Programming Over Serialised Data

SPLV 2023

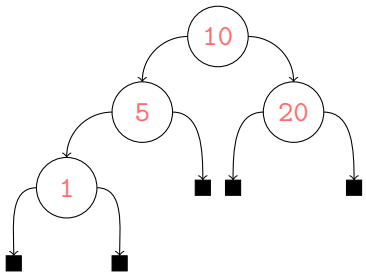
Guillaume Allais

July 24^{th}–28^{th} 2023

Contents

Motivation

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)
```

Figure 1: Summing the content of a binary tree, in Idris 2

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 defining a function summing up all of the nodes' contents. This function 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, 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)
```

Figure 2: A list of bytes representing a serialised binary tree

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); }}
```

Figure 3: Summing the content of a *serialised* binary tree, in C

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 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) [10, 2] as implemented in Idris 2 [4] empowers us to do just that purely in library code.

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 3.3, `view` in Section 3.4.4). 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.

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 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 [12], something we will explain in Section 1.4.

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.

3 Generic Programming over Serialised Data

Last but not least, as Altenkirch and McBride demonstrated [1]: "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 3.5).

```
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 Introduction to Idris

Idris 2 is a pure, strict, dependently typed functional language whose syntax is close to Haskell's and Agda's. Let us start with the basics: function and inductive types.

1.1 Basic Functions and Inductive Types

We can add top-level definitions by first declaring a name and its accompanying type and then giving defining clauses.

```
id : a -> a
id = \ x => x
```

```
(.) : (b -> c) -> (a -> b) -> a -> c
(g . f) x = g (f x)
```

On the left here we have the identity function, defined as an alias to an anonymous lambda function taking an argument `x` and immediately returning it. On the right we have the infix composition operator; its one defining clause states that applying the composition of `g` and `f` to an input `x` amounts to apply `g` to (`f x`).

Implicit Prenex Polymorphism

Unbound variables are automatically generalised in prenex position at the most general type possible and at quantity `0`. For example, the type of `id` written (`a -> a`) really means (`forall a. a -> a`) or, equivalently, (`{0 a : Type} -> a -> a`).

Quantities

Each binding site is annotated with a quantity potentially restricting the usage of the bound variable.

- `0` means that the value is runtime irrelevant and will be erased during compilation. This is typically useful for types or invariants e.g. a proof that a value is non-zero.
- `1` means that the value is linear: it needs to be used exactly once. This is typically useful for resources that get updated destructively e.g. file handles.
- `␣` means that the value is unrestricted.

The Empty Type

The empty type has no inhabitant and is therefore an ideal encoding of falsity. In Idris 2, it is called `Void` and is defined as an inductive type with no constructor.

```
data Void : Type where
```

Syntax Highlighting

The code in this document is semantically highlighted. Keywords are `black`, types are `blue`, definitions are `green`, constructors are `red`, bound variables are `purple`.

Using `Void`, we can define what it means for a type to be uninhabited: (`Not a`) is provable whenever from the assumption that `a` holds we can derive a proof of `Void`. In other words, whenever we have a function from `a` to `Void`

```
Not : Type -> Type
Not a = a -> Void
```

Types Are Terms

Declaring a type alias is exactly the same as introducing a new top-level definition. Types are arbitrary terms of type `Type`.

It is famously provable that if `a` holds then its double negation also holds. The converse is however neither provable nor disprovable in Idris 2.

```
doubleNeg : a -> Not (Not a)
doubleNeg x = \ f => f x
```

The Unit Type

The `Unit` has exactly one inhabitant, and thus is a good representation of things that are trivially true. It is defined as a record with constructor `MkUnit` and no field (we will shortly see a non-trivial record definition in Section 1.1).

```
record Unit where
  constructor MkUnit
```

We will see in Section 1.3 that it is provably true that all values of type `Unit` are equal to each other.

