Generic Proofs and Constructions in Agda

Samuel Klumpers
6057314

April 6, 2023

Contents

Abstract

This thesis aims to introduce the concepts of the structure identity principle, numerical representations, and ornamentations. These concepts are then combined to simplify the presentation and verification of finger trees, demonstrating the generalizability and improved compactness and security of the resulting code.

Part I
Outline

1 Problem Statement

2 Contributions

Part II
Related work

Part III
Preliminary work

3 The Cost of Verification

The dependently typed functional programming language Agda [Tea23] can, when restricted to its reasonable parts, be translated into readable, and safe, Haskell [Coc+22]. However, the intrinsic safety of languages like Agda can also lead to code duplication by encouraging the use of multiple variants of the same datatype: enforcing coverage checking forces the head function on List to return a Maybe. The Maybe can be avoided by moving to the length-indexed list type Vec, at the cost of duplicating functions like _++_, which we want to have for both types.

Of course, not differently from Haskell, a similar problem arises when implementing binary trees as a more efficient alternative to lists. Furthermore, the proofs of the same properties will differ between list and tree, and tend to be more difficult for the latter. In general, switching between implementations of an interface will not only duplicate code, but also (and sometimes more than) duplicate the effort of verification.

On the other hand, the expressive power of dependent types is also such that part of the problems arising from safety or efficiency can be dealt with inside

Agda itself; the work in [DM14] and [KG16], provides the means to relate similar datatypes, such as lists and vectors, using the mechanism of ornamentation. In fact, the algebraic nature of ornaments yields the definition of the vector type for free, provided we relate lists to natural numbers [McB14].

From another point of view, lists and trees are not so different at all, provided we look at them through the interface of one-sided flexible arrays; this idea noted in [Oka98] and formalized in [HS22] where both are shown to be instances of numerical representations by calculating them from a numeral system.

When two types are isomorphic and equivalent under an interface, proofs of properties of these implementations become interconvertible. While this is achievable through meta-programming, substituting conversions to and from into the proof terms, this is internally expressible in Cubical Agda. By using structured equivalences and univalence, [Ang+20] gives a way to derive what is necessary to show that two implementations are equal.

In Section 4, we will follow [Ang+20], and look at how proofs on unary naturals can be transported to the binary naturals. Then in Section 5 we recall how numeral systems in particular induce container types in [HS22], which we attempt to reformulate in the language of ornaments in Subsection 5.2, using the framework of [KG16]. In Section 6 we investigate how we can make the earlier methods more easily accessible to the user, and, ourselves, when we give a description of finger trees in Section 7.

4 Proving about binary using unary: the Structure Indentity Principle

Let us quickly review the small set of features in Cubical Agda that we will be using extensively throughout this article.[1]

In Cubical Agda, the primitive notion of 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.

While the benefits are overwhelming for us, this is not completely without downsides, such as that the negation of axiom K 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.

On the positive side, this different perspective gives intuitive interpretations to some proofs of equality, like

[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 6.

```
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
```

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 in the next section.

4.1 Binary numbers

Let us demonstrate an application of univalence by exploiting 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. However, the number of cases for a proof about binary numbers also grows quicker than it would for unary numbers. 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. 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 can be promoted to 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.

4.2 Use as definition: functions from specifications

As an example, we will define addition of binary numbers. We could transport binary operations

```
BinOp : Type → Type
BinOp A = A → A → A
```

to get

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

but this would be rather inefficient, incurring an $O(n+m)$ overhead when adding $n+m$. It is more efficient to define addition on Leibniz directly, making 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

```
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) (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
  -- similar clauses omitted
```

Now we can easily extract the definition of plus and its correctness with respect to _+_

```
  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))
```

We remark Def is close in concept to refinement types[3], but importantly, the equality proof is relevant for us, and the value is inferred rather than given. [4]

