stored in buffers.

10.1 Related Work

This work sits at the intersection of many domains: data-generic programming, the efficient runtime representation of functional data, programming over serialised values, and the design of serialisation formats. Correspondingly, a lot of related work is worth discussing. In many cases the advantage of our approach is precisely that it is at the intersection of all of these strands of research.

10.1.1 Data-Generic Programming. There is a long tradition of data-generic programming [Gibbons 2006] and we will mostly focus here on the approach based on the careful reification of a precise universe of discourse as an inductive family in a host type theory, and the definition of generic programs by induction over this family.

One early such instance is Pfeifer and Rueß' 'polytypic proof construction' [Pfeifer and Rueß 1999] meant to replace unsafe meta-programs deriving recursors (be they built-in support, or user-written tactics).

In his PhD thesis, Morris [Morris 2007] declares various universes for strictly positive types and families and defines by generic programming further types (the type of one-hole contexts), modalities (the universal and existential predicate lifting over the functors he considers), and functions (map, boolean equality).

Löh and Magalhães [Löh and Magalhães 2011] define a more expressive universe over indexed functors that is closed under composition and fixpoints. They also detail how to define additional generic construction such as a proof of decidable equality, various recursion schemes, and zippers. This work, quite similar to our own in its presentation, offers a natural candidate universe for us to use to extend our library.

10.1.2 Efficient Runtime Representation of Inductive Values. Although not dealing explicitly with programming over serialised data, Monnier's work [Monnier 2019] with its focus on performance and in particular on the layout of inductive values at runtime, partially motivated our endeavour. Provided that we find a way to get the specialisation and partial evaluation of the generically defined views, we ought to be able to achieve –purely in user code– Monnier's vision of a representation where n-ary tuples have constant-time access to each of their component.

10.1.3 Working on Serialised Data. LoCal [Vollmer et al. 2019] is the work that originally motivated the design of this library. We have demonstrated that generic programming within a dependently typed language can yield the sort of benefits other language can only achieve by inventing entirely new intermediate languages and compilation schemes.

LoCal was improved upon with a re-thinking of the serialisation scheme making the approach compatible with parallel programming [Koparkar et al. 2021]. This impressive improvement is a natural candidate for future work on our part: the authors demonstrate it is possible to reap the benefits of both programming over serialised data and dividing up the work over multiple processors with almost no additional cost in the case of a purely sequential execution.

10.1.4 Serialisation Formats. The PADS project [Mandelbaum et al. 2007] aims to let users quickly, correctly, and compositionally describe existing formats they have no control over. As they reminds us, ad-hoc serialisation formats abound be it in networking, logging, billing, or as output of measurement equipments in e.g. gene sequencing or molecular biology. Our current project is not offering this kind of versatility as we have decided to focus on a specific serialisation format with strong guarantees about the efficient access to subtrees. But our approach to defining correct-by-construction components could be leveraged in that setting too and bring users strong guarantees about the traversals they write.

ASN.1 [Larmouth 1999] gives users the ability to define a high-level specification of the exchange format (the 'abstract syntax') to be used in communications without the need to concern themselves with the actual encoding as bit patterns (the 'transfer syntax'). This separation between specification and implementation means that parsing and encoding can be defined once and for all by generic programming (here, a compiler turning specifications into code in the user's host language of choice). The main difference is once again our ability to program in a correct-by-construction manner over the values thus represented.

Yallop's automatic derivation of serialisers using an OCaml preprocessor [Yallop 2007] highlights the importance of empowering domain experts to take advantage of the specifics of the problem they are solving to minimise the size of the encoded data. By detecting sharing using a custom equality function respecting α-equivalence instead of the default one, he was able to serialise large lambda terms using only a quarter of the bytes OCaml's standard library marshaller.

10.2 Limitations and Future Work

Although our design is already proven to be functional by two implementations in Idris 2 and Agda respectively, we can always do better. In this section we are going to see what benefits future work could bring across the whole project.

10.2.1 A More Robust Library. For sake of ease of presentation we have not dealt with issues necessitating buffer resizing: in Section 8, we defined `execSerialising` by allocating a fixed size buffer and not worrying whether the whole content would fit. A real library would need to adopt a more robust approach akin to the one used in the implementation of Idris 2's own serialisation code: whenever we are about to write a byte to the buffer, we make sure there is either enough space left or we grow it.