The Type of Booleans

We now have our first inductive type with more than one constructor: the type of boolean values. It offers two constructors: `True` and `False`.

```
data Bool = True | False
```

We can now define functions by pattern-matching over values of type `Bool`.

```
not : Bool -> Bool
not True = False
not False = True
```

Total Functions

Provided that you included `%default total`, the mere fact that a recursive function typechecks tells you that it is

1. *covering* that is to say its patterns handle all possible inputs
2. *terminating* that is to say that it is guaranteed to return a value on all possible inputs

The patterns in pattern matching definition can either be a constructor fully applied to further patterns ($\texttt{C}\ p_1 \cdots p_n$), a catchall pattern `_`, or a binding site `b`.

```
and : Bool -> Bool -> Bool
and False _ = False
and _ b = b
```

Overlapping Patterns

It is perfectly fine to use overlapping patterns to minimise the number of cases one need to spell out. One should however be aware that this means that all the equations spelt out by a pattern-matching definition may not hold definitionally: pattern-matching definitions have a first-match semantics and as long as prior clauses cannot be dismissed as impossible, the function will not reduce.

The Tuple Type Constructor

We sometimes want to pair two values, in which case it is convenient to have a tuple type. This our first 'proper' record definiton as it has two fields corresponding to each one of the tuple's components.

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

Records Are Datatypes

Record declarations are elaborated to a single-constructor data declaration together with one projection per field. For instance, the definition of `Tuple` as a record amounts to the following declarations:

```
data Tuple : Type -> Type -> Type where
  (#) : a -> b -> Tuple a b

fst : Tuple a b -> a
fst (x # y) = x

snd : Tuple a b -> b
snd (x # y) = y
```

Additionally, Idris 2 offers further syntactic sugar such as postfix dotted projections (`.fst` and `.snd`) and record updates ({ `fst :=` t , `snd $=` f }).

The Sum Type Constructor

```
data Either : Type -> Type -> Type where
  Left : a -> Either a b
  Right : b -> Either a b
```

The Type of Natural Numbers

For our first recursive type, we define the type of natural number. It is the smallest type that contains zero (`Z`) and is closed under successor (`S`).

```
data Nat = Z | S Nat
```

Runtime Optimisation of Types

During compilation, all the types whose runtime representation (obtained by erasing all the runtime irrelevant and forced arguments) looks like a primitive type are mapped directly to their native counterparts.
In particular all the values of type `Nat` are represented as runtime by a GMP-style unbounded integer.

The List Type Constructor

```
data List : Type -> Type where
  Nil : List a
  (::) : a -> List a -> List a
```

```
map : (a -> b) -> List a -> List b
map f [] = []
map f (x :: xs) = f x :: map f xs
```

1.2 Dependent Types and Indexed Families

The Vec Type Family

```
data Vec : Nat -> Type -> Type where
  Nil : Vec Z a
  (::) : a -> Vec n a -> Vec (S n) a
```

The Fin Type Family

```
data Fin : Nat -> Type where
  Z : Fin (S n)
  S : Fin n -> Fin (S n)
```

```
lookup : Vec n a -> Fin n -> a
lookup (x :: xs) Z = x
lookup (x :: xs) (S k) = lookup xs k
```

Dependent Pattern Matching

Large Elimination

Note that we can also declare families by defining a function proceeding by induction over the index. The (`` `is` ``) verb takes a type `a` and a boolean `b` and returns a type (`a` `` `is` `` `b`) stating that `a` does or does not hold depending on the value of `b`.

```
is : Type -> Bool -> Type
a `is` True = a
a `is` False = Not a
```

We can readily use this definition to state that `Unit` is inhabited and `Void` is not, and provide proofs of our claims.

```
UnitIsTrue : Unit `is` True
UnitIsTrue = ()
```

```
VoidIsFalse : Void `is` False
VoidIsFalse = \ x => x
```

The real power of dependent types really shines when the boolean value is not statically known. In the following example, `b` is a function input like any other and so the type (`Not a` `` `is` `` `not b`) is stuck. However in each of the function's defining clauses, `b` has been pattern-matched on and so has taken a canonical form. As a consequence the return type and the types of the other arguments have been *refined* by substituting `True` (respectively `False`) for `b` and then evaluating the resulting function calls. This allows us to prove the statement by appealing to two distinct principles: double negation of the assumption in one case, and a direct appeal to the assumption in the other. That last case is accepted because Idris 2 can see that both (`a` `` `is` `` `False`) and (`Not a` `` `is` `` `True`) evaluate to (`Not a`) and so they can be used interchangeably.

```
NotIsNot : (b : Bool) -> a `is` b -> Not a `is` not b
NotIsNot True = doubleNeg
NotIsNot False = \ x => x
```

