Restoring (part of) the friendship between recursion schemes and without-K (provisional)

I'll have to grab a UU-template at some point

Samuel Klumpers
6057314

March 14, 2023

Contents

This document is generated from a literate agda file!

Abstract

The preliminary goal of this thesis is to introduce, among others, the concepts of the structure identity principle, numerical representations, and ornamentations, which are then combined to simplify the presentation and verification of finger trees, as a demonstration of the generalizability and improved compactness and security of the resulting code.

1 Introduction (to be updated; out of sync since reducing friction)

Most of the time when we are Agda-ing [Tea23] we are trying to un-Haskell ourselves, e.g., not take the head of an empty list. In this example, we can make `head` total by switching to length-indexed lists: vectors. We have now effectively doubled the size of our code base, since functions like `_++_` which we had for lists, will also have to be reimplemented for vectors.

To make things worse; often, after coping with the overloaded names resulting from Agda-ing by shoving them into a different namespace, we also find out that lists nor vectors are efficient containers to begin with. Maybe binary trees are better. We now need four times the number of definitions to keep everything working, and, if we start proving things, we will also have to prove everything fourfold. (Not to mention that reasoning about trees is probably going to be harder than reasoning about lists). This inefficiency has sparked (my) interest in ways to deal with the situation.

Following [DM14] and [KG16], we can describe the relation between list and vector using the mechanism of ornamentation. This leads them to define the concept of patches, which can aid us when defining `_++_` for the second time by forcing the new version to be coherent. In fact, the algebraic nature of ornaments can even get us the definition of the vector type for free, if we started by defining lists relative to natural numbers [McB14]. Such constructions rely heavily on descriptions of datastructures and often come with limitations in their expressiveness. These descriptions in turn impose additional ballast on the programmer, leading us to investigate reflection like in [EC22] as a means to bring datatypes and descriptions closer when possible.

From a different direction, [HS22] gives methods by which we can show two implementations of some structure to be equivalent. With this, we can simply transport all proofs about `_++_` we have for lists over to the implementation for trees, provided that we show them to be equivalent as appendable containers. This process can also be automated by some heavy generics, but instead, we resort to cubical; which hosts a range of research like [Ang+20] tailored to the problem describing equivalences of structures.

We can liken the situation to movement on a plane, where ornamentation moves us vertically by modifying constructors or indices, and structured equivalences move us horizontally to and from equivalent but more equivalent implementations. In this paper, we will investigate a variety of means of moving around structures and proofs, and ways to make this more efficient or less intrusive.

Currently, all sections mainly reintroduce or reformulate existing research, with some spots of new ideas and original examples here and there. In Section 2, we will look at how proofs on unary naturals can be moved to binary naturals. Then in Section 3 we recall how numeral systems in particular induce container types, which we attempt to reformulate in the language of ornaments in Subsection 3.2.

2 How Cubical Agda helps our binary numbers (ready)

Let us quickly review the small set of features in Cubical Agda that we will be using extensively throughout this article.[1] We note that there are some downsides to cubical, such as that

```
{-# OPTIONS --cubical #-}
```

also implies the negation of axiom K, which in turn complicates both some termination checking and some universe levels.[2] Furthermore, if we use certain homotopical constructions, like set quotients, we will also have to prove that our types are sets, before we can use them.

Of course, this downside is more than offset by the benefits of changing our primitive notion of equality, which we will see makes it easier to show that "equivalent" structures behave identically. Here, equality arises not (directly) from the indexed inductive definition we are used to, but rather from the presence of the interval type I. This type represents a set of two points i0 and i1, which are considered "identified" in the sense that they are connected by a path. To define a function out of this type, we also have to define the function on all the intermediate points, which is why we call such a function a "path". Terms of other types are then considered identified when there is a path between them.

As an added benefit, this different perspective gives intuitive interpretations to some proofs of equality, like

```
sym : x ≡ y → y ≡ x
sym p i = p (~ i)
```

where ~_ is the interval reversal, swapping i0 and i1, so that sym simply reverses the given path.

Furthermore, because we can now interpret paths in records and function differently, we get a host of "extensionality" for free. For example, a path in $A \to B$ is indeed a function which takes each i in I to a function $A \to B$. Using this, function extensionality becomes tautological

```
funExt : (∀ x → f x ≡ g x) → f ≡ g
funExt p i x = p x i
```

[1] [VMA19] gives a comprehensive introduction to cubical agda.

[2] In particular, this prompts rather far-reaching (but not fundamental) changes to the code of previous work, such as to that of [KG16] in Section 4.

Finally, while in "non-univalent" Agda bijections or isomorphisms do not play such a central role, much of our work will rest on equivalences, as the "HoTT-compatible" generalization of bijections. This is because the Glue type tells us that equivalent types fit together in a new type, in a way that guarantees univalence

```
    ua : ∀ {A B : Type ℓ} → A ≃ B → A ≡ B
```

This essentially states that "equivalent types are identified", such that type isomorphisms like $1 \to A \simeq A$ actually become paths $1 \to A \equiv A$, making it so that we can transport proofs along them. We will demonstrate this by a slightly more practical example.