In our library, the data types descriptions currently need to be defined as values in the host language. This opens up the opportunity for bugs if, say, we write a server in Idris 2 and a client in Agda and accidentally use two slightly different descriptions in the projects. This could be solved at the language level by equipping our dependently typed languages with type providers like Idris 1 had [Christiansen 2013]. This way the format could be loaded at compile time from the same file thus ensuring all the components are referring to the exact same specification.

10.2.2 A More Efficient Library. Looking at the code generated by Idris 2, we notice that our generic programs are not specialised and partially evaluated even when the types they are working on are statically known. Refactoring the library to use a continuation-passing-style approach does help the compiler generate slightly more specialised code but the results are in our opinion not good enough to justify forcing users to program in this more cumbersome style. A possible alternative would be to present users with macros rather than generic programs so that the partial evaluation would be guaranteed to happen at typechecking time. This however makes the process of defining the generic programs much more error prone. A more principled approach would be to extend Idris 2 with a proper treatment of staging e.g. by using a two-level type theory as suggested by Kovács [Kovács 2022].

Our serialisation format has been designed to avoid pointer-chasing and thus ensures entire subtrees can be easily copied by using the raw bytes. Correspondingly it currently does not support sharing. This could however be a crucial feature for trees with a lot of duplicated nodes and we would like to allow users to, using the same interface, easily pick between different serialisation formats so that the library ends up using the one that suits their application best. To this end, we could take inspiration from Yallop's definition of preprocessors generating serialisers [Yallop 2007]. It maintains an object map containing the already serialised nodes and uses it to maximally detect sharing and maintain it both when serialising and deserialising.

Our current approach allows us to define a correct-by-construction `sum` operating directly on serialised data but it does not eliminate the call stack used in the naïve functional implementation. Converting a fold to a tail recursive function in a generic manner is a well studied problem and the existing solutions [McBride 2008; Tomé Cortiñas and Swierstra 2018] should be fairly straightforward, if time-consuming, to port to our setting.

10.2.3 A More Expressive Universe of Descriptions. We have used a minimal universe to demonstrate our approach but a practical application would require the ability to store more than just raw bytes. An easy extension is to add support for all of the numeric types of known size that Idris 2 offers (`Bits{8,16,32,64}`, `Int{8,16,32,64}`), for `Bool` as well as a unbounded data such as `Nat`, or `String` as long as an extra offset is provided for each value.

The storage of values smaller than a byte (here `Bool`) naturally raises the question of bit packing: why store eight booleans as eight bytes when they could fit in a single one? Our recent work [Allais 2023] on the efficient runtime representation of inductive families as values of Idris 2's primitive types points us in the direction of a solution.

A natural next candidate is a universe allowing the definition of parametrised types [Löh and Magalhães 2011]: we should be able to implement functions over arbitrary (`List a`) values stored in a buffer, provided that we know that `a` is serialisable. This was already an explicit need in ASN.1 [Larmouth 1999], reflecting that protocols often leave 'holes' where the content of the protocol's higher layer is to be inserted.

Next, we will want to consider a universe of indexed data: we can currently natively model algebraic datatypes such as lists or trees, we can use the host language to compute the description of vectors by induction on their length, but we cannot model arbitrary type families [Dybjer 1994] e.g. correct-by-construction red-black trees.

Last but not least we may want to have a universe of descriptions closed under least fixpoints [Morris 2007] in order to represent rose trees for instance.

10.2.4 A More Expressive Library. Using McBride's generalisation of one hole contexts [McBride 2008] we ought to be able to give a more precise type to the combinator `(#)` used to build serialisation processes. When defining the serialisation of a given subtree, we ought to have access to pointers to the result of serialising any subtree to the left of it. In particular this would make building complete binary trees a lot faster by allowing us to rely on `copyData` for duplicating branches rather than running the computation twice.