4.3 Structure Identity Principle

Now ℕ with N.+ form, in particular, a magma. The same goes for Leibniz and plus, but notice that a path in a Σ type is just a Σ of paths! This means that we get a path from (ℕ, N.+) to (Leibniz, plus). More generally, a magma is simply a type X with some structure, which is a function $f : X \to X \to X$ in the case of a magma. We can see that paths between magmas correspond to paths p between the underlying types X and paths over p 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

```
  MagmaStr : Type → Type
  MagmaStr = BinOp
```

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 the case of MagmaStr, 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
```

[3]À la Data.Refinement.

[4]Unfortunately, normalizing an application of a defined-by function also causes a lot of unnecessary wrapping and unwrapping, so Def is mostly only useful for presentation.

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
```

5 Types from Specification: Ornamentation and Calculation

While the practical applications of the last example do not stretch very far[5], we can generalize the idea to many other types and structures. In the same vein, 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.

But rather than inductively defining an array-like type and then showing that it is represented by a lookup function, we can go the other way around and define types by insisting that they are 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 are our starting point for defining more complex datastructures based on simpler ones, so let us demonstrate such a calculation.

5.1 Numerical representations: from numbers to containers

We can compute the type of vectors starting from ℕ.[6] 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

```
Fin-def : ∀ n → Def (Σ[ m ∈ ℕ ] m < n)
Fin-def zero =
  (Σ[ m ∈ ℕ ] m < 0)      ≡⟨ ⊥-strict (λ ()) ⟩
  ⊥                        ■ use-as-def
```

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

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

```
    Fin-def (suc n) =
      (Σ[ m ∈ ℕ ] m < suc n)  ≡⟨ ua (<-split n) ⟩
      ⊤ ⊎ (Σ[ m ∈ ℕ ] m < n) ≡⟨ cong (⊤ ⊎_) (by-definition (Fin-def n)) ⟩
      ⊤ ⊎ defined-by (Fin-def n) ∎ use-as-def

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

using

```
    <-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

```
    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?

We can implement the following interface using Vec

```
    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
```

and show that this satisfies some usual 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)
```

However, 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[7]

```
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
  f (suc n) (x , xs) (inr i) = f n xs i
VectorArray .tail (x , xs) = xs
```

but now the equality allows us to transport proofs from Lookup to Vec.[8]

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 $(A + B) \to C = (A \to C) \times (B \to C)$*?*

5.2 Numerical representations as ornaments

We could perform 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". This kind of "modifying constructors" is formalized by ornamentation as exposed in [McB14] and [KG16], which 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 to go from one description to the other. Furthermore, one-sided ornaments (ornamental descriptions) lets us describe new datatypes by recording the modifications to an existing description.

Put some minimal definitions here.

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.[9]

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

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

[8]Except that due to the simplicity of this case, the laws are trivial for Vec as well.

[9]These and similar examples are also documented in [KG16]

```
NatD : Desc ⊤ ℓ-zero
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
```
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 ⊤ ℓ-zero
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.

5.3 Heterogeneization

Here is a small pattern that occurs from time to time when one separates data based on what it "can do" rather than what it "actually is". For example, if you are making a game in Haskell, you might feel the need to maintain a list of "Drawables", which may be of different types.

Regardless of the means, the end result is going to be a "heterogeneous list". In Haskell, this is resolved by using an existentially quantified list, which, informally speaking, can contain any type implementing some constraint, but can only be inspected as if it contains the intersection of all types implementing this constraint.

This of course ports fairly directly to Agda, but is cumbersome when we just want to make a pile of different types, and becomes impractical if we want to be able to inspect the elements. The alternative is to split our heterogeneous list into two parts; one tracking the types, and one tracking the values. In practice, this means that we implement a heterogeneous list as a list of values indexed over a list of types.