Arbitrary Computations at Typechecking Time

The So Indexed Family

```
data So : Bool -> Type where
  Oh : So True
```

```
SoNotNotSo : So (not b) -> Not (So b)
SoNotNotSo soNot Oh = case soNot of {}
```

Empty Case Tree

```
NotSoSoNot : {b : Bool} -> Not (So b) -> So (not b)
NotSoSoNot {b = True} notSo = case notSo Oh of {}
NotSoSoNot {b = False} notSo = Oh
```

Named Application

1.3 The Equality Relation and Proofs

As we have seen in Section 1.2, some values are seen as equal to each other simply by computation. This internal notion of equality which is decided automatically by Idris 2 is called *judgemental* or *definitional* equality. Users can define a different (but related) notion of equality

```
data (===) : {a : Type} -> (x : a) -> a -> Type where
  Refl : x === x
```

Its one data constructor `Refl` can only ever be used to construct trivial proofs of equality.

Eta Laws

So called η-laws establish that some types have a unique canonical form.

We can for instance prove that all of the elements of the unit type are equal to each other.

```
etaUnit : (x, y : Unit) -> x === y
etaUnit MkUnit MkUnit = Refl
```

We can also prove that any tuple is equal to the pairing of its first and second projection.

```
etaTuple : (p : Tuple a b) -> p === (fst p # snd p)
etaTuple (x # y) = Refl
```

Proofs by Case Analysis

```
notInvolutive : (b : Bool) -> not (not b) === b
notInvolutive True = Refl
notInvolutive False = Refl
```

```
andAssociative : (a, b, c : Bool) ->
  and a (and b c) === and (and a b) c
andAssociative False _ _ = Refl
andAssociative True _ _ = Refl
```

Proofs by Induction

```
mapFusion : (g : b -> c) -> (f : a -> b) -> (xs : List a) ->
  map g (map f xs) === map (g . f) xs
mapFusion g f [] = Refl
mapFusion g f (x :: xs) = cong (g (f x) ::) (mapFusion g f xs)
```

1.4 The Singleton Family

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* [12] 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 can for instance write the following function adding three singleton-wrapped natural numbers:

```
add3 : {0 m, n, p : Nat} ->
  Singleton m -> Singleton n -> Singleton p ->
  Singleton (m + n + p)
add3 m n p = [| [| m + n |] + p |]
```

1.5 Views

A view in the sense of Wadler [17], and subsequently refined by McBride and McKinna [11] 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
```

2 Generic Programming

> If you wish to make apple pie from scratch,
> you must first invent the universe
>
> — Carl Sagan

We saw in the previous chapter a number of inductive data types. Had we wanted to, we could have defined an induction principle for each one of them. We are now going to see that this can be done once and for all by using generic programming.

In order to talk generically about an entire class of datatypes without needing to modify the host language we are going to perform a universe construction [3, 14, 8]. 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 [15].

2.1 A Universe of Discourse

The key observation driving this approach is that inductive types are defined by listing the types of each one of their constructors. Intuitively we can think of each constructor type as the description of the shape of a single "layer" of inductive tree. Some of these constructors have recursive positions (represented below by puzzle-shaped ports) in which sub-trees will be inserted while others do not, in which case they serve as base cases.

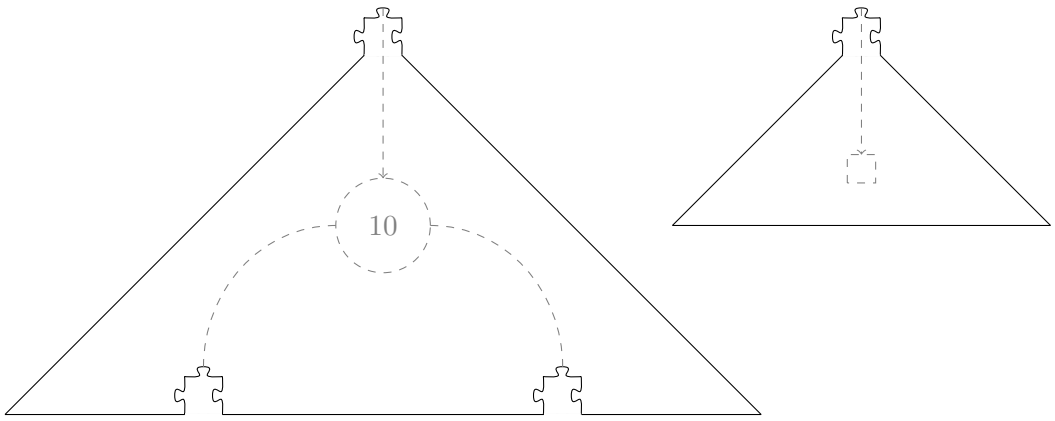


Figure 2.1: Constructors `node` and `leaf` as Layer Shapes

Multi-Sorted Shapes

In the figure we are using a single puzzle piece to denote both the type of the overall tree the constructor produces and the type of the possible subtrees.
In a more general setting however we could have multiple puzzle pieces: either because we are mutually defining multiple inductive types, or because we are defining an indexed family.

By defining a language to describe the shape of these pieces, we are able to define arbitrary inductive types. For brevety's sake we will use a small language of datatype descriptions throughout. In our setting constructors are essentially arbitrarily nested tuples of values of type unit, bytes, and recursive substructures.

Desc