Last, but not least we currently do not support in-place updates to the data stored in a buffer. This could however be beneficial for functions like `map`. It remains to be seen whether we can somehow leverage Idris 2's linear quantity annotation to provide users with serialised value that can be safely updated in place. This would turn our ongoing metaphor involving Hoare triples [Hoare 1969], heap pointers, and separation logic [Reynolds 2002] into a bona fide shallow embedding. Poulsen, Rouvoet, Tolmach, Krebbers, and Visser's pioneering work [Poulsen et al. 2018; Rouvoet 2021] on definitional interpreters for languages with references and the use of a shallowly embedded separation logic to minimise bookkeeping give us a clear set of techniques to adapt to our setting.

ACKNOWLEDGMENTS

We would like to thank Wouter Swierstra for his helpful comments on a draft of this paper.

This research was partially funded by the Engineering and Physical Sciences Research Council (grant number EP/T007265/1).

REFERENCES

Guillaume Allais. 2023. Builtin Types Viewed as Inductive Families. In *Programming Languages and Systems - 32nd European Symposium on Programming, ESOP 2023, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2023, Paris, France, April 22-27, 2023, Proceedings (Lecture Notes in Computer Science, Vol. 13990)*, Thomas Wies (Ed.). Springer, 113–139. https://doi.org/10.1007/978-3-031-30044-8_5

Thorsten Altenkirch and Conor McBride. 2002. Generic Programming within Dependently Typed Programming. In *Generic Programming, IFIP TC2/WG2.1 Working Conference on Generic Programming, July 11-12, 2002, Dagstuhl, Germany (IFIP Conference Proceedings, Vol. 243)*, Jeremy Gibbons and Johan Jeuring (Eds.). Kluwer, 1–20.

Robert Atkey. 2018. Syntax and Semantics of Quantitative Type Theory. In *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2018, Oxford, UK, July 09-12, 2018*, Anuj Dawar and Erich Grädel (Eds.). ACM, 56–65. https://doi.org/10.1145/3209108.3209189

Marcin Benke, Peter Dybjer, and Patrik Jansson. 2003. Universes for Generic Programs and Proofs in Dependent Type Theory. *Nordic J. of Computing* 10, 4 (Dec. 2003), 265–289. http://dl.acm.org/citation.cfm?id=985799.985801

Edwin C. Brady. 2021. Idris 2: Quantitative Type Theory in Practice. In *35th European Conference on Object-Oriented Programming, ECOOP 2021, July 11-17, 2021, Aarhus, Denmark (Virtual Conference) (LIPIcs, Vol. 194)*, Anders Møller and Manu Sridharan (Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 9:1–9:26. https://doi.org/10.4230/LIPIcs.ECOOP.2021.9

David Raymond Christiansen. 2013. Dependent type providers. In *Proceedings of the 9th ACM SIGPLAN workshop on Generic programming, WGP 2013, Boston, Massachusetts, USA, September 28, 2013*, Jacques Carette and Jeremiah Willcock (Eds.). ACM, 25–34. https://doi.org/10.1145/2502488.2502495

Peter Dybjer. 1994. Inductive Families. *Formal Aspects Comput.* 6, 4 (1994), 440–465. https://doi.org/10.1007/BF01211308

Jeremy Gibbons. 2006. Datatype-Generic Programming. In *Datatype-Generic Programming - International Spring School, SSDGP 2006, Nottingham, UK, April 24-27, 2006, Revised Lectures (Lecture Notes in Computer Science, Vol. 4719)*, Roland Carl Backhouse, Jeremy Gibbons, Ralf Hinze, and Johan Jeuring (Eds.). Springer, 1–71. https://doi.org/10.1007/978-3-540-76786-2_1

Tatsuya Hagino. 1987. A Typed Lambda Calculus with Categorical Type Constructors. In *Category Theory and Computer Science, Edinburgh, UK, September 7-9, 1987, Proceedings (Lecture Notes in Computer Science, Vol. 283)*, David H. Pitt, Axel Poigné, and David E. Rydeheard (Eds.). Springer, 140–157. https://doi.org/10.1007/3-540-18508-9_24

C. A. R. Hoare. 1969. An Axiomatic Basis for Computer Programming. *Commun. ACM* 12, 10 (1969), 576–580. https://doi.org/10.1145/363235.363259