We will demonstrate that we can express this operation generically as an ornament. For this, we make a small adjustment to RDesc to track a type parameter separately from the fields. Using this we define an ornament-computing function, which given a description computes an ornamental description on top of it:

```
HetO′ : (D : RDesc ⊤ ℓ-zero) (E : RDesc ⊤ ℓ-zero) (x : Ḟ (λ _ → D) (μ (λ _ → E) Type) Type tt) → ROrnDes
HetO′ (ṿ is) E x = ṿ (map-ṿ is x)
  where
  map-ṿ : (is : List ⊤) → Ṗ is (μ (λ _ → E) Type) → Ṗ is (Inv !)
  map-ṿ [] _ = _
  map-ṿ (_ ∷ is) (x , xs) = ok x , map-ṿ is xs
HetO′ (σ S D) E (s , x) = ∇ s (HetO′ (D s) E x)
HetO′ (ṗ D) E (A , x) = Δ[ _ ∈ A ] ṗ (HetO′ D E x)

HetO : (D : RDesc ⊤ ℓ-zero) → OrnDesc (μ (λ _ → D) Type tt) ! λ _ → D
HetO D (ok (con x)) = HetO′ D D x
```

This ornament relates the original unindexed type to a type indexed over it; we see that this ornament largely keeps all fields and structure indentical, only performing the necessary bookkeeping in the index, and adding extra fields before parameters.

As an example, we adapt the list description

```
ListD : Desc ⊤ ℓ-zero
ListD _ = σ Bool λ
  { false → ṿ []
  ; true → ṗ (ṿ [ tt ]) }

List′ : Type ℓ → Type ℓ
List′ A = μ ListD A tt
```

which is easily heterogeneized to an HList. In fact, HetO seems to act functorially; if we lift Maybe like

```
MaybeD : Desc ⊤ ℓ-zero
MaybeD _ = σ Bool (λ
  { false → ṿ []
```

```
    ; true → ṗ (v̩ []) })

  Maybe : Type ℓ → Type ℓ
  Maybe A = μ MaybeD A tt

  HMaybeD = ⌊ HetO (MaybeD tt) ⌋
  HMaybe = μ HMaybeD ⊤
```
then we can lift functions like head as
```
  head : List′ A → Maybe A
  head (con (false , _)) = con (false , _)
  head (con (true , a , _)) = con (true , a , _)

  hhead : (As : List′ Type) → HList As → HMaybe (head As)
  hhead (con (false , _)) (con _) = con _
  hhead (con (true , A , _)) (con (a , _)) = con (a , _ , _)
```

6 Getting more from less: Generic Constructions

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 4, 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![10] We will reuse the machinery from [KG16] to illustrate how the definitions in Section 4 were actually forced for a large part.

First, we remark that μ is internalization of the representation of simple[11] datatypes as W-types. Thus, we will assume that one of the sides of the equivalence is always represented as an initial algebra of a polynomial functor, and hence the μ of a Desc′.

6.1 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

[10]The latter perhaps less so, because it is useful to show a map to be monic.
[11]Of course, indexed datatypes are indexed W-types, mutually recursive datatypes are represented yet differently...

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
      v̧ : (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 type computing function and not a datatype, this descent becomes invalid[12], and mapFold fails the termination check. We resolve this by defining Base as a datatype

```
    data Base (X : Set₁) : Desc′ → Set₁ where
      in-v̧ : ∀ {n} → Vec X n → Base X (v̧ 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.[13]

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 summarized by the diagram

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

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 $(\mu F, \text{con})$, then $X \cong \mu F$ would follow immediately, so it is equivalent to ask for the algebras to be isomorphic instead.

6.1.1 Datatypes as initial algebras

To characterize when such algebras are isomorphic, we reiterate some basic category theory, simultaneously rephrasing it in Agda terms.[14]

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

[12]Refer to the without K page.

[13]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.

[14]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.