```
data Desc : Type where
  None : Desc
  Byte : Desc
  Pair : (d, e : Desc) -> Desc
  Rec : Desc
```

`None` is the description of values of type unit. `Byte` is the description of bytes. `Prod` gives us the ability to pair two descriptions together. Last but not least, `Rec` is a position for a subtree.

We can immediately give these description a meaning using large elimination: provided a `Type` to use for the `Rec` position, we can return the type of nested tuples associated to each description.

```
Meaning : Desc -> Type -> Type
Meaning None x = ()
Meaning Byte x = Bits8
Meaning (Pair d e) x = Tuple (Meaning d x) (Meaning e x)
Meaning Rec x = x
```

`None` is interpreted using the unit type, `Byte` is mapped to the built-in type of bytes `Bits8`, `Pair` gives rise to tuples, and `Rec` is interpreted using the parameter.

Constructor

We represent a constructor as a record packing together a name for the constructor and the description of its arguments

```
record Constructor where
  constructor (::)
  name : String
  description : Desc
```

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
Leaf = "Leaf" :: None
```

```
Node : Constructor
Node = "Node" :: Pair Rec (Pair Byte Rec)
```

Data

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 where
  constructor MkData
  {consNumber : Nat}
  constructors : Vect consNumber Constructor
```

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.

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

Syntactic Sugar for Lists

List literals are desugared by selecting, in a type-directed manner, an appropriate `Nil` and `(::)`. This allows us to reuse them for `List`, `Vec`, and any other user-defined list-like type.

Now that we know how to declare inductive types, we need to give a semantics for these declarations.

2.2 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 [9], itself building on Hagino's categorically-inspired definition of a lambda calculus with a generic notion of datatypes [7].

2.2.1 Constructor Choice

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) 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} -> Index cs -> Desc
description (MkIndex k) = description (index 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} -> (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.

Elaboration of String Literals

String literals are elaborated by selecting, in a type-directed manner, an appropriate `fromString` function in the scope and applying it to the literal.
In particular, this means that the function above can be used to implicitly coerce patterns of the form `"Leaf"` or `"Node"` to the corresponding (`Index Tree`) pattern, thus providing (programmable) sugar.

2.2.2 Initial Algebra Semantics

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 -> 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 -> 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 the function `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.

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

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

Type-directed Disambiguation

The leftmost `(#)` used in each definition corresponds to the `Mu` constructor while later ones are `Tuple` constructors. We can use this uniform notation for all of these pairing notions because all constructor applications (and more generally all function applications) are disambiguated in a type-directed manner.

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)
```