Paul Hudak. 1996. Building Domain-Specific Embedded Languages. *ACM Comput. Surv.* 28, 4es (1996), 196. https://doi.org/10.1145/242224.242477

Patrik Jansson and Johan Jeuring. 1997. Polyp - A Polytypic Programming Language. In *Conference Record of POPL'97: The 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Papers Presented at the Symposium, Paris, France, 15-17 January 1997*, Peter Lee, Fritz Henglein, and Neil D. Jones (Eds.). ACM Press, 470–482. https://doi.org/10.1145/263699.263763

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

András Kovács. 2022. Staged compilation with two-level type theory. *Proc. ACM Program. Lang.* 6, ICFP (2022), 540–569. https://doi.org/10.1145/3547641

John Larmouth. 1999. *ASN.1 Complete.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

Andres Löh and José Pedro Magalhães. 2011. Generic programming with indexed functors. In *Proceedings of the seventh ACM SIGPLAN workshop on Generic programming, WGP@ICFP 2011, Tokyo, Japan, September 19-21, 2011*, Jaakko Järvi and Shin-Cheng Mu (Eds.). ACM, 1–12. https://doi.org/10.1145/2036918.2036920

Grant Malcolm. 1990. Data Structures and Program Transformation. *Sci. Comput. Program.* 14, 2-3 (1990), 255–279. https://doi.org/10.1016/0167-6423(90)90023-7

Yitzhak Mandelbaum, Kathleen Fisher, David Walker, Mary F. Fernández, and Artem Gleyzer. 2007. PADS/ML: a functional data description language. In *Proceedings of the 34th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2007, Nice, France, January 17-19, 2007*, Martin Hofmann and Matthias Felleisen (Eds.). ACM, 77–83. https://doi.org/10.1145/1190216.1190231

Conor McBride. 2008. Clowns to the left of me, jokers to the right (pearl): dissecting data structures. In *Proceedings of the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2008, San Francisco, California, USA, January 7-12, 2008*, George C. Necula and Philip Wadler (Eds.). ACM, 287–295. https://doi.org/10.1145/1328438.1328474

Conor McBride. 2016. I Got Plenty o' Nuttin'. In *A List of Successes That Can Change the World - Essays Dedicated to Philip Wadler on the Occasion of His 60th Birthday (Lecture Notes in Computer Science, Vol. 9600)*, Sam Lindley, Conor McBride, Philip W. Trinder, and Donald Sannella (Eds.). Springer, 207–233. https://doi.org/10.1007/978-3-319-30936-1_12

Conor McBride and James McKinna. 2004. The view from the left. *J. Funct. Program.* 14, 1 (2004), 69–111. https://doi.org/10.1017/S0956796803004829

Conor McBride and Ross Paterson. 2008. Applicative programming with effects. *J. Funct. Program.* 18, 1 (2008), 1–13. https://doi.org/10.1017/S0956796807006326

Gavin Mendel-Gleason. 2012. *Types and verification for infinite state systems.* Ph. D. Dissertation. Dublin City University.

Stefan Monnier. 2019. Inductive types deconstructed: the calculus of united constructions. In *Proceedings of the 4th ACM SIGPLAN International Workshop on Type-Driven Development, TyDe@ICFP 2019, Berlin, Germany, August 18, 2019*, David Darais and Jeremy Gibbons (Eds.). ACM, 52–63. https://doi.org/10.1145/3331554.3342607

Peter W. J. Morris. 2007. *Constructing Universes for Generic Programming.* Ph. D. Dissertation. University of Nottingham, UK. https://ethos.bl.uk/OrderDetails.do?uin=uk.bl.ethos.519405

Holger Pfeifer and Harald Rueß. 1999. Polytypic Proof Construction. In *Theorem Proving in Higher Order Logics, 12th International Conference, TPHOLs'99, Nice, France, September, 1999, Proceedings (Lecture Notes in Computer Science, Vol. 1690)*, Yves Bertot, Gilles Dowek, André Hirschowitz, Christine Paulin-Mohring, and Laurent Théry (Eds.). Springer, 55–72. https://doi.org/10.1007/3-540-48256-3_5