2.1 Binary numbers

Let us motivate the cubical method by showing the equivalence of the "Peano" naturals and the "Leibniz" naturals. Recall that the Peano naturals are defined as

```
    data ℕ : Type where
      zero : ℕ
      suc : ℕ → ℕ
```

This definition enjoys a simple induction principle and has many proofs of its properties in standard libraries. However, it is too slow to be of practical use: most arithmetic operations defined on ℕ have time complexity in the order of the value of the result.

Of course, the alternative are the more performant binary numbers: the time complexities for binary numbers are usually logarithmic in the resultant values, but these are typically less well-covered in terms of proofs. This does not have to be a problem, because the ℕ naturals and the binary numbers should be equivalent after all!

Let us make this formal. We define the Leibniz naturals as follows:

```
    data Leibniz : Set where
      0b : Leibniz
      _1b : Leibniz → Leibniz
      _2b : Leibniz → Leibniz
```

Here, the 0b constructor encodes 0, while the _1b and _2b constructors respectively add a 1 and a 2 bit, under the usual interpretation of binary numbers:

```
    toℕ : Leibniz → ℕ
    toℕ 0b = 0
    toℕ (n 1b) = 1 N.+ 2 N.· toℕ n
    toℕ (n 2b) = 2 N.+ 2 N.· toℕ n
```

This defines one direction of the equivalence from ℕ to Leibniz, for the other direction, we can interpret a number in ℕ as a binary number by repeating the successor operation on binary numbers:

```
    bsuc : Leibniz → Leibniz
    bsuc 0b = 0b 1b
    bsuc (n 1b) = n 2b
```

```
bsuc (n 2b) = (bsuc n) 1b

fromℕ : ℕ → Leibniz
fromℕ ℕ.zero = 0b
fromℕ (ℕ.suc n) = bsuc (fromℕ n)
```
To show that toℕ is an isomorphism, we have to show that it is the inverse of fromℕ. For this, by induction on Leibniz and basic arithmetic on ℕ we see that
```
toℕ-suc : ∀ x → toℕ (bsuc x) ≡ ℕ.suc (toℕ x)
```
so toℕ respects successors. Similarly, by induction on ℕ we get
```
fromℕ-1+2· : ∀ x → fromℕ (1 N.+ 2 N.· x) ≡ (fromℕ x) 1b
```
and
```
fromℕ-2+2· : ∀ x → fromℕ (2 N.+ 2 N.· x) ≡ (fromℕ x) 2b
```
so that fromℕ respects even and odd numbers. We can then prove that applying toℕ and fromℕ after each other is the identity by repeating these lemmas
```
ℕ↔L : Iso ℕ Leibniz
ℕ↔L = iso fromℕ toℕ sec ret
  where
  sec : section fromℕ toℕ
  sec 0b = refl
  sec (n 1b) = fromℕ-1+2· (toℕ n) · cong _1b (sec n)
  sec (n 2b) = fromℕ-2+2· (toℕ n) · cong _2b (sec n)

  ret : retract fromℕ toℕ
  ret ℕ.zero = refl
  ret (ℕ.suc n) = toℕ-suc (fromℕ n) · cong ℕ.suc (ret n)
```
This isomorphism generalizes into an equivalence
```
ℕ≃L : ℕ ≃ Leibniz
ℕ≃L = isoToEquiv ℕ↔L
```
which, finally, lets us identify ℕ and Leibniz by univalence
```
ℕ≡L : ℕ ≡ Leibniz
ℕ≡L = ua ℕ≃L
```
The path ℕ≡L then allows us to transport properties from ℕ directly to Leibniz, e.g.,
```
isSetL : isSet Leibniz
isSetL = subst isSet ℕ≡L N.isSetℕ
```
This can be generalized even further to transport proofs about operations from ℕ to Leibniz.

2.2 Use as definition: functions from specifications

As an example, we will define addition of binary numbers. We could take
```
BinOp : Type → Type
BinOp A = A → A → A

_+′_ : BinOp Leibniz
_+′_ = subst BinOp ℕ≡L N._+_
```

But this would be rather inefficient, incurring an $O(n+m)$ overhead when adding $n+m$, so it would be better define addition on Leibniz directly. We would prefer to give a definition which makes use of the binary nature of Leibniz, while agreeing with the addition on ℕ. Such a definition can be derived from the specification "agrees with _+_", so we implement the following syntax for giving definitions by equational reasoning, inspired by the "use-as-definition" notation from [HS22]:

```
Def : {X : Type a} → X → Type a
Def {X = X} x = Σ' X λ y → x ≡ y

defined-by : {X : Type a} {x : X} → Def x → X
defined-by = fst

by-definition : {X : Type a} {x : X} → (d : Def x) → x ≡ defined-by d
by-definition = snd
```

which infers the definition from the right endpoint of a path using an implicit pair type

```
record Σ' (A : Set a) (B : A → Set b) : Set (ℓ-max a b) where
  constructor _use-as-def
  field
    {fst} : A
    snd : B fst

open Σ'

infix 1 _use-as-def
```