```
    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

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

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 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.[15]

6.1.2 Accessibility

As a consequence, we get that X is isomorphic to μD when 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 6.1. 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: strict initial objects satisfy $a \times \emptyset \cong \emptyset$, but for the $X \mapsto 1 + X$-algebras $\mathbb{N}$ and $2^{\mathbb{N}}$ clearly $2^{\mathbb{N}} \times \mathbb{N} \cong \mathbb{N}$ does not hold. On the other hand, the "obvious" sufficient condition to let C^F have strict initial objects is that F is a left adjoint, but then the carrier of the initial algebra is simply $\bot$.

Looking back at Section 4, 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

[15] For now...

```
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 gives a section of f, making it an iso. This leads us to claim

Claim 6.1. If there is a mono $f : FX \to X$ and X is well-founded for f, then X is an initial F-algebra.

6.1.3 Proof sketch of Claim 6.1

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 $(X, f) \to (Y, g)$.

This section is about as digestable as a brick.

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"[16], 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
```

[16]This is formalized in the standard-library with many other examples.

6.1.4 Example

Let us redo the proof in Section 4, now using this result. Recall the description of naturals NatD. To show that Leibniz is isomorphic to Nat, we will need a NatD-algebra and a proof of its well-foundedness. We define the algebra

```
bsuc' : Base Leibniz₁ NatD ⟶ Leibniz₁
bsuc' zero     = 0b₁
bsuc' (succ n) = bsuc₁ n

L-Alg : Algebra (Ḟ NatD)
L-Alg .Carrier = Leibniz₁
L-Alg .forget = bsuc'
```

For well-foundedness, we use something similar to views [mcbride] (it differs because it views right through the entire structure, instead of a single layer)

```
data Peano-View : Leibniz₁ ⟶ Type₁ where
  as-zero : Peano-View 0b₁
  as-suc : (n : Leibniz₁) (v : Peano-View n) ⟶ Peano-View (bsuc₁ n)

view-1b : ∀ {n} ⟶ Peano-View n ⟶ Peano-View (n 1b₁)
view-2b : ∀ {n} ⟶ Peano-View n ⟶ Peano-View (n 2b₁)
view : (n : Leibniz₁) ⟶ Peano-View n
```

where the mutually recursive proof of view is "almost trivial". Well-foundedness follows fairly immediately

```
view⟶Acc : ∀ {n} ⟶ Peano-View n ⟶ Acc NatD bsuc' n
Wf-bsuc : Wf NatD bsuc'
Wf-bsuc n = view⟶Acc (view n)
```

Injectivity of bsuc_1 happens to be harder to prove from retractions than directly, so we prove it directly, from which the wanted statement follows

```
L≃μN : Leibniz₁ ≃ μ NatD
L≃μN = Wf+inj≃μ NatD L-Alg Wf-bsuc λ x y p ⟶ inj-bsuc x y p
```

Note that in this case it took us more code to prove the same statement! However, we stress that the code that we did write became more forced, and might be more amenable to automation.

7 Case Study: FingerTrees

Fingertrees are often (rightfully so) referred to as "the fastest persistent datastructure for most purposes", but while simpler than implementations achieving the same bounds, they are still challenging to reason about; in this section, we will investigate how we can fit the description and analysis of fingertrees, or variants upon them, into the frameworks of calculating datastructures and ornamental programming.

We compare the work in calculating datastructures to solving associativity equations in groups by shifting to the Cayley representation, such as in [..]

Part IV

Proposal

8 Planning

9 Temporary

Todo list

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`.

[Coc+22] Jesper Cockx et al. "Reasonable Agda is Correct Haskell: Writing Verified Haskell Using Agda2hs". In: *Proceedings of the 15th ACM SIGPLAN International Haskell Symposium*. Haskell 2022. Ljubljana, Slovenia: Association for Computing Machinery, 2022, pp. 108–122. ISBN: 9781450394383. DOI: `10.1145/3546189.3549920`. URL: `https://doi.org/10.1145/3546189.3549920`.

[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`.

[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`.