2.3 Generic Functions

We claimed that `Desc` is a description language for a class of strictly positive endofunctors on `Type`. The function `Meaning` gave us their action on objects, and we can now define by generic programming their action on morphisms. We once again proceed by induction on the description.

```
fmap : (d : Desc) -> (x -> y) -> Meaning d x -> Meaning d y
fmap None f v = v
fmap Byte f v = v
fmap (Pair 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.

2.3.1 Fold

We claimed that `Mu` gives us the initial fixpoint for the `Data` algebras i.e. that given any other algebra over a type `a`, from a term of type (`Mu cs`), we can compute an `a`. This is witnessed by the following generic definition of the `fold` function:

```
fold : {cs : Data} -> Alg cs a -> Mu cs -> a
fold alg t
  = let (k # v) = t in
    let rec = assert_total (fmap _ (fold alg) v) in
    alg k rec
```

We first match on the term's top constructor, use `fmap` 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 and that the whole definition is therefore total.

2.3.2 Safe Fold by Manual Supercompilation

As we are now going to see, we can easily bypass the need for `assert_total` by mutually defining `fold` together with an inlined and specialised version of (`fmap _ (fold alg)`). `fold` becomes a simple pattern-match on the input's head constructor followed by an immediate call to `alg` on the result of the supercompiled `fmapfold` function.

```
fold : {cs : Data} -> Alg cs a -> Mu cs -> a
fold alg (k # v) = alg k (fmapfold alg _ v)
```

The implementation of the function `fmapfold` is unsurprising: it has the structure of (`fmap f`) but calls to `f` have been replaced with recursive calls to `fold`.

```
fmapfold : {cs : Data} -> Alg cs a -> (d : Desc) ->
           Meaning d (Mu cs) -> Meaning d a
fmapfold alg None v = v
fmapfold alg Byte v = v
fmapfold alg (Pair d e) (v # w) = (fmapfold alg d v # fmapfold alg e w)
fmapfold alg Rec v = fold alg v
```

Although this is systematically doable, we find it cumbersome and so prefer to use `assert_total` in this kind of situation. In an ideal type theory these supercompilation steps, whose sole purpose is to satisfy the totality checker, would be automatically performed by the compiler [13].

Further generic programming can yield other useful programs e.g. a generic proof that tree equality is decidable or a generic definition of zippers [8].

2.4 Generic Proofs

To declare a generic induction principle for our inductive types, we need a way to state that a property already holds true for all of the subtrees of a given constructor.

2.4.1 The Pointwise Lifting of a Predicate

We can compute the predicate lifting corresponding to universal quantification over all subtrees. It takes a description `d`, a predicate over a type `x` and returns a predicate over meanings of `d` with `x` subtrees.

```
All : (d : Desc) -> (x -> Type) -> Meaning d x -> Type
All None p v = ()
All Byte p v = ()
All (Pair d e) p (v # w) = (All d p v, All e p w)
All Rec p v = p v
```

The definition proceeds by induction on the description. If the description is `None` or `Byte` then the predicate is trivially true hence the use of the unit type. Faced with a `Pair` we demand that the predicate holds true of both components. Last but not least, `Rec` marks a subtree position and so we demand that the predicate holds true of the value of type (`Meaning Rec x`) i.e. `x`.

We can readily implement a very simple proof: if a predicate over `x` is known to be universally true then we can prove that its lifting holds of all the (`Meaning d x`). We once again proceed by induction over the definition.

```
all : (p : x -> Type) -> (f : (v : x) -> p v) ->
  (d : Desc) -> (v : Meaning d x) -> All d p v
all p f None v = ()
all p f Byte v = ()
all p f (Pair d e) (v # w) = (all p f d v, all p f e w)
all p f Rec v = f v
```

As we are going to see shortly, this will prove useful in our implementation of the generic induction principle.

2.4.2 The Generic Induction Principle

Let us start by explicitly stating the step case of the induction principle: given a predicate `p` over trees, for any choice of constructor `k` and any corresponding layer `v`, from the assumption that `p` holds true of all of the subtrees contained in `v` we should be able to prove that `p` holds true of the tree (`k # v`).

```
InductionStep : {cs : Data} -> (p : Mu cs -> Type) -> Type
InductionStep p
  = (k : Index cs) -> (v : Meaning (description k) (Mu cs)) ->
    All (description k) p v -> p (k # v)
```

The induction principle for a datatype described by `cs` is therefore the function that, given a proof of the induction step case for `cs` and predicate `p`, returns a function proving that any tree `t` of type (`Mu cs`) necessarily satisfies `p`.