With this we can define addition on Leibniz and show it agrees with addition on ℕ in one motion

Tidy up these terms

```
plus-def : ∀ x y → Def (fromℕ (toℕ x N.+ toℕ y))
plus-def 0b y      = ℕ↔L .rightInv y use-as-def
plus-def (x 1b) 0b =
  bsuc (fromℕ (toℕ x N.+ (toℕ x N.+ ℕ.zero) N.+ ℕ.zero))
    ≡⟨ cong (bsuc ∘ fromℕ) (NP.+-zero (2 N.· toℕ x)) ⟩
  bsuc (fromℕ (toℕ x N.+ (toℕ x N.+ ℕ.zero)))
    ≡⟨ fromℕ-1+2· (toℕ x) ⟩
  fromℕ (toℕ x) 1b
    ≡⟨ cong _1b (ℕ↔L .rightInv x) ⟩
  x 1b ∎ use-as-def
plus-def (x 1b) (y 1b) =
  fromℕ ((1 N.+ 2 N.· toℕ x) N.+ (1 N.+ 2 N.· toℕ y))
    ≡⟨ cong fromℕ (Eq.eqToPath (eq (toℕ x) (toℕ y))) ⟩
  fromℕ (2 N.+ (2 N.· (toℕ x N.+ toℕ y)))
    ≡⟨ fromℕ-2+2· (toℕ x N.+ toℕ y) ⟩
  fromℕ (toℕ x N.+ toℕ y) 2b
    ≡⟨ cong _2b (by-definition (plus-def x y)) ⟩
  defined-by (plus-def x y) 2b ∎ use-as-def
    where
```

```
        eq : ∀ x y
           → (1 N.+ 2 N.· x) N.+ (1 N.+ 2 N.· y) Eq.≡ 2 N.+ (2 N.· (x N.+ y))
        eq = NS.solve-∀
    -- similar clauses omitted

    plus : ∀ x y → Leibniz
    plus x y = defined-by (plus-def x y)

    plus-coherent : ∀ x y → fromℕ (x N.+ y) ≡ plus (fromℕ x) (fromℕ y)
    plus-coherent x y = cong fromℕ
      (cong₂ N._+_ (sym (ℕ↔L .leftInv x)) (sym (ℕ↔L .leftInv _))) ·
        by-definition (plus-def (fromℕ x) (fromℕ y))
```

2.3 Structure Identity Principle

We see that as a consequence (modulo some PathP lemmas), we get a path from (ℕ, N.+) to (Leibniz, plus). More generally, we can view a type X combined with a function $f : X \to X \to X$ as a kind of structure, which in fact coincides with a magma. We can see that paths between magmas correspond to paths between the underlying types X and paths over this between their operations f. This observation is further generalized by the Structure Identity Principle (SIP), formalized in [Ang+20]. Given a structure, which in our case is just a binary operation

Use BinOp

```
    MagmaStr : Type → Type
    MagmaStr A = A → A → A
```

this principle produces an appropriate definition "structured equivalence" ι. The ι is such that if structures X, Y are ι-equivalent, then they are identified. In this case, the ι asks us to provide something with the same type as plus-coherent, so we have just shown that the plus magma on Leibniz

```
    MagmaL : Magma
    fst MagmaL = Leibniz
    snd MagmaL = plus
```

and the _+_ magma on ℕ and are identical

```
    Magmaℕ≃MagmaL : Magmaℕ ≡ MagmaL
    Magmaℕ≃MagmaL = equivFun (MagmaΣPath _ _) proof
      where
      proof : Magmaℕ ≃[ MagmaEquivStr ] MagmaL
      fst proof = ℕ≃L
      snd proof = plus-coherent
```

As a consequence, properties of _+_ directly yield corresponding properties of plus. For example,

```
    plus-assoc : Associative _≡_ plus
    plus-assoc = subst
      (λ A → Associative _≡_ (snd A))
      Magmaℕ≃MagmaL
      ℕ-assoc
```

3 Specifying types (ready)

While the practical applications of the last example do not stretch very far[3], the approach generalizes to the more relevant containers and their associated laws. In the same vein as the last section, we could define a simple but inefficient array type, and a more efficient implementation using trees. Then we can show that these are equivalent, such that when the simple type satisfies a set of laws, trees will satisfy them as well. We could then start developing all sorts of complex implementations fine-tuned to each operation and figure out how these are equivalent to some simpler type, but let us first take a step back, and investigate how we can make this approach run smoothly in a simpler example.

Rather than inductively defining a container and then showing that it is represented by a lookup function, we can go the other way around and define a type by insisting that it is equivalent to such a function. This approach, in particular the case in which one calculates a container with the same shape as a numeral system, was dubbed numerical representations in [Oka98], and has some formalized examples in, e.g., [HS22] and [KG16]. Numerical representations form the starting point for defining more complex datastructures based on simpler ones, so let us demonstrate such a calculation.