Casper Bach Poulsen, Arjen Rouvoet, Andrew Tolmach, Robbert Krebbers, and Eelco Visser. 2018. Intrinsically-typed definitional interpreters for imperative languages. *Proc. ACM Program. Lang.* 2, POPL (2018), 16:1–16:34. https://doi.org/10.1145/3158104

John C. Reynolds. 2002. Separation Logic: A Logic for Shared Mutable Data Structures. In *17th IEEE Symposium on Logic in Computer Science (LICS 2002), 22-25 July 2002, Copenhagen, Denmark, Proceedings.* IEEE Computer Society, 55–74. https://doi.org/10.1109/LICS.2002.1029817

Arjen Rouvoet. 2021. *Correct by Construction Language Implementations.* Ph. D. Dissertation. Delft University of Technology, Netherlands. https://doi.org/10.4233/uuid:f0312839-3444-41ee-9313-b07b21b59c11

Carlos Tomé Cortiñas and Wouter Swierstra. 2018. From algebra to abstract machine: a verified generic construction. In *Proceedings of the 3rd ACM SIGPLAN International Workshop on Type-Driven Development, TyDe@ICFP 2018, St. Louis, MO, USA, September 27, 2018*, Richard A. Eisenberg and Niki Vazou (Eds.). ACM, 78–90. https://doi.org/10.1145/3240719.3241787

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

Philip Wadler. 1987. Views: A Way for Pattern Matching to Cohabit with Data Abstraction. In *Conference Record of the Fourteenth Annual ACM Symposium on Principles of Programming Languages, Munich, Germany, January 21-23, 1987.* ACM Press, 307–313. https://doi.org/10.1145/41625.41653

Jeremy Yallop. 2007. Practical generic programming in OCaml. In *Proceedings of the ACM Workshop on ML, 2007, Freiburg, Germany, October 5, 2007*, Claudio V. Russo and Derek Dreyer (Eds.). ACM, 83–94. https://doi.org/10.1145/1292535.1292548

A SAFE IMPLEMENTATIONS OF FOLD

We include below the alternative definitions of `fold` (respectively processing inductive data and data stored in a buffer) which are seen as total by Idris 2. Each of them is mutually defined with what is essentially the supercompilation of (`\ d => fmap d (fold alg)`).

```
parameters {cs : Data nm} (alg : Alg cs a)

  fold : Data.Mu cs -> a
  fmapFold : (d : Desc{}) ->
             Data.Meaning d (Data.Mu cs) -> Data.Meaning d a

  fold (k # t) = alg k (fmapFold (description k) t)

  fmapFold None t = t
  fmapFold Byte t = t
  fmapFold (Prod d e) (s # t)
    = (fmapFold d s # fmapFold e t)
  fmapFold Rec t = fold t

parameters {cs : Data nm} (alg : Alg cs a)

  fold : Pointer.Mu cs t -> IO (Singleton (fold alg t))
  fmapFold : (d : Desc{}) -> forall t. Pointer.Meaning d cs t ->
             IO (Singleton (fmapFold alg d t))

  fold ptr
    = do k # t <- out ptr
         rec <- fmapFold (description k) t
         pure (alg k <$> rec)

  fmapFold d ptr = poke ptr >>= go d where

    go : (d : Desc{}) -> forall t. Poke d cs t ->
         IO (Singleton (fmapFold alg d t))
    go None {t} v = rewrite etaUnit t in pure (pure ())
    go Byte v = pure v
    go (Prod d e) (v # w)
      = do v <- fmapFold d v
           w <- fmapFold e w
           pure [| v # w |]
    go Rec v = fold v
```

B ACCESS PATTERNS: VIEWING VS. POKING

In this example we implement `rightmost`, the function walking down the rightmost branch of our type of binary trees and returning the content of its rightmost node (if it exists).