```
induction : {cs : Data} -> (p : Mu cs -> Type) ->
  (step : InductionStep p) -> (t : Mu cs) -> p t
induction p step (k # v)
  = step k v $ assert_total
  $ all p (induction p step) (description k) v
```

The implementation is similar to `fold`'s: we first pattern-match on the tree's head constructor, use `all` to recursively apply (`induction p step`) to all the subtrees and then conclude by applying the step function.

We had to, just like in the definition of `fold`, make use of `assert_total`. This is once again due to the fact that the recursive calls performed via a call to a higher-order function (here the call to `all`, and in the definition of `fold` the call to `fmap`) obfuscates

the fact that all the recursive calls are perform on strict subterms. We could once again manually supercompile (`all p (induction p step) _`) to obtain a pair of definitions that is seen to be safe by Idris 2. This is left as an exercise for the reader.

3 Programming Over Serialised Data

3.1 Revisiting Our Universe Definition

Now that we are specifically focusing on programming over serialised data, we are going to revisit our universe of descriptions to incorporate some useful invariants.

Desc

Let us start with `Desc` itself. We have modified it to add three indices corresponding to three crucial invariants being tracked.

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

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 (the context in which the description appears tells us whether it is currently the rightmost branch) whereas `static` and `offsets` are 'output' indices (the description itself tells us what these sizes are).

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.

Constructors

A constructor description is now represented as a record packing together not only a name for the constructor, and the description of its arguments (which is, by virtue of being used at the toplevel, in rightmost position) but also 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
```

This definition (and the following one for `Data`) is parametrised over the notion of constructor name for reasons that will become obvious in Section 3.2

Data

Datatype descriptions are once again given by the pairing of the number of constructors together with a vector of descriptions for each one of these.

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

3.2 Choosing a Serialisation Format

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 chapter 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 **??**).

3.2.1 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.

Dependent Type Providers

Various components of a big software project written in potentially different dependently typed languages can be kept in sync by having the serialisation format be stored in a single configuration file and loaded during typechecking by a dependent type provider [6].
Unfortunately Idris 2 does not yet support Dependent Type Providers like Idris 1 did.

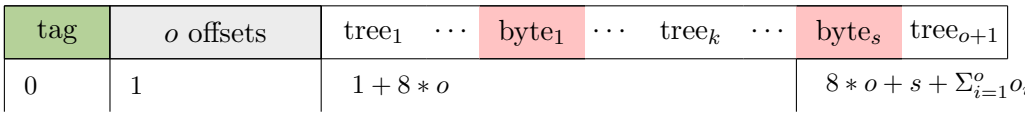
The encoding of a data description starts with a byte giving us the number of constructors, followed by these constructors' respective descriptions serialised one after the other. `None` is represented by `00`, `Byte` is represented by `01`, (`Prod d e`) is represented by `02` followed by the representation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

3.2.2 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.

Leaf	Node				
00	01	offset	left subtree	byte	right subtree
0	0	1	9	$9 + o_1$	$10 + o_1$

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

3.3 Meaning as Pointers

Now that we know the serialisation format, we can give a meaning to constructor and data descriptions as pointers into a buffer.

3.3.1 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)
```

The pointer type is parametrised by the `Data` description of the buffer's content it points to, and indexed by a tree representing the value that is stored in serialised format in the buffer. These tree indices will be erased at runtime and so are only here to be used in the specification layer of our library.

Indexing a Family by its Meaning

Indexing a complex but efficient representation by its simpler and obviously correct counterpart is a common technique that allows for the correct-by-construction implementation of complex programs [5].

For now, it is enough to think of these tree indices as a lightweight version of the 'points to' assertions used in separation logic [16] when reasoning about imperative programs. The actual definition of the record type is as follows:

```
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.

3.3.2 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 pos 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.