3.1 Numerical representations: from numbers to containers

We can compute the type of vectors starting from ℕ.[4] For simplicity, we define them as a type computing function via the "use-as-definition" notation from before. We expect vectors to be represented by

```
Lookup : Type → ℕ → Type
Lookup A n = Fin n → A
```

where we use the finite type Fin as an index into vector. Using this representation as a specification, we can compute both Fin and a type of vectors. The finite type can be computed from the evident definition

This is probably more convincing with equational reasoning.

```
Fin-def : ∀ n → Def (Σ[ m ∈ ℕ ] m < n)
Fin-def zero = ⊥-strict (λ ()) use-as-def
Fin-def (suc n) =
  ua (<-split n) ·
  cong (⊤ ⊎_) (by-definition (Fin-def n)) use-as-def

Fin : ℕ → Type
Fin n = defined-by (Fin-def n)
```

using

[3]Considering that ℕ is a candidate to be replaced by a more suitable unsigned integer type when compiling to Haskell anyway.

[4]This is adapted (and fairly abridged) from [HS22]

```
<-split : ∀ n → (Σ[ m ∈ ℕ ] m < suc n) ≃ (⊤ ⊎ (Σ[ m ∈ ℕ ] m < n))
```
Likewise, vectors can be computed by applying a sequence of type isomorphisms

The reasoning is a bit too close in indentation to the terms?

```
Vec-def : ∀ A n → Def (Lookup A n)
Vec-def A zero = isContr→≡Unit isContr⊥→A use-as-def
Vec-def A (suc n) =
  ((⊤ ⊎ Fin n) → A)
    ≡⟨ ua Π⊎≃ ⟩
  (⊤ → A) × (Fin n → A)
    ≡⟨ cong₂ _×_
         (UnitToTypePath A)
         (by-definition (Vec-def A n)) ⟩
  A × (defined-by (Vec-def A n)) ∎ use-as-def

Vec : ∀ A n → Type
Vec A n = defined-by (Vec-def A n)
```

SIP doesn't mesh very well with indexed stuff, does HSIP help?

Of course, a container would not be of much use without lookup functions, so we define an interface

```
record Array (V : Type → ℕ → Type) : Type₁ where
  field
    lookup : ∀ {A n} → V A n → Fin n → A
    tail : ∀ {A n} → V A (suc n) → V A n
```
which at the very least has to satisfy laws like
```
record ArrayLaws {C} (Arr : Array C) : Type₁ where
  field
    lookup∘tail : ∀ {A n} (xs : C A (suc n)) (i : Fin n)
                → Arr .lookup (Arr .tail xs) i ≡ Arr .lookup xs (inr i)
```
We could directly show that Vec satisfies this, but now that we defined Vec from Lookup we might as well use that.

The implementation of arrays as functions is very straightforward
```
FunArray : Array Lookup
FunArray .lookup f = f
FunArray .tail f = f ∘ inr
```
and clearly satisfies our interface
```
FunLaw : ArrayLaws FunArray
FunLaw .lookup∘tail _ _ = refl
```
We can implement arrays based on Vec as well[5]
```
VectorArray : Array Vec
VectorArray .lookup {n = n} = f n
  where
  f : ∀ {A} n → Vec A n → Fin n → A
  f (suc n) (x , xs) (inl _) = x
```

[5]Note that, like any other type computing representation, we pay the price by not being able to pattern match directly on our type.

```
    f (suc n) (x , xs) (inr i) = f n xs i
  VectorArray .tail (x , xs) = xs
```
which again allows us to transport proofs from Lookup to Vec.[6]

As you can see, taking "use-as-definition" too literally prevents Agda from solving a lot of metavariables.

This computation can of course be generalized to any arity zeroless numeral system; unfortunately beyond this set of base types, this "straightforward" computation from numeral system to container loses its efficacy. In a sense, the n-ary natural numbers are exactly the base types for which the required steps are convenient type equivalences like $(A + B) \to C = (A \to C) \times (B \to C)$*?*

3.2 Numerical representations as ornaments

We could peform the same computation for Leibniz, which would yield the type of binary trees, but we note that these computations proceed with roughly the same pattern: each constructor of the numeral system gets assigned a value, and is amended with a field holding a number of elements and subnodes using this value as a "weight". But wait! Such modifications of constructors are already made formal by the concept of ornamentation!

Ornamentation, as exposed in [McB14] and [KG16], lets us formulate what it means for two types to have a "similar" recursive structure. This is achieved by interpreting (indexed inductive) datatypes from descriptions, between which an ornament is seen as a certificate of similarity, describing which fields or indices need to be introduced or dropped. Furthermore, a one-sided ornament: an ornamental description, lets us describe new datatypes by recording the modifications to an existing description.

Again not sure how much space I should use to reiterate Desc, Orn, and OrnDesc.