The first implementation is the most straightforward: use `view` to obtain the top constructor as well as an entire layer of deserialised values and pointers to substructures and inspect the constructor. If we have a leaf then there is no byte to return. If we have a node then call `rightmost` recursively and inspect the result: if we got `Nothing` back we are at the rightmost node and can return the current byte, otherwise simply propagate the result.

```
rightmost : Pointer.Mu Tree t -> IO (Maybe Bits8)
rightmost ptr = case !(view ptr) of
  "Leaf" # _ => pure Nothing
  "Node" # _ # b # r => do
    mval <- rightmost r
    case mval of
      Just _ => pure mval
      Nothing => pure (Just (getSingleton b))
```

In the alternative implementation we use `out` to expose the top constructor and then, in the node case, call `poke` multiple times to get our hands on the pointer to the right subtree. We inspect the result of recursively calling `rightmost` on this subtree and only deserialise the byte contained in the current node if the result we get back is `Nothing`.

```
rightmost : Pointer.Mu Tree t -> IO (Maybe Bits8)
rightmost ptr = case !(out ptr) of
  "Leaf" # _ => pure Nothing
  "Node" # el => do
    (_ # br) <- poke el
    (b # r) <- poke br
    mval <- rightmost !(poke r)
    case mval of
      Just _ => pure mval
      Nothing => do
        b <- poke b
        pure (Just (getSingleton b))
```

This will give rise to two different access patterns: the first function will have deserialised all of the bytes stored in the nodes along the tree's rightmost path whereas the second will only have deserialised the rightmost byte. Admittedly deserialising a byte is not extremely expensive but in a more realistic example we could have for instance been storing arbitrarily large values in these nodes. In that case it may be worth trading convenience for making sure we are not doing any unnecessary work.

C ANALOGY TO SEPARATION LOGIC

Some readers may feel uneasy about the fact that parts of our library are implemented using Idris 2 escape hatches. This section justifies this practice by drawing an analogy to separation logic and highlighting that this practice corresponds to giving an axiomatisation of the runtime behaviour of our library's core functions.

C.1 Interlude: Separation Logic

A Hoare triple [Hoare 1969] of the form

$$\{\, P \,\}\ e\ \{\, v.\, Q \,\}$$

states that under the precondition P, and binding the result of evaluating the expression e as v, we can prove that Q holds. One of the basic predicates of separation logic [Reynolds 2002] is a 'points to' assertion $(\ell \mapsto t)$ stating that the label ℓ points to a memory location containing t.

A separation logic proof system then typically consists in defining a language and providing axioms characterising the behaviour of each language construct. The simplest example involving memory is perhaps a language with pointers to bytes and a single `deref` construct dereferencing a

pointer. We can then give the following axiom

$$\{\ \ell \mapsto bs\ \}\ \texttt{deref}\ \ell\ \{\ v.\ bs = v * \ell \mapsto bs\ \}$$

to characterise `deref` by stating that the value it returns is precisely the one the pointer is referencing, and that the pointer is still valid and still referencing the same value after it has been dereferenced.

The axioms can be combined to prove statements about more complex programs such as the following silly one for instance. Here we state that if we dereference the pointer a first time, discard the result and then dereference it once more then we end up in the same situation as if we had dereferenced it only once.

$$\{\ \ell \mapsto bs\ \}\ \texttt{deref}\ \ell;\ \texttt{deref}\ \ell\ \{\ v.\ bs = v * \ell \mapsto bs\ \}$$

Note that in all of these rules bs is only present in the specification layer. `deref` itself cannot possibly return bs directly, it needs to actually perform an effectful operation that will read the memory cell's content.

C.2 Characterising Our Library

We are going to explain that we can see our library as a small embedded Domain Specific Language (eDSL) [Hudak 1996] that has `poke` and `out` as sole language constructs. Our main departure from separation logic is that we want to program in a correct-by-construction fashion and so the types of `poke` and `out` have to be just as informative as the axioms we would postulate in separation logic. This dual status of the basic building blocks being both executable programs *and* an axiomatic specification of their respective behaviour is precisely why their implementations in Idris 2 necessarily uses unsafe features.

We are going to write $\ell \overset{[\![\, d \,]\!]\ (\mu\ cs)}{\longmapsto} t$ for the assumption that we own a pointer ℓ of type (`Pointer.Meaning d cs t`), and $\ell \overset{\mu\ cs}{\longmapsto} t$ for the assumption that we own a pointer ℓ of type (`Pointer.Mu cs t`). In case we do not care about the type of the pointer at hand (e.g. because it can be easily inferred from the context), we will simply write $\ell \mapsto t$.