Pattern-Matching Bind

3.3.3 Reading a Tree From a File

Just like we can write trees to files, we can also read trees from files. The function `readFromFile` takes a data description and a filepath and returns a pointer to the root of the tree contained in the file (if any).

```
readFromFile : {default True safe : Bool} ->
               (cs : Data nm) -> String ->
               IO (Exists (Pointer.Mu cs))
readFromFile cs fp
  = do Right buf <- createBufferFromFile fp
         | Left err => failWith (show err)
       skip <- getInt buf 0
       when safe $ do
         cs' <- getData buf 8
         unless (eqData cs cs') $ failWith $ unlines
           [ "Description mismatch:"
           , "expected:"
           , show cs
           , "but got:"
           , show cs'
           ]
       let pos = skip + 8
       pure (Evidence t (MkMu buf pos (!(rawSize buf) - pos)))

  where 0 t : Data.Mu cs -- postulated as an abstract value
```

The first step is to attempt to load the file as a buffer of raw bytes. Once that's done, we decode the offset corresponding to the header size and, provided that the function was called in safe mode, decode the data description contained in the header and compare it to the data description the user wants to load the file's content as. Finally, we manufacture a pointer pointing immediately past the header i.e. at the start of the data block. Note that the pointer is indexed over an arbitrary runtime irrelevant tree `t` that we happily postulate because one ought to exist if the file is indeed valid.

Default Values

Implicit arguments marked `default` can be assigned a default value. If the caller does not explicitly sets a value for this argument then the default value is used.

3.4 Inspecting a Buffer's Content

Now that we have pointers and can save and load the tree they are standing for, we are only missing the ability to look at the content they are pointing to.

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 the following generic correct-by-construction version of `fold` operating over trees stored in a buffer (cf. Section 3.5) that looks almost exactly like its pure counterpart.

```
fold : {cs : Data nm} -> (alg : Alg cs a) ->
       forall t. Pointer.Mu cs t ->
       IO (Singleton (Data.fold alg t))
fold alg ptr
  = do k # t <- out ptr
       rec <- assert_total (fmap _ _ (fold alg) t)
       pure (alg k <$> rec)
```

3.4.1 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.

```
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.

3.4.2 Extracting One Layer

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

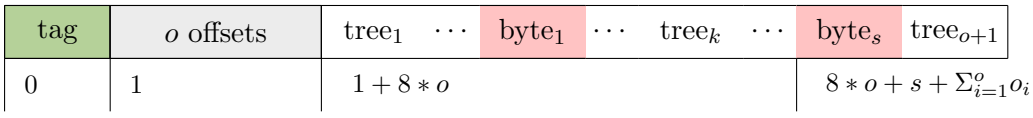
This function can easily be implemented by induction on the description and repeatedly calling `poke` to expose the values one by one.

```
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
```

3.4.3 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 3.2.2 and repeated below, 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)
```

As a first step, we can define the function `getIndex` to get our hands on the index of the head constructor. 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`.

```
getIndex : {cs : Data nm} -> forall t. Pointer.Mu cs t -> IO (Index cs)
getIndex mu = do
  tag <- getBits8 (muBuffer mu) (muPosition mu)
  let Just k = natToFin (cast tag) (consNumber cs)
    | _ => failWith "Invalid representation"
  pure (MkIndex k)
```

The `out` function type states that given a pointer to a tree `t` of type `cs` we can get a value of type (`Out cs t`) i.e. we can get a view revealing 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
  k <- getIndex mu
  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 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.

3.4.4 Offering a Convenient View

We can combine `out` and `layer` to obtain the `view` function we used in our motivating examples in Section 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)
```

Cost of View

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.
For instance, there is no need to deserialise all of the nodes encountered when descending down the rightmost branch of a tree if our only goal is to return the value of the leaf at the end.

3.4.5 Generic Deserialisation

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.

3.5 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 # v <- out ptr
       rec <- assert_total (fmap _ _ (fold alg) v)
       pure (alg k <$> rec)
```

We once again (cf. Section 2.3.1) 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.

3.6 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.

3.6.1 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.

3.6.2 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 3.2.2 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 1.4. 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.

3.6.3 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`. This is spectacularly faster than a deep copying process traversing the tree.

3.6.4 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 3.3.2.

3.6.5 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 3.6 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.

Bibliography

[1] Thorsten Altenkirch and Conor McBride. "Generic Programming within Dependently Typed Programming". In: *Generic Programming, IFIP TC2/WG2.1 Working Conference on Generic Programming, July 11-12, 2002, Dagstuhl, Germany*. Ed. by Jeremy Gibbons and Johan Jeuring. Vol. 243. IFIP Conference Proceedings. Kluwer, 2002, pp. 1–20.