This links back to the construction in the previous section, since ℕ and Vec share the same recursive structure, so Vec can be formed by introducing indices and adding a field holding an element at each node.[7]

However, instead deriving List from ℕ generalizes to Leibniz with less notational overhead, so lets tackle that case first. For this, we have to give a description of ℕ to work with

Clearly this can use more explanation (the question is, how much?)

```
  NatD : Desc ⊤
  NatD _ = σ Bool λ
    { false ⟶ ṿ []
    ; true ⟶ ṿ [ tt ] }
```
Recall that σ adds a field, upon which the rest of the description may vary, and ṿ lists the recursive fields and their indices (which can only be tt). We can now write down the ornament which adds fields to the suc constructor

[6] Except that due to the simplicity of this case, the laws are trivial for Vec as well.
[7] These and similar examples are also documented in [KG16]

```
NatD-ListO : Type → OrnDesc ⊤ ! NatD
NatD-ListO A (ok _) = σ Bool λ
  { false → ṿ _
  ; true → Δ A (λ _ → ṿ (ok _ , _)) }
```
Here, the σ and ṿ are forced to match those of NatD, but the Δ adds a new field. With the least fixpoint and description extraction from [KG16], this is sufficient to define List. Note that we cannot hope to give an unindexed ornament from Leibniz

```
LeibnizD : Desc ⊤
LeibnizD _ = σ (Fin 3) λ
  { zero           → ṿ []
  ; (suc zero)     → ṿ [ tt ]
  ; (suc (suc zero)) → ṿ [ tt ] }
```
into trees, since trees have a very different recursive structure! Instead, we must keep track at what level we are in the tree so that we can ask for adequately many elements:

```
power : ℕ → (A → A) → A → A
power ℕ.zero f = λ x → x
power (ℕ.suc n) f = f ∘ power n f

Two : Type → Type
Two X = X × X

LeibnizD-TreeO : Type → OrnDesc ℕ ! LeibnizD
LeibnizD-TreeO A (ok n) = σ (Fin 3) λ
  { zero           → ṿ _
  ; (suc zero)     → Δ (power n Two A) λ _ → ṿ (ok (suc n) , _)
  ; (suc (suc zero)) → Δ (power (suc n) Two A) λ _ → ṿ (ok (suc n) , _) }
```
We use the power combinator to ensure that the digit at position n, which has weight 2^n in the interpretation of a binary number, also holds its value times 2^n elements. This makes sure that the number of elements in the tree shaped after a given binary number also is the value of that binary number.

This "folding in" technique using the indices to keep track of structure seems to apply more generally. Let us explore this a bit further, and return later to the generalization of the pattern from numeral systems to datastructures.

3.3 Folding in

Let us describe this procedure of folding a complex recursive structure into a simpler structure more generally. In particular, we will demonstrate that for linear datatypes, such as ℕ and Leibniz, and for a given unindexed datatype, there is always an indexed datatype isomorphic to it at some index, and an ornament from the linear type to the indexed type.

Suppose we are given a description, the first thing we can do to simplify it is collect all fields in one place

```
RField : RDesc ⊤ → Type
RField (v̦ is) = ⊤
RField (σ S D) = Σ S λ s → RField (D s)
```
Next, we will certainly have to count the number of recursive occurrences we are tracking, so we define
```
-- note to self, I should probably make v̦ _not_ overlap
-- so not everything links here
data Number : Type where
  𝟙 : Number
  v̦ : ∀ n → (Fin n → Number) → Number
```
where 𝟙 records that we are at the top level, and v̦ denotes that we are below a constructor with some number of recursive fields. This simplifies our task to implementing the types in
```
nested : Desc ⊤ → Desc Number
nested d n = σ (Fields (d tt) n) λ a → v̦ [ subnodes a ]
```
such a way that we get an isomorphism
```
nested-eq : ∀ D → μ D tt ≃ μ (nested D) 𝟙
```
Thus, Fields is forced to have a leaf constructor like
```
data Fields (d : RDesc ⊤) : Number → Type where
  leaf : RField d → Fields d 𝟙
  node : ∀ n {f : Fin n → Number}
         → ((i : Fin n) → Fields d (f i)) → Fields d (v̦ n f)
```
if nested is to work at 𝟙. The node constructor makes sure that if we have collection of Fields, then we can gather them in a field at a higher level. We can then count the subnodes of a given Fields as
```
subnodes : ∀ {n} {d : RDesc ⊤} → Fields d n → Number
subnodes (leaf x) = v̦ (RSize _ x) λ _ → 𝟙
subnodes (node n f) = v̦ n (subnodes ∘ f)
```
where RSize counts the number of recursive fields of a particular branch
```
RSize : (d : RDesc ⊤) → (a : RField d) → ℕ
RSize (v̦ is) a = length is
RSize (σ S D) a = RSize (D (fst a)) (snd a)
```
Note that subnodes effectively keeps the shape of the previous field, but unfolds the recursive fields at the bottom of the tree by one level.