C.2.1 Axioms for `poke`. Thinking in terms of Hoare triples, if we have a pointer ℓ to a term t known to be a single byte then (`poke` ℓ) will return a byte bs and allow us to observe that t is equal to that byte.

$$\{\ \ell \overset{[\![\, \texttt{Byte} \,]\!]\ (\mu\ cs)}{\longmapsto} t\ \}\ \texttt{poke}\ \ell\ \{\ bs.\ t = bs * \ell \mapsto t\ \}$$

Similarly, if the pointer ℓ is for a pair then (`poke` ℓ) will reveal that the term t can be taken apart into the pairing of two terms t_1 and t_2 and return a pointer for each of these components.

$$\{\ \ell \overset{[\![\, \texttt{Prod}\ d_1\ d_2 \,]\!]\ (\mu\ cs)}{\longmapsto} t\ \}$$

$$\texttt{poke}\ \ell$$

$$\{\ (\ell_1,\ \ell_2).\ \exists t_1.\ \exists t_2.\ t = (t_1\ \#\ t_2) * \ell \mapsto t * \ell_1 \overset{[\![\, d_1 \,]\!]\ (\mu\ cs)}{\longmapsto} t_1 * \ell_2 \overset{[\![\, d_2 \,]\!]\ (\mu\ cs)}{\longmapsto} t_2\ \}$$

Last but not least, poking a pointer with the `Rec` description will return another pointer for the same value but at a different type.

$$\{\ \ell \overset{[\![\, \texttt{Rec} \,]\!]\ (\mu\ cs)}{\longmapsto} t\ \}\ \texttt{poke}\ \ell\ \{\ \ell_1.\ \ell_1 \overset{\mu\ cs}{\longmapsto} t * \ell \mapsto t\ \}$$

C.2.2 Example of a Derived Rule for `layer`. Given that `layer` is defined in terms of `poke`, we do not need to postulate any axioms to characterise it and can instead prove lemmas. We will skip the proofs here but give an example of a derived rule. Using the description (`Prod Rec` (`Prod Byte Rec`)) of the arguments to a node in our running example of binary trees, `layer`'s behaviour would be characterised by the following statement.

$$\{\ \ell \xmapsto{[\![\texttt{Prod Rec}\ (\texttt{Prod Byte Rec})]\!]\ (\mu\, cs)} t\ \}$$

$$\texttt{layer}\ \ell$$

$$\{\ (\ell_1,\, bs,\, \ell_2).\ \exists t_1.\ \exists t_2.\ t = (t_1\ \#\ bs\ \#\ t_2) * \ell_1 \xmapsto{\mu\, cs} t_1 * \ell_2 \xmapsto{\mu\, cs} t_2 * \ell \mapsto t\ \}$$

It states that provided a pointer to such a meaning, calling `layer` would return a triple of a pointer ℓ_1 for the left subtree, the byte bs stored in the node, and a pointer ℓ_2 for the right subtree.

C.2.3 Axiom for `out`. The only other construct for our small DSL is the function `out`. Things are a lot simpler here as the return type is not defined by induction on the description. As a consequence we only need the following axiom.

$$\{\ \ell \xmapsto{\mu\, cs} t\ \}\ \texttt{out}\ \ell\ \{\ (k,\, \ell_1).\ \exists t_1.\ t = (k\ \#\ t_1) * \ell \mapsto t * \ell_1 \xmapsto{[\![\, cs_k\,]\!]\ (\mu\, cs)} t_1\ \}$$

It states that under the condition that ℓ points to t, (`out` ℓ) returns a pair of an index and a pointer to the meaning of the description associated to that index by cs, and allows us to learn that t is constructed using that index and that meaning.

By combining `out` and `layer` we could once more define a derived rule and prove e.g. that every tree can be taken apart as either a leaf or a node with a pointer to a left subtree, a byte, and a pointer to a right subtree i.e. what `view` does in our library.

Received 20 February 2007; revised 12 March 2009; accepted 5 June 2009