[2] Robert Atkey. "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*. Ed. by Anuj Dawar and Erich Grädel. ACM, 2018, pp. 56–65. DOI: 10.1145/3209108.3209189.

[3] Marcin Benke, Peter Dybjer, and Patrik Jansson. "Universes for Generic Programs and Proofs in Dependent Type Theory". In: *Nordic J. of Computing* 10.4 (Dec. 2003), pp. 265–289. ISSN: 1236-6064. URL: http://dl.acm.org/citation.cfm?id=985799.985801.

[4] Edwin C. Brady. "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)*. Ed. by Anders Møller and Manu Sridharan. Vol. 194. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021, 9:1–9:26. DOI: 10.4230/LIPIcs.ECOOP.2021.9.

[5] Edwin C. Brady, James McKinna, and Kevin Hammond. "Constructing Correct Circuits: Verification of Functional Aspects of Hardware Specifications with Dependent Types". In: *Proceedings of the Eighth Symposium on Trends in Functional Programming, TFP 2007, New York City, New York, USA, April 2-4. 2007*. Ed. by Marco T. Morazán. Vol. 8. Trends in Functional Programming. Intellect, 2007, pp. 159–176.

[6] David Raymond Christiansen. "Dependent type providers". In: *Proceedings of the 9th ACM SIGPLAN workshop on Generic programming, WGP 2013, Boston, Massachusetts, USA, September 28, 2013*. Ed. by Jacques Carette and Jeremiah Willcock. ACM, 2013, pp. 25–34. DOI: 10.1145/2502488.2502495. URL: https://doi.org/10.1145/2502488.2502495.

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

[8] Andres Löh and José Pedro Magalhães. "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*. Ed. by Jaakko Järvi and Shin-Cheng Mu. ACM, 2011, pp. 1–12. DOI: 10.1145/2036918.2036920. URL: https://doi.org/10.1145/2036918.2036920.

[9] Grant Malcolm. "Data Structures and Program Transformation". In: *Sci. Comput. Program.* 14.2-3 (1990), pp. 255–279. DOI: 10.1016/0167-6423(90)90023-7. URL: https://doi.org/10.1016/0167-6423(90)90023-7.

[10] Conor McBride. "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*. Ed. by Sam Lindley et al. Vol. 9600. Lecture Notes in Computer Science. Springer, 2016, pp. 207–233. DOI: 10.1007/978-3-319-30936-1_12.

[11] Conor McBride and James McKinna. "The view from the left". In: *J. Funct. Program.* 14.1 (2004), pp. 69–111. DOI: 10.1017/S0956796803004829.

[12] Conor McBride and Ross Paterson. "Applicative programming with effects". In: *J. Funct. Program.* 18.1 (2008), pp. 1–13. DOI: 10.1017/S0956796807006326. URL: https://doi.org/10.1017/S0956796807006326.

[13] Gavin Mendel-Gleason. "Types and verification for infinite state systems". PhD thesis. Dublin City University, 2012.

[14] Peter W. J. Morris. "Constructing Universes for Generic Programming". PhD thesis. University of Nottingham, UK, 2007. URL: https://ethos.bl.uk/OrderDetails.do?uin=uk.bl.ethos.519405.

[15] Holger Pfeifer and Harald Rueß. "Polytypic Proof Construction". In: *Theorem Proving in Higher Order Logics, 12th International Conference, TPHOLs'99, Nice, France, September, 1999, Proceedings*. Ed. by Yves Bertot et al. Vol. 1690. Lecture Notes in Computer Science. Springer, 1999, pp. 55–72. DOI: 10.1007/3-540-48256-3_5. URL: https://doi.org/10.1007/3-540-48256-3_5.

[16] John C. Reynolds. "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, 2002, pp. 55–74. DOI: 10.1109/LICS.2002.1029817. URL: https://doi.org/10.1109/LICS.2002.1029817.

[17] Philip Wadler. "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, 1987, pp. 307–313. DOI: 10.1145/41625.41653.

List of Idris 2 Features