I then tried and realized how unpleasant even the functions from the original type to the nested type are to write.

As a trivialty, we get that any type, interpreted as a container, always decomposes as an ornament over a "numerical" base type. This links to the construction of binary heaps in [KG16], as in hindsight, starting from the usual binary heaps would yield binary numbers and their binary heap ornament (in a much less useful package).

Or at least, that was where I was trying to go with this, but I notice that this still is a bit further away.

4 Reducing friction (work in progress)

The setup some approaches in earlier sections require makes them tedious or impractical to apply. In this section we will look at some ways how part of this problem could be alleviated through generics, or by alternative descriptions of concepts like equivalences through the lens of initial algebras.

In later sections we will construct many more equivalences between more complicated types than before, so we will dive right into the latter. Reflecting upon Section 2, we see that when one establishes an equivalence, most of the time is spent working out a series of tedious lemmas to show that the conversion functions are mutual inverses, which tend to be relatively easy to define. We take away two things from this; the first is that the conversion functions are perhaps too obvious, and the second is that we should really avoid talking about sections and retractions lest we incur tedium![8] In fact, the machinery from [KG16] will come in handy to demonstrate to what extent our hand was actually forced in Section 2.

We note that like how ℕ is the fixpoint of NatD, one of the sides of the equivalence is almost always going to be a datatype, and hence the μ of a Desc′, so we will use this as the working assumption for the following section.

4.1 Well-founded monic algebras are initial

Unfortunately, the machinery from [KG16] relies on axiom K for a small but crucial part. To be precise, in a cubical setting, the type μ as given stops being initial for its base functor! In this section, we will be working with a simplified and repaired version. Namely, we simplify Desc′ to

```
  data Desc′ : Set₁ where
    ṿ : (n : ℕ) → Desc′
    σ : (S : Set) (D : S → Desc′) → Desc′
```

To complete the definition of μ

```
  data μ (D : Desc′) : Set₁ where
    con : Base (μ D) D → μ D
```

we will need to implement Base. We remark that in [KG16], the recursion of mapFold is a structural descent in ⟦ D' ⟧ (μ D). Because ⟦_⟧ is a function and not a datatype, this descent becomes invalid, and mapFold fails the termination check. We resolve this by defining Base as a datatype

```
  data Base (X : Set₁) : Desc′ → Set₁ where
    in-ṿ : ∀ {n} → Vec X n → Base X (ṿ n)
    in-σ : ∀ {S D} → Σ[ s ∈ S ] (Base X (D s)) → Base X (σ S D)
```

such that this descent is allowed by the termination checker without axiom K.[9]

Recall that the Base functors of descriptions are special polynomial functors, and the fixpoint of a base functor is its initial algebra. The situation so far is

[8]The latter perhaps less so, because it is useful to show a map to be monic.

[9]This has, again by the absence of axiom K, the consequence of pushing the universe levels up by one. However, this is not too troublesome, as equivalences can go between two levels, and indeed types are equivalent to their lifts.

summarized by the diagram

$$\begin{tikzcd} & F\mu_F \arrow[d, "\mathsf{con}"] \\ X \arrow[r, dashed, leftrightarrow, "e"'] & \mu F \end{tikzcd}$$

so, we are looking for sufficient conditions on X to get the equivalence $e : X \cong \mu F$. Note that when $X \cong \mu F$, then there necessarily is an initial algebra $FX \to X$. Conversely, if the algebra (X, f) is isomorphic to $(\mu F, \mathsf{con})$, then $X \cong \mu F$ would follow immediately, so it is equivalent to ask for the algebras to be isomorphic instead.

4.1.1 Algebras of endofunctors

To characterize when such algebras are isomorphic, we reiterate some basic category theory, simultaneously rephrasing it in Agda terms.[10]

Maybe category theory reference

Let C be a category, and let a, b, c be objects of C, so that in particular we have identity arrows $1_a : a \to a$ and for arrows $g : b \to c, f : a \to b$ composite arrows $gf : a \to c$ subject to associativity. In our case, C is the category of types, with ordinary functions as arrows.

Recall that an endofunctor, which is simply a functor F from C to itself, assigns objects to objects and sends arrows to arrows

```
    F₀ : Type ℓ ⟶ Type ℓ
    fmap : (A ⟶ B) ⟶ F₀ A ⟶ F₀ B
```

These assignments are subject to the identity and composition laws

```
    f-id    : (x : F A)
            ⟶ mapF id x ≡ x

    f-comp : (g : B ⟶ C) (f : A ⟶ B) (x : F A)
            ⟶ mapF (g ∘ f) x ≡ mapF g (mapF f x)
```

An F-algebra is just a pair of an object a and an arrow $Fa \to a$

```
    record Algebra (F : Type ℓ ⟶ Type ℓ) : Type (ℓ-suc ℓ) where
      field
        Carrier : Type ℓ
        forget : F Carrier ⟶ Carrier
```

Algebras themselves again form a category C^F. The arrows of C^F are the arrows f of C such that the following square commutes

$$\begin{tikzcd} Fa \arrow[r, "Ff"] \arrow[d, "U_a"'] & Fb \arrow[d, "U_b"] \\ a \arrow[r, "f"'] & b \end{tikzcd}$$

[10] We are not reusing a pre-existing category theory library for the simple reasons that it is not that much work to write out the machinery explicitly, and that such libraries tend to phrase initial objects in the correct way, which is too restrictive for us.

So we define

```
    Alg→-Sqr F A B f = f ∘ A .forget ≡ B .forget ∘ F .fmap f
```

and

```
    record Alg→ (RawF : RawFunctor ℓ)
                (AlgA AlgB : Algebra (RawF .F₀)) : Type ℓ where
      constructor alg→

      field
        mor : AlgA .Carrier → AlgB .Carrier
        coh : ∥ Alg→-Sqr RawF AlgA AlgB mor ∥₁
```

Note that we take the propositional truncation of the square, such that algebra maps with the same underlying morphism become propositionally equal

```
    Alg→-Path : {F : RawFunctor ℓ} {A B : Algebra (F .F₀)}
              → (g f : Alg→ F A B) → g .mor ≡ f .mor → g ≡ f
```

The identity and composition in C^F arise directly from those of the underlying arrows in C.

Recall that an object $\emptyset$ is initial when for each other object a, there is an unique arrow $! : \emptyset \to a$. By reversing the proofs of initiality of μ and the main result of this section, we obtain a slight variation upon the usual definition. Namely, unicity is often expressed as contractability of a type

```
    isContr A = Σ[ x ∈ A ] (∀ y → x ≡ y)
```

Instead, we again use a truncation

```
    weakContr A = Σ[ x ∈ A ] (∀ y → ∥ x ≡ y ∥₁)
```

but note that this also, crucially, slightly stronger than connectedness. We I think define initiality for arbitrary relations

```
    record Initial (C : Type ℓ) (R : C → C → Type ℓ′)
                   (Z : C) : Type (ℓ-max (ℓ-suc ℓ) ℓ′) where
      field
        initiality : ∀ X → weakContr (R Z X)
```

such that it closely resembles the definition of least element. Then, A is an initial algebra when

```
    InitAlg RawF A = Initial (Algebra (RawF .F₀)) (Alg→ RawF) A
```

By basic category theory (using the usual definition of initial objects), two initial objects a and b are always isomorphic; namely, initiality guarantees that there are arrows $f : a \to b$ and $g : b \to a$, which by initiality must compose to the identities again.

Similarly, we get that

```
    InitAlg-≃ : (F : Functor ℓ) (A B : Algebra (F .RawF .F₀))
              → InitAlg (F .RawF) A → InitAlg (F .RawF) B
              → A .Carrier ≃ B .Carrier
```

However, we only have the equalities from the isomorphism inside a propositional truncation. But fortunately, being an equivalence is a property, so we can eliminate from the truncations to get the wanted result.

Note that even though we warned ourselves, we are still talking about sections and retractions to establish that f is an equivalence! However, this result also makes sure we will not have to speak of them again.[11]

4.1.2 Accessibility

In terms of "when is $X \cong \mu D$", this reduces the problem to showing that X is an initial algebra for the base functor of D; μD is initial by its fold, and by induction on μD using the squares of algebra maps.

Remark 4.1. We need (in general) not hope μD is a strict initial object in the category of algebras. For a strict initial object, having a map $a \to \emptyset$ implies $a \cong \emptyset$. This is not the case here. In general strict initial objects satisfy $a \times \emptyset \cong \emptyset$. Consider the algebras $(\mathbb{N}, (0, \mathrm{suc}))$ and $(2^{\mathbb{N}}, f)$, for which clearly not $2^{\mathbb{N}} \times \mathbb{N} \cong \mathbb{N}$. (In particular, F being a left adjoint is sufficient to make C^F have strict initial objects, but then the carrier of the initial algebra is simply $\bot$.)

I think.

Looking back at Section 2, we see that Leibniz is an initial $F : X \mapsto 1 + X$ algebra because for any other algebra, the image of 0b is fixed, and by bsuc all other values are determined by chasing around the square. Thus, we are looking for a similar structure on $f : FX \to X$ that supports recursion.

Clearly we will need something stronger than $FX \cong X$, as in general a functor can have many fixpoints. For this, we define what it means for an element x to be accessible by f. This definition uses a mutually recursive datatype as follows: We state that an element x of X is accessible when there is an accessible y in its fiber over f

```
data Acc D f x where
  acc : (y : fiber f x) ⟶ Acc' D f D (fst y) ⟶ Acc D f x
```

Accessibility of an element x of Base A E is defined by cases on E; if E is ṿ n and x is a Vec A n, then x is accessible if all its elements are; if x is σ S E', then x is accessible if snd x is

```
data Acc' D f where
  acc-ṿ : All (Acc D f) x ⟶ Acc' D f (ṿ n) (in-ṿ x)
  acc-σ : Acc' D f (E s) x ⟶ Acc' D f (σ S E) (in-σ (s , x))
```

Consequently, X is well-founded for an algebra when all its elements are accessible

```
Wf D f = ∀ x ⟶ Acc D f x
```

We can see well-foundedness as an upper bound on the size of X, if it were larger than μD, some of its elements would inevitably get out of reach of an algebra. *Now* having $FX \cong X$ also gives us a lower bound, but remark that having a well-founded injection $f : FX \to X$ is already sufficient, as accessibility makes this an epi and an iso. Thus, we claim

I think, but I'm not sure how much surjective=epi, injective=mono and "(Type a) is balanced" I can safely sweep under the carpet.

Claim 4.1. If there is a mono $f : FX \to X$ and X is well-founded for f, then X is an initial F-algebra.

[11] In this section...

4.1.3 Proof sketch of Claim 4.1

Let us be on our way. Suppose X is well-founded for the mono $f : FX \to X$. To show that (X, f) is initial, let us take another algebra (Y, g), and show that there is a unique arrow $(X, f) \to (Y, g)$.

By Acc-recursion and because all x are accessible, we can define a plain map into Y

```
Wf-rec : (D : Desc′) (X : Algebra (Ḟ D)) → Wf D (X .forget)
          → (Ḟ D A → A) → X .Carrier → A
```

This construction is an instance of the concept of "well-founded recursion"[12], so we let ourselves be inspired by these methods. In particular, we prove an irrelevance lemma

```
Wf-rec-irrelevant : ∀ x′ y′ x a b → rec x′ x a ≡ rec y′ x b
```

which implies the unfolding lemma

```
unfold-Wf-rec : ∀ x′ → rec (cx x′) (cx x′) (wf (cx x′))
                       ≡ f (Base-map (λ y → rec y y (wf y)) x′)
```

The unfolding lemma ensures that the map we defined by Wf-rec is a map of algebras. The proof that this map is unique proceeds analogously to that in the proof that μD is initial, but here we instead use Acc-recursion

```
Wf+inj⟶Init : (D : Desc′) (X : Algebra (Ḟ D)) → Wf D (X .forget)
               → injective (X .forget) → InitAlg (RawḞ D) X
```

Thus, we conclude that X is initial. The main result is then a corollary of initiality of X and the isomorphism of initial objects

```
Wf+inj≡μ : (D : Desc′) (X : Algebra (Ḟ D)) → Wf D (X .forget)
            → injective (X .forget) → X .Carrier ≡ μ D
```

4.1.4 Example

Let us redo the proof in Section 2, now using this result.

5 Is equivalence too strong (finger trees)

6 Discussion and future work (aka the union of my to-do list and the actual future work section)

7 Temporary

Todo list

[12]This is formalized in the https://agda.github.io/agda-stdlib/Induction.WellFounded.html with many other examples.

References

[Ang+20] Carlo Angiuli et al. *Internalizing Representation Independence with Univalence*. 2020. DOI: 10.48550/ARXIV.2009.05547. URL: https://arxiv.org/abs/2009.05547.

[DM14] PIERRE-ÉVARISTE DAGAND and CONOR McBRIDE. "Transporting functions across ornaments". In: *Journal of Functional Programming* 24.2-3 (Apr. 2014), pp. 316–383. DOI: 10.1017/s0956796814000069. URL: https://doi.org/10.1017%2Fs0956796814000069.

[EC22] Lucas Escot and Jesper Cockx. "Practical Generic Programming over a Universe of Native Datatypes". In: *Proc. ACM Program. Lang.* 6.ICFP (Aug. 2022). DOI: 10.1145/3547644. URL: https://doi.org/10.1145/3547644.

[HS22] Ralf Hinze and Wouter Swierstra. "Calculating Datastructures". In: *Mathematics of Program Construction*. Ed. by Ekaterina Komendantskaya. Cham: Springer International Publishing, 2022, pp. 62–101. ISBN: 978-3-031-16912-0.

[KG16] HSIANG-SHANG KO and JEREMY GIBBONS. "Programming with ornaments". In: *Journal of Functional Programming* 27 (2016), e2. DOI: 10.1017/S0956796816000307.

[McB14] Conor McBride. "Ornamental Algebras, Algebraic Ornaments". In: 2014.

[Oka98] Chris Okasaki. *Purely Functional Data Structures*. USA: Cambridge University Press, 1998. ISBN: 0521631246.

[Tea23] Agda Development Team. *Agda*. 2023. URL: https://agda.readthedocs.io/en/v2.6.3/.

[VMA19] Andrea Vezzosi, Anders Mörtberg, and Andreas Abel. “Cubical Agda: A Dependently Typed Programming Language with Univalence and Higher Inductive Types”. In: *Proc. ACM Program. Lang.* 3.ICFP (July 2019). DOI: `10.1145/3341691`. URL: `https://doi.org/10.1145/3341691`.